Mobile CSP||B

by

Beeta Vajar

Ph.D. Thesis

Department of Computing
Faculty of Engineering & Physical Sciences
University of Surrey

December 2009
**Declaration**

This thesis and the work to which it refers are the results of my own efforts. Any ideas, data, images or text resulting from the work of others (whether published or unpublished) are fully identified as such within the work and attributed to their originator in the text, bibliography or in footnotes. This thesis has not been submitted in whole or in part for any other academic degree or professional qualification. I agree that the University has the right to submit my work to the plagiarism detection service TurnitinUK for originality checks. Whether or not drafts have been so-assessed, the University reserves the right to require an electronic version of the final document (as submitted) for assessment as above.
Contents

1 Introduction
   1.1 Motivation ........................................... 5
   1.2 Technical Background ................................ 7
      1.2.1 The B-Method .................................. 7
      1.2.2 CSP (Communicating Sequential Processes) ....... 10
      1.2.3 Combining CSP and the B-Method ................. 15
      1.2.4 CSP || B ...................................... 15
   1.3 Objectives ........................................... 21
   1.4 Thesis Outline ....................................... 23

2 New results for CSP || B .................................... 25
   2.1 Deadlock freedom and hiding communication channels .. 25
   2.2 Deadlock freedom of two combined components .......... 27
   2.3 Deadlock freedom of the whole system .................. 33
   2.4 Consistency of a mixed machine and its controller ..... 35
      2.4.1 Divergence freedom ............................. 36
      2.4.2 Deadlock freedom ............................... 41
   2.5 Discussion ........................................... 42

3 Mobile CSP || B ............................................ 45
   3.1 CSP semantics of B machines .......................... 46
   3.2 Mobile CSP || B ..................................... 50
   3.3 Developing B machines ................................ 52
   3.4 Developing a CSP Controller .......................... 52
   3.5 Parallel combination .................................. 55
   3.6 Divergence freedom of a controller and its machines ... 59
      3.6.1 Conditions for divergence freedom ............... 61
      3.6.2 Verification of divergence freedom ............... 63
3.7 Divergence freedom of the whole system ... 101
3.8 Deadlock freedom ... 107
3.9 Refinement ...
  3.9.1 Refinements of CSP controllers ... 110
  3.9.2 Refinements of B machines ... 112
3.10 Discussion ... 118

4 Case Study
4.1 Flight tickets sale system ... 121
4.2 Design and specification of B machines ... 123
4.3 System design and specification in mobile CSP
  4.3.1 Data types ... 127
  4.3.2 Sets ... 128
  4.3.3 Function ... 128
  4.3.4 Channels ... 128
  4.3.5 Sell agencies ... 130
  4.3.6 The Return office ... 135
  4.3.7 Controller ... 138
4.4 Coding mobility into standard CSP ... 138
4.5 Verification of the system divergence freedom
  4.5.1 Assertions on control points ... 139
  4.5.2 Control Loop Invariants in Sell agencies ... 141
  4.5.3 Establishing Sell agencies are CLI preserver ... 142
  4.5.4 Control Loop Invariants in the Return office ... 147
  4.5.5 Establishing the Return office is CLI preserver ... 148
  4.5.6 Divergence freedom establishment ... 153
4.6 Verification of the system deadlock freedom ... 153
4.7 Discussion ... 153

5 Related works and comparison
  5.1 SPIN(Simple Promela Interpreter) ... 159
  5.2 SMV ... 161
  5.3 VeriSoft ... 161
  5.4 Mobile UNITY ... 162
  5.5 Occam-π ... 163
  5.6 PiOZ ... 164
  5.7 ZCCS ... 164
  5.8 CSP-OZ ... 165
  5.9 Object-Z/CSP ... 166
  5.10 Circus ... 167
  5.11 π | B ... 168
  5.12 CSP2B ... 169
5.13 ProB .......................................................... 170

6 Conclusion and Future Work ........................................... 173
  6.1 Conclusion ......................................................... 173
  6.2 Future Work ...................................................... 175
    6.2.1 Mobile control point channels ............................... 176
    6.2.2 Extension for Deadlock freedom verification ............... 177
    6.2.3 New machine creation ...................................... 177

Bibliography .............................................................. 179

A AMN specification of a flight machine .......................... 187
B CSP specification of Flight tickets sale system ............... 189
C wp proofs of Flight tickets sale system ....................... 197
Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my principal supervisor, Professor Steve Schneider, for his invaluable guidance and comments and his great support at every step of my research.

My warmest thanks also go to my co-supervisor, Dr Helen Trehearn, who kindly supported me in this work.

I am also very grateful to my friend Dr Zhe Xia for all his help and assistance.

My love and appreciation go to my parents for their continuous encouragement and endless support.

I am forever indebted to my brother without whom I would never be at this stage of my life. It is to him that I dedicate this thesis.
Abstract

Formal methods are mathematically based languages for producing verifiable, consistent and more reliable formal specifications which leads to the construction of trustworthy and maintainable computer programs.

Most formal methods can be classified as state-based or event-based formal methods. State-based formal methods, such as the B-Method, are capable of describing data aspects of the system but they are not able to describe behavioural aspects or concurrency. On the other hand, by using event-based formal methods, such as CSP, we are not able to describe data aspects of the system which results in difficulty to describe systems which contain state transitions. Over the years, the idea of combining state and event based formal methods has been proposed in order to design systems in which both data and behavioural aspects are described.

The idea of creating a combination of state and event based formal methods which is able to describe mobility and dynamic patterns has also been raised in formal method integration. This additional functionality is suitable for modelling agent systems or peer-to-peer networks where consideration of mobility is important.

$CSP \parallel B$ is a combination of CSP and B in which CSP processes are used as control executives for B machines. This architecture enables a B machine and its controller to interact and communicate with each other while working in parallel. The architecture has focused on sequential CSP processes as dedicated controllers for B machines.

This thesis introduces Mobile $CSP \parallel B$, a formal framework based on $CSP \parallel B$ which enables us to specify and verify concurrent systems with mobile architecture as well as the previous static architecture. In Mobile $CSP \parallel B$, a parallel combination of CSP processes act as the controller for the B machines and these B machines can be transferred between CSP processes during the system execution.
Chapter 1

Introduction

In this chapter, we present the motivations for the development of our work. In addition, the technical background of this thesis is reviewed in section 1.2. Furthermore, we discuss the objectives of our work in section 1.3. Finally, an overview of this document is presented in section 1.4.

1.1 Motivation

As our lives get more dependent on computers, the need for better quality software increases. Computers with high quality software are better able in satisfying the user requirements. There have always been problems in software development such as: long development times, higher cost than originally calculated and inability to meet all the user needs. The more complex a program is, the more errors may occur in programming. Programmers claim that most of the programming time is taken up by debugging and more often, removing an error will cause other new errors.

One significant reason that software does not work properly is that the user requirements may not have been clearly explained or understood before producing the software. This is why we need specification as the first step before writing a program. Specifying a system means that we provide a concise as well as correct description for the required system. A specification defines the system in a precise way. It describes what the program should do and how it is expected to behave. The success in developing a correct program begins with an unambiguous, consistent and complete specification. Specifications which use mathematical notations are called formal specifications. Formal methods are mathematically based languages for producing formal specifications. Their available software tools can be
then used to check the specified systems. As a result, formal methods are able to produce verifiable, consistent and more reliable specifications which leads to construct trustworthy and maintainable computer programs.

There are a variety of formal methods used for formal specification and verification of systems. Most of these formal methods can be classified as state-based or event-based formal methods. State-based formal methods, such as VDM (Vienna Development Method) [38, 10, 53, 48], Z [65, 52] and the B-Method [57, 73, 41], focus on data aspects of systems. They are concerned with describing the state of the systems and a number of operations describe state transitions in the system. Event-based formal methods are concerned with behavioural aspects of the systems. They describe the events happened in the system, without being concerned with the state transitions. Among the event-based formal methods are CCS (Calculus of Communicating Systems) [49, 28], π-calculus [55] and CSP (Communicating Sequential Processes) [56, 36] which are all process algebras (or process calculus). Process algebras are mathematical formalisms used to model concurrent systems.

Despite the advantages mentioned above, using a single kind of formal method is not capable to cover all programmers' desires. Each formal method is designed to describe some specific aspects of systems. State-based formal methods are capable to describe data aspects of the system but they are not able to describe behavioural aspects or concurrency. On the other hand, by using event-based formal methods, we are not able to describe data aspects of the system which results difficulty to describe systems which contain state transitions.

Over the years, the idea of combining formal methods has been raised in order to describe systems with more aspects. Following from this idea, combining state and event based formal methods has been created in software development. The main result of this idea is to create a reliable and consistent model for concurrent, complex and critical systems. System designers are interested to design more accurate systems and this combination enables the creation of a formal system description concerning both data and behavioural aspects. There have been some attempts in combining state and event based formal methods in a system description which go some way to realising the potential of integrating formal methods in system development. Some combined languages have already been proposed in formal methods integration. However, most of formal methods integration efforts are concentrated on combining Z or Object-Z [62] with process algebras. Some of these are ZCCS, CSP-OZ, Object-Z/CSP and Circus. There have been some attempts in combining the B-Method with process algebras such as: CSP2B, ProB and CSP ∥ B.

The idea of creating a combination of state and event based formal methods which is able to describe mobility and dynamic patterns has also been raised in formal
method integration. This additional functionality is suitable for modelling agent systems or peer-to-peer networks where consideration of mobility is important. This idea appears in two formal methods integration approaches: PiOZ and \( \pi | B \). As \( \pi \)-calculus is natural for expressing mobile structures, both approaches use \( \pi \)-calculus in their architecture. All approaches above except for CSP \( || B \) are described further in Chapter 5. CSP \( || B \) will be described in detail later in this chapter.

In addition to formal methods integrations, some other approaches have also been created for describing and modeling concurrency or both concurrency and state transitions. Some of these approaches are Input/Output Automata [45], Mazurkiewicz traces [46], event structures [51] and vector languages [60, 59, 61]. Also, RAISE [35] is a formal method created for describing both data and behavioural aspects of the systems especially concurrent systems.

This thesis introduces Mobile CSP \( || B \), a new combination between the B-Method and CSP which is used for specification and verification of mobile combined communicating systems.

1.2 Technical Background

This section is the technical background to our work. We first present a brief description of the B-Method and CSP. We then explain in detail the CSP \( || B \) framework and we review all the previous achievements in this work.

1.2.1 The B-Method

The B-Method was invented by Jean-Raymond Abrial who also invented the Z notation. It is based on the same principles as the Z notation plus guarded commands and refinement. B is capable to be applied for modelling safety critical systems. By using the existing B tools, it is possible to prove and ensure the required safety properties in the system. The usage of B has been raised in industry and academic environments. It is believed as a suitable formal method for specifying and verifying rail transport systems and has been used in the Paris Metro Line 14, Meteor project [15], the first driverless metro in the city of Paris. In this project, the B-Method has been used to develop the safety critical software of the automatic pilots.

The B-Method has also been used in smart cards applications. BSmart [23] is a tool based on the B-Method to develop Java Card [21] applications (applications run on smart cards are written in the Java programming language). In BSmart, the user is able to produce the abstract B specification and the refinement of the
card services. BSmart is then able to automatically generate the Java Card code from the B refinement.

The B-Method is a state-based formal method which produces reliable and consistent program code in 3 stages: abstract specification, refinement and implementation. In the B-Method, specifications are presented as a mathematical model called abstract machines. The method uses a state-based formal specification language called Abstract Machine Notation (AMN) as a common language in all three steps. An abstract machine describes the state of the system in terms of mathematical notations such as: sets, relations, functions and sequences. The local state information of the machine is presented by its variables. All of the machine's variables are introduced in the VARIABLES clause. All the information about the variables such as their types, their relationships with each other and any restrictions on their possible values is presented in the INVARIANT clause. It describes the properties of the variables which the machine must always maintain during the execution.

An abstract machine also consists of operations which change the state of the system. The operations of a B machine are introduced in the OPERATIONS clause. Each operation has a precondition or guard. Pre-conditioned operations have the form PRE P THEN S END, where P is the precondition and S is the body of the operation. A precondition represents a predicate on the state of the machine which should hold when the operation is executed in order to ensure that the operation behaves as expected to behave. A pre-conditioned operation is able to be executed when its precondition is not satisfied, but there will not be any guarantees about the resulting execution. Guarded operations have the form SELECT G THEN R END, where G is a guard and R is the body of the operation. A guard is a predicate on the state of the machine which must hold in order to enable the operation to be executed. A guarded operation is not able to be executed when its guard is not satisfied.

An abstract machine consists of the INITIALISATION clause which represents the initial state of the machine before its operations are executed. It contains some assignments in which all variables are assigned some value initially.

If E is an AMN statement and Q is a predicate, the notation \([E]Q\) denotes all conditions which must be true when executing E to guarantee to achieve a post condition in which Q is true. As \([E]Q\) contains all preconditions which guarantee to achieve Q when executing E, it is called the weakest precondition for E to achieve Q. The weakest precondition for E to achieve Q is also written as: \(wp(E, Q)\).

If S and T are two AMN statements, then sequential composition S; T is a statement in which S is executed first and if S terminates, then T is executed from the resulting state. In this case, the final state of S; T is the state that T
reaches after execution. If $S$ does not terminate, then $S; T$ does not terminate and $T$ will not be executed. The weakest precondition for $S; T$ to achieve the post condition $Q$ is calculated as: $wp(S; T, Q) = wp(S, wp(T, Q))$.

The machine we are working with should be internally consistent. It must be consistent in its initial state and also in all possible states it achieves after executing its operations. To ensure the initial consistency of the machine, the INITIALIZATION clause $T$ must establish the invariant $I$. In other words, $[T]I$ must be true. In addition, for any pre-conditioned or guarded operations which are called inside their precondition or guard respectively, the state reached after their execution must always establish the invariant $I$. In other words, $I \land P \Rightarrow [S]I$ and $I \land G \Rightarrow [R]I$ must always be true for any pre-conditioned or guarded operations respectively.

Refinements are the intermediate stage of a program development process from abstract specification machine to implementation. A refinement is a machine which is more concrete than the original abstract specification. Refinement is a technique that is repeated several times to refine the previous machine. In each step of refinement, we make the previous machine less abstract and more close to implementation machine. However, all properties of the abstract machine are preserved by a refinement.

The B software development process is terminated at implementation. An implementation is a machine, derived from the refinement stage, from which the program code can be generated.

The B-Method is a formal method supported by many comprehensive tools described below:

- **B-Toolkit**
  B-Toolkit [2] provides the facilities some of which are described below:
  - Editor: to enter the machine definition
  - Syntax checking
  - Type checking
  - Animation: A way to check if the machine behaves in accordance with what the user wants and expects from the system
  - Ability to generate proof of internal consistency obligations in a machine - part of these proof obligations are automatically proven and the remaining can be proved by the user
  - Creation of refinement and proof of the correctness of the refinement
  - Creation of implementation
  - Automatic code generation in C
• **ProB**
  Some of the facilities provided by ProB [7, 42, 43] are listed below:
  - Interactive or fully automatic graphical animation
  - Analysing
  - Deadlock checking

• **Atelier B**
  Atelier B [1] provides the facilities some of which are listed below:
  - Syntax analysing
  - Type checking
  - Generating proof obligations - automatically proving part of proof obligations - interactive prover allows user to prove the remaining proof obligations
  - Automatic code generation in C or Ada

**Example: Student Debtors**

In this chapter, we use a simple example to demonstrate the difference in specifying a system in B, CSP and CSP \parallel B. The Student Debtors example is about a number of students in a university who have borrowed student loan from the university during their study and they have now finished their study and are going to graduate. Each student can only graduate after the loan has been fully paid up. The system clears the students' debt and certifies them as graduates.

The B specification of the Student Debtors system is shown in Figure 1.1. The student number of all debtors is given to the system by the set `StudentNumber`. The set `graduate` contains the student number of those who have paid up their loan and have now been certified as graduates. The set `loan` contains the student number of those who have not paid up their loan yet. Operation `settle` is used to clear a student's debt. It receives the student number of a student and removes the number from the debtor's list. Operation `graduation` is used to certify a student as a graduate. It receives the student number of a student and adds the number to the graduates list. Initially no debtor has paid up his or her loan. As a result, all students are in debt at the beginning of the system execution and none has yet been certified as a graduate.

**1.2.2 CSP (Communicating Sequential Processes)**

CSP was originally developed by C. A. R. Hoare as a concurrent programming language. However, the original version has been widely developed since then
1.2. TECHNICAL BACKGROUND

MACHINE StudentDebtors

SETS StudentNumber

VARIABLES loan, graduate

INVARIANT loan ⊆ StudentNumber &
graduate ⊆ StudentNumber &
loan ∩ graduate = {}

INITIALISATION
loan := StudentNumber ||
graduate := {}

OPERATIONS

settle (number) ≡
PRE
number ∈ loan
THEN
loan := loan - {number}
END;

graduation (number) ≡
PRE
number ∈ StudentNumber - loan
THEN
graduate := graduate U {number}
END
END

Figure 1.1: B specification of Student Debtors system

and it is still a research interest subject for increasing its capability of describing concurrent systems.

CSP is a theoretical notation or language for specifying and verifying concurrent systems. Concurrent systems are known as complex systems whose programming and analysing has been always an important issue in software development. CSP provides a framework for describing and analysing interacting aspects of concurrent systems. Concurrent systems consist of interacting components known as processes. Each process works independently and may interact with its environment and other processes in the system. A process performs various events which describe its behaviour. CSP is an event-based formal language for designing and analysing a system behaviour by events happened in the system.

The simplest process in CSP is STOP, a process which does nothing. Process
SKIP is a process which does nothing but it indicates successful termination.

If $P$ is a process and $a$ is an event, then the prefixing expression $a \rightarrow P$ is a process which can perform the event $a$ and then behave like the process $P$.

Recursion can be defined in CSP specifications. To describe a process which runs forever, we can use a recursive definition. For instance, in a recursive definition $P = a \rightarrow P$, the process $P$ can perform the event $a$ repeatedly. It is also possible to define a collection of processes by mutual recursion. For instance, we can define a mutual recursive definition as described below:

$$P = a \rightarrow Q$$
$$Q = b \rightarrow P$$

In this definition, the process $P$ can perform the event $a$ and then behave like the process $Q$. Also, the process $Q$ can perform the event $b$ and then behave like the process $P$.

In CSP, two operators, external choice and internal choice can be used to provide a choice between a number of processes in the system. An external choice expression $P \sqcap Q$ is a process which is ready to behave either like $P$ or $Q$. In external choice, the process whose first event is offered first by the environment will be chosen to be executed. An internal choice expression $P \sqcap Q$ is also a process which is ready to behave either like $P$ or $Q$, but the environment has no control over the choice. It is a non-deterministic choice made by the process $P \sqcap Q$ internally.

The hiding operator $\backslash$ can be used to make some events internal to a process. If $P$ is a process and $A$ is a set of events, then $P \backslash A$ is process $P$ in which the events in $A$ become internal.

In CSP, processes are able to pass messages to each other by using channels. A channel can carry values of a special type. If $c$ is a channel which can carry values of type $T$, then $c?x \rightarrow P(x)$ is a process which can input any value $x$ of type $T$ along channel $c$ and then behave like the process $P(x)$. $c!v \rightarrow P$ is a process which can output $v$ along channel $c$ and then behave like the process $P$. If a process inputs or outputs a value $t$ along channel $c$, it corresponds to performing the event $c.t$. The set of all events associated with channel $c$ is denoted by $\{|c|\}$. In other words: $\{|c|\} = \{c.t | t \in T\}$.

A trace $tr$ of a process $P$ is a finite sequence of events which $P$ is able to perform. The set of all possible traces of process $P$ is denoted by $\text{traces}(P)$. The set $\Sigma$ is denoted as the universal set of all possible events.

If $S(tr)$ is a predicate on $tr$ and $P$ is a process, then $P$ satisfies $S(tr)$ if $S(tr)$ holds for all traces $tr$ of $P$. This is denoted by $P \text{ sat } S(tr)$. In other words:

$$P \text{ sat } S(tr) = \forall tr \in \text{traces}(P) \cdot S(tr).$$
1.2. TECHNICAL BACKGROUND

A divergence of a process $P$ is a sequence of events $tr$ after which $P$ can do an infinite sequence of internal events. The set of all possible divergences of a process $P$ is denoted by $\mathcal{D}[P]$.

A pair $(tr, X)$ is a stable failure of a process $P$ if $P$ can perform trace $tr$ and then reaches a stable state in which it refuses all events in $X$. In this case, the set $X$ is called a refusal of process $P$. The set of all possible stable failures of a process $P$ is denoted by $\mathcal{SF}[P]$.

A failure of a process $P$ is a pair $(tr, X)$ in which either $tr$ is a divergence of $P$ or $(tr, X)$ is a stable failure of $P$. As a result, if a process $P$ is divergence free then all failures of $P$ are the stable failures of $P$. The set of all possible failures of a process $P$ is denoted by $\mathcal{F}[P]$ or $\text{failures}(P)$.

For any two processes $P$ and $Q$, if $\mathcal{D}[P] = \mathcal{D}[Q]$ and $\mathcal{F}[P] = \mathcal{F}[Q]$, then $P$ and $Q$ are said to be equivalent in the failures/divergences model which is denoted by $P =_F Q$.

For any two processes $P$ and $Q$, if $\text{failures}(Q) \subseteq \text{failures}(P)$ and $\text{traces}(Q) \subseteq \text{traces}(P)$, then $Q$ is said to be a failure refinement of $P$ which is denoted by $P \preceq F Q$. It means that all behaviour $Q$ can do, $P$ is able to do.

Processes can interact with each other by working in parallel. The parallel combination of processes means that they execute concurrently and should synchronise on the common events. Processes may work in parallel with each other in two different ways: Alphabetized parallel and Interface parallel.

A set of all possible events that a process is able to perform is called the interface of that process. If the interfaces of processes $P$ and $Q$ are $A$ and $B$ respectively, then an Alphabetized parallel combination of the two processes is denoted by $P_A || B Q$. In this combination, $P$ and $Q$ must synchronise on the events in the intersection of $A$ and $B$.

An Interface parallel combination of two processes $P$ and $Q$ is denoted by $P_A \parallel Q$. In this combination, the two processes synchronise only on the events in the interface $A$.

Processes may execute concurrently in the system without synchronisation on common events. In this concurrent execution, processes do not interact with each other and each process execute independently. This combination is called Interleaving. The interleaving of two processes $P$ and $Q$ is denoted by $P || A Q$.

Parallel combinations can result in deadlock in concurrent systems. Deadlock state is a situation in which no more event is able to be executed next in the system. As processes must synchronise on the common events, a situation may occur in which each process is only able to do one or more common events but some of other processes in the system are not able to execute any of those common
events at that point. Therefore the deadlock state is the state in which the execution of the system is blocked.

CSP is supported by software tools described below:

- **ProBE (Process Behaviour Explorer)**
  ProBE [5] is an animator for CSP models. It is used to explore the behaviour of the CSP processes and their interaction with other processes in the system. While a process is executing, in each stage, ProBE displays the trace of events which have happened until that stage. It also displays the possible events which can happen next and the user can select one to perform next.

- **FDR (Failures-Divergence Refinement)**
  FDR [5] is known as a model and refinement checker. It offers automatic analysis of the CSP processes and also it permits automatic refinement checking between the processes. FDR provides the opportunity to prove safety and liveness properties in the system by checking divergence, deadlock, trace refinement, and failures refinement.

- **ProB**
  In addition to being a B software tool, ProB also accepts CSP specifications and can be used as a CSP tool. It can be used for animation and for checking the deadlock freedom of a CSP specification. However, deadlock freedom can be checked for a limited number of executions which can be chosen by the user. In addition, ProB is not able to check the divergence freedom of the CSP specifications.

CSP has been applied to various industrial applications such as an industrial verification project for the International Space Station (ISS) [16, 17]. In this project deadlock freedom and divergence freedom of a fault-tolerant data management system used by the ISS has been verified by using CSP specifications and the CSP tool FDR.

CSP has also been used to design and verify the security protocols such as the Needham-Schroeder public-key protocol [44]. The protocol contains an initiator A and a responder B which communicate with each other over an insecure network in which an attacker can intercept all messages going between A and B and even inject false messages between them. In [44], the protocol is first specified in CSP. Then, FDR is used to check the security of the protocol and to correct the protocol in order to ensure that no intruder can attack the protocol.

**Example: Student Debtors**

The CSP specification of the Student Debtors system is shown in Figure 1.2. The student number of all debtors is given to the system by the set StudentNumber.
1.2. TECHNICAL BACKGROUND

\[ \text{StudentNumber} = \{1, \ldots, 100\} \]

channel in : StudentNumber

channel settle, graduation

\[ P = \text{in?no} \rightarrow \text{settle} \rightarrow \text{graduation} \rightarrow P \]

Figure 1.2: CSP specification of Student Debtors system

After receiving a student number, the system first clears the student's debt and then certifies the student as a graduate.

1.2.3 Combining CSP and the B-Method

Using CSP and the B-Method together gives us the ability to describe and analyse a computer system in a better way because they are each good for different views of the system. As a result, it is worthwhile using a combination of CSP and B in formal development of computer systems to achieve different aspects of the system description.

Combining CSP and B specifications is an approach which has been achieved in order to use both advantages of these two formal methods in system development. By using CSP, we will be able to control the behaviour and the execution order of B machines' operations. Up to now, there have been several approaches in combining CSP and the B-Method such as: CSP2B, ProB and CSP \( \parallel B \). Both CSP2B and ProB are described in detail in Chapter 5.

1.2.4 CSP \( \parallel B \)

In [70], Treharne and Schneider introduce CSP \( \parallel B \), a parallel combination between CSP and B in which a CSP process is used as a control executive for a B machine. Each operation in B machine corresponds to a channel with the same name in the body of CSP controller. For each B machine's operation \( bb \leftarrow \text{op}(aa) \) there is a channel \( \text{op} \) in the body of the CSP controller which carries data types the same as the types of \( aa \) and \( bb \). This provides the means for CSP controller and its controlled B machine to synchronise with each other and also to communicate with each other while working in parallel. The architecture of the parallel combination of a B machine and its controller is illustrated in Figure 1.3. For each B operation, there is one channel between the CSP controller and the B machine. This architecture enables the CSP controller to interact and communicate with its controlled B machine through these channels. A B machine and its controller can send or receive values from each other through these channels. For instance, the CSP controller sends the value of \( aa \) to the B machine...
and receives the B machine's output, \( bb \), through channel \( op \). This means that in addition to control the execution order of the B operations, CSP controller and B machine can communicate with each other and they can exchange data and information through these channels while working in parallel in the system.

In general the CSP controller, \( LOOP \), is defined by a family of processes \( S(p) \) as below:

\[
LOOP \triangleq S(0) \\
S(0) \triangleq R_0 \\
\vdots \\
S(n) \triangleq R_n
\]

where each \( R_p \) is a CSP process expression in which \( S(p) \)s may subsequently be reached.

After developing a CSP control executive for a particular B machine, the next step is to ensure that this control executive is suitable for the B machine. It is a suitable control executive if we can prove that the controller is consistent with its controlled B machine.

Operations in a B machine can be pre-conditioned or guarded. If an operation is called when its precondition or guard is true, it will behave as described by the operation body. On the other hand, if an operation is called outside its precondition, it may execute but there are no guarantees about the resulting execution. This is modelled as a divergence. If an operation is called when its guard is false, it will not execute. If all the next possible operations are the
operations whose guard is false, it is modelled as a deadlock. More information
and details about divergence and deadlock of a B machine is discussed in Chapter
3.

A CSP process is a suitable execution controller for a B machine if it does not
call any B operation outside its precondition, and at each stage, the guard of at
least one of the operations enabled by the controller must be true. Thus, in order
to prove the consistency of a CSP control executive and its controlled B machine,
we need to prove that the combined system is divergence free and deadlock free.

[70] introduces a technique to prove the divergence freedom in a combined system
containing a recursive (non-terminating) CSP process as the controller and a B
machine which contains only pre-conditioned operations. In this technique, a
control loop invariant, CLI, is used in order to ensure consistency in the system.
A theorem is presented in [70] which states that if a CLI can be found with the
following two conditions, then the system is divergence free. Firstly, CLI should
be true at the beginning of the system execution after initialisation of the B
machine. Secondly, CLI must be established at every recursive call.

Continuing on from this achievement, Treharne and Schneider [58] provide a
theorem to prove deadlock freedom in the above system. Their theorem is based
on the fact that the system should have already been proved to be divergence
free. This theorem states that if the CSP controller is non-discriminating on all
operations of its controlled B machine, which means that whenever it calls an
operation of the B machine it is able to accept any output from the B machine,
and the B machine is open on its all operations, in other words it is non-blocking,
then if CSP controller is deadlock free then the whole system is deadlock free.
The definitions of non-discriminating and open are as follows in [58]:

**Definition 1.** A CSP process P is non-discriminating on a set of partial events
PE if for any failure \((tr, X) \in SF[P]\) and subset \(CV \subseteq PE\), we have that
\((\forall v \in CV \cdot \exists w \cdot v.w \in X) \Rightarrow (tr, X \cup \{ CV \}) \in SF[P]\)

**Definition 2.** A process Q is open on a set of partial events PE if given any
\((tr, X) \in SF[Q]\) and \(e \in PE\), there is some \(w\) such that \(e.w \notin X\).

In large and complex systems, we usually need abstraction in order to make the
system considerably simplified and more understanding. Abstraction can happen
in the system by hiding the control channels, which are the channels corresponding
to B operations. In [58], a theorem is provided to establish divergence freedom
in these kind of abstract systems. This theorem states that if the original com-
bined system is divergence free, then the divergence freedom of the combined
system with hidden control channels will be resulted following by the divergence
freedom of the CSP controller whose control channels are hidden.
All the previous methods for establishing consistency have been focused on a parallel combination of a single CSP controller and a single B machine which contains only pre-conditioned operations. However, the systems we are generally concerned with are a collection of parallel combinations which interact with each other. These kind of systems have the form of \( \|_i (P_i \parallel M_i) \), in which each \( P_i \) is a CSP controller for B machine \( M_i \). \cite{69} proposes an architecture for systems containing combined components. This architecture is illustrated in Figure 1.4. In this architecture, each B machine interacts only with its CSP controller. CSP controllers interact with each other and with environment on separate channels excluding the control channels. In \cite{58, 69}, Treharne and Schneider provide theorems to prove the divergence freedom and deadlock freedom of parallel combinations of controlled components in which the B machines contain only pre-conditioned operations. They prove that if each \( P_i \parallel M_i \) is divergence free, then the whole combined system, \( \|_i (P_i \parallel M_i) \), is divergence free. Also, if the whole combined system is divergence free, to establish the overall deadlock freedom, it is sufficient to check that the parallel combination of controllers, \( \|_i P_i \), is deadlock free.

![Figure 1.4: Architecture of parallel combination of controlled components](image)

In \cite{68}, Treharne and Schneider introduce a condition which is sufficient to establish deadlock freedom for a non-terminating CSP process controlling a B machine which contains only guarded operations. This condition states that for each process expression \( R_p \) of the controller LOOP if the B machine invariant and CLI hold before the body of \( R_p \) is executed, then for all traces of \( R_p \) which are before a recursive call the state reached after that trace establishes the guard of at least one of the next possible operations. A theorem is proved in \cite{68} which states that if this condition holds for the body of LOOP, then the parallel combination is deadlock free.
The condition above was for a recursive (non-terminating) CSP controller. [68] also discusses about checking deadlock freedom of the combination of a CSP controller containing STOP with a B machine containing only guarded operations. It provides a way to show that whenever the trace reached at the stage in which STOP is the next event, this blocking is acceptable and it does not mean the deadlock we consider. In other words, reaching the event STOP means that the CSP process has been successfully terminated. [68] uses a new event called block which is situated before STOP in the body of the CSP controller. This new event corresponds to a new guarded operation block in the B machine which does nothing but skip. Whenever the CSP process reaches the event block, termination is acceptable at this stage. By using this technique, deadlock is unacceptable at any point of the execution but only at the point in which event block is the next event. A condition is introduced in [68] which is sufficient to establish deadlock freedom. This condition states that for each process expression $R_p$ of the controller LOOP if the B machine invariant and CLI hold before the body of $R_p$ is executed, then for all traces of $R_p$ which are before a recursive call and do not contain block and cannot be followed by block, the state reached after that trace establishes a guard of at least one of the next possible operations. A theorem is proved in [68] which states that if this condition holds for the body of LOOP, then the parallel combination is deadlock free.

In [27], Evans, Schneider and Treharne present the usage of CSP $\parallel B$ approach in developing distributed systems. They use CSP $\parallel B$ combination for developing a file transmission protocol. The protocol controls the transfer of data files from a sender to a receiver over a medium. The medium is unreliable and it has the potential to lose the files. The sender accepts different files from the external environment and then sends the files to the receiver over the medium. The protocol has been modelled in CSP $\parallel B$ in such a way that the sender and the receiver are each a pair of a CSP controller and its controlled B machine. The medium is modelled as a CSP process. Therefore, the whole system is the parallel combination of the two combined components and the medium. After designing the protocol in CSP $\parallel B$ architecture, the system has been checked for divergence freedom and deadlock freedom. Divergence freedom verification in this system means to establish that whenever the sender is to send a file, there must always be some unsent files to send to the receiver. Deadlock freedom verification means to establish that while three parts of the system (the sender, the medium and the receiver) are working in parallel, the execution of the system will never be blocked.

CSP $\parallel B$ framework has recently been used successfully in specifying and verifying a large complex system with a mixture of distributed and embedded systems. In [22], a platoon of Cristal vehicles has been specified based on CSP $\parallel B$ architecture and then the consistency of the system has been ensured and established.
by using the theorems provided in $CSP \parallel B$ approach for consistency verification. A platoon is a set of Cristal vehicles which move following the path of the leader in a row. Each Cristal vehicle is designed as two combined components: engine machine in parallel with engine controller, and driving system machine in parallel with driving system controller. The divergence freedom and deadlock freedom of the parallel combination of these two combined components has been successfully established by using the theorems. This results the consistency of each Cristal in the system. With each Cristal, a CSP process Net is associated which receives information from the CSP part of that Cristal and then sends it to the CSP part of the following Cristal. In other words, each process Net in the system is a CSP process which does not have any machine and it communicates and synchronises with other CSP processes in the system. Therefore, the whole platoon is the parallel combination of all the Cristals and all the processes Nets. By using the theorems, the divergence freedom and deadlock freedom of the whole platoon has been successfully verified and established.

Recently, a translator tool has been introduced in [71] which automatically translates an xUML model into $CSP \parallel B$. The intention is that the generated $CSP \parallel B$ specification can then be checked and verified for divergence freedom and deadlock freedom. The tool is able to take an xUML model as an input and then to generate and output a $CSP \parallel B$ specification. It generates machine readable B machines and CSP controllers. So, the generated B and CSP specifications can then be separately checked by the B and CSP software tools for internal consistency and then we can use the theorems for establishing the divergence freedom and deadlock freedom of the whole system. At the time of writing this thesis, the specification of some B machines generated by the translator may not be completed and the user needs to complete their specification.

Example: Student Debtors

To specify the Student Debtors system in $CSP \parallel B$, the B machine should be specified as described before in Section 1.2.1. The CSP controller DEBTORS should be specified as shown in Figure 1.5. The student number of all debtors is given to the system by the set $StudentNumber$. Whenever the system receives a student number, if the received student number belongs to one of the debtors, then the system first clears the student's debt and then certifies the student as a graduate. Otherwise, the system does not do anything and it will be ready to receive another student number.

To verify the system divergence freedom, we assign the control loop invariant for $P(S)$ as below:

CLI : $S = loan$

This CLI holds at the beginning for $S = StudentNumber$ and at every recursive
1.3. Objectives

Previous CSP || B research has not covered the consistency verification of all possible kinds of systems which we can design in accordance with the CSP || B architecture. For instance, deadlock freedom verification of the whole combined system has been focused on the systems whose B machines contain only pre-conditioned operations. Having B machines containing guarded operations, we are only able to check the deadlock freedom of the parallel combination between each B machine and its controller. In addition, there has been no consideration of a system whose B machines contain both pre-conditioned and guarded operations.

This thesis aims to complete the consistency verification in CSP || B. We provide theorems and techniques to establish the consistency of parallel combinations of controlled components containing any kinds of B machine operations. This includes the systems in which each B machine can contain both pre-conditioned and guarded operations.
In CSP || B, each CSP process can be the control executive of only one B machine and each B machine has only one CSP process as its controller. The architecture has focused on sequential CSP processes as dedicated controllers for B machines.

The main objective of this thesis is to generalise CSP || B architecture in designing a new framework which enables us to describe and verify systems in which a parallel combination of CSP processes are the controller of B machines and each single B machine can be controlled by different CSP processes during the execution.

Our work begins with providing a new architecture in which a single CSP process can be the control executive of one or more B machines. We then extend our approach to design a system which contains parallel combination of several CSP controllers each controlling several B machines.

In order to let CSP processes work with different B machines during the execution, we have introduced mobility in our new architecture which means that a parallel combination of CSP processes is the control executive of the B machines and these B machines can be transferred between CSP processes during the system execution. In other words, each CSP process can receive a machine or give a machine to another CSP process during the execution. This new mobile architecture is shown in Figure 1.6. An example of these kinds of systems is peer-to-peer networks in which data (B machines) can be transferred between the connected nodes (CSP controllers).

![Figure 1.6: Mobile CSP || B architecture](image)

The following step is the consistency verification. We first prove a theorem for establishing divergence freedom of the parallel combination between each controller and its controlled B machines. This includes the machines it owns initially and the machines it receives from other controllers during the execution. We then provide theorems in order to establish divergence freedom and deadlock freedom
of the whole mobile system containing several CSP controllers each controlling
several B machines.

The final contribution of this thesis is a case study which demonstrates all our
achievements in developing an example of a peer-to-peer network.

In summary, by the end of this document, we intend to have provided a formal
framework based on $CSP \parallel B$ which enables us to specify and verify concurrent
systems with mobile architecture as well as the previous static architecture.

1.4 Thesis Outline

In Chapter 2, we present new techniques and theorems proved during this research
for consistency verification of the systems that have not been covered before in
static $CSP \parallel B$.

Mobile $CSP \parallel B$ is introduced in Chapter 3. We start this chapter by describing
how we are able to define CSP semantics for a B machine which then results
in the fact that a B machine can be seen as a CSP process. The remainder of
this chapter presents our Mobile $CSP \parallel B$ framework. We describe how CSP
controllers can be combined and communicate with each other in the system and
how they can exchange B machines during the system execution. In addition,
we provide theorems which are sufficient to establish the divergence freedom and
deadlock freedom of the mobile combined communicating systems.

Chapter 4, presents a flight tickets sale system: a case study on Mobile $CSP \parallel B$
arquitectura. First, we describe the system and present its B and CSP speci-
fications. Then, we describe the consistency verification strategy by using the
theorems and wp technique, introduced in Chapter 3.

A comparison with other related works is given in Chapter 5.

Finally, conclusions and ideas for future research are presented in Chapter 6.
Chapter 2

New results for CSP \( \parallel B \)

This chapter presents new results in consistency verification of CSP \( \parallel B \) specifications. All techniques and theorems for establishing consistency in the CSP \( \parallel B \) approach have been focused on systems in which B machines contain only preconditioned operations, or a parallel combination of a single CSP process controlling a single B machine which contains only guarded operations. However, we would like to be able to check consistency in all possible kinds of combined systems involving both guards and preconditions.

In this chapter, we present new techniques and theorems proved during this research in order to check the consistency of system configurations that have not been covered before. First, we present two theorems to establish that in an individual CSP \( \parallel B \) controlled component containing any kind of B machine operations, hiding the CSP events which are not B machine operations does not affect the deadlock properties of the parallel combination. Next, we present theorems for establishing deadlock freedom of systems which contain parallel combinations of controlled components containing any kind of B machine operations. Finally, we introduce theorems and techniques in order to establish consistency of a single CSP process controlling a single B machine containing both pre-conditioned and guarded operations.

2.1 Deadlock freedom and hiding communication channels

In this section, we consider a parallel combination of a CSP controller and its controlled B machine. We also consider the same parallel combination where
the communication channels of the CSP controller are hidden. By providing two theorems, we prove that deadlock freedom of either of these parallel combinations implies the deadlock freedom of the other parallel combination.

**Theorem 1.** Supposing \( P \) is a CSP controller for \( B \) machine \( M \), and \( A \subseteq (\alpha(P) - \alpha(M)) \). If \( (P \setminus A) \parallel M \) is deadlock free, then \( P \parallel M \) is deadlock free.

**Proof.** The two parallel combinations are shown in Figures 2.1 and 2.2. We prove this theorem by contradiction:

We assume that \( P \parallel M \) has a deadlock.

Then \( \exists tr \in \text{traces}(P \parallel M) \cdot (tr, \alpha(P)) \) is a failure of \( P \parallel M \)

So, according to CSP semantics of hiding: \((tr \setminus A, \alpha(P \setminus A))\) is a failure of \((P \parallel M) \setminus A\)

But \( A \cap \alpha(M) = \emptyset \Rightarrow (P \parallel M) \setminus A = (P \setminus A) \parallel M \)

So, \((tr \setminus A, \alpha(P \setminus A))\) is a failure of \((P \setminus A) \parallel M \)

So, \((P \setminus A) \parallel M \) has a deadlock.

We obtain a contradiction, thus establishing the theorem. \( \square \)

---

**Theorem 2.** Supposing \( P \) is a CSP controller for \( B \) machine \( M \), and \( A \subseteq (\alpha(P) - \alpha(M)) \). If \( P \parallel M \) is deadlock free, then \( (P \setminus A) \parallel M \) is deadlock free.

**Proof.** The two parallel combinations are shown in Figures 2.1 and 2.2 (these are the same Figures as used in Theorem 1). We prove this theorem by contradiction:

We assume that \((P \setminus A) \parallel M \) has a deadlock.
2.2. DEADLOCK FREEDOM OF TWO COMBINED COMPONENTS

Then \( \exists \text{tr}' \in \text{traces}((P \setminus A) \parallel M) \bullet (\text{tr}', \alpha(P \setminus A)) \) is a failure of \((P \setminus A) \parallel M\)

\( \forall \text{tr}' \in \text{traces}((P \setminus A) \parallel M) \exists \text{tr} \in \text{traces}(P \parallel M) \bullet \text{tr}' = \text{tr} \setminus A \)

So, \((\text{tr} \setminus A, \alpha(P \setminus A))\) is a failure of \((P \setminus A) \parallel M\)

But \(A \cap \alpha(M) = \emptyset \implies (P \parallel M) \setminus A = (P \setminus A) \parallel M\)

So, \((\text{tr} \setminus A, \alpha(P \setminus A))\) is a failure of \((P \parallel M) \setminus A\)

So, according to CSP semantics of hiding; \((\text{tr}, \alpha(P))\) is a failure of \(P \parallel M\)

So, \(P \parallel M\) has a deadlock.

We obtain a contradiction, thus establishing the theorem.

2.2 Deadlock freedom of two combined components

In this section, we present three different conditions each in the context of a theorem for deadlock freedom verification of a system containing two combined components. If any of these three conditions is satisfied in a system containing two combined components, then the system is ensured to be deadlock free.

Theorem 3. Supposing \(P_1\) is a CSP controller for B machine \(M_1\), and \(P_2\) is a CSP controller for B machine \(M_2\). If \((P_1 \parallel M_1)\) and \((P_2 \parallel M_2)\) are deadlock free, and \(#(\alpha(P_1) \cap \alpha(P_2)) = 1\), then \((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)\) is deadlock free.

![Diagram](image-url)

Figure 2.3: Parallel combinations with one common event between two controllers

Proof. The system is shown in Figure 2.3. We prove this theorem by contradiction:
We assume that \((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)\) is not deadlock free. So:
\[
\exists \text{tr} \in \text{traces}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)) \cdot (\text{tr}, \Sigma) \in \text{failures}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2))
\]
We assume \(c\) is the common event between \(P_1\) and \(P_2\). So: \(\alpha(P_1) \cap \alpha(P_2) = \{c\}\)

We introduce \(\text{tr}_1, \text{tr}_2, \text{tr}_3, \text{tr}_4, X_1, X_2, X_3, X_4\) as described below:
\[
\begin{align*}
\text{tr}_1 &= \text{tr} \upharpoonright \alpha(P_1) \\
\text{tr}_2 &= \text{tr} \upharpoonright \alpha(P_2) \\
\text{tr}_3 &= \text{tr} \upharpoonright \alpha(M_1) \\
\text{tr}_4 &= \text{tr} \upharpoonright \alpha(M_2) \\
(\text{tr}_1, X_1) &\in \text{failures}(P_1) \\
(\text{tr}_2, X_2) &\in \text{failures}(P_2) \\
(\text{tr}_3, X_3) &\in \text{failures}(M_1) \\
(\text{tr}_4, X_4) &\in \text{failures}(M_2)
\end{align*}
\]
\[X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma \quad (1)\]

\(X_1 \cup X_2\) is what \(P_1 \parallel M_1\) refuses. \(P_1 \parallel M_1\) is deadlock free. So:
\[
\exists \alpha \in (\alpha(P_1) \cup \alpha(M_1)) \cdot \alpha \not\in (X_1 \cup X_2)
\]
We consider each of these possibilities in turn:

1. If \(\alpha\) is not event \(c\):
\[
\begin{align*}
\alpha \in \alpha(P_1) \quad & \alpha(P_1) \cap \alpha(P_2) = \{c\} \\
\implies \alpha \not\in \alpha(P_2) & \implies \alpha \not\in X_2 \\
(a \not\in \alpha(P_2)) \land (\alpha(M_2) \subseteq \alpha(P_2)) & \implies a \not\in \alpha(M_2) \implies a \not\in X_4 \\
(a \not\in X_2) \land (a \not\in X_4) & \implies a \not\in (X_2 \cup X_4) \\
(a \not\in (X_2 \cup X_4)) \land (a \not\in (X_1 \cup X_3)) & \implies a \not\in (X_1 \cup X_2 \cup X_3 \cup X_4) \\
(a \not\in (X_1 \cup X_2 \cup X_3 \cup X_4)) \land (a \in \Sigma) & \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \neq \Sigma
\end{align*}
\]
This contradicts (1). So it contradicts the fact that \((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)\) has deadlock.

2. If \(\alpha\) is event \(c\):
\[
\begin{align*}
(c \in \alpha(P_1)) \land (\alpha(P_1) \cap \alpha(M_2) = \emptyset) & \implies c \not\in \alpha(M_2) \implies c \not\in X_4 \\
(c \not\in (X_1 \cup X_3)) \land (c \not\in X_4) & \implies c \not\in (X_1 \cup X_3 \cup X_4) \\
(c \not\in (X_1 \cup X_3 \cup X_4)) \land (X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma) & \implies c \in X_2
\end{align*}
\]
2.2. DEADLOCK FREEDOM OF TWO COMBINED COMPONENTS

On the other hand, $X_2 \cup X_4$ is what $P_2 \parallel M_2$ refuses. $P_2 \parallel M_2$ is deadlock free. So: $\exists b : (\alpha(P_2) \cup \alpha(M_2)) \land b \not\in (X_2 \cup X_4)$

\[
\begin{align*}
    c &\in X_2 \\
    b &\not\in (X_2 \cup X_4)
\end{align*} \implies b \neq c
\]

$(b \in \alpha(P_2)) \land (b \neq c) \implies b \in (\alpha(P_2) - \{c\})$

Similar to what have been proved for $a$ in section 1, we can achieve the same result for $b$:

$b \not\in (X_1 \cup X_2 \cup X_3 \cup X_4) \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \neq \Sigma$

This result also contradicts (1). So, it contradicts the fact that $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ has deadlock.

In both cases we obtain a contradiction, thus establishing the theorem. \qed

**Theorem 4.** Supposing $P_1$ is a CSP controller for B machine $M_1$, $P_2$ is a CSP controller for B machine $M_2$, $(P_1 \parallel M_1)$ and $(P_2 \parallel M_2)$ are deadlock free, and there is only one common channel between $P_1$ and $P_2$. If at least one of the CSP controllers is non-discriminating on the channel common to $P_1$ and $P_2$, then $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ is deadlock free.

![Figure 2.4: Parallel combinations with one common channel between two controllers](image)

*Proof.* The system is shown in Figure 2.4. We assume $c$ is the common channel between $P_1$ and $P_2$, and $P_2$ is non-discriminating on channel $c$. According to [58], if $P_2$ is non-discriminating on channel $c$, it means that at any stage of the execution, $P_2$ can either refuse all events in $\{c\}$ or none of them (Definition 1, page 17). We prove this theorem by contradiction:
We assume that \((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)\) is not deadlock free. So:

\(\exists \tau \in \text{traces}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)) \cdot (\tau, \Sigma) \in \text{failures}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2))\)

We introduce \(\tau_1, \tau_2, \tau_3, \tau_4, X_1, X_2, X_3, X_4\) as described below:

\[
\begin{align*}
\tau_1 &= \tau \upharpoonright \alpha(P_1) \\
\tau_2 &= \tau \upharpoonright \alpha(P_2) \\
\tau_3 &= \tau \upharpoonright \alpha(M_1) \\
\tau_4 &= \tau \upharpoonright \alpha(M_2)
\end{align*}
\]

\((\tau_1, X_1) \in \text{failures}(P_1)\)

\((\tau_2, X_2) \in \text{failures}(P_2)\)

\((\tau_3, X_3) \in \text{failures}(M_1)\)

\((\tau_4, X_4) \in \text{failures}(M_2)\)

Without loss of generality \(X_1, X_2, X_3, X_4\) are maximal refusals and

\[
X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma \quad (1)
\]

\(X_1 \cup X_3\) is what \(P_1 \parallel M_1\) refuses. \(P_1 \parallel M_1\) is deadlock free. So:

\(\exists a \in (\alpha(P_1) \cup \alpha(M_1)) \cdot a \notin (X_1 \cup X_3)\)

We consider each of these possibilities in turn:

1. If \(a \notin \{[c]\}\):

\(a \in \alpha(P_1) \implies a \notin \alpha(P_2) \implies a \notin X_2\)

\((a \notin \alpha(P_2)) \land (\alpha(M_2) \subseteq \alpha(P_2)) \implies a \notin \alpha(M_2) \implies a \notin X_4\)

\((a \notin X_2) \land (a \notin X_4) \implies a \notin (X_2 \cup X_4)\)

\((a \notin (X_2 \cup X_3)) \land (a \notin (X_1 \cup X_3)) \implies a \notin (X_1 \cup X_2 \cup X_3 \cup X_4)\)

\((a \notin (X_1 \cup X_2 \cup X_3 \cup X_4)) \land (a \in \Sigma) \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \neq \Sigma\)

This contradicts (1). So it contradicts the fact that \((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)\) has deadlock.

2. If \(a \in \{[c]\}\):

We assume \(T\) is the type of channel \(c\). So: \(\exists v \in T \cdot a = c.v\)

\((c.v \in \alpha(P_1)) \land (\alpha(P_1) \cap \alpha(M_2) = \emptyset) \implies c.v \notin \alpha(M_2) \implies c.v \notin X_4\)

\((c.v \notin (X_1 \cup X_3)) \land (c.v \notin X_4) \implies c.v \notin (X_1 \cup X_3 \cup X_4)\)

\((c.v \notin (X_1 \cup X_3 \cup X_4)) \land (X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma) \implies c.v \in X_2\)
2.2. DEADLOCK FREEDOM OF TWO COMBINED COMPONENTS

On the other hand, $X_2 \cup X_4$ is what $P_2 \parallel M_2$ refuses. $P_2 \parallel M_2$ is deadlock free. So: $\exists b \in (\alpha(P_2) \cup \alpha(M_2)) \cdot b \notin (X_2 \cup X_4)$

$c, v \in X_2$

$P_2$ is non-discriminating on $c$
$b \notin X_2$

$X_2$ is the maximal refusal of $P_2$

\[
(b \notin \alpha(P_1)) \land (\alpha(M_1) \subseteq \alpha(P_1)) \implies b \notin \alpha(M_1) \implies b \notin X_1
\]

\[
(b \notin X_1 \land b \notin X_3) \implies b \notin X_1 \cup X_3
\]

\[
(b \notin (X_2 \cup X_4)) \land (b \notin (X_1 \cup X_3)) \implies b \notin (X_1 \cup X_2 \cup X_3 \cup X_4)
\]

\[
(b \notin (X_1 \cup X_2 \cup X_3 \cup X_4)) \land (b \in \Sigma) \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \neq \Sigma
\]

This result also contradicts (1). So, it contradicts the fact that $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ has deadlock.

In both cases we obtain a contradiction, thus establishing the theorem. \qed

**Theorem 5.** Supposing $P_1$ is a CSP controller for $B$ machine $M_1$, $P_2$ is a CSP controller for $B$ machine $M_2$, and $(P_1 \parallel M_1)$ and $(P_2 \parallel M_2)$ are deadlock free. If one of the CSP controllers is open on the channels common to $P_1$ and $P_2$, and the other controller is non-discriminating on the common channels, then $(P_1 \parallel M_1) \parallel (P_2 \parallel M_2)$ is deadlock free.

![Figure 2.5: Parallel combinations with more than one common channels between two controllers](image-url)
Proof. The system is shown in Figure 2.5. We assume \( P_2 \) is open on the channels common to \( P_1 \) and \( P_2 \), and \( P_1 \) is non-discriminating on the common channels. According to [58], if \( P_2 \) is open on a channel, it means that at any stage of the execution, at least one of the events associated with that channel is not refused by \( P_2 \) (Definition 2, page 17). We prove this theorem by contradiction:

We assume that \( (P_1 \parallel M_1) \parallel (P_2 \parallel M_2) \) is not deadlock free. So:

\[
\exists tr \in \text{traces}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2)) \bullet (tr, \Sigma) \in \text{failures}((P_1 \parallel M_1) \parallel (P_2 \parallel M_2))
\]

We introduce \( tr_1, tr_2, tr_3, tr_4, X_1, X_2, X_3, X_4 \) as described below:

\[
\begin{align*}
tr_1 &= tr \upharpoonright \alpha(P_1) \\
tr_2 &= tr \upharpoonright \alpha(P_2) \\
tr_3 &= tr \upharpoonright \alpha(M_1) \\
tr_4 &= tr \upharpoonright \alpha(M_2) \\
(tr_1, X_1) &\in \text{failures}(P_1) \\
(tr_2, X_2) &\in \text{failures}(P_2) \\
(tr_3, X_3) &\in \text{failures}(M_1) \\
(tr_4, X_4) &\in \text{failures}(M_2)
\end{align*}
\]

Without loss of generality, \( X_1, X_2, X_3, X_4 \) are maximal refusals \( \land \)

\[
X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma \quad (1)
\]

\( X_1 \cup X_3 \) is what \( P_1 \parallel M_1 \) refuses. \( P_1 \parallel M_1 \) is deadlock free. So:

\[
\exists a \in (\alpha(P_1) \cup \alpha(M_1)) \bullet a \not\in (X_1 \cup X_3)
\]

We consider each of these possibilities in turn:

1. If \( a \not\in (\alpha(P_1) \cap \alpha(P_2)) \):

   \[
   a \in \alpha(P_1) \implies a \not\in \alpha(P_2) \implies a \not\in X_2
   \]

   \( a \not\in \alpha(P_2) \land (\alpha(M_2) \subseteq \alpha(P_2)) \implies a \not\in \alpha(M_2) \implies a \not\in X_4 \)

   \( a \not\in X_2 \land (a \not\in X_4) \implies a \not\in (X_2 \cup X_4) \)

   \( a \not\in (X_2 \cup X_4) \land (a \not\in (X_1 \cup X_3)) \implies a \not\in (X_1 \cup X_2 \cup X_3 \cup X_4) \)

   \( (a \not\in (X_1 \cup X_2 \cup X_3 \cup X_4)) \land (a \in \Sigma) \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \not= \Sigma \)

This contradicts (1). So, it contradicts the fact that \( (P_1 \parallel M_1) \parallel (P_2 \parallel M_2) \) has deadlock.
2. If \( a \in (\alpha(P_1) \cap \alpha(P_2)) \):

It follows that \( a = c.v \) such that \( c \) is one of the common channels between \( P_1 \) and \( P_2 \), and \( T \) is the type of channel \( c \).

\[
(c.v \in \alpha(P_1)) \land (\alpha(P_1) \cap \alpha(M_2) = \emptyset) \implies c.v \not\in \alpha(M_2) \implies c.v \not\in X_4
\]

\[
(c.v \not\in (X_1 \cup X_3)) \land (c.v \not\in X_4) \implies c.v \not\in (X_1 \cup X_3 \cup X_4)
\]

\[
(c.v \not\in (X_1 \cup X_3 \cup X_4)) \land (X_1 \cup X_2 \cup X_3 \cup X_4 = \Sigma) \implies c.v \in X_2
\]

On the other hand, \( P_2 \) is open on channel \( c \). So: \( \exists w \in T : c.w \not\in X_2 \quad (2) \)

\[
c.w \not\in \alpha(M_2) \implies c.w \not\in X_4 \quad (3)
\]

\[
c.w \not\in \alpha(M_1) \implies c.w \not\in X_3 \quad (4)
\]

\[
c.v \not\in X_1
\]

\[
P_1 \text{ is non-discriminating on channel } c \quad \Longrightarrow \{ [c] \not\in X_1 \implies c.w \not\in X_1 \quad (5)\}
\]

\[
P_1 \text{ is non-discriminating on channel } c \}
\]

(2), (3), (4), (5) \implies c.w \not\in (X_1 \cup X_2 \cup X_3 \cup X_4)

\[
(c.w \not\in (X_1 \cup X_2 \cup X_3 \cup X_4)) \land (c.w \in \Sigma) \implies (X_1 \cup X_2 \cup X_3 \cup X_4) \not= \Sigma
\]

This result also contradicts (1). So, it contradicts the fact that \((P_1 || M_1) || (P_2 || M_2)\) has deadlock.

In both cases we obtain a contradiction, thus establishing the theorem. \( \square \)

### 2.3 Deadlock freedom of the whole system

In this section, we present some conditions in the context of a theorem which guarantee the deadlock freedom of the whole combined communicating system. If these conditions are satisfied in a system containing parallel combinations of controlled components, then the system is ensured to be deadlock free.

In the following theorem, synchronisation is restricted under particular conditions. Condition 3 states that it should not be more than one synchronisation event that a pair of controlled components shares with the rest of the system.
Theorem 6. If a system can be divided into $n$ different parts ($n \geq 2$), each called $S_i$ ($1 \leq i \leq n$), such that:

1. Each $S_i$ contains at most two CSP controllers and their controlled B machines.
2. $S_1, S_2, ..., S_n$ are deadlock free.
3. $\forall 1 \leq i \leq n \cdot \#(\alpha S_i \cap (\alpha S_1 \cup \alpha S_2 \cup ... \cup \alpha S_{i-1} \cup \alpha S_{i+1} \cup \alpha S_{i+2} \cup ... \cup \alpha S_n)) \leq 1$

then the system is deadlock free.

Proof. An example of a system divided into $n$ different parts is shown in Figure 2.6. We prove this theorem by contradiction:

We assume that the system, $SYS$, is not deadlock free.

So: $\exists tr \in traces(SYS) \cdot (tr, \Sigma) \in failures(SYS)$

We introduce $tr_1, tr_2, ..., tr_n, X_1, X_2, ..., X_n$ as described below:

$$
tr_1 = tr \upharpoonright \alpha(S_1) \\
tr_2 = tr \upharpoonright \alpha(S_2) \\
\vdots \\
tr_n = tr \upharpoonright \alpha(S_n) \\
(tr_1, X_1) \in failures(S_1) \\
(tr_2, X_2) \in failures(S_2) \\
\vdots \\
(tr_n, X_n) \in failures(S_n)
$$

Without loss of generality $\forall 1 \leq i \leq n \cdot X_i$ is the maximal refusal of $S_i$ and

$$X_1 \cup X_2 \cup ... \cup X_n = \Sigma \quad (1)$$

$X_i$ is what $S_i$ refuses. $S_i$ is deadlock free. So: $\exists \alpha \in \alpha(S_i) \cdot \alpha \notin X_i$

We consider each of these possibilities in turn:

1. If $\alpha \notin (\alpha S_i \cap (\alpha S_1 \cup \alpha S_2 \cup ... \cup \alpha S_{i-1} \cup \alpha S_{i+1} \cup \alpha S_{i+2} \cup ... \cup \alpha S_n))$

$$
a \notin \bigcup_{k \neq i} X_k \\
a \notin X_i \\
\implies a \notin \bigcup_{k=1}^{n} X_k \implies \bigcup_{k=1}^{n} X_k \notin \Sigma
$$
This contradicts (1). So it contradicts the fact that the system has deadlock.

2. If \( a \in (\alpha S_i \cap (\bigcup_{k \neq j} \alpha S_k)) \):

So, \( \exists j \neq i \cdot (a \in (\alpha S_i \cap \alpha S_j) \land a \in X_j) \)

\( S_j \) is deadlock free. So: \( \exists b \in \alpha S_j \cdot b \notin X_j \)

\( (a \in X_j) \land (b \notin X_j) \implies b \neq a \)

\( \#((\alpha S_j \cap (\bigcup_{k \neq j} \alpha S_k))) \leq 1 \)

\( (\alpha S_j \cap \alpha S_i) = \{a\} \)

\( b \notin \bigcup_{k \neq j} \alpha S_k \implies b \notin \bigcup_{k \neq j} X_k \)

\( b \notin \bigcup_{k \neq j} X_k \implies b \notin X_j \)

This result also contradicts (1). So, it contradicts the fact that the system has deadlock.

In both cases we obtain a contradiction, thus establishing the theorem.

\[\square\]

Figure 2.6: Parallel combinations of controlled components divided into pairs

### 2.4 Consistency of a mixed machine and its controller

As we described before in Chapter 1, an operation in a B machine is either pre-conditioned or guarded. Pre-conditioned operations have the form \texttt{PRE S THEN T END}, where \( S \) is the precondition and \( T \) is the body of the operation. A pre-conditioned operation is able to be executed when its precondition is not satisfied, but there will not be any guarantees about the resulting execution. Guarded operations have the form \texttt{SELECT G THEN R END}, where \( G \) is a
CHAPTER 2. NEW RESULTS FOR CSP || B

Guard and \( R \) is the body of the operation. A guarded operation is not able to be executed when its guard is not satisfied.

In this section, we present our proof strategy in order to establish divergence freedom and deadlock freedom of a parallel combination between a CSP controller \( P \) and a mixed B machine \( M \). A mixed B machine is a B machine which contains both pre-conditioned and guarded operations. For instance, a mixed machine \( M \) containing two pre-conditioned operations and two guarded operations can be specified as below:

\[
\text{MACHINE M}
\]

\[
\text{OPERATIONS}
\]

\[
a = \text{PRE } p(a) \text{ THEN } body(a) \text{ END;}
\]

\[
b = \text{PRE } p(b) \text{ THEN } body(b) \text{ END;}
\]

\[
c = \text{SELECT } g(c) \text{ THEN } body(c) \text{ END;}
\]

\[
d = \text{SELECT } g(d) \text{ THEN } body(d) \text{ END}
\]

where:

\[
p(a): \text{precondition of operation } a
\]

\[
body(a): \text{body of operation } a
\]

\[
p(b): \text{precondition of operation } b
\]

\[
body(b): \text{body of operation } b
\]

\[
g(c): \text{guard of operation } c
\]

\[
body(c): \text{body of operation } c
\]

\[
g(d): \text{guard of operation } d
\]

\[
body(d): \text{body of operation } d
\]

Assuming a system containing \( M \) (described above) as the B machine and a CSP process \( P \) as the CSP controller, \( P \parallel M \) is shown in Figure 2.7.

In our proof strategy, we change guarded operations of the mixed B machine to pre-conditioned operations and we introduce a new machine which then the divergence freedom of the original system results from the divergence freedom of the new system. For deadlock freedom verification, we drop the precondition of the pre-conditioned operations in the mixed B machine and we introduce a new machine which then the deadlock freedom of the original system results from the deadlock freedom of the new system.

### 2.4.1 Divergence freedom

If we change any guarded operation \( \text{SELECT } G \text{ THEN } R \text{ END} \) in a mixed B machine \( M \) to pre-conditioned operation \( \text{PRE } TRUE \text{ THEN IF } G \text{ THEN } R \text{ END END} \),
2.4. CONSISTENCY OF A MIXED MACHINE AND ITS CONTROLLER

we will have a new machine $M_1$ which has only pre-conditioned operations. For instance, if we assume the mixed B machine $M$ as described above, machine $M_1$ will be specified as below:

MACHINE $M_1$

OPERATIONS

$a = \text{PRE } p(a) \text{ THEN } body(a) \text{ END;}

b = \text{PRE } p(b) \text{ THEN } body(b) \text{ END;}

c = \text{PRE TRUE THEN}

\quad \text{IF } g(c) \text{ THEN } body(c)

\quad \text{END}

\quad \text{END;}

d = \text{PRE TRUE THEN}

\quad \text{IF } g(d) \text{ THEN } body(d)

\quad \text{END}

\quad \text{END}

\text{END}

In fact, we add a precondition $\text{TRUE}$ to the guarded operations and we also change the guarded statement $\text{SELECT}$ to the statement $\text{IF}$. The reason of changing the statement $\text{SELECT}$ to $\text{IF}$ is because $\text{SELECT}$ is not allowed to be used in the body of B machine operations in $\text{CSP} \parallel B$. This is explained more in Chapter 3. Precondition $\text{TRUE}$ is always correct, so new operations will always be called inside their precondition and they cannot make any divergence in B machine $M_1$. The remainder of this section presents the steps towards proving
that the divergences of $M$ is a subset of the divergences of $M_1$ which then enables us to establish the divergence freedom of $P \parallel M$ by establishing the divergence freedom of $P \parallel M_1$.

**Lemma 1.** If $tr$ is a finite sequence of operations, then $wp(tr, Q)_{M_1} \Rightarrow wp(tr, Q)_M$, where $wp(tr, Q)_{M_1}$ and $wp(tr, Q)_M$ represent the weakest precondition for $tr$ to achieve post condition $Q$ according to the specifications of the operations in $M_1$ and $M$ respectively.

**Proof.** We prove this lemma by using the Principle of Induction on $tr$.

**Base case:** $tr = ()$

This case is easily proved as $wp(\emptyset, Q)_{M_1} = wp(\emptyset, Q)_M = Q$

**Inductive case:** $(op) \subseteq tr''$

Assume Inductive Hypothesis for $tr''$. $tr'$ is a finite sequence of operations and $op$ is an operation. There are two possibilities for $op$:

**Situation 1:** $op$ is a pre-conditioned operation in $M$

1. $wp((op) \subseteq tr'', Q)_{M_1} = wp((op), wp(tr'', Q)_{M_1})_{M_1}$
2. $wp(tr'', Q)_{M_1} \Rightarrow wp(tr'', Q)_M$ Induction assumption
3. $wp((op), wp(tr'', Q)_{M_1})_{M_1} \Rightarrow wp((op), wp(tr'', Q)_M)_{M_1}$
4. $wp((op) \subseteq tr'', Q)_{M_1} \Rightarrow wp((op), wp(tr'', Q)_M)_{M_1}$
5. $wp((op), wp(tr'', Q)_M)_{M_1} = wp((op), wp(tr'', Q)_M)_M$ $op$ in $M$ is the same as $op$ in $M_1$
6. $wp((op) \subseteq tr'', Q)_{M_1} \Rightarrow wp((op), wp(tr'', Q)_M)_M$ $4, 5$
7. $wp((op) \subseteq tr'', Q)_{M_1} \Rightarrow wp((op) \subseteq tr'', Q)_M$ $6$
Situation 2: \( \text{op} \) is a guarded operation in \( M \)

We assume \( \text{op} = \text{SELECT } G \text{ THEN } R \text{ END} \) in \( M \). Therefore, \( \text{op} = \text{PRE TRUE THEN IF } G \text{ THEN } R \text{ END END} \) in \( M_1 \).

1. \( \wp((\text{op}) \land \text{tr}', Q)_{M_1} = \wp((\text{op}), \wp(\text{tr}', Q)_{M_1})_{M_1} \) \( \ \wp \text{ in } M_1 \)

2. \( \wp(\text{tr}', Q)_{M_1} \Rightarrow \wp(\text{tr}', Q)_{M} \) \( \ \text{Induction assumption} \)

3. \( \wp((\text{op}), \wp(\text{tr}', Q)_{M_1})_{M_1} \Rightarrow \wp((\text{op}), \wp(\text{tr}', Q)_{M})_{M_1} \) \( \ \text{2} \)

4. \( \wp((\text{op}) \land \text{tr}', Q)_{M_1} \Rightarrow \wp((\text{op}), \wp(\text{tr}', Q)_{M})_{M_1} \) \( \ \text{1, 3} \)

5. \( \wp((\text{op}), \wp(\text{tr}', Q)_{M_1})_{M_1} = \wp(\text{IF } G \text{ THEN } R \text{ END}, \wp(\text{tr}', Q)_{M}) \) \( \ \text{op in } M_1 \)

6. \( \wp((\text{op}), \wp(\text{tr}', Q)_{M_1})_{M_1} = G \Rightarrow \wp(\text{R}, \wp(\text{tr}', Q)_{M}) \land \neg G \Rightarrow \wp(\text{tr}', Q)_{M} \)

7. \( \wp((\text{op}), \wp(\text{tr}', Q)_{M})_{M_1} \Rightarrow G \Rightarrow \wp(\text{R}, \wp(\text{tr}', Q)_{M}) \)

8. \( \wp((\text{op}) \land \text{tr}', Q)_{M_1} \Rightarrow G \Rightarrow \wp(\text{R}, \wp(\text{tr}', Q)_{M}) \) \( \ \text{4, 7} \)

9. \( \wp((\text{op}) \land \text{tr}', Q)_{M} = \wp((\text{op}), \wp(\text{tr}', Q)_{M})_{M} \) \( \ \wp \text{ in } M \)

10. \( \wp((\text{op}) \land \text{tr}', Q)_{M} = G \Rightarrow \wp(\text{R}, \wp(\text{tr}', Q)_{M}) \) \( \ \text{9, op in } M \)

11. \( \wp((\text{op}) \land \text{tr}', Q)_{M_1} \Rightarrow \wp((\text{op}) \land \text{tr}', Q)_{M} \) \( \ \text{8, 10} \)

Since we have proved the Base case and the Inductive case, it establishes the lemma.

\( \square \)

The corollary below follows immediately from Lemma 1 using \( \text{true} \) for \( Q \).

Corollary 1. If \( \text{tr} \) is a finite sequence of operations, then:
\( \wp(\text{tr}, \text{true})_{M_1} \Rightarrow \wp(\text{tr}, \text{true})_{M} \)
Now, we are able to present the next lemma stating that the divergences of \( M \) is a subset of the divergences of \( M_1 \).

**Lemma 2.** \( \mathcal{D}[M] \subseteq \mathcal{D}[M_1] \)

**Proof.** We introduce \( tr \) as an arbitrary sequence of operations.

\[
wp(init; tr, true)_{M_1} = wp(init, wp(tr, true)_{M_1}) \quad (1)
\]

Corollary 1 \( \implies \) \( wp(init, wp(tr, true)_{M_1}) \implies wp(init; wp(tr, true)_{M}) \) \( (2) \)

\[
wp(init; wp(tr, true)_{M}) = wp(init; tr, true)_{M} \quad (3)
\]

(2), (3) \( \implies \) \( wp(init, wp(tr, true)_{M_1}) \implies wp(init; tr, true)_{M} \) \( (4) \)

(1), (4) \( \implies \) \( wp(init; tr, true)_{M_1} \Rightarrow wp(init; tr, true)_{M} \)

Up to now, we have proved that if \( tr \) is not a divergence of \( M_1 \), then it is not a divergence of \( M \). This result implies the fact that if \( tr \) is a divergence of \( M \), then it is also a divergence of \( M_1 \). In other words:

\[ \neg wp(init; tr, true)_{M} \Rightarrow \neg wp(init; tr, true)_{M_1} \]

But, we introduced \( tr \) as an arbitrary sequence of operations. Therefore:

\[ \forall tr \ (tr \in \mathcal{D}[M] \Rightarrow tr \in \mathcal{D}[M_1]) \]

Thus: \( \mathcal{D}[M] \subseteq \mathcal{D}[M_1] \)

The next corollary follows from Lemma 2.

**Corollary 2.** \( \mathcal{D}[P \mid M] \subseteq \mathcal{D}[P \mid M_1] \)

The next theorem follows immediately from Corollary 2.

**Theorem 7.** supposing \( P \) is a CSP controller for a mixed B machine \( M \). \( (P \mid M) \) is divergence free if \( (P \mid M_1) \) is divergence free, where \( M_1 \) is the same B machine as \( M \) except any guarded operation \( \text{SELECT } G \text{ THEN } R \text{ END} \) in \( M \) is changed to \( \text{PRE TRUE THEN IF } G \text{ THEN } R \text{ END END} \) in \( M_1 \).

[70] has already proved a theorem to establish the divergence freedom of a parallel combination between a CSP controller and its controlled B machine which contains only pre-conditioned operations. By using this theorem, if \( (P \mid M_1) \) is established to be divergence free, then \( (P \mid M) \) is also divergence free.
2.4.2 Deadlock freedom

Our proof of deadlock freedom is based on that \((P \parallel M)\) is divergence free. This condition ensures that the system is always in the stable state during the execution. If \((P \parallel M)\) is not divergence free, there is no point to check deadlock freedom because there is no guarantee about its behaviour after reaching divergent state and it always may have deadlock after the divergent state.

If an operation contains precondition, whatever the precondition is, it will execute whenever it is called even if its precondition is not satisfied. If we change any pre-conditioned operation \(PRE S \quad THEN \quad T \quad END\) in a mixed B machine \(M\) to guarded operation \(SELECT \quad TRUE \quad THEN \quad T \quad END\), we will have a new machine, \(M_2\), which has only guarded operations. For instance, if we assume the mixed B machine \(M\) as described at the beginning of Section 2.4, machine \(M_2\) will be specified as below:

**MACHINE** \(M_2\)

```
OPERATIONS
a = SELECT TRUE THEN body(a) END;
b = SELECT TRUE THEN body(b) END;
c = SELECT g(c) THEN body(c) END;
d = SELECT g(d) THEN body(d) END
END
```

The new guarded operations in \(M_2\) contain the guard \(TRUE\) and we know there is no guarded statement in the body of operations in B machines (see Chapter 3 for more information). Therefore, these new guarded operations are always accepted by \(M_2\) and they can never be refused for execution. The rest of the operations are not changed, so their acceptance or refusal is the same in \(M\) and \(M_2\). Thus, we obtain the following corollary:

**Corollary 3.** If \((P \parallel M)\) is divergence free, then \((P \parallel M) =_{PD} (P \parallel M_2)\)

The next theorem follows immediately from Corollary 3.

**Theorem 8.** Supposing \(P\) is a CSP controller for a mixed B machine \(M\) and \((P \parallel M)\) is divergence free. \((P \parallel M)\) is deadlock free if \((P \parallel M_2)\) is deadlock free, where \(M_2\) is the same B machine as \(M\) except any pre-conditioned operation \(PRE S \quad THEN \quad T \quad END\) in \(M\) is changed to \(SELECT \quad TRUE \quad THEN \quad T \quad END\) in \(M_2\).

[68] has already proved a theorem to establish deadlock freedom of a parallel combination between a CSP controller and its controlled B machine which con-
tains only guarded operations. By using this theorem, we can check the deadlock freedom of \((P \parallel M_2)\) and the result is the same for \((P \parallel M)\).

2.5 Discussion

CSP \(\parallel B\) is among the method integrations in which the consistency verification of the combined specification is considered and some verification strategies had already been proved expressing the conditions required for divergence freedom and deadlock freedom of the systems specified in CSP \(\parallel B\) architecture. However, the consistency verification efforts did not cover all possible kinds of systems which can be specified in CSP \(\parallel B\). The previous theorems and techniques had provided the conditions which ensure the consistency of a single CSP process controlling a single B machine which contains only pre-conditioned operations, or the whole combined communicating system in which the B machines contain only pre-conditioned operations, or systems containing a parallel combination of a single CSP process controlling a single B machine which contains only guarded operations.

In this chapter, we extended the consistency verification in CSP \(\parallel B\) and successfully provided new theorems which express the conditions which must hold to ensure the consistency of the systems which had not been covered before. Section 2.4 presents theorems and techniques to establish the consistency of a parallel combination between a CSP controller and its controlled B machine which contains both pre-conditioned and guarded operations. This section in addition to the other two theorems which had already been proved for consistency verification of single controlled components now enable us to check the consistency of all possible kinds of single controlled components in a combined communicating system. Furthermore, in this chapter we provided conditions which must hold to establish the deadlock freedom of the whole combined communicating system. The previous deadlock freedom verification of the whole system had only provided the conditions required for establishing deadlock freedom for the whole system in which B machines contain only pre-conditioned operations which is extremely straightforward.

We believe the significant advantage of our work in this chapter is that the theorems we provided for consistency verification of systems containing two or more combined components do not depend on the kind of operations in B machines. The previous work for proving the consistency of the whole system was based on the fact that the operations in all B machines are only pre-conditioned. The theorems we proved in this chapter can be used for the systems containing all possible kinds of B machines: B machines with pre-conditioned operations, or B machines with guarded operations, or even B machines with both pre-conditioned
and guarded operations. In our verification strategy provided in this chapter, the kind of B machine operations are only important in proving the consistency of each combined component separately, which with our work in section 2.4 we are now able to prove it for all kinds of B machine operations in a combined component. To establish the deadlock freedom of the whole system, there is no restriction in the kind of B machine operations in the system.

In consistency proof of the whole system, we need to prove the consistency of each individual combined component separately. If a parallel combination of a controller and its controlled B machine in a large system has already been proved to be deadlock free, then if we abstract this large system by hiding some of the communication channels of that combined component, then according to Theorem 2 it is guaranteed that the new form of the combined component is also deadlock free. Therefore, we do not need to check the deadlock freedom of the new form of combined component again. On the other hand, Theorem 1 is helpful in the opposite way: when a system has been abstracted and then we need to have some or all of communication channels of a controller visible in the system. In this case, if the abstract version of the combined component has already been established to be deadlock free, then we do not need to check the deadlock freedom of the same combined component when its communication channels are now visible. In addition, in a large system with several numbers of channels between controllers, if a controller contains lots of communication channels, then it is much easier to check the deadlock freedom of the parallel combination between that controller and its controlled B machine in an abstracted version when the communication channels are hidden. Then according to Theorem 1, the result is true for the original version.

To conclude this chapter, we presented theorems and techniques to provide the conditions which can establish the consistency of the systems containing all possible kinds of B machine operations in CSP || B architecture. Our achievements besides previous works enable us to check and verify the consistency of any system specified in CSP || B. The theorems we provided in this chapter can also be applied in consistency verification of any future approaches which are based on the parallel combination of CSP and Event-B [4].
Chapter 3

Mobile CSP || B

This chapter presents the Mobile CSP || B framework. We start this chapter by giving a detailed discussion about Morgan's CSP semantics for action systems which allows a B machine to be considered as a CSP process which then enables B machines and CSP processes to work in parallel with each other in a system. We then prove some lemmas detecting when divergence can happen in a B machine during the execution.

Section 3.2 provides an introduction to the structure of Mobile CSP || B. In sections 3.3 and 3.4, we define how a B machine and a CSP controller is developed in our mobile architecture respectively. Section 3.5 describes the whole mobile combined communicating system. In this section, we explain how CSP controllers can work in parallel, communicate with each other and exchange their owned B machines during the system execution. Section 3.6 presents a proof strategy containing lemmas and corollaries which lead to prove a theorem for establishing the divergence freedom of the parallel combination between a controller and the machines it works with during the execution. The next two sections provide the theorems to establish the divergence freedom and deadlock freedom in the whole mobile combined communicating system. In Section 3.9, we provide some proofs to establish that a refinement of a component can be used instead of the component in the system and the substitution does not have any effect on the system consistency properties. Finally, Section 3.10 presents a summary and some final considerations.
3.1 CSP semantics of B machines

In [50], Morgan presents a semantic link between CSP and action systems. He introduces traces, failures and divergences semantics of CSP for action systems by using weakest precondition formulae.

An action system consists of variables, an initialisation and a set of actions. An action $G \rightarrow \text{com}$ consists of a guard, $G$, and a command, $\text{com}$. The guard is a predicate and the command is a statement on the system's variables. An action is executed by executing its command which can only happen if its guard holds. An action is called enabled if it is able to be executed. An action system is represented by a pair $(A, \text{ini})$, where $A$ is the set of actions and $\text{ini}$ is the initialisation.

The weakest precondition semantics for an action $G \rightarrow \text{com}$ to achieve the postcondition $\alpha$ is given in [50] as follows:

**Definition 1.** $\text{wp}(G \rightarrow \text{com}, \alpha) \triangleq G \Rightarrow \text{wp}(\text{com}, \alpha)$

Actions do not satisfy Dijkstra's Law of the Excluded Miracle [25]. In other words, they do not satisfy $\text{wp}(G \rightarrow \text{com}, \text{false}) = \text{false}$. However, the command part $\text{com}$ and the initialisation satisfy this law.

[50] also introduces $\overline{\text{wp}}(\text{com}, \alpha)$ which denotes the states in which it is possible that $\text{com}$ establishes $\alpha$. In other words, $\overline{\text{wp}}(\text{com}, \alpha)$ denotes the states in which it is not certain that $\text{com}$ will establish $\neg \alpha$. The definition is as follows in [50]:

**Definition 2.** For any command $\text{com}$ and postcondition $\alpha$:

$\overline{\text{wp}}(\text{com}, \alpha) \triangleq \neg \text{wp}(\text{com}, \neg \alpha)$

If $tr$ is a sequence of actions, then $\overline{tr}$ is the sequential composition of its elements, with $\emptyset = \text{skip}$. The weakest precondition for a sequence of actions $tr$ to achieve a postcondition $Q$ means the weakest precondition for the sequential composition of its elements to achieve the postcondition $Q$. In other words:

$\text{wp}(tr, Q) = \overline{\text{wp}}(\overline{tr}, Q)$.

A trace of the action system $(A, \text{ini})$ is a finite sequence of actions. Supposing the set $A^*$ to be the set of all finite sequences of members of $A$, then the traces of the action system is given in [50] as follows:

**Definition 3.** The traces $tr \in A^*$ of the action system $(A, \text{ini})$ are those for which $\overline{\text{wp}}(\text{ini}; tr, \text{true})$ is true.

A pair $(tr, R)$ is a failure of an action system if after execution of the sequence of actions $tr$, the action system reaches a state in which none of the guards of the
actions in $R$ holds. In this case, the set $R$ is called a refusal of the action system.
The failures of an action system are defined in [50] as follows:

Definition 4. Supposing $R \subseteq A$ and $tr \in A^*$. The failures $(tr, R)$ of the action system $(A, ini)$ are those for which $\overline{wp}(\text{init}; tr, \neg gd(R))$ is true, where $gd(R)$ is the disjunction of the guards of the actions in $R$.

Divergence happens in an action system when an enabled action $G \rightarrow \text{com}$ is executed but the execution of its command does not terminate. As stated in [50], $wp(\text{com}, \text{true})$ implies termination of $\text{com}$. Therefore, divergence happens in an action system when an enabled action $G \rightarrow \text{com}$ is executed in a state where $wp(\text{com}, \text{true})$ is not true. The divergences of an action system are defined in [50] as follows:

Definition 5. The divergences $tr \in A^*$ of the action system $(A, ini)$ are those for which $wp(\text{init}; tr, \text{false})$ is true.

Finally, [50] presents a theorem which states that an action system with traces, failures and divergences based on definitions 3, 4, 5 can be seen as a CSP process.

Operations in a B machine can be understood as actions. A guarded operation $\text{SELECT} \ G \ \text{THEN} \ R \ \text{END}$ can be understood as an action in which $G$ is the guard and $R$ is the command. In other words, a guarded operation $\text{SELECT} \ G \ \text{THEN} \ R \ \text{END}$ can be understood as the action $G \rightarrow R$. A pre-conditioned operation $\text{PRE} \ P \ \text{THEN} \ S \ \text{END}$ can be understood as an action whose guard is the predicate $true$ and command is $\text{PRE} \ P \ \text{THEN} \ S \ \text{END}$. In other words, a pre-conditioned operation $\text{PRE} \ P \ \text{THEN} \ S \ \text{END}$ can be understood as the action $true \rightarrow \text{PRE} \ P \ \text{THEN} \ S \ \text{END}$. As a pre-conditioned operation is understood as an action whose guard is always true, it is always an enabled action and can always be executed. As the command part $com$ of actions and the initialisation must satisfy Dijkstra’s Law of the Excluded Miracle, the body of B operations and the initialisation are not allowed to contain $\text{SELECT}$ or $\text{ANY}$ (or $\text{Let}$ which is a special kind of $\text{ANY}$) statements.

The explanations above yield that a B machine is an action system in which operations take the role of actions. Therefore, we are able to define CSP semantics of traces, failures and divergences for B machines. Thus, a B machine can be seen as a CSP process. This result makes it possible for B machines to work in parallel with CSP processes in a system. CSP $\parallel B$ is an approach based on this achievement. Morgan’s work in [50] enables us to have a system which is a parallel combination of CSP processes and B machines.
Now, we present the following corollary:

**Corollary 1.** For a pre-conditioned operation \( \text{PRE} P \text{ THEN } S \text{ END} \) and post-condition \( Q \),

\[
wp(\text{true} \rightarrow \text{PRE} P \text{ THEN } S \text{ END}, Q) = wp(\text{PRE} P \text{ THEN } S \text{ END}, Q)
\]

**Proof.** The equivalence above is made precise since according to Definition 1:

\[
wp(\text{true} \rightarrow \text{PRE} P \text{ THEN } S \text{ END}, Q) = \text{true} \Rightarrow wp(\text{PRE} P \text{ THEN } S \text{ END}, Q)
\]

\[\square\]

In fact, we have established that in the weakest precondition formulas of Morgan's CSP semantics for a B machine, we can use the pre-conditioned operation \( \text{PRE} P \text{ THEN } S \text{ END} \) instead of the action \( \text{true} \rightarrow \text{PRE} P \text{ THEN } S \text{ END} \). This is what we do from now on for all \( wp \) formulas we use.

**Lemma 1.** If init is the Initialisation clause of B machine \( M \), then:

\[
\forall tr \in \text{traces}(M) \bullet (wp(\text{init}; tr, \text{true}) \iff tr \notin D[M])
\]

**Proof.** This lemma is a result of Definition 5.

1. \( wp(\text{init}; tr, \text{false}) \iff tr \in D[M] \) \quad \text{Definition 5}

2. \( \neg wp(\text{init}; tr, \text{true}) \iff tr \in D[M] \) \quad 1, Definition 2

3. \( wp(\text{init}; tr, \text{true}) \iff tr \notin D[M] \) \quad 2

\[\square\]

**Lemma 2.** The execution of guarded operations can never cause any divergence in a B machine if there is no pre statement in their body.

**Proof.** We assume \( tr \) is a trace of B machine \( M \) and it is not a divergence. We also assume that \( op = \text{SELECT } G \text{ THEN } R \text{ END} \) is a guarded operation in \( M \). Now we prove that \( tr \upharpoonright (op) \) also is not a divergence of \( M \).
3.1. CSP SEMANTICS OF B MACHINES

1. \( \text{wp}(\text{init}; \text{tr}, \text{true}) \)
   Initial assumption, Lemma 1

2. \( \text{wp}(\text{SELECT } \text{G} \text{ THEN } \text{R END}, \text{true}) = (G \Rightarrow \text{true}) \quad \text{wp}(\text{R}, \text{true}) = \text{true} \)

3. \( \text{wp}(\text{init}; \text{tr}; \text{SELECT } \text{G} \text{ THEN } \text{R END}, \text{true}) = \quad \text{wp}(\text{init}; \text{tr}, G \Rightarrow \text{true}) \)

4. \( \text{wp}(\text{init}; \text{tr}; \text{SELECT } \text{G} \text{ THEN } \text{R END}, \text{true}) = \quad \text{wp}(\text{init}; \text{tr}, \text{true}) \)

5. \( \text{wp}(\text{init}; \text{tr}; \text{SELECT } \text{G} \text{ THEN } \text{R END}, \text{true}) = 4, 1 \)

6. \( \text{wp}(\text{init}; \text{tr}; \text{tr} \not\subseteq \langle \text{op} \rangle, \text{true}) \)

7. \( \text{tr} \not\subseteq \langle \text{op} \rangle \notin \mathcal{D}[M] \)

Lemma 3. If there is no pre statement in the body of pre-conditioned operations in a B machine, then divergence happens in this B machine when a pre-conditioned operation is called outside its precondition.

Proof. We assume \( \text{tr} \) is a trace of B machine \( M \) and it is not a divergence. We also assume that \( \text{op} = \text{PRE } P \text{ THEN } S \text{ END} \) is a pre-conditioned operation in \( M \). Now we prove that \( \text{tr} \not\subseteq \langle \text{op} \rangle \) is a divergence of \( M \) if \( P \) does not hold when the operation \( \text{op} \) is called.

1. \( \text{wp}(\text{PRE } P \text{ THEN } S \text{ END}, \text{true}) = P \land \text{true} \quad \text{wp}(S, \text{true}) = \text{true} \)

2. \( \text{wp}(\text{PRE } P \text{ THEN } S \text{ END}, \text{true}) = P \)

3. \( \text{wp}(\text{init}; \text{tr}; \text{PRE } P \text{ THEN } S \text{ END}, \text{true}) = \quad \text{wp}(\text{init}; \text{tr}, P) \)

4. \( \text{wp}(\text{init}; \text{tr}; \text{PRE } P \text{ THEN } S \text{ END}, \text{true}) \Rightarrow \quad \text{Lemma 1} \)

   \( \text{tr} \not\subseteq \langle \text{op} \rangle \notin \mathcal{D}[M] \)

5. \( \text{wp}(\text{init}; \text{tr}, P) \Leftrightarrow \text{tr} \not\subseteq \langle \text{op} \rangle \notin \mathcal{D}[M] \)

6. \( \neg \text{wp}(\text{init}; \text{tr}, P) \Leftrightarrow \text{tr} \not\subseteq \langle \text{op} \rangle \in \mathcal{D}[M] \)

\( \square \)
This result shows that if $P$ is not guaranteed when the operation $op$ is called, then divergence happens in the B machine. Thus, divergence happens in $M$ when an operation is called outside its pre-condition.

\[ \square \]

### 3.2 Mobile CSP $|\parallel| B$

In standard $CSP |\parallel| B$, a controlled component consists of a CSP controller $P$ in parallel with a B machine $M$. Operations $op$ with inputs $s$ and outputs $t$ are declared in machines $M$ as $t \leftarrow op(s)$. In the combination they are treated as CSP channels $op.s.t$. Standard $CSP |\parallel| B$ has a static architecture in which one B machine works in parallel with only one CSP controller and each CSP controller can be the controller of only one B machine. So, the behaviour of the parallel combination is predictable as we have fixed controlled components during the system execution.

In Mobile $CSP |\parallel| B$, we intend to create a mobile architecture in which B machines are able to be transferred from one controller to another controller and each controller can work with more than one B machine at the same time. As controllers can exchange B machines between each other, B machines can have different controllers during their execution.

For each B machine in the system, we introduce a unique machine channel called machine reference. CSP controllers use machine references as the link to interact with B machines. A machine reference is the only channel through which a CSP controller and a B machine can communicate with each other. As a result, a controller is only able to work with a machine if it owns that machine’s reference. In other words, possession of a machine means having that machine’s reference. This makes it clear that in order for machines to be exchanged between controllers, machine references must be passed around between controllers in the system. Therefore, when a machine is going to be passed from one controller to another, the sender controller passes that B machine’s reference to the other controller, as illustrated on the right in Figure 3.1. It shows that B machine $M_1$ is passed from CSP controller $P_1$ to $P_2$. The figure also shows the difference between Static $CSP |\parallel| B$ architecture and Mobile $CSP |\parallel| B$ architecture.

B machine $M$ with machine reference $z$ is presented in the system as $z : M$. Each variable $n$ in $M$ is replaced with $z.n$. All operations $op$ in $M$ are replaced with $z.op$. So, operation calls of the machine $z : M$ correspond to the communication $z.op.s.t$, and the machine reference $z$ can itself be passed between controllers.
In order to pass machine references between controllers, we introduce special channels called *control points* on which machine references are passed around. When a machine is passed from one controller to another, the sender controller passes that B machine’s reference to the other controller through the control point channel which exists between those two controllers.

We require that only one CSP controller is in possession of \( z \) at any one time, so that when \( z \) is passed from \( P_1 \) to \( P_2 \) then \( P_1 \) is no longer able to use \( z \) to call the operations of the machine. This will be the cornerstone for reasoning about the action of controllers on a mobile machine: that a controller has exclusive control over a machine it is using, and other controllers cannot interfere with its use of the machine.

We introduce \( MR \) as the set of machine references, \( CP \) as the set of control points, and \( C \) as the set of regular CSP channels. Each channel \( c \) in the set of regular channels \( C \) has a *type* denoted \( \text{type}(c) \). The type of channels in \( CP \) (control points, which pass machine references) is \( MR \): in other words, they pass values from the set \( MR \). Each machine reference in \( MR \) is associated with a particular B machine. The type of a machine reference \( z \) is the set of operations (with inputs and outputs) of the unique machine \( M \) that is associated with \( z \).

Each control point can carry one specific type of B machines. If a system contains only one type of B machines, then we introduce one set \( MR \) as the set of machine references. If a system contains different types of B machines, then we introduce different sets \( MRs \), each for one particular type of B machines. In this situation, the type of control point channels in \( CP \) may be different from each other according to the type of the machines they can carry. For instance, if a system contains two different types of B machines, then we introduce \( MR_1 \) as the set of machine references for one type of machines and \( MR_2 \) as the set of machine references for the other type of machines. Therefore, the type of each control point in this system can be either \( MR_1 \) or \( MR_2 \). As a result, if two types of machines should be passed from one controller \( P_1 \) to another controller \( P_2 \) in the system, two control points should be introduced, one for passing machine
references of $MR_1$ and the other one for passing machine references of $MR_2$ from $P_1$ to $P_2$. In this chapter, we use the general form MR as the type of control points in our system. This can be then changed to the type of each control point if there are different sets of machine references in the system.

### 3.3 Developing B machines

In Mobile CSP $| B$, B machines can be specified as the normal B machine specification. However, as explained before in Section 3.1, a B machine is used as an action system. So, the initialisation and the body of its operations must satisfy Dijkstra’s Law of the Excluded Miracle.

In addition, we are going to provide consistency verification of our framework later in this chapter. In our divergence freedom verification, we are going to use Lemma 2 and Lemma 3. Therefore, the B machines in our system must be specified in such a way to use these two lemmas for verification. Also, in our divergence freedom proof strategy, we will use $wp$ formulae for operations in the body of CSP controllers. As it is not possible to predict all the machines a controller is going to work with during the execution, operations with the same name in different machines must have an identical specification.

According to all explanations above, the rules for B machines are as follows:

1. The body of operations and the initialisation are not allowed to contain $SELECT$ or $ANY$ (or $Let$ which is a special kind of $ANY$) statements.

2. The pre statement is not allowed to appear in the body of operations, either in pre-conditioned or in guarded operations. A pre statement can only be used as the precondition statement of a pre-conditioned operation.

3. If different machines in the system have an operation with the same name, the specification of those operations must be identical in all those B machines.

### 3.4 Developing a CSP Controller

Consider $LOOP$ as a mutual recursive CSP process. A family of processes $N_i$ is used to define $LOOP$. 
3.4. DEVELOPING A CSP CONTROLLER

$$\text{LOOP} \equiv N_1$$

$$N_1 \equiv P_1$$

$$\vdots$$

$$N_n \equiv P_n$$

where each $P_i$ is a CSP process expression in which the possible $N_i$s can subsequently be reached.

The static channels $\chi(\text{LOOP})$ of CSP process $\text{LOOP}$ are given by its communication channels and control points. Any particular control point in the alphabet of $\text{LOOP}$ will be either incoming or outgoing with respect to $\text{LOOP}$, and is not permitted to be both. We identify the incoming control points within $\chi(\text{LOOP})$ as $\chi_i(\text{LOOP})$. The outgoing control points within $\chi(\text{LOOP})$ are denoted $\chi_o(\text{LOOP})$. The communication channels within $\chi(\text{LOOP})$ are denoted $\chi_c(\text{LOOP})$. These three sets are pairwise disjoint, and their union is $\chi(\text{LOOP})$:

$$\chi_i(\text{LOOP}) \cap \chi_o(\text{LOOP}) = \emptyset$$

$$\chi_i(\text{LOOP}) \cap \chi_c(\text{LOOP}) = \emptyset$$

$$\chi_o(\text{LOOP}) \cap \chi_c(\text{LOOP}) = \emptyset$$

$$\chi_i(\text{LOOP}) \cup \chi_o(\text{LOOP}) \cup \chi_c(\text{LOOP}) = \chi(\text{LOOP})$$

The syntax of the processes $P_i$s is defined by the following BNF:

$$P ::= \text{SKIP} | c ? x \rightarrow P(x) | c ! v \rightarrow P | \text{cp}_1 ? w \rightarrow P(w)$$

$$| z \cdot \text{op}_1 ! t \rightarrow P(t) | \text{cp}_2 ! z \rightarrow P (x \notin s(P))$$

$$| P' \square P'' | P' \sqcap P'' | \text{if} \ b \ \text{then} \ P' \ \text{else} \ P'' | e \rightarrow P | \text{N}(E_1, \ldots, E_k)$$

where $b$ is a boolean expression, $e$ is an atomic CSP event which is not a $B$ operation, $c \in \chi_c(\text{LOOP})$, $\text{cp}_1 \in \chi_i(\text{LOOP})$, $\text{cp}_2 \in \chi_o(\text{LOOP})$, $v$ and $z$ are variables of type $\text{type}(c)$, $z$ and $w$ are variables of type $\text{MR}$, $t \leftarrow \text{op}(s)$ is an operation of the $B$ machine associated with $z$, $s(P)$ is the set of machine references owned by $P$ (Definition 8, page 55).

Control points are introduced as the only channels which can carry other channels, machine references. The type of communication channels are not allowed to refer to machine references. In other words, $\text{type}(c) \cap \text{MR} = \emptyset$ and $\text{type}(c) \cap \text{P} \text{MR} = \emptyset$.

In $\text{N}(E_1, \ldots, E_k)$, the type of each expression $E_i$ is either of these below:

1. $\text{type}(E_i) = \text{MR}$

2. $\text{type}(E_i) = \text{P} \text{MR}$

3. $\text{type}(E_i) = A \cdot (A \cap \text{MR} = \emptyset \land A \cap \text{P} \text{MR} = \emptyset)$
Each expression $E_i$ referring to machine references should not be repeated among $E_1, \ldots, E_k$. In addition, each variable referring to machine references should not be repeated in expressions $E_1, \ldots, E_k$. In other words:

1. $\forall 1 \leq i, j \leq k \cdot ((\text{type}(E_i) = \text{type}(E_j) = \text{MR}) \Rightarrow (\{E_i\} \cap \{E_j\} = \emptyset))$
2. $\forall 1 \leq i, j \leq k \cdot ((\text{type}(E_i) = \text{type}(E_j) = \text{IP} \land \text{MR}) \Rightarrow (E_i \cap E_j = \emptyset))$
3. $\forall 1 \leq i, j \leq k \cdot ((\text{type}(E_i) = \text{MR} \land \text{type}(E_j) = \text{IP} \land \text{MR}) \Rightarrow (\{E_i\} \cap E_j = \emptyset))$

Definition 6. The set of free variables in a CSP process is defined as below:

$$
\begin{align*}
fv(SKIP) &= \emptyset \\
fv(c!z \rightarrow P(z)) &= fv(P(z)) - \{z\} \\
fv(c!v \rightarrow P) &= \{v\} \cup fv(P) \\
fv(cp_1?w \rightarrow P(w)) &= fv(P(w)) - \{w\} \\
fv(cp_2!z \rightarrow P) &= \{z\} \cup fv(P) \\
fv(z.op_is?t \rightarrow P(t)) &= \{z\} \cup \{s\} \cup (fv(P(t)) - \{t\}) \\
fv(P' \sqcap P'') &= fv(P') \cup fv(P'') \\
fv(P' \sqcap P'') &= fv(P') \cup fv(P'') \\
fv(if \ b \ then \ P' \ else \ P'') &= var(b) \cup fv(P') \cup fv(P'') \\
fv(e \rightarrow P) &= fv(P) \\
fv(N(E_1, \ldots, E_k)) &= (\bigcup_{i=1}^k \{E_i\} \cdot E_i \text{ is not a set}) \cup (\bigcup_{i=1}^k E_i \cdot E_i \text{ is a set})
\end{align*}
$$

where $\text{var}(b)$ is a set which contains all the variables of predicate $b$.

Sequential processes are then defined recursively as follows:

$$
N_i(y_{i1}, \ldots, y_{ik}) \triangleq P_i \quad (1 \leq i \leq n) \quad \text{where } fv(P_i) \subseteq fv(N_i(y_{i1}, \ldots, y_{ik}))
$$

In valuation, parameters $y_{i1}, \ldots, y_{ik}$ are given a value of their type. The type of each parameter $y_{ig}$ is either of these below:

1. $\text{type}(y_{ig}) = \text{MR}$
2. $\text{type}(y_{ig}) = \text{IP} \land \text{MR}$
3. $\text{type}(y_{ig}) = A \cdot (A \cap \text{MR} = \emptyset \land A \cap \text{IP} = \emptyset)$

In $N_i(y_{i1}, \ldots, y_{ik})$, each parameter $y_{ig}$ referring to machine references should not be repeated among $y_{i1}, \ldots, y_{ik}$. In addition, each variable referring to machine references should not be repeated in parameters $y_{i1}, \ldots, y_{ik}$. In other words:
3.5 Parallel combination

A mobile combined communicating system including $n$ controllers and $m$ machines is represented as:

$$LOOP_1 \parallel LOOP_2 \parallel \ldots \parallel LOOP_n \parallel z_1 : M_1 \parallel z_2 : M_2 \parallel \ldots \parallel z_m : M_m$$
where $z_1, z_2, \ldots, z_m$ are the machine references for B machines $M_1, M_2, \ldots, M_m$ respectively, and $\forall 1 \leq i, j \leq m \cdot (i \neq j \Rightarrow z_i \neq z_j)$.

Mutual recursive CSP processes can be composed in parallel, provided they have no machine references in common. They must also differ on their incoming control points and their outgoing control points. In addition, each control point in the system must have both a sender and a receiver. In other words, any outgoing (or incoming) control point in one controller must be an incoming (or outgoing) control point of one of the other controllers in the system.

According to all explanations above, the rules for parallel combination of controllers are as follows:

1. $\forall 1 \leq i, j \leq n \cdot s(LOOP_i) \cap s(LOOP_j) = \emptyset$
2. $\forall 1 \leq j, k \leq n \cdot \chi_i(LOOP_j) \cap \chi_i(LOOP_k) = \emptyset$
3. $\forall 1 \leq i, j \leq n \cdot \chi_o(LOOP_i) \cap \chi_o(LOOP_j) = \emptyset$
4. $\bigcup_{j=1}^n \chi_i(LOOP_j) = \bigcup_{i=1}^n \chi_o(LOOP_i)$

The free variables of the parallel combination of controllers is given as follows:

$$fv(LOOP_1 || LOOP_2 || \ldots || LOOP_n) = \bigcup_{i=1}^n fv(LOOP_i)$$

When a system is constructed, each variable referring to machine references must be given a different concrete value.

The incoming control points, outgoing control points and communication channels for the parallel combination of controllers are given as follows:

$$\chi_i(LOOP_1 || LOOP_2 || \ldots || LOOP_n) = \bigcup_{j=1}^n \chi_i(LOOP_j)$$

$$\chi_o(LOOP_1 || LOOP_2 || \ldots || LOOP_n) = \bigcup_{i=1}^n \chi_o(LOOP_i)$$

$$\chi_c(LOOP_1 || LOOP_2 || \ldots || LOOP_n) = \bigcup_{i=1}^n \chi_c(LOOP_i)$$

Note that incoming and outgoing control points need not be disjoint in parallel combinations: a control point that is both incoming and outgoing has both ends within the parallel combination and hence connects two of the controllers. The rules 2 and 3 for parallel combination of controllers ensures that no further controllers will use that control point.

The language of process terms and the rules for parallel combination of controllers have been designed to ensure that at any point in the system execution, only one
controller has possession of any machine reference. Controllers do not share any
machine references to begin with, and when a machine reference is passed along
a control point to another controller, it is not retained by the sending controller.

In order to define the traces of parallel composition, it is necessary to keep track
of the machine references as they are used and passed between controllers. We
can define the projection of a trace onto a particular controller \( \text{LOOP} \) given the
channels \( \chi_i(\text{LOOP}) \), \( \chi_o(\text{LOOP}) \), \( \chi_c(\text{LOOP}) \), provided we also know the set
of machine references \( s \) owned by the controller.

The projection of a trace \( tr \) onto \( \chi(\text{LOOP}) \) and a set of machine references \( s \) can
be defined inductively as follows where \( tr \upharpoonright \chi(\text{LOOP}), s \) means the projection of
\( tr \) onto alphabet of controller \( \text{LOOP} \) who owns the set of machine references \( s \) :

\[
\begin{align*}
() \upharpoonright \chi(\text{LOOP}), s & = () \\
((cp.z) \upharpoonright tr) \upharpoonright \chi(\text{LOOP}), s & = \begin{cases} 
(c.p.z) \upharpoonright (tr \upharpoonright \chi(\text{LOOP}), s \cup \{z\}) & \text{if } cp \in \chi_i(\text{LOOP}) \land z \notin s \\
(c.p.z) \upharpoonright (tr \upharpoonright \chi(\text{LOOP}), s - \{z\}) & \text{if } cp \in \chi_o(\text{LOOP}) \land z \in s \\
tr \upharpoonright \chi(\text{LOOP}), s & \text{if } cp \notin \chi_i(\text{LOOP}) \cup \chi_o(\text{LOOP}) \land z \notin s \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
((c) \upharpoonright tr) \upharpoonright \chi(\text{LOOP}), s & = \begin{cases} 
(c) \upharpoonright (tr \upharpoonright \chi(\text{LOOP}), s) & \text{if } c \in \chi_c(\text{LOOP}) \\
tr \upharpoonright \chi(\text{LOOP}), s & \text{if } c \notin \chi_c(\text{LOOP}) \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
((z.op) \upharpoonright tr) \upharpoonright \chi(\text{LOOP}), s & = \begin{cases} 
(z.op) \upharpoonright (tr \upharpoonright \chi(\text{LOOP}), s) & \text{if } z \in s \\
tr \upharpoonright \chi(\text{LOOP}), s & \text{if } z \notin s \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}
\]

This enables a definition of the traces of the parallel combination of controllers
to be given:

\[
\text{traces}(\text{LOOP}_1 || \ldots || \text{LOOP}_n) = \{ tr \mid \forall 1 \leq i \leq n \bullet tr \upharpoonright \chi(\text{LOOP}_i), s(\text{LOOP}_i) \in \text{traces}(\text{LOOP}_i) \}
\]

**Example 2.** Assume a mobile combined communicating system, \( \text{SYS} \), containing
two controllers \( \text{LOOP}_1 \) and \( \text{LOOP}_2 \) and one \( \text{B} \) machine \( \text{M} \) with machine reference
\( \text{mc} \) as shown in Figure 3.2. \( cp \) is a control point such that \( cp \in \chi_o(\text{LOOP}_1) \) and
MACHINE M
VARIABLES n
INVARIANT n ∈ 0..1
INITIALISATION n := 0

OPERATIONS
up =
PRE n = 0
THEN n := 1
END;
down =
PRE n = 1
THEN n := 0
END

Figure 3.2: An example of a system containing two controllers and one machine

$$LOOP_1(x) = P_1(x)$$
$$P_1(x) = x.up \rightarrow cp!x \rightarrow P_2$$
$$P_2 = dp?w \rightarrow P_1(w)$$

$$LOOP_2 = Q_1$$
$$Q_1 = cp!x \rightarrow x.down \rightarrow Q_2(x)$$
$$Q_2(x) = dp!x \rightarrow Q_1$$

$$SYS = LOOP_1(mc) || LOOP_2 || mc : M$$

**Figure 3.3:** Transfer of machine M through control point cp

$$cp \in x_1(LOOP_2)$$. Whenever these two controllers synchronise on cp, LOOP_1 gives the machine reference to LOOP_2 through cp which means that the machine is transferred from LOOP_1 to LOOP_2. This is shown in Figure 3.3. The figure illustrates that the machine whose reference is mc is passed from LOOP_1 to LOOP_2 when the event cp.mc happens in the system during the execution. In this system, $$tr = \langle mc.up, cp.mc, mc.down \rangle$$ is a trace of the parallel combination of controllers, $$LOOP_1(mc) || LOOP_2$$, and the trace of each controller is as below:

$$tr \upharpoonright x(LOOP_1(mc)), \{mc\} = \langle mc.up, cp.mc \rangle$$

$$tr \upharpoonright x(LOOP_2), \emptyset = \langle cp.mc, mc.down \rangle$$

Definition of the failures of the parallel combination of CSP processes in Mobile CSP || B is given as below:
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

Definition 9. If \( P_1, \ldots, P_n \) are CSP processes, then

\[
\text{failures}(P_1 \parallel \ldots \parallel P_n) = \\
\{(tr, X) \mid \forall 1 \leq i \leq n \bullet (tr \upharpoonright \chi(P_i), s(P_i)), X_i) \in \text{failures}(P_i) \land \bigcup_{i=1}^{n} X_i = X\}
\]

3.6 Divergence freedom of a controller and its machines

In this section we present theorems and techniques for establishing divergence freedom of the parallel combination between a CSP controller, \( LOOP \), and the machines it works with during the execution. The machines that \( LOOP \) works with are the machines it owns initially and the machines it receives from other controllers during the execution.

If \( LOOP \) is not divergence free, then the parallel combination between \( LOOP \) and any machine might have divergence. Therefore, the first step is to establish that \( LOOP \) is divergence free.

If \( LOOP \) is divergence free, then any divergence in the parallel combination between \( LOOP \) and a machine can only arise from that machine. Thus, the divergence freedom verification strategy should then focus on the behaviour of the machines while working in parallel with \( LOOP \). According to Lemma 2, Lemma 3 and the second rule presented in Section 3.3 for B machines, a machine will have divergence when it contains at least one pre-conditioned operation and its pre-conditioned operations are called outside their precondition by \( LOOP \). As a result, we should establish that \( LOOP \) always calls the operations of all the machines it works with during the execution within their precondition.

The key point is that we are allowing B machines to be passed from one controller to another. The divergence freedom verification technique developed for the earlier static case uses the fact that a B machine is completely controlled by a single controller and so the state of the machine at any particular point is only dependent on what that controller has done. This made proof easier because the machine was bound to a particular controller. However, with mobile machines this is no longer the case. A controller typically receives a machine from another controller without knowing its state in advance, and so the divergence freedom between a machine and a controller needs to take the combined behaviour of the controllers into account.

In order to keep proofs manageable we want to develop ways of considering the contribution of each controller separately. We need to check the state of the machines while passing from one controller to another as when a B machine is passed from one controller to another, the target controller does not have any control over the current state of the machine. Therefore, we need to ensure that a machine is always transferred to another controller in a correct state where its
operations are called appropriately. In order to achieve this, for each control point we assign an assertion on the state of the machine whose reference is passed along that control point. The intention is that whenever a machine reference is passed to a CSP controller along a control point, it is guaranteed that the assertion is satisfied.

The notation $\text{assert}(cp_z)$ denotes the assertion of the control point $cp$ for a machine whose reference is $z$. For instance, $\text{assert}(cp_z) : z.n = 0$ means that the variable $n$ of the machine with machine reference $z$ must be zero when passing through $cp$. For each control point, one assertion is assigned which is used for all machines being passed through it. This means that for a machine with machine reference $w$, the assertion of $cp$ is $\text{assert}(cp_z)$ with $w$ substituted for $z$ which is $w.n = 0$.

In order to establish that the parallel combination between $\text{LOOP}$ and any machine it works with during the execution is divergence free, we translate the body of CSP processes $N_i$ into a sequence of AMN (page 8) statements. Each process $N_i$ is defined in terms of a process expression $P_i$ as stated before. We translate each process expression $P_i$ into a sequence of AMN statements. So, we define an AMN construct, $\text{trans}(P_i)$, to be the translation of the body of each process $N_i$.

**Definition 10.** The translation of each CSP expression into an AMN statement is defined as below:

- $\text{trans}(\text{SKIP}) = \text{SELECT} \ false \ \text{THEN} \ \text{skip} \ \text{END}$
- $\text{trans}(c?!x \rightarrow P(x)) = \text{ANY} \ x \ \text{WHERE} \ x : \text{type}(c) \ \text{THEN} \ \text{trans}(P(x)) \ \text{END}$
- $\text{trans}(c!v \rightarrow P) = \text{PRE} \ v : \text{type}(c) \ \text{THEN} \ \text{skip} \ \text{END}; \ \text{trans}(P)$
- $\text{trans}(cp?!z \rightarrow P(x)) = \text{ANY} \ z \ \text{WHERE} \ z : \text{MR} \ \text{THEN}
  \text{SELECT} \ \text{assert}(cp_z) \ \text{THEN} \ \text{trans}(P(z)) \ \text{END}$
- $\text{trans}(cp!z \rightarrow P) = \text{PRE} \ \text{assert}(cp_z) \ \text{THEN} \ \text{skip} \ \text{END}; \ \text{trans}(P)$
- $\text{trans}(x.opl?t \rightarrow P(t)) = t \leftarrow x.op(s); \ \text{trans}(P(t))$
- $\text{trans}(P' \sqcap P'') = \text{CHOICE} \ \text{trans}(P') \ \text{OR} \ \text{trans}(P'') \ \text{END}$
- $\text{trans}(P' \sqcup P'') = \text{CHOICE} \ \text{trans}(P') \ \text{OR} \ \text{trans}(P'') \ \text{END}$
- $\text{trans}(\text{if} \ b \ \text{then} \ P' \ \text{else} \ P'') = \text{IF} \ b \ \text{THEN} \ \text{trans}(P') \ \text{ELSE} \ \text{trans}(P'') \ \text{END}$
- $\text{trans}(e \rightarrow P) = \text{skip}; \ \text{trans}(P)$
- $\text{trans}(N(E_1, \ldots, E_k)) = \text{rec} := N(E_1, \ldots, E_k)$

The last clause introduces a program counter $\text{rec}$ to handle recursive calls. Observe that inputs $c?!z$ and $cp?!z$ are translated to the ANY statement, which models an assumption that the value being received is of the correct type. In the case of a machine reference, $cp?!z$ also contains a SELECT statement, which models an extra assumption that the machine is in a state satisfying $\text{assert}(cp_z)$. 


3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

Outputs clv and cplz are translated to PRE statement rather than ANY and SELECT statements. v in clv and z in cplz are the parameters of the process so there are already some predicates on their value before this stage. Therefore, there is no need to have ANY in the translation. Instead, we use the PRE statement, which models a guarantee that the condition is met on output values. In the case of a machine reference, there is also no SELECT statement in the translation of cplz. This is because we are going to use weakest precondition formulae in our consistency verification strategy and the PRE statement is the suitable statement in order to detect when the assertion is not ensured by the sender process, which corresponds to divergence in the system.

Note that the translation of external and internal choices are the same. The reason is that we want to check that if each of the branches $P'$ or $P''$ happens, the system is divergence free. So we do not care if the branches happened with our choice or without our control. We are just concerned that there are two possible branches which either can happen.

**Example 3.** Assuming the system of Figure 3.2. If assert(cpz) : $z.n = 1$ and assert(dpw) : $w.n = 0$, then the translation of $P_1(z)$ and $P_2$ are as follows:

\[
\text{trans}(P_1(z)) = \begin{cases} 
\text{PRE } z.n = 0 \text{ THEN } z.n := 1 \text{ END; PRE } z.n = 1 \text{ THEN skip END;} \\
rec := P_2
\end{cases}
\]

\[
\text{trans}(P_2) = \begin{cases} 
\text{ANY } w \text{ WHERE } w : \text{MR THEN} \\
\text{SELECT } w.n = 0 \text{ THEN rec := } P_1(w) \text{ END} \\
\text{END}
\end{cases}
\]

3.6.1 Conditions for divergence freedom

Supposing for a process $N_i$ in LOOP we can find an invariant referring to all free variables in $N_i$ such that if this invariant is true then whenever $N_i$ is called to be executed, it calls the operations of all the machines it owns at the beginning of that recursive call through their precondition. If we can establish that this invariant holds at every recursive call of $N_i$, and if the state of the machines $N_i$ receives always satisfy the related assertions, then $N_i$ always calls the operations of all the machines it works with through their precondition at all the time during the execution.

If we can establish the conditions above for each process $N_i$ $(1 \leq i \leq n)$ in LOOP, then the parallel combination between LOOP and any machine it works with during the execution is ensured to be divergence free. As this invariant should be true at each recursive call, we call it Control Loop Invariant, CLI.

We now present the definition below for LOOP which contains the conditions we explained above:
Definition 11. LOOP is called CLI preserver if for each \( N_i \) \((1 \leq i \leq n)\) in LOOP, a Control Loop Invariant, CL\( I_i \), can be found such that:

1. \([\text{init}_1; \text{init}_2; \ldots; \text{init}_m; \text{rec} := N_1](\text{CL}_1)\)

2. \(\forall 1 \leq i \leq n \cdot ((\text{rec} = N_i \land \text{CL}_i) \Rightarrow \left[\text{trans}(P_i)\right](\forall 1 \leq j \leq n \cdot (\text{rec} = N_j \Rightarrow \text{CL}_j))\)

where \(M_1, \ldots, M_m\) are the machines that LOOP owns at the beginning of the execution and \(\text{init}_1, \ldots, \text{init}_m\) are the Initialisation clause of machines \(M_1, \ldots, M_m\) respectively.

Example 4. In continuation of Example 3, LOOP\(_1\) in Figure 3.2 is CLI preserver as we can introduce CL\(_{P_1(z)}\) : \(z.n = 0\) and CL\(_{P_2}\) : true such that the two conditions in Definition 11 hold:

Condition 1:
\[\text{[trans}(P_1(z))][\text{rec} = P_2 \Rightarrow \text{CL}_{P_1(z)}] = z.n = 0\]
\(\text{[trans}(P_1(z))][\text{rec} = P_1(z) \land \text{CL}_{P_1(z)}] = (\text{rec} = P_1(z) \land z.n = 0)\)
\(\text{[trans}(P_1(z))][\text{rec} = P_1(z) \land z.n = 0] \Rightarrow z.n = 0 \checkmark\)

Condition 2:
\[\forall w \cdot (w \in MR \Rightarrow [\text{SELECT } w.n = 0 \text{ THEN } \text{rec} := P_1(w) \text{ END}] \Rightarrow \text{rec} = P_1(w) \Rightarrow \text{CL}_{P_1(w)})\]
\[\text{[trans}(P_2)][\text{rec} = P_1(w) \Rightarrow \text{CL}_{P_1(w)}] = \forall w \cdot (w \in MR \Rightarrow \text{true}) = \text{true}\]
\(\text{[trans}(P_2)][\text{rec} = P_1(w) \Rightarrow \text{CL}_{P_1(w)}] = (\text{rec} = P_2 \land \text{true}) \Rightarrow \text{true} \checkmark\)

According to all explanations above, we now provide the following theorem which demonstrates all the conditions needed for establishing divergence freedom of the parallel combination between LOOP and the machines it works with during the execution.
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

Theorem 1. If LOOP is a divergence free CSP controller and it is CLI preserver, then:

1. The parallel combination between LOOP and any machine it owns initially is divergence free.

2. The parallel combination between LOOP and any machine it receives during the execution with the correct state (according to the assertion of that control point) is divergence free provided the machine does not have divergence until the point it is being passed to LOOP.

3.6.2 Verification of divergence freedom

In this section we present the following definitions, lemmas and corollaries as the steps towards proving Theorem 1. The results achieved in this section can be then used for the verification of the whole mobile combined communicating system.

For each CSP process $P_i$ in LOOP, we define a special process $TP_i$. The body of $TP_i$ is the same as $P_i$ but the call to the indexed processes in $P_i$ is replaced by $update \rightarrow SKIP$. For instance, if in the body of $P_i$ there is a call to the indexed process $N_j$, $N_j$ is replaced by $update_{N_j} \rightarrow SKIP$ in $TP_i$. In other words, process $TP_i$ is the terminating version of the process $P_i$. $SKIP$ in the body of $P_i$ remains the same and will not be changed in $TP_i$.

We introduce a function, called term, which produces the terminating version of a process. It defines termination for recursive calls and distributes over all other operators. In other words, term is a function which produces the process $TP_i$ for each process $P_i$. Thus, $TP_i = term(P_i)$.

Definition 12. The terminating version of a CSP controller is defined as below:

\[
\begin{align*}
term(SKIP) &= SKIP \\
term(c?x \rightarrow P(x)) &= c?x \rightarrow term(P(x)) \\
term(clv \rightarrow P) &= clv \rightarrow term(P) \\
term(cp1?w \rightarrow P(w)) &= cp1?w \rightarrow term(P(w)) \\
term(cp2lz \rightarrow P) &= cp2lz \rightarrow term(P) \\
term(x. opl? t \rightarrow P(t)) &= x. opl? t \rightarrow term(P(t)) \\
term(P' \sqcap P'') &= term(P') \sqcap term(P'') \\
term(P' \sqcup P'') &= term(P') \sqcup term(P'') \\
term(if b then P' else P'') &= if b then term(P') else term(P'') \\
term(e \rightarrow P) &= e \rightarrow term(P) \\
term(N(E_1,\ldots,E_k)) &= update_{N(E_1,\ldots,E_k)} \rightarrow SKIP
\end{align*}
\]
Whenever a CSP controller is working with a B machine, the sequence of that machine's operations which are executed is always a subsequence of the events performed by that controller during this parallel combination. This is the controller who controls the execution order of the operations of the machine while they are working with each other in parallel. As we described before, the divergences of a B machine are defined in [50] in terms of the weakest precondition formulae for machine's traces. In this section, we introduce the weakest precondition semantics for traces of a CSP process. We then use these semantics in our proof strategy for divergence freedom verification of machines while working with a CSP controller in parallel.

**Definition 13.** The weakest precondition semantics for traces of a CSP process $P$ are as follows:

\[
\begin{align*}
wp(\emptyset, Q) &= Q \\
wp((c,x) \leadsto tr, Q) &= wp(tr, Q) \\
wp((x.op.s) \leadsto tr, Q) &= wp(t \leftarrow x.op(s), wp(tr, Q)) \\
wp(\{e\} \leadsto tr, Q) &= wp(tr, Q) \\
wp((update_{N(E_1, \ldots, E_n)} \leadsto tr, Q) &= wp(rec := N(E_1, \ldots, E_n), wp(tr, Q)) \\
wp((c.p.z) \leadsto tr, Q) &= \begin{cases} 
assert(cp_z) \land wp(tr, Q) & \text{if } cp \in \chi_{c}(P) \\
assert(cp_z) \Rightarrow wp(tr, Q) & \text{if } cp \in \chi_{c}(P)
\end{cases}
\end{align*}
\]

**Lemma 4.** If $tr \leadsto (\forall) \in traces(TP_i)$ and $\exists 1 \leq j \leq n \cdot \text{foot}(tr) = \text{update}_{N_j}$ then $wp(\text{trans}(P_i), Q) \Rightarrow wp(tr, Q)$ where $1 \leq i \leq n$.

**Proof.** We prove this lemma by using the Principle of Induction on $P_i$.

Base case 1: $P_i = SKIP$

1. $P_i = SKIP$ Initial assumption
2. $TP_i = SKIP$ 1, Definition 12
3. $tr = \emptyset$ 2, $tr \leadsto (\forall) \in traces(TP_i)$
4. $\#1 \leq j \leq n \cdot \text{foot}(\emptyset) = \text{update}_{N_j}$ 3
5. The condition part does not hold 3, 4
6. Lemma is true 5
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

Base case 2: \( P_i = N(E_1, \ldots, E_k) \)

1. \( P_i = N(E_1, \ldots, E_k) \)  
   Initial assumption

2. \( TP_i = \text{update}_{N(E_1, \ldots, E_k)} \rightarrow \text{SKIP} \)  
   1, Definition 12

3. \( tr = (\text{update}_{N(E_1, \ldots, E_k)}) \)  
   2, \( tr \wedge (\exists x) \in \text{traces}(TP_i) \wedge \ootnote{foot (tr) = \text{update}_{N(E_1, \ldots, E_k)}}

4. \( wp(tr, Q) = wp(\text{rec} := N(E_1, \ldots, E_k), Q) \)  
   3, Definition 13

5. \( \text{trans}(P_i) = \text{rec} := N(E_1, \ldots, E_k) \)  
   1, Definition 10

6. \( wp(\text{trans}(P_i), Q) = wp(\text{rec} := N(E_1, \ldots, E_k), Q) \)  
   5

7. \( wp(\text{trans}(P_i), Q) = wp(tr, Q) \)  
   6, 4

8. \( wp(\text{trans}(P_i), Q) \Rightarrow wp(tr, Q) \)  
   7

Inductive case 1: \( e \rightarrow P' \)

Assume Inductive Hypothesis for \( P' \).

1. \( \text{term}(e \rightarrow P') = e \rightarrow TP' \)  
   Definition 12

2. \( tr \wedge (\exists x) \in \text{traces}(e \rightarrow TP') \wedge \ootnote{\exists 1 \leq j \leq n \bullet \text{foot}(tr) = \text{update}_{N_i}} \)

3. \( tr = (e) \wedge tr' \in \text{traces}(TP') \)  
   \( tr \in \text{traces}(e \rightarrow TP') \)

4. \( tr' \wedge (\exists x) \in \text{traces}(TP') \wedge \ootnote{\exists 1 \leq j \leq n \bullet \text{foot}(tr') = \text{update}_{N_i}} \)

5. \( wp(\text{trans}(P'), Q) \Rightarrow wp(tr', Q) \)  
   4, Induction assumption

6. \( wp(tr, Q) = wp((e) \wedge tr', Q) \)  
   3

7. \( wp((e) \wedge tr', Q) = wp(tr', Q) \)  
   Definition 13
8 \( \text{wp}(tr, Q) = \text{wp}(tr', Q) \)  

9 \( \text{trans}(e \rightarrow P') = \text{skip}; \text{trans}(P') \)  

10 \( \text{wp}(\text{trans}(e \rightarrow P'), Q) = \)  
\( \text{wp}(\text{skip}; \text{trans}(P'), Q) \)  

11 \( \text{wp}(\text{trans}(e \rightarrow P'), Q) = \)  
\( \text{wp}(\text{skip}, \text{wp}(\text{trans}(P'), Q)) \)  

12 \( \text{wp}(\text{trans}(e \rightarrow P'), Q) = \text{wp}(\text{trans}(P'), Q) \)  

13 \( \text{wp}(\text{trans}(e \rightarrow P'), Q) \Rightarrow \text{wp}(tr', Q) \)  

14 \( \text{wp}(\text{trans}(e \rightarrow P'), Q) \Rightarrow \text{wp}(tr, Q) \)  

Inductive case 2: \( c!v \rightarrow P' \)

Assume Inductive Hypothesis for \( P' \).

1 \( \text{term}(c!v \rightarrow P') = c!v \rightarrow TP' \)  

2 \( tr \cap (\forall) \in \text{traces}(c!v \rightarrow TP') \wedge \)  
\( \exists 1 \leq j \leq n \rightarrow \text{foot}(tr) = \text{update}_{N_j} \)  

3 \( tr = (c.v) \cap tr' \rightarrow tr' \in \text{traces}(TP') \)  

4 \( tr' \cap (\forall) \in \text{traces}(TP') \wedge \)  
\( \exists 1 \leq j \leq n \rightarrow \text{foot}(tr') = \text{update}_{N_j} \)  

5 \( \text{wp}(\text{trans}(P'), Q) \Rightarrow \text{wp}(tr', Q) \)  

6 \( \text{wp}(tr, Q) = \text{wp}((c.v) \cap tr', Q) \)  

7 \( \text{wp}((c.v) \cap tr', Q) = \text{wp}(tr', Q) \)  

8 \( \text{wp}(tr, Q) = \text{wp}(tr', Q) \)  

9 \( \text{trans}(c!v \rightarrow P') = \)  
\( \text{PRE} v : \text{type}(c) \text{ THEN } \text{skip END}; \text{trans}(P') \)
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

10 \( \text{wp}(\text{trans}(c!v \rightarrow P'), Q) = \)
\varepsilon: \text{type}(c) \land \text{wp}(\text{trans}(P'), Q)

11 \( \text{wp}(\text{trans}(c!v \rightarrow P'), Q) \Rightarrow \text{wp}(\text{trans}(P'), Q) \)

12 \( \text{wp}(\text{trans}(c!v \rightarrow P'), Q) \Rightarrow \text{wp}(tr', Q) \)

13 \( \text{wp}(\text{trans}(c!v \rightarrow P'), Q) \Rightarrow \text{wp}(tr, Q) \)

Inductive case 3 : \( cp!z \rightarrow P' \)
Assume Inductive Hypothesis for \( P' \).

1 \( \text{term}(\text{cp!z} \rightarrow P') = \text{cp!z} \rightarrow TP' \) \hspace{1cm} Definition 12

2 \( tr \wedge \langle \forall \rangle \in \text{traces}(\text{cp!z} \rightarrow TP') \land
\exists 1 \leq j \leq n \cdot \text{foot}(tr) = \text{update}_N_j \) \hspace{1cm} Initial assumption

3 \( tr = (\text{cp}.z) \wedge tr' \cdot tr' \in \text{traces}(TP') \) \hspace{1cm} \text{tr} \in \text{traces}(\text{cp!z} \rightarrow TP')

4 \( tr' \wedge \langle \forall \rangle \in \text{traces}(TP') \land
\exists 1 \leq j \leq n \cdot \text{foot}(tr') = \text{update}_N_j \)

5 \( \text{wp}(\text{trans}(P'), Q) \Rightarrow \text{wp}(tr', Q) \) \hspace{1cm} 4, Induction assumption

6 \( \text{wp}(tr, Q) = \text{wp}((\text{cp}.z) \wedge tr', Q) \)

7 \( \text{wp}((\text{cp}.z) \wedge tr', Q) =
\text{assert}(cp_z) \land \text{wp}(tr', Q) \) \hspace{1cm} Definition 13

8 \( \text{wp}(tr, Q) = \text{assert}(cp_z) \land \text{wp}(tr', Q) \)

9 \( \text{trans}(\text{cp!z} \rightarrow P') =
\text{PRB assert}(cp_z) \text{ THEN skip END; trans}(P') \) \hspace{1cm} Definition 10

10 \( \text{wp}(\text{trans}(\text{cp!z} \rightarrow P'), Q) =
\text{assert}(cp_z) \land \text{wp}(\text{trans}(P'), Q) \)
Inductive case 4: \( z.o.p!s?t \rightarrow P'(t) \)

Assume Inductive Hypothesis for \( P'(t) \).

\[
\begin{align*}
1 & \quad \text{term}(z.o.p!s?t \rightarrow P'(t)) = \quad \text{Definition 12} \\
& \quad z.o.p!s?t \rightarrow T P'(t) \\
2 & \quad tr \cap \{x\} \in \text{traces}(z.o.p!s?t \rightarrow T P'(t)) \land \\
& \quad \exists 1 \leq j \leq n \cdot \text{foot}(tr) = \text{update}_{N_j} \\
3 & \quad \exists tr', x \cdot (tr = \{z.o.p.s.x\} \cap tr' \land \\
& \quad \text{tr'} \in \text{traces}(T P'(x))) \\
4 & \quad tr' \cap \{x\} \in \text{traces}(T P'(x)) \land \\
& \quad \exists 1 \leq j \leq n \cdot \text{foot}(tr') = \text{update}_{N_j} \\
5 & \quad \wp(\text{trans}(P'(x)), Q) \Rightarrow \wp(tr', Q) \quad \text{Induction assumption} \\
6 & \quad \wp(tr, Q) = \wp((z.o.p.s.x) \cap tr', Q) \quad 3 \\
7 & \quad \wp((z.o.p.s.x) \cap tr', Q) = \\
& \quad \wp(x \leftarrow z.o.p(s), \wp(tr', Q)) \quad \text{Definition 13} \\
8 & \quad \wp(tr, Q) = \wp(x \leftarrow z.o.p(s), \wp(tr', Q)) \quad 6, 7 \\
9 & \quad \text{trans}(z.o.p!s?t \rightarrow P'(t)) = \\
& \quad t \leftarrow z.o.p(s); \text{trans}(P'(t)) \quad \text{Definition 10} \\
10 & \quad \wp(\text{trans}(z.o.p!s?t \rightarrow P'(t)), Q) = \\
& \quad \wp(t \leftarrow z.o.p(s); \text{trans}(P'(t)), Q) \quad 9 \\
11 & \quad \wp(\text{trans}(z.o.p!s?t \rightarrow P'(t)), Q) = \\
& \quad \wp(t \leftarrow z.o.p(s), \wp(\text{trans}(P'(t)), Q)) \quad 10
\end{align*}
\]
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

12 \( wp(t \leftarrow z\cdot op(s), wp(trans(P'(t)), Q)) = \) \( wp(x \leftarrow z\cdot op(s), wp(trans(P'(x)), Q)) = \) \( wp(x \leftarrow z\cdot op(s), wp(trans(P'(x)), Q)) \)

13 \( wp(trans(z\cdot op!s?t \rightarrow P'(t)), Q) = \) \( wp(x \leftarrow z\cdot op(s), wp(trans(P'(x)), Q)) \)

14 \( wp(x \leftarrow z\cdot op(s), wp(trans(P'(x)), Q)) \) \( \Rightarrow \) \( wp(x \leftarrow z\cdot op(s), wp(tr', Q)) \)

15 \( wp(trans(z\cdot op!s?t \rightarrow P'(t)), Q) \) \( \Rightarrow \) \( wp(x \leftarrow z\cdot op(s), wp(tr', Q)) \)

16 \( wp(trans(z\cdot op!s?t \rightarrow P'(t)), Q) \) \( \Rightarrow wp(tr, Q) \)

Inductive case 5 : \( c?v \rightarrow P'(v) \)
Assume Inductive Hypothesis for \( P'(v) \).

1 \( \text{term}(c?v \rightarrow P'(v)) = c?v \rightarrow TP'(v) \) \( \) Definition 12

2 \( tr \prec \langle \forall \rangle \in traces(c?v \rightarrow TP'(v)) \land \)
\( \exists 1 \leq j \leq n \cdot \text{foot}(tr) = \text{update}_{N_j} \) \( \) Initial assumption

3 \( \exists tr', x \cdot (x \in \text{type}(c) \land tr = (c.x) \cap tr') \land tr' \in traces(TP'(x)) \)

4 \( tr' \prec \langle \forall \rangle \in traces(TP'(x)) \land \)
\( \exists 1 \leq j \leq n \cdot \text{foot}(tr') = \text{update}_{N_j} \) \( \) \( 2, 3 \)

5 \( wp(trans(P'(x)), Q) \Rightarrow wp(tr', Q) \) \( \) Induction assumption

6 \( wp(tr, Q) = wp((c.x) \cap tr', Q) \) \( \) \( 3 \)

7 \( wp((c.x) \cap tr', Q) = wp(tr', Q) \) \( \) Definition 13

8 \( wp(tr, Q) = wp(tr', Q) \) \( \) \( 6, 7 \)
\[ \text{Definition 10} \]
\[
\begin{align*}
\text{trans}(c?v \rightarrow P'(v)) &= \text{ANY } v \text{ WHERE } v : \text{type}(c) \text{ THEN } \text{trans}(P'(v)) \text{ END} \\
\text{wp}(\text{trans}(c?v \rightarrow P'(v)), Q) &= \\
&\forall v \cdot (v \in \text{type}(c) \Rightarrow \text{wp}(\text{trans}(P'(v)), Q)) \\
\text{wp}(\text{trans}(c?v \rightarrow P'(v)), Q) &\Rightarrow \\
&\text{wp}(\text{trans}(P'(v)), Q) \\
\text{wp}(\text{trans}(c?v \rightarrow P'(v)), Q) &\Rightarrow \text{wp}(\text{tr}', Q) \\
\text{wp}(\text{trans}(c?v \rightarrow P'(v)), Q) &\Rightarrow \text{wp}(\text{tr}, Q)
\end{align*}
\]

Inductive case 6: \( cp?z \rightarrow P'(z) \)
Assume Inductive Hypothesis for \( P'(z) \).

\[
\begin{align*}
\text{Definition 12} \\
\text{term}(cp?z \rightarrow P'(z)) &= cp?z \rightarrow TP'(z) \\
\text{Initial assumption} \\
\exists tr \cdot tr' \in \text{traces}(cp?z \rightarrow TP'(z)) \land \\
&\exists 1 \leq j \leq n \cdot \text{foot}(tr) = \text{update}\_N_j \\
\exists tr', x \cdot (x \in MR \land tr = \langle cp.x \rangle \cap tr' \land \\
&tr' \in \text{traces}(TP'(x)) \\
\exists tr' \cdot (tr' \in \text{traces}(TP'(x)) \land \\
&\exists 1 \leq j \leq n \cdot \text{foot}(tr') = \text{update}\_N_j \\
\text{Induction assumption} \\
\text{Induction assumption} \\
\text{Definition 13}
\end{align*}
\]
8 \( wp(tr, Q) = (\text{assert}(cp_x) \Rightarrow wp(tr', Q)) \)

9 \( \text{trans}(cp?z \rightarrow P'(z)) = \)
\[
\text{ANY } z \text{ WHERE } z : MR \text{ THEN} \\
\text{SELECT } \text{assert}(cp_x) \text{ THEN} \\
\text{trans}(P'(z)) \text{ END} \\
\text{END}
\]

10 \( wp(\text{trans}(cp?z \rightarrow P'(z)), Q) = \)
\[
\forall z \bullet (z \in MR \Rightarrow \\
(\text{assert}(cp_x) \Rightarrow wp(\text{trans}(P'(z)), Q)))
\]

11 \( wp(\text{trans}(cp?z \rightarrow P'(z)), Q) \Rightarrow \\
(\text{assert}(cp_x) \Rightarrow wp(\text{trans}(P'(z)), Q))
\]

12 \( wp(\text{trans}(cp?z \rightarrow P'(z)), Q) \Rightarrow \\
(\text{assert}(cp_x) \Rightarrow wp(tr', Q))
\]

13 \( wp(\text{trans}(cp?z \rightarrow P'(z)), Q) \Rightarrow wp(tr, Q) \)

Inductive case 7 : \( P' \sqcap P'' \)

Assume Inductive Hypothesis for \( P' \) and \( P'' \).

1 \( \text{term}(P' \sqcap P'') = TP' \sqcap TP'' \) \hspace{1cm} Definition 12

2 \( tr \cap \langle \checkmark \rangle \in \text{traces}(TP' \sqcap TP'') \) \hspace{1cm} Initial assumption

3 \( \exists 1 \leq j \leq n \bullet \text{foot}(tr) = \text{update}_j \) \hspace{1cm} Initial assumption

4 \( tr \cap \langle \checkmark \rangle \in \text{traces}(TP') \lor \\
tr \cap \langle \checkmark \rangle \in \text{traces}(TP'') \)

5 \( tr \cap \langle \checkmark \rangle \in \text{traces}(TP') \Rightarrow \\
(wp(\text{trans}(P'), Q) \Rightarrow wp(tr, Q)) \)

3, Induction assumption
6 \[ \text{tr} \cap \langle \forall \rangle \in \text{traces}(\text{TP}^n) \Rightarrow \]
\[ (\text{wp}(\text{trans}(\text{P}''), Q) \Rightarrow \text{wp}(\text{tr}, Q)) \] 3, Induction assumption

7 \[ \text{trans}(\text{P}' \sqcap \text{P}'') = \]
\[ \text{CHOICE} \text{ trans}(\text{P}') \text{ OR } \text{trans}(\text{P}'') \text{ END} \] Definition 10

8 \[ \text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) = \]
\[ \text{wp}(\text{trans}(\text{P}'), Q) \land \text{wp}(\text{trans}(\text{P}''), Q) \] 7

9 \[ \text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) \Rightarrow \text{wp}(\text{trans}(\text{P}'), Q) \] 8

10 \[ \text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) \Rightarrow \text{wp}(\text{trans}(\text{P}''), Q) \] 8

11 \[ \text{tr} \cap \langle \forall \rangle \in \text{traces}(\text{TP}') \Rightarrow \]
\[ (\text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) \Rightarrow \text{wp}(\text{tr}, Q)) \] 5, 9

12 \[ \text{tr} \cap \langle \forall \rangle \in \text{traces}(\text{TP}'') \Rightarrow \]
\[ (\text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) \Rightarrow \text{wp}(\text{tr}, Q)) \] 6, 10

13 \[ \text{wp}(\text{trans}(\text{P}' \sqcap \text{P}''), Q) \Rightarrow \text{wp}(\text{tr}, Q) \] 11, 12, 4

Inductive case 8 : \( P' \sqcap P'' \)
Assume Inductive Hypothesis for \( P' \) and \( P'' \).
The proof has the same structure as the proof of Inductive case 7.

Inductive case 9 : if \( b \) then \( P' \) else \( P'' \)
Assume Inductive Hypothesis for \( P' \) and \( P'' \).

1 \[ \text{term}(\text{if } b \text{ then } P' \text{ else } P'') = \text{if } b \text{ then } \text{TP}' \text{ else } \text{TP}'' \] Definition 12

2 \[ \text{tr} \cap \langle \forall \rangle \in \text{traces}(\text{if } b \text{ then } \text{TP}' \text{ else } \text{TP}'') \] Initial assumption

3 \[ \exists 1 \leq j \leq n \ast \text{foot}(\text{tr}) = \text{update}_{N_j} \] Initial assumption

4 \[ b \Rightarrow \text{tr} \cap \langle \forall \rangle \in \text{traces}(\text{TP}') \] 2
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

5 \( \neg b \Rightarrow tr \land (\forall) \in traces(TP'') \)

6 \( b \Rightarrow (wp(trans(P'), Q) \Rightarrow wp(tr, Q)) \)

7 \( \neg b \Rightarrow (wp(trans(P''), Q) \Rightarrow wp(tr, Q)) \)

8 \( \text{trans(if } b \text{ then } P' \text{ else } P'' = IF } b \text{ THEN } trans(P') \text{ ELSE } trans(P'') \text{ END} \)

9 \( wp(trans(if b then P' else P''), Q) = b \Rightarrow wp(trans(P'), Q) \land \neg b \Rightarrow wp(trans(P''), Q) \)

10 \( wp(trans(if b then P' else P''), Q) \Rightarrow b \Rightarrow wp(tr, Q) \land \neg b \Rightarrow wp(tr, Q) \)

11 \( wp(trans(if b then P' else P''), Q) \Rightarrow b \Rightarrow wp(tr, Q) \land \neg b \Rightarrow wp(tr, Q) \)

12 \( wp(trans(if b then P' else P''), Q) \Rightarrow wp(tr, Q) \)

Since we have proved the Base case and the Inductive case, it establishes Lemma 4.

\(\square\)

Lemma 5. If \( R \Rightarrow wp(trans(P_i), Q) \) then

\( \forall tr \cdot ((tr \land (\forall) \in traces(TP_i) \land foot(tr) = update_{N_i}) \Rightarrow (R \Rightarrow wp(tr, Q))) \)

where \( 1 \leq i, j \leq n \).

Proof. This follows from Lemma 4.

1 \( R \Rightarrow wp(trans(P_i), Q) \)

2 \( tr \land (\forall) \in traces(TP_i) \land foot(tr) = update_{N_i} \)

3 \( wp(trans(P_i), Q) \Rightarrow wp(tr, Q) \)

4 \( R \Rightarrow wp(tr, Q) \)

\(\square\)
**CHAPTER 3. MOBILE CSP**

**Corollary 2.** If \((\text{rec} = N_i \land \text{CLI}_i) \Rightarrow \wp(\text{trans}(P_i), \text{rec} = N_j \Rightarrow \text{CLI}_j)\) then
\[
\forall tr \cdot ((tr \prec (\checkmark) \in \text{traces}(TP_i) \land \text{foot}(tr) = \text{update}_{N_j}) \Rightarrow
((\text{rec} = N_i \land \text{CLI}_i) \Rightarrow \wp(tr, \text{rec} = N_j \Rightarrow \text{CLI}_j)))
\]
where \(1 \leq i, j \leq n\).

**Proof.** This follows from Lemma 5 using \((\text{rec} = N_i \land \text{CLI}_i)\) for \(R\) and
\((\text{rec} = N_j \Rightarrow \text{CLI}_j)\) for \(Q\). \(\Box\)

Recall that this section is concerned with proving Theorem 1, we are therefore restricting our attention to LOOP that is CLI preserver. For such LOOP the next corollary follows immediately.

**Corollary 3.** \(
\forall tr \cdot ((tr \prec (\checkmark) \in \text{traces}(TP_i) \land \text{foot}(tr) = \text{update}_{N_j}) \Rightarrow
((\text{rec} = N_i \land \text{CLI}_i) \Rightarrow \wp(tr, \text{rec} = N_j \Rightarrow \text{CLI}_j)))
\]
where \(1 \leq i, j \leq n\).

Now, we present the next lemma for the traces of \(TP_i\) which are followed by \((\checkmark)\) but their last element is not \(\text{update}_{N_i}\).

**Lemma 6.** If \(tr \prec (\checkmark) \in \text{traces}(TP_i)\) and \(\forall 1 \leq j \leq n \cdot \text{foot}(tr) \neq \text{update}_{N_j}\) then \(\wp(\text{trans}(P_i), \text{true}) \Rightarrow \wp(tr, \text{true})\) where \(1 \leq i \leq n\).

**Proof.** We prove this lemma by using the Principle of Induction on \(P_i\).

**Base case 1:** \(P_i = \text{SKIP}\)

1. \(P_i = \text{SKIP}\) \hspace{1cm} Initial assumption
2. \(\text{trans}(P_i) = \)
\[\text{SELECT} \text{false} \text{ THEN skip END}\]
3. \(\wp(\text{trans}(P_i), \text{true}) = \text{true}\)
4. \(TP_i = \text{SKIP}\) \hspace{1cm} 1, Definition 12
5. \(tr = ()\) \hspace{1cm} 4, \(tr \prec (\checkmark) \in \text{traces}(TP_i) \land \forall 1 \leq j \leq n \cdot \text{foot}(tr) \neq \text{update}_{N_j}\)
6. \(\wp(tr, \text{true}) = \text{true}\)
7. \(\wp(\text{trans}(P_i), \text{true}) \Rightarrow \wp(tr, \text{true})\) \hspace{1cm} 3, 6
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES  75

Base case 2: $P_i = N(E_1, \ldots, E_k)$

1. $P_i = N(E_1, \ldots, E_k)$ \hspace{1cm} Initial assumption

2. $TP_i = update_{N(E_1, \ldots, E_k)} \rightarrow SKIP$ \hspace{1cm} 1, Definition 12

3. $tr = \langle update_{N(E_1, \ldots, E_k)} \rangle$ \hspace{1cm} 2, $tr \wedge \langle \forall \rangle \in traces(TP_i)$

4. $foot(tr) = update_{N(E_1, \ldots, E_k)}$ \hspace{1cm} 3

5. The condition part does not hold \hspace{1cm} 3, 4

6. Lemma is true \hspace{1cm} 5

The proof of Inductive cases 1, 2, 3, 4, 5, 6, 7, 8, 9 has the same structure as the proof of Lemma 4 but we consider $true$ instead of $Q$.

\[ \square \]

Lemma 7. $\forall tr \cdot (tr \wedge \langle \forall \rangle \in traces(TP_i) \Rightarrow ((rec = N_i \wedge CLI_i) \Rightarrow wp(tr, true)))$
where $1 \leq i \leq n$.

Proof. As $LOOP$ is CLI preserver, the lemma is proved as below:

1. $tr \wedge \langle \forall \rangle \in traces(TP_i)$ \hspace{1cm} Initial assumption

2. $(rec = N_i \wedge CLI_i) \Rightarrow$
   \hspace{1cm} Definition 11
   \hspace{1cm} $wp(tr, true)$
   \hspace{1cm} $\forall 1 \leq j \leq n \cdot (rec = N_j \Rightarrow CLI_j))$

3. $(rec = N_i \wedge CLI_i) \Rightarrow wp(tr, true)$ \hspace{1cm} 2

There are two possibilities for $tr$:

Case 1: $\exists 1 \leq j \leq n \cdot foot(tr) = update_{N_j}$

4. $wp(tr, Q) \Rightarrow wp(tr, Q)$ \hspace{1cm} 1, Lemma 4

5. $wp(tr, true) \Rightarrow wp(tr, true)$ \hspace{1cm} 4, using $true$ for $Q$

6. $(rec = N_i \wedge CLI_i) \Rightarrow wp(tr, true)$ \hspace{1cm} 3, 5
Case 2: $\forall 1 \leq j \leq n \cdot \text{foot}(tr) \neq \text{update}_N$.

7 $\text{wp}(\text{trans}(P_i), \text{true}) \Rightarrow \text{wp}(tr, \text{true})$ 1, Lemma 6

8 $(\text{rec} = N_i \land \text{CLI}) \Rightarrow \text{wp}(tr, \text{true})$ 3, 7

Lemma 8. $\forall \text{tr} \cdot ((\text{tr} \in \text{traces}(TP_i) \land \text{foot}(tr) \neq \langle \nu \rangle) \Rightarrow \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \nu \rangle \in \text{traces}(TP_i))$

where $1 \leq i \leq n$.

Proof. We prove this lemma by using the Principle of Induction on $P_i$.

Base case 1 : $P_i = \text{SKIP}$

1 $P_i = \text{SKIP}$ Initial assumption

2 $TP_i = \text{SKIP}$ 1, Definition 12

3 $tr = \langle \rangle$ 2, $tr \in \text{traces}(TP_i)$ $\land \text{foot}(tr) \neq \langle \nu \rangle$

4 $tr \cap \langle \nu \rangle \in \text{traces}(TP_i)$ 2, 3

5 $\exists tr' = \langle \rangle \cdot tr \cap tr' \cap \langle \nu \rangle \in \text{traces}(TP_i)$ 4

Base case 2 : $P_i = N(E_1, \ldots, E_k)$

1 $P_i = N(E_1, \ldots, E_k)$ Initial assumption

2 $TP_i = update_N(E_1, \ldots, E_k) \rightarrow \text{SKIP}$ 1, Definition 12

3 $tr = \langle \rangle \lor tr = \langle \text{update}_N(E_1, \ldots, E_k) \rangle$ 2, $tr \in \text{traces}(TP_i)$ $\land \text{foot}(tr) \neq \langle \nu \rangle$

Situation 1 : $tr = \langle \rangle$

4 $tr = \langle \rangle$ 3, Situation 1

5 $tr \cap \langle \text{update}_N(E_1, \ldots, E_k) \rangle \cap \langle \nu \rangle \in \text{traces}(TP_i)$ 2, 4

6 $\exists tr' = \langle \text{update}_N(E_1, \ldots, E_k) \rangle \cdot tr \cap tr' \cap \langle \nu \rangle \in \text{traces}(TP_i)$ 5
Situation 2: $tr = \langle \text{update}_N(b_1, \ldots, b_k) \rangle$

7. $tr = \langle \text{update}_N(b_1, \ldots, b_k) \rangle$ 3, Situation 2

8. $tr \cap \emptyset \cap (\triangledown) \in \text{traces}(TP_t)$ 2, 7

9. \exists tr' = \emptyset \cdot tr \cap tr' \cap (\triangledown) \in \text{traces}(TP_t)$ 8

Inductive case 1: $e \rightarrow P'$

Assume Inductive Hypothesis for $P'$.

1. $\text{term}(e \rightarrow P') = e \rightarrow TP'$ Definition 12

2. $\forall tr_1 \cdot ((tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq (\triangledown)) \Rightarrow \exists tr'' \cdot tr_1 \cap tr'' \cap (\triangledown) \in \text{traces}(TP'))$ Induction assumption

3. $tr = \emptyset \lor$
   $tr = \langle e \rangle \cap tr_1 \cdot$
   $(tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq (\triangledown))$

Situation 1: If $tr = \emptyset$

4. $tr = \emptyset$ 3, Situation 1

5. \exists tr'' \cdot tr'' \cap (\triangledown) \in \text{traces}(TP') 2, assuming $tr_1 = \emptyset$

6. \exists tr'' \cdot \langle e \rangle \cap tr'' \cap (\triangledown) \in \text{traces}(e \rightarrow TP') 5

7. \exists tr'' \cdot \emptyset \cap (e) \cap tr'' \cap (\triangledown) \in \text{traces}(e \rightarrow TP') 6

8. \exists tr'' \cdot tr \cap (e) \cap tr'' \cap (\triangledown) \in \text{traces}(e \rightarrow TP') 7, 4

9. \exists tr' = \langle e \rangle \cap tr'' \cdot tr \cap tr' \cap (\triangledown) \in \text{traces}(e \rightarrow TP') 8
Situation 2: If \( \tau = (a) \cdot \tau_1 \cdot (\tau_1 \in \text{traces}(TP')) \land \text{foot} (\tau_1) \neq (\checkmark) \)
1. \( \tau = (a) \cdot \tau_1 \cdot (\tau_1 \in \text{traces}(TP') \land \text{foot} (\tau_1) \neq (\checkmark)) \) 3, Situation 2
2. \( \exists \tau'' \cdot \tau_1 \cdot \tau'' \cdot (\checkmark) \in \text{traces} (TP') \) 10, 2
3. \( \exists \tau'' \cdot (a) \cdot \tau_1 \cdot \tau'' \cdot (\checkmark) \in \text{traces} (e \rightarrow TP') \) 11
4. \( \exists \tau'' \cdot \tau \cdot \tau'' \cdot (\checkmark) \in \text{traces} (e \rightarrow TP') \) 12, 10
5. \( \exists \tau'' \cdot \tau' = \tau'' \cdot \tau \cdot \tau'' \cdot (\checkmark) \in \text{traces} (e \rightarrow TP') \) 13

Inductive case 2: \( clv \rightarrow P' \)
Assume Inductive Hypothesis for \( P' \).
1. \( \text{term} (clv \rightarrow P') = clv \rightarrow TP' \) Definition 12
2. \( \forall \tau_1 \cdot ((\tau_1 \in \text{traces}(TP')) \land \text{foot} (\tau_1) \neq (\checkmark)) \Rightarrow \exists \tau'' \cdot \tau_1 \cdot \tau'' \cdot (\checkmark) \in \text{traces} (TP') \) Induction assumption
3. \( \tau = () \lor \tau = (c.v) \cdot \tau_1 \cdot (\tau_1 \in \text{traces}(TP') \land \text{foot} (\tau_1) \neq (\checkmark)) \)

Situation 1: If \( \tau = () \)
4. \( \tau = () \) 3, Situation 1
5. \( \exists \tau'' \cdot \tau'' \cdot (\checkmark) \in \text{traces}(TP') \) 2, assuming \( \tau_1 = () \)
6. \( \exists \tau'' \cdot (c.v) \cdot \tau_1 \cdot \tau'' \cdot (\checkmark) \in \text{traces}(c!v \rightarrow TP') \) 5
7. \( \exists \tau'' \cdot () \cdot (c.v) \cdot \tau'' \cdot (\checkmark) \in \text{traces}(c!v \rightarrow TP') \) 6
8. \( \exists \tau'' \cdot \tau \cdot (c.v) \cdot \tau'' \cdot (\checkmark) \in \text{traces}(c!v \rightarrow TP') \) 7, 4
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

\[ \exists \ tr' = \langle c. v \rangle \cap tr'' \bullet \]
\[ \ tr \cap tr' \langle \checkmark \rangle \in \text{traces}(c!v \rightarrow TP') \]

Situation 2: If \( tr = \langle c. v \rangle \cap tr_1 \bullet (tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq \langle \checkmark \rangle) \)

\[ \exists \ tr'' = tr_1 \cap tr'' \langle \checkmark \rangle \in \text{traces}(TP') \]
\[ \exists \ tr'' = tr_1 \cap tr'' \langle \checkmark \rangle \in \text{traces}(c!v \rightarrow TP') \]

Inductive case 3: \( cplz \rightarrow P' \)
Assume Inductive Hypothesis for \( P' \).

\[ \text{term}(cplz \rightarrow P') = cplz \rightarrow TP' \]
Definition 12

\[ \forall tr_1 \bullet ((tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq \langle \checkmark \rangle) \Rightarrow \exists tr'' \bullet tr_1 \cap tr'' \langle \checkmark \rangle \in \text{traces}(TP')) \]
Induction assumption

\[ tr = \langle \rangle \lor \]
\[ tr = \langle cpl.z \rangle \cap tr_1 \bullet \]
\[ (tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq \langle \checkmark \rangle) \]
\[ tr \in \text{traces}(cplz \rightarrow TP') \land \]
\[ \text{foot}(tr) \neq \langle \checkmark \rangle \]

Situation 1: If \( tr = \langle \rangle \)

\[ tr = \langle \rangle \]
3, Situation 1

\[ \exists tr'' \bullet tr'' \langle \checkmark \rangle \in \text{traces}(TP') \]
2, assuming \( tr_1 = \langle \rangle \)

\[ \exists tr'' \bullet \]
\[ \langle cpl.z \rangle \cap tr'' \langle \checkmark \rangle \in \text{traces}(cplz \rightarrow TP') \]
5
CHAPTER 3. MOBILE CSP

Inductive case 4: \( c?v \rightarrow P'(v) \)

Assume Inductive Hypothesis for \( P'(v) \).

1. \( \text{term}(c?v \rightarrow P'(v)) = c?v \rightarrow TP'(v) \quad \text{Definition 12} \)

2. \( \forall x, tr_1 \cdot ((x \in \text{type}(c) \land tr_1 \in \text{traces}(TP'(x)) \land \text{foot}(tr_1) \neq (\checkmark)) \implies \exists tr'' \cdot tr_1 \cap tr'' \cap (\checkmark) \in \text{traces}(TP'(x))) \quad \text{Induction assumption} \)

3. \( tr = () \lor (c.x) \cap tr_1 \cdot (tr \in \text{traces}(c?v \rightarrow TP'(v)) \land (tr_1 \in \text{traces}(TP'(x)) \land \text{foot}(tr_1) \neq (\checkmark)) \quad \text{foot}(tr) \neq (\checkmark) \)

4. \( \exists tr'' \cdot \)

5. \( (tr_1 \in \text{traces}(TP'(x)) \land \text{foot}(tr_1) \neq (\checkmark)) \}

6. \( ) \cap (cp.x) \cap tr'' \cap (\checkmark) \in \text{traces}(cplz \rightarrow TP') \)

7. \( \exists tr'' \cdot \)

8. \( tp \cap (cp.x) \cap tr'' \cap (\checkmark) \in \text{traces}(cplz \rightarrow TP') \)

9. \( \exists tr' = (cp.z) \cap tr'' \cdot \)

10. \( tr \cap tr' \cap (\checkmark) \in \text{traces}(cplz \rightarrow TP') \)

Situation 2: If \( tr = (cp.x) \cap tr_1 \cdot (tr_1 \in \text{traces}(TP') \land \text{foot}(tr_1) \neq (\checkmark)) \)

11. \( \exists tr'' \cdot \)

12. \( tr_1 \cap tr'' \cap (\checkmark) \in \text{traces}(TP') \)

13. \( \exists tr'' \cdot (cp.z) \cap tr_1 \cap tr'' \cap (\checkmark) \in \text{traces}(cplz \rightarrow TP') \)

14. \( \exists tr' = tr'' \cdot tr \cap tr' \cap (\checkmark) \in \text{traces}(cplz \rightarrow TP') \)
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

Situation 1: If \( \text{tr} = \{ \} \)

4. \( \text{tr} = \{ \} \)

3, Situation 1

5. \( \forall x \in \text{type}(c) \ \exists \text{tr}'' \bullet \text{tr}'' \circ \langle x \rangle \in \text{traces}(\text{TP}'(x)) \)

2, assuming \( \text{tr}_1 = \{ \} \)

6. \( \forall x \in \text{type}(c) \ \exists \text{tr}'' \bullet \)

\( \langle c.x \rangle \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

5

7. \( \forall x \in \text{type}(c) \ \exists \text{tr}'' \bullet \)

\( \langle x \rangle \circ \langle x \rangle \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

6

8. \( \forall x \in \text{type}(c) \ \exists \text{tr}'' \bullet \)

\( \text{tr} \circ \langle c.x \rangle \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

7, 4

9. \( \forall x \in \text{type}(c) \ \exists \text{tr}' = \langle c.x \rangle \circ \text{tr}'' \bullet \)

\( \text{tr} \circ \text{tr}' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

8

Situation 2: If \( \text{tr} = \langle c.x \rangle \circ \text{tr}_1 \bullet (\text{tr}_1 \in \text{traces}(\text{TP}'(x)) \land \text{foot}(\text{tr}_1) \neq \langle x \rangle) \)

10. \( \text{tr} = \langle c.x \rangle \circ \text{tr}_1 \bullet (\text{tr}_1 \in \text{traces}(\text{TP}'(x)) \land \text{foot}(\text{tr}_1) \neq \langle x \rangle) \)

3, Situation 2

11. \( \exists \text{tr}'' \bullet \text{tr}_1 \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(\text{TP}'(x)) \)

10, 2

12. \( \exists \text{tr}'' \bullet \langle c.x \rangle \circ \text{tr}_1 \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

11

13. \( \exists \text{tr}'' \bullet \text{tr} \circ \text{tr}'' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

12, 10

14. \( \exists \text{tr}' = \text{tr}'' \bullet \text{tr} \circ \text{tr}' \circ \langle x \rangle \in \text{traces}(c?u \rightarrow \text{TP}'(v)) \)

13
Inductive case 5: \( cp?z \rightarrow P'(z) \)
Assume Inductive Hypothesis for \( P'(z) \).

1. \( \text{term}(cp?z \rightarrow P'(z)) = cp?z \rightarrow TP'(z) \)  
   \( \text{Definition 12} \)

2. \( \forall x, \text{tr}_1 \cdot \\
   \left( (x \in MR \land \text{tr}_1 \in \text{traces}(TP'(x)) \land \\
   \text{foot}(\text{tr}_1) \neq \langle \checkmark \rangle ) \Rightarrow \\
   \exists \text{tr}'' \cdot \text{tr}_1 \bowtie \text{tr}'' \bowtie \langle \checkmark \rangle \in \text{traces}(TP'(x)) \right) 
\)  
   \( \text{Induction assumption} \)

3. \( \text{tr} = () \lor \\
   \text{tr} = (cp.x) \bowtie \text{tr}_1 \cdot \\
   (\text{tr}_1 \in \text{traces}(TP'(x)) \land \text{foot}(\text{tr}_1) \neq \langle \checkmark \rangle ) \)

Situation 1: If \( \text{tr} = () \)

4. \( \text{tr} = () \)  
   \( 3, \text{Situation 1} \)

5. \( \forall x \in MR \exists \text{tr}'' \cdot \text{tr}'' \bowtie \langle \checkmark \rangle \in \text{traces}(TP'(x)) \)  
   \( 2, \text{assuming } \text{tr}_1 = () \)

6. \( \forall x \in MR \exists \text{tr}'' \cdot \\
   (cp.x) \bowtie \text{tr}'' \bowtie \langle \checkmark \rangle \in \text{traces}(cp?z \rightarrow TP'(x)) \)

7. \( \forall x \in MR \exists \text{tr}'' \cdot \\
   () \bowtie (cp.x) \bowtie \text{tr}'' \bowtie \langle \checkmark \rangle \in \text{traces}(cp?z \rightarrow TP'(x)) \)

8. \( \forall x \in MR \exists \text{tr}'' \cdot \\
   \text{tr} \bowtie (cp.x) \bowtie \text{tr}'' \bowtie \langle \checkmark \rangle \in \text{traces}(cp?z \rightarrow TP'(x)) \)

9. \( \forall x \in MR \exists \text{tr}' = (cp.x) \bowtie \text{tr}'' \cdot \\
   \text{tr} \bowtie \text{tr}' \bowtie \langle \checkmark \rangle \in \text{traces}(cp?z \rightarrow TP'(x)) \)

10. \( \forall x \in MR \exists \text{tr}_1 \cdot \\
    \text{tr}_1 \bowtie \langle \checkmark \rangle \in \text{traces}(cp?z \rightarrow TP'(x)) \)
Situation 2: If \( tr = \langle cp.x \rangle \cdot tr_1 \cdot (tr_1 \in traces(TP'(x)) \land foot(tr_1) \neq (\checkmark)) \)

10. \( tr = \langle cp.x \rangle \cdot tr_1 \cdot (tr_1 \in traces(TP'(x)) \land foot(tr_1) \neq (\checkmark)) \) 3, Situation 2

11. \( \exists tr'' \cdot tr_1 \cdot tr'' \cdot (\checkmark) \in traces(TP'(x)) \)

12. \( \exists tr'' \cdot (cp.x) \cdot tr_1 \cdot tr'' \cdot (\checkmark) \in traces(cp?z \rightarrow TP'(x)) \)

13. \( \exists tr'' \cdot tr \cdot tr'' \cdot (\checkmark) \in traces(cp?z \rightarrow TP'(x)) \)

14. \( \exists tr' = tr'' \cdot tr \cdot tr' \cdot (\checkmark) \in traces(cp?z \rightarrow TP'(x)) \)

Inductive case 6: \( z. oplst \rightarrow P'(t) \)

Assume Inductive Hypothesis for \( P'(t) \).

1. \( \text{term}(z. oplst \rightarrow P'(t)) = z. oplst \rightarrow TP'(t) \)  
   
   Definition 12

2. \( \forall x, tr_1 \cdot ((z. op.s.x) \in traces(z. oplst \rightarrow TP'(t)) \land tr_1 \in traces(TP'(x)) \land foot(tr_1) \neq (\checkmark)) \)  
   \( \Rightarrow \exists tr'' \cdot tr_1 \cdot tr'' \cdot (\checkmark) \in traces(TP'(x)) \)  
   Induction assumption

3. \( tr = () \)  
   \( \forall x \cdot (\langle z. op.s.x \rangle \in traces(z. oplst \rightarrow TP'(t)) \land tr_1 \in traces(TP'(x)) \land foot(tr_1) \neq (\checkmark)) \)  
   \( \Rightarrow tr \in traces \)  
   \( tr \in traces \)  
   \( (z. oplst \rightarrow TP'(t)) \land \)  
   \( foot(tr) \neq (\checkmark) \)

Situation 1: If \( tr = () \)

4. \( tr = () \) 3, Situation 1

5. \( \forall x \cdot ((z. op.s.x) \in traces(z. oplst \rightarrow TP'(t)) \Rightarrow \exists tr'' \cdot tr'' \cdot (\checkmark) \in traces(TP'(x)) \)  
   2, assuming \( tr_1 = () \)

6. \( \forall x \cdot ((z. op.s.x) \in traces(z. oplst \rightarrow TP'(t)) \Rightarrow \exists tr'' \cdot (z. op.s.x) \cdot tr'' \cdot (\checkmark) \in traces(z. oplst \rightarrow TP'(t)) \)  
   5
\[
\forall x \bullet ((z \cdot \text{op} \cdot s \cdot x) \in \text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)) \Rightarrow \\
\exists t'' \bullet t \cap (z \cdot \text{op} \cdot s \cdot x) \cap t'' \cap (\checkmark) \in \\
\text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)))
\]

\[
\forall x \bullet ((z \cdot \text{op} \cdot s \cdot x) \in \text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)) \Rightarrow \\
\exists t'' \bullet t' \cap (z \cdot \text{op} \cdot s \cdot x) \cap t'' \cap (\checkmark) \in \\
\text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)))
\]

\[
\forall x \bullet ((z \cdot \text{op} \cdot s \cdot x) \in \text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)) \Rightarrow \\
\exists t'' \bullet t' = (z \cdot \text{op} \cdot s \cdot x) \cap t'' \bullet \\
t' \cap t'' \cap (\checkmark) \in \text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t)))
\]

Situation 2: If \( t' = (z \cdot \text{op} \cdot s \cdot x) \cap t_1 \bullet (t_1 \in \text{traces}(TP'(x)) \land \text{foot}(t_1) \neq (\checkmark)) \)

\[
t' = (z \cdot \text{op} \cdot s \cdot x) \cap t_1 \bullet \\
(t_1 \in \text{traces}(TP'(x)) \land \text{foot}(t_1) \neq (\checkmark))
\]

\[
\exists t'' \bullet t_1 \cap t'' \cap (\checkmark) \in \text{traces}(TP'(x))
\]

\[
\exists t'' \bullet (z \cdot \text{op} \cdot s \cdot x) \cap t_1 \cap t'' \cap (\checkmark) \in \\
\text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t))
\]

\[
\exists t'' \bullet t \cap t'' \cap (\checkmark) \in \text{traces}(z \cdot \text{op} \cdot s \cdot t \rightarrow T'P(t))
\]

Inductive case 7: \( P' \Box P'' \)

Assume Inductive Hypothesis for \( P' \) and \( P'' \).

1. \( \text{term}(P' \Box P'') = T'P' \Box T'P'' \) Definition 12

2. \( t' \in \text{traces}(T'P' \Box T'P'') \) 1, Initial assumption

3. \( \text{foot}(t') \neq (\checkmark) \) Initial assumption
3.6. DIVERGENCE FREEDOM OF A CONTROLLER AND ITS MACHINES

4. \( \text{tr} \in \text{traces}(TP') \lor \text{tr} \in \text{traces}(TP'') \) \hspace{1cm} 2

5. \( \text{tr} \in \text{traces}(TP') \Rightarrow \)
\( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP') \) \hspace{1cm} 3, Induction assumption

6. \( \text{tr} \in \text{traces}(TP') \Rightarrow \)
\( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP' \cap TP'') \) \hspace{1cm} 5

7. \( \text{tr} \in \text{traces}(TP'') \Rightarrow \)
\( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP'') \) \hspace{1cm} 3, Induction assumption

8. \( \text{tr} \in \text{traces}(TP'') \Rightarrow \)
\( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP' \cap TP'') \) \hspace{1cm} 7

9. \( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP' \cap TP'') \) \hspace{1cm} 4, 6, 8

Inductive case 8: \( P' \cap P'' \)
Assume Inductive Hypothesis for \( P' \) and \( P'' \).
The proof has the same structure as the proof of Inductive case 7.

Inductive case 9: \( \text{if } b \text{ then } P' \text{ else } P'' \)
Assume Inductive Hypothesis for \( P' \) and \( P'' \).

1. \( \text{term(if } b \text{ then } P' \text{ else } P'') = \)
\( \text{if } b \text{ then } TP' \text{ else } TP'' \)
\hspace{1cm} Definition 12

2. \( \text{tr} \in \text{traces(if } b \text{ then } TP' \text{ else } TP'') \) \hspace{1cm} 1, Initial assumption

3. \( \text{foot(tr)} \neq \langle \sqrt{\rangle} \) \hspace{1cm} Initial assumption

4. \( b \Rightarrow \text{tr} \in \text{traces}(TP') \land \neg b \Rightarrow \text{tr} \in \text{traces}(TP'') \) \hspace{1cm} 2

5. \( \text{tr} \in \text{traces}(TP') \Rightarrow \)
\( \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP') \) \hspace{1cm} 3, Induction assumption

6. \( b \Rightarrow \exists \text{tr}' \cdot \text{tr} \cap \text{tr}' \cap \langle \sqrt{\rangle} \in \text{traces}(TP') \) \hspace{1cm} 4, 5
\[ b \Rightarrow \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(\text{if } b \text{ then } TP' \text{ else } TP'') \]

8 \[ \tau \in \text{traces}(TP'') \Rightarrow \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(TP'') \]

9 \[ \neg \tau \Rightarrow \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(TP'') \]

10 \[ \neg \tau \Rightarrow \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(TP'') \]

11 \[ \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(\text{if } b \text{ then } TP' \text{ else } TP'') \]

Since we have proved the Base case and the Inductive case, it establishes the lemma. \(\square\)

**Lemma 9.** \( \forall 1 \leq i \leq n \cdot TP_i \text{ sat } ((\text{rec } = N_i \land CL_i) \Rightarrow wp(\tau, \text{true})) \)

**Proof.** There are two possibilities for \( \tau \):

**Case 1:** \( \text{foot}(\tau) = (\text{true}) \)

1 \[ (\text{rec } = N_i \land CL_i) \Rightarrow wp(\text{front}(\tau), \text{true}) \] \hspace{1cm} \text{Lemma 7}

2 \[ wp(\text{front}(\tau) \land (\text{true}), \text{true}) = wp(\text{front}(\tau), \text{true}) \] \hspace{1cm} \text{Definition 13}

3 \[ wp(\tau, \text{true}) = wp(\text{front}(\tau), \text{true}) \]

4 \[ (\text{rec } = N_i \land CL_i) \Rightarrow wp(\tau, \text{true}) \]

**Case 2:** \( \text{foot}(\tau) \neq (\text{true}) \)

1 \[ \exists \tau' \cdot \tau \mathbin{\tau'} \land (\text{true}) \in \text{traces}(TP_i) \] \hspace{1cm} \text{Lemma 8}

2 \[ (\text{rec } = N_i \land CL_i) \Rightarrow wp(\tau \mathbin{\tau'}, \text{true}) \] \hspace{1cm} \text{Lemma 7}

3 \[ (\text{rec } = N_i \land CL_i) \Rightarrow wp(\tau, wp(\tau', \text{true})) \]

4 \[ (\text{rec } = N_i \land CL_i) \Rightarrow wp(\tau, \text{true}) \]

\[ wp(\tau', \text{true}) \Rightarrow \text{true} \]

\(\square\)
Lemma 10. \( \forall tr \in \text{traces}(\text{LOOP}) \cdot tr = tr_1 \cap tr_2 \cap \ldots \cap tr_k \land \\
tr_1 \cap (\text{update}_{N_2}) \cap (\checkmark) \in \text{traces}(TP_{j_1}) \land \\
tr_2 \cap (\text{update}_{N_2}) \cap (\checkmark) \in \text{traces}(TP_{j_2}) \land \\
\ldots \\
tr_k \cap (\text{update}_{N_{j_{k+1}}}) \cap (\checkmark) \in \text{traces}(TP_{j_k}) \land \\
j_1 = 1 \land 1 \leq j_2, \ldots, j_k, j_{k+1} \leq n \)

\[ \Rightarrow \]

\( \text{wp}(\text{init}_1; \ldots; \text{init}_m; tr \cap (\text{update}_{N_{j_{k+1}}}); \text{rec} := N_{j_{k+1}} \land \text{CL}_i_{j_{k+1}}) \)

Proof. We prove this lemma by using the Principle of Induction on \( k \).

Base case : \( k = 1 \)

1 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; \text{rec} := N_1, \text{CL}_1) \)

Definition 11

2 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; \text{rec} := N_1, \text{rec} = N_1 \land \text{CL}_1) \)

1

3 \( (\text{rec} = N_1 \land \text{CL}_1) \Rightarrow \\
\text{wp}(tr_1 \cap (\text{update}_{N_2}); \text{rec} = N_2 \Rightarrow \text{CL}_2) \)

Corollary 3

4 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; \text{rec} := N_1, \text{wp}(tr_1 \cap (\text{update}_{N_2}); \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

2, 3

5 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; \text{rec} := N_1; tr_1, \text{wp}(\text{update}_{N_2}; \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

4

6 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; \text{rec} := N_1; tr_1, \text{wp}(\text{rec} := N_2, \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

5, Definition 13

7 \( \forall k, j_k \cdot (k \geq 1 \land 1 \leq j_k \leq n \Rightarrow \text{update}_{N_{j_k}} \notin \sigma(tr_k)) \\
\text{update}_N \notin \chi(\text{LOOP}) \)

wp(\text{init}_1; \ldots; \text{init}_m; tr_1; \text{rec} := N_1, \text{wp}(\text{rec} := N_2, \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

6, 7

9 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; tr_1, \text{wp}(\text{rec} := N_1; \text{rec} := N_2, \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

8

10 \( \text{wp}(\text{init}_1; \ldots; \text{init}_m; tr_1, \text{wp}(\text{rec} := N_2, \text{rec} = N_2 \Rightarrow \text{CL}_2)) \)

9
\begin{center}
\begin{align*}
11 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1, \\
10 & \quad \quad wp(\text{rec} := N_{j_0}, \text{rec} = N_{j_0} \land CL_{j_0})) \\
12 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1, \\
11, \text{Definition 13} & \quad \quad wp((\text{update}_{N_{j_0}}), \text{rec} = N_{j_0} \land CL_{j_0})) \\
13 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1 \circ (\text{update}_{N_{j_0}}), \\
12 & \quad \quad \text{rec} = N_{j_0} \land CL_{j_0})
\end{align*}
\end{center}

Inductive case:

Assume Inductive Hypothesis for \(k = q\). We prove that the lemma is true for \(k = q+1\).

\begin{center}
\begin{align*}
1 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1 \circ \ldots \circ \tau_q \circ (\text{update}_{N_{j_{q+1}}}), \\
\text{Induction assumption} & \quad \quad \text{rec} := N_{j_{q+1}} \land CL_{j_{q+1}}) \\
2 & \quad (\text{rec} := N_{j_{q+1}} \land CL_{j_{q+1}}) \Rightarrow \\
\text{Corollary 3} & \quad \quad wp(\tau_{q+1} \circ (\text{update}_{N_{j_{q+2}}}), \text{rec} := N_{j_{q+2}} \Rightarrow CL_{j_{q+2}}) \\
3 & \quad (\text{rec} := N_{j_{q+1}} \land CL_{j_{q+1}}) \Rightarrow \\
3, \text{Definition 13} & \quad \quad wp(\tau_{q+1}, wp(\text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \Rightarrow CL_{j_{q+2}})) \\
4 & \quad (\text{rec} := N_{j_{q+1}} \land CL_{j_{q+1}}) \Rightarrow \\
4 & \quad \quad wp(\tau_{q+1}, \text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \land CL_{j_{q+2}})) \\
5 & \quad (\text{rec} := N_{j_{q+1}} \land CL_{j_{q+1}}) \Rightarrow \\
5 & \quad \quad wp(\tau_{q+1}; \text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \land CL_{j_{q+2}}) \\
6 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1 \circ \ldots \circ \tau_q \circ (\text{update}_{N_{j_{q+1}}}), \\
1, 6 & \quad \quad \text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \land CL_{j_{q+2}})) \\
7 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1 \circ \ldots \circ \tau_q \circ (\text{update}_{N_{j_{q+1}}}) \\
7 & \quad \quad \text{tr}_{\tau_{q+1}; \text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \land CL_{j_{q+2}}}) \\
8 & \quad wp(\text{init}_1; \ldots; \text{init}_m; \tau_1 \circ \ldots \circ \tau_q \circ (\text{update}_{N_{j_{q+1}}}) \\
7 & \quad \quad \text{tr}_{\tau_{q+1}; \text{rec} := N_{j_{q+2}}, \text{rec} := N_{j_{q+2}} \land CL_{j_{q+2}}}) \\
9 & \quad \forall k, j_k \cdot (k \geq 1 \land 1 \leq j_k \leq n \Rightarrow \text{update}_{N_{j_k}} \notin \sigma(\tau_k)) \quad \text{update}_{N} \notin \chi(\text{LOOP})
\end{align*}
\end{center}
The following lemma follows from Lemma 10.

Lemma 11. LOOP sat $wp(init_1; \ldots; init_m; tr, true)$

Proof. $\forall tr \in traces(LOOP) \exists x \geq 1, j_1 = 1, 1 \leq j_1, \ldots, j_x \leq n \bullet$

\[
tr = tr_1 \sqcap tr_2 \sqcap \ldots \sqcap tr_x \sqcap \\
tr_1 \sqcap \langle update_{N_{j_1}} \rangle \sqcap \langle \forall \rangle \in traces(TP_{j_1}) \sqcap \\
tr_2 \sqcap \langle update_{N_{j_2}} \rangle \sqcap \langle \forall \rangle \in traces(TP_{j_2}) \sqcap \\
\vdots \\
tr_{x-1} \sqcap \langle update_{N_{j_{x-1}}} \rangle \sqcap \langle \forall \rangle \in traces(TP_{j_{x-1}}) \sqcap \\
tr_x \in traces(TP_{j_x})
\]
There are two possibilities for $x$:

Case 1: $x = 1$

1. $tr \in traces(TP_1)$

2. $wp(init_1; \ldots; init_m; \ rec := N_1, CLI_1)$

3. $wp(init_1; \ldots; init_m; \ rec := N_1, rec = N_1 \land CLI_1)$

4. $(rec = N_1 \land CLI_1) \Rightarrow wp(tr, true)$

5. $wp(init_1; \ldots; init_m; \ rec := N_1, wp(tr, true))$

6. $wp(init_1; \ldots; init_m; \ rec := N_1; tr, true)$

7. $update_{N_1} \notin \sigma(tr)$

8. $wp(init_1; \ldots; init_m; tr; rec := N_1, true)$

9. $wp(init_1; \ldots; init_m; tr, wp(rec := N_1, true))$

10. $wp(init_1; \ldots; init_m; tr, true)$

Case 2: $x \geq 2$

1. $wp(init_1; \ldots; init_m; tr_1 \ldots tr_{x-1} \ (update_{N_x}),$
   $\ rec = N_x \land CLI_{N_x})$

2. $(rec = N_x \land CLI_{N_x}) \Rightarrow wp(tr_x, true)$

3. $wp(init_1; \ldots; init_m; tr_1 \ldots tr_{x-1} \ (update_{N_x}),$
   $\ wp(tr_x, true))$  

4. $wp(init_1; \ldots; init_m; tr_1 \ldots tr_{x-1},$
   $\ wp(update_{N_x}) \ (tr_x, true))$

5. $wp(init_1; \ldots; init_m; tr_1 \ldots tr_{x-1},$
   $\ wp(rec := N_x, wp(tr_x, true)))$
Lemma 12. If \( \text{tr} \) is a sequence of events in LOOP such that \( \exists cp. z \in \sigma(\text{tr}) \bullet cp \in CP, \) and \( M \) is a B machine with machine reference \( z \), then \( \wp(\text{tr}, Q) \Rightarrow \wp(\text{tr} \upharpoonright z.\alpha M, Q) \) where \( Q \) is a predicate on the state of \( M \).

Proof. We prove this lemma by using the Principle of Induction on \( \text{tr} \).

Base case: \( \text{tr} = \{ \} \)

1 \( \text{tr} = \{ \} \) Initial assumption

2 \( \wp(\text{tr}, Q) = Q \) 1, Definition 13

3 \( \text{tr} \upharpoonright z.\alpha M = \{ \} \) 1

4 \( \wp(\text{tr} \upharpoonright z.\alpha M, Q) = Q \) 3, Definition 13

5 \( \wp(\text{tr}, Q) = \wp(\text{tr} \upharpoonright z.\alpha M, Q) \) 2, 4

6 \( \wp(\text{tr}, Q) \Rightarrow \wp(\text{tr} \upharpoonright z.\alpha M, Q) \) 5
Inductive case 1: \((e) \sim tr'\)
Assume Inductive Hypothesis for \(tr'\).

1. \(\text{wp}(\langle e \rangle \sim tr', Q)\)  
   Initial assumption

2. \(\text{wp}(tr', Q)\)  
   1, Definition 13

3. \(\text{wp}(tr', Q) \Rightarrow \text{wp}(tr' \mid z.\alpha M, Q)\)  
   Induction assumption

4. \(\text{wp}(tr' \mid z.\alpha M, Q)\)  
   2, 3

5. \(tr' \mid z.\alpha M = \langle (e) \sim tr' \rangle \mid z.\alpha M\)  
   \(e \not\in z.\alpha M\)

6. \(\text{wp}(\langle (e) \sim tr' \rangle \mid z.\alpha M, Q)\)  
   4, 5

Inductive case 2: \((c.x) \sim tr'\)
Assume Inductive Hypothesis for \(tr'\). The proof has the same structure as the proof of Inductive case 1 but we consider \(c.x\) instead of \(e\).

Inductive case 3: \((cp.w) \sim tr'\)
Assume Inductive Hypothesis for \(tr'\). There are two possibilities for \(cp\):

1. \(cp \in \chi_o(LOOP)\)
2. \(cp \in \chi_1(LOOP)\)

Situation 1: \(cp \in \chi_o(LOOP)\)

1. \(cp \in \chi_o(LOOP)\)  
   Situation 1

2. \(\text{wp}(\langle cp.w \rangle \sim tr', Q)\)  
   Initial assumption

3. \(\text{assert}(cp.w) \land \text{wp}(tr', Q)\)  
   2, 1, Definition 13

4. \(\text{wp}(tr', Q)\)  
   3

5. \(\text{wp}(tr', Q) \Rightarrow \text{wp}(tr' \mid z.\alpha M, Q)\)  
   Induction assumption

6. \(\text{wp}(tr' \mid z.\alpha M, Q)\)  
   4, 5
7 \ tr' \updownarrow z.\alpha M = ((cp.w) \cap tr') \updownarrow z.\alpha M \quad cp.w \notin z.\alpha M

8 \ wp(((cp.w) \cap tr') \updownarrow z.\alpha M, Q) \quad 6, 7

Situation 2: \ cp \in \chi_i(LOOP)

We assume \ w is the reference of machine \ M_1. In addition, we assume \ VAR(z : M) and \ VAR(w : M_1) are the set of all variables in \ z : M and \ w : M_1 respectively.

1 \ cp \in \chi_i(LOOP) \quad \text{Situation 2}

2 \ wp((cp.w) \cap tr', Q) \quad \text{Initial assumption}

3 \ assert(cp_w) \Rightarrow wp(tr', Q) \quad 2, 1, \text{Definition 13}

4 \ wp(tr', Q) \Rightarrow wp(tr' \updownarrow z.\alpha M, Q) \quad \text{Induction assumption}

5 \ assert(cp_w) \Rightarrow wp(tr' \updownarrow z.\alpha M, Q) \quad 3, 4

6 \ wp(tr' \updownarrow z.\alpha M, Q) \quad 5, var(Q) \cap var(assert(cp_w)) = \emptyset
    \quad VAR(z : M) \cap VAR(w : M_1) = \emptyset
    \quad assert(cp_w) = True \lor assert(cp_w) = False

7 \ tr' \updownarrow z.\alpha M = ((cp.w) \cap tr') \updownarrow z.\alpha M \quad cp.w \notin z.\alpha M

8 \ wp(((cp.w) \cap tr') \updownarrow z.\alpha M, Q) \quad 6, 7

Inductive case 4: \ (z.op.s.t) \cap tr'

Assume Inductive Hypothesis for \ tr'.

1 \ wp((z.op.s.t) \cap tr', Q) \quad \text{Initial assumption}

2 \ wp(t \leftarrow z.op(s), wp(tr', Q)) \quad 1, \text{Definition 13}

3 \ wp(tr', Q) \Rightarrow wp(tr' \updownarrow z.\alpha M, Q) \quad \text{Induction assumption}

4 \ wp(t \leftarrow z.op(s), wp(tr' \updownarrow z.\alpha M, Q)) \quad 2, 3

5 \ wp((z.op.s.t) \cap (tr' \updownarrow z.\alpha M), Q) \quad 4, \text{Definition 13}
6 \((z \text{.op.s.t}) \cap (tr' \mid z.AM) = ((z \text{.op.s.t}) \cap tr') \mid z.AM\) \(z \text{.op.s.t} \in z.AM\)

7 \(wp(((z \text{.op.s.t}) \cap tr') \mid z.AM, Q)\) 5, 6

Inductive case 5: \(<w \text{.op.s.t}> tr' \bullet w \neq z>\)

Assume Inductive Hypothesis for \(tr'\). We assume \(w\) is the reference of machine \(M_1\). In addition, we assume \(VAR(z : M)\) and \(VAR(w : M_1)\) are the set of all variables in \(z : M\) and \(w : M_1\) respectively.

1 \(wp(<w \text{.op.s.t} > \cap tr', Q)\) Initial assumption

2 \(wp(t < w \text{.op}(s), wp(tr', Q))\) 1, Definition 13

3 \(wp(tr', Q) \Rightarrow wp(tr' \mid z.AM, Q)\) Induction assumption

4 \(wp(t < w \text{.op}(s), wp(tr' \mid z.AM, Q))\) 2, 3

5 \(wp(tr' \mid z.AM, Q)\) 4, \(VAR(z : M) \cap VAR(w : M_1) = \emptyset\)

6 \(tr' \mid z.AM = ((z \text{.op.s.t}) \cap tr') \mid z.AM\) \(w \text{.op.s.t} \notin z.AM\)

7 \(wp(((z \text{.op.s.t}) \cap tr') \mid z.AM, Q)\) 5, 6

\(\square\)

Corollary 4. If \(tr\) is a sequence of events in LOOP such that \(\not\exists \text{cp.x} \in \sigma(tr) \bullet \text{cp} \in CP\), and \(M\) is a B machine with machine reference \(z\), then \(wp(tr, true) \Rightarrow wp(tr \mid z.AM, true)\).

Proof. This corollary follows from Lemma 12 using \(true\) for \(Q\).

\(\square\)

Lemma 13. LOOP sat

\((\text{foot}(tr) = \text{cp.x} \land \text{cp} \in \chi_e(LOOP)) \Rightarrow wp(init_1; \ldots; init_m; tr; assert(cp_2))\)

Proof. This lemma follows from Lemma 11.
Lemma 14. If $z_1, z_2, \ldots, z_m$ are the machine references for $B$ machines $M_1, M_2, \ldots, M_m$ respectively, then: $\forall 1 \leq i \leq m \cdot \text{LOOP sat}$

\[
\begin{align*}
\text{foot}(tr) &= cp.z_i \\
\exists dp \in CP \cdot dp.z_i &\in \sigma(\text{front}(tr))
\end{align*}
\]

\[
\implies wp(init_1; \ldots; init_m; tr, assert(cp_{z_i}))
\]

Proof. This lemma follows from Lemma 12 and Lemma 13.

1. $wp(init_1; \ldots; init_m; tr, assert(cp_{z_i}))$  \hspace{1cm} \text{Lemma 13}

2. $wp(init_1; \ldots; init_m; \text{front}(tr) \cap (cp.z_i), assert(cp_{z_i}))$  \hspace{1cm} 1

3. $wp(init_1; \ldots; init_m; \text{front}(tr), wp((cp.z_i), assert(cp_{z_i})))$  \hspace{1cm} 2

4. $wp((cp.z_i), assert(cp_{z_i})) = assert(cp_{z_i})$  \hspace{1cm} \text{Definition 13,} \\
\hspace{1cm} cp \in \chi_o(LOOP)$
The lemma above establishes that whenever \( \text{LOOP} \) is giving one of the machines it owns initially to another controller through a control point, the state of the machine satisfies the assertion of that control point.

**Lemma 15.** If \( z_1, z_2, \ldots, z_m \) are the machine references for \( B \) machines \( M_1, M_2, \ldots, M_m \) respectively, then: \( \forall 1 \leq i \leq m \cdot \text{LOOP} \sat (\exists cp, z_i \in \sigma(tr) \bullet cp \in CP) \Rightarrow tr \uparrow z_i.\alpha M_i \notin D[z_i : M_i]\)

**Proof.** This lemma is a result of Lemma 11, Corollary 4 and Lemma 1.
Lemma 16. If \( z_1, z_2, \ldots, z_m \) are the machine references for \( B \) machines \( M_1, M_2, \ldots, M_m \) respectively, then: \( \forall 1 \leq i \leq m \cdot \text{LOOP sat} \)

\[
\text{foot}(tr) = cp.z_i \\
cp \in X_e(\text{LOOP}) \\
\exists dp \in CP \bullet dp.z_i \in \sigma(\text{front}(tr)) \}
\]

\[\implies tr \upharpoonright z_i.AM_i \not\in D[z_i : M_i] \]

Proof. This lemma follows from Lemma 14.

1. \( wp(\text{init}_1; \ldots; \text{init}_m; tr \upharpoonright z_i.AM_i, \text{assert}(cp_{z_i})) \) Lemma 14
2. \( wp(\text{init}_i; tr \upharpoonright z_i.AM_i, \text{assert}(cp_{z_i})) \) 1
3. \( wp(\text{init}_i; tr \upharpoonright z_i.AM_i, \text{true}) \) 2, \( \text{assert}(cp_{z_i}) \Rightarrow \text{true} \)
4. \( tr \upharpoonright z_i.AM_i \not\in D[z_i : M_i] \) 3, Lemma 1 

The next corollary follows immediately from Lemma 15 and Lemma 16.

Corollary 5. The parallel combination between \( \text{LOOP} \) and any machine it owns initially is divergence free from the beginning of the execution and as long as they are working with each other. If \( \text{LOOP} \) gives one of these machines to another controller, the parallel combination between \( \text{LOOP} \) and that machine is divergence free from the beginning of the execution until it gives it to another controller.

Lemma 17. If \( M \) is a \( B \) machine with machine reference \( z \), then: \( \text{LOOP sat} \)

\[
tr = tr_1 \cap (cp.z) \cap tr_2 \\
cp \in X_e(\text{LOOP}) \\
\exists dp.z \in \sigma(tr_2) \bullet dp \in CP \}
\]

\[\implies (\text{assert}(cp_{z_i}) \Rightarrow wp(tr_2 \upharpoonright z.AM_i, \text{true})) \]

Proof. This lemma follows from Corollary 4 and Lemma 11.
Lemma 18. If $M$ is a B machine with machine reference $z$ and $tr_z$ is the trace of $M$ when it is given to LOOP and $tr_z$ is not a divergence, then: LOOP sat
\[
tr = tr_1 \cap (cp \cdot x) \cap tr_2,
\]
\[
\{\begin{align*}
& cp \in \chi_i(LOOP) \\
& \#dp, x \in \sigma(tr_2) \cdot dp \in CP
\end{align*}
\right\} \implies (assert(cp_z) \Rightarrow tr_2 \cap (tr_2 \mid z.\alpha M) \notin D[z : M])
\]

Proof. This lemma follows from Lemma 17. We assume $init$ is the Initialisation clause of machine $M$.

1. $assert(cp_z) \Rightarrow wp(init; tr_z, assert(cp_z))$ 
   Definition of $tr_z$

2. $assert(cp_z) \Rightarrow wp(tr_2 \mid z.\alpha M, true)$ 
   Lemma 17

3. $assert(cp_z) \Rightarrow wp(init; tr_z, wp(tr_2 \mid z.\alpha M, true))$ 
   1, 2

4. $assert(cp_z) \Rightarrow wp(init; tr_z \cap (tr_2 \mid z.\alpha M), true)$ 
   3

5. $assert(cp_z) \Rightarrow tr_2 \cap (tr_2 \mid z.\alpha M) \notin D[z : M]$ 
   4, Lemma 1
Lemma 19. If $M$ is a $B$ machine with machine reference $x$, then: $LOOP$ sat

\[
\begin{align*}
tr &= tr_1 \cap (cp.x) \cap tr_2 \\
cp &\in \chi_i(LOOP) \\
foot(tr_2) &= dp.z \\
dp &\in \chi_o(LOOP) \\
\exists ep.z \in \sigma(front(tr_2)) \cdot ep \in CP
\end{align*}
\]

\[\Rightarrow (assert(cp_x) \Rightarrow wp(tr_2 \uplus z.aM, assert(dp_x)))
\]

Proof. This lemma follows from Lemma 12 and Lemma 13.

1. \(wp(init_1; \ldots; init_m; tr, assert(dp_z))\) \hspace{1cm} Lemma 13
2. \(wp(init_1; \ldots; init_m; tr_1 \cap (cp.x) \cap tr_2, assert(dp_z))\)
3. \(wp(init_1; \ldots; init_m; tr_1 \cap (cp.x) \cap front(tr_2), \)
\hspace{1cm} \(wp((dp.x), assert(dp_z)))\)
4. \(wp((dp.x), assert(dp_z)) = assert(dp_z)\) \hspace{1cm} Definition 13, \(dp \in \chi_o(LOOP)\)
5. \(wp(init_1; \ldots; init_m; tr_1 \cap (cp.x) \cap front(tr_2), assert(dp_z))\) \hspace{1cm} 3, 4
6. \(wp(init_1; \ldots; init_m; tr_1 \cap (cp.x), wp(front(tr_2), assert(dp_z)))\) \hspace{1cm} 5
7. \(wp(front(tr_2), assert(dp_z)) \Rightarrow wp(front(tr_2) \uplus z.aM, assert(dp_z))\) \hspace{1cm} Lemma 12
8. \(wp(init_1; \ldots; init_m; tr_1 \cap (cp.x), \)
\hspace{1cm} \(wp(front(tr_2) \uplus z.aM, assert(dp_z)))\)
9. \(wp(init_1; \ldots; init_m; tr_1, \)
\hspace{1cm} \(wp((cp.x) \cap (front(tr_2) \uplus z.aM), assert(dp_z)))\)
10. \(wp(init_1; \ldots; init_m; tr_1, \)
\hspace{1cm} \(assert(cp_x) \Rightarrow wp(front(tr_2) \uplus z.aM, assert(dp_z)))\) \hspace{1cm} 9, Definition 13, \(cp \in \chi_i(LOOP)\)
11. \(assert(cp_x) \Rightarrow wp(front(tr_2) \uplus z.aM, assert(dp_z))\) \hspace{1cm} 10
12. \(front(tr_2) \uplus z.aM = (front(tr_2) \cap (dp.x)) \uplus z.aM\) \hspace{1cm} \(dp_z \not\in z.aM\)
The lemma above establishes that if LOOP receives a machine with the correct state according to the assertion of that control point, then whenever it passes that machine to another controller through a control point, the state of the machine satisfies the assertion of that control point.

Lemma 20. If $M$ is a B machine with machine reference $z$, then: LOOP sat

\[
\begin{align*}
tr &= tr_1 \cap (cp_z) \cap tr_2 \\
cp &= \chi_1(LOOP) \\
foot(tr_2) &= dp_z \\
dp &= \chi_0(LOOP) \\
\exists ep.z \in \sigma(front(tr_2)) \cdot ep \in CP
\end{align*}
\]

\[\rightarrow (assert(cp_z) \Rightarrow wp(tr_2 \upharpoonright z.M, assert(dp_z)))\]

Proof. This lemma follows from Lemma 19.

1. \[assert(cp_z) \Rightarrow wp(tr_2 \upharpoonright z.M, assert(dp_z)) \quad \text{Lemma 19}\]

2. \[assert(cp_z) \Rightarrow wp(tr_2 \upharpoonright z.M, true) \quad 1, assert(dp_z) \Rightarrow true\]

Lemma 21. If $M$ is a B machine with machine reference $z$ and $tr_z$ is the trace of $M$ when it is given to LOOP and $tr_z$ is not a divergence, then: LOOP sat

\[
\begin{align*}
tr &= tr_1 \cap (cp_z) \cap tr_2 \\
cp &= \chi_1(LOOP) \\
foot(tr_2) &= dp_z \\
dp &= \chi_0(LOOP) \\
\exists ep.z \in \sigma(front(tr_2)) \cdot ep \in CP
\end{align*}
\]

\[\Rightarrow (assert(cp_z) \Rightarrow tr_z \not\in \mathcal{D}[z : M])\]

Proof. This lemma follows from Lemma 20. We assume init is the Initialisation clause of machine $M$. 


3.7. DIVERGENCE FREEDOM OF THE WHOLE SYSTEM

1. \( \text{assert}(cp_z) \Rightarrow \text{wp}(\text{init}; tr_z, \text{assert}(cp_z)) \) Definition of \( tr_z \)

2. \( \text{assert}(cp_z) \Rightarrow \text{wp}(tr_z \upharpoonright z \cdot M, \text{true}) \) Lemma 20

The rest of the proof has the same structure as Lemma 18.

The next corollary follows from Lemma 18 and Lemma 21.

**Corollary 6.** If LOOP receives a machine with the correct state according to the assertion of that control point and if the machine does not have divergence until the point it is being passed to LOOP, then the parallel combination between LOOP and that machine is divergence free as long as they are working with each other. If LOOP gives that machine to another controller, the parallel combination between LOOP and that machine is divergence free until LOOP gives that machine to another controller.

Corollary 5 and Corollary 6 establish Theorem 1. Hence according to Theorem 1, if LOOP is a divergence free CSP controller and if for each \( N_i (1 \leq i \leq n) \) in LOOP, a Control Loop Invariant, CLI, can be found such that the two conditions in Definition 11 are satisfied, then the parallel combination between LOOP and any machine it works with during the execution is divergence free as long as the state of the machines LOOP receives always satisfy the related assertions and they do not have divergence until the point they are passed to LOOP.

**Example 5.** Consider LOOP_1 and \( M \) in Figure 3.2 in which LOOP_1 owns \( M \) initially. LOOP_1 is established to be divergence free (e.g. by using FDR, see Chapter 4). In addition, in Example 4, we have established that LOOP_1 is CLI preserver. Therefore according to Theorem 1, the parallel combination between LOOP_1 and \( M \) is divergence free from the beginning of the execution until LOOP_1 passes \( M \) to another controller through control point \( cp \). Also, if the variable \( n \) of the machines it receives during the execution through control point \( dp \) are always 0, then the parallel combination between LOOP_1 and any machine it receives through \( dp \) is divergence free. In Figure 3.2, we assumed that the system contains only one B machine, \( M \). Therefore, if the variable \( n \) of \( M \) is 0 whenever it is passed to LOOP_1 through \( dp \), then the parallel combination between LOOP_1 and \( M \) is divergence free until LOOP_1 gives \( M \) to LOOP_2 through \( cp \).

### 3.7. Divergence freedom of the whole system

In this section we discuss how divergence freedom of a mobile combined communicating system can be established. If the parallel combination of CSP controllers
is not divergence free, then the whole system will have divergence. Therefore, the first step is to establish that the CSP part of the system is divergence free. If the CSP part is divergence free, then any divergence in the system can only arise from the B machines. Thus, the second step in divergence freedom verification is to establish that the operations of all machines in the system are always called inside their precondition by the controllers during the execution.

If the CSP controllers in a system have the following four conditions, then the operations of all B machines in the system will always be called inside their precondition at all the time during the execution:

1. If a controller has a machine initially, it always calls the machine's operations inside their precondition.

2. If a controller receives a machine with the correct state according to the related assertion, then it always calls the machine's operations inside their precondition.

3. Whenever a controller gives a machine it owns initially to another controller, it always passes that machine to the receiver controller with the correct state according to the related assertion.

4. If a controller receives a machine with the correct state according to the assertion of that control point, then whenever it gives that machine to another controller, it always passes that machine to the receiver controller with the correct state according to the related assertion.

We have already proved in previous section that all four conditions above exist in a CSP controller if it is CLI preserver. Therefore, if all CSP controllers in the system are CLI preserver, then the operations of all B machines are always called inside their precondition during the execution which then results the divergence freedom of the whole system.

The remainder of this section presents the steps towards proving that in order to establish the divergence freedom of the overall combination, it is sufficient to establish that the parallel combination of the CSP controllers is divergence free and all controllers in the system are CLI preserver.

The lemma below establishes that in a system containing only one B machine, if the parallel combination of controllers is divergence free and all controllers are CLI preserver, then whenever this machine is exchanged between two controllers, it will always be exchanged with the correct state according to the related assertion.
Lemma 22. Supposing LOOP_1, LOOP_2, ..., LOOP_n are the CSP controllers and M is the only B machine in a mobile combined communicating system. If the parallel combination of controllers is divergence free and all controllers are CLI preservers, then: LOOP_1 || LOOP_2 || ... || LOOP_n || z : M sat
∀ cp ∈ CP • (cp.z ∈ σ(tr) ⇒ wp(init; tr' | z.aM, assert(cp_z)))
where z is the reference of machine M, init is the Initialisation clause of machine M and tr' ≤ tr ∧ foot(tr') = cp.z.

Proof. We prove this lemma by using the Principle of Induction on the number of control points cp_k in tr. cp_k is the k-th control point in tr.

Base case: k = 1
we want to prove that the lemma is true for the first control point cp_1. In other words, we want to prove that:
wp(init; tr_1 | z.aM, assert(cp_1)) where tr_1 ≤ tr ∧ foot(tr_1) = cp_1.z

We assume cp_1 ∈ X_o(LOOP_j). LOOP_j is the first controller which works with M from the beginning of the execution until it gives M to another controller along cp_1.

1 tr_1 ≤ tr ∧ foot(tr_1) = cp_1.z Initial assumption
2 cp_1 ∈ X_o(LOOP_j) Initial assumption
3 cp ∈ CP • cp.z ∈ σ(front(tr_1)) Initial assumption
4 tr'_1 = tr_1 | χ(LOOP_j), s(LOOP_j) Introducing tr'_1
5 wp(init; tr'_1 | z.aM, assert(cp_1z)) 1, 2, 3, 4, Lemma 14
6 tr'_1 | z.aM = tr_1 | z.aM 1, 2, 3, 4
7 wp(init; tr'_1 | z.aM, assert(cp_1z)) 5, 6

Inductive case:
we want to prove that if the lemma is true for the first k number of control points cp_k, then it is true for the first k + 1 number of control points cp_{k+1}. In other words, we want to prove that:
wp(init; tr_k | z.aM, assert(cp_{k+1z}) ⇒ wp(init; tr_{k+1} | z.aM, assert(cp_{k+1z}))
where tr_k ≤ tr ∧ foot(tr_k) = cp_k.z ∧ tr_{k+1} ≤ tr ∧ foot(tr_{k+1}) = cp_{k+1}.z.

We assume cp_k ∈ X_o(LOOP_j). When LOOP_j receives M along cp_k, it is the only controller which works with M until it gives M to another controller along cp_{k+1}. 
1. \( tr_k \leq tr \land \text{foot}(tr_k) = cp_k \cdot z \)  
   Initial assumption

2. \( tr_{k+1} \leq tr \land \text{foot}(tr_{k+1}) = cp_{k+1} \cdot z \)  
   Initial assumption

3. \( cp_k \in \chi(LOOP_2) \land cp_{k+1} \in \chi(LOOP_2) \)  
   Initial assumption

4. \( wp(init; tr_k \upharpoonright z \cdot M, assert(cp_{k+1})) \)  
   Induction assumption

5. \( a = tr_{k+1} \upharpoonot \# tr_k \)  
   Introducing a

6. \( \exists cp \cdot z \in \sigma(\text{front}(a)) \cdot cp \in CP \)  
   Initial assumption

7. \( \text{assert}(cp_{k+1}) \Rightarrow \)  
   \( wp((a \upharpoonright \chi(LOOP_2), s(LOOP_2)) \upharpoonright z \cdot M, \)  
   \( \text{assert}(cp_{k+1})) \)  
   1, 2, 3, 5, 6, Lemma 19

8. \( (a \upharpoonright \chi(LOOP_2), s(LOOP_2)) \upharpoonright z \cdot M = a \upharpoonright z \cdot M \)  
   1, 2, 3, 5, 6

9. \( \text{assert}(cp_{k+1}) \Rightarrow wp(a \upharpoonright z \cdot M, assert(cp_{k+1})) \)  
   7, 8

10. \( wp(init; tr_k \upharpoonright z \cdot M, wp(a \upharpoonright z \cdot M, assert(cp_{k+1}))) \)  
    4, 9

11. \( wp(init; (tr_k \upharpoonright z \cdot M) \land (a \upharpoonright z \cdot M), assert(cp_{k+1})) \)  
    10

12. \( wp(init; tr_k \upharpoonot a \upharpoonright z \cdot M, assert(cp_{k+1})) \)  
    11

13. \( wp(init; tr_{k+1} \upharpoonright z \cdot M, assert(cp_{k+1})) \)  
    12

Now, we are able to provide a theorem that demonstrates the conditions for divergence freedom of mobile combined communicating systems containing only one B machine.

**Theorem 2.** Suppose \( LOOP_1, LOOP_2, ..., LOOP_n \) are the CSP controllers and \( M \) is the only B machine in a mobile combined communicating system. If the parallel combination of controllers is divergence free and all controllers are CLI preserver, then the whole system, \( SYS = LOOP_1 \parallel LOOP_2 \parallel ... \parallel LOOP_n \parallel z : M \), is divergence free where \( z \) is the reference of machine \( M \).

**Proof.** This theorem follows from Lemmas 1, 15, 17 and 22.
3.7. DIVERGENCE FREEDOM OF THE WHOLE SYSTEM

1. \( D[\text{LOOP}_1 \parallel \text{LOOP}_2 \parallel \ldots \parallel \text{LOOP}_n] = \emptyset \)  
   Initial assumption

2. \( tr \in \text{traces}(SYS) \)  
   Introducing arbitrary \( tr \)

3. \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \) \( \lor \) \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)  
   \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)

Situation 1: \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)

4. \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)  
   Situation 1

5. \( \exists 1 \leq j \leq n \bullet z \in s(\text{LOOP}_j) \)  
   Initial owner of \( M \)

6. \( (tr \upharpoonright x(\text{LOOP}_j), s(\text{LOOP}_j)) \upharpoonright z.\alpha M \notin D[z : M] \)  
   \( 4, 5, \text{Lemma 15} \)

7. \( (tr \upharpoonright x(\text{LOOP}_j), s(\text{LOOP}_j)) \upharpoontright z.\alpha M = tr \upharpoontright z.\alpha M \)  
   \( 4, 5 \)

8. \( tr \upharpoonright z.\alpha M \notin D[z : M] \)  
   \( 6, 7 \)

9. \( \forall tr \in \text{traces}(SYS) \bullet \)  
   \( (\exists cp_z \in \sigma(tr) \bullet cp \in CP) \Rightarrow tr \upharpoontright z.\alpha M \notin D[z : M] \)  
   \( 2, 4, 8 \)

Situation 2: \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)

10. \( \exists cp_z \in \sigma(tr) \bullet cp \in CP \)  
   Situation 2

11. \( \exists cp \in CP \bullet \)  
   \( (tr = a \land b \land \text{foot}(a) = cp_z \land \exists dp_z \in \sigma(b) \bullet dp \in CP) \)  
   \( 10 \)

12. \( wp(\text{init}; a \upharpoonright x.\alpha M, \text{assert}(cp_a)) \)  
   \( 11, \text{Lemma 22} \)

13. \( \exists 1 \leq x \leq n \bullet cp \in x_i(\text{LOOP}_x) \)  
   \( cp \in CP \)

14. \( \text{assert}(cp_x) \Rightarrow wp((b \upharpoonright x(\text{LOOP}_x), s(\text{LOOP}_x)) \upharpoontright z.\alpha M, \text{true}) \)  
   \( 11, 13, \text{Lemma 17} \)

15. \( (b \upharpoonright x(\text{LOOP}_x), s(\text{LOOP}_x)) \upharpoontright z.\alpha M = b \upharpoontright z.\alpha M \)  
   \( 11, 13 \)

16. \( \text{assert}(cp_x) \Rightarrow wp(b \upharpoonright z.\alpha M, \text{true}) \)  
   \( 14, 15 \)

17. \( wp(\text{init}; a \upharpoonright x.\alpha M, wp(b \upharpoonright z.\alpha M, \text{true})) \)  
   \( 12, 16 \)

18. \( wp(\text{init}; (a \upharpoonright z.\alpha M) \upharpoontright (b \upharpoonright z.\alpha M), \text{true}) \)  
   \( 17 \)
Now, we can achieve the following result from Situations 1 and 2:

\[ \forall tr \in traces(SYS) \cdot (\exists cp \cdot x \in \sigma(tr) \cdot cp \in CP) \Rightarrow tr \mid z \cdot M \notin D[z : M] \]

The next theorem follows immediately from Theorem 2. It states the conditions which are sufficient to establish the divergence freedom of a mobile combined communicating system containing more than one B machines.

**Theorem 3.** Supposing LOOP\(_1\), LOOP\(_2\), ..., LOOP\(_n\) are the CSP controllers and M\(_1\), M\(_2\), ..., M\(_m\) are the B machines in a mobile combined communicating system. If the parallel combination of controllers is divergence free and all controllers are CLI preserver, then the whole system, SYS = LOOP\(_1\) || LOOP\(_2\) || ... || LOOP\(_n\) || z\(_1\) : M\(_1\) || z\(_2\) : M\(_2\) || ... || z\(_m\) : M\(_m\), is divergence free where z\(_1\), z\(_2\), ..., z\(_m\) are the machine references for B machines M\(_1\), M\(_2\), ..., M\(_m\) respectively.

**Proof.** We prove this theorem by contradiction. We assume that the system is not divergence free.

1. \(\mathcal{D}[SYS] \neq \emptyset\)  
   Contradictory assumption

2. \(tr \in \mathcal{D}[SYS]\)  
   1, Introducing arbitrary tr

3. \(\mathcal{D}[\text{LOOP}_1 || \text{LOOP}_2 || ... || \text{LOOP}_n] = \emptyset\)  
   Initial assumption

4. \(\forall 1 \leq j \leq m \cdot \mathcal{D}[\text{LOOP}_1 || \text{LOOP}_2 || ... || \text{LOOP}_n || z_j : M_j] = \emptyset\)  
   Theorem 2

5. \(\forall 1 \leq j \leq m \cdot tr \mid z_j \cdot M_j \notin \mathcal{D}[z_j : M_j]\)  
   4

6. \(tr \notin \mathcal{D}[SYS]\)  
   3, 5
3.8 DEADLOCK FREEDOM

This result is a contradiction, thus establishing the theorem. □

3.8 Deadlock freedom

This section establishes conditions on controllers and B machines which are sufficient to ensure the deadlock freedom of our mobile system.

Deadlock can occur because of two different reasons during the execution of the system. One of these reasons is in CSP part and the other one is in B part as described below:

1. The parallel combination of CSP controllers is not deadlock free. Therefore, deadlock occurs during the system execution while controllers are working in parallel.

2. During the execution, we achieve a situation in which one or more operations are the only possible events which can happen next but the execution of all of them is blocked. These operations can either belong to one single B machine or can belong to different B machines in the system. In other words, deadlock can occur when all the machines whose one or more operations are the only next possible events are not able to execute those operations. The execution of an operation can be blocked because of one of the reasons below:
   - The guard is false so the execution is blocked.
   - The guard is true, but there is blocking statement in the body of the operation which is blocked and so the execution is blocked.
   - The operation is pre-conditioned, so there is no guard to be false, but there is blocking statement in the body of the operation which is blocked while the precondition is true and so the execution is blocked.

According to the first rule presented in Section 3.3 for B machines, the body of operations in B machines are not allowed to contain blocking statements. Therefore, if all B machines in the system contain only pre-conditioned operations, then the B machines do not contribute to any deadlocking behaviour. This is because preconditions do not block, and we are not allowing blocking statements within the bodies of operations.

According to all explanations above, the theorem below is presented for establishing the deadlock freedom of a system in which the CSP part is deadlock free and the operations in all B machines are pre-conditioned. In addition, the theorem is based on the fact that the system should have already been proved to be
divergence free. The reason for adding this condition is that if a divergent state is reached, there is no guarantee about the behaviour of the system afterwards. A divergent system may refuse all events and stay in the divergent state. In other words, the set of all possible events might be the refusal of the system. If the system has divergence, the deadlock may always occur after the system reaches divergent state. So, a non divergence free system may always have deadlock. Therefore, there is no point to check deadlock freedom of a non divergence free system because it always may have deadlock after the divergent state. As a result, we concentrate here to prove the deadlock freedom of a mobile system which is divergence free. The failures of a system consists of both stable failures and the failures resulted by divergences. Thus, if the system is divergence free, then the failures are the same as the stable failures. We have already proved Theorem 3 for establishing the divergence freedom in mobile combined communicating systems. By using Theorem 3, we are able to establish the divergence freedom of the system.

**Theorem 4.** Supposing LOOP₁, LOOP₂, ..., LOOPₙ are the CSP controllers and M₁, M₂, ..., Mₘ are the B machines in a mobile combined communicating system such that all B machines contain only pre-conditioned operations and the system is divergence free. If the parallel combination of controllers is deadlock free, then the whole system, SYS = LOOP₁ || LOOP₂ || ... || LOOPₙ || z₁ : M₁ || z₂ : M₂ || ... || zₘ : Mₘ, is deadlock free where z₁, z₂, ..., zₘ are the machine references for B machines M₁, M₂, ..., Mₘ respectively.

**Proof.** We prove this theorem by contradiction. We assume that the system is not deadlock free.

So: \( \exists tr \in traces(SYS) \bullet (tr, \Sigma) \in failures(SYS) \)

We introduce tr₁, ..., trₙ, trM₁, ..., trMₘ, X₁, ..., Xₙ, XM₁, ..., XMₘ as described below:

\[
tr_1 = tr \upharpoonright \chi(LOOP_1), s(LOOP_1) \\
\vdots \\
tr_n = tr \upharpoonright \chi(LOOP_n), s(LOOP_n) \\
tr_{M_1} = tr \upharpoonright z_1 \cdot M_1 \\
\vdots \\
tr_{M_m} = tr \upharpoonright z_m \cdot M_m
\]
3.9 REFINEMENT

One important issue which is considered in our work is to allow the refinements of the components into our framework. The intention is to be able to substitute a component by its refinement in such a way that the substitution does not violate the system consistency properties. In this section we prove that if we have a mobile combined communicating system and this system is divergence free and deadlock free, then if we use a refinement of B machines or CSP controllers instead of them in the system, the system remains divergence free and deadlock free. This enables us to use a refinement of a component instead of the component in the system.

\[
\begin{align*}
(tr_1, X_1) &\in \text{failures}(\text{LOOP}_1) \\
\vdots \\
(tr_n, X_n) &\in \text{failures}(\text{LOOP}_n) \\
(tr_{M_1}, X_{M_1}) &\in \text{failures}(M_1) \\
\vdots \\
(tr_{M_m}, X_{M_m}) &\in \text{failures}(M_m)
\end{align*}
\]

\[X_1 \cup \ldots \cup X_n \cup X_{M_1} \cup \ldots \cup X_{M_m} = \Sigma\]

1. \(X_1 \cup \ldots \cup X_n \cup X_{M_1} \cup \ldots \cup X_{M_m} = \Sigma\) Contradictory assumption

2. \(\forall 1 \leq i \leq m \cdot X_{M_i} = \emptyset\) Initial assumption

3. \(X_{M_1} \cup \ldots \cup X_{M_m} = \emptyset\) 2

4. \(\exists a \in \Sigma \cdot a \notin X_1 \cup \ldots \cup X_n\) Initial assumption

5. \(\exists a \in \Sigma \cdot a \notin X_1 \cup \ldots \cup X_n \cup X_{M_1} \cup \ldots \cup X_{M_m}\) 4, 3

6. \(X_1 \cup \ldots \cup X_n \cup X_{M_1} \cup \ldots \cup X_{M_m} \neq \Sigma\) 5

This result is a contradiction, thus establishing the theorem.

\[\square\]

3.9 Refinement

One important issue which is considered in our work is to allow the refinements of the components into our framework. The intention is to be able to substitute a component by its refinement in such a way that the substitution does not violate the system consistency properties. In this section we prove that if we have a mobile combined communicating system and this system is divergence free and deadlock free, then if we use a refinement of B machines or CSP controllers instead of them in the system, the system remains divergence free and deadlock free. This enables us to use a refinement of a component instead of the component in the system.
3.9.1 Refinements of CSP controllers

In this section, we prove that \( \text{LOOP}' \) a refinement of a CSP controller \( \text{LOOP} \) can be used instead of \( \text{LOOP} \) in the system. To achieve this, we should prove that all behaviour \( \text{LOOP}' \) can do in the system, \( \text{LOOP} \) is also able to do. Therefore, the substitution does not introduce any new behaviour in the system. Each CSP controller is working in parallel with other CSP controllers and with B machines in the system during the execution. Thus, we should prove that \( \text{LOOP}' \) does not introduce any new behaviour while working in parallel with other CSP controllers and with B machines in the system.

Lemma 23. If \( P \) and \( Q \) are two CSP processes, then:

\[
P \sqsubseteq_P P' \\
\chi(P') = \chi(P) \\
s(P') = s(P)
\implies traces(P' \parallel Q) \subseteq traces(P \parallel Q)
\]

Proof. We introduce \( t_r \) as an arbitrary trace of \( P' \parallel Q \).

1. \( t_r \in traces(P' \parallel Q) \) Introducing arbitrary \( t_r \)

2. \( t_r \parallel \chi(P'), s(P') \in traces(P') \land t_r \parallel \chi(Q), s(Q) \in traces(Q) \) 1

3. \( traces(P') \subseteq traces(P) \) \( P \sqsubseteq_P P' \)

4. \( t_r \parallel \chi(P'), s(P') \in traces(P) \) 2, 3

5. \( t_r \parallel \chi(P'), s(P') = t_r \parallel \chi(P), s(P) \) \( \chi(P') = \chi(P) \land s(P') = s(P) \)

6. \( t_r \parallel \chi(P), s(P) \in traces(P) \) 4, 5

7. \( t_r \parallel \chi(P), s(P) \in traces(P) \land t_r \parallel \chi(Q), s(Q) \in traces(Q) \) 2, 6

8. \( t_r \in traces(P \parallel Q) \) 7

9. \( traces(P' \parallel Q) \subseteq traces(P \parallel Q) \) 1, 8

\( \blacksquare \)
Lemma 24. If \( P \) and \( Q \) are two CSP processes, then:

\[
\begin{align*}
P \subseteq_F P' \\
\chi(P') &= \chi(P) \\
s(P') &= s(P)
\end{align*}
\]

\[\implies \text{failures}(P' \parallel Q) \subseteq \text{failures}(P \parallel Q)\]

Proof. We introduce \((tr, X)\) as an arbitrary failure of \( P' \parallel Q \).

1. \((tr, X) \in \text{failures}(P' \parallel Q)\) \hspace{1cm} \text{Introducing arbitrary \((tr, X)\)}

2. \[\left( ((tr \upharpoonright \chi(P'), s(P')), X_{P'}) \in \text{failures}(P') \wedge ((tr \upharpoonright \chi(Q), s(Q)), X_Q) \in \text{failures}(Q) \wedge X_{P'} \cup X_Q = X \right) \]

3. \text{failures}(P') \subseteq \text{failures}(P) \hspace{1cm} P \subseteq_F P'

4. \[\left( ((tr \upharpoonright \chi(P'), s(P')), X_{P'}) \in \text{failures}(P) \right) \hspace{1cm} 1, \text{Definition 9}

5. \[\chi(P') = \chi(P) \wedge s(P') = s(P)\]

6. \[\left( ((tr \upharpoonright \chi(P), s(P)), X_{P'}) \in \text{failures}(P) \right) \hspace{1cm} 4, 5

7. \[\left( ((tr \upharpoonright \chi(P), s(P)), X_{P'}) \in \text{failures}(P) \wedge ((tr \upharpoonright \chi(Q), s(Q)), X_Q) \in \text{failures}(Q) \wedge X_{P'} \cup X_Q = X \right) \]

8. \((tr, X) \in \text{failures}(P \parallel Q)\) \hspace{1cm} 7, \text{Definition 9}

9. \text{failures}(P' \parallel Q) \subseteq \text{failures}(P \parallel Q) \hspace{1cm} 1, 8

\[\square\]

The next corollary follows immediately from Lemma 23 and Lemma 24.

Corollary 7. If \( P \) and \( Q \) are two CSP processes, then:

\[
\begin{align*}
P \subseteq_F P' \\
\chi(P') &= \chi(P) \\
s(P') &= s(P)
\end{align*}
\]

\[\implies P \parallel Q \subseteq_F P' \parallel Q\]

Now, we are able to present the following corollary.
Corollary 8. If LOOP\(_1\), LOOP\(_2\), ..., LOOP\(_n\) are the CSP controllers and M\(_1\), M\(_2\), ..., M\(_m\) are the B machines in a mobile combined communicating system, then: \(\forall 1 \leq i \leq n\) •

\[
\begin{align*}
\text{LOOP}_i & \subseteq_F \text{LOOP}'_i \\
\chi(\text{LOOP}'_i) &= \chi(\text{LOOP}_i) \\
s(\text{LOOP}'_i) &= s(\text{LOOP}_i)
\end{align*}
\]

\[
\text{LOOP}_i \parallel (\bigcup_{j \in \{1, ..., n\} - \{i\}} \text{LOOP}_j) \parallel (\bigcup_{j \in \{1, ..., m\} - \{i\}} z_j : M_j) \subseteq_F \\
\text{LOOP}'_i \parallel (\bigcup_{j \in \{1, ..., n\} - \{i\}} \text{LOOP}_j) \parallel (\bigcup_{j \in \{1, ..., m\} - \{i\}} z_j : M_j)
\]

where \(z_1, z_2, ..., z_m\) are the machine references for B machines M\(_1\), M\(_2\), ..., M\(_m\) respectively.

Proof. This corollary follows from Corollary 7 and the fact that B machines are understood in our system as CSP processes.

\[\square\]

Theorem 5. If a mobile combined communicating system is divergence free and deadlock free, then any refinement LOOP' of a CSP controller LOOP which has the same static channels as LOOP and owns the same machines initially as LOOP can be a substitute of LOOP in the system and the new system is guaranteed to be divergence free and deadlock free.

Proof. This theorem follows from Corollary 8.

\[\square\]

3.9.2 Refinements of B machines

As the refinement of a B machine is created in B environment, in this section we prove that a refinement machine created in B environment is also a failure refinement of its abstract machine in CSP environment. In order for a refinement machine M' to be a failure refinement of an abstract machine M, the traces and the failures of M' must be the subset of traces and failures of M respectively.

At the beginning of this chapter, we described that a B machine can be seen as a CSP process if the body of its operations do not contain SELECT or ANY (or Let which is a special kind of ANY ) statements. This has been introduced in Section 3.3 as the first rule for B machines. Therefore, in order for a refinement machine to be used in our architecture, the body of its operations must not contain SELECT or ANY (or Let which is a special kind of ANY ) statements.

As we mentioned at the beginning of Section 3.9, we are considering a mobile combined communicating system which has already been established to be divergence
free and deadlock free. According to our deadlock freedom verification strategy presented in Section 3.8, deadlock freedom is established in a system whose B machines contain only pre-conditioned operations. This results in the fact that all abstract B machines in the established deadlock free system contain only pre-conditioned operations which also means that their refinement machines contain only pre-conditioned operations. Continuing from now, we will concentrate our proof on this. In addition, according to the second rule presented in Section 3.3 for B machines, there must be no pre statement in the body of operations in any refinement machine which is used in our system.

When an abstract machine $M$ is refined, the specification of its operations may change in its refinement machine $M'$. As a result, the weakest precondition for a sequence of operations to achieve a post condition $Q$ may be different in $M$ and $M'$. Therefore, while using $wp$ formulas, we should indicate that a $wp$ is calculated according to the specification of operations in which one of the machines: $M$ or $M'$. Thus, we introduce the definition below which is used to differ between the $wp$ formulas in $M$ and $M'$ while using them in our proofs in this section.

**Definition 14.** If $M$ is a B machine with initialisation clause init and $tr$ is a finite sequence of operations, then $wp(tr, Q)_M$ and $wp(init; tr, Q)_M$ represent the weakest precondition which is calculated according to the specification of operations in $M$.

One requirement for a machine to be a refinement of an abstract machine is that whenever the abstract and refinement machines are in linked states (the states that meet both the invariant of abstract machine and the invariant of refinement machine), then if the precondition of an operation is true in the abstract machine, it is guaranteed that the precondition of that operation is also true in the refinement machine. In other words, in any states that the abstract machine and its refinement can jointly be in, the precondition of an operation in a refinement machine should be true if the operation is within its abstract precondition. This is expressed formally as $I \land J \land P \Rightarrow P'$ in [57] where $I$ and $J$ are the invariant of the abstract machine and the refinement machine respectively, and $P$ and $P'$ are the precondition of an operation in the abstract machine and in the refinement machine respectively. More details can be found in [57] Chapter 14. Based on this fact, we are able to present the following lemma.

**Lemma 25.** Supposing $M'$ is a refinement machine of abstract machine $M$ and init$_{M'}$ and init$_M$ are the initialisation clauses of $M'$ and $M$ respectively. If $tr$ is a finite sequence of operations in $M'$, then:

$$wp(init_M; tr, true)_M \Rightarrow wp(init_{M'}; tr, true)_{M'}$$
Proof. We prove this lemma by using the Principle of Induction on \( tr \).

**Base case:** \( tr = () \)

This case is easily proved as \( \text{wp}(\text{init}_M; true)_M = \text{wp}(\text{init}_{M'}; true)_{M'} = true \)

**Inductive case:** \( tr' \neq (op) \)

Assume Inductive Hypothesis for \( tr' \). \( tr' \) is a finite sequence of operations and \( op \) is an operation. We assume that the operation \( op \) is specified as \( op = \text{PRE } P \text{ THEN } S \text{ END} \) in \( M \) and as \( op = \text{PRE } P' \text{ THEN } S' \text{ END} \) in \( M' \). We also assume that \( I \) is the invariant of the abstract machine \( M \) and \( J \) is the invariant of the refinement machine \( M' \).

\[
\begin{align*}
1. & \quad \text{wp}(\text{init}_M; tr', true)_M \Rightarrow \text{wp}(\text{init}_{M'}; tr', true)_{M'} & \text{Induction assumption} \\
2. & \quad \text{wp}(\text{init}_M; tr', true)_M \Rightarrow \text{wp}(\text{init}_M; tr', I)_M & \text{Internal consistency} \\
3. & \quad \text{wp}(\text{init}_{M'}; tr', true)_{M'} \Rightarrow \text{wp}(\text{init}_{M'}; tr', J)_{M'} & \text{Internal consistency} \\
4. & \quad \text{wp}(\text{init}_M; tr', true)_M \Rightarrow \text{wp}(\text{init}_M; tr', J)_M & 1, 3 \\
5. & \quad \text{wp}(\text{init}_M; tr', P)_M \Rightarrow \text{wp}(\text{init}_M; tr', true)_M & P \Rightarrow true \\
6. & \quad \text{wp}(\text{init}_M; tr', P)_M \Rightarrow \text{wp}(\text{init}_{M'}; tr', P')_{M'} & 5, 2, 4, \\
\text{I} & \land J & \land P \Rightarrow P' \\
7. & \quad \text{wp}(\text{init}_M; tr'; op, true)_M = \text{wp}(\text{init}_M; tr'; \text{wp}(op, true)_M)_M & \text{wp in } M \\
8. & \quad \text{wp}(op, true)_M = P & \text{wp}(S, true) = true \\
9. & \quad \text{wp}(\text{init}_M; tr'; op, true)_M = \text{wp}(\text{init}_M; tr', P)_M & 7, 8 \\
10. & \quad \text{wp}(\text{init}_M; tr'; op, true)_M \Rightarrow \text{wp}(\text{init}_{M'}; tr', P')_{M'} & 9, 6 \\
11. & \quad \text{wp}(op, true)_{M'} = P' & \text{wp}(S', true) = true \\
12. & \quad \text{wp}(\text{init}_M; tr'; op, true)_M \Rightarrow \text{wp}(\text{init}_{M'}; tr'; \text{wp}(op, true)_{M'})_{M'} & 10, 11
\end{align*}
\]


3.9. REFINEMENT

Supposing $M'$ is a refinement machine of abstract machine $M$ and $init_{M'}$ and $init_M$ are the initialisation clauses of $M'$ and $M$ respectively. If $tr$ is a finite sequence of operations in $M'$, then:

$$wp(init_M; tr', false)_M \Rightarrow wp(init_{M'}; tr, false)_{M'}$$

Proof. We prove this lemma by using the Principle of Induction on $tr$.

Base case: $tr = ()$

This case is easily proved as $wp(init_M, false)_M \Rightarrow wp(init_{M'}; tr', false)_{M'}$

Inductive case: $tr' \subseteq \langle op \rangle$

Assume Inductive Hypothesis for $tr'$. $tr'$ is a finite sequence of operations and $op$ is an operation. We assume that the operation $op$ is specified as $op = PRE P \ THEN \ S \ END$ in $M$ and as $op = PRE P' \ THEN \ S' \ END$ in $M'$.

1. $wp(init_M; tr'; op, false)_M = wp(init_{M'}; tr', wp(op; M')_M$

2. $wp(op; false)_M = false \Rightarrow wp(S, false) = false$

3. $wp(init_M; tr'; op, false)_M = wp(init_{M'}; tr', false)_{M'}$

4. $wp(init_M; tr', false)_M \Rightarrow wp(init_{M'}; tr', false)_{M'}$

Induction assumption

5. $wp(init_M; tr'; op, false)_M \Rightarrow wp(init_{M'}; tr', false)_{M'}$

$\square$

Lemma 26.
The next three lemmas state that the traces, divergences and failures of a refinement machine are always the subset of the traces, divergences and failures of its abstract machine respectively.

Lemma 27. If $M'$ is a refinement machine of abstract machine $M$, then:

$$\text{traces}(M') \subseteq \text{traces}(M)$$

Proof. This lemma follows from Lemma 26. We introduce $tr$ as an arbitrary finite sequence of operations in $M'$.

1. $wp(\text{init}_M; tr, false)_M \Rightarrow wp(\text{init}_{M'}; tr, false)_{M'}$  
   \hspace{1cm} \text{Lemma 26}

2. $\neg wp(\text{init}_{M'}; tr, false)_{M'} \Rightarrow \neg wp(\text{init}_M; tr, false)_M$

3. $wp(\text{init}_{M'}; tr, true)_{M'} \Rightarrow wp(\text{init}_M; tr, true)_M$
   \hspace{1cm} 2, Definition 2

4. $tr \in \text{traces}(M') \Rightarrow tr \in \text{traces}(M)$
   \hspace{1cm} 3, Definition 3

5. $\text{traces}(M') \subseteq \text{traces}(M)$
   \hspace{1cm} 4, $tr$ introduced arbitrarily

$\square$
Lemma 28. If $M'$ is a refinement machine of abstract machine $M$, then:
$\mathcal{D}[M'] \subseteq \mathcal{D}[M]$

Proof. This lemma follows from Lemma 25. We introduce $tr$ as an arbitrary finite sequence of operations in $M'$.

1. $wp(init_M; tr, true)_M \Rightarrow wp(init_{M'}; tr, true)_{M'}$  
   Lemma 25

2. $\neg wp(init_M; tr, true)_M \Rightarrow \neg wp(init_{M'}; tr, true)_{M'}$

3. $wp(init_M; tr, false)_M \Rightarrow \neg wp(init_{M'}; tr, false)_{M'}$  
   2, Definition 2

4. $tr \in \mathcal{D}[M'] \Rightarrow tr \in \mathcal{D}[M]$  
   3, Definition 5

5. $\mathcal{D}[M'] \subseteq \mathcal{D}[M]$  
   4, $tr$ introduced arbitrarily

The next lemma follows from Lemma 27 and Lemma 28.

Lemma 29. If $M'$ is a refinement machine of abstract machine $M$, then:
failures($M'$) $\subseteq$ failures($M$)

Proof. As $M$ and $M'$ contain only pre-conditioned operations with no guarded statements in their body, whenever one of their operations is called inside its precondition in a stable state, they do not refuse the execution of that operation. In other words, all operations will always be executed in $M$ and $M'$ if they are in a stable state.

1. failures($M$) = $\{(tr, \{\}) | tr \in traces(M)\} \cup$ $\{(tr, X) | tr \in \mathcal{D}[M] \land X \subseteq cM\}$  
   Initial assumption

2. failures($M'$) = $\{(tr, \{\}) | tr \in traces(M')\} \cup$ $\{(tr, X) | tr \in \mathcal{D}[M'] \land X \subseteq cM'\}$  
   Initial assumption

3. traces($M'$) $\subseteq$ traces($M$)  
   Lemma 27

4. $\{(tr, \{\}) | tr \in traces(M')\} \subseteq$ $\{(tr, \{\}) | tr \in traces(M)\}$  
   3

5. $\mathcal{D}[M'] \subseteq \mathcal{D}[M]$  
   Lemma 28
Corollary 9. If $M'$ is a refinement machine of abstract machine $M$, then:

$M \subseteq_P M'$

Corollary 10. If $\text{LOOP}_1, \text{LOOP}_2, \ldots, \text{LOOP}_n$ are the CSP controllers and $M_1, M_2, \ldots, M_m$ are the $B$ machines in a mobile combined communicating system, then:

\[ \forall 1 \leq i \leq m \cdot M_i \subseteq_P M_i' \Rightarrow \]
\[ \text{LOOP}_1 \parallel \text{LOOP}_2 \parallel \cdots \parallel \text{LOOP}_n \parallel z_i : M_i \parallel (\|_{j \in \{1, \ldots, m\} \setminus \{i\}} z_j : M_j) \subseteq_P \]
\[ \text{LOOP}_1 \parallel \text{LOOP}_2 \parallel \cdots \parallel \text{LOOP}_n \parallel z_i : M_i' \parallel (\|_{j \in \{1, \ldots, m\} \setminus \{i\}} z_j : M_j) \]

where $z_1, z_2, \ldots, z_m$ are the machine references for $B$ machines $M_1, M_2, \ldots, M_m$ respectively.

Proof. This corollary follows from Corollary 7 and the fact that $B$ machines are understood in our system as CSP processes.

\[ \square \]

Theorem 6. If a mobile combined communicating system is divergence free and deadlock free, then any refinement machine $M'$ of an abstract machine $M$ which satisfies the first and second rules presented in Section 3.3 for $B$ machines can be a substitute of $M$ in the system and the new system is guaranteed to be divergence free and deadlock free.

Proof. This theorem follows from Corollary 9 and Corollary 10.

\[ \square \]

3.10 Discussion

In this chapter we introduced our Mobile CSP $|| B$ framework. In the previous static version of CSP $|| B$, for each operation in a $B$ machine, there is one channel between that machine and its controller. However in our work, a machine's reference is the only channel through which a CSP controller and that $B$
3.10. DISCUSSION

machine can interact with each other. Despite static CSP $|$ $B$ in which each controller is dedicated for one B machine, we designed our framework in such a way that controllers are able to work with different machines during the execution. Controllers exchange machines between each other by exchanging the machine references. This results in two facts about our system architecture:

1. In Mobile CSP $|$ $B$, we have introduced mobile channels: machine references are mobile channels and they can move around the system during the execution.

2. In Mobile CSP $|$ $B$, mobile channels (machine references) are allowed to be passed between processes through static channels (control points) in the system. In other words, machine references are channels but they are also treated as value and they can be passed between processes through control point channels in the system.

In addition, in this chapter we defined and verified the conditions which guarantee the divergence freedom and deadlock freedom consistency of the systems specified and designed in Mobile CSP $|$ $B$. However, the deadlock freedom verification presented in this chapter has been focused on the systems in which the operations of B machines are all pre-conditioned. The CLI preserver definition which we introduced in this chapter as Definition 11 has a key role in consistency verification of the systems. In our consistency verification strategy, all CSP controllers must be CLI preserver which means that the conditions in Definition 11 must hold for all CSP controllers in the system.

Refinements of the components are allowed into our Mobile CSP $|$ $B$ framework. In Section 3.9, we proved that a refinement of a component (either a CSP controller or a B machine) can be used instead of the component in the system and the substitution does not have any effect on the system consistency properties.

In Mobile CSP $|$ $B$, we can also design systems in which control points carry more than one machine reference. In other words, controllers can exchange more than one machine between each other at once and through one single control point channel. However, it should be remembered that all machines are passed in one direction from one sender to one receiver. In consistency verification, for this kind of control point we must assign an assertion on the state of all the machines whose reference is passed along that control point. For instance, assume a system in which $cp!z!w$ is in the body of controller $LOOP_1$ and $cp?x?y$ is in the body of controller $LOOP_2$. In this system, whenever these two controllers synchronise on $cp$, two machines are passed from $LOOP_1$ to $LOOP_2$ at the same time and through one single control point channel $cp$. To verify the consistency of this system, for the control point $cp$ we should assign an assertion on the state of both machines.
B machines in Mobile CSP || B can be dropped from the system during the execution. This can be used to design systems in which some B machines might not be used anymore or should not be used anymore during the execution and so they can be dropped from the system. We are able to specify it in the body of CSP controllers. Whenever a machine should be dropped from the system, the variable referring to its reference should be dropped from the set of free variables in its controller. For instance, a CSP controller \texttt{LOOP} can be specified as below:

\begin{align*}
\text{LOOP} &= P_1(z, w) \\
P_1(z, w) &= z.\text{op} \rightarrow P_2(w) \\
P_2(w) &= w.\text{op} \rightarrow \text{cp?}x \rightarrow P_1(w, x)
\end{align*}

In the CSP controller \texttt{LOOP} above, process \texttt{P1} calls the operation \texttt{op} of a \texttt{B} machine with machine reference \texttt{z} and then \texttt{z} does not appear in the free variables in \texttt{P2}. From now on, \texttt{LOOP} does not have access to \texttt{z}. Also, no other controller in the system has received \texttt{z}, so no other controller can access to that machine in the system. This means that the \texttt{B} machine with machine reference \texttt{z} is no longer accessible in the system.

Our mobile architecture enables us to design systems in which CSP controllers can terminate during the system execution. This can happen when \texttt{SKIP} appears in the body of a process. \texttt{SKIP} can be used in our framework to design systems in which controllers can finish their work during the system execution or even leave the system after finishing their work. It can be assumed that whenever a controller terminates, the communication channels and control point channels between that controller and other controllers are automatically deleted from the system. This is because other controllers cannot synchronise with that controller anymore on their common events so those common events can never happen again. This results in the fact that other controllers cannot communicate with that controller anymore on their common communication channels and they cannot also exchange machines with that controller anymore on their common control point channels.

Furthermore, it can be assumed that whenever a controller terminates, its machines will be dropped from the system with their machine reference channel. The reason is that no other controller can ever access those machines. In other words, other controllers can never receive any of those machines from that controller as they cannot work with that controller anymore. This results in the fact that whenever a controller terminates, if it owns some machines, its machines will be no longer accessible in the system. If it is required to design a system in which all machines should remain in the system at all times during the system execution, we can specify our CSP controllers in such a way that they give all their machines to other controllers before termination.
Chapter 4

Case Study

In this chapter, we present a case study within the Mobile CSP $|$ $B$ framework. The case study is a flight tickets sale system which is described in Section 4.1. It presents the usage of Mobile CSP $|$ $B$ architecture in designing and developing peer-to-peer networks. We first provide a Mobile CSP $|$ $B$ model of a flight tickets sale system and then we verify the consistency of our model by using the theorems proved in Chapter 3. The specifications of B machines and CSP part of the system are described in sections 4.2 and 4.3 respectively. Section 4.4 demonstrates how the mobile CSP specification in Section 4.3 can be coded into a standard CSP specification. Section 4.5 presents the processes of divergence freedom verification of our system. Deadlock freedom is verified in section 4.6. Finally, we present some conclusions on the case study in Section 4.7.

A simplified version of this case study was presented in [72].

4.1 Flight tickets sale system

This case study is designed as a flight tickets sale system in which tickets of different flights are sold or cancelled by agencies.

The system contains a number of sell agencies which sell tickets of different flights to customers, and it contains one Return office which cancels customers' tickets. Sell agencies can only sell flight tickets and they are not able to cancel any tickets of the flights. The Return office is only responsible for cancelling tickets and it is not able to sell any flight tickets.

If sell agencies or the Return office want to sell or cancel a ticket of a flight, they should have access to the information of that flight. Otherwise, they are not
able to sell or cancel any tickets. This description of the system makes it clear what should be modelled as the B machines and what should be modelled as the CSP controllers in the system. The B machines in our system are the individual flights. They manage all the booking information of the flights. So, each machine represents one of the flights in the system. Sell agencies and the Return office play the role of the CSP controllers in our system. The Return office is one CSP controller and each Sell agency is also one CSP controller in the system.

Each flight machine is given a unique machine reference which is the channel used by sell agencies and the Return office to contact and communicate with that flight machine. A Sell agency or the Return office can sell or cancel a ticket of a flight only if they own that machine's reference. If they want to sell or cancel a ticket of a flight but they do not have that machine's reference, they ask others in the system to give them that machine's reference. Receiving a machine's reference means receiving the channel of communication with that machine. In other words, the receiver becomes the owner of that machine. So, after receiving the machine's reference, the Sell agency or the Return office can work with the machine to sell or cancel tickets.

When a customer wants to buy a ticket of a flight from a Sell agency, if the Sell agency already owns that flight machine, it sells a ticket of that flight to that customer. If the Sell agency does not own that machine, it asks other Sell agencies and the Return office to pass the machine to it. If it receives the machine, it can then sell a ticket of the machine to that customer.

When a customer wants to cancel a ticket of a flight and asks the Return office for cancellation, if the Return office owns that flight machine, it cancels that customer's ticket. But if it does not have that machine, it asks sell agencies to pass the machine to it. If it receives the machine, it then cancels the customer's ticket.

Sell agencies and the Return office behave in such a way so that they do not keep the machines when they cannot use them anymore. A full machine cannot be used anymore by sell agencies as all the tickets have already been sold and there is no more ticket available to be sold next. So, if a flight owned by a Sell agency is full after selling a ticket, the Sell agency passes that full machine to the Return office. On the other hand, an empty machine cannot be used anymore by the Return office as there is no sold ticket in the machine to be cancelled next. So, if a flight owned by the Return office is empty after cancelling a ticket, the Return office passes that empty machine to one of the sell agencies. In other words, a Sell agency does not keep full machines with itself and the Return office does not keep empty machines with itself.

A Sell agency asks for a machine in order to sell a ticket of that flight. So, the machine it receives must not be already full. On the other hand, the Return office
4.2 DESIGN AND SPECIFICATION OF B MACHINES

Asks for a machine in order to cancel a ticket of that flight. So, the machine it receives must not be empty. The receiver does not have any control on the state of the machine it receives. So, our system should be designed such that these conditions are always ensured by the sender. If a Sell agency receives a request from the Return office for passing one of its machines, it should first check the current state of the machine to ensure it is not empty. It then passes that machine to the Return office only if the machine is not empty. If a Sell agency receives a request from another Sell agency for passing one of its machines, it passes that machine to the requester only if the machine is not full. On the other hand, if the Return office receives a request for passing one of its machines, it should first check the current state of the machine to ensure it is not full. It then passes that machine to the requester Sell agency only if the machine is not full.

We introduce a parameter $i$ which is used to identify each Sell agency. Also, a set $S$ is introduced for each Sell agency and for the Return office which contains all the machine references owned by the process. As a result, $\text{SellAgency}(i, S)$ represents the Sell agency $i$ which currently owns the machine references in $S$, and $\text{ReturnOffice}(S)$ represents the Return office which currently owns the machine references in $S$.

For the purposes of our case study, we will use two flight machines: $\text{flight1}$ and $\text{flight2}$, two sell agencies: $\text{SellAgency1}$ and $\text{SellAgency2}$, and a Return office. We also assume that at the beginning of the system execution, each Sell agency owns one of the flight machines. Therefore, the whole system is as below:

$$\text{ReturnOffice}() || \text{SellAgency}(1, \{\text{mr1}\}) || \text{SellAgency}(2, \{\text{mr2}\}) || \text{mr1} : \text{flight1} || \text{mr2} : \text{flight2}$$

where $\text{mr1}$ and $\text{mr2}$ are the machine references of machines $\text{flight1}$ and $\text{flight2}$ respectively. The initial configuration of the system is shown in Figure 4.1.

4.2 Design and specification of B machines

Each B machine manages the information of one flight in the system. The structure of all flights in our system are the same so the B machines have the same specification but with different names.

Each B machine contains the information about that particular flight such as: the number of seats of the flight, and the number of tickets which have already been sold. It also contains some operations for state transitions in the flight such as selling or cancelling tickets and some other operations for finding out the current state of the flight such as whether it is empty or full.
SETS
A set RESPONSE is introduced whose elements are used by the machine as the messages which are sent to the controllers.

SETS RESPONSE = \{Full, Empty, Available, YES, NO, IncorrectInput\}

CONSTANTS and PROPERTIES
Each B machine contains 2 constants: Passport and seats. Passport is a set defining all valid passport numbers which can be accepted as the customers' passport number. seats is a number representing the total number of seats in the flight to be sold to customers.

CONSTANTS Passport, seats

PROPERTIES Passport \subseteq N_1 \&
seats \in N_1

VARIABLES and INVARIANT
There are 2 variables in each B machine: customer and sold. customer is a set which records the passport number of people who have bought a ticket of the flight. sold is a number which represents the number of tickets which have already been sold.
4.2. DESIGN AND SPECIFICATION OF B MACHINES

VARIABLES\hspace{1em} customer, sold

INVARIANT\hspace{1em} customer \subseteq \text{Passport} \&
\hspace{1em} sold \in \mathbb{N}

INITIALISATION

Initially, the flight is empty. In other words, no ticket has been sold to anybody at the beginning of the execution. As a result, the INITIALISATION clause sets customer to empty set and it sets sold to 0.

\begin{align*}
\text{INITIALISATION} & \quad \text{customer} := \{\} \quad \mid \\
& \quad \text{sold} := 0
\end{align*}

OPERATIONS

Each B machine has 4 operations which are described below:

- **sell**: The operation receives a customer's passport number as an input to sell a ticket of the flight. As the precondition, the input must be a valid passport number and the flight must not be full. If the input passport number has not already been used to purchase a ticket, the operation sells a ticket and it also outputs a message informing its controller whether the machine becomes full after selling that ticket or there are still some unsold tickets available. If the input passport number has already been used to purchase a ticket, the operation outputs a message informing its controller that the passport number is an incorrect input.

\begin{verbatim}
response \leftarrow sell(pp) \equiv \\
\text{PRE} \quad pp \in \text{Passport} \&
\quad sold \neq seats \quad \text{THEN} \\
\hspace{1em} \text{IF} \ pp \in \text{Passport} - \text{customer} \\
\hspace{1em} \text{THEN} \\
\hspace{2em} \text{customer} := \text{customer} \cup \{pp\} \quad \mid \\
\hspace{2em} \text{sold} := \text{sold} + 1 \quad \mid \\
\hspace{2em} \text{IF} \ sold + 1 = seats \\
\hspace{2em} \text{THEN} \ \text{response} := \text{Full} \\
\hspace{2em} \text{ELSE} \ \text{response} := \text{Available} \\
\hspace{2em} \text{END} \\
\hspace{2em} \text{ELSE} \ \text{response} := \text{IncorrectInput} \\
\hspace{2em} \text{END} \\
\hspace{1em} \text{END} \quad \text{END;}
\end{verbatim}
- **cancel**: The operation receives a customer's passport number as an input to cancel the customer's ticket. As the precondition, the input must be a valid passport number and the flight must not be empty. If the input passport number has already been used to purchase a ticket, the operation cancels the customer's ticket and it also outputs a message informing its controller whether the machine becomes empty after cancelling that ticket or there are still some sold tickets in the flight machine. If the input passport number has not already been used to purchase a ticket, the operation outputs a message informing its controller that the passport number is an incorrect input.

\[
\text{response} \leftarrow \text{cancel}(pp) \equiv \\
\text{PRE} \\
pp \in \text{Passport} \quad \& \\
sold \neq 0 \\
\text{THEN} \\
\quad \text{IF } pp \in \text{customer} \\
\quad \text{THEN} \\
\quad \quad \text{customer} := \text{customer} - \{pp\} \quad \| \\
\quad \quad \text{sold} := \text{sold} - 1 \quad \| \\
\quad \quad \text{IF } \text{sold} - 1 = 0 \\
\quad \quad \text{THEN } \text{response} := \text{Empty} \\
\quad \quad \text{ELSE } \text{response} := \text{Available} \\
\quad \quad \text{END} \\
\quad \text{ELSE } \text{response} := \text{IncorrectInput} \\
\quad \text{END} \\
\text{END}; \\
\]

- **empty**: It is the query operation which checks to find out whether the flight is currently empty. It then informs the controller about the result of its search by outputting the message YES or NO.

\[
\text{response} \leftarrow \text{empty} \equiv \\
\text{BEGIN} \\
\quad \text{IF } \text{sold} = 0 \\
\quad \text{THEN } \text{response} := \text{YES} \\
\quad \text{ELSE } \text{response} := \text{NO} \\
\text{END} \\
\text{END}; \\
\]

- **full**: It is the query operation which checks to find out whether the flight is currently full. It then informs the controller about the result of its search by outputting the message YES or NO.
4.3 SYSTEM DESIGN AND SPECIFICATION IN MOBILE CSP

response ← full ≜
BEGIN
  IF sold = seats
  THEN response := YES
  ELSE response := NO
END
END

The full AMN specification of a flight machine is attached in Appendix A.

After specifying the flight machines in AMN, we used ProB to verify the internal consistency of our B machines and to explore the behaviour of their operations. As all B machines have the same structure in our system, a single B machine specification was checked, analysed and animated in ProB. The B machine was proved to be internally consistent and its behaviour was in accordance to what we expected it to be.

4.3 System design and specification in mobile CSP

In this section, we describe the CSP specification of our system according to our mobile architecture. The Return office and sell agencies are each specified as a CSP process which describes their behaviour in the system. All control points are introduced as channels in the CSP part of the system. The CSP specification also defines which control point is used to pass the flight machines between which controllers.

The connection between flight machines and controllers is described in this section. A unique machine reference is introduced for each flight machine. In addition, it is defined which controller is the controller of each machine at the beginning of the execution.

4.3.1 Data types

Four data types should be introduced in the system:

- Flights: introduces the flights which exist in the system.
- RESPONSE: contains the messages that sell agencies and the Return office may receive from the flight machines.
- Operations: represents the B operations with their possible inputs and outputs which may happen during the execution.
• **MR:** introduces the machine references in the system.

```plaintext
datatype Flights = flight1 | flight2
datatype RESPONSE = Full | Empty | Available | YES | NO | IncorrectInput
datatype Operations = sell.Passport.RESPONSE | cancel.Passport.RESPONSE |
                      empty.RESPONSE | full.RESPONSE
datatype MR = mr1 | mr2
```

### 4.3.2 Sets

There are two sets to be introduced:

- **Agencies:** introduces the number of the sell agencies which exist in the system.
- **Passport:** defines all valid passport numbers which can be accepted in the system as the customers’ passport number.

```plaintext
Agencies = {1, 2}
Passport = {1, 2, 3}
```

As we are going to use ProBE to execute the CSP part of the system and to check the behaviour of the CSP controllers, we use small sets of Agencies and Passport in our system. The system can be easily extended after the behaviour of the system is checked in its small version.

### 4.3.3 Function

A function `ref` is used to assign a unique machine reference for each machine in the system. By mapping each machine to a machine reference, the flight machines are given a unique machine reference which then can be used to communicate with the CSP processes.

```plaintext
ref (flight1) = mr1
ref (flight2) = mr2
```

### 4.3.4 Channels

- **mr1:** the machine reference of flight1 through which the machine's operations are called.
- **mr2:** the machine reference of flight2 through which the machine's operations are called.
• \( cp.i.j \mid i,j \in Agencies \): a control point channel used to pass the flight machines between sell agencies.

• \( dp.i \mid i \in Agencies \): a control point channel used to pass the flight machines from the Return office to sell agencies.

• \( ep.i \mid i \in Agencies \): a control point channel used to pass the flight machines from sell agencies to the Return office.

• \( \text{customerToBuy} \): a channel through which a customer asks a Sell agency to buy a ticket.

• \( \text{customerToCancel} \): a channel through which a customer asks the Return office to cancel a ticket.

• \( \text{request} \): a channel through which sell agencies ask each other for a machine.

• \( \text{ask} \): a channel through which the Return office asks the sell agencies for a machine.

• \( \text{require} \): a channel through which sell agencies ask the Return office for a machine.

• \( \text{sorry} \): a channel through which a Sell agency informs another Sell agency that it does not own the requested machine.

• \( \text{notHaveIt} \): a channel through which a Sell agency informs the Return office that it does not own the requested machine. Also, the Return office informs a Sell agency through this channel that it does not own the requested machine.

• \( \text{emptyMachine} \): a channel through which a Sell agency informs the Return office that the requested machine is currently empty.

• \( \text{fullMachine} \): a channel through which the Return office informs a Sell agency that the requested machine is currently full.

• \( \text{MachineIsFull} \): a channel through which a Sell agency informs another Sell agency that the requested machine is currently full.

\[
\begin{align*}
mr1 & : Operations \\
mr2 & : Operations \\
cp.i.j & \mid i,j \in Agencies : MR \\
dp.i & \mid i \in Agencies : MR \\
ep.i & \mid i \in Agencies : MR
\end{align*}
\]
4.3.5 Sell agencies

A Sell agency can perform the following behaviours repeatedly during the execution:

1. A customer wants to buy a flight ticket. The Sell agency is informed of which flight the customer wants to buy a ticket and it also receives the customer’s passport number. If the Sell agency owns that flight machine,
it sells one ticket of that flight to the customer. In the case that the flight is full after that sale, the Sell agency does not keep the machine anymore and it passes the full machine to the Return office. If the Sell agency does not have that flight machine, it performs one of the behaviours below:

- It asks the other Sell agency for that machine. There could be three responses. It either receives the machine from that Sell agency, or it receives a message from that Sell agency saying that the machine is full, or it receives a message from that Sell agency saying that it does not have that machine. In the latter case, it asks the Return office for that machine.
- It asks the Return office for that machine. There could be three responses. It either receives the machine from the Return office, or it receives a message from the Return office saying that the machine is full, or it receives a message from the Return office saying that it does not have that machine. In the latter case, the Sell agency asks the other Sell agency for that machine.

2. The Sell agency receives a request from the other Sell agency for a flight machine. Depending on the requested machine, the Sell agency performs one of the behaviours below:

- If the machine is the Sell agency's full machine, in other words the Sell agency is waiting to pass it to the Return office, the Sell agency sends a message to the requester Sell agency saying that the machine is full and it does not send the machine to the requester Sell agency.
- If the Sell agency owns that machine and the machine is not its full machine, it gives the machine to the requester Sell agency.
- If it does not own that machine, it sends a message to the requester Sell agency saying that it does not have that machine.

3. The Sell agency receives a request from the Return office for a flight machine. Depending on the requested machine, the Sell agency performs one of the behaviours below:

- If the machine is the Sell agency's full machine, in other words the Sell agency has already been waiting to pass it to the Return office, the Sell agency gives that full machine to the Return office immediately and without checking the current state of the machine.
- If the Sell agency owns that machine and the machine is not its full machine, it first checks whether the machine is currently empty or not. If the machine is empty, it sends a message to the Return office saying that the machine is empty, and it does not send that machine to the
Return office. If the machine is not empty, then it gives that machine to the Return office.

- If it does not own that machine, it sends a message to the Return office saying that it does not have that machine.

4. The Sell agency receives an empty machine from the Return office.

A Sell agency gives a machine to another Sell agency only if the receiver has already asked the sender for that particular machine. If a Sell agency does not receive a request from the other Sell agency for a machine, it never passes any machine to the other Sell agency. In addition, a Sell agency gives a non full machine to the Return office, only if the Return office has already asked the Sell agency for that particular machine. If a Sell agency does not receive a request from the Return office for a non full machine, it does not pass any non full machine to the Return office. However, if a Sell agency sells a ticket of a flight and the flight becomes full after that sale, it passes that full machine to the Return office without having already received any request from the Return office for that machine.

On the other hand, a Sell agency receives a machine from another Sell agency only if it has already asked the sender for that particular machine. It never receives a machine from the other Sell agency without asking for that machine in advance. In addition, a Sell agency never receives a non empty machine from the Return office if it does not ask the Return office for that machine in advance. However, a Sell agency may receive an empty machine from the Return office without having already asked the Return office for that empty machine.

The structure of the sell agencies in our system are the same. So, they have the same specification but with different parameter $i$ and different set $S$. The specification of the Sell agency $i$ which owns the machine references in $S$ is described as below:

\[
SellAgency(i, S) = P_1(i, S)
\]

\[
P_1(i, S) = customerToBuy.i? flight?pn \rightarrow \text{if ref(flight) } \in S
\]

\[
\text{then } P_2(i, S, flight, pn)
\]

\[
\text{else } P_3(i, S, flight, pn)
\]

\[
\text{request?j!i?flight } \rightarrow \text{if ref(flight) } \in S
\]

\[
\text{then } P_4(i, S, j, flight)
\]

\[
\text{else (sorry.i.j } \rightarrow P_1(i, S))
\]
4.3. SYSTEM DESIGN AND SPECIFICATION IN MOBILE CSP

\[\text{ask.i?flight} \rightarrow \begin{cases} \text{if ref(flight)} \in S \\
\text{then P}_5(i, S, \text{flight}) \\
\text{else (notHaveIt.i} \rightarrow \text{P}_1(i, S)) \end{cases}\]

\[\text{dp.i?z} \rightarrow \text{P}_1(i, S \cup \{z\})\]

\[\text{P}_2(i, S, \text{flight}, pn) = \text{ref(flight)}.\text{sell!pn?resp} \rightarrow \begin{cases} \text{if resp} = \text{Full} \\
\text{then P}_6(i, S, \text{flight}) \\
\text{else P}_1(i, S) \end{cases}\]

\[\text{P}_3(i, S, \text{flight}, pn) = (\text{request.i?j!flight} \rightarrow \begin{cases} \text{((cp.j.i?w} \rightarrow \text{P}_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn))} \\
\text{if w} = \text{ref(flight)} \\
\text{then P}_2(i, S \cup \{w\}, \text{flight}, pn) \\
\text{else P}_3(i, S, \text{flight}, pn)) \end{cases}\]

\[\text{ask.i?f} \rightarrow \begin{cases} \text{if ref(f)} \in S \\
\text{then \text{cp.j.i}?ref(f)} \rightarrow \text{P}_6(i, S - \{\text{ref(f)}\}, \text{flight}, pn) \\
\text{else (sorry.i?j} \rightarrow \text{P}_3(i, S, \text{flight}, pn)) \end{cases}\]

\[\text{(require.i?flight} \rightarrow \begin{cases} \text{((dp.i?w} \rightarrow \text{P}_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn))} \\
\text{if w} = \text{ref(flight)} \\
\text{then P}_2(i, S \cup \{w\}, \text{flight}, pn) \\
\text{else P}_3(i, S, \text{flight}, pn)) \end{cases}\]

\[\text{(request?j!w?f} \rightarrow \begin{cases} \text{if ref(f)} \in S \\
\text{then \text{cp.j.i}?ref(f)} \rightarrow \text{P}_6(i, S - \{\text{ref(f)}\}, \text{flight}, pn) \\
\text{else (sorry.i?j} \rightarrow \text{P}_3(i, S, \text{flight}, pn)) \end{cases}\]

\[\text{(ask.i?f} \rightarrow \begin{cases} \text{if ref(f)} \in S \\
\text{then P}_6(i, S, \text{flight}, pn, f) \\
\text{else (notHaveIt.i} \rightarrow \text{P}_3(i, S, \text{flight}, pn)) \end{cases}\]

\[\text{dp.i?w} \rightarrow \begin{cases} \text{if w} = \text{ref(flight)} \\
\text{then P}_2(i, S \cup \{w\}, \text{flight}, pn) \\
\text{else P}_3(i, S \cup \{w\}, \text{flight}, pn) \end{cases}\]
$P_4(i, S, j, \text{flight}) = \text{cp.i.j!ref} (\text{flight}) \rightarrow P_1(i, S - \{\text{ref} (\text{flight})\})$

$P_6(i, S, \text{flight}) = \text{ref} (\text{flight}). \text{empty?resp} \rightarrow$

if resp = YES
then (emptyMachine.i \rightarrow P_1(i, S))
else (ep.i!ref (fligt) \rightarrow P_1(i, S - \{\text{ref} (\text{flight})\}))

$P_8(i, S, \text{flight}) = (\text{cp.i.j!ref} (\text{flight}) \rightarrow P_1(i, S - \{\text{ref} (\text{flight})\}))$

\(\Box\)

\(\text{request?i?j?f} \rightarrow \text{if ref(f) = ref(\text{flight})}
\)

then (MachineIsFull.i.j \rightarrow P_6(i, S, \text{flight}))
else $P_6(i, S, \text{flight}, j, f)$

\(\Box\)

\(\text{ask.i?f} \rightarrow \text{if ref(f) = ref(\text{flight})}
\)

then (\text{cp.i.lref(\text{flight})} \rightarrow P_1(i, S - \{\text{ref} (\text{flight})\}))
else $P_6(i, S, \text{flight}, f)$

\(\Box\)

\(\text{dp.i?w} \rightarrow P_8(i, S \cup \{w\}, \text{flight})\)

$P_7(i, S, j, \text{flight}, j, f) = \text{if ref}(f) \in S$

then (cp.i.j!ref(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight}))
else (sorry.i.j \rightarrow P_8(i, S, \text{flight}))

$P_8(i, S, \text{flight}, f) = \text{if ref}(f) \in S$

then $P_6(i, S, \text{flight}, f)$
else (notHave.f.i \rightarrow P_8(i, S, \text{flight}))

$P_9(i, S, \text{flight}, f) = \text{ref}(f). \text{empty?resp} \rightarrow$

if resp = YES
then (emptyMachine.i \rightarrow P_6(i, S, \text{flight}))
else (\text{cp.i!ref} (f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight}))
4.3. SYSTEM DESIGN AND SPECIFICATION IN MOBILE CSP

\[ P_{10}(i, S, \text{flight}, pn, f) = \text{ref}(f).\text{empty} \rightarrow \]
\[ \text{if } \text{resp} = \text{YES} \]
\[ \text{then } (\text{emptyMachine}.i \rightarrow P_3(i, S, \text{flight}, pn)) \]
\[ \text{else } (\text{sp}.i!\text{ref}(f) \rightarrow P_3(i, S - \{\text{ref}(f)\}, \text{flight}, pn)) \]

4.3.6 The Return office

The Return office can perform the following behaviours repeatedly during the execution:

1. A customer wants to cancel a ticket of a flight. The Return office is informed of which flight the customer wants to cancel the ticket and it also receives the customer's passport number. If the Return office owns that flight machine, it cancels the customer's ticket. In the case that the machine is empty after that cancellation, the Return office does not keep the machine anymore and it passes the empty machine to one of the sell agencies. If the Return office does not have that flight machine, it asks a Sell agency for that particular machine. There could be three responses. It either receives the machine from that Sell agency, or it receives a message from that Sell agency saying that the machine is empty, or it receives a message from that Sell agency saying that it does not have that machine. In the latter case, the Return office asks another Sell agency for that machine.

2. It receives a request from a Sell agency for a flight machine. Depending on the requested machine, the Return office performs one of the behaviours below:
   - If the machine is the Return office's empty machine, in other words the Return office has already been waiting to pass it to a Sell agency, the Return office gives that empty machine to the requester Sell agency immediately and without checking the current state of the machine.
   - If the Return office owns that machine and the machine is not its empty machine, it first checks whether the machine is currently full or not. If the machine is full, it sends a message to the requester Sell agency saying that the machine is full, and it does not send the machine to that Sell agency. If the machine is not full, it gives that machine to the Sell agency.
   - If it does not own that machine, it sends a message to the Sell agency saying that it does not have that machine.

3. It receives a full machine from a Sell agency.
The Return office gives a non empty machine to a Sell agency only if the receiver Sell agency has already asked the Return office for that particular machine. If the Return office does not receive a request from a Sell agency for a non empty machine, it does not pass any non empty machine to the sell agencies. However, if the Return office cancels a ticket of a flight and the flight becomes empty after that cancellation, it passes that empty machine to a Sell agency without having already received any request from that Sell agency for that particular machine.

On the other hand, the Return office receives a non full machine from a Sell agency only if it has already asked the sender Sell agency for that particular machine. It never receives a non full machine from a Sell agency without asking for that machine in advance. However, the Return office may receive a full machine from the sell agencies without having already asked the sender for that full machine.

The specification of the Return office which owns the machine references in $S$ is described as below:

$\text{ReturnOffice}(S) = R_1(S)$

$R_1(S) = customerToCancel?flight?pn \rightarrow$ if $\text{ref}(flight) \in S$
then $R_2(S,flight,pn)$
else $R_3(S,flight,pn)$

$\square$

$require?j?flight \rightarrow$ if $\text{ref}(flight) \in S$
then $R_6(S,flight,j)$
else $(\text{notHave}j \rightarrow R_1(S))$

$\square$

$\square_{j \in \text{Agencies}} ep.j?z \rightarrow R_4(S \cup \{z\})$

$R_2(S,flight, pn) = \text{ref}(flight).\text{cancel}!pn?resp \rightarrow$ if $\text{resp} = \text{Empty}$
then $R_4(S,flight)$
else $R_1(S)$
4.3. SYSTEM DESIGN AND SPECIFICATION IN MOBILE CSP

\[ R_3(S, \text{flight}, pn) = \text{ask?(j?flight) \to (ep?j?w \rightarrow R_2(S \cup \{w\}, \text{ref}^{-1}(w), pn)} \]

\[ \text{notHaveIt.j} \rightarrow R_3(S, \text{flight}, pn) \]

\[ \text{emptyMachine.j} \rightarrow R_3(S) \]

\[ \text{require?j?f} \rightarrow \text{if ref(f) \in S} \]

\[ \text{then R_{10}(S, \text{flight}, pn, j, f)} \]

\[ \text{else (notHaveIt.j \rightarrow R_3(S, \text{flight}, pn))} \]

\[ \text{j\inAgencies} \]

\[ \text{ep?j?w} \rightarrow \text{if w = ref(flight)} \]

\[ \text{then R_2(S \cup \{w\}, \text{ref}^{-1}(w), pn)} \]

\[ \text{else R_3(S \cup \{w\}, \text{flight}, pn)} \]

\[ R_4(S, \text{flight}) = \text{\bigwedge}_{j\inAgencies} \]

\[ \text{dp?j?ref(flight) \rightarrow R_1(S - \{\text{ref(flight)}\})} \]

\[ \text{require?j?f} \rightarrow \text{if ref(f) = ref(flight)} \]

\[ \text{then dp?j?ref(flight) \rightarrow R_1(S - \{\text{ref(flight)}\})} \]

\[ \text{else R_4(S, \text{flight}, j, f)} \]

\[ \text{j\inAgencies} \]

\[ \text{ep?j?w} \rightarrow R_4(S \cup \{w\}, \text{flight}) \]

\[ R_5(S, \text{flight}, j) = \text{ref(flight)}, \text{full?resp} \rightarrow \text{if resp = YES} \]

\[ \text{then R_6(S, j)} \]

\[ \text{else R_7(S, \text{flight}, j)} \]

\[ R_6(S, j) = \text{fullMachine.j} \rightarrow R_1(S) \]

\[ R_7(S, \text{flight}, j) = \text{dp?j?ref(flight) \rightarrow R_4(S - \{\text{ref(flight)}\})} \]
4.3.7 Controller

A CSP process, Controller, is introduced which is the parallel combination of sell agencies and the Return office. As we said before, we assume that at the beginning of the system execution, each Sell agency owns one of the flight machines. As a result, Controller is specified as shown below:

\[
\text{Controller} = \text{ReturnOffice}() || \text{SellAgency}(1, \{mr1\}) || \text{SellAgency}(2, \{mr2\})
\]

4.4 Coding mobility into standard CSP

In this section we describe how mobile CSP specification of our system can be coded as a standard CSP specification which can be then accepted by the CSP software tools: ProBE and FDR.

In our system's CSP specification, we first introduce \( mr1 \) and \( mr2 \) as the elements of the data type \( MR \). In standard CSP, we are not allowed to introduce them again in the CSP specification as the channels since they are data elements. To solve this problem, we use a new channel, \( MC \), to represent the call of the machine operations through the machine references. As a result, channel \( MC \) is a channel which carries the values of \( MR \) and \( Operations \) and it is introduced in the standard CSP specification as shown below:

\[
\text{channel} \ MC : MR.Operations
\]
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

By using channel \( MC \), any call of a machine operation \( \text{ref}(\text{flight}).\text{op}!s?t \) in the body of the CSP controllers is modelled as \( MC.\text{ref}(\text{flight}).\text{op}!s?t \) in the standard CSP description of our system. Channel \( MC \) does not change the architecture of our system. It is only used to create a CSP specification which can be then accepted by ProBE and FDR.

In order to model \( \text{ref}^{-1} \) in standard CSP specification, we introduce a new function \( \text{refinv} \) which is the inverse of function \( \text{ref} \). Function \( \text{refinv} \) is introduced in standard CSP specification as below:

\[
\text{refinv}(mr1) = \text{flight1} \\
\text{refinv}(mr2) = \text{flight2}
\]

By using function \( \text{refinv} \), any \( \text{ref}^{-1}(w) \) in the body of the CSP controllers is modelled as \( \text{refinv}(w) \) in the standard CSP description of our system.

In mobile CSP specification of our system, control point channels were introduced in the form of \( cp.i.j, dp.i \) and \( ep.i \) where \( i,j \in \text{Agencies} \). However, in standard CSP specification, there are three control point channels: \( cp, dp \) and \( ep \). Any \( \Box_{j \in \text{Agencies}} ep.j?w \) and \( \Box_{j \in \text{Agencies}} dp.j!\text{ref}(\text{flight}) \) in the body of the Return office are modelled as \( ep?j?w \) and \( dp?j!\text{ref}(\text{flight}) \) respectively in the standard CSP description of our system.

The full standard CSP specification of our flight tickets sale system is attached in Appendix B.

After coding the system in standard CSP, the behaviour of each Sell agency and the Return office was checked individually by using ProBE. We then used ProBE to explore the execution of \( \text{Controller} \) in order to check their behaviour while working in parallel in the system. In addition, \( \text{Controller} \) was established to be divergence free and deadlock free by using FDR.

4.5 Verification of the system divergence freedom

In this section we verify the divergence freedom of our system by using Theorem 3 presented in Chapter 3. Verification of divergence freedom contains two steps:

1. The parallel combination of sell agencies and the Return office must be established to be divergence free.

2. Sell agencies and the Return office must be CLI preserver.

\( \text{Controller} \) has already been established to be divergence free by using FDR. As a result, the first step in our system divergence freedom verification is established.
The next step is to establish that the sell agencies and the Return office are CLI preserver. In other words, conditions 1 and 2 of Definition 11 in Chapter 3 must be established for sell agencies and the Return office. In order to achieve this, we should first assign assertions for control points in our system. Then, we should define control loop invariants for the processes in sell agencies and in the Return office. The rest of this section presents the steps towards proving that sell agencies and the Return office are CLI preserver which then results the divergence freedom of the system at the end of this section.

### 4.5.1 Assertions on control points

If a machine is passed from a Sell agency to another Sell agency, it should not be already full as there is no more ticket available to be sold next. In addition, if a machine is passed from a Sell agency to the Return office, it should not be empty as there is no sold ticket in the machine to be canceled. Moreover, if a machine is passed from the Return office to a Sell agency, it should not be already full as there is no more ticket available to be sold next. Thus, the important issues which should be considered as the safety properties of our system are as below:

- A Sell agency should never pass a full machine to another Sell agency.
- The Return office should never pass a full machine to a Sell agency.
- A Sell agency should never pass an empty machine to the Return office.

There are three families of control points in our system: $cp$, $dp$ and $ep$. $cp$ is the family of control points where the machines are passed between sell agencies, $dp$ is the family of control points where the machines are passed from the Return office to a Sell agency, and $ep$ is the family of control points where the machines are passed from a Sell agency to the Return office. Note that there is one family of control points $cp$ for passing machines between sell agencies. However, there are two separate families of control points, $dp$ and $ep$, for passing machines between sell agencies and the Return office. The reason is that when a machine is passed from a Sell agency to the Return office, the machine should not be empty. On the other hand, when a machine is passed from the Return office to a Sell agency, the machine should not be full. As a result, there should be two different assertions each for one direction between a Sell agency and the Return office. Therefore, we introduce two separate families of control points each for one direction.

The assertions we assign for each control point depends on which controller is the receiver at that point. If the receiver is a Sell agency, the assertion should be something to ensure that the machine is not full. If the receiver is the Return
office, the assertion should be something to ensure that the machine is not empty. To summarise, the assertions of control points should be categorised as below:

1. Assertion of the control point which passes the machines from a Sell agency to another Sell agency:
   - Machine is not full

2. Assertion of the control point which passes the machines from the Return office to a Sell agency:
   - Machine is not full

3. Assertion of the control point which passes the machines from a Sell agency to the Return office:
   - Machine is not empty

According to what was discussed above, the assertions of control points should be assigned as below:

1. `assert(cp.i,jz) : z.sold != z.seats`
2. `assert(dp.i,z) : z.sold != z.seats`
3. `assert(ep.i,z) : z.sold != 0`

### 4.5.2 Control Loop Invariants in Sell agencies

We introduce the Control Loop Invariant, \( CLI \), for each process \( P_x (1 \leq x \leq 10) \) in a Sell agency. We introduce the CLI for each process according to the behaviour of the process. Each CLI includes predicates on the type of parameters and on the current state of the parameters at the beginning of each recursive call. For instance, process \( P_2(i, S, \text{flight}, pn) \) sells a ticket of machine \( \text{flight} \). As a result, \( P_2(i, S, \text{flight}, pn) \) should own \( \text{flight} \) at the beginning of the recursive call. So, \( \text{ref(flight)} \in S \) is one of the predicates we introduce in \( CLI_{P_2(i,S,\text{flight},pn)} \). In addition, all the machines owned by \( P_2(i, S, \text{flight}, pn) \) as well as \( \text{flight} \) should not be full. This is expressed by \( \forall k \in S : k.sold \neq k.seats \).

\[
CLI_{P_1(i,S)} : i \in \text{Agencies} \land S \subseteq MR \land \forall k \in S : k.sold \neq k.seats
\]

\[
CLI_{P_2(i,S,\text{flight},pn)} : CLI_{P_1(i,S)} \land \text{flight} \in \text{Flights} \land \text{ref(flight)} \in S \land pn \in \text{Passport}
\]
4.5.3 Establishing Sell agencies are CLI preserver

We assume \( \text{init}_1 \) and \( \text{init}_2 \) are the initialisation of machines \( \text{flight}_1 \) and \( \text{flight}_2 \) respectively. As the sell agencies have the same structure in our system, we check conditions 1 and 2 of Definition 11 in Chapter 3 for a sample Sell agency: 

\[
\text{SellAgency}(i, \{ \text{mri} \}) \cdot i \in \text{Agencies},
\]

Condition 1:

\[
[\text{init}_i; \text{rec} := P_1(i, \{ \text{mri} \})](\text{CLIP}_1(i, \{ \text{mri} \})) = [\text{init}_i](\text{CLIP}_1(i, \{ \text{mri} \})) =
\]

\[
[\text{init}_i](i \in \text{Agencies} \land \{ \text{mri} \} \subseteq \text{MR} \land \text{mri}.\text{sold} \neq \text{mri}.\text{seats}) =
\]

\[
i \in \text{Agencies} \land \{ \text{mri} \} \subseteq \text{MR} \land 0 \neq \text{mri}.\text{seats}
\]

\( \checkmark \)

Condition 2:

\* \( P_1(i, S) \)

\( P_1(i, S) = 1 \bigcirc 2 \bigcirc 3 \bigcirc 4 \)

\( \text{trans}(P_1(i, S)) = \text{CHOICE trans}(1) \text{ OR trans}(2) \text{ OR trans}(3) \text{ OR trans}(4) \text{ END} \)
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

\[ [\text{trans}(P_1(i, S))](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P)) = \]
\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P)) \land \]
\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P)) \land \]
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P)) \land \]
\[ [\text{trans}(4)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P)) \]

1 = \text{customerToBuy}(i, \text{flight}, \text{pm}) \rightarrow \text{if } \text{ref}(\text{flight}) \in S
   \begin{align*}
   & \text{then } P_2(i, S, \text{flight}, \text{pm}) \\
   & \text{else } P_3(i, S, \text{flight}, \text{pm})
   \end{align*}

\((\text{rec }= P_1(i, S) \land \text{CLI}_P) \Rightarrow [\text{trans}(1)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

2 = \text{request}(i, \text{flight}) \rightarrow \text{if } \text{ref}(\text{flight}) \in S
   \begin{align*}
   & \text{then } P_4(i, S, j, \text{flight}) \\
   & \text{else } (\text{sorry}.i.j \rightarrow P_1(i, S))
   \end{align*}

\((\text{rec }= P_1(i, S) \land \text{CLI}_P) \Rightarrow [\text{trans}(2)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

3 = \text{ask}(i, \text{flight}) \rightarrow \text{if } \text{ref}(\text{flight}) \in S
   \begin{align*}
   & \text{then } P_5(i, S, \text{flight}) \\
   & \text{else } (\text{notHaveIt}.i \rightarrow P_1(i, S))
   \end{align*}

\((\text{rec }= P_1(i, S) \land \text{CLI}_P) \Rightarrow [\text{trans}(3)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

4 = \text{dp}(i, z) \rightarrow P_1(i, S \cup \{z\})

\((\text{rec }= P_1(i, S) \land \text{CLI}_P) \Rightarrow [\text{trans}(4)](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

\((\text{rec }= P_1(i, S) \land \text{CLI}_P) \Rightarrow [\text{trans}(P_1(i, S))](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

\(\checkmark\)

\(\checkmark P_2(i, S, \text{flight}, \text{pm})\)

\((\text{rec }= P_2(i, S, \text{flight}, \text{pm}) \land \text{CLI}_P) \Rightarrow [\text{trans}(P_2(i, S, \text{flight}, \text{pm}))](\forall 1 \leq x \leq 10 \bullet (\text{rec }= P_x \Rightarrow \text{CLI}_P))\)

\(\checkmark\)
\* \( P_3(i, S, \text{flight}, pn) \)

\[ P_3(i, S, \text{flight}, pn) = 1 \circ 2 \circ 3 \circ 4 \circ 5 \]

\[ \text{trans}(P_3(i, S, \text{flight}, pn)) = \text{CHOICE trans(1)} \]
\[ \quad \text{OR trans(2) OR trans(3)} \]
\[ \quad \text{OR trans(4) OR trans(5)} \]
\[ \text{END} \]

\[ \text{trans}(P_3(i, S, \text{flight}, pn)) \]
\[ [V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] = \]
\[ [\text{trans(1)}][V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] \land \]
\[ [\text{trans(2)}][V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] \land \]
\[ [\text{trans(3)}][V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] \land \]
\[ [\text{trans(4)}][V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] \land \]
\[ [\text{trans(5)}][V 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_P)] \]

\[ 1 = \text{request.} i \cdot j \cdot \text{flight} \rightarrow ((\text{cp.} j \cdot i ? w \rightarrow P_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn)) \]
\[ \quad \text{notHaveIt.} i \rightarrow P_3(i, S, \text{flight}, pn)) \]
\[ \quad (\text{MachineIsFull.} j \cdot i \rightarrow P_1(i, S)) \]

\[ 2 = \text{require.} i \cdot \text{flight} \rightarrow ((\text{dp.} i ? w \rightarrow P_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn)) \]
\[ \quad \text{notHaveIt.} i \rightarrow P_3(i, S, \text{flight}, pn)) \]
\[ \quad (\text{fullMachine.} i \rightarrow P_1(i, S)) \]

\[ 3 = \text{request.} j \cdot i ? f \rightarrow (\text{if } \text{ref}(f) \in S \]
\[ \quad \text{then } \text{cp.} i \cdot j \cdot \text{ref}(f) \rightarrow P_3(i, S - \{\text{ref}(f)\}, \text{flight}, pn) \]
\[ \quad \text{else } (\text{sorry.} i \cdot j \rightarrow P_3(i, S, \text{flight}, pn))) \]
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

\( (\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLI}_{P_3(i, S, \text{flight}, pn)} \Rightarrow \) \\
[\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( 4 = \text{ask} . i ? f \rightarrow (\text{if } \text{ref}(f) \in S \) \\
then \ P_{10}(i, S, \text{flight}, pn, f) \) \\
else \ (\text{notHave} . f \rightarrow P_3(i, S, \text{flight}, pn))) \)

\( (\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLI}_{P_3(i, S, \text{flight}, pn)} \Rightarrow \) \\
[\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( 5 = \text{dp} . i ? w \rightarrow (\text{if } w = \text{ref(\text{flight})} \) \\
then \ P_2(i, S \cup \{w\}, \text{flight}, pn) \) \\
else \ P_3(i, S \cup \{w\}, \text{flight}, pn) \)

\( (\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLI}_{P_3(i, S, \text{flight}, pn)} \Rightarrow \) \\
[\text{trans}(5)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( (\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLI}_{P_3(i, S, \text{flight}, pn)} \Rightarrow \) \\
[\text{trans}(P_3(i, S, \text{flight}, pn))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( \checkmark \)

\( \checkmark \ P_4(i, S, j, \text{flight}) \)

\( (\text{rec} = P_4(i, S, j, \text{flight}) \land \text{CLI}_{P_4(i, S, j, \text{flight})} \Rightarrow \) \\
[\text{trans}(P_4(i, S, j, \text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( \checkmark \)

\( \checkmark \ P_5(i, S, \text{flight}) \)

\( (\text{rec} = P_5(i, S, \text{flight}) \land \text{CLI}_{P_5(i, S, \text{flight})} \Rightarrow \) \\
[\text{trans}(P_5(i, S, \text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x})) \)

\( \checkmark \)
\[ P_6(i, S, \text{flight}) \]
\[ P_6(i, S, \text{flight}) = 1 \Box 2 \Box 3 \Box 4 \]

\[ \text{trans}(P_6(i, S, \text{flight})) = \text{CHOICE trans}(1) \text{ OR trans}(2) \]
\[ \text{OR trans}(3) \text{ OR trans}(4) \]
\[ \text{END} \]

\[ [\text{trans}(P_6(i, S, \text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) = \]
\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]
\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]
\[ [\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]

\[ 1 = \text{ep.} \text{!ref}(\text{flight}) \rightarrow P_1(i, S - \{\text{ref}(\text{flight})\}) \]
\[ (\text{rec} = P_6(i, S, \text{flight}) \land \text{CLI}_P(i, S, \text{flight}) \Rightarrow \]
\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]

\[ 2 = \text{request}?:j[/i]?f \rightarrow \text{if } \text{ref}(f) = \text{ref}(\text{flight}) \]
\[ \text{then } (\text{MachineIsFull.i}.j \rightarrow P_6(i, S, \text{flight})) \]
\[ \text{else } P_7(i, S, \text{flight}, j, f) \]
\[ (\text{rec} = P_6(i, S, \text{flight}) \land \text{CLI}_P(i, S, \text{flight}) \Rightarrow \]
\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]

\[ 3 = \text{ask}.i?[f \rightarrow \text{if } \text{ref}(f) = \text{ref}(\text{flight}) \]
\[ \text{then } (\text{ep.} \text{!ref}(\text{flight}) \rightarrow P_1(i, S - \{\text{ref}(\text{flight})\})) \]
\[ \text{else } P_5(i, S, \text{flight}, f) \]
\[ (\text{rec} = P_6(i, S, \text{flight}) \land \text{CLI}_P(i, S, \text{flight}) \Rightarrow \]
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]

\[ 4 = \text{dp}.i?[w \rightarrow P_6(i, S \cup \{w\}, \text{flight}) \]
\[ (\text{rec} = P_6(i, S, \text{flight}) \land \text{CLI}_P(i, S, \text{flight}) \Rightarrow \]
\[ [\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P(x)) \land \]

\[ \text{END} \]

\[ \text{END} \]

\[ \text{END} \]
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

\[ \text{CLIP}_0(i, S, \text{flight}) \Rightarrow \text{trans}(P_0(i, S, \text{flight})) \]
\[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) \]

\[ \text{CLIP}_1(i, S, \text{flight}, j, f) \]
\[ \text{trans}(P_1(i, S, \text{flight}, j, f)) \]
\[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) \]

\[ \text{CLIP}_2(i, S, \text{flight}, j, f) \]
\[ \text{trans}(P_2(i, S, \text{flight}, j, f)) \]
\[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) \]

\[ \text{CLIP}_3(i, S, \text{flight}, j, f) \]
\[ \text{trans}(P_3(i, S, \text{flight}, j, f)) \]
\[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) \]

\[ \text{CLIP}_4(i, S, \text{flight}, p, n, f) \]
\[ \text{trans}(P_4(i, S, \text{flight}, p, n, f)) \]
\[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) \]

The full wp proof for each process \( P_x (1 \leq x \leq 10) \) is attached in Appendix C.

4.5.4 Control Loop Invariants in the Return office

We introduce the Control Loop Invariant, \( \text{CLI} \), for each process \( P_x (1 \leq x \leq 10) \) in the Return office. We introduce the CLI for each process according to the behaviour of the process. Each CLI includes predicates on the type of parameters.
and on the current state of the parameters at the beginning of each recursive call. For instance, process $R_2(S, \text{flight}, pn)$ cancels a ticket of machine \textit{flight}. As a result, $R_2(S, \text{flight}, pn)$ should own \textit{flight} at the beginning of the recursive call. So, \text{ref} (\text{flight}) \in S$ is one of the predicates we introduce in $CLIR_{R_2(S, \text{flight}, pn)}$. In addition, all the machines owned by $R_2(S, \text{flight}, pn)$ as well as \text{flight} should not be empty. This is expressed by $\forall k \in S \cdot k.\text{sold} \neq 0$.

$$CLIR_{R_1(S)}: \quad S \subseteq MR \land \forall k \in S \cdot k.\text{sold} \neq 0$$

$$CLIR_{R_2(S, \text{flight}, pn)}: \quad CLIR_{R_1(S)} \land \text{flight} \in \text{Flights} \land \text{ref} (\text{flight}) \in S \land pn \in \text{Passport}$$

$$CLIR_{R_3(S, \text{flight}, pn)}: \quad CLIR_{R_1(S)} \land \text{flight} \in \text{Flights} \land \text{ref} (\text{flight}) \in (MR - S) \land pn \in \text{Passport}$$

$$CLIR_{R_4(S, \text{flight})} : \quad S \subseteq MR \land \text{flight} \in \text{Flights} \land \text{ref} (\text{flight}).\text{sold} = 0 \land \forall k \in (S - \{\text{ref} (\text{flight})\}) \cdot k.\text{sold} \neq 0 \land \text{ref} (\text{flight}) \in S$$

$$CLIR_{R_5(S, \text{flight}, j)} : \quad CLIR_{R_1(S)} \land \text{flight} \in \text{Flights} \land \text{ref} (\text{flight}) \in S \land j \in \text{Agencies}$$

$$CLIR_{R_6(S, j)} : \quad CLIR_{R_1(S)} \land j \in \text{Agencies}$$

$$CLIR_{R_7(S, \text{flight}, j)} : \quad CLIR_{R_6(S, \text{flight}, j)} \land \text{ref} (\text{flight}).\text{sold} \neq \text{ref} (\text{flight}).\text{seats}$$

$$CLIR_{R_8(S, \text{flight}, j, f)} : \quad CLIR_{R_7(S, \text{flight}, j)} \land j \in \text{Agencies} \land \text{ref} (f) \in MR \land \text{ref} (f) \neq \text{ref} (\text{flight})$$

$$CLIR_{R_9(S, \text{flight}, j, f)} : \quad CLIR_{R_8(S, \text{flight}, j, f)} \land j \in \text{Agencies} \land \text{ref} (f) \in S \land \text{ref} (f) \neq \text{ref} (\text{flight})$$

$$CLIR_{R_{10}(S, \text{flight}, pn, j, f)} : \quad CLIR_{R_6(S, \text{flight}, pn)} \land j \in \text{Agencies} \land \text{ref} (f) \in S$$

4.5.5 Establishing the Return office is CL1 preserver

In this section, we establish conditions 1 and 2 of Definition 11 in Chapter 3 for the Return office.

Condition 1:

As the Return office does not have any machine initially, Condition 1 is established as below:

$$\text{rec} := R_1(\{\}) (CLIR_{R_1(\{\})) = \{\} \subseteq MR$$

\checkmark
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

Condition 2:

\[ R_1(S) \]
\[ R_1(S) = 1 \cup 2 \cup 3 \]

\[ \text{trans}(R_1(S)) = \text{CHOICE trans(1) OR trans(2) OR trans(3) END} \]

\[
[\text{trans}(R_1(S))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) = \\
[\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \land \\
[\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \land \\
[\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a}))
\]

\[ 1 = \text{customerToCancel?flight?pn} \rightarrow \text{if} \ \text{ref(flight)} \in S \]
\[ \quad \text{then } R_2(S, \text{flight}, \text{pn}) \]
\[ \quad \text{else } R_3(S, \text{flight}, \text{pn}) \]

\[ (\text{rec} = R_1(S) \land \text{CLI}_{R_1(S)}) \Rightarrow [\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \]

\[ 2 = \text{require?j?flight} \rightarrow \text{if} \ \text{ref(flight)} \in S \]
\[ \quad \text{then } R_5(S, \text{flight}, j) \]
\[ \quad \text{else } \text{notHaveIt.j} \rightarrow R_1(S) \]

\[ (\text{rec} = R_1(S) \land \text{CLI}_{R_1(S)}) \Rightarrow [\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \]

\[ 3 = \Box_{j \in \text{Agencies}} \text{ep.j?z} \rightarrow R_1(S \cup \{x\}) \]

\[ (\text{rec} = R_1(S) \land \text{CLI}_{R_1(S)}) \Rightarrow [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \]

\[ (\text{rec} = R_1(S) \land \text{CLI}_{R_1(S)}) \Rightarrow \\
[\text{trans}(R_1(S))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_a})) \]

\[ \checkmark \]
\[ + R_2(S, \text{flight}, pn) \]
\[
(\text{rec} = R_2(S, \text{flight}, pn) \land CLI_{R_2}(S, \text{flight}, pn)) \Rightarrow \\
[\text{trans}(R_2(S, \text{flight}, pn))](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x}))
\]
\[
\checkmark
\]

\[ + R_3(S, \text{flight}, pn) \]
\[
R_3(S, \text{flight}, pn) = 1 \, \square \, 2 \, \square \, 3
\]
\[
\text{trans}(R_3(S, \text{flight}, pn)) = \text{CHOICE} \ \text{trans}(1) \ \text{OR} \ \text{trans}(2) \ \text{OR} \ \text{trans}(3) \ \text{END}
\]
\[
[\text{trans}(R_3(S, \text{flight}, pn))](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x})) = \\
[\text{trans}(1)](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x})) \land \\
[\text{trans}(2)](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x})) \land \\
[\text{trans}(3)](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x}))
\]

\[ 1 = \text{ask}\ ? j \ ? \text{flight} \rightarrow (\text{ep} \ ? j \ ? w \rightarrow R_2(S \cup \{w\}, \text{ref}^{-1}(w), pn))
\]
\[
\square
\]
\[
\text{notHaveIt} \ ? j \rightarrow R_3(S, \text{flight}, pn)
\]
\[
\square
\]
\[
\text{emptyMachine} \ ? j \rightarrow R_1(S)
\]
\[
(\text{rec} = R_3(S, \text{flight}, pn) \land CLI_{R_3}(S, \text{flight}, pn)) \Rightarrow \\
[\text{trans}(1)](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x}))
\]

\[ 2 = \text{require}\ ? j \ ? f \rightarrow \text{if} \ \text{ref} \ (f) \in S
\]
\[
\text{then} \ R_{10}(S, \text{flight}, pn, j, f)
\]
\[
\text{else} \ (\text{notHaveIt} \ ? j \rightarrow R_3(S, \text{flight}, pn))
\]
\[
(\text{rec} = R_3(S, \text{flight}, pn) \land CLI_{R_3}(S, \text{flight}, pn)) \Rightarrow \\
[\text{trans}(2)](\forall 1 \leq x < 10 \cdot (\text{rec} = R_x \Rightarrow CLI_{R_x}))
\]

\[ 3 = \square \ \text{\scriptsize{\text{\}}}}_{\text{Agencies}} \ \text{ep} \ ? j \ ? w \rightarrow \text{if} \ w = \text{ref} \ (\text{flight})
\]
\[
\text{then} \ R_3(S \cup \{w\}, \text{ref}^{-1}(w), pn)
\]
\[
\text{else} \ R_3(S \cup \{w\}, \text{flight}, pn)
\]
4.5. VERIFICATION OF THE SYSTEM DIVERGENCE FREEDOM

\[ \text{(rec} = R_3(S, \text{flight}, pn) \land \text{CLI}_R_3(S, \text{flight}, pn) \Rightarrow \] 
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]

\[ (\text{rec} = R_3(S, \text{flight}, pn) \land \text{CLI}_R_3(S, \text{flight}, pn) \Rightarrow \] 
\[ [\text{trans}(R_3(S, \text{flight}, pn))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]

\( \checkmark \)

\* \( R_4(S, \text{flight}) \)

\( R_4(S, \text{flight}) = 1 \otimes 2 \otimes 3 \)

\[ \text{trans}(R_4(S, \text{flight})) = \text{CHOICE trans}(1) \text{ OR trans}(2) \text{ OR trans}(3) \text{ END} \]

\[ [\text{trans}(R_4(S, \text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) = \]
\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \land \]
\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \land \]
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]

\( 1 = \Box_{j\in \text{Agencies}} \text{dp}_j \cdot \text{ref}(\text{flight}) \rightarrow R_1(S - \{\text{ref(\text{flight})}\}) \)

\[ (\text{rec} = R_4(S, \text{flight}) \land \text{CLI}_R_4(S, \text{flight}) \Rightarrow \] 
\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]

\( 2 = \text{require}\?j\?f \rightarrow \text{if } \text{ref}(f) = \text{ref(\text{flight})} \] 
\[ \text{then } \text{dp}_j \cdot \text{ref(\text{flight})} \rightarrow R_1(S - \{\text{ref(\text{flight})}\}) \] 
\[ \text{else } R_3(S, \text{flight}, j, f) \]

\[ (\text{rec} = R_4(S, \text{flight}) \land \text{CLI}_R_4(S, \text{flight}) \Rightarrow \] 
\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]

\( 3 = \Box_{j\in \text{Agencies}} \text{ep}_j \cdot \text{?w} \rightarrow R_4(S \cup \{w\}, \text{flight}) \)

\[ (\text{rec} = R_4(S, \text{flight}) \land \text{CLI}_R_4(S, \text{flight}) \Rightarrow \] 
\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R_x)) \]
(\text{rec} = R_4(S, \text{flight}) \land CLI_{R_4(s,flight)}) \Rightarrow \\
[\text{trans}(R_4(S, \text{flight}))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_4}))
\checkmark

\bullet R_5(S, \text{flight}, j) \\
(\text{rec} = R_5(S, \text{flight}, j) \land CLI_{R_5(s,flight,j)}) \Rightarrow \\
[\text{trans}(R_5(S, \text{flight}, j))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_5}))
\checkmark

\bullet R_6(S, j) \\
(\text{rec} = R_6(S, j) \land CLI_{R_6(s,j)}) \Rightarrow \\
[\text{trans}(R_6(S, j))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_6}))
\checkmark

\bullet R_7(S, \text{flight}, j) \\
(\text{rec} = R_7(S, \text{flight}, j) \land CLI_{R_7(s,flight,j)}) \Rightarrow \\
[\text{trans}(R_7(S, \text{flight}, j))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_7}))
\checkmark

\bullet R_8(S, \text{flight}, j, f) \\
(\text{rec} = R_8(S, \text{flight}, j, f) \land CLI_{R_8(s,flight,j,f)}) \Rightarrow \\
[\text{trans}(R_8(S, \text{flight}, j, f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_8}))
\checkmark

\bullet R_9(S, \text{flight}, j, f) \\
(\text{rec} = R_9(S, \text{flight}, j, f) \land CLI_{R_9(s,flight,j,f)}) \Rightarrow \\
[\text{trans}(R_9(S, \text{flight}, j, f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_e \Rightarrow CLI_{R_9}))
\checkmark
4.6. VERIFICATION OF THE SYSTEM DEADLOCK FREEDOM

\[ R_{10}(s, flight, pn, j, f) \]
\[ (rec = R_{10}(s, flight, pn, j, f) \land CLI_{R_{10}}(s, flight, pn, j, f)) \Rightarrow \]
\[ [trans(R_{10}(s, flight, pn, j, f))](\forall 1 \leq x \leq 10 \cdot (rec = R_x \Rightarrow CLI_{R_x})) \]

✓

The full wp proof for each process \( R_x (1 \leq x \leq 10) \) is attached in Appendix C.

4.5.6 Divergence freedom establishment

Sell agencies and the Return office have been successfully established to be CLI preserver in sections 4.5.3 and 4.5.5. All conditions in Theorem 3 presented in Chapter 3 are true. Thus, according to Theorem 3 our system is divergence free. In other words, operations are always called within their precondition, as expected.

4.6 Verification of the system deadlock freedom

The deadlock freedom of our system is verified by using Theorem 4 presented in Chapter 3. First, all B machines should contain only pre-conditioned operations. The structure of the operations in B machines in our system has been designed such that they all contain only pre-conditioned operations. The second step in our verification is that the system should be divergence free. We have already proved in previous section that the system is divergence free. The final step is to establish that the parallel combination of sell agencies and the Return office is deadlock free. Controller has already been established to be deadlock free by using FDR. All conditions in Theorem 4 are true. Thus, according to Theorem 4 our system is deadlock free.

4.7 Discussion

In this chapter, we used a case study to illustrate the key aspects of theoretical results of Chapter 3. The case study demonstrates the applicability of our Mobile CSP \(B\) framework in specifying and verifying peer-to-peer networks. It shows that the theorems we proved during this research activity are sufficient and capable to verify the consistency of a mobile combined communicating system. In this chapter, we illustrated all steps from developing a Mobile CSP \(B\) model to verifying its consistency. We detailed how two separate parts of the system, B and CSP, should be designed and specified in order to communicate with each
other during the execution. Using Mobile CSP $|$ B architecture, we were able to specify both behavioural and data aspects of the flight tickets sale system.

As our framework can be coded into the original constructs of CSP and B, we are able to use the comprehensive and highly efficient software tools of CSP and the B-Method to analyse, animate and check the behaviour and the consistency of the two parts of a mobile system separately. This shows the advantage of using the B-Method as our chosen state-based formal method and CSP as the process algebra in our framework. In our case study, we used ProB to check, analyse and animate the specification of the B machines. In addition, we used powerful CSP software tools, ProBE and FDR, in the consistency verification. ProBE was used to explore the system execution to check the behaviour of the CSP controllers in the system during the execution. It was useful to check if the CSP part of the system behaves according to what we expect. FDR was used to check divergence freedom and deadlock freedom of the parallel combination of the controllers. These were the conditions in Theorems 3 and 4 presented in Chapter 3 which had to be true to establish the consistency of our system.

At the beginning of our system development, we assumed a small flight ticket sale system which contains two sell agencies and two flight machines. We can easily extend our system and develop a larger system with more sell agencies and flight machines. In order to extend our system, we only need to change small parts of the CSP specification as listed below:

- We should add the new flights to datatype Flights.
- The new machine references should be added to datatype MR.
- In mobile CSP specification of the system, the new machine references should also be added as the new channels.
- Set Agencies should be changed to the number of the sell agencies in the extended system.
- Function ref should be extended to map the new machines to their machine references.
- The new sell agencies and machine references should be added into the body of the process Controller.

The rest of the system remains the same as before.

During the consistency verification of our system, we found out that the weakest precondition proof might be difficult for people who do not have a good mathematics background. The person should be capable to work out and deal with
4.7. DISCUSSION

In order to do the wp proof correctly, logical proofs will be more difficult if operations of the machines are more complex. For instance, if there are several predicates as the precondition, or if the body of the operations contain several assignments, then it needs more effort to work out the weakest preconditions. In addition, more effort is needed to work out the wp proof correctly if the body of CSP processes are more complex. For instance, if the body of the CSP processes are long and contain lots of channels with inputs and outputs, or several external and internal choices, or several if then else statements, then it will be difficult to follow and track the proof of wp correctly. We believe that for the weakest precondition proof to be applicable on any large scale, tool support would be necessary.

Mobile CSP || B does not have dynamic component creation. It is a suitable framework for the systems whose components are the same from the beginning and during the execution, or for the systems whose components may terminate or leave the system during the execution. In a real flight tickets sale system, new flights are added into the system during the execution. Also, in reality, new agencies may be added into the system as the new sell agencies during the execution. However, in our flight tickets sale system there is no possibility for new flights or new sell agencies to be added into the system during the system execution. We will discuss this in Chapter 6.

Assigning the correct assertion for control points is one of the key points in consistency verification. For each control point, the assertion should be assigned depending on who is the receiver and what the receiver is going to do with the machine it receives along that control point. In our flight tickets sale system, the receiver controllers are either sell agencies or the Return office. So, the assertions are easily assigned as either "machine is not full" or "machine is not empty". However, in a large system which contains several controllers each doing different tasks, assigning a correct assertion for each control point will need more effort.

Another key point in consistency verification is introducing control loop invariants. CLI of each process should be chosen very carefully and according to the behaviour of that process. Our flight tickets sale system is a small system which contains three controllers two of which are sell agencies having the same structure. So, we only needed to define CLIs for the processes in a sample of sell agencies and CLIs for the processes in the Return office. However, defining correct CLIs for the processes in controllers will need more work in large systems containing several controllers each having different structures.

In our small flight tickets sale system, if one of the controllers needs a machine, there are only two other processes to ask for that machine. However, if we extend our system to a large one containing several numbers of sell agencies, it will not be as easy as our small example. If a Sell agency needs a flight machine, it has
to ask other sell agencies and the Return office one by one for that machine. In other words, the Sell agency has to continue asking each one until the owner of that machine is found. Even in this case, after a long search for that machine, the requester Sell agency might be finally told by the owner that the machine is already full. The same problem can also occur for the Return office in the system during the execution: it has to ask each Sell agency one by one to find the owner of the machine and even then it might be told by the owner that the machine is empty. Another problem is if a controller needs a machine, the owner of that machine may not be working with the machine and may not need that machine but it is busy by doing something else and cannot receive the request message to send its machine. In other words, the machine is ready to be used and no one is using it but the controller who wants that machine cannot access to it until the owner has finished its work and is free to receive the request message and then sends the machine.

After doing this case study, a new idea raised that it would be a useful technique if we add a special process to our system which does not sell or cancel any tickets and it is only responsible for keeping the flight machines in the system. Initially, all the machines must be in possession of this machines keeper. During the execution, controllers should ask the machines keeper for any machine they need to work with. After controllers finish their work with a flight machine, they should give the machine back to the machines keeper. This should happen even if a machine is full after selling a ticket or if a machine is empty after cancelling a ticket. In other words, sell agencies and the Return office do not pass any machine between each other during the execution.

By using this method, whenever a Sell agency or the Return office needs to work with a flight machine, it only needs to ask one special process for that machine instead of asking several controllers. If the machines keeper does not have that machine, it means that the machine is being used by one of the other controllers at the moment. Another advantage of this method is that no machine is kept anywhere without being used while someone else needs to work with it.

As the sell agencies and the Return office give all their machines back to the machines keeper after they finish their work with the machines, the machines keeper may own any kinds of machines during the execution such as full, empty, or a machine with both sold tickets and unsold tickets. This also means that sell agencies and the Return office do not need to check the current state of their owned machines when giving them back to the machines keeper. However, this is the machines keeper's responsibility to check the current state of machines before passing them to the sell agencies or to the Return office. Whenever a Sell agency wants a machine, the machines keeper should first check whether the machine is full or not and then it should give the machine to the requester Sell agency only if the machine is not full. On the other hand, whenever the Return office wants a
machine, the machines keeper should first check whether the machine is empty or not and then it should give the machine to the Return office only if the machine is not empty.

If we have the machines keeper in our flight tickets sale system, there will not be any control points between any two controllers. Control point channels will be only situated between the machines keeper and the controllers. So, we will have less channels in our system specification. There should be two control points each for one direction between any controller and the machines keeper. This is because of different assertions which should be assigned for each direction. For all control points which passes the flight machines to the machines keeper, the assertion is the predicate True as any kind of machines can be passed to the machines keeper. However, when a machine is passed from the machines keeper to a Sell agency, it should not be full and when a machine is passed from the machines keeper to the Return office, it should not be empty. As the components would be essentially the same, wp proofs can be essentially reused. In a large system with several numbers of machines, we may need more than one machines keeper in the system.

To conclude this chapter, we developed an example of a peer-to-peer network in Mobile CSP || B framework. In addition, by having the powerful tool supports and effective theorems proved in this thesis, we were able to successfully verify the consistency of our example. Furthermore, it was revealed that for a relatively small example like our case study, a lot of proofs is required in consistency verification of the system. Therefore, more effort is needed in consistency verification of large systems. Moreover, by doing this case study we realised that in developing a large system containing several controllers and machines, it would be more efficient if we design some extra processes in our system as the machines keepers which do not work with machines and they are only used to keep the machines and to exchange the machines in the system during the execution.
Chapter 5

Related works and comparison

In this chapter, some relevant frameworks, software tools and specification languages are reviewed and compared with Mobile CSP \( \parallel B \). We begin with the review of verification software tools, model checkers and languages created for concurrent, distributed and dynamic systems. Then, we discuss how other combinations of a state-based formal method and a process algebra compare with our framework. Finally, we discuss how other two combinations of CSP and B compare with ours.

5.1 SPIN (Simple Promela Interpreter)

SPIN [9] is a software tool for formal verification of concurrent and distributed systems written in a modelling language called Promela (Process Meta Language). The syntax notation of Promela is based on Dijkstra's guarded command language [24] and the CSP language. This combination is able to describe state transitions happening in order. In a Promela model, a number of processes execute concurrently in the system. They can communicate with each other by using shared variables and channels. SPIN is able to check the consistency of a model system specified in Promela and to detect different errors in the system such as deadlocks and assertion violations.

Despite the fact that CSP processes can describe solely the behavioral aspects of the system without concerning state transitions, data aspects are described in Promela model in the body of processes. Promela accepts different types of variables such as integers, arrays and records. There are global variables which processes can share with each other. Also, each process can have its own local
variables which are used only by that process. The body of a process consists of a sequence of statements. If a statement contains a guard, it can be executed only if the guard is true. If a statement cannot be executed, the execution of that process is blocked until that statement can be executed.

Processes can communicate with each other via channels such that one process can put a message into a channel and the other process can get a message out of that channel. This communication can be either synchronous or asynchronous. In asynchronous communication, processes execute independently and the statements of different processes do not occur at the same time. However, in synchronous communication, sending a message by one process and receiving the message by another process should occur at the same time.

The communication between processes in SPIN has some differences with the original constructs of CSP. In SPIN, each channel has a particular capacity. As a result, a process can put a message into a channel if the channel is not already full and also a process can get a message out of a channel if the channel is not empty. In addition, communications are limited through channels: each process can either send a message or receive a message through a channel. So, a process cannot have both input and output through a single channel at the same time. Moreover, in synchronous communication, the synchronisation can only happen between two processes such that one should have the send statement and the other one should have the receive statement. As a result, more than two processes cannot synchronise with each other through a channel. Furthermore, unlike CSP specifications in which the number of processes are fixed at the beginning and during the execution, in SPIN new processes can be dynamically created at any point during the execution.

Given a model system specified in Promela, SPIN can be used as a simulator to find property violations in the system. In simulation, if there is a choice point at which there are more than one enabled transitions, SPIN can either simulate the system by randomly choosing a transition branch or it can ask the user to choose a transition branch. In both cases, SPIN cannot search all possible traces to find the errors. The reason is that in random simulation mode, SPIN might not always find the errors as it simulates the system randomly. Also, SPIN in guided simulation mode might not always find the errors as the user, who chooses a transition branch, may choose a transition branch which does not have an error.

The main success factor of SPIN is its ability to verify Promela models by automatic exhaustive searching. SPIN can automatically search all possible transition branches in the system to find the errors. This means that at each choice point, SPIN chooses an unexplored transition branch and after finishing the branch, it comes back to that choice point and chooses another transition branch which has not been explored before. If SPIN detects an error, it outputs the information
about the error. In SPIN’s output, the user can find the execution path and the number of execution steps in which that error occurs. The error that SPIN finds is not always the first error which occurs in the system. If it finds a long violating trace, it will be difficult for a user to follow the trace to find the reason of that error. To solve this problem, users are able to tell SPIN to search for shorter violating traces or even to search for minimal length error traces. They can set a bound on the depth of SPIN’s search. So, SPIN will then search for a particular length of traces to find errors.

5.2 SMV

SMV [8] is a model checker for concurrent systems written in SMV input language. A program in the SMV input language consists of one or more modules and the system specification. A module contains a number of assignment statements. SMV can be used as a tool for checking the program against the specification. It checks whether the specification is satisfied. If SMV detects that the specification is violated, it produces a counterexample and shows an execution path to the violation.

In a SMV program, all of the assignment statements are executed in parallel and at the same time. However, it is possible to define processes in a SMV program in order to make them execute asynchronously. A process is an instance of a module and like modules the body of a process contains of some assignment statements. During the execution, the program chooses a process non-deterministically and executes all the assignment statements in that process in parallel. After finishing this process, the program chooses another process again non-deterministically and starts to execute the new process. In other words, at each stage of the execution, only one process can be executed and this process is chosen non-deterministically. Therefore, processes can be used for assignments which are not wanted to be executed concurrently in the system. However, the disadvantage is that there is no control on the order of the execution of the processes as the execution of the processes is chosen non-deterministically.

5.3 VeriSoft

VeriSoft [11, 33, 32, 34] is a model checker which automatically searches for errors such as: deadlocks, divergences and assertion violations in programs written in different programming languages such as C, C++ and Tcl. It is a tool for analysing and testing concurrent systems in which several processes execute concurrently in the system. In the case of detecting an error, VeriSoft details the cause of the error by performing a scenario leading to the error.
The key difference between VeriSoft and our approach is that VeriSoft is used to test and check the correctness of software applications which have been developed in programming languages. However, our framework is an approach to verify system specifications and to check the consistency of the specifications before generating code. The verified specifications then can be written in different programming languages.

5.4 Mobile UNITY

Mobile UNITY [47], an extension of the parallel program design language UNITY [20], is a language and proof logic for specifying and reasoning about concurrent mobile systems. In Mobile UNITY, components can move around and execute at different locations and they can interact and communicate with each other during the execution. In Mobile UNITY notation, mobility is modeled as the change of the location of components. In other words, the change in the location of components provides means to model movement in the system. It allows the description of location-sensitive behaviour, e.g., interaction at the same place, or within a certain distance. Each component in Mobile UNITY has a distinguished location variable. The location of each component is modeled by assignment of a value to its location variable. The movement of a component is modeled by the change of value of its location variable.

A Mobile UNITY specification consists of a number of programs, a Components section and an Interactions section. A program consists of some variables and a set of assignment statements. The Components section introduces all the components that exist in the system. In Mobile UNITY, dynamic creation of new components is not allowed during the execution. Each component of the system is an instance of a program. It is possible that some or all components to be the instances of the same program. The Interactions section contains statements which define how different components interact and communicate with each other in the system.

Mobile UNITY proof logic is employed to verify the safety and liveness properties of a system expressed in the Mobile UNITY notation. Mobile UNITY has no notion of refinement.

We believe using CSP language in our approach makes flow of control more explicit in contrast to approaches that use control variables. In addition, Mobile CSP $\parallel B$ framework supports a refinement approach: it enables the refinements of components to be a substitute of the original components in the system while the system consistency is guaranteed to remain. Furthermore, as our framework can be coded into the original constructs of CSP and B, we are able to use software
tools of CSP and the B-Method to analyse, animate and check the behaviour and the consistency of the two parts of a mobile system separately.

5.5 Occam-$\pi$

Occam [12] is a programming language based on CSP in which a number of processes execute concurrently and communicate with each other through channels. An Occam process contains a number of actions which are either an assignment, an input or an output. Unlike CSP, processes in an Occam program are able to process data.

Occam-$\pi$ [6, 13, 14] is an extension of Occam which uses $\pi$-calculus semantics as well as CSP in order to create more dynamic systems. In Occam-$\pi$ programs, mobile data, mobile channels and mobile processes can be declared.

In communication between processes in Occam-$\pi$, whenever a process outputs a mobile variable, it loses that variable and does not have any access to it afterwards. This is similar to our work in which machine references can be passed around the system and whenever a process passes a machine reference to another controller, it does not have any access to that machine reference afterwards.

Mobile processes in Occam-$\pi$ can move around the network through channels. A mobile process is able to suspend itself, then move from its local environment to another environment, and be reactivated in the new environment. After reactivation, the process is executed from the same state at which it suspended.

A channel in Occam has two ends which are two processes communicating with each other through that channel. Depending on the direction of the channel, one process is client and the other one is server process. In Occam-$\pi$, a mobile channel is able to move around the network. As a result, different processes can communicate with each other through a mobile channel at run-time.

The idea of moving channels in Occam-$\pi$ is similar to our Mobile CSP $\parallel$ B architecture in which machine references can move around the system during the system execution. However, in our framework, B machines always have one end of their reference channel and this is only the other end which can belong to different processes in the system. Another difference is that in Mobile CSP $\parallel$ B, only machine references are mobile channels. Communication channels and control points are static channels in Mobile CSP $\parallel$ B architecture and they are fixed in the system from the beginning and during the execution. This enabled us to present theorems and techniques for establishing consistency in our mobile architecture.

Despite Mobile CSP $\parallel$ B which does not have dynamic component creation, in Occam-$\pi$, new processes can be dynamically created at run-time.
Unlike in Occam-\(\pi\), we are able to verify the consistency of the systems specified in Mobile CSP \(\parallel\) B. In principle, our work can be coded into Occam-\(\pi\) so a system can be first verified in Mobile CSP \(\parallel\) B and then the verified system can be coded into Occam-\(\pi\).

### 5.6 PiOZ

PiOZ [66] is a specification technique which is an integration of Object-Z and \(\pi\)-calculus. This framework is used to model dynamic systems by embedding \(\pi\)-calculus into Object-Z. In PiOZ, \(\pi\)-style process definitions are included into Object-Z classes such that a class has both Object-Z part and \(\pi\) process part. The \(\pi\) process part is an extension of the \(\pi\)-calculus process syntax in which the body of a process can include operations of that class and also include the Object-Z state guards.

By using \(\pi\)-calculus, communication channels between components can be passed around the system and also new channels can be dynamically created between components in a system specified in PiOZ. Our Mobile CSP \(\parallel\) B framework also supports the movement of channels. However, in our approach, only channels for communicating with machines (machine references) are mobile and can be passed around the system. In addition, it is not possible to create new channels in Mobile CSP \(\parallel\) B.

[66] states that invariant and liveness properties of systems in PiOZ can be verified by adopting an assertional-style proof logic presented in [26]. [66] provides an example and explain how the assertional-style proof can be used to prove the no message loss property in a system specified in PiOZ. We believe the consistency verification strategy provided in our framework is more explicit. In addition, software tools in CSP and B can also be used to reason and check different aspects of the system behaviour.

One limitation of the PiOZ approach is that only monadic channels are used in this formal framework. This means that a message sent or received along a channel can consist of only one name. In addition, currently there is no tool support for PiOZ.

### 5.7 ZCCS

ZCCS [31] is a specification language which is a combination of Z and CCS. A ZCCS specification consists of two parts: Z part and CCS part. First, the data are introduced and defined in the Z part by using the Z language. These defined data are then used in the CCS part of the specification.
One problem with ZCCS is that it is not possible to invoke Z operations directly by their name in CCS part of the specification. Instead, a group of agents should be used to apply an operation schema. So, using Z operation schemas in Z part can produce very complex CCS part of the ZCCS specification.

Another problem is that in CCS specifications we only can have either input or output carried by a channel. So it is not possible to have both input and output in a single channel at the same time in a ZCCS specification.

Mobile CSP || $B$ framework enables us to specify and verify concurrent systems with mobile architecture. However, ZCCS do not support mobility.

### 5.8 CSP-OZ

CSP-OZ [26] is a combination of CSP and Object-Z. It is an extension of Object-Z in the way that an Object-Z class is defined by using the communication channels and CSP processes as well as the standard Object-Z class definitions. A CSP-OZ class contains of three parts: channel definitions (the interface of the class), CSP processes, and Z schemas which are defined the same as in Object-Z. Each operation schema corresponds to a channel. So the type of a channel is a schema type. The CSP processes describe the behaviour of that class by using the channels defined in the interface. Therefore, each event in the body of the processes corresponds to an operation with its parameters.

The main strength of CSP-OZ is that CSP operators such as parallel composition, hiding, interleaving, internal choice and external choice can be applied to CSP-OZ classes. However, CSP-OZ is not supported by any software tools.

A failure-divergence semantics of CSP-OZ is defined in [29] in the way that it defines a failure-divergence semantics of CSP-OZ classes with empty CSP-part which contains only the channel definitions and the Z-part. Then, the semantics of a CSP-OZ class is given by the parallel composition of two CSP-OZ classes; one of them with empty CSP-part and only contains the channel definitions and the Z-part, and the other one with empty Z-part and only contains the channel definitions and the CSP-part.

In contrast to Mobile CSP || $B$, CSP-OZ does not support mobility. In CSP-OZ, each CSP-OZ class has its own CSP-part which describe the behaviour of the class. In other words, each class has a specific behaviour defined in CSP-part of its specification. Our approach, however, we believe has more dynamic architecture. In Mobile CSP || $B$, as B machines can be exchanged between controllers, B machine's behaviour is controlled by different CSP controllers during the execution. As each controller may invoke different operations of a B machine, the B
5.9 Object-Z/CSP

Object-Z/CSP [64] is a combination of Object-Z and CSP for specifying, refining and verifying concurrent systems. In Object-Z/CSP, Object-Z classes are given CSP failures-divergences semantics. This enables the creation of a combined specification in which Object-Z classes in Z part of the specification can be used as CSP processes in CSP part of the specification. After specifying the component processes as Object-Z classes, the whole system is then specified by using CSP operators to describe how the components interact with each other in the system.

Classes can communicate and interact with each other on their common named operations. For instance, if two classes have an operation with the same name $op$, these two classes synchronise with each other on the operation $op$. This can be used to model message passing communication between classes. Message passing communication can be modelled by renaming the two operations, each in one class, to a common name. For instance, assuming that an operation $op_1$ consisting of output in a class should synchronise with operation $op_2$ consisting of input in another class. In order to model message passing between these two classes, operations $op_1$ and $op_2$ can be renamed to a common name so they can then synchronise with each other. If two classes have an operation with the same name $op$ but we do not want them to synchronise with each other, the operation $op$ can be renamed to different names in these two classes in order to prevent them from synchronising.

[64] presents two methods to verify the refinements of Object-Z classes in the combined specifications. The first method uses the CSP definition of refinement. In this method, the failures of a class and the failures of its refinement are calculated and compared with each other. The result should establish that the failures of the refinement class is a subset of the failures of the abstract class. [64] also defines a state-based method to verify the refinements of the Object-Z classes. The method is based on the work presented in [39] where two refinement relations, called upward and downward simulations, are introduced and are proved to be sound and complete with respect to CSP refinement. [64] adapts the idea of upward and downward simulations presented in [39] and introduces two upward and downward simulation rules for classes in the integrated Object-Z/CSP notation. In order to establish that a class $C$ is a refinement of a class $A$, these rules must be hold between two classes $C$ and $A$.

To verify properties of the CSP system specifications in Object-Z/CSP, both state and behavioural properties are needed to be considered as Object-Z/CSP
is a combination of Object-Z and CSP. [64] uses the laws of the CSP operators presented in [36] and the logic for Object-Z presented in [63] to verify the CSP system whose components are Object-Z classes instead of CSP processes.

Object-Z/CSP is a close work to CSP-OZ as they are both integrations of Object-Z and CSP. Similar to CSP-OZ, Object-Z/CSP gives CSP failures-divergences semantics to Object-Z classes. However, in Object-Z/CSP there is no CSP-part inside of classes. In fact, Object-Z classes in Object-Z/CSP are specified in accordance to Object-Z notations and the specified classes are then used as CSP processes in CSP part of the system. This is similar to what we have done in our work: we define the CSP semantics for B machines which then enables us to use B machines as CSP processes in our system. In addition, like Object-Z classes in Object-Z/CSP, B machines in CSP \( B \) are specified in accordance to B machine specification notations and then the specified B machines are considered as CSP processes in CSP part of the system and they can work in parallel with CSP processes in the system.

Our Mobile CSP \( B \) framework provides an extra ability in specifying and verifying concurrent systems. It enables us to specify and verify concurrent systems with mobile architecture as well as static architecture.

In addition, we do not need to provide any methods to verify the refinements of components in our system. Our system contains two kinds of components: B machines and CSP processes. A refinement of a B machine can be verified by using B tools such as BToolkit, and refinements of CSP processes can be verified by using FDR. As we have proved in Chapter 3, refinements of components can be used instead of components in the system and the substitution does not have any effect on the system consistency properties.

## 5.10 Circus

Circus [3] is a combination of CSP and Z for specifying concurrent systems. It is a language which is able to design and specify both state and behavioural aspects of a system. As a combination of Z and CSP, Circus describes the data aspects of the system by Z and the behavioural aspects of the system by CSP. It also includes a refinement technique which is used to refine the abstract Circus specifications. Refinement can happen in different levels from abstract specification to an implementation.

In [30], a translating tool called JCircus is presented which automatically translates a Circus program into Java. This translator accepts a Circus program written in LATEX and automatically produces a Java program which can be then used for animation and simulation.
In contrast to Circus for which the software tools have to be specially created, we can code a Mobile CSP $\parallel B$ model into the original constructs of CSP and B and as a result we are able to use the comprehensive and highly efficient software tools of CSP and the B-Method to verify the behaviour and the consistency of each part of the system separately.

In addition, Mobile CSP $\parallel B$ provides the facility to describe the systems in which data aspects can be updated and accessible for different controllers in the system during the execution. This is why we believe our mobile framework can be used to design peer-to-peer networks. However, this architecture cannot be designed in Circus specification. In other words, one important advantage of our work is the ability of modelling mobile systems which is not possible in the current version of Circus. Recently, [67] presented semantics of mobile processes in Hoare and He’s Unifying Theories of Programming (UTP) [37]. The future objective of this work is to use the presented UTP semantics of mobile processes to include mobility into Circus in such a way that processes can move around in the system.

5.11 $\pi | B$

[40] introduces $\pi | B$, a framework which is based on a combination of $\pi$-calculus and the B-Method. As $\pi$-calculus is used to describe mobile systems, this formal framework provides specifications in which the dynamic patterns of behaviour of a system is described by $\pi$-calculus, and the data aspects of the system is described by the B-Method. [40] illustrates this mobile combination using an example of a peer-to-peer network in which B machines, like nodes in a peer-to-peer network, can join or leave the interaction regularly.

The system is allowed to dynamically create multiple instances of B machines and each instance has its own state. For each instance of a B machine, a unique channel is referred to as the machine reference. This channel is used to communicate with B machines. In order to execute operations of B machine instances, the $\pi$ processes which communicate with the B machine instances, referred to as mediators in [40], call operations of B machine instances through machine references. In other words, a B machine receives the requests to execute its operations via its unique machine reference. Similar to our work, machine instances in $\pi | B$ are mobile and they can be transferred between mediators along control point channels.

[40] presents a theorem for establishing the divergence freedom of the combined system which means that all operations of B machine instances are called within their precondition.
The framework presented in [40] is similar to our Mobile CSP \( \parallel B \) architecture. However, there are some major differences between \( \pi \mid B \) and our approach. Using \( \pi \)-calculus instead of CSP in system specification makes some differences in system behaviour. \( \pi \)-calculus is a formal method which is able to describe mobile systems and explain their behaviours whereas, we have created mobility in CSP \( \parallel B \) approach which had already been a static architecture.

In \( \pi \mid B \), different instances of B machines can be created in the system during the execution. However, in Mobile CSP \( \parallel B \) architecture, all B machines which work in the system exist from the beginning and new machines cannot be created in the system during the execution.

In order to establish the consistency of the combined system, [40] has only considered the divergence freedom of the system. It provides a theorem for establishing divergence freedom in \( \pi \mid B \) architecture without concerning about deadlock in system verification. Whereas, we have been able to provide theorems and techniques for establishing not only the divergence freedom but also deadlock freedom in our mobile system.

In order to allow B machines to be considered in parallel with \( \pi \) processes, a \( \pi \)-style operational semantics is provided for B machines in \( \pi \mid B \). However, the semantics is only for B machines whose operations do not have input and output parameters. As a result, operations with inputs and outputs cannot be used in B machines in \( \pi \mid B \). We believe the significant advantage of our work is that B already has a CSP-style semantics through Morgan's action systems [50]. In addition, in our Mobile CSP \( \parallel B \) framework, B machines are allowed to have operations with inputs and outputs. Therefore, a B machine and its CSP controller can send or receive values from each other while working in parallel.

### 5.12 CSP2B

In [18], Butler presents CSP2B, a tool for describing a combination of CSP and B in a system. CSP2B is a tool which accepts a CSP-like specification and converts it to a B machine. We write the CSP-like specification according to the source notation of CSP2B. Some features of CSP such as internal choices and event hiding are not supported by CSP2B. After writing the CSP-like specification, CSP2B generates a B machine from it. The generated B machine has the same information and its operations are named the same as events in the CSP-like specification.

CSP2B is a tool to control the execution order of the operations in a B machine. To achieve this, we enter the B machine and the CSP-like specification. In the CSP-like specification, we should add a clause called CONJOINS with the name
of the B machine. Clause \textit{CONJOINS} signifies that the CSP-like specification controls the execution order of the operations in the B machine. CSP2B then generates a B machine from the CSP-like specification. This generated machine includes the original B machine by using clause \textit{INCLUDES} following by the name of the original B machine. Each operation of the generated machine includes a guarded call to the corresponding operation of the included machine. This guarding of the call in each operation of the generated machine ensures that the order of execution of operations is according to the CSP-like specification.

CSP2B is a related work to our approach in the way that a CSP specification controls the execution order of the operations in a B machine. Also, similar to Mobile CSP $\parallel B$, in CSP2B it is possible to have more than one CSP processes working in parallel with each other as a controller for a B machine.

A major limitation with CSP2B is that a CSP process or a parallel composition of CSP processes can control the execution order of only one B machine's operations. However, in Mobile CSP $\parallel B$, a CSP process or a parallel composition of CSP processes can control the execution order of several B machines' operations. In addition, we are able to have mobile systems in which CSP controllers are able to exchange their controlled B machines during the execution.

\subsection{5.13 ProB}

ProB [7, 42, 19, 43] is a B software tool for animation, model checking and consistency checking. ProB is an animator which can be used to manually check deadlock freedom of a B machine. The user can try execution of different sequences of operations to find out if there is a deadlock in the machine. There is an enabled operations list on the screen in which the name of some operations are appeared at each stage of the execution. This means that only one of the operations written in that list can be executed next. So, the user can only choose the next operation from one of these operations in the list. If there is no operation in the list, it means that deadlock has happened at that stage of the execution and no more operation is able to be executed next. Also, ProB detects any divergence while checking deadlock in the B machine. It treats preconditions as guards in the system execution which means that whenever the precondition or guard of an operation is not satisfied, this operation will not be appeared in the enabled operations list and it cannot be executed as the next operation. In other words, ProB is an animator which does not allow divergence to happen during the execution.

ProB can automatically check the deadlock freedom of a B machine without execution which means that we do not need to try execution of sequences of operations to find out if there is a deadlock in the machine. However, deadlock
freedom can be checked for a limited number of executions which can be chosen by the user.

As well as a software tool for $B$, ProB can be used as an animator and consistency checker for the previous static $CSP \parallel B$ approach. It is able to accept both $B$ machine and its CSP controller specification together and check the consistency of the parallel combination. Among the operations enabled to happen next by the controller at each stage, ProB only offers operations to be executed next in enabled operations list on the screen whose preconditions and guards are satisfied. ProB can also automatically check the consistency of the combined system. However, it is not possible to check the consistency in a combined system containing non-terminating processes as ProB can only check the consistency for a limited number of executions as described above. Similar to CSP2B, ProB can solely check the consistency of the parallel combination of one $B$ machine and its controller. It is not able to support more dynamic and more interactive combined systems in which a CSP process can work concurrently with more than one $B$ machine, and each machine can be passed from one process to another process during the execution.
Chapter 6

Conclusion and Future Work

In this chapter we present our conclusions, provide a summary of the major points and evaluate the contributions of this thesis. In addition, we present our ideas for future research directions.

6.1 Conclusion

CSP $\| B$ is an approach which combines state and event based models. Its advantage is that complex systems can be described and their verification ensures consistency of the models.

The consistency verification which had been presented before in CSP $\| B$ did not cover all possible kinds of combined communicating systems which can be modelled in this architecture. For instance, there had been no consideration for consistency verification of a system in which $B$ machines contain both preconditioned and guarded operations. Also, having $B$ machines which contain only guarded operations, we were only able to check the deadlock freedom of each controlled component separately and it was not possible to check the deadlock freedom of the whole system. In Chapter 2, we successfully completed the consistency verification of CSP $\| B$ by providing new theorems for establishing divergence freedom and deadlock freedom of the systems that had not been covered before.

The main contribution of this thesis is introducing Mobile CSP $\| B$, a new version of CSP $\| B$ which supports the description of mobility. This additional functionality is suitable for modelling agent systems or peer-to-peer networks where consideration of mobility is important. In Chapter 3, we have extended
approach to include mobility and we have developed a formal framework so that we can verify the consistency of \( CSP \parallel B \) specifications that include mobile aspects. This allows us to extend the range of specifications that we can write in \( CSP \parallel B \) and retains our philosophy of not changing the underlying \( CSP \) and \( B \) semantics. In Mobile \( CSP \parallel B \) architecture, each CSP controller can work with more than one B machine at the same time during the execution. A parallel combination of CSP processes act as the controller for the B machines, and each single B machine can be controlled by different CSP processes during the execution. By introducing mobility, each CSP process can receive a (mobile) machine or give it to another CSP process during the execution. Each machine has a unique machine channel called machine reference. It is the only channel through which a CSP controller and a B machine can communicate with each other. Machine references are mobile channels and can move around in the system during the execution. They can also be treated as data and can be passed between CSP processes through static channels (control points) in the system. Controllers exchange machines between each other by exchanging the machine references.

In addition, in Chapter 3 we defined and verified the conditions which guarantee the divergence freedom and deadlock freedom consistency of the systems specified and designed in Mobile \( CSP \parallel B \). We provided a theorem to establish divergence freedom of the whole mobile combined communicating system containing several CSP controllers each controlling several B machines, by establishing properties for each CSP controller separately. In Mobile \( CSP \parallel B \), divergence freedom verification must include two steps. First, the CSP part (the parallel combination of CSP controllers) must be established to be divergence free. This can be done by using FDR. Next, it must be established that divergence will never arise from B machines while working in parallel with different CSP controllers in the system. Divergence arises in a B machine when its operations are called outside their precondition by the controllers. Thus, the second step in divergence freedom verification is to establish that the operations of all machines in the system are always called inside their precondition by the controllers during the execution. In Chapter 3, we presented a proof strategy in which each controller is separately checked for divergence freedom on machines it controls at some point during its execution, and then all of these individual checks are combined to an overall divergence freedom result. We also provided a theorem for establishing deadlock freedom of mobile combined communicating systems. However, our deadlock freedom verification is concerned with divergence free systems in which B machines do not contribute to any deadlocking behaviour. In other words, we restricted the B machines to preconditioned operations. Our theorem states that deadlock freedom of the parallel combination of CSP controllers implies deadlock freedom of the overall system.

In Chapter 4, we used our Mobile \( CSP \parallel B \) framework to specify and verify
6.2. FUTURE WORK

an example of a peer-to-peer network. We detailed all steps, from developing a Mobile CSP \( \parallel B \) model of a flight tickets sale system, to verifying its consistency. The theorems we proved in Chapter 3 were successfully capable at verifying the consistency of our mobile combined communicating system. The case study also demonstrated the ability of the language to describe CSP controllers interacting with numerous machines at the same time. During the consistency verification of our case study, it was revealed that for a relatively small system like our flight tickets sale system, a lot of logical proofs were required. This might be difficult and complicated for people who do not have a good mathematics background. Consistency verification of large systems or the systems with more functionalities will require many more and complex logical proofs. Therefore, we believe that for the weakest precondition proof to be applicable on any large scale, tool support would be necessary.

Our Mobile CSP \( \parallel B \) framework adopts similar concepts to the architecture proposed in \( \pi \mid B \). However, \( \pi \mid B \) was limited to systems without inputs and outputs and was restricted to a framework to support divergence freedom verification. In Mobile CSP \( \parallel B \), we can deal with specifications that contain inputs and outputs and in addition to divergence freedom, we can check for deadlock freedom. These two checks are the verification that should be carried out in order to ensure that a Mobile CSP \( \parallel B \) specification is consistent. In [54] Roscoe introduces a new operator in the CSP+ language (a variant of CSP) which brings mobility into the system in the way that the right to use particular events are transferred from one process to another along special rights channels. As a result, the alphabets of the processes change dynamically through the execution. Our approach is similar, in that channels can be transferred between processes. However, our work is motivated by the desire to retain access to the supporting CSP and B toolsets, and so we aim to minimise the extension required to enable the form of mobility we aim to model.

The approach taken in \( \pi \mid B \) also allows dynamic creation of machines and channels, and reconfiguration of the network. In contrast, Mobile CSP \( \parallel B \) provides a more static network between the controllers. However, developing the framework to enable reconfiguration, and dynamic machine and channel creation would be an interesting avenue of future research which is discussed in the next section.

6.2 Future Work

In this section, possible extensions to Mobile CSP \( \parallel B \) are discussed. The majority of future work plans listed below are concerned with bringing the description of dynamic patterns and more aspects of mobility into the framework.
6.2.1 Mobile control point channels

In Mobile CSP $| B_1$, we have introduced mobile channels as the machine references which can move around in the system during the execution. One future work plan could be to extend the mobility in our current architecture and to include mobile control points instead of the static one in Mobile CSP $| B$. Our current framework is focused on control points as dedicated channels between two controllers which pass machine references in one direction from one controller to another. In other words, a control point is a one way channel fixed between two controllers in the system. By letting control points move around in the system, they can be used by any two controllers for passing machine references. Therefore, fewer control points will be needed in the system which means we can abstract a large system to a smaller one with fewer channels.

Similar to machine references, in this new architecture control points can also be passed between controllers in the system. In other words, control point channels can be passed as data values through communication channels. The difference is that B machines always have one end of their reference channel and the other end can be passed between controllers during the system execution. However, both ends of a control point must be passed to the two controllers who want to use that control point to exchange a machine reference between each other. Also, like machine references, when a controller passes an end of a control point to another controller, it must not be able to use that control point anymore. This ensures that at any one time, only two controllers can use a control point to exchange machine references between each other.

As we intend to use CSP tools FDR and ProBE, we should describe the movement of control point channels in the system in standard CSP. For achieve this, we can use the same solution as machine references: introducing a new channel to represent the passing machine references through control points. The system can be coded for tool analysis into standard CSP by treating the control points as data values rather than as channels, and declaring a new channel $CCP$ (Channel for Control Points) which carries control points as the first value, and then machine references as further value. This new channel can be introduced in standard CSP specification as shown below:

```
channel CCP : CP.MR
```

By using channel $CCP$, any $cp! z$ and $cp? z$ in the body of the CSP controllers will be modelled as $CCP.cp! z$ and $CCP.cp? z$ respectively in the standard CSP specification of our system.

New theorems are needed to be proved and some new proof strategies will be needed for consistency verification of this new architecture. Our consistency verification strategy presented in Chapter 3 is based on the fact that each control
6.2. FUTURE WORK

Point has a unique sender and a unique receiver. This enabled us to assign assertions for each control point in the system and to introduce weakest precondition definition for traces of a controller. The assertion we assign for a control point is based on the receiver and what the receiver is going to do with the machine it receives along that control point. As the receiver controller is changed whenever a control point moves around the system, we are not able to assign a unique assertion for each control point in the system. Therefore, more thought and justification are required for establishing the consistency of these kinds of systems.

6.2.2 Extension for Deadlock freedom verification

The deadlock freedom verification of our Mobile CSP $\parallel B$ framework has been focused on the systems in which the operations of B machines are all pre-conditioned. An extension is required for deadlock freedom verification in order to allow B machines to have guarded operations or to have both pre-conditioned and guarded operations. This extension requires new theorems to be proved in order to establish the deadlock freedom of the whole system. The theorem we proved in Chapter 3, is based on the fact that any deadlock in the system can only arise from the CSP part of the system. So, if the parallel combination of controllers is deadlock free, then the whole system is deadlock free. However, this will not be the case anymore if B machines contain guarded operations. When the guard of a guarded operation is false, the execution of that operation is blocked. A theorem must be proved to say that: if at each stage of the system execution, there is at least one operation whose guard holds and this operation is one of the operations that can happen next in the system, then the whole system is deadlock free.

6.2.3 New machine creation

One possible extension for the current architecture is that new machines could be dynamically created in the system during the execution. Having new machines in the system means that new machine references must be dynamically created in the system during the execution.

In $\pi \mid B$, a machine generator is introduced which creates machine instances in the system. The binding operator $\nu$ in $\pi$-calculus is used to create new machine references. We might be able to use this idea in our Mobile CSP $\parallel B$ framework to create new machine references in the system.

We can design a special process in our system which generates new machine references and gives them to the controllers. In other words, this process is a machine generator which generates new machines during the execution. For each controller in the system, we should introduce a control point channel between
this controller and the machine generator. The direction of this control point
must be from the machine generator to the controller. In other words, it must be
an incoming control point for the controller. This control point is then used for
passing the new machine references from the machine generator to the controller.

In order to dynamically create new machine references in our architecture, we
can bring the \( \pi \)-calculus binding operator \( \nu \) into the syntax of the processes in
Mobile CSP \( \parallel B \). This operator enables the machine generator to create a new
machine reference which can then be used both as a channel and as data which is
passed through other channels in the system. For instance, a machine generator
\( P \) can be specified as: \( P = (\nu z)(cp!z \rightarrow P) \). In this case, \( P \) creates a machine
reference \( z \) and then it gives \( z \) to a controller through control point \( cp \) which is
situated between \( P \) and the receiver controller.

After adding the \( \pi \)-calculus binding operator \( \nu \) into our framework, we should
investigate how to describe machine reference creation in standard CSP. We have
already achieved a way to describe mobile channels in standard CSP specification.
We might be able to find a solution for this new operator as well. Otherwise,
we will not be able to use CSP tools FDR and ProBE to check the CSP part of
the system which is crucial in establishing the divergence freedom and deadlock
freedom of the system as described in Chapter 3. In this situation, a new proof
strategy must be provided for consistency verification of the system.

One solution for modelling the creation of new machines in standard CSP could
be to define a finite set, for instance \( \text{newMR} \), in advance whose elements can
be then used during the execution as new machine references of new created
machines in the system. By introducing this set, we can model the creation of
new machines. However, we restrict the creation of new machines to a finite
particular number of creation which is the number of elements in \( \text{newMR} \). The
machine generator above can be specified in standard CSP as shown below:

\[
\text{channel create : newMR}
\]

\[
P(S) = \text{create!}z \rightarrow \text{cp!}z \rightarrow P(S - \{z\})
\]

\[
\text{Generator} = P(\text{newMR})
\]

The event \( \text{create!}z \) represents the creation of a new machine reference in the
system. As the type of channel \( \text{create} \) is \( \text{newMR} \), new created machine references
will be always the elements of the set \( \text{newMR} \). Also according to the specification
of \( P(S) \), each time when a machine reference is created, it is chosen from one of
the elements in \( \text{newMR} \) which has not been already used for any other machines
created before in the system.
Bibliography


Appendices
AMN specification of a flight machine

MACHINE Flight
SETS RESPONSE = {Full, Empty, Available, YES, NO, IncorrectInput}
CONSTANTS Passport, seats
PROPERTIES Passport <: NAT & seats : NAT
VARIABLES customer, sold
INVARIANT customer <: Passport & sold : NAT
INITIALISATION customer := {} || sold := 0

OPERATIONS
response < -- sell(pp) =
PRE
pp : Passport & sold / = seats
THEN
IF pp : Passport - customer THEN
  customer := customer\/{pp} || sold := sold + 1 ||
IF sold + 1 = seats
THEN response := Full
ELSE response := Available
END
ELSE response := IncorrectInput
END;

response < -- cancel(pp) =
PRE
pp : Passport &
sold / = 0
THEN
IF pp : customer
THEN
customer := customer - {pp} ||
sold := sold - 1 ||
IF sold - 1 = 0
THEN response := Empty
ELSE response := Available
END
ELSE response := IncorrectInput
END
END;

response < -- empty =
BEGIN
IF sold = 0
THEN response := YES
ELSE response := NO
END
END;

response < -- full =
BEGIN
IF sold = seats
THEN response := YES
ELSE response := NO
END
END
END
Appendix B

CSP specification of Flight tickets sale system

datatype Flights = flight1 | flight2

datatype RESPONSE = Full | Empty | Available | YES | NO | IncorrectInput

datatype Operations = sell.Passport.RESPONSE | cancel.Passport.RESPONSE |
empty.RESPONSE | full.RESPONSE

datatype MR = mr1 | mr2

Agencies = {1, 2}

Passport = {1, 2, 3}

ref(flight1) = mr1

ref(flight2) = mr2

refinv(mr1) = flight1

refinv(mr2) = flight2

channel MC : MR.Operations
channel cp : Agencies.Agencies.MR
channel dp : Agencies.MR
channel ep : Agencies.MR
channel customerToBuy : Agencies.Flights.Passport
channel customerToCancel : Flights.Passport
channel request : Agencies.Agencies.Flights
channel sorry : Agencies.Agencies
APPENDIX B. CSP SPECIFICATION OF FLIGHT TICKETS SALE SYSTEM

channel ask : Agencies.Flights
channel require : Agencies.Flights
channel notHaveIt : Agencies
channel fullMachine : Agencies
channel emptyMachine : Agencies
channel MachinesIsFull : Agencies.Agencies

SellAgency(i, S) = P1(i, S)

\[ P1(i, S) = \text{customerToBuy.i?flight?pn} \rightarrow (\text{if member(ref(flight), S)} \text{ then } P2(i, S, flight, pn)) \]
\[ \text{else } P3(i, S, flight, pn)) \]
\[
\text{request?f!i?flight} \rightarrow (\text{if member(ref(flight), S)}
\text{ then } P4(i, S, j, flight))
\text{else (sorry.i.j} \rightarrow \text{P1(i, S)))} \]
\[
\text{ask.i?flight} \rightarrow (\text{if member(ref(flight), S)}
\text{ then } P5(i, S, flight))
\text{else (notHaveIt.i} \rightarrow \text{P1(i, S)))}
\]
\[
\text{dp.i?z} \rightarrow \text{P1(i, union(S, \{z\}))}
\]

\[ P2(i, S, flight, pn) = MC.\text{ref(flight).sell!pn?resp} \rightarrow \text{if resp == Full}
\text{then P6(i, S, flight)}
\text{else P1(i, S)}\]
\[ P3(i, S, \text{flight}, pn) = (\text{request} \cdot j \cdot \text{flight} \rightarrow \]
\[ ((\text{cp} \cdot j \cdot i \cdot w \rightarrow P2(i, \text{union}(S, \{w\}), \text{ref}\text{inv}(w), pn))]
\[ | (\text{sorry} \cdot j \cdot i \rightarrow P3(i, S, \text{flight}, pn))]
\[ | (\text{MachineIsFull} \cdot j \cdot i \rightarrow P1(i, S)))]
\]
\[ (\text{require} \cdot i \cdot \text{flight} \rightarrow \]
\[ ((\text{dp} \cdot i \cdot w \rightarrow P2(i, \text{union}(S, \{w\}), \text{ref}\text{inv}(w), pn))]
\[ | (\text{notHaveIt} \cdot i \rightarrow P3(i, S, \text{flight}, pn))]
\[ | (\text{fullMachine} \cdot i \rightarrow P1(i, S)))]
\]
\[ (\text{request} \cdot j \cdot i \cdot f \rightarrow \]
\[ (\text{if member}(\text{ref}(f), S)
\[ \text{then } \text{cp} \cdot i \cdot j \cdot \text{ref}(f) \rightarrow P3(i, \text{diff}(S, \{\text{ref}(f)\}), \text{flight}, pn)
\[ \text{else } (\text{sorry} \cdot i \cdot j \rightarrow P3(i, S, \text{flight}, pn)))]
\]
\[ (\text{ask} \cdot i \cdot f \rightarrow (\text{if member}(\text{ref}(f), S)
\[ \text{then } P10(i, S, \text{flight}, pn, f)
\[ \text{else } (\text{notHaveIt} \cdot i \rightarrow P3(i, S, \text{flight}, pn)))]
\]
\[ (\text{dp} \cdot i \cdot w \rightarrow \text{if } w == \text{ref}(\text{flight})
\[ \text{then } P2(i, \text{union}(S, \{w\}), \text{flight}, pn)
\[ \text{else } P3(i, \text{union}(S, \{w\}), \text{flight}, pn))]
\]

\[ P4(i, S, j, \text{flight}) = \text{cp} \cdot i \cdot j \cdot \text{ref} \rightarrow P1(i, \text{diff}(S, \{\text{ref}(\text{flight})\})) \]

\[ P5(i, S, \text{flight}) = \text{MC} \cdot \text{ref}(\text{flight}) \cdot \text{empty} \rightarrow \text{resp} \rightarrow \]
\[ \text{if resp == YES}
\[ \text{then } (\text{emptyMachine} \cdot i \rightarrow P1(i, S))
\[ \text{else } (\text{cp} \cdot i \cdot \text{ref}(\text{flight}) \rightarrow P1(i, \text{diff}(S, \{\text{ref}(\text{flight})\}))) \]
APPENDIX B. CSP SPECIFICATION OF FLIGHT TICKETS SALE SYSTEM

\[ P_6(i, S, \text{flight}) = (\text{ep}.i!\text{ref}(	ext{flight}) \rightarrow P_1(i, \text{diff}(S, \{\text{ref}(	ext{flight})\}))) \]

\[ \quad \quad \quad \text{[request].f!f} \rightarrow \text{if} \ \text{ref}(f) == \text{ref}(\text{flight}) \]
\[ \quad \quad \quad \text{then} \ (\text{MachinesFull}.i.j \rightarrow P_6(i, S, \text{flight})) \]
\[ \quad \quad \quad \text{else} \ P_7(i, S, \text{flight}, j, f) \]

\[ (\text{ask}.i?f \rightarrow \]
\[ \quad \quad \text{if} \ \text{ref}(f) == \text{ref}(\text{flight}) \]
\[ \quad \quad \text{then} \ (\text{ep}.i!\text{ref}(	ext{flight}) \rightarrow P_1(i, \text{diff}(S, \{\text{ref}(	ext{flight})\}))) \]
\[ \quad \quad \text{else} \ P_8(i, S, \text{flight}, f)) \]

\[ (\text{dp}.i?w \rightarrow P_6(i, \text{union}(S, \{w\}), \text{flight})) \]

\[ P_7(i, S, \text{flight}, j, f) = \text{if member}(\text{ref}(f), S) \]
\[ \quad \text{then} \ (\text{cp}.i.j!\text{ref}(f) \rightarrow P_6(i, \text{diff}(S, \{\text{ref}(f)\}), \text{flight})) \]
\[ \quad \text{else} \ (\text{sorry}.i.j \rightarrow P_6(i, S, \text{flight})) \]

\[ P_8(i, S, \text{flight}, f) = \text{if member}(\text{ref}(f), S) \]
\[ \quad \text{then} \ P_9(i, S, \text{flight}, f) \]
\[ \quad \text{else} \ (\text{notHaveIt}.i \rightarrow P_6(i, S, \text{flight})) \]

\[ P_9(i, S, \text{flight}, f) = \text{MC}.\text{ref}(f).\text{empty}?\text{resp} \rightarrow \]
\[ \quad \text{if} \ \text{resp} == \text{YES} \]
\[ \quad \text{then} \ (\text{emptyMachine}.i \rightarrow P_6(i, S, \text{flight})) \]
\[ \quad \text{else} \ (\text{ep}.i!\text{ref}(f) \rightarrow P_6(i, \text{diff}(S, \{\text{ref}(f)\}), \text{flight})) \]

\[ P_{10}(i, S, \text{flight}, \text{pn}, f) = \text{MC}.\text{ref}(f).\text{empty}?\text{resp} \rightarrow \]
\[ \quad \text{if} \ \text{resp} == \text{YES} \]
\[ \quad \text{then} \ (\text{emptyMachine}.i \rightarrow P_3(i, S, \text{flight}, \text{pn})) \]
\[ \quad \text{else} \ (\text{ep}.i!\text{ref}(f) \rightarrow P_3(i, \text{diff}(S, \{\text{ref}(f)\}), \text{flight}, \text{pn})) \]
\[
\text{ReturnOffice}(S) = R1(S)
\]

\[
R1(S) = \text{customerToCancelFlight}(pn) \rightarrow (\text{if member(ref(flight), S)}
\begin{align*}
    &\text{then } R2(S, \text{flight}, pn) \\
    &\text{else } R3(S, \text{flight}, pn)
\end{align*}
\]

\[
\text{require j flight} \rightarrow (\text{if member(ref(flight), S)}
\begin{align*}
    &\text{then } R5(S, \text{flight}, j) \\
    &\text{else } (\text{notHaveIt } j \rightarrow R1(S))
\end{align*}
\]

\[
\text{ep j z} \rightarrow R1(\text{union}(S, \{z\}))
\]

\[
R2(S, \text{flight}, pn) = \text{MC.ref(flight).cancel!pn?resp} \rightarrow \text{if resp == Empty}
\begin{align*}
    &\text{then } R4(S, \text{flight}) \\
    &\text{else } R1(S)
\end{align*}
\]

\[
R3(S, \text{flight}, pn) = \text{ask j flight} \rightarrow (\text{ep j w} \rightarrow R2(\text{union}(S, \{w\}), \text{refinv}(w), pn)
\begin{align*}
    &\text{notHaveIt } j \rightarrow R3(S, \text{flight}, pn) \\
    &\text{emptyMachine } j \rightarrow R1(S)
\end{align*}
\]

\[
\text{require } j f \rightarrow (\text{if member(ref(f), S)}
\begin{align*}
    &\text{then } R10(S, \text{flight}, pn, j, f) \\
    &\text{else } (\text{notHaveIt } j \rightarrow R3(S, \text{flight}, pn)))
\end{align*}
\]

\[
\text{ep j w} \rightarrow \text{if w == ref(flight)}
\begin{align*}
    &\text{then } R2(\text{union}(S, \{w\}), \text{refinv}(w), pn) \\
    &\text{else } R3(\text{union}(S, \{w\}), \text{flight}, pn)
\end{align*}
\]
APPENDIX B. CSP SPECIFICATION OF FLIGHT TICKETS SALE SYSTEM

\[ R4(S, \text{flight}) = dp.j!\text{ref(flight)} \rightarrow R1(\text{diff}(S, \{\text{ref(flight)}\})) \]

\[
\text{require} j?f \rightarrow \begin{cases} 
\text{if ref(f) == ref(flight)} \\
\text{then dp.j!ref(flight)} \rightarrow R1(\text{diff}(S, \{\text{ref(flight)}\})) \\
\text{else R8(S, flight, j, f))}
\end{cases}
\]

\[ ep?j?w \rightarrow R4(\text{union}(S, \{w\}), \text{flight}) \]

\[ R5(S, \text{flight}, j) = M\text{C. ref(flight). full?resp} \rightarrow \begin{cases} 
\text{if resp == YES} \\
\text{then R6(S, j)} \\
\text{else R7(S, flight, j)}
\end{cases} \]

\[ R6(S, j) = \text{fullMachine.j} \rightarrow R1(S) \]

\[ R7(S, \text{flight}, j) = dp.j!\text{ref(flight)} \rightarrow R1(\text{diff}(S, \{\text{ref(flight)}\})) \]

\[ R8(S, \text{flight}, j, f) = \begin{cases} 
\text{if member(\text{ref(f)}, S)} \\
\text{then R9(S, flight, j, f)} \\
\text{else (notHaveIt.j \rightarrow R4(S, flight))}
\end{cases} \]

\[ R9(S, \text{flight}, j, f) = M\text{C. ref(f). full?resp} \rightarrow \begin{cases} 
\text{if resp == YES} \\
\text{then fullMachine.j} \rightarrow R4(S, \text{flight}) \\
\text{else dp.j!ref(f)} \rightarrow R4(\text{diff}(S, \{\text{ref(f)}\}), \text{flight})
\end{cases} \]

\[ R10(S, \text{flight}, pn, j, f) = M\text{C. ref(f). full?resp} \rightarrow \begin{cases} 
\text{if resp == YES} \\
\text{then fullMachine.j} \rightarrow R3(S, \text{flight}, pn) \\
\text{else dp.j!ref(f)} \rightarrow R3(\text{diff}(S, \{\text{ref(f)}\}), \text{flight}, pn)
\end{cases} \]
\[ A(i) = \{ \text{customerToBuy}.i, \text{request}.i, \text{request}.j.i, \text{sorry}.i, \text{sorry}.j.i, \text{MachineIsFull}.i.j, \text{ask}.i, \text{require}.i, \text{notHaveIt}.i, \text{dp}.i, \text{MC}.z.sell, \text{ep}.i, \text{cp}.i, \text{cp}.j.i, \text{fullMachine}.i, \text{MC}.z.empty, \text{emptyMachine}.i \mid j : \{1, 2\}, z : \text{MR} \} \]

\[ B = \{ \text{customerToCancel}, \text{ask}, \text{require}, \text{notHaveIt}, \text{ep}, \text{MC}.z.cancel, \text{emptyMachine}, \text{dp}, \text{MC}.z.full, \text{fullMachine} \mid z : \text{MR} \} \]

\[ \text{Controller} = \text{ReturnOffice}() \parallel [B \parallel \text{union}(A(1), A(2))] (\text{SellAgency}(1, \{\text{mr}1\}) \parallel [\{ \text{request}, \text{sorry}, \text{MachineIsFull}, \text{cp} \} \parallel \text{SellAgency}(2, \{\text{mr}2\})) \]
Appendix C

wp proofs of Flight tickets sale system

+ $P_1(i, S)$

$P_1(i, S) = 1 \lor 2 \lor 3 \lor 4$

$\text{trans}(P_1(i, S)) = \text{CHOICE trans}(1) \lor \text{trans}(2) \lor \text{trans}(3) \lor \text{trans}(4) \text{ END}$

$[\text{trans}(P_1(i, S))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) =
[\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \land
[\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \land
[\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \land
[\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x))$

$1 = \text{customerToBuy.} i ? \text{flight?pn} \rightarrow \text{if ref(flight) } \in S$

$\quad \text{then } P_2(i, S, \text{flight, pn})$

$\quad \text{else } P_3(i, S, \text{flight, pn})$
APPENDIX C. WP PROOFS OF FLIGHT TICKETS SALE SYSTEM

\[ \text{trans}(1) = \]
\[ \text{PRE} \ i \in \text{Agencies} \text{ THEN} \]
\[ \text{ANY flight\,, \, pn WHERE flight : Flights} \land \ pn : \text{Passport} \text{ THEN} \]
\[ \text{IF } \text{ref}(\text{flight}) \in S \]
\[ \text{THEN } \text{rec} := P_2(i, S, \text{flight}, pn) \]
\[ \text{ELSE } \text{rec} := P_3(i, S, \text{flight}, pn) \]
\[ \text{END} \]
\[ \text{END} \]

\[ \left[ \text{IF } \text{ref}(\text{flight}) \in S \right. \]
\[ \left. \text{THEN } \text{rec} := P_2(i, S, \text{flight}, pn) \right] \]
\[ (\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) = \]
\[ \left. \text{ELSE } \text{rec} := P_3(i, S, \text{flight}, pn) \right] \]
\[ \text{END} \]

\[ \left( \text{ref}(\text{flight}) \in S \land \text{CLI}_{P_2}(i, S, \text{flight}, pn) \right) \]
\[ \lor \]
\[ \left( \text{ref}(\text{flight}) \notin S \land \text{CLI}_{P_3}(i, S, \text{flight}, pn) \right) = \]

\[ \text{CLI}_{P_1}(i, S) \land \text{flight} \in \text{Flights} \land \text{ref}(\text{flight}) \in \text{MR} \land \text{pn} \in \text{Passport} \]

\[ \left[ \text{trans}(1) \right](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) = \]
\[ i : \text{Agencies} \land \]
\[ \forall \text{flight\,, \, pn} \cdot (\text{flight} : \text{Flights} \land \text{pn} : \text{Passport} \Rightarrow \]
\[ \text{CLI}_{P_1}(i, S) \land \text{flight} \in \text{Flights} \land \text{ref}(\text{flight}) \in \text{MR} \land \text{pn} \in \text{Passport} \]

The result above shows that:
\[ (\text{rec} = P_1(i, S) \land \text{CLI}_{P_1}(i, S)) \Rightarrow \]
\[ \left[ \text{trans}(1) \right](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) \]
\[ \checkmark \]

\[ 2 = \text{request\,?j\,?flight} \rightarrow \text{if ref(\text{flight}) \in S} \]
\[ \text{then } P_4(i, S, j, \text{flight}) \]
\[ \text{else (sorry\,i\,j} \rightarrow P_1(i, S)) \]
trans(2) =
PRE i ∈ Agencies THEN
    ANY j, flight WHERE j ∈ Agencies ∧ flight ∈ Flights THEN
        IF ref(flight) ∈ S
            THEN rec := P_{4}(i, S, j, flight)
        ELSE trans(sorry. i.j → P_{1}(i, S))
    END
END

trans(sorry. i.j → P_{1}(i, S)) = PRE i ∈ Agencies ∧ j ∈ Agencies
    THEN skip
    END; rec := P_{1}(i, S)

[trans(sorry. i.j → P_{1}(i, S))](∀ 1 ≤ x ≤ 10 • (rec = P_{x} ⇒ CLIP_{x})) =
i ∈ Agencies ∧ j ∈ Agencies ∧ CLIP_{1}(i, 1)

\[
\begin{align*}
&\text{IF ref(flight) ∈ S} \\
&\text{THEN rec := P_{4}(i, S, j, flight)} \\
&\text{ELSE trans(sorry. i.j → P_{1}(i, S))}\end{align*}
\]

\[
\forall 1 ≤ x ≤ 10 • (rec = P_{x} ⇒ CLIP_{x})
\]

\[
\left(\begin{array}{c}
\text{ref(flight) ∈ S ∧ CLIP}_{1}(i, S, j, \text{flight}) \\
\text{∨} \\
\text{ref(flight) ∉ S ∧ i ∈ Agencies ∧ j ∈ Agencies ∧ CLIP}_{1}(i, 1)
\end{array}\right)
\]

\[
= j ∈ Agencies ∧ CLIP_{1}(i, 1)
\]

[trans(2)](∀ 1 ≤ x ≤ 10 • (rec = P_{x} ⇒ CLIP_{x})) =
i ∈ Agencies ∧ ∀ j, flight (j ∈ Agencies ∧ flight ∈ Flights ⇒
\quad j ∈ Agencies ∧ CLIP_{1}(i, 1))

The result above shows that:
(rec = P_{1}(i, S) ∧ CLIP_{1}(i, S)) ⇒
[trans(2)](∀ 1 ≤ x ≤ 10 • (rec = P_{x} ⇒ CLIP_{x}))
\begin{align*}
3 & \Rightarrow \text{ask}.i? \text{flight} \rightarrow \text{if} \ \text{ref} (\text{flight}) \in S \\
& \quad \text{then} \ P_5(i, S, \text{flight}) \\
& \quad \text{else} \ (\text{notHaveIt}.i \rightarrow P_1(i, S))
\end{align*}

\text{trans}(3) =
\begin{align*}
\text{PRE } i \in \text{Agencies} \ \text{THEN} \\
\hspace{1cm} \text{ANY} \ \text{flight} \ \text{WHERE} \ \text{flight} \in \text{Flights} \ \text{THEN} \\
\hspace{2cm} \text{IF} \ \text{ref} (\text{flight}) \in S \\
\hspace{3cm} \text{THEN} \ \text{rec} := P_5(i, S, \text{flight}) \\
\hspace{2cm} \text{ELSE} \ \text{trans}(\text{notHaveIt}.i \rightarrow P_1(i, S)) \\
\hspace{1cm} \text{END}
\end{align*}
\text{END}

\text{trans}(\text{notHaveIt}.i \rightarrow P_1(i, S)) =
\begin{align*}
\text{PRE } i \in \text{Agencies} \ \text{THEN} & \text{skip} \ \text{END}; \ \text{rec} := P_1(i, S) \\
\end{align*}

\begin{align*}
\text{trans}(\text{notHaveIt}.i \rightarrow P_1(i, S)) \left[ (\forall 1 \leq x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLIP_x) \right] \\
= CLIP_1(i, S)
\end{align*}

\begin{align*}
\left[ \begin{align*}
\text{IF} \ \text{ref} (\text{flight}) \in S \\
\text{THEN} \ \text{rec} := P_5(i, S, \text{flight}) \\
\text{ELSE} \ \text{trans}(\text{notHaveIt}.i \rightarrow P_1(i, S)) \\
\text{END}
\end{align*} \right] \left[ (\forall 1 \leq x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLIP_x) \right] \\
= CLIP_1(i, S)
\end{align*}

\begin{align*}
\text{trans}(3) \left[ (\forall 1 \leq x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLIP_x) \right] = \\
i \in \text{Agencies} \land \forall \text{flight} \cdot (\text{flight} \in \text{Flights} \Rightarrow CLIP_1(i, S))
\end{align*}

The result above shows that:

\begin{align*}
(\text{rec} = P_1(i, S) \land CLIP_1(i, S)) \Rightarrow \\
\text{trans}(3) \left[ (\forall 1 \leq x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLIP_x) \right]
\end{align*}
4 = dp.i?z → P_1(i, S \cup \{z\})

trans(4) =
ANY z WHERE z ∈ MR THEN
  SELECT z.sold ≠ z.seats
  THEN rec := P_1(i, S \cup \{z\})
END
END

\[ \text{trans}(4)(\forall 1 \leq z \leq 10 \bullet (\text{rec} = P_z \Rightarrow CLIP_z)) = \]
\[ \forall z \bullet (z \in MR \Rightarrow (z.sold \neq z.seats \Rightarrow CLIP_{P_1(i, S \cup \{z\}))}) \]

The result above shows that:
(rec = P_1(i, S) \land CLIP_{P_1(i, S)}) \Rightarrow
\[ \text{trans}(4)(\forall 1 \leq z \leq 10 \bullet (\text{rec} = P_z \Rightarrow CLIP_z)) \]
✓

The results above show that:
(rec = P_1(i, S) \land CLIP_{P_1(i, S)}) \Rightarrow
\[ \text{trans}(P_1(i, S))(\forall 1 \leq z \leq 10 \bullet (\text{rec} = P_z \Rightarrow CLIP_z)) \]
✓

* P_2(i, S, flight, pn)

\[ \text{trans}(P_2(i, S, flight, pn)) = \text{resp} \leftarrow \text{ref(flight).sell(pn)}; \]
\[ \text{IF} \ \text{resp} = \text{Full} \]
\[ \text{THEN} \ \text{rec} := \text{P}_0(i, S, \text{flight}) \]
\[ \text{ELSE} \ \text{rec} := \text{P}_1(i, S) \]
\[ \text{END} \]

\[ \begin{cases} 
\text{IF} \ \text{resp} = \text{Full} \\
\text{THEN} \ \text{rec} := \text{P}_0(i, S, \text{flight}) \\
\text{ELSE} \ \text{rec} := \text{P}_1(i, S) \\
\text{END} \end{cases} \]
\[ (\forall 1 \leq z \leq 10 \bullet (\text{rec} = P_z \Rightarrow CLIP_z)) = \]
\[
\begin{align*}
&\left( \text{resp} = \text{Full} \land CFLP_{P_6(i, S, \text{flight})} \right) \\
&\lor \\
&\left( \text{resp} \neq \text{Full} \land CFLP_{P_1(i, S)} \right)
\end{align*}
\]

\[
\text{[trans}(P_2(i, S, \text{flight}, pn))\text{]}(\forall 1 \leq z \leq 10 \cdot (\text{rec} = P_z \Rightarrow CFLP_{P_z})) = \\
\text{[resp } \leftarrow \text{ ref}(\text{flight}).\text{sell}(pn)](\text{resp} = \text{Full} \land CFLP_{P_6(i, S, \text{flight})} \lor \\
\text{resp} \neq \text{Full} \land CFLP_{P_1(i, S)}) \\
\]

\[
pn \in \text{Passport} \land \text{ ref}(\text{flight}).\text{sold} \neq \text{ ref}(\text{flight}).\text{seats} \land \\
\text{IF } pn \in \text{Passport - customer THEN} \\
\text{customer} := \text{customer} \cup \{pn\} || \\
\text{ref}(\text{flight}).\text{sold} := \text{ref}(\text{flight}).\text{sold} + 1 || \\
\text{IF } \text{ref}(\text{flight}).\text{sold} + 1 = \text{ref}(\text{flight}).\text{seats} \text{ THEN resp } := \text{Full} \text{ ELSE resp } := \text{Available} \text{ END} \\
\text{ELSE resp } := \text{IncorrectInput} \text{ END} \\
\text{END}
\]

\[
\left( \text{resp} = \text{Full} \land CFLP_{P_6(i, S, \text{flight})} \right) \\
\lor \\
\left( \text{resp} \neq \text{Full} \land CFLP_{P_1(i, S)} \right)
\]

\[
A = \\
\text{customer} := \text{customer} \cup \{pn\} || \\
\text{ref}(\text{flight}).\text{sold} := \text{ref}(\text{flight}).\text{sold} + 1 || \\
\text{IF } \text{ref}(\text{flight}).\text{sold} + 1 = \text{ref}(\text{flight}).\text{seats} \text{ THEN resp } := \text{Full} \text{ ELSE resp } := \text{Available} \text{ END} \\
\text{END}
\]

\[
\left( \text{resp} = \text{Full} \land CFLP_{P_6(i, S, \text{flight})} \right) \\
\lor \\
\left( \text{resp} \neq \text{Full} \land CFLP_{P_1(i, S)} \right)
\]
\[
\begin{align*}
\text{ref}(\text{flight}).\text{sold} + 1 = \text{ref}(\text{flight}).\text{seats} \land i \in \text{Agencies} \land S \subseteq MR \land \\
\forall k \in (S - \{\text{ref}(\text{flight})\}) \land k.\text{sold} \neq k.\text{seats} \land \text{ref}(\text{flight}) \in S \\
\lor \\
\text{ref}(\text{flight}).\text{sold} + 1 \neq \text{ref}(\text{flight}).\text{seats} \land i : \text{Agencies} \land S \subseteq MR \land \\
\forall k \in S \land k.\text{sold} \neq k.\text{seats}
\end{align*}
\]

\[
\begin{align*}
\text{IF } pn \in \text{Passport} - \text{customer} \text{ THEN} \\
\text{customer} := \text{customer} \cup \{pn\} || \\
\text{ref}(\text{flight}).\text{sold} := \text{ref}(\text{flight}).\text{sold} + 1 || \\
\text{IF } \text{ref}(\text{flight}).\text{sold} + 1 = \text{ref}(\text{flight}).\text{seats} \text{ THEN } \text{resp} := \text{Full} \\
\text{ELSE } \text{resp} := \text{Available} \\
\text{END} \\
\text{ELSE } \text{resp} := \text{IncorrectInput} \\
\text{END}
\end{align*}
\]

\[
\begin{align*}
\text{resp} = \text{Full} \land \text{CLI}_{P_b(i,S,\text{flight})} \\
\lor \\
\text{resp} \neq \text{Full} \land \text{CLI}_{P_a(i,S)}
\end{align*}
\]

\[
\begin{align*}
\text{CLI}_{P_b(i,S,\text{flight},pn)} \Rightarrow \\
(i \in \text{Agencies} \land S \subseteq MR \land \forall k \in (S - \{\text{ref}(\text{flight})\}) \land k.\text{sold} \neq k.\text{seats} \\
\land \text{ref}(\text{flight}) \in S)
\end{align*}
\]

\[
\begin{align*}
\text{CLI}_{P_b(i,S,\text{flight},pn)} \Rightarrow i : \text{Agencies} \land S \subseteq MR \land \forall k \in S \land k.\text{sold} \neq k.\text{seats}
\end{align*}
\]

\[
\begin{align*}
\text{CLI}_{P_a(i,S,\text{flight},pn)} & \Rightarrow \left( \text{ref}(\text{flight}).\text{sold} + 1 = \text{ref}(\text{flight}).\text{seats} \right) \\
\lor \\
\left( \text{ref}(\text{flight}).\text{sold} + 1 \neq \text{ref}(\text{flight}).\text{seats} \right)
\end{align*}
\]

\[
\begin{align*}
\text{CLI}_{P_a(i,S,\text{flight},pn)} \Rightarrow A
\end{align*}
\]
$\text{CLI}_{P_2(i,S,flight,pn)} \Rightarrow \text{CLI}_{P_1(i,S)}$

$\text{CLI}_{P_2(i,S,flight,pn)} \Rightarrow (\text{pn} \in \text{Passport - customer} \lor \text{pn} \notin \text{Passport - customer})$

The results above show that:

$\text{CLI}_{P_2(i,S,flight,pn)} \Rightarrow \left(\begin{array}{l}
\text{pn} \in \text{Passport - customer} \land A \\
\lor \\
\text{pn} \notin \text{Passport - customer} \land \text{CLI}_{P_1(i,S)}
\end{array}\right)$

$\text{CLI}_{P_2(i,S,flight,pn)} \Rightarrow (\text{pn} \in \text{Passport} \land \text{ref(flight).sold} \neq \text{ref(flight).seats})$

The results above show that:

$(\text{rec} = P_2(i,S,flight,pn) \land \text{CLI}_{P_2(i,S,flight,pn)}) \Rightarrow$

$[\text{trans}(P_2(i,S,flight,pn))](\forall 1 \leq x \leq 10 \cdot ($

$\text{trans}(P_2(i,S,flight,pn)) = 1 \lor 2 \lor 3 \lor 4 \lor 5$

$\text{trans}(P_3(i,S,flight,pn)) =$

$\text{CHOICE} \text{trans}(1) \lor \text{trans}(2) \lor \text{trans}(3) \lor \text{trans}(4) \lor \text{trans}(5) \text{ END}$
\(1 = \text{request.}\ i\ \text{flight} \rightarrow A\)

\[
\text{trans}(1) = \begin{align*}
\text{PRE } i \in \text{Agencies } \land \text{flight} \in \text{Flights} \text{ THEN} \\
\text{ANY } j \text{ WHERE } j \in \text{Agencies} \\
\text{THEN } \text{trans}(A) \\
\text{END} \\
\text{END}
\end{align*}
\]

\(A = A.1 \diamond A.2 \diamond A.3\)

\(\text{trans}(A) = \text{CHOICE } \text{trans}(A.1) \text{ OR } \text{trans}(A.2) \text{ OR } \text{trans}(A.3) \text{ OR } \text{END}\)

\[
[\text{trans}(A)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) = \\
[\text{trans}(A.1)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \land \\
[\text{trans}(A.2)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \land \\
[\text{trans}(A.3)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x))
\]

\(A.1 = \text{cp.}\ j\ i\ ?w \rightarrow P_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn)\)

\[
\text{trans}(A.1) = \begin{align*}
\text{ANY } w \text{ WHERE } w \in \text{MR} \text{ THEN} \\
\text{SELECT } w.\text{sold} \neq w.\text{seats} \\
\text{THEN } \text{rec} := P_2(i, S \cup \{w\}, \text{ref}^{-1}(w), pn) \\
\text{END} \\
\text{END}
\end{align*}
\]

\[
[\text{trans}(A.1)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) = \\
\forall w \bullet (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow \text{CLIP}_x(i, S \cup \{w\}, \text{ref}^{-1}(w), pn)))
\]

\(A.2 = \text{sorry.}\ j\ i \rightarrow P_3(i, S, \text{flight}, pn)\)

\[
\text{trans}(A.2) = \begin{align*}
\text{PRE } j \in \text{Agencies } \land \ i \in \text{Agencies} \text{ THEN } \text{skip END; } \text{rec} := P_3(i, S, \text{flight}, pn) \\
\text{END}
\end{align*}
\]

\[
[\text{trans}(A.2)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) = \\
j \in \text{Agencies } \land \text{CLIP}_x(i, S, \text{flight}, pn)
\]
\[ A.3 = \text{MachinesPull}(i, j) \rightarrow P_1(i, S) \]

\[ \text{trans}(A.3) = \]
\[ \text{PRE } j \in \text{Agencies} \land i \in \text{Agencies} \text{ THEN skip END; rec := } P_1(i, S) \]
\[ (\exists A.3)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2)) = j \in \text{Agencies} \land CLIP_1 \]

\[ (\exists A)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2)) = \]
\[ j \in \text{Agencies} \land CLIP_2 \land \]
\[ \forall w \cdot (w \in MR \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow CLIP_3)) \]

\[ (\exists 1)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2)) = \]
\[ i \in \text{Agencies} \land \text{flight} \in \text{Flights} \land \]
\[ \forall j \cdot (j \in \text{Agencies} \Rightarrow (\exists A)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2))) = \]
\[ i \in \text{Agencies} \land \text{flight} \in \text{Flights} \land \]
\[ \forall j \cdot (j \in \text{Agencies} \Rightarrow \]
\[ j \in \text{Agencies} \land CLIP_2 \land \]
\[ \forall w \cdot (w \in MR \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow CLIP_3)) \]

The result above shows that:
\[ (\exists 1)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2)) \Rightarrow \]
\[ (\exists 1)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLIP_2)) \]
\[ \checkmark \]

2 = require i.flight → B

\[ \text{trans}(2) = \text{PRE } i \in \text{Agencies} \land \text{flight} \in \text{Flights} \text{ THEN skip END; trans}(B) \]

\[ B = B.1 \circ B.2 \circ B.3 \]

\[ \text{trans}(B) = \text{CHOICE trans}(B.1) \text{ OR trans}(B.2) \text{ OR trans}(B.3) \text{ END} \]
B.1 = dp.i?w → P_2(i, S \cup \{w\}, ref^{-1}(w), pn)

trans(B.1) =
ANY w WHERE w : MR THEN
  SELECT w.sold ≠ w.seats
  THEN rec := P_2(i, S \cup \{w\}, ref^{-1}(w), pn)
  END
END

[trans(B.1)](\forall 1 \leq x \leq 10 \bullet (rec = P_x \Rightarrow CLIp_x)) =
\forall w \bullet (w \in MR \Rightarrow (w.sold ≠ w.seats \Rightarrow CLlp_2(i, S \cup \{w\}, ref^{-1}(w), pn)))

B.2 = notHaveIt.i → P_3(i, S, flight, pn)

trans(B.2) =
PRE i \in Agencies THEN skip END; rec := P_3(i, S, flight, pn)
[trans(B.2)](\forall 1 \leq x \leq 10 \bullet (rec = P_x \Rightarrow CLIp_x)) = CLlp_3(i, S, flight, pn)

B.3 = fullMachine.i → P_1(i, S)

trans(B.3) =
PRE i \in Agencies THEN skip END; rec := P_1(i, S)
[trans(B.3)](\forall 1 \leq x \leq 10 \bullet (rec = P_x \Rightarrow CLIp_x)) = CLlp_1(i, S)

[trans(B)](\forall 1 \leq x \leq 10 \bullet (rec = P_x \Rightarrow CLIp_x)) =
CLlp_3(i, S, flight, pn) \land
\forall w \bullet (w \in MR \Rightarrow (w.sold ≠ w.seats \Rightarrow CLlp_2(i, S \cup \{w\}, ref^{-1}(w), pn)))
\[\text{[trans(2)](} \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) = \]
\[\text{CLIP}_x(i, S, \text{flight, pn}) \land \]
\[\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow \text{CLIP}_x(i, S \cup \{w\}, \text{ref}^{-1}(w), \text{pn})))\]

The result above shows that:
\[\text{[trans(2)](} \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) = \]
\[\checkmark\]

3 = request?j?i?f → C

\[\text{[trans(3)]} = \]
\[\text{PRE } i \in \text{Agencies THEN} \]
\[\text{ANY } j, f \text{ WHERE } j \in \text{Agencies} \land f \in \text{Flights} \]
\[\text{THEN } \text{trans(C)} \]
\[\text{END}\]
\[\text{END}\]

\[\text{[trans(cp.i,j\text{ref}(f) → P}_3(i, S - \{\text{ref}(f)\}, \text{flight, pn}))]\]
\[\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x) = \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLIP}_3(i, S - \{\text{ref}(f)\}, \text{flight, pn})\]

\[\text{[trans(sorry.i.j → P}_3(i, S, \text{flight, pn}))]\]
\[\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_3) = j \in \text{Agencies} \land \text{CLIP}_3(i, S, \text{flight, pn})\]

\[\text{[trans(C)](} \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_3) = \]
\[\left( \text{ref}(f) \in S \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLIP}_3(i, S - \{\text{ref}(f)\}, \text{flight, pn}) \right) \]
\[\lor \]
\[\left( \text{ref}(f) \notin S \land j \in \text{Agencies} \land \text{CLIP}_3(i, S, \text{flight, pn}) \right)\]

\[\text{[trans(3)](} \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_3) = \]
\[i \in \text{Agencies} \land \forall j, f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow \]
\[\text{[trans(C)](} \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_3))\]
\( CL_{P_3}(i, S, \text{flight}, pn) \Rightarrow \\
\forall j, f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow CL_{P_3}(i, S - \{\text{ref}(f)\}, \text{flight}, pn)) \\
CL_{P_3}(i, S, \text{flight}, pn) \Rightarrow \\
\forall j, f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow j \in \text{Agencies} \land CL_{P_3}(i, S, \text{flight}, pn)) \\
CL_{P_3}(i, S, \text{flight}, pn) \Rightarrow \\
\forall j, f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow (\text{ref}(f) \in S \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats}) \lor \text{ref}(f) \notin S) \\
CL_{P_3}(i, S, \text{flight}, pn) \Rightarrow \\
\forall j, f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow (\text{trans}(C))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CL_{P_3}))) \\
\text{The result above shows that:} \\
(\text{rec} = P_3(i, S, \text{flight}, pn) \land CL_{P_3}(i, S, \text{flight}, pn)) \Rightarrow \\
(\text{trans}(3))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CL_{P_3}))) \\
\checkmark \\
4 = \text{ask.i}\?f \rightarrow D \\
\text{trans}(4) = \\
\text{PRE} \ i \in \text{Agencies} \ \text{THEN} \\
\quad \text{ANY} \ f \ \text{WHERE} \ f \in \text{Flights} \\
\quad \text{THEN} \ \text{trans}(D) \\
\quad \text{END} \\
\text{END} \\
\text{trans}(D))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CL_{P_3}))) = \\
\left( CL_{P_3(i, S, \text{flight}, pn, f)} \lor \right. \\
\left. (\text{ref}(f) \notin S \land CL_{P_3}(i, S, \text{flight}, pn)) \right) = CL_{P_3}(i, S, \text{flight}, pn)
\[ \text{trans}(4) \] (\( \forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CL}_{P_x}) \)) =
\[
i \in \text{Agencies} \land \forall f \cdot (f \in \text{Flights} \Rightarrow \text{CL}_{P_3(i,f,flight,pm)})
\]

The result above shows that:
\[
(\text{rec} = P_3(i,S,flight,pm) \land \text{CL}_{P_3(i,S,flight,pm)}) \Rightarrow
\[
[\text{trans}(4)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CL}_{P_x}))
\]

\[ 5 = dp.i?w \rightarrow E \]

\[ \text{trans}(5) = \]
\[
\text{ANY } w \text{ WHERE } w \in \text{MR THEN}
\]
\[
\text{SELECT } w.\text{sold} \neq w.\text{seats}
\]
\[
\text{THEN trans}(E)
\]
\[
\text{END}
\]

\[ \text{trans}(E) (\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CL}_{P_x})) = \]
\[
(\forall \quad (w = \text{ref} (\text{flight}) \land \text{CL}_{P_2(i,SU(w),flight,pm)})
\]
\[
\lor \quad (w \neq \text{ref} (\text{flight}) \land \text{CL}_{P_3(i,SU(w),flight,pm)})
\]

\[ \text{trans}(5) (\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CL}_{P_x})) = \]
\[
\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow
\]
\[
[\text{trans}(E) (\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CL}_{P_x})))
\]

\[ \text{CL}_{P_3(i,S,flight,pm)} \Rightarrow \]
\[
\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow
\]
\[
\text{CL}_{P_3(i,SU(w),flight,pm)})
\]

\[ \text{CL}_{P_3(i,S,flight,pm)} \Rightarrow \]
\[
\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow
\]
\[
w = \text{ref} (\text{flight}) \lor (w \neq \text{ref} (\text{flight}) \land \text{CL}_{P_3(i,SU(w),flight,pm)}))
\]
The result above shows that:
\[(\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLIP}_3(i, S, \text{flight}, pn)) \Rightarrow \]
\[\text{trans}(5)(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \quad \checkmark \]

The results above show that:
\[(\text{rec} = P_3(i, S, \text{flight}, pn) \land \text{CLIP}_3(i, S, \text{flight}, pn)) \Rightarrow \]
\[\text{trans}(P_3(i, S, \text{flight}, pn)))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \quad \checkmark \]

\[\checkmark \]

\[P_4(i, S, j, \text{flight})\]
\[\text{trans}(P_4(i, S, j, \text{flight})) = \]
\[\text{PRE} \; \text{ref}(\text{flight}).\text{sold} \neq \text{ref}(\text{flight}).\text{seats} \]
\[\text{THEN} \; \text{skip} \; \text{END}; \; \text{rec} := P_1(i, S) - \{\text{ref}(\text{flight})\} \]
\[\text{trans}(P_4(i, S, j, \text{flight}))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) = \]
\[\text{ref}(\text{flight}).\text{sold} \neq \text{ref}(\text{flight}).\text{seats} \land \text{CLIP}_1(i, S - \{\text{ref}(\text{flight})\}) \quad \checkmark \]

The result above shows that:
\[(\text{rec} = P_4(i, S, j, \text{flight}) \land \text{CLIP}_4(i, S, j, \text{flight})) \Rightarrow \]
\[\text{trans}(P_4(i, S, j, \text{flight}))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) \quad \checkmark \]

\[P_5(i, S, \text{flight})\]
\[\text{trans}(P_5(i, S, \text{flight})) = \]
\[\text{resp} \leftarrow \text{ref}(\text{flight}).\text{empty}; \]
\[\text{IF} \; \text{resp} = \text{YES} \]
\[\text{THEN} \; \text{trans}(\text{emptyMachine}.i \rightarrow P_1(i, S)) \]
\[\text{ELSE} \; \text{trans}(\text{emptyMachine}.i \rightarrow P_1(i, S) \rightarrow P_1(i, S - \{\text{ref}(\text{flight})\})) \]
\[\text{END} \]
\[\text{trans}(\text{emptyMachine}.i \rightarrow P_1(i, S)) = \]
\[\text{PRE} \; i \in \text{Agencies} \; \text{THEN} \; \text{skip} \; \text{END}; \; \text{rec} := P_1(i, S) \]
\[\text{trans}(\text{emptyMachine}.i \rightarrow P_1(i, S))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) = \text{CLIP}_1(i, S) \quad \checkmark \]
trans(ep.i!ref(flight) → P₁(i, S – {ref(flight)})) =

PRE ref(flight).sold ≠ 0 THEN skip END; rec := P₁(i, S – {ref(flight)})

[trans(ep.i!ref(flight) → P₁(i, S – {ref(flight)}))] :
(∀ 1 ≤ x ≤ 10 • (rec = Pₓ ⇒ CLIₓ))
= ref(flight).sold ≠ 0 ∧ CLI₁(i, S – {ref(flight)})

[trans(P₅(i, S, flight))] (∀ 1 ≤ x ≤ 10 • (rec = Pₓ ⇒ CLIₓ))
=

[IF ref(flight).sold = 0 THEN resp := YES ELSE resp := NO END]

resp = YES ∧ CLI₁(i, S)
∨
(resp ≠ YES ∧ ref(flight).sold ≠ 0 ∧ CLI₁(i, S – {ref(flight)}))

= 

(ref(flight).sold = 0 ∧ CLI₁(i, S)
∨
(ref(flight).sold ≠ 0 ∧ CLI₁(i, S – {ref(flight)}))

CLI₅(i, S, flight) ⇒ CLI₁(i, S)

CLI₅(i, S, flight) ⇒ CLI₁(i, S – {ref(flight)})

CLI₅(i, S, flight) ⇒ (ref(flight).sold = 0 ∨ ref(flight).sold ≠ 0)

CLI₅(i, S, flight) ⇒ 

(ref(flight).sold = 0 ∧ CLI₁(i, S)
∨
(ref(flight).sold ≠ 0 ∧ CLI₁(i, S – {ref(flight)}))
The result above shows that:

\((\text{rec } = P_6(i, S, \text{flight}) \land \text{CLI}_{P_6(i, S, \text{flight})}) \Rightarrow\)
\([\text{trans}(P_6(i, S, \text{flight}))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x}))\)

\(\checkmark\)

\(\star P_6(i, S, \text{flight})\)

\(P_6(i, S, \text{flight}) = 1 \Box 2 \Box 3 \Box 4\)

\(\text{trans}(P_6(i, S, \text{flight})) = \)

\(\text{CHOICE } \text{trans}(1) \text{ OR trans}(2) \text{ OR trans}(3) \text{ OR trans}(4) \text{ END}\)

\([\text{trans}(P_6(i, S, \text{flight}))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x})) =\)
\([\text{trans}(1))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x})) \land\)
\([\text{trans}(2))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x})) \land\)
\([\text{trans}(3))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x})) \land\)
\([\text{trans}(4))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x}))\)

\(1 = \text{ep.i}!\text{ref}(\text{flight}) \to P_1(i, S - \{\text{ref}(\text{flight})\})\)

\(\text{trans}(1) =\)

\(\text{PRE } \text{ref}(\text{flight}).\text{sold } \neq 0 \text{ THEN skip END; rec := } P_1(i, S - \{\text{ref}(\text{flight})\})\)

\([\text{trans}(1))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x})) =\)
\(\text{ref}(\text{flight}).\text{sold } \neq 0 \land \text{CLI}_{P_1(i, S - \{\text{ref}(\text{flight})\})}\)

The result above shows that:

\((\text{rec } = P_6(i, S, \text{flight}) \land \text{CLI}_{P_6(i, S, \text{flight})}) \Rightarrow\)
\([\text{trans}(1))(\forall 1 \leq x \leq 10 \cdot (\text{rec } = P_x \Rightarrow \text{CLI}_{P_x}))\)

\(\checkmark\)
2 = request?j/i?j → A

[trans(A)](∀1 ≤ x ≤ 10 • (rec = P_x ⇒ CLIP_x)) =

\[
\begin{align*}
    &\Big( \text{ref}(f) = \text{ref} (\text{flight}) \land j \in \text{Agencies} \land CLIP_{P_0(i,S,flight)} \\
    &\quad \lor \quad \text{ref}(f) \neq \text{ref} (\text{flight}) \land CLIP_{P_0(i,S,flight),j,f} \Big) \\
\end{align*}
\]

[trans(2)](∀1 ≤ x ≤ 10 • (rec = P_x ⇒ CLIP_x)) =

\[
\begin{align*}
    i \in \text{Agencies} \land \forall j,f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow \\
    \quad [\text{trans}(A)](∀1 ≤ x ≤ 10 • (rec = P_x ⇒ CLIP_x))
\end{align*}
\]

\[
\begin{align*}
    &\text{CLIP}_{P_0(i,S,flight)} \Rightarrow \\
    &\forall j,f \cdot (j \in \text{Agencies} \land f \in \text{Flights}) \Rightarrow (j \in \text{Agencies} \land CLIP_{P_0(i,S,flight)}) \\
\end{align*}
\]

\[
\begin{align*}
    &\text{CLIP}_{P_0(i,S,flight)} \Rightarrow \\
    &\forall j,f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow \\
    &\quad CLIP_{P_0(i,S,flight),j} \land j \in \text{Agencies} \land f \in \text{Flights}) \\
\end{align*}
\]

\[
\begin{align*}
    &\text{CLIP}_{P_0(i,S,flight)} \Rightarrow \\
    &\forall j,f \cdot (j \in \text{Agencies} \land f \in \text{Flights} \Rightarrow \\
    &\quad (\text{ref}(f) = \text{ref} (\text{flight}) \lor \text{ref}(f) \neq \text{ref} (\text{flight})) \\
\end{align*}
\]

The results above show that:

\[
\begin{align*}
    &\text{rec} = P_0(i,S,flight) \land CLIP_{P_0(i,S,flight)} \Rightarrow \\
    &[\text{trans}(2)](∀1 ≤ x ≤ 10 • (rec = P_x ⇒ CLIP_x)) \\
\end{align*}
\]
3 = ask.i?f → B

\[\text{trans}(B)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) =

\[
\left( \begin{array}{l}
\text{ref}(f) = \text{ref}(\text{flight}) \land \text{ref}(\text{flight}).\text{sold} \neq 0 \land \text{CLI}_{P_1(i,S-\{\text{ref}(\text{flight})\})} \\
\text{ref}(f) \neq \text{ref}(\text{flight}) \land \text{CLI}_{P_0(i,S,\text{flight},f)}
\end{array} \right)
\]

\[
\left( \begin{array}{l}
\text{ref}(f) = \text{ref}(\text{flight}) \land \text{ref}(\text{flight}).\text{sold} \neq 0 \land \text{CLI}_{P_1(i,S-\{\text{ref}(\text{flight})\})} \\
\text{ref}(f) \neq \text{ref}(\text{flight}) \land \text{CLI}_{P_0(i,S,\text{flight})} \land \text{ref}(f) \in \text{MR}
\end{array} \right)
\]

\[\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) =

i \in \text{Agencies} \land \forall f \cdot (f \in \text{Flights} \Rightarrow

\[\text{trans}(B)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P))
\]

\[\text{CLI}_{P_0(i,S,\text{flight})} \Rightarrow
\]

\[\forall f \cdot (f \in \text{Flights} \Rightarrow (\text{ref}(\text{flight}).\text{sold} \neq 0 \land \text{CLI}_{P_1(i,S-\{\text{ref}(\text{flight})\})}))
\]

\[\text{CLI}_{P_0(i,S,\text{flight})} \Rightarrow
\]

\[\forall f \cdot (f \in \text{Flights} \Rightarrow (\text{CLI}_{P_0(i,S,\text{flight})} \land \text{ref}(f) \in \text{MR}))
\]

\[\text{CLI}_{P_0(i,S,\text{flight})} \Rightarrow
\]

\[\forall f \cdot (f \in \text{Flights} \Rightarrow (\text{ref}(f) = \text{ref}(\text{flight}) \lor \text{ref}(f) \neq \text{ref}(\text{flight})))
\]

The results above show that:

\[\langle \text{rec} = P_0(i,S,\text{flight}) \land \text{CLI}_{P_0(i,S,\text{flight})} \rangle \Rightarrow
\]

\[\text{trans}(3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P))\]

\[\check\]

4 = \text{dp.i?} w \rightarrow P_0(i,S \cup \{w\},\text{flight})

\[\text{trans}(4)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_P)) =

\[\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq w.\text{seats} \Rightarrow \text{CLI}_{P_0(i,S \cup \{w\},\text{flight})}))
\]
The result above shows that:
\[
\text{rec} = P_6(i, S, \text{flight}) \land CLI_{P_6}(i, S, \text{flight}) \Rightarrow \text{trans}(i) \forall x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLI_{P_x})
\]

The results above show that:
\[
\text{rec} = P_8(i, S, \text{flight}) \land CLI_{P_8}(i, S, \text{flight}) \Rightarrow \text{trans}(P_8(i, S, \text{flight})) \forall x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLI_{P_x})
\]

\[P_7(i, S, \text{flight}, j, f)\]
\[
\text{trans}(P_7(i, S, \text{flight}, j, f)) =
\]
IF \(\text{ref}(f) \in S\) THEN \(\text{trans}(\text{cp}.i.j!\text{ref}(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight}))\)
ELSE \(\text{trans}(\text{sorry}.i.j \rightarrow P_6(i, S, \text{flight}))\)

\[\text{trans}(\text{cp}.i.j!\text{ref}(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight}))\]
\(\forall x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLI_{P_x})\)

\[\text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land CLI_{P_6}(i, S - \{\text{ref}(f)\}, \text{flight})\]

\[\text{trans}(\text{sorry}.i.j \rightarrow P_6(i, S, \text{flight}))\]
\(\forall x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLI_{P_x})\)

\[\text{trans}(P_7(i, S, \text{flight}, j, f))\]
\(\forall x \leq 10 \cdot \text{rec} = P_x \Rightarrow CLI_{P_x})\)

\(\text{ref}(f) \in S \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land CLI_{P_6}(i, S - \{\text{ref}(f)\}, \text{flight})\)
\(
\lor
\)
\(\text{ref}(f) \notin S \land j \in \text{Agencies} \land CLI_{P_6}(i, S, \text{flight})\)

\(CLI_{P_7}(i, S, \text{flight}, j, f) \Rightarrow CLI_{P_6}(i, S - \{\text{ref}(f)\}, \text{flight})\)

\(CLI_{P_7}(i, S, \text{flight}, j, f) \Rightarrow j \in \text{Agencies} \land CLI_{P_6}(i, S, \text{flight})\)
\[
CLI_{P_t(i, S, \text{flight}, j, f)} \Rightarrow \left( \begin{array}{l}
\text{ref}(f) \in S \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \\
\lor \\
\text{ref}(f) \notin S
\end{array} \right)
\]

The results above show that:

\[
(\text{rec} = P_t(i, S, \text{flight}, j, f) \land CLI_{P_t(i, S, \text{flight}, j, f)}) \Rightarrow \\
[\text{trans}(P_t(i, S, \text{flight}, j, f))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLI_{P_x}))
\]

\[
\checkmark
\]

\[
P_8(i, S, \text{flight}, f)
\]

\[
\text{trans}(P_8(i, S, \text{flight}, f)) = \\
\text{IF } \text{ref}(f) \in S \\
\text{THEN } \text{rec} := P_8(i, S, \text{flight}, f) \\
\text{ELSE } \text{trans}(\text{notHaveIt}.i \rightarrow P_8(i, S, \text{flight})) \\
\text{END}
\]

\[
[\text{trans}(\text{notHaveIt}.i \rightarrow P_8(i, S, \text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLI_{P_x}))
\]

\[
= CLI_{P_8(i, S, \text{flight})}
\]

\[
[\text{trans}(P_8(i, S, \text{flight}, f))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow CLI_{P_x})) = \\
\left( \begin{array}{l}
\text{ref}(f) \in S \land CLI_{P_8(i, S, \text{flight}, f)} \\
\lor \\
\text{ref}(f) \notin S \land CLI_{P_8(i, S, \text{flight})}
\end{array} \right)
\]

\[
\left( \begin{array}{l}
\text{ref}(f) \in S \land CLI_{P_8(i, S, \text{flight})} \land \text{ref(\text{flight})} \neq \text{ref}(f) \\
\lor \\
\text{ref}(f) \notin S \land CLI_{P_8(i, S, \text{flight})}
\end{array} \right)
\]

\[
CLI_{P_8(i, S, \text{flight}, f)} \Rightarrow CLI_{P_8(i, S, \text{flight})}
\]

\[
CLI_{P_8(i, S, \text{flight}, f)} \Rightarrow \text{ref(\text{flight})} \neq \text{ref}(f)
\]
\[ CLI_{P_6(i, S, \text{flight}, f)} \Rightarrow (\text{ref}(f) \in S \lor \text{ref}(f) \notin S) \]

The results above show that:
\[
(rec = P_6(i, S, \text{flight}, f) \land CLI_{P_6(i, S, \text{flight}, f)}) \Rightarrow \\
[\text{trans}(P_6(i, S, \text{flight}, f))] (\forall 1 \leq x \leq 10 \cdot (rec = P_x \Rightarrow CLI_{P_x}))
\]
\[
\checkmark
\]

\[ P_6(i, S, \text{flight}, f) \]
\[ \text{trans}(P_6(i, S, \text{flight}, f)) = \\
\text{resp} \leftarrow \text{ref}(f).\text{empty}; \\
\text{IF resp = YES} \\
\text{THEN trans}(\text{emptyMachine}.i \rightarrow P_6(i, S, \text{flight})) \\
\text{ELSE trans}(\text{ep}.i!\text{ref}(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight})) \]
\[ \text{END} \]
\[
[\text{trans}(\text{emptyMachine}.i \rightarrow P_6(i, S, \text{flight}))] \\
(\forall 1 \leq x \leq 10 \cdot (rec = P_x \Rightarrow CLI_{P_x}))
= CLI_{P_6(i, S, \text{flight})}
\]
\[
[\text{trans}(\text{ep}.i!\text{ref}(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight}))] \\
(\forall 1 \leq x \leq 10 \cdot (rec = P_x \Rightarrow CLI_{P_x}))
= \text{ref}(f).\text{sold} \neq 0 \land CLI_{P_6(i, S - \{\text{ref}(f)\}, \text{flight})}
\]
\[
\begin{cases}
\text{IF resp = YES} \\
\text{THEN trans}(\text{emptyMachine}.i \rightarrow P_6(i, S, \text{flight})) \\
\text{ELSE trans}(\text{ep}.i!\text{ref}(f) \rightarrow P_6(i, S - \{\text{ref}(f)\}, \text{flight})) \\
\text{END} \\
(\forall 1 \leq x \leq 10 \cdot (rec = P_x \Rightarrow CLI_{P_x}))
\end{cases}
= \\
\left( \text{resp = YES} \land CLI_{P_6(i, S, \text{flight})} \lor \right.
\left( \text{resp \neq YES} \land \text{ref}(f).\text{sold} \neq 0 \land CLI_{P_6(i, S - \{\text{ref}(f)\}, \text{flight}} \right)
\[ \text{trans}(P_9(i, S, \text{flight}, f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x)) =
\]
\[
\begin{aligned}
&\text{IF } \text{ref}(f).\text{sold} = 0 \\
&\text{THEN } \text{resp} := \text{YES} \\
&\text{ELSE } \text{resp} := \text{NO} \\
&\text{END}
\end{aligned}
\]
\[
\left( \text{resp} = \text{YES} \land \text{CLIP}_6(i, S, \text{flight}) \right) \\
\lor \\
\left( \text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq 0 \land \text{CLIP}_6(i, S - \{\text{ref}(f)\}, \text{flight}) \right)
\]
\[
\left( \text{ref}(f).\text{sold} = 0 \land \text{CLIP}_6(i, S, \text{flight}) \right) \\
\lor \\
\left( \text{ref}(f).\text{sold} \neq 0 \land \text{CLIP}_6(i, S - \{\text{ref}(f)\}, \text{flight}) \right)
\]

\(\text{CLIP}_6(i, S, \text{flight}, f) \Rightarrow \text{CLIP}_6(i, S, \text{flight})\)

\(\text{CLIP}_6(i, S, \text{flight}, f) \Rightarrow \text{CLIP}_6(i, S - \{\text{ref}(f)\}, \text{flight})\)

\(\text{CLIP}_6(i, S, \text{flight}, f) \Rightarrow (\text{ref}(f).\text{sold} = 0 \lor \text{ref}(f).\text{sold} \neq 0)\)

The results above show that:
\((\text{rec} = P_9(i, S, \text{flight}, f) \land \text{CLIP}_6(i, S, \text{flight}, f)) \Rightarrow \)
\[\text{trans}(P_9(i, S, \text{flight}, f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow \text{CLIP}_x))
\]

\[\checkmark\]

\[P_{10}(i, S, \text{flight}, pn, f)\]
\[
\text{trans}(P_{10}(i, S, \text{flight}, pn, f)) =
\text{resp} \leftarrow \text{ref}(f).\text{empty};
\text{IF } \text{resp} = \text{YES}
\text{THEN } \text{trans}(\text{emptyMachine}.i \rightarrow P_3(i, S, \text{flight}, pn))
\text{ELSE } \text{trans}(\text{ep}.i!\text{ref}(f) \rightarrow P_3(i, S - \{\text{ref}(f)\}, \text{flight}, pn))
\text{END} \]
APPENDIX C. WP PROOFS OF FLIGHT TICKETS SALE SYSTEM

\[\text{trans}(\text{emptyMachine}.i \rightarrow P_3(i, S, \text{flight}, pn))\]

\((\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x}))\)

\[\text{CLI}_{P_x}(i, S, \text{flight}, pn)\]

\[\text{trans}(\text{ep.}i!\text{ref}(f) \rightarrow P_3(i, S - \{\text{ref}(f)\}, \text{flight}, pn))\]

\((\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x}))\)

\[\text{ref}(f).\text{sold} \neq 0 \land \text{CLI}_{P_x}(i, S - \{\text{ref}(f)\}, \text{flight}, pn)\]

\[IF \ \text{resp} = \text{YES} \]

THEN \ \text{trans}(\text{emptyMachine}.i \rightarrow P_3(i, S, \text{flight}, pn))

ELSE \ \text{trans}(\text{ep.}i!\text{ref}(f) \rightarrow P_3(i, S - \{\text{ref}(f)\}, \text{flight}, pn))

END

\((\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x}))\)

\[
\left(\text{resp} = \text{YES} \land \text{CLI}_{P_3}(i, S, \text{flight}, pn)\right) \\
\left(\text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq 0 \land \text{CLI}_{P_3}(i, S - \{\text{ref}(f)\}, \text{flight}, pn)\right)
\]

\[\text{trans}(P_{10}(i, S, \text{flight}, pn, f))(\forall 1 \leq x \leq 10 \cdot (\text{rec} = P_x \Rightarrow \text{CLI}_{P_x}))\]

\[
\left(\text{ref}(f).\text{sold} = 0\right) \\
\left(\text{THEN \ \text{resp} := YES}\right) \\
\left(\text{ELSE \ \text{resp} := NO}\right) \\
\text{END}
\]

\[
\left(\text{resp} = \text{YES} \land \text{CLI}_{P_3}(i, S, \text{flight}, pn)\right) \\
\left(\text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq 0 \land \text{CLI}_{P_3}(i, S - \{\text{ref}(f)\}, \text{flight}, pn)\right)
\]

\[
\left(\text{ref}(f).\text{sold} = 0 \land \text{CLI}_{P_3}(i, S, \text{flight}, pn)\right) \\
\left(\text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq 0 \land \text{CLI}_{P_3}(i, S - \{\text{ref}(f)\}, \text{flight}, pn)\right)
\]
$CL_{P_{10}}(i, S, \text{flight}, pn, f) \Rightarrow CL_{P_9}(i, S, \text{flight}, pn)$

$CL_{P_{10}}(i, S, \text{flight}, pn, f) \Rightarrow CL_{P_9}(i, S, \{\text{ref}(f)\}, \text{flight}, pn)$

$CL_{P_{10}}(i, S, \text{flight}, pn, f) \Rightarrow (\text{ref}(f).\text{sold} = 0 \lor \text{ref}(f).\text{sold} \neq 0)$

The results above show that:

$(\text{rec} = P_{10}(i, S, \text{flight}, pn, f) \land CL_{P_{10}}(i, S, \text{flight}, pn, f)) \Rightarrow$

$[\text{trans}(P_{10}(i, S, \text{flight}, pn, f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = P_x \Rightarrow CL_{P_x}))$
\* \( R_1(S) \)

\( R_1(S) = 1 \land 2 \land 3 \)

\[ \text{trans}(R_1(S)) = \text{CHOICE trans(1) OR trans(2) OR trans(3) END} \]

\[ \begin{align*}
\text{trans}(R_1(S)) & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) ) \land \\
\text{trans}(1) & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) ) \\
\text{trans}(2) & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) ) \\
\text{trans}(3) & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) )
\end{align*} \]

\( 1 = \text{customerToCancel?flight?pn} \rightarrow \begin{cases} 
\text{if } \text{ref(flight)} \in S \\
\quad \text{then } R_2(S, \text{flight}, \text{pn}) \\
\quad \text{else } R_3(S, \text{flight}, \text{pn}) 
\end{cases} \)

\[ \begin{align*}
\text{trans(1)} & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) ) \\
\text{trans(2)} & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) ) \\
\text{trans(3)} & \equiv (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R) )
\end{align*} \]

\( 2 = \text{require?j?flight} \rightarrow \begin{cases} 
\text{if } \text{ref(flight)} \in S \\
\quad \text{then } R_5(S, \text{flight}, \text{f}) \\
\quad \text{else } (\text{notHaveIt.j} \rightarrow R_1(S))
\end{cases} \)

The result above shows that:
\[ (\text{rec} = R_1(S) \land \text{CLI}_{R_1(S)}) \Rightarrow [\text{trans(1)}(\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_R))] \]
\[
\begin{align*}
\text{IF } & \text{ref(flight)} \in S \\
\text{THEN } & \text{rec := R}_5(S, \text{flight}, j) \\
\text{ELSE } & \text{trans(notHaveIt, } j \rightarrow R_1(S))
\end{align*}
\]

\[
(\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4})) =
\]

\[
\left(\begin{array}{c}
\text{ref(flight)} \in S \land \text{CLI}_{R_4}(S, \text{flight}, j) \\
\text{OR } \text{ref(flight)} \notin S \land j \in \text{Agencies} \land \text{CLI}_{R_4}(S)
\end{array}\right) = \text{CLI}_{R_4}(S) \land j \in \text{Agencies}
\]

\[
[\text{trans}(2)](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4})) =
\]

\[
\forall j, \text{flight} \bullet ((j \in \text{Agencies} \land \text{flight} \in \text{Flights}) \Rightarrow (\text{CLI}_{R_4}(S) \land j \in \text{Agencies}))
\]

The result above shows that:

\[
(\text{rec = } R_4(S) \land \text{CLI}_{R_4}(S)) \Rightarrow [\text{trans}(2)](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4}))
\]

\[
3 = \Box_{j \in \text{Agencies}} ep.1?z \rightarrow R_1(S \cup \{z\})
\]

\[
\text{trans}(3) = \text{CHOICE } \text{trans}(ep.1?z \rightarrow R_1(S \cup \{z\})) \text{ OR } \text{trans}(ep.2?z \rightarrow R_1(S \cup \{z\}))
\]

\[
[\text{trans}(ep.1?z \rightarrow R_1(S \cup \{z\}))](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4})) =
\]

\[
\forall z \bullet (z \in \text{MR} \Rightarrow (z.\text{sold} \neq 0 \Rightarrow \text{CLI}_{R_4}(\text{su}\{z\}))
\]

\[
[\text{trans}(ep.2?z \rightarrow R_1(S \cup \{z\}))](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4})) =
\]

\[
\forall z \bullet (z \in \text{MR} \Rightarrow (z.\text{sold} \neq 0 \Rightarrow \text{CLI}_{R_4}(\text{su}\{z\}))
\]

\[
[\text{trans}(3)](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4})) =
\]

\[
\forall z \bullet (z \in \text{MR} \Rightarrow (z.\text{sold} \neq 0 \Rightarrow \text{CLI}_{R_4}(\text{su}\{z\}))
\]

The result above shows that:

\[
(\text{rec = } R_4(S) \land \text{CLI}_{R_4}(S)) \Rightarrow [\text{trans}(3)](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4}))
\]

\[
\checkmark
\]

The results above show that:

\[
(\text{rec = } R_4(S) \land \text{CLI}_{R_4}(S)) \Rightarrow [\text{trans}(R_4(S))](\forall 1 \leq x \leq 10 \bullet (\text{rec = } R_x \Rightarrow \text{CLI}_{R_4}))
\]

\[
\checkmark
\]
$\star \ R_2(S, \text{flight}, pn)$

\[
\text{trans}(R_2(S, \text{flight}, pn)) = \text{resp} \leftarrow \text{ref}\,(\text{flight}).\text{cancel}(pn);
\]

\[
\begin{align*}
\text{IF} \ \text{resp} = \text{Empty} \\
\text{THEN} \ \text{rec} := R_4(S, \text{flight}) \\
\text{ELSE} \ \text{rec} := R_1(S) \ \text{END}
\end{align*}
\]

\[
[\text{IF} \ \text{resp} = \text{Empty} \\
\text{THEN} \ \text{rec} := R_4(S, \text{flight}) \\
\text{ELSE} \ \text{rec} := R_1(S) \ \text{END}]
\]

\[
(\text{resp} = \text{Empty} \land \text{CLI}_{R_4}(S, \text{flight})) \\
\lor \\
(\text{resp} \neq \text{Empty} \land \text{CLI}_{R_4}(S))
\]

\[
[\text{trans}(R_2(S, \text{flight}, pn))][(\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x}))] =
\]

\[
(\text{resp} = \text{Empty} \land \text{CLI}_{R_4}(S, \text{flight})) \\
\lor \\
(\text{resp} \neq \text{Empty} \land \text{CLI}_{R_4}(S))
\]

\[
\begin{align*}
\text{pn} \in \text{Passport} & \land \text{ref}\,(\text{flight}).\text{sold} \neq 0 \\
\text{IF} \ \text{pn} \in \text{ref}\,(\text{flight}).\text{customer} \ \text{THEN} \\
\text{ref}\,(\text{flight}).\text{customer} & := \text{ref}\,(\text{flight}).\text{customer} - \{\text{pn}\} \\
\text{ref}\,(\text{flight}).\text{sold} & := \text{ref}\,(\text{flight}).\text{sold} - 1 \\
\text{IF} \ \text{ref}\,(\text{flight}).\text{sold} - 1 = 0 \\
\text{THEN} \ \text{resp} := \text{Empty} \\
\text{ELSE} \ \text{resp} := \text{Available} \\
\text{END} \\
\text{ELSE} \ \text{resp} := \text{IncorrectInput} \\
\text{END}
\end{align*}
\]

\[
(\text{resp} = \text{Empty} \land \text{CLI}_{R_4}(S, \text{flight})) \\
\lor \\
(\text{resp} \neq \text{Empty} \land \text{CLI}_{R_4}(S))
\]
\[ A = \]

\[
\begin{align*}
  \text{ref(flight).customer} & := \text{ref(flight).customer} - \{ \text{pn} \} \quad || \\
  \text{ref(flight).sold} & := \text{ref(flight).sold} - 1 \quad || \\
  \text{IF } \text{ref(flight).sold} - 1 & = 0 \\
  \text{THEN } \text{response} & := \text{Empty} \\
  \text{ELSE } \text{response} & := \text{Available} \\
  \text{END} \\
\end{align*}
\]

\[
\left( \text{resp} = \text{Empty} \land \text{CLI}_{R_4}(S, S_{flight}) \right) \\
\lor \\
\left( \text{resp} \neq \text{Empty} \land \text{CLI}_{R_4}(S) \right)
\]

\[
\left( \text{ref(flight).sold} - 1 = 0 \land S \subseteq MR \land \text{flight} \in \text{Flights} \land \\
\forall k \in (S - \{ \text{ref(flight)} \}) \land k.\text{sold} \neq 0 \land \text{ref(flight)} \in S \\
\lor \\
\text{ref(flight).sold} - 1 \neq 0 \land S \subseteq MR \land \forall k \in S \land k.\text{sold} \neq 0 \right)
\]

\[
\left[ \text{IF } \text{pn} \in \text{ref(flight).customer} \text{ THEN} \\
\text{ref(flight).customer} & := \text{ref(flight).customer} - \{ \text{pn} \} \quad || \\
\text{ref(flight).sold} & := \text{ref(flight).sold} - 1 \quad || \\
\text{IF } \text{ref(flight).sold} - 1 & = 0 \\
\text{THEN } \text{resp} & := \text{Empty} \\
\text{ELSE } \text{resp} & := \text{Available} \\
\text{END} \\
\right] \\
\text{ELSE } \text{resp} := \text{IncorrectInput} \\
\text{END} \\
\]

\[
\left( \text{resp} = \text{Empty} \land \text{CLI}_{R_4}(S_{flight}) \right) \\
\lor \\
\left( \text{resp} \neq \text{Empty} \land \text{CLI}_{R_4}(S) \right)
\]

\[
\left( \text{pn} \in \text{ref(flight).customer} \land A \right) \\
\lor \\
\left( \text{pn} \notin \text{ref(flight).customer} \land \text{CLI}_{R_4}(S) \right)
\]
APPENDIX C. WP PROOFS OF FLIGHT TICKETS SALE SYSTEM

\[ CLIR_2(S, flight, pn) \Rightarrow (S \subseteq MR \wedge \text{flight} \in \text{Flights} \wedge \forall k \in (S - \{\text{ref(\text{flight})}\}) \bullet k.\text{sold} \neq 0 \wedge \text{ref(\text{flight})} \in \mathcal{S}) \]

\[ CLIR_2(S, flight, pn) \Rightarrow (S \subseteq MR \wedge \forall k \in S \bullet k.\text{sold} \neq 0) \]

\[ CLIR_2(S, flight, pn) \Rightarrow (\text{ref(\text{flight}).sold - 1 = 0} \vee \text{ref(\text{flight}).sold - 1} \neq 0) \]

\[ CLIR_2(S, flight, pn) \Rightarrow A \]

\[ CLIR_2(S, flight, pn) \Rightarrow CLIR_1(S) \]

\[ CLIR_2(S, flight, pn) \Rightarrow (pn \in \text{ref(\text{flight}).customer} \vee pn \notin \text{ref(\text{flight}).customer}) \]

The results above show that:

\[ CLIR_2(S, flight, pn) \Rightarrow \begin{cases} pn \in \text{ref(\text{flight}).customer} \wedge A \\ pn \notin \text{ref(\text{flight}).customer} \wedge CLIR_1(S) \end{cases} \]

\[ CLIR_2(S, flight, pn) \Rightarrow (pn \in \text{Passport} \wedge \text{ref(\text{flight}).sold} \neq 0) \]

The results above show that:

\[ (\text{rec} = R_2(S, flight, pn) \wedge CLIR_2(S, flight, pn)) \Rightarrow \]

\[ [\text{trans}(R_2(S, flight, pn))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLIR_x)) \]

✓

\[ R_3(S, flight, pn) \]

\[ R_3(S, flight, pn) = 1 \blacklozenge 2 \blacklozenge 3 \]

\[ \text{trans}(R_3(S, flight, pn)) = \text{CHOICE} \text{trans(1) OR trans(2) OR trans(3) END} \]

\[ [\text{trans}(R_3(S, flight, pn))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLIR_x)) = \]

\[ [\text{trans}(1)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLIR_x)) \wedge \]

\[ [\text{trans}(2)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLIR_x)) \wedge \]

\[ [\text{trans}(3)](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLIR_x)) \]
\[ 1 = \text{ask?flight} \rightarrow A \]

\[ A = A.1 \sqcup A.2 \sqcup A.3 \]

\[ [\text{trans}(A)] = \text{CHOICE} \text{trans}(A.1) \text{OR trans}(A.2) \text{OR trans}(A.3) \text{END} \]

\[ [\text{trans}(A)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_w \Rightarrow \text{CLIR}_R)) = \]
\[ [\text{trans}(A.1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) \land \]
\[ [\text{trans}(A.2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) \land \]
\[ [\text{trans}(A.3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) \]

\[ A.1 = \text{ep?w} \rightarrow R_2(S \cup \{w\}, \text{ref}^{-1}(w), \text{pn}) \]

\[ [\text{trans}(A.1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) = \]
\[ \forall w \cdot (w \in M \Rightarrow (w \cdot \text{sold} \neq 0 \Rightarrow \text{CLIR}_R(S \cup \{w\}, \text{ref}^{-1}(w), \text{pn}))) \]

\[ A.2 = \text{notHaveIt?j} \rightarrow R_3(S, \text{flight}, \text{pn}) \]

\[ [\text{trans}(A.2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) = \]
\[ j \in \text{Agencies} \land \text{CLIR}_R(S, \text{flight}, \text{pn}) \]

\[ A.3 = \text{emptyMachine?j} \rightarrow R_1(S) \]

\[ [\text{trans}(A.3)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) = \]
\[ j \in \text{Agencies} \land \text{CLIR}_R(S) \land \]
\[ [\text{trans}(A)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_R)) = \]
\[ j \in \text{Agencies} \land \text{CLIR}_R(S, \text{flight}, \text{pn}) \land \]
\[ \forall w \cdot ((w \in M \Rightarrow (w \cdot \text{sold} \neq 0 \Rightarrow \text{CLIR}_R(S \cup \{w\}, \text{ref}^{-1}(w), \text{pn}))) \]

\[ \text{trans}(1) = \text{PRE flight} \in \text{Flights} \]
\[ \quad \text{THEN ANY } j \text{ WHERE } j \in \text{Agencies} \]
\[ \quad \text{THEN } \text{trans}(A) \]
\[ \quad \text{END} \]
\[ \text{END} \]
[\text{trans}(1)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x})) = \\
flight \in \text{Flights} \quad \land \\
\forall j \cdot (j \in \text{Agencies} \Rightarrow [\text{trans}(A)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x})))

The result above shows that:

(\text{rec} = R_3(S, \text{flight}, \text{pn}) \land \text{CLI}_{R_3(S, \text{flight}, \text{pn}))} \Rightarrow \\
[\text{trans}(1)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x}))

\checkmark

2 = \text{require} ?? f \rightarrow \text{if} \; \text{ref}(f) \in S \\
\text{then} \; R_{10}(S, \text{flight}, \text{pn}, j, f) \\
\text{else} \; (\text{notHaveIt}, j \rightarrow R_3(S, \text{flight}, \text{pn}))

\begin{align*}
\begin{array}{c}
\text{IF} \; \text{ref}(f) \in S \; \text{THEN} \\
\text{rec} := R_{10}(S, \text{flight}, \text{pn}, j, f) \; \text{ELSE} \\
\text{trans} (\text{notHaveIt}, j \rightarrow R_3(S, \text{flight}, \text{pn}))
\end{array} \\
\text{END}
\end{align*}

= (\text{ref}(f) \in S \land \text{CLI}_{R_{10}(S, \text{flight}, \text{pn}, j, f)} \\
\lor \\
\text{ref}(f) \notin S \land j \in \text{Agencies} \land \text{CLI}_{R_3(S, \text{flight}, \text{pn})})

= j \in \text{Agencies} \land \text{CLI}_{R_3(S, \text{flight}, \text{pn})}

[\text{trans}(2)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x})) = \\
\forall j, f \cdot ((j \in \text{Agencies} \land f \in \text{Flights}) \Rightarrow (j \in \text{Agencies} \land \text{CLI}_{R_3(S, \text{flight}, \text{pn})}))

The result above shows that:

(\text{rec} = R_3(S, \text{flight}, \text{pn}) \land \text{CLI}_{R_3(S, \text{flight}, \text{pn}))} \Rightarrow \\
[\text{trans}(2)] (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_x}))

\checkmark
\[ 3 = \square_{j \in \text{Agencies}} \text{ep}\_j?w \rightarrow \text{if } w = \text{ref}(\text{flight}) \]

\[ \text{then } R_0(S \cup \{w\}, \text{ref}^{-1}(w), pn) \]

\[ \text{else } R_3(S \cup \{w\}, \text{flight}, pn) \]

\[
\begin{align*}
\text{IF } w = \text{ref}(\text{flight}) \text{ THEN} \\
\text{rec} := R_0(S \cup \{w\}, \text{ref}^{-1}(w), pn) \\
\text{ELSE rec} := R_3(S \cup \{w\}, \text{flight}, pn) \\
\text{END}
\end{align*}
\]

\[
\begin{align*}
  (\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x))
\end{align*}
\]

\[
\begin{align*}
  = \left( w = \text{ref}(\text{flight}) \land \text{CLIR}_x(S \cup \{w\}, \text{ref}^{-1}(w), pn) \right)
\quad \lor \\
  (w \neq \text{ref}(\text{flight}) \land \text{CLIR}_x(S \cup \{w\}, \text{flight}, pn))
\end{align*}
\]

\[
\begin{align*}
\text{if trans}(3)\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right] \Rightarrow \\
\forall w \cdot (w \in \text{MR} \Rightarrow (w.\text{sold} \neq 0 \Rightarrow D))
\end{align*}
\]

The result above shows that:

\[
(\text{rec} = R_3(S, \text{flight}, pn) \land \text{CLIR}_R(S, \text{flight}, pn)) \Rightarrow \\
\text{trans}(3)\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right]
\]

\checkmark

The results above show that:

\[
(\text{rec} = R_3(S, \text{flight}, pn) \land \text{CLIR}_R(S, \text{flight}, pn)) \Rightarrow \\
\text{trans}(R_3(S, \text{flight}, pn))\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right]
\]

\checkmark

\[ R_4(S, \text{flight}) \]

\[ R_4(S, \text{flight}) = 1 \square 2 \square 3 \]

\[ \text{trans}(R_4(S, \text{flight})) = \text{CHOICE} \text{ trans}(1) \text{ OR trans}(2) \text{ OR trans}(3) \text{ END} \]

\[
\begin{align*}
\text{trans}(R_4(S, \text{flight})))\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right] = \\
\text{trans}(1))\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right] \land \\
\text{trans}(2))\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right] \land \\
\text{trans}(3))\left[ \forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLIR}_x) \right]
\end{align*}
\]
1 = \square_{j \in \text{Agencies}} \text{dp}.j!\text{ref(flight)} \rightarrow R_1(S - \{\text{ref(flight)}\})

\text{trans}(1) = \text{CHOICE} \text{trans}(\text{dp}.1!\text{ref(flight)} \rightarrow R_1(S - \{\text{ref(flight)}\}))
\text{trans}(\text{dp}.2!\text{ref(flight)} \rightarrow R_1(S - \{\text{ref(flight)}\}))
\text{END}

[\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow CLIR_x)) =
\text{ref(flight).sold} \neq \text{ref(flight).seats} \land CLIR_1(S - \{\text{ref(flight)}\})

The result above shows that:
\(\text{rec} = R_4(S, \text{flight}) \land CLIR_4(S, \text{flight}) \Rightarrow\)
[\text{trans}(1)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow CLIR_x))
\checkmark

2 = \text{require}?j!?f \rightarrow \text{if} \text{ ref}(f) = \text{ref(flight)}
\quad \text{then} \text{ dp}.j!\text{ref(flight)} \rightarrow R_1(S - \{\text{ref(flight)}\})
\quad \text{else} \text{ R}_5(S, \text{flight}, j, f)

\left[
\begin{array}{l}
\text{IF} \text{ ref}(f) = \text{ref(flight)} \text{ THEN} \\
\text{dp}.j!\text{ref(flight)} \rightarrow R_1(S - \{\text{ref(flight)}\}) \\
\text{ELSE} \text{ R}_5(S, \text{flight}, j, f)
\end{array}\right]

(\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow CLIR_x))
= (\text{ref}(f) = \text{ref(flight)} \land \text{ref(flight).sold} \neq \text{ref(flight).seats} \land
\text{CLI}_R_1(S - \{\text{ref(flight)}\}))
\lor
(\text{ref}(f) \neq \text{ref(flight)} \land \text{CLI}_R_4(S, \text{flight}) \land j \in \text{Agencies} \land \text{ref}(f) \in \text{MR})
= E

[\text{trans}(2)](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow CLIR_x)) =
\forall j, f \cdot ((j \in \text{Agencies} \land f \in \text{Flights}) \Rightarrow E)
\[ CLI_{R_4}(S, \text{flight}) \Rightarrow \]
\[ \forall j, f \bullet (j \in \text{Agencies} \land f \in \text{Flights}) \Rightarrow \]
\[ \left( \text{ref(flight).sold} \neq \text{ref(flight).seats} \land CLI_{R_4}(S - \{\text{ref(flight)}\}) \right) \]

\[ CLI_{R_4}(S, \text{flight}) \Rightarrow \]
\[ \forall j, f \bullet (j \in \text{Agencies} \land f \in \text{Flights}) \Rightarrow \]
\[ \left( CLI_{R_4}(S, \text{flight}) \land j \in \text{Agencies} \land \text{ref(f)} \in \text{MR} \right) \]

The results above show that:
\[(\text{rec} = R_4(S, \text{flight}) \land CLI_{R_4}(S, \text{flight})) \Rightarrow \]
\[ [\text{trans(2)}] (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x})) \]
\[ \checkmark \]

\[ 3 = \square_{j \in \text{Agencies}} \text{ep}. j?w \rightarrow R_4(S \cup \{w\}, \text{flight}) \]

\[ \text{trans(3)} = \text{CHOICE trans(ep}.1?w \rightarrow R_4(S \cup \{w\}, \text{flight}) \]
\[ \text{OR trans(ep}.2?w \rightarrow R_4(S \cup \{w\}, \text{flight}) \]
\[ \text{END} \]

\[ [\text{trans(3)}] (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x})) = \]
\[ \forall w \bullet (w \in \text{MR} \Rightarrow (w.\text{sold} \neq 0 \Rightarrow CLI_{R_4}(S \cup \{w\}, \text{flight}))) \]

The result above shows that:
\[(\text{rec} = R_4(S, \text{flight}) \land CLI_{R_4}(S, \text{flight})) \Rightarrow \]
\[ [\text{trans(3)}] (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x})) \]
\[ \checkmark \]

The results above show that:
\[(\text{rec} = R_4(S, \text{flight}) \land CLI_{R_4}(S, \text{flight})) \Rightarrow \]
\[ [\text{trans}(R_4(S, \text{flight}))] (\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x})) \]
\[ \checkmark \]
\[ R_6(S, \text{flight}, j) \]
\[
\text{trans}(R_6(S, \text{flight}, j)) = \text{resp} \leftarrow \text{ref(\text{flight}).full};
\]
\[
\text{IF resp = YES}
\]
\[
\text{THEN rec := } R_6(S, j)
\]
\[
\text{ELSE rec := } R_7(S, \text{flight}, j)
\]
\[
\text{END}
\]

\[
[\text{trans}(R_6(S, \text{flight}, j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CI}_R_a)) =
\]
\[
\begin{cases}
\text{IF } \text{ref(\text{flight}).sold = ref(\text{flight}).seats} \\
\text{THEN resp := YES}
\end{cases}
\begin{cases}
\text{resp = YES} \\
\text{resp \neq YES}
\end{cases}
\]
\[
\begin{cases}
\text{resp = YES} \\
\text{resp \neq YES}
\end{cases}
\text{\land} \text{\ CI}_R_a(S, j)
\]
\[
\text{\lor} \text{\ CI}_R_a(S, \text{flight}, j)
\]

\[ \text{CI}_R_a(S, \text{flight}, j) \Rightarrow \text{CI}_R_a(S, j) \]

\[ \text{CI}_R_a(S, \text{flight}, j) \Rightarrow \begin{cases}
\text{ref(\text{flight}).sold = ref(\text{flight}).seats} \\
\text{ref(\text{flight}).sold \neq ref(\text{flight}).seats}
\end{cases} \]

The results above show that:
\[
(\text{rec} = R_6(S, \text{flight}, j) \land \text{CI}_R_a(S, \text{flight}, j)) \Rightarrow
\]
\[
[\text{trans}(R_6(S, \text{flight}, j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CI}_R_a))
\]

\[ R_6(S, j) \]

\[
[\text{trans}(R_6(S, j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CI}_R_a)) =
\]
\[
\text{j \in Agencies} \land \text{CI}_R_a(S)
\]

The result above shows that:
\[
(\text{rec} = R_6(S, j) \land \text{CI}_R_a(S, j)) \Rightarrow
\]
\[
[\text{trans}(R_6(S, j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CI}_R_a))
\]

\[ \checkmark \]
\( R_7(S,\text{flight},j) \)

\[
[\text{trans}(R_7(S,\text{flight},j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_7})) =
\text{ref(\text{flight}).sold } \neq \text{ref(\text{flight}).seats } \land \text{CLI}_{R_7}(S-(\text{ref(\text{flight}))})
\]

The result above shows that:

\((\text{rec} = R_7(S,\text{flight},j) \land \text{CLI}_{R_7}(S,\text{flight},j)) \Rightarrow
[\text{trans}(R_7(S,\text{flight},j))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_7})) \)

\( \checkmark \)

\( R_6(S,\text{flight},j,f) \)

\[
\text{trans}(R_6(S,\text{flight},j,f)) = I F \text{ ref(f) } \in S
\]
\[
\quad \quad \text{THEN rec := } R_6(S,\text{flight},j,f)
\]
\[
\quad \quad \text{ELSE trans(notHaveIt.j } \rightarrow R_6(S,\text{flight}))
\]
\[
\quad \quad \quad \text{END}
\]

\[
[\text{trans}(\text{notHaveIts.j } \rightarrow R_6(S,\text{flight}))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_6})) =
j \in \text{Agencies } \land \text{CLI}_{R_6}(S,\text{flight})
\]

\[
[\text{trans}(R_6(S,\text{flight},j,f))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{R_6})) =
\left( \begin{array}{c}
\text{ref(f) } \in S \land \text{CLI}_{R_6}(S,\text{flight}) \land j \in \text{Agencies } \land \text{ref(f) } \neq \text{ref(\text{flight})} \\
\lor \\
\text{ref(f) } \notin S \land j \in \text{Agencies } \land \text{CLI}_{R_6}(S,\text{flight})
\end{array} \right)
\]

\[
\text{CLI}_{R_6}(S,\text{flight},j,f) \Rightarrow (j \in \text{Agencies } \land \text{CLI}_{R_6}(S,\text{flight}))
\]

\[
\text{CLI}_{R_6}(S,\text{flight},j,f) \Rightarrow (\text{ref(f) } \neq \text{ref(\text{flight})})
\]

\[
\text{CLI}_{R_6}(S,\text{flight},j,f) \Rightarrow (\text{ref(f) } \in S \lor \text{ref(f) } \notin S)
\]

The results above show that:

\((\text{rec} = R_6(S,\text{flight},j,f) \land \text{CLI}_{R_6}(S,\text{flight},j,f)) \Rightarrow
[\text{trans}(R_6(S,\text{flight},j,f))](\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_{P_6})) \)

\( \checkmark \)
\[ R_9(S, \text{flight}, j, f) \]

\[
\text{trans}(R_9(S, \text{flight}, j, f)) = \text{resp} \leftarrow \text{ref}(f).\text{full};
\]

\[
\begin{align*}
\text{IF} & \text{ resp} = \text{YES} \\
\text{THEN} & \text{trans}(\text{fullMachine}.j \rightarrow R_4(S, \text{flight})) \\
\text{ELSE} & \text{trans}(\text{dp}.j!\text{ref}(f) \rightarrow R_4(S - \{\text{ref}(f)\}, \text{flight}))
\end{align*}
\]

\[
\text{IF} \text{ resp} = \text{YES} \\
\text{THEN} \text{trans}(\text{fullMachine}.j \rightarrow R_4(S, \text{flight}))
\]

\[
\text{ELSE} \text{trans}(\text{dp}.j!\text{ref}(f) \rightarrow R_4(S - \{\text{ref}(f)\}, \text{flight}))
\]

\[
(\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_x)) =
\]

\[
\text{IF} \text{ resp} = \text{YES} \\
\text{THEN} \text{trans}(\text{fullMachine}.j \rightarrow R_4(S, \text{flight}))
\]

\[
\text{ELSE} \text{trans}(\text{dp}.j!\text{ref}(f) \rightarrow R_4(S - \{\text{ref}(f)\}, \text{flight}))
\]

\[
(\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_x)) =
\]

\[
\begin{align*}
\text{resp} = \text{YES} & \land j \in \text{Agencies} \land \text{CLI}_x(S, \text{flight}) \\
\lor & \text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLI}_x(S - \{\text{ref}(f)\}, \text{flight})
\end{align*}
\]

\[
\text{IF} \text{ ref}(f).\text{sold} = \text{ref}(f).\text{seats} \\
\text{THEN} \text{resp} := \text{YES} \\
\text{ELSE} \text{resp} := \text{NO}
\]

\[
\begin{align*}
\text{resp} = \text{YES} & \land j \in \text{Agencies} \land \text{CLI}_x(S, \text{flight}) \\
\lor & \text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLI}_x(S - \{\text{ref}(f)\}, \text{flight})
\end{align*}
\]
\[
\left( \text{ref}(f).\text{sold} = \text{ref}(f).\text{seats} \land j \in \text{Agencies} \land CLI_{R_4(S,\text{flight})} \right) \\
\lor \\
\left( \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land CLI_{R_4(S-\{\text{ref}(f)\},\text{flight})} \right)
\]

\[
CLI_{R_4(S,\text{flight},j,f)} \Rightarrow (j \in \text{Agencies} \land CLI_{R_4(S,\text{flight})})
\]

\[
CLI_{R_4(S,\text{flight},j,f)} \Rightarrow CLI_{R_4(S-\{\text{ref}(f)\},\text{flight})}
\]

\[
CLI_{R_4(S,\text{flight},j,f)} \Rightarrow (\text{ref}(f).\text{sold} = \text{ref}(f).\text{seats} \lor \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats})
\]

The results above show that:
(\text{rec} = R_9(S,\text{flight},j,f) \land CLI_{R_4(S,\text{flight},j,f)}) \Rightarrow
[\text{trans}(R_9(S,\text{flight},j,f))](\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x}))

\[
\begin{align*}
\oplus R_{10}(S,\text{flight},pn,j,f) \\
\text{trans}(R_{10}(S,\text{flight},pn,j,f)) = \\
\text{resp} \leftarrow \text{ref}(f).\text{full}; \\
\text{IF resp} = \text{YES} \\
\text{THEN trans(fullMachine.j} \rightarrow R_9(S,\text{flight},pn)) \\
\text{ELSE trans(dp.j}!\text{ref}(f) \rightarrow R_9(S-\{\text{ref}(f)\},\text{flight},pn)) \\
\text{END}
\end{align*}
\]

(\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x}))
= \text{j} \in \text{Agencies} \land CLI_{R_{10}(S,\text{flight},pn)}

[\text{trans}(\text{fullMachine.j} \rightarrow R_9(S,\text{flight},pn))]
(\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x}))

= \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land CLI_{R_9(S-\{\text{ref}(f)\},\text{flight},pn)}

\[
\begin{align*}
\text{IF resp} = \text{YES} \\
\text{THEN trans(fullMachine.j} \rightarrow R_9(S,\text{flight},pn)) \\
\text{ELSE trans(dp.j}!\text{ref}(f) \rightarrow R_9(S-\{\text{ref}(f)\},\text{flight},pn)) \\
\text{END}
\end{align*}
\]

(\forall 1 \leq x \leq 10 \bullet (\text{rec} = R_x \Rightarrow CLI_{R_x}))
\[
\left( \text{resp} = \text{YES} \land j \in \text{Agencies} \land \text{CLI}_R(S, \text{flight}, \text{pn}) \right) \\
\lor \\
\left( \text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLI}_R(S-\{\text{ref}(f)\}, \text{flight}, \text{pn}) \right)
\]

\[
\text{[trans}(R_{10}(S, \text{flight}, \text{pn}, j, f))((\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R))) = \\
\left[ \begin{array}{c}
\text{IF \text{ref}(f).sold} = \text{ref}(f).\text{seats} \\
\text{THEN \text{resp} := YES} \\
\text{ELSE \text{resp} := NO}
\end{array} \right] \\
\text{END}
\]

\[
\left( \text{resp} = \text{YES} \land j \in \text{Agencies} \land \text{CLI}_R(S, \text{flight}, \text{pn}) \right) \\
\lor \\
\left( \text{resp} \neq \text{YES} \land \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats} \land \text{CLI}_R(S-\{\text{ref}(f)\}, \text{flight}, \text{pn}) \right)
\]

\[
\text{CLI}_R(S, \text{flight}, \text{pn}, j, f) \Rightarrow (j \in \text{Agencies} \land \text{CLI}_R(S, \text{flight}, \text{pn}))
\]

\[
\text{CLI}_R(S, \text{flight}, \text{pn}, j, f) \Rightarrow \text{CLI}_R(S-\{\text{ref}(f)\}, \text{flight}, \text{pn})
\]

\[
\text{CLI}_R(S, \text{flight}, \text{pn}, j, f) \Rightarrow \\
(\text{ref}(f).\text{sold} = \text{ref}(f).\text{seats} \lor \text{ref}(f).\text{sold} \neq \text{ref}(f).\text{seats})
\]

The results above show that:
\[
(\text{rec} = R_{10}(S, \text{flight}, \text{pn}, j, f) \land \text{CLI}_R(S, \text{flight}, \text{pn}, j, f)) \Rightarrow \\
[\text{trans}(R_{10}(S, \text{flight}, \text{pn}, j, f))((\forall 1 \leq x \leq 10 \cdot (\text{rec} = R_x \Rightarrow \text{CLI}_R))] \\
\checkmark
\]