Mobile B Machines

D.A. Karkinsky

Submitted for the Degree of Doctor of Philosophy from the University of Surrey

Unis

School of Electronics and Physical Sciences University of Surrey Guildford, Surrey GU2 7XH, U.K.

July 2007

© D.A. Karkinsky 2007
Abstract

Specifications and implementations of systems need to be concerned with the interactions that can occur within a system and model the data structures appropriately. We are interested in combinations of formal methods which consider the state and dynamic requirements of a system. We recognise that many such combinations already exist, including, CSP || B and Circus, but we are concerned with a state description, being accessed and updated by control components with dynamically reconfigurable interconnections. Our work is motivated by what we see in Peer-to-Peer networks and Object-Oriented systems where instantiation and dynamically reconfigurable interconnection are essential paradigms. For example, in a Peer-to-Peer network nodes can act as both server and client in exchanging data to complete a certain task. Nodes are also independent and can leave or join the network at any time. In Object-Oriented systems, an object instance can be created with a unique reference. This reference can be used by other objects to communicate with the object. Our aim is to provide a formal framework which supports this kind of interaction so that the integrity of each active object or node is preserved, and so that we can reason about the overall behaviour of the system.

The approach we consider in this thesis is a combination of the π-calculus and the B-Method. In order to be able to reason about specifications based on both these notations we need common semantics. We define an approach which enables the interpretation of a B machine as a π-calculus labelled transition system. This allows the integration of machines into parallel combination with π-agents. As a result, this work extends B machines with instantiation and π-calculus dynamic reconfiguration capabilities.

We use a behavioural type-system with variant types to maintain low level server/client style consistencies between instances of machines and π-process agents. (For example, all agents call operations that relate to some machine in the specification.) Using the type system, we identify a class of π-agents whose behaviour with respect to the machine instances allows a weakest pre-condition style proof to be carried out on the agents. We use this property to define an approach for detecting agents that might cause a machine instance to diverge.
Acknowledgements

I would like to thank my supervisors Dr Helen Treharne and Professor Steve Schneider for their continued support, patience and useful comments. I would also like to acknowledge Dr David Pitt for his help when I came to Surrey to complete the PhD.

Finally I would like to thank my parents and friends for their understanding.
...to my grandfather Professor Toma Dankov
# Contents

1 Introduction  
   1.1 Background ................................................................. 4  
   1.1.1 Failures-divergences semantics of machines ................. 4  
   1.1.2 Consistency of combined specifications ....................... 5  
   1.2 Thesis aims ................................................................. 7  
   1.2.1 Motivating dynamic behaviour using an example ............. 8  
   1.2.2 Properties and benefits of a π-calculus and B integration  10  
   1.3 Thesis overview ............................................................ 12  

2 Theoretical foundations of the B-Method  
   2.1 Overview of the B-Method ................................................ 13  
   2.2 B predicate language and interpretation ............................ 14  
   2.3 The MACHINE declaration ............................................. 14  
   2.4 Abstract Machine Notation and semantics ....................... 15  
   2.5 Some useful properties .................................................. 20  
   2.6 Predicate type system .................................................... 21  

3 Theoretical foundations of the π-calculus  
   3.1 Process syntax ............................................................. 23  
   3.1.1 Binding, α-conversion ................................................ 24  
   3.1.2 Examples of π-calculus agents ................................... 26  
   3.2 Operational semantics ................................................... 28  
   3.2.1 Structural congruence ................................................ 28  
   3.2.2 Reduction .............................................................. 30  
   3.2.3 Labelled transition semantics .................................... 32
### Contents

3.2.4 Some useful results ................................................................. 36
3.3 Reasoning about processes using traces ........................................ 36
3.4 Type system ................................................................................ 37
  3.4.1 Type grammar and process syntax ........................................... 38
  3.4.2 The variant construct ........................................................... 38
  3.4.3 Typing derivations .................................................................. 40
  3.4.4 Typed labelled transition rules .............................................. 44
  3.4.5 Properties of a type system .................................................. 44

4 Linking the B-Method and $\pi$-calculus ......................................... 47
  4.1 Preliminary labelled transition system ........................................... 47
    4.1.1 A set of states and a set of labels ....................................... 47
    4.1.2 A set of initial states ....................................................... 48
    4.1.3 A labelled transition relation for machines without I/O .......... 49
  4.2 Correspondence with Morgan's failures-divergences semantics for action systems ....................................................... 51
    4.2.1 Preliminaries ...................................................................... 51
    4.2.2 Correspondence ................................................................... 52
  4.3 Linking $\pi$-calculus and B-Method ............................................ 58
    4.3.1 The link between $M$ and $\pi$ .............................................. 59
    4.3.2 The $\pi$-calculus type-system and $M$ .................................... 60
    4.3.3 $\pi$-style operational semantics for Machines without I/O ....... 62
    4.3.4 Examples with machine Clock ............................................. 63
  4.4 Some results about machine agents ............................................... 65
  4.5 The $MG\text{E}\text{N}\text{E}\text{RATOR}$ specification ........................................ 68

5 Constructing Combined Specifications .......................................... 71
  5.1 Combined specification in $\pi$ and B .......................................... 71
    5.1.1 Mediators .......................................................................... 71
  5.2 Constructing a mediator ............................................................ 79
    5.2.1 Mediators with parallel, hiding and infinite replication .......... 80
    5.2.2 Sequential finite controllers ............................................... 92
  5.3 Control systems ........................................................................... 102
  5.4 Semantic definition of machine divergence freedom ...................... 107
## Contents

6 Verification of a Combined Specification 109  
6.1 Machine consistent mediators ........................................... 109  
6.2 Example ............................................................................. 115  
6.3 Some useful results about $DIV$ ........................................... 121  
6.4 Machine consistent mediators with infinite replication .......... 152  
6.5 Consistency with semantic definition of machine divergence freedom .. 157  

7 Resource Allocation System 169  
7.1 System overview ................................................................. 169  
7.2 Specification ................................................................. 169  
7.2.1 Putting the system together ............................................. 172  
7.2.2 Demonstrating dynamic behaviour .................................. 173  
7.3 Machine divergence freedom verification .......................... 174  
7.3.1 Establishing consistency using $DIV$ ............................ 175  
7.4 Discussion ................................................................. 179  

8 Conclusion 183  
8.1 Contribution ................................................................. 183  
8.2 Related work ................................................................. 185  
8.3 Future work ................................................................. 186  
8.3.1 Extension to machines with input and output ................. 187  
8.3.2 Polyadic control points ................................................. 190  
8.3.3 Dynamic control points ................................................. 190
List of Figures

1.1 Napster architecture ................................................................. 2
1.2 Example component: MACHINE and controller ....................... 4
1.3 A CSP and B combined system architecture .............................. 6

2.1 The MACHINE declaration ........................................................ 15
2.2 The Clock machine. ................................................................. 16
2.3 Relational Semantics vs Predicate Transformer Semantics ............ 17

3.1 Hidden Channel is Announced (Bound Output) .......................... 26

4.1 The familiar transition of Clock ................................................ 51
4.2 Executing an operation in π-calculus ....................................... 58
4.3 Architecture between machines and processes ........................... 59
4.4 π-calculus style LTS of Clock machine .................................... 64
4.5 Machine Creation ................................................................. 65
4.6 Machine Creation Transitions .................................................. 65

5.1 Dynamic transfer of a machine ................................................ 72
5.2 Architecture we want to avoid ................................................ 73
5.3 Simple proof method for mediators ........................................... 78
5.4 Proof method related to Lemma ................................................. 80

7.1 The B description of a Node ..................................................... 170
7.2 SERVER_i in its start state ....................................................... 170
7.3 Description of a server controller ............................................. 171
7.4 Possible responses to inc_i (and also to req_i) from SERVER_i .......... 171
7.5 The architecture of RAS ........................................................ 172
List of Figures

7.6  $SERVER_i$ following creation of a node ....................................................... 173
7.7  Requesting and passing a node from $SERVER_k$ to $SERVER_i$ ................. 173
7.8  Identifying a controller ................................................................. 174
7.9  Assertions associated with the control points within RAS ................. 174
Chapter 1

Introduction

Implementations of distributed systems involve setting up a network or networks, managing the communication that occur between the nodes in a network and also the transfer of data between nodes. Networks can be static and made up of a fixed number of nodes or they can be more dynamic in which the number of nodes in a network may vary. When proposing an implementation of a distributed system, developers must be concerned with network architectures, their capacity, and bandwidth [14]. For example, when deciding upon an architecture a decision has to be made as to whether each part of a network performs the same role, acting as a server and a client. A network in which the nodes can act as a client or a server is considered to be a Peer-to-Peer (P2P) network.

In a P2P network clients can connect to each other directly. This is not the case in a standard client-server network. The difference can be exemplified with a file sharing system. In a client-server model, clients must transfer files via the server and the server must have sufficient capacity to deal with the storage of all the files. However, in a P2P network the central server is used to setup and manage the connection between clients but the file transfer is done between the clients directly. The clients would need to be able store the files to be transferred. Clients can contact the server for additional services. In a Peer-to-Peer network the bandwidth of the servers in a network do not impact on the bandwidth available to the clients because they are connecting directly to each other. One example of a popular P2P network was Napster[47], which was an application of a dynamic distributed system which facilitated music sharing between users. When a user joined a network they announced the files they wished to share to a central server. A search request from a user results in that user being informed as to the other peers in the network who have these files available for download. A protocol then supports the connection to the identified peers so that the file can be downloaded. The peers only connect to the server in order to join the network and for announcing and searching for files. For example, Figure 1.1 illustrates a message exchange between a node and a server and the link between nodes which would be initiated following the message exchanges.

P2P networks are typically large-scale with potentially millions of nodes which join and leave regularly, and from the above description they are autonomous but co-operate to
Chapter 1. Introduction

Figure 1.1: Napster architecture

share and retrieve resources. Much of the research which has been conducted in the area of P2P Networks has focused on describing possible network architectures and also on simulations to reason about the performance of networks. Orthogonally, the formal methods community has also contributed to reasoning about distributed systems but has focused on proving the correctness of communication protocols. For example, the process algebra CSP [24], has been used to prove properties of communication protocols and security protocols [13, 40]. The research emphasis has not been on examining networks which change or evolve in their architecture. We are interested in exploring how to specify systems which are dynamic in nature (mobility) so that we can support:

- dynamic instantiation,
- dynamic interaction,
- evolving capability.

In this thesis we will focus on the first two aspects. We want to be able to describe dynamic systems and be able to reason about how data is passed around in such systems. In the case of P2P networks this means being able to support varying the number of nodes in a network, and changing the way nodes are connected in a network.

The viewpoint from which we begin our investigation into modelling distributed systems is commonly referred to as the multi-view approach [6]. This approach is analogous to modelling an object using traditional engineering methods. In traditional engineering when presenting a model of some 3-D object, it is normal to present a diagram that shows the object from several positions (views). A simple 3-D object would have four views: front, side, top and the perspective. Objects with more complicated faces will
require more views so that we can accurately convey the nature of an object. Consistency between the views is expressed by measuring the lengths of the same edges in the different views and determining that they are the equal. Without consistency the views of an object cannot be combined into a unified view.

In modelling systems using formal methods, at least two views are always identifiable; these are the dynamic-view and the state-view. The dynamic-view is predominantly concerned with concurrency and execution control such as loops, branches and procedure calls. The state-view is concerned with variables their relationships, and procedures to change the values of the variables. Notations such as the process algebra CSP is suitable for capturing dynamic-view requirements whereas state based notations such as B-Method [1], Z [69] and Object-Z [57] are suitable for capturing state-view requirements. Thus, if we want to use the multi-view approach to specifying systems, using the most appropriate notations, we will be required to specify at least two models. Verifying the consistency of these two models requires a common semantics between the notations used.

Examples of integrated formal notations which combine both the dynamic and state-based view are CSP2B [9], CSP||B [56], Circus [68], and CSP-OZ [58]. All of these notations make use of CSP and B, Object-Z or Z. The CSP2B approach converts the CSP view of a specification into the B notation and therefore does not retain separation of the two views. In contrast separation of the views is an inherent part of CSP||B. Furthermore, CSP||B provides techniques for establishing consistency between a B specification and a CSP specification, and these will be reviewed in the next section. Circus combines Z and CSP but instead does not force the same degree of separation as in CSP||B since it allows Z constructs and CSP constructs to be freely mixed within a specification. This is because the underlying semantics is based on the Unified Theories of Programming [25] and CSP and Z are re-defined in this model. The benefit of this approach is that it includes an expressive refinement calculus but any tools for Circus will need to be built from scratch. However, a model checker [20] is emerging for the approach and also an automatic translator from Circus to JCSP [19]. Finally, CSP-OZ is an approach which combines CSP and Object-Z, retains the separation of concerns between the notations and supports compositional verification based on slicing techniques [8].

We consider retaining separation between the two views as important because we believe that it results in clearer specifications and also enables the use of existing tools and techniques. We also wish to retain this separation when specifying systems which are dynamic in nature. This is important so that dynamic creation and dynamic interaction occurring within a system can be represented solely in its dynamic-view. Trying to incorporate these aspects into a system's state-view would result in complicated models. Up until now, none of the above approaches have presented a mature approach to modelling systems with mobility. An extension to Circus is being investigated which relates to incorporating mobility [61].
In the following sections we review the CSP||B approach with particular emphasis on its architectural limitations related to dynamic creation and dynamic interaction.

1.1 Background

The Formal Methods group at the University of Surrey have been working on developing techniques for specifying and verifying distributed systems so that the patterns of behaviour are made explicit and captured completely separately from the specification of the data in the system. The approach is referred to as CSP||B. CSP is a process algebra, first introduced by Hoare [24], that is concerned with the evolution of systems as they execute sequences of events. The B-Method, proposed by Abrial [1], focuses on defining how the state of a system can be queried and updated through operations. Early papers detailing the CSP||B approach [63, 64] focused on the sequential aspects of CSP and were concerned with identifying how sequences of events could control the way the data was being updated. The CSP processes are referred to as controllers. Consider the example in Figure 1.2 which illustrates a controller, called $RCtrl$, cycling through a sequence of operation calls of the $Repeater$ machine. Note that each operation is associated with an event. We will revisit this correspondence in Chapter 4.

**MACHINE Repeater**

**VARIABLES** $n$

**INITIALISATION**

$n := 0$

**OPERATIONS**

\[
\begin{align*}
\text{reset} & \equiv n := 0; \\
\text{inc} & \equiv \text{PRE } n \geq 0 \text{ THEN } n := n + 1 \text{ END} \\
\text{do} & \equiv \text{PRE } n > 0 \text{ THEN} \\
& \quad \text{CHOICE } n := n + 1 \text{ OR } n := n - 1 \text{ END} \\
& \end{align*}
\]

**END**

$RCtrl = \text{inc} \rightarrow \text{do} \rightarrow \text{reset} \rightarrow RCtrl$

Figure 1.2: Example component: MACHINE and controller

1.1.1 Failures-divergences semantics of machines

In order to consider the composition of a controller and a machine, the B machine needs to be provided with a CSP semantics. Morgan [37] provides failures-divergences
1.1. Background

semantics for event systems in terms of the weakest precondition of a sequence of operations, and this enables other CSP semantics to be given to B machines [56].

A sequence of operation calls is a trace of $M$ if it can possibly occur, that is when it is not guaranteed to block. This is defined as follows

**Definition 1.1.1.** The traces of a machine $M$ are those for which $\neg wp(init; \bar{t}_r, false)$ holds.

$\bar{t}_r$ is the sequential composition of the operation calls of $M$, and $init$ is the initialisation of the system. The CSP trace $(inc, do, inc, inc, do, reset)$ is considered a trace of the *Repeater* machine since $\neg wp(init; inc; do; inc; inc; do; reset, false)$ holds. Note that there is a one-to-one correspondence between the CSP events and operation calls. The CSP||B approach only concerns with non-blocking B machines (apart from in [64]) and so all sequences of operation calls are possible traces.

In order to differentiate between divergent and non-divergent traces we need to determine whether the sequence is guaranteed to terminate following initialisation (i.e. establish $true$). A sequence is a divergence if it is not guaranteed to establish $true$ (e.g. $\neg wp(init; inc; do; do, true)$). Furthermore, this means that at least one operation within that sequence is called outside its precondition. In this example, the second invocation of the $do$ operation may occur in a state where the precondition $n > 0$ does not hold (because it is possible to reach a state in which $n = 0$).

A sequence does not diverge if it is guaranteed to terminate. This is defined as follows

**Definition 1.1.2.** Non-divergent traces of a machine $M$ are those for which $wp(init; \bar{t}_r, true)$ holds.

Since $wp(init; inc; do; reset, true)$ holds, the trace $(inc, do, reset)$ is non-divergent. This ensures that all preconditions hold when the operations are called.

1.1.2 Consistency of combined specifications

B and CSP were chosen because they were individually mature notations with strong tool support [18, 39]. The goal in their integration was to preserve the original semantics of both languages whilst building a framework for defining and reasoning about a combined system. The overhead of keeping both descriptions separate is the additional proof obligations that need to be proved for each particular system in order to show that the combined views are consistent. Consistency in the CSP||B approach means demonstrating divergence freedom and deadlock freedom of the process/machines pairs. Divergence-freedom is at the core of the approach and this must be established before considering any other safety and liveness properties of interest. Ensuring divergence freedom means the safe execution of operations within their preconditions. This is an
important property because the invariants of machines in a combined specification need to be preserved. If operations are called outside their preconditions then no guarantees can be made about the state of the machines and hence their invariants.

The first paper on CSP||B [63] only considered one process and one machine and this pair was referred to as a component. In Figure 1.2 it is clear that the preconditions of the operations hold when the operations are called by $R\text{Ctrl}$. As we stated above, divergent behaviour can occur by performing repeated $do$ operation calls since the precondition may not always hold. Previous results [62] have identified conditions sufficient to guarantee $P \parallel M$ to be divergence free for a controller $P$ and a machine $M$. These results require the identification of a control loop invariant (CLI) on the state of the B machine $M$, which must be true at every recursive call. For example, an appropriate CLI for the component described in Figure 1.2 is $n = 0$. It is established by considering the semantics of the B operations as they are called within the controller, and essentially computing the weakest precondition required to establish the CLI. Discharging this weakest precondition proof obligation is referred to as consistency checking. Consistency checking in this context was based on a CLI technique [62].

Later work by Schneider and Treharne developed a particular architectural style of specification which enables a composition of these components. The architecture ensures that any interaction between components is restricted to communication between controllers [55]. Hence, machines cannot communicate with each other directly, but only via their respective controllers, as is shown in Figure 1.3. The controller processes can also perform events which do not have corresponding B operations.

Theoretical results have been established in the approach to show that divergence freedom and deadlock freedom can be proven for specifications which are made up of several B machines and CSP processes [55, 56]. In [55] Schneider and Treharne establish that a number of $P_i \parallel M_i$ components, where $P_i$ is a CSP process and $M_i$ is a machine, are divergence free once we have shown that each component $P_i \parallel M_i$ is divergence free. Consistency checking via the CLI technique is used to prove individual components and then, from the CSP semantics of parallel composition, divergence freedom is preserved.

Some attempts have been made to provide tool support for the CLI technique. For
1.2. Thesis aims

example, in [7] an evaluation of a shallow embedding of B in the theorem prover PVS, proposed by Munoz et al. [38], concluded that the consistency checks could not be supported in within this framework. The proof obligations that were produced using the embedding were not of the correct form. Consistency checking means checking each precondition so that each operation can be executed safely. In this work the first precondition was always assumed to be true which was not adequate. However, the thesis did clarify the role of control predicates in the control loop invariant technique. They were seen as predicates which captured the information held within the process parameters of the CSP controllers and their relationship with the underlying state in the B machines. In this thesis we will also use a notion of control predicates but it will be slightly different, as we shall see in Chapter 5.

More recently, Evans and Treharne [17] have proposed a deep embedding of the B relational model again in PVS. They also utilise use of previous work by Dutetre and Evans [13, 40] to prove properties based on CSP traces, sequences of events. CSP has three main semantic models that can be used to reason about CSP process: traces model, stable failures, and failures-divergences model [46, 53]. Using the B relational model and the traces model within PVS allows the proofs provided by Treharne in [62] to be mechanised and support for consistency checking can be provided within a theorem prover.

The architecture in CSP∥B is based on $P_i \parallel M_i$ component building block. The notion of defining specifications based on encapsulated components is also being considered by Lau et al. [29]. In their emerging work they consider a component as an encapsulated object with an invocation unit. These units are not like the CSP processes in CSP∥B because they do not provide any control. However, in Lau et al.’s work they define a set of connectors to group components together. When systems are composed using these connectors they exhibit some properties by virtue of the composition. The composition is hierarchical and does not allow any dynamic treatment of the components, i.e. they cannot be dynamically re-configured to interact with other components during the evolution.

The architecture in CSP∥B is also very strict. It does not support:

- the dynamic creation of B machines since all $M_i$ components have to be identified upfront,
- dynamic interaction since all CSP channels in the $P_i$ components are statically defined and each $M_i$ component is tightly coupled with its $P_i$ controller.

1.2 Thesis aims

In the previous section we noted that the architecture of a CSP∥B specification is not flexible. In a P2P network we need to be able to create new nodes and pass nodes
between servers, and this requires a dynamic architecture. We also need to reason about the integrity of the data held within a system.

Thus, we need a formal framework which enables us to reason about dynamic creation of state and mobility of state. We could consider extending CSP||B to deal with these aspects. The following example shows that it may be possible to hard-code a notion of dynamic behaviour in CSP but we would also need to extend the verification techniques in order to reason about the integrity of the data as it is passed around in a specification. It may be possible to do this in CSP||B but it may not provide a general solution, in this thesis we prefer to examine a new combination of a mobile process calculus and B. The reason is that we want to see what benefits can be gained from using an established mobile calculus rather than extending CSP to capture the notion of mobility. We feel that this research will inform possible extensions of CSP||B to deal with mobility. In choosing to integrate the \( \pi \)-calculus with B the major issues to be addressed in developing our new \( \pi \mid B \) framework are the following:

- establishing a common semantic framework so we can consider \( \pi \) and B in parallel,
- structuring the allowed communication patterns between \( \pi \) and B (using a type system),
- verifying that \( \pi \)-specifications call the B operations within their preconditions and ensure that this remains true when the B machines instances are being passed around in a specification.

In this thesis we concentrate on developing a verification framework to ensure that machine instances do not diverge as a result of the \( \pi \)-specification calling operations outside their preconditions. As in CSP||B this is the core property that needs to be verified in order to ensure consistency between an event-based and state-based specification. It is the most important property because other properties such as deadlock freedom can only be verified after divergence freedom has been established.

### 1.2.1 Motivating dynamic behaviour using an example

Generally dynamic instantiation and reconfiguration are not native to CSP. For example, a standard CSP example is the "Dining Philosophers Problem" [53]. Each philosopher and fork has the same behaviour so using the CSP renaming mechanism, it is possible to instantiate \( n \) philosophers, \( n \) forks and reason about the behaviour of that system. However, the model does not cover the situation where another philosopher wishes to join the system with one fork or when a philosopher wishes to leave the system. Note that what is true about a model with \( n \) philosophers and \( n \) forks is also true about a model with \( n + 1 \) philosophers and \( n + 1 \) forks. In some systems we do not need to be concerned about such additional behaviour. However, there are systems where dynamic instantiation and reconfiguration are essential in order to provide correct behaviour. For example, consider the following system where the system’s progress is dependent on the number of active components.
Example 1.2.1. The Insectologist Field Day.
In a certain field in a village called Leskovodol at summer there are a number of grasshoppers. A number of insectologists are having a field day trying to catch them.

The principal aim of an insectologist is to catch a grasshopper. The principal aim of the grasshopper is to avoid the insectologists and find a mate.

The principal way an insectologist catches a grasshopper is to sit in one spot on the field and listen for the grasshopper’s mating call. Once he hears such a call he would identify the location of the grasshopper and indicate to his colleagues that he is pursuing the grasshopper at that location. An insectologist can chase at most one grasshopper. He may catch a grasshopper if he is near a grasshopper and sees it skip.

A caught grasshopper has no further useful behaviour.

The principal technique for finding a mate available to the grasshopper is generating a mating call which gives away his location. If a grasshopper hears the call of another grasshopper he can choose to skip towards the other grasshopper otherwise it remains still, generating calls whenever it can.

An important technique for avoiding insectologists available to a grasshopper is to stop calling once it detects that an insectologist is near. In such situation the grasshopper would remain still (unless it hears a call of another grasshopper). Hopefully, the insectologist would then hear the call of another grasshopper near by, and think that the first grasshopper has moved on. If none of his colleagues are moving to that location he would alter his course in pursuit, otherwise he would remain still and listen for a call from a new location or see something skip.

In order to build a CSP model of the system above n insectologists and s active grasshoppers need to be specified. As grasshoppers are caught in the system s decreases.

Suppose a requirement of the system is that in all evolutions of the system an insectologist may eventually catch a grasshopper. Suppose another requirement is that as long as there are two distinct grasshoppers in the field they can eventually find each other. In order to build a model of the system with the above properties dynamic creation is required. Without dynamic creation, new grasshoppers cannot be added. Thus, in the case where the number of insectologists becomes greater than the number of grasshoppers it is possible to reach a state where each grasshopper has an insectologist that is near it. The insectologist would sit and listen for a new call but no grasshopper would feel safe to generate a new call. A \( \pi \)-calculus model would not suffer this drawback because the possibility of a new grasshopper joining the field would always be available.
In addition to dynamic creation it is also important to ensure unique identity of the grasshopper components and this is illustrated using the following example. A form of dynamic creation can be modelled in CSP using internal choice indexed over an infinite set and interleaving.

**Example 1.2.2.** Given \( P \subseteq \mathbb{N} \),

\[
GRASSHOPPER(P) = \bigcap_{x \in \mathbb{N} \setminus P} (create(x) \rightarrow ACTIVEHOPPER(x) ||| GRASSHOPPER(P \cup \{x\}))
\]

However, one has to be careful with the process above because its semantics do not actually mean, that under all contexts (or environments), the identities of all ACTIVEHOPPER components that the GRASSHOPPER\( (P)\) process can generate are all distinct. This would cause problems with composition of various distributed parts of a system unless instantiation is managed centrally. A \( \pi \)-calculus process modelling this behaviour is as follows:

\[
GRASSHOPPER_{\pi} = !\nu \nu (create(\nu).0 \mid ACTIVEHOPPER_{\pi}(\nu))
\]

In essence \( GRASSHOPPER_{\pi} \) is replicated infinitely. Each replication can output the channel \( \nu \), which can be interpreted as a unique identity, on channel \( create \). The semantics of the calculus means that \( \nu \) is distinct from any other free channel found in any other part of the system. Consider the following environments in each case,

\[
\begin{align*}
CSP &: ACTIVEHOPPER(1) ||| GRASSHOPPER(\emptyset) \\
\pi &: ACTIVEHOPPER_{\pi}(z) \mid GRASSHOPPER_{\pi}
\end{align*}
\]

In CSP we might reach a point where two grasshoppers are instantiated with identity 1. However, in the \( \pi \)-calculus \( z \) is considered to be a free channel so there is no possibility that \( GRASSHOPPER_{\pi} \) can instantiate another grasshopper with channel \( z \). Thus, the implications of \( \pi \)-calculus dynamic instantiation are slightly stronger.

The above example illustrated the need for flexibility when considering dynamic instantiation and interaction. We showed that \( \pi \)-calculus has built in constructs to manage this whereas in CSP it had to be achieved in a hard-coded way.

### 1.2.2 Properties and benefits of a \( \pi \)-calculus and B integration

The \( \pi \)-calculus is a process algebra with operational semantics in the style of [44]. There are fully abstract denotational models for the \( \pi \)-calculus such as [60, 23, 22]. However, (unlike in CSP) these have not been based on classic set theory hence are difficult to use in practice. Therefore, integrating \( \pi \)-calculus and B must take place on the basis of \( \pi \)'s operational semantics. Although, it is recognised that such semantics do not offer extensional accounts of the behaviour of processes, it is a starting point for giving any
1.2. Thesis aims

Furthermore, the π-calculus is primarily a name-passing algebra, which means that it
does not come boxed with basic data values such as integers and sets. A remarkable
approach to including them in the syntax is to model them as processes. For example,
in [33] the advised method for incorporating the values \{true, false\} in the syntax is
to encode them as follows (where we use pseudo language to illustrate that each value
requires a process with three states),

\textbf{Definition 1.2.1.} Given a channel } x, \textit{true } \triangleq \textit{1. Input on } x \textit{ channels t and f},
\textit{2. output on t},
\textit{3. stop.}

\textit{false } \triangleq \textit{1. Input on } x \textit{ channels t and f},
\textit{2. output on f},
\textit{3. stop.}

This approach of modelling data as processes has influenced a study on the expressibility
of the π-calculus with respect to other languages. For example, it has been found
that the typed π-calculus can act as a suitable abstraction [50] for object oriented lan-
guages such as the Object Calculus of Abadi and Cardelli. Other studies in this domain
are [65, 66, 26, 43].

In our work we draw inspiration from many of the above mentioned studies. Most im-
portantly of all, we recognise the need for a type-system over the π-calculus agents when
considering systems with machines. We considered the type-system expressed in [21]
which is specifically designed for enforcing requirements on the channels in client/server
architectures using session types. However, the simpler type-system of Sangiorgi et al.
in [52] is sufficient for our purposes because we only consider machines without input
and output parameters.

Unlike the research on expressibility of the π-calculus, we take an abstract view of state
and operations and therefore determining whether the B-Method syntax is completely
encodable into the π-calculus is not part of this study. In giving a π-semantics to B
machines the principal question that we ask is: if there is an encoding what transitions
would we like a machine to support. We capture the behaviour of a machine directly
into the labelled transition relations for the π-calculus. Furthermore, whilst the ap-
proach of encoding data as processes mentioned above is robust, we needed three states
to model each truth value. Therefore, it is not clear how a specification based on such
an approach would deal with complex data or indeed scale down for model checking.
In general when attempting to model state in a process algebra one is faced with a
state explosion problem and therefore there is much to be gained from separating out
the data-rich aspects of a specification, which is not needed to determine the flow of
control. Therefore, allowing B to handle data for the \( \pi \)-calculus offers the merit of reduced process complexity.

We have noted above that the \( \pi \)-calculus can act as a suitable abstraction for object oriented languages which means that at an implementation level the \( \pi \)-calculus relates to object oriented languages in the same way that the B syntax relates to imperative languages. Some research in the B community has also focused on identifying relationships between the B method and object-oriented systems, for example [31, 11, 59]. Snook et al. points out most object-oriented systems follow a hierarchical structure. This is echoed in Lau et al.’s [29] observations but they go further in saying that the classes at the bottom of the hierarchy are mainly concerned with state whereas those higher up deal with coordinating behaviour. Therefore, the separation of state and behaviour expressed in this thesis corresponds to reasoning about systems in the object oriented domain, where the objects concerned with state could be modelled in B and those with coordinating behaviour could be described using \( \pi \)-calculus.

1.3 Thesis overview

Chapter 2 presents background material on the B-Method and the specification language for abstract machines. Chapter 3 presents the particular style of \( \pi \)-calculus used in the thesis. Chapter 4 begins the technical contribution of the thesis, by showing how to combine \( \pi \)-calculus and B. This is done by defining a labelled transition semantics for B machines, so that they can interact with \( \pi \)-calculus processes. A type system is used to structure interaction between \( \pi \) processes and B machines. Chapter 5 discusses specifications of systems combining \( \pi \)-calculus and B, focusing on their construction and reasoning about them. In particular we focus on the verification of machine divergence freedom, which means checking that machine operations are called when their preconditions are satisfied. Chapter 6 presents a technique for verifying machine divergence freedom by means of rely-guarantee, weakest-precondition style proof obligations. Chapter 7 presents an example specification called the Resource Allocation Service (RAS), demonstrating the mobile and dynamic features of our new \( \pi \mid B \) framework. Finally, Chapter 8 concludes and indicates directions for further research.
Chapter 2

Theoretical foundations of the $B$-Method

2.1 Overview of the B-Method

The B-Method [1] provides a methodology and notation for modelling requirements, specifying component interfaces, capturing design decisions, providing implementations and maintenance, within the framework of a formal software development life cycle. The key principles are incremental construction of layered software components and correspondingly the incremental verification and validation of these components. It is an industrial strength method and has been used to develop transport systems including the Paris Metro [3] and New-York Subway [16]. The Paris Météor system is a driverless system controlling 8 stations, 9 trains over 8.5km servicing 350,000 passengers per day. The Canarsie Line in New York is also a driverless system controlling 24 stations, 53 trains over 17km and is required to be operational 24 hours per day, 7 days per week, and involves interoperability between lines and suppliers.

The B-Method is based on first-order predicate calculus with set notation. It includes a simple 'pseudo' programming language called Abstract Machine Notation(AMN). AMN includes mechanisms for structuring components which enforce information-hiding and data encapsulation similar to those found in object-oriented approaches. This facilitates the abstract specification of different components within a large development, while enforcing rigorous control of component interfaces. Abstract specifications can be verified to ensure the correctness of software components.

The B-Method also supports refinement steps by which abstract models are transformed into lower-level, more concrete specifications from which code can be generated. The AMN can be used during the whole development lifecycle. In this thesis we focus on the abstract specification mechanism of B and how to combine these with $\pi$-calculus specifications.
Currently there are several tools which support the B-Method such as the B-Toolkit [39] and ProB [30]. The toolkit is made up of a suite of tools to support the formal software development lifecycle described above. The tools are also capable of translating implementations into highly maintainable, separately compilable source code (C) and executables.

2.2 B predicate language and interpretation

The B-Method is essentially a first-order predicate calculus language with set notation. In this thesis we denote the set of all B predicates as \( \mathcal{R} \) and use \( R \) to denote a member of \( \mathcal{R} \). We also identify a value-domain into which B predicates are interpreted as \( \mathcal{D}_B \).

As in any first-order predicate calculus language, the B-Method is defined on an infinite set of names which are used for free and bound variable names in expressions. We denote the set of all free names of a predicate \( R \), with \( \text{fn}(R) \) which is finite and the set of bound names with \( \text{bn}(R) \).

For a given \( R \), we define a function \( \text{val} \) which returns values for free names during its interpretation.

**Definition 2.2.1. Valuation function**

Given a B predicate \( R \), a valuation function \( \text{val} \) for \( R \) is defined as follows,

\[
\text{val} \in \text{fn}(R) \rightarrow \mathcal{D}_B
\]

Valuations, which are disjoint on the set of variables on which they are defined, can be combined. The notation \([\text{val}_1, \text{val}_2]\) where \( \text{val}_1 \) and \( \text{val}_2 \) are valuations such that \( \text{dom}(\text{val}_1) \cap \text{dom}(\text{val}_2) = \emptyset \) is the common extension of \( \text{val}_1 \) and \( \text{val}_2 \).

**Example 2.2.1.** Let \( \text{val}_1 = \{(x, 1)\} \) and \( \text{val}_2 = \{(y, 2)\} \). Then,

\[
[\text{val}_1, \text{val}_2] = \{(x, 1), (y, 2)\}
\]

Note that the bound names of any predicate we consider here are chosen to be different from any free names of the predicate. Given a predicate of the form \( \exists x. R \) we say \( \text{val} \) satisfies \( \exists x. R \) if there is a value \( s \in \mathcal{D}_B \) such that interpreting \( R \) with \([\text{val}, \{(x, s)\}]\) is true. Similarly, given a predicate of the form \( \forall x. R \) we say \( \text{val} \) satisfies \( \forall x. R \) if for all values \( s \in \mathcal{D}_B \), interpreting \( R \) with \([\text{val}, \{(x, s)\}]\) is true.

2.3 The \textsc{Machine} declaration

The main specification construct within the B-Method is a machine. A machine declares a list of variables and operations which modify and query those variables. The notation used to write a machine is AMN discussed below. In this thesis we use \( M \) to
Abstract Machine Notation and semantics

2.4. Abstract Machine Notation and semantics

We have already seen that operations can take the form \textit{PRE R THEN S END}. Definition 2.4.1 identifies all the AMN syntax used in specifying initialisation clauses and operations of machines. Note, \textit{S_1; S_2} is not permitted in the bodies of operations. Rather this construct will be relevant when we consider sequences of operations.

Throughout the thesis we use the specification of a typical \textit{Clock} shown in Figure 2.2 with two operations \textit{tick} and \textit{tock} to illustrate various aspects of our work. When executing an operation such as \textit{tick}, the precondition is assumed to be true, and it is the responsibility of the caller of the operation to ensure that its precondition is indeed true. Executing \textit{tick} changes the state of the machine and the resulting state must satisfy the invariant. Clearly, executing \textit{tick} followed by \textit{tock} is valid but \textit{tick} followed by \textit{tick} is invalid and we will discuss this in more detail in the sections that follow.

\textbf{MACHINE} M

\begin{itemize}
  \item \textbf{SETS} Identifier; Identifier = \{Constant, \ldots, Constant\}
  \item \textbf{CONSTANTS} c_L, \ldots, c_m
  \item \textbf{PROPERTIES} J
  \item \textbf{VARIABLES} \text{var}_1, \ldots, \text{var}_n
  \item \textbf{INVARIANT} I
  \item \textbf{INITIALISATION} T
  \item \textbf{OPERATIONS}
    \begin{itemize}
      \item operation \triangleq \textit{PRE R THEN S END}
    \end{itemize}
\end{itemize}

\text{Figure 2.1: The} \textit{MACHINE} \text{declaration}

denote a generic machine whose pattern is given in Figure 2.1. The \textit{SETS} clause defines the finite sets upon which a specification is built. Constants within a specification are declared in the \textit{CONSTANTS} clause and the predicate \textit{J} in the \textit{PROPERTIES} clause constrain those constants. The clause \textit{VARIABLES} contains a list of state variables in \textit{M} as mentioned above. The \textit{INVARIANT} clause gives the predicate \textit{I} which defines an invariance on the system. It must declare some constraint on each state variable. The \textit{INITIALISATION} \textit{T} defines the assignments to set the initial states of \textit{M}. An \textit{OPERATION} takes the form \textit{PRE R THEN S END} and enables us to reach subsequent states. We will examine those further in Section 2.4.
MACHINE Clock
VARIABLES nn
INVARIANT nn = 1 ∨ nn = 2

INITIALISATION nn := 1
OPERATIONS
  tick ≡ PRE nn = 1 THEN nn := 2 END ;
  tock ≡ PRE nn = 2 THEN nn := 1 END
END

Figure 2.2: The Clock machine.

Definition 2.4.1.

\[ S := \begin{array}{l}
  \text{skip} | \\
  x := E | \\
  S_1 || S_2 | \\
  S_1; S_2 | \\
  \text{IF } R \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END} | \\
  \text{CHOICE } S_1 \text{ OR } S_2 \text{ END} | \\
  \text{PRE } R \text{ THEN } S \text{ END} | \\
  \text{SELECT } R \text{ THEN } S \text{ END} | \\
  \text{ANY } x \text{ WHERE } R \text{ THEN } S \text{ END}
\end{array} \]

The expression \( x := E \) is called an assignment statement and denotes that \( x \) becomes identified with the value of \( E \) (e.g. \( x := 0 \) and \( x := y + 2 \)). The construct \( S_1 || S_2 \) is a concurrent execution of commands \( S_1 \) and \( S_2 \). Both \( S_1 \) and \( S_2 \) must not overlap on the variables which are changed (i.e. if \( S_1 \equiv x := E \) and \( S_2 \equiv y := F \) it is always the case that \( x \) is not the same as \( y \)). The \textsc{if} and \textsc{choice} statements specify branching in execution. A \textsc{select} statement introduces a guard \( R \). The difference between a precondition and a guard is that the guard blocks the execution of the statement if the machine state before the execution does not satisfy it. In contrast a statement with a precondition can be executed from any state. We revisit these differences later. An \textsc{any} statement introduces a local variable \( x \) with properties described in \( R \).

As mentioned above an AMN statement captures a state transition of a machine. There are two way to define a denotation for state transitions: by a relation between machine states, and by a predicate transformer. In this thesis we think of a machine state as a valuation \textsc{val} that satisfies a certain predicate. With that in mind in Figure 2.3 we have shown how the relational model and the predicate transformer model relate to each other. In the simplest case, executing an AMN statement \( S \) from a machine state \textsc{valbefore} reaches a new machine state \textsc{valafter} which may give new values to the variables. In a relational denotation of \( S \), these states are related. A predicate transformer...
denotation relates predicates $R_{\text{before}}$ and $R_{\text{after}}$ if executing from a machine state which satisfies $R_{\text{before}}$ we can reach a machine state which satisfies $R_{\text{after}}$. In that sense the two approaches are equivalent to each other but it is found that the latter gives rise to simpler definitions.

The predicate transformer mechanism used in the case of the B-Method is Dijkstra's weakest-precondition calculus [12].

**Definition 2.4.2. Weakest Precondition**

Given a B predicate $R_a$ and AMN statement $S$,

$$[S]R_a$$

is the least B predicate $R_b$ which must be true of the state before the execution of $S$ for $S$ to terminate and establish a state where $R_a$ is true.

The notation $[S]P$ is equivalent to $wp(S, P)$ for any $S$. We shall use the latter because we already used square brackets for joining separated state in Section 2.2.

We present the predicate transformer definitions which are inductive over the syntax.

**Definition 2.4.3. Assuming** $wp(S_1, R_a)$ and $wp(S_2, R_a)$ are defined for predicate
$R_a$, and AMN expressions $S_1$, $S_2$,

\[
\begin{align*}
wp(\text{skip}, R_a) &= R_a \\
wp(x := E, R_a) &= R_a\{E/x\} \\
wp(x := E \parallel y := F, R_a) &= wp(x, y := E, F, R_a) \\
wp(S_1; S_2, R_a) &= wp(S_1, wp(S_2, R_a)) \\
wp(\text{IF } R_c \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END, } R_a) &= (R_c \Rightarrow wp(S_1, R_a)) \land \neg(R_c) \Rightarrow wp(S_2, R_a)) \\
wp(\text{CHOICE } S_1 \text{ OR } S_2 \text{ END, } R_a) &= wp(S_1, R_a) \land wp(S_2, R_a) \\
wp(\text{PRE } R_c \text{ THEN } S_1 \text{ END, } R_a) &= R_c \land wp(S_1, R_a) \\
wp(\text{SELECT } R_c \text{ THEN } S_1 \text{ END, } R_a) &= R_c \Rightarrow wp(S_1, R_a) \\
wp(\text{ANY } x \text{ WHERE } R_c \text{ THEN } S_1 \text{ END, } R_a) &= \forall(x). (R_c \Rightarrow wp(S_1, R_a))
\end{align*}
\]

In the definition of $wp(x := E, R_a)$, the notation $R_a\{E/x\}$ denotes the substitution of every free occurrence of $x$ with $E$ in the predicate $R_a$. Furthermore, observe that weakest precondition is defined above for parallel composition of simple assignments. In the general case, it is defined as a reduction on two substitutions in normalised form as discussed in [54].

We illustrate the workings of Definition 2.4.3 with the aid of some examples below.

**Example 2.4.1.** This example calculates the initial condition on $nn$ and $mm$ so that the parallel assignment establishes the postcondition $nn > mm$,

\[
\begin{align*}
wp(nn := nn + 1 \parallel mm := mm + 1, nn > mm) \\
&= wp(nn, mm := nn + 1, mm + 1, nn > mm) \\
&= (nn > mm)\{nn+1, mm+1/nn, mm\} \\
&= (nn + 1 > mm + 1) \\
&= nn > mm
\end{align*}
\]

**Example 2.4.2.** This example shows an AMN command which can reach the desired postcondition $nn > 1$ only if it is executed from a state which does not satisfy the precondition (i.e. the command is wrong with respect to the desired postcondition and therefore a correct execution of the command cannot establish the postcondition),

\[
\begin{align*}
wp(\text{PRE } nn = 0 \text{ THEN ANY } xx \text{ WHERE } xx \in \mathbb{N} \text{ THEN } nn := x \text{ END END, } nn > 1) \\
&= (nn = 0) \land wp(ANY xx \text{ WHERE } xx \in \mathbb{N} \text{ THEN } nn := x \text{ END END, } nn > 1) \\
&= nn = 0 \land \forall xx. (xx \in \mathbb{N} \Rightarrow wp(nn := xx, nn > 1)) \\
&= nn = 0 \land \forall xx. (xx \in \mathbb{N} \Rightarrow nn > 1\{xx/nn\}) \\
&= nn = 0 \land \forall xx. (xx \in \mathbb{N} \Rightarrow xx > 1) \\
&= nn = 0 \land false \\
&= false
\end{align*}
\]
As in [12], the weakest precondition operator is monotonic with respect to implication and distributive with respect to conjunction of predicates.

**Definition 2.4.4.** For any AMN statement $S$,

1. $\text{wp}(S, R_1 \Rightarrow R_2) = \text{wp}(S, R_1) \Rightarrow \text{wp}(S, R_2)$
2. $\text{wp}(S, R_1 \land R_2) = \text{wp}(S, R_1) \land \text{wp}(S, R_2)$

**Definition 2.4.5.** Two AMN commands $S_1$ and $S_2$ are equivalent if, for all predicates $R$, $\text{wp}(S_1, R) = \text{wp}(S_2, R)$.

The following special conditions are defined for AMN commands [1].

**Definition 2.4.6.** Aborting and termination

1. $\text{abt}(S) = \forall R. \neg \text{wp}(S, R)$
2. $\text{trm}(S) = \neg \text{abt}(S)$

An execution of $S$ from a machine state that satisfies $\text{abt}(S)$ cannot establish any final predicate $R$. Executing from such a machine state is called divergence. The machine states that satisfy the predicate $\text{trm}(S)$ are those for which an execution of $S$ will terminate and establish a machine state which satisfies $R$.

**Example 2.4.3.** A classic example of an aborting command is the following,

$$\text{trm}(\text{PRE false THEN skip END}) = \text{false} \land \text{wp}(\text{skip}, R) = \text{false}$$

for any $R$.

Note that,

$$\text{abt}(\text{PRE false THEN skip END}) = \text{true}$$

Due to the fact that the precondition of this statement is deliberately false, $\text{abt}(S)$ tells us that we can execute this statement from any machine state and not terminate.

Another special predicate we consider here is a mechanism for linking before and after machine states [1].

**Definition 2.4.7.** Given AMN statement $S$,

$$\text{prdx}(S) = \neg \text{wp}(S, \bar{x} \neq \bar{x}')$$

where $\bar{x}$ and $\bar{x}'$ are two distinct lists of variable names of identical length. The list $\bar{x}$ is the variable names on which $S$ is defined and $\bar{x}'$ is a fresh list.
Following classical convention, in \( \text{prd}_x(S) \) the list \( x \) hold the before-values and \( x' \) hold the after-values. Thus, the predicate \( \text{prd} \) is the weakest precondition such that it is not the case that after the execution of the statement we can distinguish \( x \) and \( x' \).

**Example 2.4.4.** For example, suppose that an operation is defined as follows,

\[ \text{operation} = \text{PRE } 0 < \text{nn} < 5 \text{ THEN } \text{nn} := \text{nn} + 1 \text{ END} \]

Assume that the machine has just one variable \( \text{nn} \).

\[
\begin{align*}
\text{prd}_{\text{nn}}(\text{operation}) &= \neg \text{wp}(\text{PRE } 0 < \text{nn} < 5 \text{ THEN } \text{nn} := \text{nn} + 1 \text{ END, } \text{nn}' \neq \text{nn}) \\
&= \neg(0 < \text{nn} < 5 \land \text{wp}(\text{nn} := \text{nn} + 1, \text{nn}' \neq \text{nn})) \\
&= \neg(0 < \text{nn} < 5) \lor \neg \text{wp}(\text{nn} := \text{nn} + 1, \text{nn}' \neq \text{nn}) \\
&= (0 < \text{nn} < 5) \Rightarrow \text{prd}_{\text{nn}}(\text{nn} := \text{nn} + 1) \\
&= (0 < \text{nn} < 5) \Rightarrow (\text{nn}' = \text{nn} + 1)
\end{align*}
\]

The predicate \( (0 < \text{nn} < 5) \Rightarrow (\text{nn}' = \text{nn} + 1) \) above formalises the relationship between the before and after value of \( \text{nn} \) following the execution of the operation.

**Example 2.4.5.** Suppose we extend Example 2.4.4 so that the machine has two variables \( \text{nn} \) as before and a new variable \( \text{mm} \),

\[
\begin{align*}
\text{prd}_{\text{nn},\text{mm}}(\text{operation}) &= (0 < \text{nn} < 5) \Rightarrow (\text{nn}' = \text{nn} + 1 \land \text{mm}' = \text{mm})
\end{align*}
\]

Essentially, after the execution of operation, \( \text{mm} \) remains unchanged.

As we will be considering AMN statements within the context of a given machine \( M \) the example above shows that we can safely extend the frame \( x \) in \( \text{prd}_x(S) \) to be the list of variables from the clause \text{VARIABLES} of \( M \) for any statement \( S \) in \( M \). Thus, in following sections we write \( \text{prd}_{\text{VARIABLES}}(S) \) when we consider \( S \) in the context of some machine \( M \) with a list of \text{VARIABLES}.

### 2.5 Some useful properties

In this section we present properties of AMN statements from [1]. They will be important in proving the results of Chapter 4 and 5.

Observe that Definition 2.4.6 is second order. A more convenient result is the following.

**Theorem 2.5.1.** For any AMN statement \( S \),

1. \( \text{abt}(S) \iff \neg \text{wp}(S, \text{true}) \)
2. \( \text{trm}(S) \iff \text{wp}(S, \text{true}) \)
2.6. Predicate type system

Deriving the weakest precondition of a sequential composition statement has the effect of nesting weakest preconditions together as can be seen in Definition 2.4.3. The following results consider the predicates \( \text{trm} \) and \( \text{prd} \) of sequential compositions in terms of the individual \( \text{trm} \) and \( \text{prd} \) predicates.

**Theorem 2.5.2.** Given a AMN statements \( S_1 \) and \( S_2 \).

1. 

\[
\text{trm}(S_1; S_2) \iff (\text{trm}(S_1) \land \\
\forall \bar{x}'. (\text{prd}_{S_1}(S_1) \Rightarrow \text{wp}(\bar{x} := \bar{x}', \text{trm}(S_2))))
\]

2. 

\[
\text{prd}_{S_1}(S_1; S_2) \iff (\text{trm}(S_1) \Rightarrow \\
\exists \bar{x}''. (\text{wp}(\bar{x}' := \bar{x}'', \text{prd}_{S_1}(S_1)) \land \text{wp}(\bar{x} := \bar{x}'', \text{prd}_{S_1}(S_2))))
\]

In the statements above the lists \( \bar{x} \), \( \bar{x}' \), and \( \bar{x}'' \) are all lists of variable names. These lists are pairwise distinct, and of identical length. In each case \( \bar{x}' \) represents the extra free names in the predicate \( \text{prd}_{S}(S) \) for each \( S \) respectively.

### 2.6 Predicate type system

The B-Method includes a type system with which predicates can be typed so that we are not required to prove meaningless expressions. In this thesis we adopt the standard typing system for \( B \) defined in [1] and consider only well-typed machines.

In the verification framework presented in Chapter 6 we will introduce predicates for verification purposes which can refer to variables, constants, and enumerated values, from the machines in our specifications. We say a predicate \( R \) belongs to machine \( M \) if all the free variables in \( R \) are variables of \( M \) and \( R \) is well-typed with respect to \( M \)'s signature.

Informally, the signature of a machine \( M \), \( \text{SIG}(M) \), is a typing environment that consists of the following:

- the **PARAMETERS** of \( M \) with their type,
- the given **SETS** of \( M \) and the types of enumerated constants with their type,
- the **CONSTANTS** with their type,
- the **VARIABLES** with their type,
- the signature of each **operation** from **OPERATIONS**.
The signature of an operation is a construct that corresponds to the name of the operation. In this thesis we only consider operations without input and output parameters.

As an example, recall the Clock machine from Figure 2.2 and the predicate $nn = 1$. Clearly, $nn = 1$ belongs to Clock since $nn$ is a variable of Clock and the predicates are well typed with respect to the signature of the machine.
Chapter 3

Theoretical foundations of the π-calculus

This chapter presents an overview of the syntax of the π-calculus. All the definitions can be found in [52] but we annotate them with examples here. Firstly we describe the process syntax. Secondly we present the operational semantics followed by a description of processes and their traces. Finally we introduce the type-system.

3.1 Process syntax

The π-calculus syntax is defined on an infinite set of names \( \mathcal{N} \). These names have no internal structure apart from what is required to distinguish one from another [15]. Their purpose is to denote the interaction points of process agents. We use small case letters \( z, q, b, a, c, d, w, v \ldots \) or words such as channel\(_i\) where \( i \) is an appropriate index to represent members of \( \mathcal{N} \).

Capital letters such as \( P \) and \( Q \) denote processes. (Sometimes we also use capitalised words such as PROCESS.) A process is sometimes referred to as an agent or a component. Agents are constructed from the following syntax.

Definition 3.1.1.

\[
\begin{align*}
P &::= Q \mid (P_1 \mid P_2) \mid (v \circ P) \mid !P \\
Q &::= 0 \mid \pi.P \mid (Q_1 + Q_2) \\
\pi &::= a(w) \\
&\quad \mid \overline{a}(w) \\
&\quad \mid \tau \\
&\quad \mid [w_1 = w_2]\pi
\end{align*}
\]

23
The process \((0)\) cannot perform any action. The process \(\pi.P\) is a process which performs the action corresponding to a prefix \(\pi\) prior to becoming process \(P\). Prefixes can be an input action \(a(w)\), output action \(\overline{a}(w)\), internal action \(\tau\), and matching prefix \([w_1 = w_2]_\pi\). The latter most prefix allows action \(\pi\) to execute only when \(w_1\) and \(w_2\) are identical objects in \(N\). The process \(Q_1 + Q_2\) is a choice between executing either \(Q_1\) or \(Q_2\) and the choice is resolved depending on which action from \(Q_1\) or \(Q_2\) is executed first. Sometimes \(Q_1 + Q_2\) is also called a summation and \(\Sigma_{i\in C} Q_i\) is used to denote indexed choice over the finite set \(C\). The process \((P_1 | P_2)\) is the concurrent execution of processes \(P_1\) and \(P_2\). \((\nu v)(P)\) creates a new name \(v\) with scope \(P\). Finally, \((\nu P)\) is an infinite number of \(P\)'s running concurrently.

### 3.1.1 Binding, \(\alpha\)-conversion

In this section we provide definitions for identifying free and bound names, \(\alpha\)-conversion and we make a note about operator precedence.

**Definition 3.1.2. Free and Bound names,**

A name \(z\) in \(\pi\)-calculus is said to be bound if it lies within an occurrence of \(a(z)\) or \((\nu z)\). The syntax \(bn(P)\) denotes the set of bound names of process \(P\) and \(fn(P)\) denotes all names in \(P\) that are not in \(bn(P)\).

We assume that all processes use the standard naming convention which states that all bound names of a process \(P\) must be chosen to be distinct, from all free names of \(P\), and all names in a substitution.

A substitution is expressed with \(\{t/b\}\), where \(t\) and \(b\) are names from \(N\), and denotes a function mapping \(b\) onto \(t\) and every other name from \(N\) not equal to \(b\) onto itself. A substitution can also be expressed as \(\{a_1, \ldots, a_n/b_1, \ldots, b_n\}\) where \(b_1, \ldots, b_n\) are distinct and denotes the functional composition of each \(\{u/b_i\}\) for \(1 \leq i \leq n\). The application of substitution \(\{t/b\}\) to a process \(P\), written \(P\{t/b\}\), denotes the substitution of every free occurrence of \(b\) in \(P\) with \(t\). A formal definition can be found in [52].

**Example 3.1.1.** Consider the following processes which do not meet the naming convention,

\[
x(a).y(a).P \\
\overline{a}(m).b(a).P \\
(b(a).P)\{w/a\} \\
(b(a).P)\{a/w\}
\]

In the first process the name \(a\) bound by the \(x\) channel is different from the name \(a\) bound by the \(y\) channel. If \(a \in fn(P)\), we say that it is bound by the inner most binding mechanism which in this case is \(y(a)\) as in other programming languages.

We have a similar situation in the process \(\overline{a}(m).b(a).P\). The name \(a\) in \(\overline{a}(m)\) is different from the name \(a\) in \(b(a)\).
3.1. Process syntax

In the third example the name a in b(a) is different from the name a used in the substitution.

In the last example the name a in b(a) is again different from the name a used in the substitution.

Process expressions such as those in Example 3.1.1 above are valid expressions but they are not useful and very confusing. As with other calculi substitutions are employed when a process does not meet the naming convention so that it is re-written to a process that does. This is called α-conversion and defined in Definition 3.1.3 below. The principal difference with other notations is that, in the π-calculus, α-conversion has to be employed actively rather than just once. This is because expressions such as (b(a).P){"a/a} and (b(a).P){a/w} can arise as result of an input action as we shall see later in Section 3.2. When such circumstances arise, the practise is to use α-conversion to change the bound occurrence of the name a to some name that does not clash before applying the substitution. This activity is common during transition derivations.

The rules for α-conversion are given below and we use the operator $P =_\pi Q$ to denote it.

**Definition 3.1.3. α-Conversion**

1. If the name q does not occur in P then $P\{q/w\}$ is the process obtained by replacing all free occurrences of w in P with q,

2. A change of bound names in a process P is the replacement of a subterm $x(w_1).Q$ of P by $x(w_2).(Q\{w_2/w_1\})$, or the replacement of a subterm $(\nu v_1)Q$ of P by $(\nu v_2)(Q\{v_2/v_1\})$ where in each case $w_2$ or $v_2$ does not occur in Q,

3. A process P is α-convertible to Q, $P =_\pi Q$ if Q can be obtained from P by a finite number of changes of bound names.

In the definition above note the precedence of the operators in the second clause. We follow the convention that a substitution binds more tightly than all process operators. Hence, the substitutions are applied to Q alone after the decision to swap the $w_1$ for $w_2$ in the $a(w_1)$ prefix or $v_1$ for $v_2$ in the $(\nu v_1)$ construct.

**Example 3.1.2.** The process above is α-converted in the following way:

$x(a).b(a).P =_\pi x(a).b(c).(P\{c/a\})$ where $c \notin fn(P) \cup bn(P)$

We also adopt the standard convention that prefixing, and replication bind more tightly than parallel composition, and prefixing more tightly than sum. Note the following examples:

**Example 3.1.3.**

\[
\begin{align*}
\pi.P & | Q \text{ is } (\pi.P) | Q \\
!P & | Q \text{ is } (!P) | Q \\
\pi.P & + Q \text{ is } (\pi.P) + Q
\end{align*}
\]
Chapter 3. Theoretical foundations of the $\pi$-calculus

3.1.2 Examples of $\pi$-calculus agents

The first important feature of the behaviour of $\pi$-calculus process agents is the ability to generate new free names at runtime. New names are distinct from all pre-existing free names of a given process hence they are also called fresh. The second important feature is that an agent can communicate names to other agents thus extending their scope and its opportunity for interaction.

Consider the following examples which show the behaviour of some $\pi$-calculus processes. The first one aims to highlight the special behaviour attributed to $\pi$-agents known as bound output. The second one focuses on the behaviour of an agent that is attempting to input. The last one examines the different interactions when concurrent execution is possible.

Example 3.1.4. The process,

\[(\nu \ b)(\overline{a}(b).P)\]

can output the name $b$ on channel $a$ and then continue as $P$. The name $b$ in the interaction is fresh and $b \neq a$. After the output on $a$ the scoping mechanism $(\nu \ b)$ is removed hence if $b \in fn(P)$ to an observer it appears that the process has gained a new channel for initiating communication as shown in Figure 3.1. Observe that by using the rules for $\alpha$-renaming we can rename the $b$ to a $c$ such that $c \notin fn((\nu \ b)(\overline{a}(b).P))$. We would then have the process,

\[(\nu \ c)(\overline{a}(c).P\{c/b\})\]

which would then output the channel $c$, $c \neq a$. However the intended behaviour captured by the continuation process $P\{c/b\}$ does not change from that of $P$ because in both cases the new name was unknown to the environment prior the interaction.

Example 3.1.5. The process,

\[a(b).P\]
can input a name over channel $a$. Suppose a name $v$ arrives on channel $a$ then the continuation process is $P\{v/b\}$. Thus, the behaviour of $P$ is changed by forcefully renaming each name $b$ in $P$ to channel $v$. If the process $a(b).P$ meets the naming convention, one can consider the sets $fn(a(b).P)$, $bn(a(b).P)$, and $(\bigcap \{fn(a(b).P) \cup bn(a(b).P)\})$ as disjoint. The channel $v$ could have been chosen from any of those three sets. While the latter is not surprising the former two can be a source of confusion.

Suppose, $v \in fn(a(b).P)$ and $v \in fn(P)$ then the continuation process $P\{v/b\}$ after the input action, is simply a process where the channel $b$ is also identified with $v$ in addition to the $v$ already in $P$. It is even possible to receive the channel $a$ on channel $a$ thus the continuation process becomes $P\{a/b\}$.

If $v \in bn(a(b).P)$ then we can $\alpha$-convert the bound name occurrence of $v$ in $a(b).P$ before proceeding because this does not change the intended behaviour of the process. For example if $b = v$, firstly we change the process to $a(w).P\{w/b\}$ for any $w \notin \text{names}(a(b).P)$ and then we apply the input action so the continuation process is $(P\{w/b\})\{b/w\}$.

**Example 3.1.6.** Consider the following process,

$$a(x).P | (\nu b)(a(b).Q)$$

where $b \in fn(Q)$

As discussed in Example 3.1.4 a name $b$ can be outputted on channel $a$ in the agent $(\nu b)(a(b).Q)$. Also the process $a(x).P$ can input a name on channel $a$ as discussed in Example 3.1.5. In the concurrent combination above these capabilities for interaction are still available individually. However, since the name $a$ is free on both agents to the left and the right of the concurrency operator and one can input while the other can output. A further possibility for action is possible. An internal action which corresponds to the two agents communicating the $b$ between themselves.

To send $b$ via channel $a$, in the right agent $b$ is $\alpha$-converted so that it is different from all free names of $a(x).P$. Once a name is chosen then, in order to make sure the naming convention is met, we rename all bound names in $a(x).P$ so that they are different from the chosen name.

Suppose that in the processes above $\alpha$-renaming was not necessary and the name $b$ was already different from all free names and bound names of $a(x).P$. Then the scope of $b$ is extended so that it acts over both the left and right continuation agents $P\{b/x\} \mid Q$. Thus the process above evolves to $(\nu b)(P\{b/x\} \mid Q)$.

Subsequently, $P\{b/x\}$ can use $b$ as an interaction point with $Q$ in further execution. Note that this channel is private between $P\{b/x\}$ and $Q$ until either process decides to output the channel to the environment.

By this and similar communications $\pi$-calculus processes are said to change configurations and move at runtime.
3.2 Operational semantics

The most commonly used semantics for the π-calculus are operational. These are given in three relations between agents; the structural congruence relation, the reduction relation and a labelled transition relation. This section revisits each of them in turn.

3.2.1 Structural congruence

As with algebra, the π-calculus has an equivalence relation by which we can change the structure of a process without altering its behaviour. In the π-calculus, this relation is called structural congruence.

We need to define the following notions before structural congruence. The occurrence of a 0 is degenerate if it occurs to the left or right of a summation process \( Q_1 + Q_2 \), and non-degenerate otherwise. For example in the process \( 0 + \pi(z).0 \) the 0 to the left is degenerate and the 0 under the output prefix is non-degenerate.

A context is an expression built from Definition 3.1.1, where a non-degenerate occurrence of 0 is replaced with \([\bullet]\). For example \((\nu v)([\bullet] | Q)\) for some π-calculus process \( Q \) is a context. We denote the π-calculus process obtained from replacing the \([\bullet]\) with \( P \), \( C[P] \).

A congruence is an equivalence relation \( S \), such that for every process \( P \) and \( Q \), if \((P, Q) \in S\) then \((C[P], C[Q]) \in S\) for every context \( C \).

Structural congruence is defined as the smallest congruence \( \equiv_\pi \) that satisfies the following set of axioms.

Definition 3.2.1. Structural Congruence

1. Rules of equational reasoning

\[
\begin{align*}
\text{REFL} & : P \equiv_\pi P \\
\text{SYM} & : \text{if } P \equiv_\pi Q \text{ then } Q \equiv_\pi P \\
\text{TRANS} & : \text{if } P \equiv_\pi Q \text{ and } Q \equiv_\pi R \text{ then } P \equiv_\pi R \\
\text{CONG} & : \text{if } P \equiv_\pi Q \text{ then } C[P] \equiv_\pi C[Q] \\
\text{ALPHA} & : \text{if } P \equiv_\pi Q \text{ then } P \equiv_\pi Q
\end{align*}
\]
2. Axioms of structural congruence

\[
\begin{align*}
SC - MAT & \quad [x = x] \pi.P \equiv_{\pi} \pi.P \\
SC - SUM - ASSOC & \quad Q_1 + (Q_2 + Q_3) \equiv_{\pi} (Q_1 + Q_2) + Q_3 \\
SC - SUM - COMM & \quad Q_1 + Q_2 \equiv_{\pi} Q_2 + Q_1 \\
SC - SUM - INAC & \quad Q + 0 \equiv_{\pi} Q \\
SC - SUM - COMP & \quad (\nu w)(Q_1 + Q_2) \equiv_{\pi} Q_1 + (\nu w)Q_2 \quad \text{if } w \notin fn(Q_1) \\
SC - PAR - ASSOC & \quad P_1 \mid (P_2 \mid P_3) \equiv_{\pi} (P_1 \mid P_2) \mid P_3 \\
SC - PAR - COMM & \quad P_1 \mid P_2 \equiv_{\pi} P_2 \mid P_1 \\
SC - PAR - INAC & \quad P \mid 0 \equiv_{\pi} P \\
SC - RES & \quad (\nu w_1)(\nu w_2)P \equiv_{\pi} (\nu w_2)(\nu w_1)P \\
SC - RES - INACT & \quad (\nu w)0 \equiv_{\pi} 0 \\
SC - RES - COMP & \quad (\nu w)(P_1 \mid P_2) \equiv_{\pi} P_1 \mid (\nu w)P_2 \quad \text{if } w \notin fn(P_1) \\
SC - REP & \quad \mathit{\mid P} \equiv_{\pi} \mathit{\mid P}
\end{align*}
\]

A detailed explanation can be found in [52]. Here we give some examples of using structural congruence.

Example 3.2.1. Using structural congruence it can be shown that restricting a name not free in a process has no effect on changing the process behaviour. If we say that a name is not free in a process this means that the name might still appear as a bound name. However, if the need arises we can change the bound occurrence to another name using \( \alpha \)-conversion. Thus, essentially it means that the name does not exist in such a way that a process may use it to interact.

\[
z \notin fn(P) \implies (\nu z)(P) \equiv P \quad (\nu z)(P) \equiv (\nu z)(P \mid 0) \equiv P \mid (\nu z)(0) \equiv P \mid 0 \equiv P
\]

We illustrate the workings of structural congruence further by proving the following propositions. These propositions were set as exercises in [52] and are believed to have been proved elsewhere in the literature. We quote them here to illustrate a proof by structural induction using \( \equiv_{\pi} \).

Proposition 3.2.1. If \( P \equiv_{\pi} Q \) can be inferred without using the axiom SC - MAT then \( fn(P) = fn(Q) \).

Proof. By structural induction,

Base Cases: It is clear that in every axiom of Definition 3.2.1 for structural congruence except SC - MAT the free names of the agents to the left and right of the \( \equiv_{\pi} \) are equal. For example if \( P \equiv (\nu z)(0) \) and \( Q \equiv 0 \) then \( P \equiv_{\pi} Q \) and clearly \( fn(P) = fn(Q) \).

(This is not true for \( [x = x] \pi.P \equiv_{\pi} \pi.P \) because \( x \in fn([x = x] \pi.P) \) however it is not necessary that \( x \in fn(\pi.P) \) for some prefix \( \pi. \))
Chapter 3. Theoretical foundations of the $\pi$-calculus

Table 3.1: Reduction relation

Inductive Case: Suppose we have that there is an agent $D$ such that $P \equiv_\pi D$ and $fn(P) = fn(D)$.

Since $P \equiv_\pi Q$ is true then by TRANS of Definition 3.2.1 it must be the case that $D \equiv_\pi Q$. Then by similar reasoning as in the base cases we can conclude that $fn(D) = fn(Q)$ and from $fn(P) = fn(D)$ we have that $fn(P) = fn(Q)$. □

Proposition 3.2.2. Every process is structurally congruent to a process of the form,

$$(\nu x_1, \ldots, x_m)(P_1 | \ldots | P_n | !Q_1 | \ldots | \ldots | !Q_k)$$

for some natural numbers $m$, $n$ and $k$ also where each $P_i$ is a summation.

Proof. Any process $P$, we have that,

$$P \equiv_\pi (\nu a)((P + 0) | (0 + 0)) \quad \text{where} \ a \notin fn(P)$$

□

3.2.2 Reduction

The reduction relation shows how a system can evolve independently of its environment. The $\pi$-calculus reduction relation is a binary relation denoted by $P \rightarrow Q$ and indicates that process $P$ can evolve to $Q$ in a single autonomous step. The reduction relation is defined using a set of inference rules. We are using the rules outlined in Table 3.1. A proof of a reduction is called a reduction derivation (see Example 3.2.2) and it is a tree whose root is the reduction that is inferred. The leaves of the tree are instances of $R - \text{INTER}$ and $R - \text{TAU}$. 
3.2. Operational semantics

Note that the rule \( R \rightarrow \text{STRUCT} \) allows the application of structural congruence to change the process both before and after a reduction. It is thus the case that the reduction relation \( P \rightarrow Q \) is closed under structural congruence \( P \equiv Q \).

Consider the following example which considers a complex interaction and illustrates the rules of Table 3.1.

**Example 3.2.2.** Consider the process,

\[
P_1 = (\nu \ c)(c(a).P | ! (\nu \ z)(\bar{c}z.0 | Q))
\]

We want to show that,

\[
P_1 \rightarrow (\nu \ c)((\nu \ m)(P^{m/a} | Q^{m/z}) | ! (\nu \ z)(\bar{c}z.0 | Q))
\]

for some channel \( m \).

Applying structural congruence and \( \alpha \)-conversion we can show that,

\[
P_1 \equiv_\pi P_2
\]

for \( P_2 = (\nu \ c)(c(a).P + 0 | (\nu \ m)(\bar{c}m.0 + 0 | Q^{m/z} + 0) | ! (\nu \ z)(\bar{c}z.0 | Q)) \)

Hence, to show the above reduction we can use \( R \rightarrow \text{STRUCT} \) and must show that,

\[
P_2 \rightarrow (\nu \ c)((\nu \ m)(P^{m/a} | Q^{m/z}) | ! (\nu \ z)(\bar{c}z.0 | Q)) \quad (3.1)
\]

To show the above reduction we can use \( R \rightarrow \text{RES} \) hence, must show that,

\[
c(a).P + 0 | (\nu \ m)(\bar{c}m.0 + 0 | Q^{m/z} + 0) | ! (\nu \ z)(\bar{c}z.0 | Q) \rightarrow (\nu \ m)(P^{m/a} | Q^{m/z}) ! (\nu \ z)(\bar{c}z.0 | Q) \quad (3.2)
\]

Showing the above reduction we can use \( R \rightarrow \text{PAR} \),

\[
c(a).P + 0 | (\nu \ m)(\bar{c}m.0 + 0 | Q^{m/z} + 0) \rightarrow (\nu \ m)(P^{m/a} | Q^{m/z}) \quad (3.3)
\]

Since \( m \) was specially chosen so that its not free in the \( c(a).P + 0 \) portion, we can use \( R \rightarrow \text{STRUCT} \) to change the process with structural congruence so that the \( m \) is on the outside.

\[
(\nu \ m)(c(a).P + 0 | \bar{c}m.0 + 0 | Q^{m/z} + 0) \rightarrow (\nu \ m)(P^{m/a} | Q^{m/z}) \quad (3.4)
\]

Then we apply \( R \rightarrow \text{RES} \),

\[
c(a).P + 0 | \bar{c}m.0 + 0 | Q^{m/z} + 0 \rightarrow P^{m/a} | Q^{m/z} \quad (3.5)
\]

By \( R \rightarrow \text{Par} \),

\[
c(a).P + 0 | \bar{c}m.0 + 0 + 0 \rightarrow P^{m/a} \quad (3.6)
\]
Since we can derive by \( R - \text{INTER} \) that,
\[
\begin{align*}
  c(a).P + 0 & \mid \tau m.0 + 0 \rightarrow \\
  P[\tau^m/a] & \mid 0
\end{align*}
\]
we can use \( R - \text{STRUCT} \) to show the above reduction.

Hence, 3.2.2 is a valid reduction of \( P1 \).

3.2.3 Labelled transition semantics

The labelled transition relation shows how a process can interact with its environment. As the reduction relation, the labelled transition relation for the \( \pi \)-calculus is defined using a set of inference rules and form a binary, reflexive, and transitive relation between processes written, \( P \xrightarrow{\alpha} Q \).

The label \( \alpha \) on the arrow above is called an action. Actions are defined as follows.

**Definition 3.2.2. Action Labels**

\[
\alpha ::= a w \mid \overline{aw} \mid \overline{a}(v) \mid \tau
\]

The label \( a w \) is an input action and results from an input prefix. The \( \overline{aw} \) label is an output action and \( \overline{a}(v) \) is an output action of a bound name \( v \). Both actions result from an output prefix as we shall see below. Finally, \( \tau \) represents internal action which can result from \( \tau \) prefix or concurrent synchronisations. These action labels are not to be confused with the actual action prefixes from the syntax Section 3.1.

We use the terminology *subject* and *object* of a label to represent the channel link involved in communication and the name being transmitted, respectively (e.g. \( \text{subj}(a w) = a \) and \( \text{obj}(a w) = w \)). Note also that \( \text{bn}(\overline{aw}) = \emptyset, \text{fn}(\overline{aw}) = \{a, w\}, \text{bn}(a w) = \emptyset, \text{fn}(a w) = \{a, w\}, \text{bn}(\overline{a}(v)) = \{v\}, \) and \( \text{fn}(\overline{a}(v)) = \{a\} \).

The axioms for inferring labelled transitions are defined in Table 3.2.

This first set of transition rules deal with *output*, *input*, *internal action*, *matching* and *choice* in a way that corresponds to the informal explanations of the syntax from Section 3.1. The input transition generates a substitution in the continuation of the process. The match transition allows the process to continue only if the names to the left and right are identical. In a summation if the left process succeeds to interact the right is discarded. (Hence, there should be one more rule with the \( R \) extension that allows the right to interact first discarding the left. The same should be noted with the other rules.)
### 3.2. Operational semantics

<table>
<thead>
<tr>
<th>OUT</th>
<th>[ \overline{a}(w).P \xrightarrow{\bar{a}w} P ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>INP</td>
<td>[ a(w_1).P \xrightarrow{a w_2} P{w_2/w_1} ]</td>
</tr>
<tr>
<td>TAU</td>
<td>[ \tau.P \xrightarrow{\tau} P ]</td>
</tr>
<tr>
<td>MAT</td>
<td>[ \pi.P \xrightarrow{\pi} P' ]</td>
</tr>
<tr>
<td>SUM-L</td>
<td>[ P \xrightarrow{\sigma} P' ]</td>
</tr>
<tr>
<td>PAR-L</td>
<td>[ P \xrightarrow{\alpha} P' ]</td>
</tr>
<tr>
<td>COMM-L</td>
<td>[ P \xrightarrow{\pi w} P' \xrightarrow{\pi w} Q' ]</td>
</tr>
<tr>
<td>CLOSE-L</td>
<td>[ P \xrightarrow{\pi(v)} P' \xrightarrow{(\nu v)} Q' ]</td>
</tr>
<tr>
<td>RES</td>
<td>[ (\nu v)P \xrightarrow{\alpha} (\nu v)P' ]</td>
</tr>
<tr>
<td>OPEN</td>
<td>[ (\nu v)P \xrightarrow{\bar{a}(v)} P' ]</td>
</tr>
<tr>
<td>REP-ACT</td>
<td>[ P \xrightarrow{\alpha} P' ]</td>
</tr>
<tr>
<td>REP-COMM</td>
<td>[ P \xrightarrow{\alpha w} P' \xrightarrow{\alpha w} P'' ]</td>
</tr>
<tr>
<td>REP-CLOSE</td>
<td>[ P \xrightarrow{\alpha(v)} P' \xrightarrow{\alpha v} P'' ]</td>
</tr>
</tbody>
</table>

Table 3.2: Labelled Transitions
Rule PAR-L allows processes in parallel to progress independently. In the case where \( \alpha \) contains a bound name we must be careful that this name does not clash with the free names of \( P \) as usual but also of \( Q \) as this would infringe on the name's uniqueness from all other names in the combination \( P' \mid Q \).

Synchronisation of processes composed with parallel is achieved through the rules COMM-L and CLOSE-R. The latter deals specifically with the communication of a bound name. An important point to note about the \( \pi \)-calculus is that communication between concurrent agents is always binary and not multi-way as in CSP [24]. Note also that when two agents communicate over parallel a \( \tau \) transition is generated. The side condition of rule CLOSE-L requires that we check whether the name to be communicated is not in the free names of the receiving agent. We discussed this point with the aid of Example 3.1.6. Although not formally necessary it is often useful to rename all bound names in \( Q \) that might clash with the fresh name being communicated.

The rule \( RES \) is used to infer actions under the restriction operator. The notation \( \text{names}(\alpha) \) denotes the set of names used in the action. Thus the rule has the side condition that the restricted name \( \nu \) is not equal to the subject or object of \( \alpha \). If the former is true then there is no action. If the latter is true, then the applicable rule is OPEN instead.

\( OPEN \) in effect removes the restriction in the process expression making the name available to the environment. The side condition \( \nu \neq a \) is necessary to block transitions when \( \nu \) is used as the subject of the action.

Finally \( REP-ACT, REP-COMM, \) and \( REP-CLOSE \) are used to derive transitions of processes of replication. They capture that \( ![P] \) is an infinite parallel composition of \( P \). Notice that deriving transitions of \( ![P] \) we can either remove the \( ! \) using \( REP-ACT \) or consider how two \( P \)'s might communicate concurrently using \( REP-COMM \) and \( REP-CLOSE \).

The following example illustrates how Definition 3.1.1 relates to Definition 3.2.2.

**Example 3.2.3.** The following basic transition derivations are true by the rules of Table 3.2,

\[
\begin{align*}
&\Rightarrow a(w_1).P \stackrel{a(w_2)}{\longrightarrow} P\{w_2/w_1\} \quad \text{where } w_1 \neq w_2 \\
&\Rightarrow \overline{a}(w).P \stackrel{w}{\longrightarrow} P \\
&\Rightarrow \tau.P \xrightarrow{\tau} P \\
&\Rightarrow (\nu w)(\overline{a}(w).P) \stackrel{w}{\longrightarrow} P \quad \text{for some } w \in \mathcal{N}
\end{align*}
\]

The following example shows a more complex transition derivation where \( \alpha \)-conversion is required actively to ensure that continuation processes maintain the naming convention.
Example 3.2.4. Consider again the process,

\[ P_1 = (\nu c)(c(a).P \ | ! (\nu z)(\overline{c}.0 \ | Q)) \]

We want to show that,

\[ P_1 \xrightarrow{r} (\nu c)((\nu m)(P^{m/a} \ | Q^{m/z}) \ | ! (\nu z)(\overline{c}.0 \ | Q)) \]

for some \( m \in \mathbb{N} \).

To show the above transition derivation we use RES and need to show that,

\[
\begin{align*}
&c(a).P \ | ! (\nu z)(\overline{c}.0 \ | Q) \\
&\xrightarrow{r} \\
&(\nu m)(P^{m/a} \ | Q^{m/z}) \ | ! (\nu z)(\overline{c}.0 \ | Q)
\end{align*}
\]

To show the above transition derivation we must use CLOSE-L, instead of COMM-L (because the agent expressions do not pattern match) and must show that

\[ c(a).P \xrightarrow{\alpha} P^{m/a} \]

and

\[
\begin{align*}
&! (\nu z)(\overline{c}.0 \ | Q) \xrightarrow{c(m)} \\
&Q^{m/z} \ | ! (\nu z)(\overline{c}.0 \ | Q)
\end{align*}
\]

Notice that CLOSE-L forces us to choose an \( m \) that does not exist (is not free) in the \( c(a).P \) component. Note also that using \( \alpha \)-renaming, we can change the bound names in \( c(a).P \) to \( c(b).P^{b/a} \) so that we can, for example, infer a transition \( c(b).P^{b/a} \xrightarrow{c} P^{b/a} \) however this is normally done only to meet the naming convention. In most circumstances choosing a fresh name is sufficient.

The first conjunct of the labelled transition derivation can be shown by application of INP directly. We apply REP-ACT to the second conjunct and need to show the following.

\[ ! (\nu z)(\overline{c}.0 \ | Q) \xrightarrow{c(m)} Q^{m/z} \]

We apply OPEN and \( \alpha \)-rename the process to reach,

\[ \overline{c}m.0 \ | Q^{m/z} \xrightarrow{\overline{c}m} Q^{m/z} \]

To show the above transition derivation, it is necessary to apply PAR-L,

\[ \overline{c}m.0 \xrightarrow{\overline{c}m} 0 \]

which in turn is discharged through application of OUT.

Hence, the transition derivation 3.2.4 is valid.
3.2.4 Some useful results

The following is a collection of results about the free names of a process throughout its evolution.

**Lemma 3.2.1.** Suppose \( P \xrightarrow{\alpha} P' \),

1. if \( \alpha = \overline{w} \) then \( a, w \in \text{fn}(P) \) and \( \text{fn}(P') \subseteq \text{fn}(P) \),
2. if \( \alpha = a \ w \) then \( a \in \text{fn}(P) \) and \( \text{fn}(P') \subseteq \text{fn}(P) \cup \{w\} \),
3. if \( \alpha = \overline{a}(w) \) then \( a \in \text{fn}(P) \) and \( \text{fn}(P') \subseteq \text{fn}(P) \cup \{w\} \),
4. if \( \alpha = \tau \) then \( \text{fn}(P') \subseteq \text{fn}(P) \).

The following lemma shows the transition of a process under substitution. Note the relationship is only one way.

**Lemma 3.2.2.** If \( P \xrightarrow{\alpha} P' \) then \( P\sigma \xrightarrow{\alpha\sigma} P'\sigma \), provided if \( \alpha = \overline{a}(w) \) then \( w \notin \text{fn}(P\sigma) \cup \text{names}(\sigma) \).

The following result identifies the relationship between structural congruence and the labelled transition relation and between the reduction and the labelled transition relation is as follows.

**Lemma 3.2.3.** Harmony Lemma

1. if \( P \equiv_{\pi} Q \) then \( P \xrightarrow{\alpha} Q \),
2. \( P \longrightarrow P' \) iff \( P \equiv_{\pi} P' \).

where \( \equiv_{\pi} \xrightarrow{\alpha} \) denotes the relational composition of \( \equiv_{\pi} \) and \( \xrightarrow{\alpha} \).

It is thus possible to reason about processes using whichever of the reduction or \( \tau \) transition is more convenient, provided we work up to structural congruence.

The proofs to the lemmas presented in this section can be found in [52]

### 3.3 Reasoning about processes using traces

The traditional method for reasoning about process equivalence in the \( \pi \)-calculus is a bisimulation relation [35]. In this thesis however, we opt for a simpler notion of equivalence that is one based on traces. Trace equivalence equates more processes than bisimulation relations (hence it is less discriminating). It can be shown that any bisimulation relation is a subset of trace equivalence; this makes trace equivalence quite useful if the purpose is to show that two processes are not bisimilar. This is done by
establishing that the two processes are not trace equivalent which is mostly straight-
forward. We define a mechanism for identifying a set of traces for a given process \( P_1 \) below. Similar definitions can be found in [4].

Firstly, we write \( P_1 \xrightarrow{tr} P_n \) as a shorthand for:

**Definition 3.3.1.** \( P_1 \xrightarrow{tr} P_n \) denotes the following,

1. \( P_1 \xrightarrow{tr_0} P_1 \) for any \( P_1 \),
2. \( P_1 \xrightarrow{tr \alpha} P_{n-1} \xrightarrow{\alpha^{-1}} P_n \) iff there exists a \( P_{n-1} \) such that \( P_1 \xrightarrow{tr'} P_{n-1} \) and \( P_{n-1} \xrightarrow{\alpha^{-1}} P_n \).

Note that \( \{ \} \) denotes an empty sequence, \( tr \alpha \) denotes the sequence \( tr \) appended with \( \alpha \) and \( \alpha^{-1} tr \) denotes \( tr \) prepended with \( \alpha \).

The set of traces of a process is then defined as follows.

**Definition 3.3.2.** Traces

Given a \( \pi \)-calculus process \( P_1 \),

\[
\text{traces}(P_1) = \{ (\alpha_1, \ldots, \alpha_{n-1}) \mid \exists (P_n).P_1 \xrightarrow{\langle \alpha_1, \ldots, \alpha_{n-1} \rangle} P_n \}
\]

The syntax \( (\alpha_1, \ldots, \alpha_n) \) denotes a sequence of actions which we call a trace. Given a trace \( tr \), we write \( tr' \xleftarrow{\tau} tr \) if the trace \( tr' \) is a prefix of \( tr \) and \( tr' \xleftarrow{\tau} tr \) if the trace \( tr' \) is a proper prefix of \( tr \). We also define a function \( \text{weak} \) as follows.

**Definition 3.3.3.** Given a trace \( tr \)

\[
\text{weak}(tr) = \begin{cases} 
\{ \} & tr = \{ \}, \\
tr' & tr = \tau^- tr', \\
\alpha^- \text{weak}(tr') & tr = \alpha^- tr' \land \alpha \neq \tau.
\end{cases}
\]

The set \( \text{traces}(P) \) gives a set of sequences of actions that might include \( \tau \) actions. The set \( \text{weak}(\text{traces}(P)) \) gives a set of sequences of actions that does not contain \( \tau \) actions.

**Definition 3.3.4.** Trace Equivalence

1. Two processes \( P \) and \( Q \) are trace equivalent denoted by \( P \approx_{tr} Q \) if \( \text{weak}(\text{traces}(P)) = \text{weak}(\text{traces}(Q)) \).
2. Two processes \( P \) and \( Q \) are strong trace equivalent denoted by \( P \sim_{tr} Q \) if \( \text{traces}(P) = \text{traces}(Q) \).

### 3.4 Type system

In this section we focus on the type-system [52, p.263] that we refer to when considering combinations of \( \pi \)-calculus processes and B machines in Chapters 4 and 5.
### 3.4.1 Type grammar and process syntax

The definition of a type system is based on a collection of types. These are presented in Table 3.3. In this section we consider only one basic type \textit{unit}, which has only one.

<table>
<thead>
<tr>
<th>Base Types</th>
<th>( B_\pi := \text{unit} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Constructors</td>
<td>( V ::= B_\pi \mid L \mid [l_1-V_1, \ldots, l_n-V_n] )</td>
</tr>
<tr>
<td>Channel Types</td>
<td>( L ::= iV \mid oV \mid &amp;V )</td>
</tr>
<tr>
<td>Types</td>
<td>( S, T ::= L \mid V )</td>
</tr>
</tbody>
</table>

#### Table 3.3: Type Grammar

Element *\#. We adopt a type constructor called the variant type which we discuss in the next section. Channel types come in three flavours with which we can define a more precise capability on a given channel. The capabilities are \(i\) for input, \(o\) for output and \(\&\) for dual input and output. Note that Definition 3.3 makes no special distinction between values and channel types other than \textit{unit}. For example, suppose \(iT\) is a channel type where \(T\) is the type \(oT_1\). This means that \(iT\) is a channel type that can input a value which is a channel type that can output a value of type \(T_1\). Furthermore, there is no limit to the depth of \(\&\), \(i\), and \(o\) operators. For example consider the channel type \(io\&o\&T\). This is done to appeal to the \(\pi\)-calculus core feature; the ability to send any channel as a value over another channel (which may itself be a channel that can send a channel e.t.c.).

Using channel types, the process syntax of Definition 3.1.1 is decorated to represent the extra information of what the channels are supposed to represent. We only decorate the \(\nu\) operator as follows,

**Definition 3.4.1.**

1. Any process \(P\) is decorated if it does not contain the \(\nu\) operator,

2. Given a decorated process \(P\), \((\nu\ w : L)(P)\) is also decorated.

The notation \(w : L\) signifies that \(w\) is assigned the type \(L\). For clarity many approaches to typing also decorate the input prefix operator but this is not necessary for performing correct typing derivations as we shall see later.

### 3.4.2 The variant construct

One particular mechanism we use over and above the core \(\pi\)-calculus syntax is the variant construct [52, p.255]. Similar constructs have been proposed in the literature.
3.4. Type system

to study encodings of Object Oriented languages [50, 65, 66, 43].

A variant type is of the form \([l_1-V_1, \ldots, l_n-V_n]\) where each \(l_i\) from 1 to \(n\) is called a label and is unique. A value of this type has the structure \(l_i-v\) where \(v\) is a value of the type corresponding to \(V_i\). The ordering in the labels does not matter. For example if we had the natural numbers \(N\) as a base type it is possible to express a variant \([apples\_N, pears\_N]\) and the corresponding values would be,

\[[apples\_0, apples\_1, \ldots, pears\_0, pears\_1, \ldots]\]

Variant values are communicated to specialised processes called variant destructors. These act in a very similar manner to the \texttt{switch} command, common in imperative languages. We add variant destructors to the syntax of Definition 3.1.1.

Definition 3.4.2.

\[P ::= \]
\[| \text{Definition 3.1.1} \]
\[| \text{case } v \text{ of } [l_1-(w_1) \triangleright P_1; \ldots ; l_n-(w_n) \triangleright P_n] \]

Effectively by this mechanism we offer a choice of processes from \(P_1\) to \(P_n\). The \(w_i\) names might appear free in the corresponding \(P_i\) process. In the construct,

\[\text{case } v \text{ of } [l_1-(w_1) \triangleright P_1; \ldots ; l_n-(w_n) \triangleright P_n] \]

however, each \(w_i\) is a bound name. Thus this mechanism is a way of binding names similar to input prefix in Definition 3.1.2. The value of \(w_i\) is instantiated at the point of process selection as the transition semantics show below.

Definition 3.4.3. Labelled transitions of a variant

\[
\begin{array}{c}
\text{case } l_i-m \text{ of } [l_1-(w_1) \triangleright P_1; \ldots ; l_n-(w_n) \triangleright P_n] \\
\end{array}
\]

\[
\begin{array}{c}
\quad \longrightarrow^\tau P_i\{m/w_i\} \\
\end{array}
\]

Note above that the label \(l_i-m\) has been selected. The process creates a \(\tau\) action and the continuation is the process \(P_i\) with \(m\) substituted for \(w_i\).

Example 3.4.1. Consider the process,

\[P \equiv \text{case } v \text{ of } [\text{dash}-(a) \triangleright \text{out}(a).0; \text{dot}-(b) \triangleright (v \ n : \#\text{unit})(\text{out}(n)).0] \]

where \(v\) is of the type \([\text{dash} \#\text{unit}, \text{dot} \#\text{unit}]\).

The process \(P\{\text{dash}^m/v\}\) where \(m\) is a channel of type \#\text{unit} evolves to \(\text{out}(m).0\). The process \(P\{\text{dot}^m/v\}\) evolves to \((v \ n)\text{out}(n).0\).
3.4.3 Typing derivations

A typing system for the π-calculus is defined inductively using a set of inference rules. A process \( P \) is well typed if there is a valid typing derivation of process \( P \). A typing derivation has as its root the proposition \( \Gamma \vdash P \) and is represented as an inverted proof tree.

The \( \Gamma \) denotes a type environment which is a store where typing information is recorded about the constants and the free names of the process being typed. A typing assignment such as \( w : L \) can be thought of as a mapping \( (w, L) \) hence \( \Gamma \) denotes a finite set of such type assignments. We use the following notation with type environments. The notation \( \text{supp}(\Gamma) \) denotes the set of names on which \( \Gamma \) is defined (the domain of \( \Gamma \)). The notation \( \Gamma(w) = L \) denotes that the name \( w \) is assigned the type \( L \) in \( \Gamma \). Given two type environments \( \Gamma_1 \) and \( \Gamma_2 \), the syntax \( \Gamma_1, \Gamma_2 \) denotes their union where, \( \Gamma_2 \) is often a single type assignment as in \( \Gamma_1, w : L \). When such operations on environments are performed it is implicitly assumed that the names on which \( \Gamma_1 \) is defined is different from the set of names on which \( \Gamma_2 \) is defined.

Performing a typing derivation involves unfolding a process \( P \) using the inference rules below. At each step this activity might free up certain bound names whose typing information is added to the typing environment for the next step. Thus in general the first step involves a processes which has no free names. Given a \( P \) such that \( \text{fn}(P) = \emptyset \), \( P \) is well typed if \( \emptyset \vdash P \) where \( \emptyset \) denotes the empty type environment. Such \( P \)s are often called programs. In our thesis we might want to begin the typing derivation of a process at arbitrary steps. In such cases we must be careful to furnish every free name of the process that is to be typed with some type in the environment. Hence the following definition.

**Definition 3.4.4. Closed type environment**

A type environment \( \Gamma \) is closed with respect to a process \( P \) if for every \( w \in \text{fn}(P) \), \( \Gamma(w) = L \) for some channel type \( L \). A process \( P \) is closed if \( \Gamma \vdash P \) for some \( \Gamma \), closed with respect to \( P \).

We can check the proposition \( \Gamma \vdash P \) with the following rules. The type system outlined in Table 3.4 is one where link types are split into capabilities for input and output. The rules are fairly straightforward and we will investigate them with the use of an example below.

The main purpose for adding capabilities to types is that it enables the capture of requirements on the use of channels in a process. For example, it is often the case that we wish to enforce that a certain channel is used only in output subject position whichever process receives it. The capabilities for input, output and both input and output are linked by rules of subtyping. The operator for subtyping takes the form \( S \leq T \) which means that \( S \) is a sub-type of \( T \). Its definition is given in Table 3.5.
### 3.4. Type system

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T-NAME</strong></td>
<td>( \Gamma, w : T \vdash w : T )</td>
</tr>
<tr>
<td><strong>T-BASE-V</strong></td>
<td>( \Gamma \vdash v : B \ v \in B )</td>
</tr>
<tr>
<td><strong>T-INPS</strong></td>
<td>( \Gamma \vdash a : iS \ \Gamma, w : S \vdash P ) ( \Gamma \vdash a(w)P )</td>
</tr>
<tr>
<td><strong>T-OUTS</strong></td>
<td>( \Gamma \vdash a : oT \ \Gamma \vdash w : T \ \Gamma \vdash P ) ( \Gamma \vdash \overline{a}(w)P )</td>
</tr>
<tr>
<td><strong>SUBSUMPTION</strong></td>
<td>( \Gamma \vdash v : S \ S \leq T ) ( \Gamma \vdash v : T )</td>
</tr>
<tr>
<td><strong>T-PAR</strong></td>
<td>( \Gamma \vdash P \ \Gamma \vdash Q ) ( \Gamma \vdash P \parallel Q )</td>
</tr>
<tr>
<td><strong>T-SUM</strong></td>
<td>( \Gamma \vdash P \ \Gamma \vdash Q ) ( \Gamma \vdash P + Q )</td>
</tr>
<tr>
<td><strong>T-MAT</strong></td>
<td>( \Gamma \vdash w_1 : #T \ \Gamma \vdash w_2 : #T \ \Gamma \vdash P ) ( \Gamma \vdash [w_1 = w_2]P )</td>
</tr>
<tr>
<td><strong>T-NIL</strong></td>
<td>( \Gamma \vdash 0 )</td>
</tr>
<tr>
<td><strong>T-REP</strong></td>
<td>( \Gamma \vdash P ) ( \Gamma \vdash \overline{P} )</td>
</tr>
<tr>
<td><strong>T-RES</strong></td>
<td>( \Gamma, w : L \vdash P ) ( \Gamma \vdash (v \ w : L)(P) )</td>
</tr>
<tr>
<td><strong>T-TAU</strong></td>
<td>( \Gamma \vdash P ) ( \Gamma \vdash \tau.P )</td>
</tr>
<tr>
<td><strong>T-VAR</strong></td>
<td>( \Gamma \vdash v : [\l_1 - T_1, \ldots \ l_n - T_n] ) for each ( i \in 1..n \ \Gamma, x_i : T_i \vdash P_i ) ( \Gamma \vdash \text{case } v \text{ of } [\l_1(x_1) \rightarrow P_1; \ldots \ l_n(x_n) \rightarrow P_n] )</td>
</tr>
<tr>
<td><strong>T-VAR-V</strong></td>
<td>( \Gamma \vdash m : T ) ( \Gamma \vdash L.m : [\l.T] )</td>
</tr>
</tbody>
</table>

Table 3.4: Typing Rules
### Table 3.5: Subtyping rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB-REF</td>
<td>$T \leq T$</td>
</tr>
<tr>
<td>SUB-TRANS</td>
<td>$S \leq S' \quad S' \leq T$</td>
</tr>
<tr>
<td>SUB-$#I$</td>
<td>$#T \leq iT$</td>
</tr>
<tr>
<td>SUB-$#O$</td>
<td>$#T \leq oT$</td>
</tr>
<tr>
<td>SUB-II</td>
<td>$S \leq T \quad iT \leq iT$</td>
</tr>
<tr>
<td>SUB-OO</td>
<td>$T \leq S \quad oS \leq oT$</td>
</tr>
<tr>
<td>SUB-BB</td>
<td>$T \leq S \quad S \leq T \quad #S \leq #T$</td>
</tr>
</tbody>
</table>

for each $1 \leq i \leq n$ $S_i \leq T_i$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB-VAR</td>
<td>$[l_1-S_1, \ldots, l_n-S_n] \leq [l_1-T_1, \ldots, l_n-T_{n+m}]$</td>
</tr>
</tbody>
</table>
In Table 3.5, perhaps counter-intuitive is the fact that \( \|T \leq oT \) and \( \|T \leq iT \) by rules SUB-\{oo, \#\}. The type \( oT \) is a super-type of \( \|T \) because one can use a channel of type \( oT \) in any situation where a channel of type \( oT \) is required. The reverse is of course not true. The same goes for a channel of type \( iT \).

Another point to make here is that, as constructs, the type \( oT \) is contravariant while \( iT \) is covariant, and \( \|T \) is invariant by rules SUB-\{oo, II, BB\} respectively. This means that it is safe to output a link with less capability than what is required, for example \( o\|T \leq oIT \) is false but \( oIT \leq o\|T \) is true. It is also safe to input a link with more capability than what is required, for example \( i\|T \leq iiT \) is true but \( iiT \leq i\|T \) is false. In contrast only \( \|oT \leq \|oT \) and \( \|iT \leq \|iT \) are true.

**Example 3.4.2.** Consider the process,

\[
P \equiv (v \ z : \|V)(out(z).0 \ | \ z(v).case \ v \ of \ [once_-(x) \rightarrow \overline{Z}(x).0, \ twice_-(y) \rightarrow \overline{Y}(y).0])
\]

Consider a typing derivation of \( P \) where \( out : ooV \) and \( V = [once-o \ unit, \ twice-o \ unit] \).

Thus we need to show that \( \{out : ooV\} \vdash P \). By rule T-RES it must be the case that,

\[
\{out : ooV, \ z : \|V\} \vdash out(z).0 \ | \ z(v).case \ v \ of \ [once_-(x) \rightarrow \overline{Z}(x).0, \ twice_-(y) \rightarrow \overline{Y}(y).0]
\]

By rule T-PAR.

\[
\{out : ooV, \ z : \|V\} \vdash out(z).0 \quad \text{and} \quad \{out : ooV, \ z : \|V\} \vdash z(v).case \ v \ of \ [once_-(x) \rightarrow \overline{Z}(x).0, \ twice_-(y) \rightarrow \overline{Y}(y).0]
\]

(3.8)

Taking the first branch by rule T-OUTS.

\[
\{out : ooV, \ z : \|V\} \vdash out : o\|V \quad \text{and} \quad \{out : ooV, \ z : \|V\} \vdash z : \|V \quad \text{and} \quad \{out : ooV, \ z : \|V\} \vdash 0
\]

(3.9)

The second conjunct is clearly true. The third is true by T-NIL. We look at the first conjunct in more detail by using SUBSUMPTION.

\[
\{out : ooV, \ z : \|V\} \vdash out : ooV \quad \text{and} \quad ooV \leq o\|V
\]

(3.10)

Clearly \( ooV \leq o\|V \) is true because by rules SUB-oo and then SUB-\#I we have that \( \|V \leq oV \) and true respectively.

Backtracking to 3.8 we now check the second conjunct. By rule T-INPS we have that,

\[
\{out : ooV, \ z : \|V\} \vdash z : iV \quad \text{and} \quad \{out : ooV, \ z : \|V, \ v : V\} \vdash case \ v \ of \ [once_-(x) \rightarrow \overline{Z}(x).0, \ twice_-(y) \rightarrow \overline{Y}(y).0]
\]

(3.11)
Chapter 3. Theoretical foundations of the \( \pi \)-calculus

The first conjunct is true by SUBSUMPTION and then SUB-\( I \). By T-VAR we check the second conjunct,

\[
\{ \text{out} : \text{ooV}, \ x : \text{V}, \ v : \text{V} \} \vdash v : \{ \text{once}_0 \text{ unit}, \ \text{twice}_0 \text{ unit} \} \quad \text{and}
\{ \text{out} : \text{ooV}, \ z : \text{V}, \ v : \text{V}, \ x : \text{o unit} \} \vdash \text{\( \overline{\text{v}} \)}(*) \text{.0} \quad \text{and}
\{ \text{out} : \text{ooV}, \ z : \text{V}, \ v : \text{V}, \ y : \text{o unit} \} \vdash \text{\( \overline{\text{y}} \)}(*) \text{.0}
\]

(3.12)

The first conjunct is clearly true while the second and third conjunct are true through application of T-OUTS and in each case T-BASE and finally T-NIL.

Hence the process is correctly typed.

3.4.4 Typed labelled transition rules

It must be noted that the definition of labelled transition rules for the typed \( \pi \)-calculus is slightly different in presentation from that in Section 3.2.3. Mainly the bound output action \( \overline{a}(v) \) is replaced with the action \( (v \ v : T)\overline{a} \ v \) to show the type of the name being outputted. Thus the transition rules of Table 3.6 apply instead of the originals in Table 3.2. Clearly with these rules we cannot infer any new transitions than what

\[
\text{OPEN} \quad \frac{P \xrightarrow{\overline{a}v} P'}{(v \ v : T)P \xrightarrow{(v \ v : T)\overline{a} v} P'} \quad a \not= v
\]

\[
\text{CLOSE-L} \quad \frac{P \xrightarrow{(v \ v : T)\overline{a} v} P' \quad Q \xrightarrow{a v} Q'}{P \ | \ Q \xrightarrow{\tau} (v \ v : T)(P' \ | \ Q')} \quad \text{where} \quad v \cap \text{fn}(Q) = \emptyset
\]

\[
\text{REP-CLOSE} \quad \frac{P \xrightarrow{(v \ v : T)\overline{a} v} P' \quad P \xrightarrow{a v} P''}{!P \xrightarrow{\tau} ((v \ v : T)(P' \ | \ P'')) \ | \ !P)}
\]

Table 3.6: Typed Transition Rules

is possible with the original semantics in Table 3.2. We will use either \( \overline{a}(v) \) or \( (v \ v : T)\overline{a} \ v \) depending on whether it is important to emphasise the type of the name \( v \) in the bound output action.

3.4.5 Properties of a type system

The substitution lemma expressed below illustrates that the application of a substitution where the names involved have matching types in a process yields a well typed process.

**Lemma 3.4.1. Substitution Lemma**

If,

...
3.4. Type system

1. \( \Gamma \vdash P \),
2. \( \Gamma(a) = S \),
3. \( \Gamma \vdash v : S \) then

\( \Gamma \vdash P \{v/a\} \).

The subject reduction theorem shows that the typing rules are consistent with the operational semantics of \( \pi \)-processes. The theorem also illustrates the types of names used in a transition of the system. It takes the view that free output is a special case of bound output. Thus, in the last case the theorem applies when \( x = v \) or when \( x \neq v \) in which case the action is equivalent to free output \( \bar{a}v \).

**Theorem 3.4.1. Subject Reduction**

If \( \Gamma \vdash P \), and \( \Gamma \) is closed, and \( P \xrightarrow{\alpha} P' \) then,

1. if \( \alpha = \tau \) then \( \Gamma \vdash P' \),
2. if \( \alpha = a v \) then there is a \( T \) such that,
   
   (a) \( \Gamma \vdash a : iT \),
   
   (b) if \( \Gamma \vdash v : T \) then \( \Gamma \vdash P' \).
3. if \( \alpha = (\nu x : T)\bar{a}v \) then there is a \( T \) such that,
   
   (a) \( \Gamma \vdash a : oT \),
   
   (b) \( \Gamma, x : T \vdash v : T \),
   
   (c) \( \Gamma, x : T \vdash P' \).

The following results allow the removal or addition of names that are not free in the process which is being typed.

**Lemma 3.4.2. Strengthening**

If \( \Gamma, x : T \vdash P \) and \( x \notin \text{fn}(P) \) then \( \Gamma \vdash P \).

**Other useful results**

**Lemma 3.4.3. Weakening**

If \( \Gamma \vdash P \) then \( \Gamma, x : T \vdash P \) for any \( x \) such that \( x \notin \text{supp}(\Gamma) \).

The proofs to the results above can be found in [52].

Finally it must be noted that the definitions in [52] are a simpler version of [42] where a requirement is enforced that names with type \( oT \) are used only in output and \( iT \) only for input. The definitions presented here and in [52] do not support this requirement explicitly. We prefer to use [52] because we want to adhere to some standardisation of the syntax and because the book offers an integrated approach. It is believed that [52] can be extended to [42] however we do not investigate this here. Instead we compensate for the drawback by typing various parts of the systems we consider separately, and by explicitly mentioning the type of the name in the environment.
Example 3.4.3. If $\Gamma, z : oT \vdash P$ is valid then we know that $P$ uses $z$ to output only. If $\Gamma, z : iT \vdash Q$ is valid then we know that $Q$ uses $z$ to input only. Thus we also know that $\Gamma, z : \#T \vdash P \mid Q$ is valid.
Chapter 4

Linking the B-Method and \( \pi \)-calculus

This chapter defines an approach to model B machines without input and output as \( \pi \)-calculus style labelled transition systems. Firstly we develop a preliminary B-Machine labelled transition system that models operations as events. This stage is not dissimilar to what one would have to undertake in-order to combine B and CSP at labelled transition system level. We illustrate that there is a relationship between the labelled transition system we developed and the more familiar representations such as Morgan’s failure-divergence semantics [37].

We then extend the preliminary labelled transitions system to a \( \pi \)-style labelled transition system. We also define a set of typing rules which integrate with those presented in Chapter 3 so that a process containing a B-Machine can be typed.

4.1 Preliminary labelled transition system

A labelled transition system consists of a set of states, a set of labels, a set of elementary or initial states, and a relation connecting the states and labels. For a given \( M \) these are organised in a tuple as follows,

Definition 4.1.1.

\[
LTS_M = (ST_M^{\downarrow}, \text{OPERATIONS}, INIT_M, \rightarrow_M)
\]

In the sub-sections that follow, we proceed to give definitions towards each of the elements of Definition 4.1.1.

4.1.1 A set of states and a set of labels

The states which we will be using are the valuations \( \text{val} \) described in Section 2.2. The reason for being that explicit about states here, is that in following sections we will
be working in both the relational and predicate transformer models for B-Machines. Primarily, it gives us the ability to interpret a given machine predicate as a set of the valuations which satisfy it. Another reason is that it gives a clearer idea of what the state of a machine is when we integrate it into a $\pi$-calculus agent, as we shall see in Section 4.3.

Consider the INVARIANT predicate $I$ of $M$ which must declare (at least) the type of every variable in the list $\text{VARIABLES}$. Then we can define a global state space for our preliminary labelled transition system as,

**Definition 4.1.2. Global state-space of $M$**

Given the INVARIANT $I$ of $M$, the global state-space of $M$, $ST_M$ is the set of valuations $\text{val}$ such that, $\text{val}$ satisfies $I$.

Taking all $vals$ which satisfy $I$ might be considered too constraining, because $I$ might express relationships between $\text{VARIABLES}$ as well as typing information. For example, if we have a conjunct $nn > mm$ within the invariant then we would not have the valuation $\{(nn, 1), (mm, 1)\}$ within $ST_M$. In general such state could form part of the state-space that is reachable only after divergence but in our case we simply take all states that satisfy the invariant and one special state that does not satisfy any predicate including $I$. This special state, $\bot$ is added explicitly to the state-space as $ST_M$ in Section 4.1.3.

**Example 4.1.1. Consider the machine Clock 2.2 where the invariant states that $nn \in 1..2$ then,**

$$ST_{\text{Clock}} = \{(nn, 1), (nn, 2)\}$$

is the state space of this machine.

The set of labels will be the names of the operations of $M$ as they are already unique identifiers. In following sections we will treat the $\text{OPERATIONS}$ clause of a machine as a set.

### 4.1.2 A set of initial states

The $\text{INITIALISATION}$ clause of $M$ provides the AMN substitution $T$ which assigns the initial values. $T$ is constructed from a sub-class of AMN substitutions. Guard and precondition are excluded as there is no sensible pre-initial state. Furthermore, it assigns a value to every $x \in \text{set-of}(\text{VARIABLES})$ and cannot refer to past states (i.e. a variable must not appear on the right hand side of an assignment). This has the effect that the predicate $\text{prdVARIABLES}(T)$ contains only decorated versions of the original $\text{VARIABLES}$.

Before proceeding we introduce the following notation. Given a list of names $\text{list}$ we denote with $\text{set-of}(\text{list})$ the set where each member is an element of the list. Furthermore, let us postulate that for any $\text{set-of}(\text{VARIABLES})$ there is a set $\text{set-of}(\text{VARIABLES}')$
such that \( \text{set-of}(\text{VARIABLES}) \cap \text{set-of}(\text{VARIABLES}') = \emptyset \) and there is a natural bijective function \( \text{dash} : \text{set-of}(\text{VARIABLES}) \to \text{set-of}(\text{VARIABLES}') \). We designate \( \text{primed}(\text{val}) \) to denote the valuation \( \text{set-of}(\text{VARIABLES}') \to \mathcal{D}_B \) given by, \( \text{val} \circ \text{dash}^{-1} \).

The notation above is necessary because we would like to keep our state space consistent in the sense that all \( \text{vals} \) within it must be functions from \( \text{set-of}(\text{VARIABLES}) \) to \( \mathcal{D}_B \). When considering predicates like \( \text{prdVARIABLES}(T) \) however we need a mechanism for modifying the variables in a controlled manner.

**Example 4.1.2.** Suppose we have \( \text{val} = \{(x, 1)\} \), evaluating \( x = 2 \) is false. With \( \text{primed}(\text{val}) = \{(x, 1) \circ (x', x)\} = \{(x', 1)\} \) we can evaluate \( x' = 2 \) and this is also false.

We are now ready to define the set of initial states as follows,

**Definition 4.1.3.** \( \text{INIT}_M \) Given the \text{INITIALISATION} clause \( T \) of \( M \), the set of initial states \( \text{INIT}_M \) are those \( \text{val} \in \text{ST}_M \) for which there is a corresponding \( \text{primed}(\text{val}) \) which satisfies \( \text{prdVARIABLES}(T) \).

Definition 4.1.3 gives us the initial nodes. Note that we do not keep the \( \text{primed}(\text{val}) \) in \( \text{INIT}_M \) but its undecorated version \( \text{val} \).

**Example 4.1.3.** The initialisation statement for Clock states that \( \text{nn} := 1 \) thus,

\[
\text{INIT}_{\text{Clock}} = \\
\{ \text{val} \mid \text{val} \in \text{ST}_M \land \text{primed}(\text{val}) \text{ satisfies } \text{prdVARIABLES}(\text{nn} := 1) \}
\]

\[
= \{ \text{val} \mid \text{val} \in \text{ST}_M \land \text{primed}(\text{val}) \text{ satisfies } -(\text{nn} \neq 1) \}
\]

\[
= \{ (\text{nn}, 1) \}
\]

### 4.1.3 A labelled transition relation for machines without I/O

The labelled transition relation for a machine \( M \) will be denoted by,

\[
\overset{\to}{\text{M}} \in \text{ST}_M^{\downarrow} \times \text{OPERATIONS} \times \text{ST}_M^{\downarrow}
\]

where \( \text{ST}_M^{\downarrow} \) denotes \( \text{ST}_M \cup \{\bot\} \) and \( \bot \) is a divergent state.

Recall from Section 2.2 the notation that allows us to combine machine states. The expression \( [\text{val}_1, \text{val}_2] \) where \( \text{val}_1 \) and \( \text{val}_2 \) are valuations such that \( \text{dom}(\text{val}_1) \cap \text{dom}(\text{val}_2) = \emptyset \) denotes the common extension of \( \text{val}_1 \) and \( \text{val}_2 \). Thus we have that \( [\text{val}_1, \text{val}_2] \) is another valuation with which we can evaluate a predicate whose free names are within both \( \text{val}_1 \) and \( \text{val}_2 \).

**Example 4.1.4.** Suppose \( \text{val}1 = \{(\text{nn}, 1)\} \) and \( \text{val}2 = \{(\text{mm}, 2)\} \) then,

\[
[\text{val}1, \text{val}2] = \{(\text{nn}, 1), (\text{mm}, 2)\}
\]

thus we can evaluate,

\[
[\text{val}1, \text{val}2] \text{ satisfies } (\text{nn} > \text{mm})
\]
This notation is necessary because for an AMN expression \( S \), where \( S \) is not constrained like the initialisation statement \( T \) in the previous section, the predicate \( \text{prd\,VARIABLES}(S) \) might refer to a before-state. In other words it produces a predicate which needs to be interpreted with a valuation assigning a value to both \( \text{VARIABLES} \) and \( \text{VARIABLES}' \).

In the definitions that follow we will write \( \text{prd\,VARIABLES}(\text{operation}) \) and \( \text{abt(\text{operation})} \) in the place of \( \text{prd\,VARIABLES}(S) \) and \( \text{abt}(S) \) where \( S \) is the GSL command defining the operation \( \text{operation} \) within \( M \).

**Definition 4.1.4. B Machines without I/O**

Given \( M \), where \( \text{val}_1 \in \text{ST}_M \), \( \text{val}_2 \in \text{ST}_M \) and \( \text{op} \in \text{OPERATIONS} \),

1. \((\text{val}_1, \text{op}, \text{val}_2) \in \rightarrow_M \) if \([\text{val}_1, \text{primed(val}_2)] \) satisfies \( \text{prd\,VARIABLES(op)} \),
2. \((\text{val}_1, \text{op}, \bot) \in \rightarrow_M \) if \( \text{val}_1 \) satisfies \( \text{abt(op)} \),
3. \((\bot, \text{op}, \text{val}_2) \in \rightarrow_M \) for all \( \text{val}_2 \in \text{ST}_M^\bot \).

The first item captures divergent behaviours leading non-deterministically to any state in \( \text{ST}_M \) as well as good behaviours leading to valid states in \( \rightarrow_M \). This is an unavoidable effect of \( \text{prd\,VARIABLES}(S) \). Some divergences in the second item overlap with those of the first (i.e. they already exist) but they are not re-introduced by Definition 4.1.4. The second item simply adds \((\text{val}_1, \text{op}, \bot)\) for every \( \text{val}_1 \) which can cause divergence. The final item introduces \( \bot \) as a divergent state mapping it to anything in \( \text{ST}_M^\bot \) for any operation \( \text{op} \).

**Example 4.1.5.** Consider the operations of machine Clock called tick and tock and its labelled transition system in Figure 4.1. It can be shown that,

\[
\begin{align*}
\text{prd}_m(\text{tick}) & \triangleq (\text{nn} = 1 \Rightarrow \text{nn}' = 2) \\
\text{abt(\text{tick})} & \triangleq (\text{nn} \neq 1) \\
\text{prd}_m(\text{tock}) & \triangleq (\text{nn} = 2 \Rightarrow \text{nn}' = 1) \\
\text{abt(\text{tock})} & \triangleq (\text{nn} \neq 2)
\end{align*}
\]

Then by Definition 4.1.4 we have that,

\[
\begin{align*}
\rightarrow_{\text{Clock}} = \{&((\text{val}, \text{tick}, \text{val})) \cup \{((\text{val}, \text{tock}, \text{val})) \mid \text{val} \in \text{ST}_{\text{Clock}}\} \cup \\
&\{((\text{val}, \text{tick}, \bot)) \mid \text{val} \in \text{ST}_{\text{Clock}} \land \text{val} \neq \{(\text{nn}, 1)\}\} \cup \\
&\{((\text{val}, \text{tock}, \bot)) \mid \text{val} \in \text{ST}_{\text{Clock}} \land \text{val} \neq \{(\text{nn}, 2)\}\} \cup \\
&\{(\bot, \text{tick, val}) \mid \text{val} \in \text{ST}_{\text{Clock}}^\bot \} \cup \{(\bot, \text{tock, val}) \mid \text{val} \in \text{ST}_{\text{Clock}}^\bot \}
\end{align*}
\]

We require that \( \text{LTS}_M \) follows Morgan’s failure-divergence semantics for action systems. We confirm this in the next section.
4.2 Correspondence with Morgan’s failures-divergences semantics for action systems

This section assumes that we are provided with a machine $M$ that is internally consistent, i.e., its initial state satisfies the invariant and every operation in $M$ executed from a state in which its precondition is true re-establishes the invariant. We show that executing a sequence of operations from $LTS_M$ follows Morgan’s failures-divergences semantics for action systems.

4.2.1 Preliminaries

Firstly we formalise the notion of trace through $LTS_M$ similar to the way it is done for the π-calculus in Section 3.3.
We write \((val_1 \overset{tr}{\Rightarrow} val_n)\) as a shorthand for the following,

Definition 4.2.1.

1. \(val_1 \overset{0}{\Rightarrow} val\) for any \(val\),
2. \(val \overset{tr \sim (a_n)}{\Rightarrow} val_n\) iff \(\exists (val_{n-1}).(val \overset{tr}{\Rightarrow} val_{n-1} \land val_{n-1} \overset{a_n}{\rightarrow_M} val_n)\)

where \(a_n \in \text{OPERATIONS}\).

Then the set of traces through \(LTS_M\) is given by the function \(\text{traces}\) as follows,

Definition 4.2.2. \(LTS_M\) Traces

\[
\text{traces}(LTS_M) = \{(a_1, \ldots, a_n) \mid \exists (val_1 \in INIT_M, val_n \in ST^k_M).val_1 \overset{(a_1, \ldots, a_n)}{\Rightarrow} val_n\}
\]

We define the set of divergent traces in \(LTS_M\) as follows,

Definition 4.2.3. Divergences

\[
D(LTS_M) = \{tr \mid tr \in \text{traces}(LTS_M) \land \exists (val \in INIT_M).(val \overset{tr}{\rightarrow_M} \bot)\}
\]

Finally we relate a sequence of labels to a AMN statement for sequential composition as follows,

Definition 4.2.4. By induction on the length of \(tr\),

\[
\text{if } tr = \emptyset \text{ then } (tr)_b = \text{skip} \\
\text{if } tr = tr \sim \text{(operation)} \text{ then } (tr)_b = (tr)_b; \text{operation}
\]

4.2.2 Correspondence

Recall from Chapter 1, Morgan’s failures-divergences semantics provides definitions for identifying a trace of the system, what is meant by a divergent trace and a failure of the system. A trace \(tr\) of the system is one for which \(\neg \text{wp}((\text{init}) \sim tr, \text{false}) = \text{true}\) where \(\text{init}\) is an initialisation statement. The effect of this statement is to check if the guards of all individual commands within \(tr\) are met. A trace is divergent if \(\neg \text{wp}((\text{init}) \sim tr, \text{true}) = \text{true}\). The effect of this statement is to check if the preconditions of all individual commands within \(tr\) are met. We must show that every trace through the labelled transition system is also identified by Morgan’s definitions and those traces which end in a divergent state are identified as divergent. Since we restrict to commands whose guards are trivially \text{true} we do not need to consider the failures of system, and this is discussed in [56].

Firstly, note that by Definition 4.1.1 an operation is executable from any state.
Lemma 4.2.1. For any $val_1 \in ST^M_M$ there is a $val_2 \in ST^M_M$ such that,

$\quad val_1 \xrightarrow{op}_M val_2$

Proof. By Definition 4.1.1 either $val_1 = \bot$, or $val_1$ satisfies $abt(op)$, or $[val_1, \text{primed}(val_2)]$ satisfies $prd_{\text{VARIABLES}}(op)$.

If $[val_1, \text{primed}(val_2)]$ satisfies $prd_{\text{VARIABLES}}(op)$ then by Theorem 2.5.2.(2), $[val_1, \text{primed}(val_2)]$ must satisfy at least $trm(op)$. We have by definition of $trm$ that $trm(op) = \neg abt(op)$ hence we have that $val_1 = \bot$, or $val_1$ satisfies $abt(op)$, or $val_1$ satisfies $\rightarrow abt(op)$.

In the case $val_1 = \bot$ we have by Definition 4.1.1 that $(\bot, op, val_2) \in \rightarrow_M$ for all $val_2 \in ST^M_M$. Hence we have the case. □

The first requirement is that all traces through $LTS_M$ are valid,

Theorem 4.2.1. The traces of $LTS_M$

Given $T$ is the initialisation statement then for all traces $tr$,

$\quad tr \in \text{traces}(LTS_M) \iff \neg wp(T; (tr)_b, false)$

Proof. By induction on the length of the trace,

Case 1 (Base Case):
Suppose $tr = ()$ then we have that () $\in \text{traces}(LTS_M)$ and clearly $\neg wp(T; (tr)_b, false) = \neg wp(T; skip, false) = \neg wp(T, false) = true$ as $T$ by definition is a non-guarded statement. Thus the case is true in both directions.

Case 2 (Inductive Case):
Assume $tr^n \in \text{traces}(LTS_M) \iff \neg wp(T; (tr^n)_b, false)$ for all traces $tr^n$ of length $n$.

Case 2.1 Suppose that $tr = tr^n \circ (op)$ for some operation $op$.

We have to show that $tr^n \circ (op) \in \text{traces}(LTS_M) \iff \neg wp(T; (tr^n)_b; op, false)$.

If $tr^n \circ (op) \in \text{traces}(LTS_M)$ then $tr^n \in \text{traces}(LTS_M)$. We apply the inductive hypothesis and derive that $\neg wp(T; (tr^n)_b, false)$ is true.

Since $op$ has no guard, by Definition 2.4.3, $wp(op, false)$ is false.
Substituting above gives \(\neg \text{wp}(T; (tr^n)_b, \text{wp}(op, false))\) is true which is equivalent to \(\neg \text{wp}(T; (tr^n)_b, op, false))\).

In the reverse case, if \(\neg \text{wp}(T; (tr^n)_b, op, false))\) and as above \(\text{wp}(op, false)\) is false we have that \(\neg \text{wp}(T; (tr^n)_b, false)\) is true. By inductive hypothesis we have that \(tr^n \in \text{traces}(LTS_M)\). Hence for some \(val \in INIT_M\) and \(val_n \in ST^+_M\) we have that \(val \overset{tr^n}{\longrightarrow} val_n\). By Lemma 4.2.1 we have that for any \(val_n\) and some \(val_{n+1}, val_n \overset{op}{\rightarrow} val_{n+1}\). Hence we have that \(tr^n \sim (op) \in \text{traces}(LTS_M)\).

The following result is used in the proof of Theorem 4.2.2 to identify a starting state and a final state for a given non-divergent trace. The lemma applies to divergent traces as well, but it does not offer any valuable result because in such cases any final state is possible.

**Lemma 4.2.2.** Suppose for some \(val^n\) we have that \(\text{primed}(val^n)\) satisfies

\[\text{prdVARIABLES}(T; (tr)_b)\]

where \(T\) is an initialisation statement.

Then for some \(val \in INIT_M\) we have that,

\[val \overset{tr}{\longrightarrow} val_n\]

**Proof.** By induction on the length of the trace,

Case 1(Base Case):
Let \(tr = \{\}\), then suppose for some \(val_n\), \(\text{primed}(val_n)\) satisfies \(\text{prdVARIABLES}(T; \text{skip})\). Then from definition of \(\text{prd}\) we have that \(\text{primed}(val_n)\) satisfies,

\[\neg \text{wp}(T; \text{skip}, \text{VARIABLES} \neq \text{VARIABLES'})\]

which is the same as \(\neg \text{wp}(T, \text{VARIABLES} \neq \text{VARIABLES'})\). Thus by Definition 4.1.3, \(val_n \in INIT_M\). Thus we have that for some \(val\),

\[val \overset{\{\}}{\longrightarrow} val_n\]

Case 2(Inductive Case):
Suppose for some \(val_m\), \(\text{primed}(val_m)\) satisfies \(\text{prdVARIABLES}(T; (tr^m)_b)\) then for some \(val \in INIT_M\) we have that,

\[val \overset{tr^m}{\longrightarrow} val_m\]

Case 2.1:
Let \(tr = tr^m \sim (op)\) for some operation \(op\).
Suppose we have that $\text{primed}(\text{val}_n)$ satisfies $\text{prdVARiABLES}(T; (\text{trm})_b; \text{op})$. By Theorem 2.5.2(2) we have that,

$$\text{prdVARiABLES}(T; (\text{trm})_b; \text{op})$$

$$\Rightarrow$$

$$\text{trm}(T; (\text{trm})_b) \Rightarrow$$

$$\exists \text{VARIABLES}'' . \text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b)) \land$$

$$\text{wp}(\text{VARIABLES} := \text{VARIABLES}'', \text{prdVARiABLES}(\text{op}))$$

Thus $\text{primed}(\text{val}_n)$ satisfies,

$$\text{trm}(T; (\text{trm})_b) \Rightarrow$$

$$\exists \text{VARIABLES}'' . \text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b)) \land$$

$$\text{wp}(\text{VARIABLES} := \text{VARIABLES}'', \text{prdVARiABLES}(\text{op}))$$

We have that $\text{primed}(\text{val}_n)$ satisfies $\text{trm}(T; (\text{trm})_b)$ because the predicate has no free names hence needs no evaluation. Thus we have that $\text{primed}(\text{val}_n)$ satisfies,

$$\exists \text{VARIABLES}'' . \text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b)) \land$$

$$\text{wp}(\text{VARIABLES} := \text{VARIABLES}'', \text{prdVARiABLES}(\text{op}))$$

Thus for some $\text{val}_m$ we have that, $[\text{primed}(\text{primed}(\text{val}_m)), \text{primed}(\text{val}_n)]$ satisfies

$$\text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b))$$

and

$$\text{wp}(\text{VARIABLES} := \text{VARIABLES}'', \text{prdVARiABLES}(\text{op}))$$

However the predicate $\text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b))$ does not contain the names $\text{VARIABLES}'$ thus we have that $\text{primed}(\text{primed}(\text{val}_m))$ satisfies,

$$\text{wp}(\text{VARIABLES}' := \text{VARIABLES}'', \text{prdVARiABLES}(T; (\text{trm})_b))$$

Removing the decorations we have that $\text{primed}(\text{val}_m)$ satisfies $\text{prdVARiABLES}(T; (\text{trm})_b)$.

So we can apply the inductive hypothesis and derive that for some $\text{val} \in \text{INIT}_M$ we have that

$$\text{val} \xrightarrow{\text{trm}} \text{val}_m$$

Above we also had that $[\text{primed}(\text{primed}(\text{val}_m)), \text{primed}(\text{val}_n)]$ satisfies

$$\text{wp}(\text{VARIABLES} := \text{VARIABLES}'', \text{prdVARiABLES}(\text{op}))$$

Removing the decorations on $\text{val}_m$ we have that

$$[\text{val}_m, \text{primed}(\text{val}_n)] \text { satisfies } \text{prdVARiABLES}(\text{op})$$

Thus by Definition 4.1.1 we have that $\text{val}_m \xrightarrow{\text{op}} \text{val}_n$. Thus by definition of $\Rightarrow$ we have that for some $\text{val}$ such that $\text{val} \in \text{INIT}_M$,

$$\text{val} \xrightarrow{\text{trm}-\text{op}} \text{val}_n$$

$\square$
The second theorem shows that a divergent trace in \( \text{LTS}_M \) is identified as divergent accordingly.

**Theorem 4.2.2. Divergence**

Given \( M \) where \( T \) is the initialisation statement for any \( tr \),

\[
tr \in D(\text{LTS}_M) \iff \neg \wp(T; (tr)_b, \ true)
\]

**Proof.** By induction on the length of the trace,

Case 1 (Base Case):
Suppose \( tr = () \) then \( () \notin D(\text{LTS}_M) \) and \( \neg \wp(T; \text{skip}, \ true) \) is false as \( T \) by definition cannot diverge. Hence, case is true vacuously in both directions.

Case 2 (Inductive Case):
Assume \( tr^n \in D(\text{LTS}_M) \iff \neg \wp(T; (tr^n)_b, \ true) \) for all traces \( tr^n \) of length \( n \).

Case 2.1:
Suppose \( tr = tr^n \setminus \langle op \rangle \) for some operation \( op \).

We have to show that \( tr^n \setminus \langle op \rangle \in D(\text{LTS}_M) \iff \neg \wp(T; (tr^n)_b; op, \ true) \).

If \( tr^n \setminus \langle op \rangle \in D(\text{LTS}_M) \) then \( tr^n \in D(\text{LTS}_M) \) or \( tr^n \notin D(\text{LTS}_M) \) and for some \( val_n \) and \( val \in \text{INIT}_M, val \xrightarrow{tr^n} val_n \) and \( val_n \xrightarrow{op} M \). The latter is possible only if \( val_n \) satisfies \( abt(op) \).

Suppose \( tr^n \in D(\text{LTS}_M) \) then we can apply the inductive hypothesis and conclude that \( \neg \wp(T; (tr^n)_b, \ true) \) is true. We have that \( \wp(op, true) \Rightarrow true \) for any \( op \) hence we have that \( \wp(T; (tr^n)_b, \ wp(op, true)) \Rightarrow \wp(T; (tr^n)_b, \ true) \) from rules of weakest precondition. Thus we have that \( \neg \wp(T; (tr^n)_b, \ wp(op, true)) \lor \wp(T; (tr^n)_b, \ true) \) and we have \( \neg \wp(T; (tr^n)_b, \ true) \). Thus it must be the case that \( \neg \wp(T; (tr^n)_b, \ wp(op, true)) \) which is equivalent to \( \neg \wp(T; (tr^n)_b; op, \ true) \).

Suppose \( tr^n \notin D(\text{LTS}_M) \) and for some \( val_n \) and \( val \in \text{INIT}_M, val \xrightarrow{tr^n} val_n \) and \( val_n \) satisfies \( abt(op) \). Then by applying the negation of the inductive hypothesis we have that \( \wp(T; (tr^n)_b, \ true) \) is true. By Theorem 2.5.2(1) we can conclude that \( \wp(T; (tr^n)_b, \ R) \) is true for any predicate \( R \). Let \( R_{val_n} \) be the least predicate that \( val_n \) satisfies. Then \( \wp(T; (tr^n)_b, \ R_{val_n}) \) is true. We have that \( val_n \) satisfies \( abt(op) \) hence \( R_{val_n} \Rightarrow abt(op) \). Hence from rules of weakest precondition we have that \( \wp(T; (tr^n)_b, abt(op)) \) is true. This implies \( \wp(T; (tr^n)_b, \neg \wp(op, true)) \). Since \( T \) and the operations of \( (tr^n)_b \) contain no guards we have that the above implies \( \neg \wp(T; (tr^n)_b, \ wp(op, true)) \) which in turn is equivalent to \( \neg \wp(T; (tr^n)_b; op, \ true) \).
4.2. Correspondence with Morgan’s failures-divergences semantics for action systems

In the reverse case, if \( \neg wp(T; (t r^n)_b; o p, \text{true}) \) is true then, 
\[
\neg wp(T; (t r^n)_b, \text{true}) \\
\lor \\
\exists \text{VARIABLES'}. \neg (\text{prdVARIABLES}(T; (t r^n)_b) \Rightarrow wp(\text{VARIABLES' := VARIABLES, trm(op))))
\]
by negation of Theorem 2.5.2(1).

Suppose, \( \neg wp(T; (t r^n)_b, \text{true}) \) is true then we can apply the inductive hypothesis and conclude that \( t r^n \in D(L T S_M) \). Thus we have that for some \( v a l \in I N I T_M \), \( v a l \overset{t r^n}{\longrightarrow} \perp \).

By Definition 4.1.1 we have that \( \perp \overset{\text{op}}{\longrightarrow} \perp \).

Suppose, 
\[
\exists \text{VARIABLES'}. \neg (\text{prdVARIABLES}(T; (t r^n)_b) \Rightarrow wp(\text{VARIABLES' := VARIABLES, trm(op))))
\]
is true.

By Definition of prd and trm we have that, 
\[
\exists \text{VARIABLES'}. \neg (\neg wp(T; (t r^n)_b, \text{VARIABLES} \neq \text{VARIABLES'}) \lor wp(\text{VARIABLES' := VARIABLES, wp(op, true))))
\]
which is equivalent to 
\[
\exists \text{VARIABLES'}. \neg wp(T; (t r^n)_b, \text{VARIABLES} \neq \text{VARIABLES'}) \land \neg wp(\text{VARIABLES' := VARIABLES, wp(op, true)})
\]
Thus there is a valuation \( v a l_n \) such that \( \text{primed}(v a l_n) \) satisfies 
\[
\neg wp(T; (t r^n)_b, \text{VARIABLES} \neq \text{VARIABLES'})
\]
and 
\[
wp(\text{VARIABLES' := VARIABLES, wp(op, true)})
\]

If \( \text{primed}(v a l_n) \) satisfies \( wp(\text{VARIABLES' := VARIABLES, wp(op, true)}) \) and by Theorem 2.5.1 \( \neg wp(op, \text{true}) \) implies \( abt(op) \) we have that \( v a l_n \) satisfies \( abt(op) \) hence, by Definition 4.1.1, \( v a l_n \overset{\text{op}}{\longrightarrow} \perp \).

Since \( \text{primed}(v a l_n) \) satisfies \( \neg wp(T; (t r^n)_b, \text{VARIABLES} \neq \text{VARIABLES'}) \) (which is equivalent to \( \text{prdVARIABLES}(T; (t r^n)_b) \)), we have by Lemma 4.2.2 that for some \( v a l \in I N I T_M \), \( v a l \overset{t r^n}{\longrightarrow} v a l_n \). Thus we have that \( v a l \overset{t r^n \neg \langle o p \rangle}{\longrightarrow} \perp \). Thus, \( t r^n \neg \langle o p \rangle \in D(L T S_M) \).

\( \Box \)
4.3 Linking π-calculus and B-Method

We define an approach which enables the interpretation of a B machine as a π-calculus labelled transition system so that it can be integrated into combinations with π-calculus processes. The interpretation is achieved through the labelled transition system $LTS_M$ explained in the previous section. The result of this work naturally extends B machines with instantiation and π-calculus dynamic reconfiguration capabilities.

Essentially, we think of a π-calculus agent as a client and a given machine $M$ as a server. In that respect, the nature of the π-calculus forces us to consider mechanisms which solve the following three technical problems.

Firstly, the actions of the π-calculus are of higher granularity than the execution semantics of a B machine operation. The execution semantics of an operation with input and output

$$\text{out} \leftarrow \text{operation(in)}$$

are that the operation with input $\text{in}$ executes spontaneously assigning the values to the outputs $\text{out}$. However, in the π-calculus inputs and outputs execute sequentially. Thus, a one to one mapping of action to operation execution as it was adopted for combinations with CSP, would not work here.

We do not consider machines with input and output, however we must provide a suitable base case for an extension later. Thus we take the view that $\text{out} \leftarrow \text{operation(in)}$ has three stages of execution: the operation selection stage-$\text{operation(in)}$, execution stage, and output stage $\text{out}$. As Figure 4.2 shows we have only operation selection followed by execution. Our operation selection does not need to consider inputs since our B operations do not have inputs and outputs.

![Figure 4.2: Executing an operation in π-calculus](image)

Secondly, we must provide a mechanism by which a π-calculus agent is only permitted to select from a predefined set of operations. Otherwise an agent might request an operation that a machine cannot execute. In a later extension it would also be necessary to ensure that inputs to operations are of an appropriate type.

Finally, suppose a system of π-calculus agents and machines come into contact with an agent that attempts to mimic the services that $M$ provides to its environment. This adds a level of non-determinism which could result in a fault at $M$, because an agent of the system might call two operations in sequence hoping that both would be executed by $M$ in sequence, however the first might be picked up by the agent mimicking
4.3. Linking $\pi$-calculus and B-Method

This problem is analogous to the well-known print server security example which justifies the use of behaviour type systems in $\pi$ [42].

For the reasons above it is essential to adopt a type—system when considering machine/$\pi$-calculus integration. We use the type-system presented in Chapter 3 which is extended here to act over machine agents.

4.3.1 The link between $M$ and $\pi$

As mentioned above we see a machine $M$ as a server servicing requests to execute operations. In contrast to the CSP||B architecture the services are offered via a single channel from $\mathcal{N}$. We use the name $z$ to denote this channel for a particular machine. We also call $z$ a machine reference to an instance of $M$. Figure 4.3 shows the process topology which we aim to achieve with the definitions that follow. The channel $z$ is decorated with type $\#V_M$ which we explain in the following section. To execute an operation a given $\pi$-agent would select the operation by making an appropriate call via the channel $z$. The instance of $M$ listening on $z$ would receive the call and then execute the requested operation before it is ready to interact again.

Firstly, we extend the state-space $ST^1_M$ adding points of machine activity as follows. $BEGIN$ is the point when $M$ is not initialised, $READY$ is the point when $M$ is ready to execute an operation, and for each operation of $M$, $BODY_{op_1}$, is when $op_i$ is being executed. If $IS = \{BEGIN, READY, BODY_{op_1}, \ldots, BODY_{op_n}\}$ where $OPERATIONS = \{op_1, \ldots, op_n\}$ then the extended state-space is given by the following definition,

**Definition 4.3.1.** $ST^1_{\pi M} = IS \times ST^1_M$.

For a given $M$, we define a system of $\pi$-processes $[s]_M(z)$ where $[s]_M(z)$ denotes a process abstraction parametrised by channel $z$ and $s \in ST^1_{\pi M}$. In Section 4.3.2 we give a syntactic definition for the process $[(READY, val)]_M(z)$ for any $val$ and the existing transition rules apply. We define $[(BODY_{op}, val)]_M(z)$ as a primitive process with
transition rules in Section 4.3.3. In the same section we will introduce transition rules for the primitive process \( (\text{BEGIN})^M(z) \). The process is written \( (\text{BEGIN})^M(z) \) because the state of the variables of \( M \) is not important at point BEGIN. Before we proceed to link these agents with a labelled transition system we integrate them into the type system of Chapter 3.

4.3.2 The \( \pi \)-calculus type-system and \( M \)

In this section we explain the type \( \#V_M \) delegated from the previous section, define the agent \( [(\text{READY}, \text{val})]^M(z) \) for any \( \text{val} \) and give rules for typing machine agents \( [s]^M(z) \) where \( s \in ST_{\pi_M} \). The reader is also asked to recall the operational semantics and type construction of the variant type from Section 3.4.2 and the typing environment that corresponds to the signature of a machine from Section 2.6.

We translate the OPERATIONS clause of \( M \) to variant type which is denoted by \( V_M \) for a particular \( M \). This type is used in a typing derivation of a combined B and \( \pi \)-calculus system. By giving a machine reference like \( z \) the type \( \#V_M \) we specify that \( z \) can be used to send or receive variant labels that correspond to execution calls for particular operations. From the point of view of a machine agent the type of \( z \) is \( iV_M \) because it is only permitted to input operation calls on \( z \). From the point of view of agents other than \( M \) the type of \( z \) is \( oV_M \) because they are only permitted to output operation calls on \( z \).

The following Definition 4.3.2 generates \( V_M \) from the signatures of OPERATIONS of \( M \). Since we do not consider I/O the definition has only one clause.

Definition 4.3.2. Given machine \( M \) where operation is defined in its OPERATIONS clause,

\[
\text{operation-unit} \in V_M \quad \text{iff operation is defined in SIG}(M)
\]

The type unit denotes the fact that operation has no inputs.

Example 4.3.1. Consider machine Clock which has two operations tick and tock,

\[ V_{Clock} = \{ \text{tick-unit}, \text{tock-unit} \} \]

is the variant type capturing the operations signature of Clock.

The labels that can pass along \( z \) of type \( V_{Clock} \) are thus, tick-(\#) to execute tick and tock-(\#) to execute tock.

Any system of agents that might come in contact with the machine on channel \( z \) must be typeable with \( z : oV_M \) in the environment. This ensures that this system only outputs on \( z \). Also, no agent can request an operation that \( M \) cannot service as the following example shows. (Note that we write typing-derivations as inverted trees with the conclusions at the top and the hypothesis at the root.)
Example 4.3.2. Consider a typing derivation which fails because a π-agent is attempting to make a Clock, ring. The inference is that of typing rule T-OUT.

\[
\begin{array}{c}
\{z : oV_{\text{Clock}}\} \vdash z : oV_{\text{Clock}} \\
\{z : oV_{\text{Clock}}\} \vdash \text{ring-\langle *\rangle : V_{\text{Clock}}}
\end{array}
\]

\[
\begin{array}{c}
\{z : oV_{\text{Clock}}\} \vdash 0
\end{array}
\]

We are now ready to give a syntactic definition of the agent \([[\text{READY}, \text{val}]]_M(z)\) for any \(M\).

Definition 4.3.3. For any \(M\), \(\text{val} \in \text{ST}_M^I\), and \(z\),

\[
[[\text{READY}, \text{val}]]_M(z) = z(a).\text{case } o \text{ of }
\]

\[
[li > [[\text{BODY}_{\text{op}(li)}], \text{val}]]_M(z); \ldots ;
\]

\[
l_i > [[\text{BODY}_{\text{op}(li)}], \text{val}]]_M(z)
\]

where \([l_i, \ldots, l_n] = V_M\) and \(\text{op}(l_i)\) is the operation that corresponds to label \(l_i\).

Thus a machine agent in a READY state, awaits the input of an operation call on its reference channel and then executes the BODY process that matches the operation call.

We now proceed to give typing rules for machine agents in state \(\text{BEGIN}\) and \(\text{BODY}\) in Table 4.1. The significance of channels \(\text{init}_M\) and \(\text{div}\) will be explained in the following section. Note we have not defined a rule for the agent \([[\text{READY}, \text{val}]]_M(z)\). As this agent is defined syntactically in Definition 4.3.3 we propose the following typing derivation.

Proposition 4.3.1. For any \(M\) and \(\text{val} \in \text{ST}_M^I\) the typing derivation

\[
\Gamma \vdash [[\text{READY}, \text{val}]]_M(z)
\]

holds, where \(\Gamma(z) = iV_M\), \(\Gamma(\text{div}) = o \text{ unit}\).

Proof. By the typing derivation \(T-\text{INP}\), followed by \(T-\text{VAR}\) and for each \(l_j \in V_M\), \(TB-\text{BODY}\), we have that,

\[
\Gamma \vdash [[\text{READY}, \text{val}]]_M(z)
\]

holds. □

Note that the theorem above also captures that machines only ever input on the machine reference channel.
4.3.3 \(\pi\)-style operational semantics for Machines without I/O

Given \(\text{LTS}_M\) for some machine \(M\) and channel \(z\) with capability \(iV_M\), the \(\pi\)-style operational semantics of \(M\) are defined in Tables 4.2 and 4.3. Rule \(RB - BODY\) in

\[
\text{RB-BODY} \quad \frac{\text{val1} \xrightarrow{\text{op}(i)} M \text{ val2}}{\text{[(BODY}_{\text{op}(i)}, \text{val1}]}_M(z) \rightarrow \text{[(READY, val2)]}_M(z) \quad \text{val2} \neq \bot}
\]

Table 4.2: Reductions of \(M\)

\[
\text{LB-BEGIN} \quad \frac{\text{val} \in \text{INIT}_M}{\text{[(BEGIN)]}_M(z) \xrightarrow{\text{init}_M} \text{[(READY, val)]}_M(z)}
\]

\[
\text{LB-BODYa} \quad \frac{\text{val1} \xrightarrow{\text{op}(i)} M \text{ val2}}{\text{[(BODY}_{\text{op}(i)}, \text{val1}]}_M(z) \xrightarrow{\tau} \text{[(READY, val2)]}_M(z) \quad \text{val2} \neq \bot}
\]

\[
\text{LB-BODYb} \quad \frac{\text{val1} \xrightarrow{\text{op}(i)} M \bot}{\text{[(BODY}_{\text{op}(i)}, \text{val1}]}_M(z) \xrightarrow{\text{div}^+} \text{[(READY, \bot)]}_M(z)}
\]

Table 4.3: Labelled transitions of \(M\)

Table 4.2 is necessary to ensure that the harmony lemma 3.2.3 extends over machine agents which is an important result in the \(\pi\)-calculus. We use mainly the rules of 4.3. By rule \(LB - BEGIN\), the agent \([(\text{BEGIN})]\_M(z)\) can initialise to the machine agent \([(\text{READY, val})]\_M(z)\) where \(\text{val} \in \text{INIT}_M\) which can process operation requests. We have set this transition to produce a visible \(\text{init}_M\) action because in our view it is useful for the environment to know that a machine has been initialised. In other applications this might be unnecessary.

Rule \(LB - BODYa\) corresponds to both divergent and non-divergent state transitions of the machine \(M\). The process \([(BODY}_{\text{op}(i)}, \text{val1}]}_M(z)\) can evolve to \([(\text{READY, val2})]}_M(z)\) by producing a \(\tau\) if \(\text{val1} \xrightarrow{\text{op}(i)} M \text{ val2}\). Note \(\text{val2}\) is not the divergent state but a divergence could still have been the cause of this transition (i.e. the system is divergent but selects a good final state by chance).

Correspondingly, \(LB - BODYb\) applies to state transitions where the final state is the divergent state. We have set this transition to produce a \(\text{div}^+\) action.

Note we have not provided transition rules for the agent \([(\text{READY, val})]}_M(z)\) for any \(M\) where the following result applies.
Lemma 4.3.1. For any machine $M$, state $\text{val} \in ST_M$ and machine reference $z$,

$$\llbracket (\text{READY}, \text{val}) \rrbracket_M(z) \xrightarrow{\alpha} \llbracket (\text{BODY}_{op(i)}, \text{val}) \rrbracket_M(z)$$

Proof. By definition of $\Longrightarrow$ we need to consider two transition derivations. The first where $\alpha = z \ s$, which is inferred using Definition 4.3.3 and rule $\text{INP}$ where we let

$$P' \triangleq \text{case } l \text{ of}
\begin{align*}
& l_1 \triangleright \llbracket (\text{BODY}_{op(h)}, \text{val}) \rrbracket_M(z); \\
& l_n \triangleright \llbracket (\text{BODY}_{op(h)}, \text{val}) \rrbracket_M(z)
\end{align*}$$

The second where $\alpha = \tau$ which is inferred using $P'$ above and Definition 3.4.3. □

In following chapters we will state that $\llbracket (\text{READY}, \text{val}) \rrbracket_M(z) \xrightarrow{\alpha} \llbracket (\text{BODY}_{op(i)}, \text{val}) \rrbracket_M(z)$ in reference to Lemma 4.3.1 because the $\tau$ after the $z \ s$ has no significance to our work and is simply a product of the variant construct.

By the transitions of Table 4.3 we have introduced an action $\text{init}M$ when a machine is initialised. More importantly before evolving to the divergent state a machine produces an $\text{div}$ action. Normally, divergence in a process algebra is interpreted as total non-determinism including the possibility of performing an infinite sequence of $\tau$ actions. Thus, machine divergence in our model is different. In a system of $\pi$ agents and machines, the design decisions we have taken will facilitate the identification of a machine divergent trace. Thus, we can use traces to reason about machine divergences in $\pi$. We have still retained some level of non-determinism however, as a diverging machine might still produce a $\tau$ instead of the $\text{div}$. The following example should illustrate these points with the aid of the $\text{Clock}$ machine.

4.3.4 Examples with machine $\text{Clock}$

The following diagram visually presents the $\pi$-style transitions that machine $\text{Clock}$ can make. We can now use $\text{Clock}$ in a $\pi$ specification.

Example 4.3.3. Consider the following process which can be interpreted as an infinite supply of $\text{Clock}$ instances.

$$MGEN \triangleq l(\nu \ z : \text{Unit} \cdot \text{Clock} :: \text{initClock} :: \text{unit})
\begin{align*}
& (\text{createClock}(z) \cdot \text{initClock} \cdot 0) \mid \llbracket \text{BEGIN} \rrbracket_{\text{Clock}(z)}
\end{align*}$$

The process is replicated infinitely. Each replicant can create a new instance of a $\text{Clock}$ as shown in Figure 4.5. Observe that machine creation corresponds to two transitions as illustrated in Figure 4.6. At first only the channel $\text{createClock}$ is available which outputs a fresh machine reference (channel $z$). Following this interaction, the environment knows the machine reference but it cannot interact on it until the machine is initialised; the initialisation is achieved by a internal synchronisation on channel $\text{initClock}$. 
Figure 4.4: \( \pi \)-calculus style LTS of *Clock* machine
4.4. Some results about machine agents

This section revisits some of the results given in Chapter 3 about the \( \pi \)-calculus and extends them to processes containing machine agents.

Firstly note that the rules of structural congruence extend over machine agents in a natural way without the need to give special cases.

**Example 4.4.1.** For some \( \pi \)-calculus agent \( D \),

\[
D \mid \llbracket (\text{BEGIN}) \rrbracket_M(z) \equiv_\pi D \mid \llbracket (\text{BEGIN}) \rrbracket_M(z) \mid 0 \\
\equiv_\pi D \mid \llbracket (\text{BEGIN}) \rrbracket_M(z) \mid (\nu w)(0) \\
\equiv_\pi (\nu w)(D \mid \llbracket (\text{BEGIN}) \rrbracket_M(z) \mid 0) \\
\equiv_\pi (\nu w)(D \mid \llbracket (\text{BEGIN}) \rrbracket_M(z))
\]

The ability to create infinite number of instances of machines is essential to our specification style. Therefore, in the final section we proceed to give a general definition and prove some results about them.
for some \( w \in \mathcal{N} \) such that \( w \neq z \) and \( w \notin \text{fn}(D) \).

In view of that any result on the \( \pi \)-calculus processes which is not dependent on the syntax also applies to processes containing machine agents. One example of such a result is the Harmony Lemma 3.2.3 where the processes \( P \) and \( Q \) might contain machine agents.

As the Substitution Lemma 3.4.1 and Subject Reduction Theorem 3.4.1 are dependent on the syntax we extend them here in Lemma 4.4.1 and Theorem 4.4.1 respectively.

**Lemma 4.4.1.** If,

1. \( \Gamma \vdash P \),
2. \( \Gamma(a) = S \),
3. \( \Gamma \vdash v : S \) then

\( \Gamma \vdash P \{v/a\} \).

**Proof.** By structural induction on \( P \).

Case 1(Base Case):

Case 1.1:
Suppose \( \Gamma \vdash \mathcal{M}(\text{BEGIN})_z, \Gamma(z) = \#V_M \) and \( \Gamma \vdash v : \#V_M \).

Then we need to show that \( \Gamma \vdash \mathcal{M}(\text{BEGIN})_z \{v/z\} \) which is true by application of rule \( \text{TB} - \text{BEGIN} \).

Case 1.2:
Suppose \( \Gamma \vdash \mathcal{M}(\text{BODY}_\text{op}, \text{val})_z, \Gamma(z) = \#V_M \) and \( \Gamma \vdash v : \#V_M \).

Then we need to show that \( \Gamma \vdash \mathcal{M}(\text{BODY}_\text{op}, \text{val})_z \{v/z\} \) which is true by application of rule \( \text{TB} - \text{BODY} \).

Case 1.3:
Suppose \( \Gamma \vdash \mathcal{M}(\text{READY}, \text{val})_z, \Gamma(z) = \#V_M \) and \( \Gamma \vdash v : \#V_M \).

Then we need to show that \( \Gamma \vdash \mathcal{M}(\text{READY}, \text{val})_z \{v/z\} \) which is true by Proposition 4.3.1.

The remaining proof is identical to Lemma 3.4.1. \( \Box \)

We extend the subject reduction theorem where agents \( P \) and \( P' \) might contain machine agents.
Theorem 4.4.1. If $\Gamma \vdash P$, and $\Gamma$ is closed, and $P \xrightarrow{\alpha} P'$ then,

1. if $\alpha = \tau$ then $\Gamma \vdash P'$,

2. if $\alpha = a : v$ then there is a $T$ such that,
   
   (a) $\Gamma \vdash a : iT$,
   (b) if $\Gamma \vdash v : T$ then $\Gamma \vdash P'$.

3. if $\alpha = (\nu x : T)\alpha v$ then there is a $T$ such that,
   
   (a) $\Gamma \vdash a : iT$,
   (b) $\Gamma, x : T \vdash v : T$,
   (c) $\Gamma, x : T \vdash P'$.

Proof. By induction on $P \xrightarrow{\alpha} P'$.

Case 1(Base Case):

Case 1.1:
Suppose $\Gamma \vdash \llbracket(BEGIN)\rrbracket_M(z)$ with $\Gamma$ closed (i.e. $\Gamma(z)$, $\Gamma(div)$ and $\Gamma(initM)$ are defined), by $LB - BEGIN$ we have that

$$\llbracket(BEGIN)\rrbracket_M(z) \xrightarrow{\text{initM}} \llbracket(READY, val)\rrbracket_M(z)$$

By rule $TB - BEGIN$ we have that $\Gamma \vdash initM : i \text{ unit}$ and we have that $\Gamma \vdash * : \text{ unit}$. By Proposition 4.3.1 we have that $\Gamma \vdash \llbracket(READY, val)\rrbracket_M(z)$ holds. Hence case holds.

Case 1.2:
Suppose $\Gamma \vdash \llbracket(BODY_{op}, val)\rrbracket_M(z)$ by $LB - BODYa$ we have that

$$\llbracket(BODY_{op}, val)\rrbracket_M(z) \xrightarrow{\tau} \llbracket(READY, val2)\rrbracket_M(z)$$

By rule $TB - BODY$ we have that $\Gamma \vdash div : o \text{ unit}$ and we have that $\Gamma \vdash * : \text{ unit}$ which is not sufficient however by Proposition 4.3.1 we have that $\Gamma \vdash \llbracket(READY, val2)\rrbracket_M(z)$ holds already. Hence, case holds.

Case 1.3:
Suppose $\Gamma \vdash \llbracket(BODY_{op}, val)\rrbracket_M(z)$ with $\Gamma$ closed (i.e. $\Gamma(z)$, $\Gamma(div)$ and $\Gamma(initM)$ are defined), by $LB - BODYb$ we have that

$$\llbracket(BODY_{op}, val)\rrbracket_M(z) \xrightarrow{\text{div}} \llbracket(READY, 1)\rrbracket_M(z)$$

By rule $TB - BODY$ we have that $\Gamma \vdash div : o \text{ unit}$ and we have that $\Gamma \vdash * : \text{ unit}$ and by Proposition 4.3.1 we have that $\Gamma \vdash \llbracket(READY, 1)\rrbracket_M(z)$ holds. Hence, case holds.

The remaining proof is identical to Theorem 3.4.1. □
4.5 The $\text{MGENERATOR}$ specification

We define a simple process that we find useful later on in our specifications.

Definition 4.5.1.

$$\text{MGENERATOR} = \exists \left( \nu z : \# V M_1, \; \text{init} M_1 : \# \text{unit} \right)$$

$$\frac{}{(\text{create} M_1(z). \text{init} M_1.0 \mid \text{BEGIN} M_1(z)) | \ldots | \left(\nu z : \# V M_n, \; \text{init} M_n : \# \text{unit} \right)$$

$$\frac{}{(\text{create} M_n(z). \text{init} M_n.0 \mid \text{BEGIN} M_n(z))}$$

where $M_1, \ldots, M_n$ are the machines we wish to consider in a specification.

Let $P \Downarrow \alpha$ denote that there exists an ouput action $\alpha$ with subject $x$, and a process $P'$ such that, $P \xrightarrow{\alpha} P'$.

The following lemma shows that $\text{MGENERATOR}$ can output on a $\text{create} M_i$ channel.

Lemma 4.5.1. If $M_1, \ldots, M_n$ are the machines we wish to consider in a specification, $\text{MGENERATOR} \downarrow \text{create} M_i$ for any $1 \leq i \leq n$.

Proof. We have that,

$$\text{MGENERATOR} \equiv_{\pi}$$

$$\left( \nu z : \# V M_i, \; \text{init} M_i : \# \text{unit} \right)(\text{create} M_i(z). \text{init} M_i.0 \mid \text{BEGIN} M_i(z)) \mid \text{MGENERATOR}$$

for any $1 \leq i \leq n$.

By transition rules $\text{PAR-L}$, $\text{RES}$, and $\text{PAR-L}$ we have that

$$\left( \left( \nu z : \# V M_i, \; \text{init} M_i : \# \text{unit} \right)(\text{create} M_i(z). \text{init} M_i.0 \mid \text{BEGIN} M_i(z)) \right) \downarrow \text{create} M_i$$

Lemma 4.5.2. Given a process $P$ where $P \equiv_{\pi} P_1 \mid \text{MGENERATOR}$ for some $P_1$ and $P \xrightarrow{\alpha} P'$ for any $\alpha$ then $P' \equiv_{\pi} P_2 \mid \text{MGENERATOR}$ for some $P_2$.

Proof. By case analysis on $P \xrightarrow{\alpha} P'$ the lemma is trivially true in each case.

The following theorem shows that $\text{MGENERATOR}$ is always capable of outputting on a $\text{create} M_i$ channel.
Theorem 4.5.1. If $M_1, \ldots, M_n$ are the machines we wish to consider in the specification, for any trace $tr$ if

$$M_{\text{GENERATOR}} \xrightarrow{tr} P'$$

then $P' \Downarrow \text{create} M_i$ for $1 \leq i \leq n$.

Proof. Let $tr^m$ be a trace of length $m$ and let $tr = tr^m$ for some $m \in \mathbb{N}$.

Then we have that,

$$M_{\text{GENERATOR}} \xrightarrow{tr^m} P'$$

We have that $M_{\text{GENERATOR}} \equiv_\pi \emptyset | M_{\text{GENERATOR}}$, then by application of Lemma 4.5.2 $m$ times we have that for some $P_2$, $P' \equiv_\pi P_2 | M_{\text{GENERATOR}}$.

By rule PAR-R and Lemma 4.5.1 we have that $P' \Downarrow \text{create} M_i$ for $1 \leq i \leq n$. □

The final result shows that $M_{\text{GENERATOR}}$ is not the cause of machine divergence. As Lemma 4.5.2 shows $M_{\text{GENERATOR}}$ always evolves to a process of the form $P_2 | M_{\text{GENERATOR}}$ for any action. By performing a transition derivation of a $\text{create} M_i(z)$ action we can show that the machine that has been created on $z$ is a component of $P_2$ and is accessible by the context. Hence, in general it is not possible to show that $M_{\text{GENERATOR}}$ is a machine divergent free system (i.e. it is possible to provide a context which executes an operation from a diverging state). The result below resolves this problem by ignoring those traces that involve communication between active machines and the context. Hence in all evolutions of $M_{\text{GENERATOR}}$ that we consider the active machines are blocked in a BEGIN state.

Let $\text{ref}(P)$ denote the set of free channels in $P$ that correspond to machine references.

Theorem 4.5.2. Suppose for some $m \in \mathbb{N}$,

$$M_{\text{GENERATOR}} \xrightarrow{(\alpha_1, \ldots, \alpha_m)} P_m$$

such that for all $i$ and $j$, where $i < j$ we have that $\text{subj}(\alpha_i) \notin \text{ref}(P_j)$.

Then,

$$\neg(P_m \Downarrow \text{div})$$

Proof. By induction on the length of the trace.

Case 1 (Base Case):
Suppose $m = 0$. We have that,

$$M_{\text{GENERATOR}} \xrightarrow{\emptyset} M_{\text{GENERATOR}}$$
where \( \text{ref}(M_{\text{GENERATOR}}) = \emptyset \) as there are no free machine references.

We need to show that,

\[
- (M_{\text{GENERATOR}} \downarrow \text{div})
\]

Which is true as there is no sequence of transition rules that would infer that,

\[
M_{\text{GENERATOR}} \xrightarrow{\text{div}} P'
\]

for any \( P' \).

Case 2 (Inductive Case):
Suppose for some \( k \in \mathbb{N} \) we have that,

\[
M_{\text{GENERATOR}} \langle \alpha_1, \ldots, \alpha_k \rangle P_k
\]

such that for all \( i \) and \( j \), where \( i < j \) we have that \( \text{subj}(\alpha_j) \notin \text{ref}(P_i) \).

Then,

\[
- (P_k \downarrow \text{div})
\]

Let \( m = k + 1 \).

We have that \( M_{\text{GENERATOR}} \equiv_{\pi} 0 \mid M_{\text{GENERATOR}} \) hence by applying Lemma 4.5.2, \( k \) times we have that for some process \( P_2, P_k \equiv_{\pi} P_2 \mid M_{\text{GENERATOR}} \). Since \( \text{ref}(M_{\text{GENERATOR}}) = \emptyset \) we have that \( \text{ref}(P_k) = \text{ref}(P_2) \).

If \( \alpha_k \) is constrained so that only those actions are allowed for which \( \text{subj}(\alpha_k) \notin \text{ref}(P_2) \) then there are two remaining possibilities. Either \( \alpha_k = \tau \) as a result of some \( \text{init}M_i \) interaction or \( \text{create}M_i \tau \) from the \( M_{\text{GENERATOR}} \) component.

In either case the action is not one where a machine agent in \( P_2 \) can move to a \((\text{BODY}_{op(l)}, \text{val})\) state for any \( \text{val} \) and variant label \( l \). We know that an \( \text{div} \) is possible only from one of these states from the rules of Table 4.3. Hence, if \( P_k \overset{\alpha_k}{\rightarrow}, P_m \) where for some \( P_3, P_m \equiv_{\pi} P_3 \mid M_{\text{GENERATOR}} \) we know that \( -(P_3 \downarrow \text{div}) \).

We know that \( -(M_{\text{GENERATOR}} \downarrow \text{div}) \) from the base case and we know that \( -(P_3 \downarrow \text{div}) \). Hence by \( \text{PAR-L} \) and \( \text{PAR-R} \) follows that

\[
- (P_m \downarrow \text{div})
\]

\( \square \)
Chapter 5

Constructing Combined Specifications

This chapter uses the π-calculus in an applied manner to capture the general characteristics of the systems we consider. The first section looks at certain properties a process must have so that it can control B machines. The next two sections specify a method for constructing such processes using hiding, parallel, infinite replication and the sequential constructs of the π-calculus respectively. We conclude this chapter by defining the term control system which is the unit of specification in our new π | B framework. We also define the property of machine divergence freedom for control systems.

5.1 Combined specification in π and B

5.1.1 Mediators

Mediators are π-processes that will run concurrently with B Machines. We think of machine instances as being transmitted from one mediator to another throughout the evolution of a system. The transmission takes place along designated channels which we refer to as control points. Similar to CSP||B, we aim to develop a structured, and compositional framework for verification of machine divergence freedom properties. For this purpose we constrain the mobility of machine instances as follows.

Firstly, after passing a machine reference to another mediator, a mediator cannot make more operation calls to that machine or pass the reference again. In a system of mediators, if initially each machine reference is associated with at most one mediator then the above property ensures that throughout the evolution of the system each machine reference is associated with at most one mediator.

In a closed system of mediators, machine mobility is internalised (in the sense of the private π-calculus [49, 5]). Mediators only receive machines instances which they do not already control as shown in Figures 5.1 and 5.2. In Figure 5.1, control of \( M \) is passed
from Mediator1 to Mediator2 via the control point so that M becomes associated with Mediator2. In Figure 5.2, control of M is passed from Mediator1 to Mediator2 via the control point but Mediator1 retains control of M. Thus, two mediators are associated with M and this is not allowed. The rest of this section formalises the above.

In a given \( \pi \)-calculus agent \( D \), we identify all free channels on which machine instances are transmitted as control points. These channels must obey the following properties which are enforced by a type system in Section 5.2.

**Definition 5.1.1. Control points**

1. Control points are used to transmit machine references only.

2. Control points are monadic channels.

3. Control points always transmit the same kind of machine.

To help us identify the control points we partition the infinite name space \( N \), in three parts. The set of all control points is denoted with \( CP \), such that \( CP \subseteq N \). Let \( CP \) range over \( cp_i \) for some appropriate index \( i \). Then, the set of control points in a given agent \( D \), given by \( fn(D) \cap CP \) is denoted with \( cp(D) \). Notice that this set might be empty but if it is not, its members are from the free channels in \( D \). There may be hidden channels within \( D \) which also transmit machine references but we do not need to take these into account because control points are communication channels between mediators.

The set of all machine references is denoted by \( MR \), such that \( MR \subseteq N \), and \( MR \cap CP = \emptyset \). Let \( MR \) range over \( b_i, z_i \) for some appropriate index \( i \). The set of machine
references in a given mediator $D$, given by $fn(D) \cap MR$ is denoted with $mref(D)$. The set $N \backslash (CP \cup MR)$ is denoted by $SN$ for standard names. For example, the agent
\[ D \equiv a.cp_1(z).x_1\ op_\cdot..0 \]
is a simple process where $a \in SN$, $cp(D) = \{cp_1\}$ and $mref(D) = \{z_1\}$ and in such a simple case we will normally omit the subscript.

We capture the desired behaviour of mediators using a predicate called mediator predicate defined below. This predicate addresses certain rely-guarantee style properties on the use of channel names. This technique is adopted from [32].

**Definition 5.1.2.** A predicate $M$ on $\pi$-calculus agents, is a mediator-predicate if whenever,
\[ M(D) \land D \xrightarrow{\alpha} D' \]
where $D$ is a $\pi$-calculus agent then

1. For any $a \in SN$ if,
   \[ \alpha = a.w \land w \in SN \text{ then } M(D'), \]
   \[ \alpha = \overline{a}.w \text{ then } w \in SN \land M(D'), \]
   \[ \alpha = \overline{a}(v) \text{ then } v \in SN \land M(D'), \]

2. For any $cp \in CP$ if,
   \[ \alpha = cp.z \land (z \in MR \land z \notin mref(D)) \text{ then } M(D'), \]
   \[ \alpha = \overline{cp}.z \text{ then } z \in MR \land z \notin mref(D') \land M(D'), \]
   \[ \alpha = \overline{cp}(z) \text{ then false, (i.e. it is not possible) } \]
3. For any \( z \in M \mathcal{R} \) if,
\[
\begin{align*}
\alpha &= z l \text{ then false} \\
\alpha &= \bar{z} l \text{ then true for some valid label } l \in V_M \land M(D'), \\
\alpha &= \bar{z} (l) \text{ then false},
\end{align*}
\]

4. If \( \alpha = \tau \) then \( M(D') \).

Example 5.1.1. Consider the following predicate,
\[
M((\nu f)(\bar{x}(f).\bar{c}(z).0)) = \text{true for any } f \in SN\{x\} \\
M(\bar{c}(z).0) = \text{true} \\
M(0) = \text{true}
\]

\( M \) is a mediator predicate according to Definition 5.1.2.

Definition 5.1.3. \( D \) is a mediator if there is a mediator predicate \( M \) such that \( M(D) \) holds.

Definition 5.1.3 and 5.1.2 identify the behavioural constraints on mediators. These will be important later in this chapter when we consider constructing mediators and composing them into larger systems. We illustrate the definitions in the following four examples.

Firstly, mediators are not permitted to output control points.

Example 5.1.2. Consider the following process which outputs a channel which is later determined to be a control point, \( b \in M \mathcal{R} \).
\[
D = (\nu cp)(\bar{x}(cp).cp(b).\bar{b} \text{ tick}(\ast).0)
\]

\( D \) has no control points initially \( (cp(D) = \emptyset) \) because \( cp \) is a hidden channel.

However,
\[
D \xrightarrow{x(cp)} D_1
\]
where \( D_1 = cp(b).\bar{b} \text{ tick}(\ast).0 \). \( D_1 \) has a control point \( cp \). However, Definition 5.1.2(1) is invalidated because whenever \( \alpha = \bar{a} (v) \) then \( v \in SN \). Therefore, \( D \) is not a mediator.

The framework proposed in this thesis does not allow dynamic communication of the control points. Thus, in addition to disallowing the output of control points, we also disallow the input of control points, which is why Definition 5.1.2(1) contains an assumption that \( w \in SN \) in the case \( \alpha = a w \). This assumption is formally justified when we consider constructing mediators in Section 5.2. The constraint on control points ensures that throughout the evolution of a mediator its set of control points can only decrease. Proposition 5.1.1 below establishes that the set of control points can only decrease by considering the remaining actions by which a mediator can evolve.
5.1. Combined specification in \( \pi \) and \( B \)

**Proposition 5.1.1.** For any \( \alpha \) and agent \( D \), if \( D \) is a mediator and \( D \xrightarrow{\alpha} D' \) then \( \text{cp}(D') \subseteq \text{cp}(D) \).

**Proof.** Assume \( D \) is a mediator then there is a mediator predicate \( M \). Assume that, \( D \xrightarrow{\alpha} D' \).

Case 1: For any \( \alpha = x \, v \) where \( x \in \mathcal{N} \) and \( v \in \mathcal{N} \) by Definition 5.1.2(.1) \( v \in S\mathcal{N} \) or by Definition 5.1.2(.2) \( v \in \mathcal{M}\mathcal{R} \) hence \( v \notin \text{CP} \). From Lemma 3.2.1 we have that \( \text{fn}(D') \subseteq \text{fn}(D) \cup \{v\} \). It, follows that \( \text{fn}(D') \cap \text{CP} \subseteq (\text{fn}(D) \cup \{v\}) \cap \text{CP} \). However since \( v \notin \text{CP} \), \( \{v\} \cap \text{CP} = \emptyset \). Hence, \( (\text{fn}(D) \cup \{v\}) \cap \text{CP} = \text{fn}(D) \cap \text{CP} \). Thus, \( \text{cp}(D') \subseteq \text{cp}(D) \).

Case 2: For any \( \alpha = \overline{x} \, (v) \) where \( x \in \mathcal{N} \) and \( v \in \mathcal{N} \) by Definition 5.1.2 \( v \in S\mathcal{N} \). From Lemma 3.2.1 we have that \( \text{fn}(D') \subseteq \text{fn}(D) \cup \{v\} \). It, follows that \( \text{fn}(D') \cap \text{CP} \subseteq (\text{fn}(D) \cup \{v\}) \cap \text{CP} \). However, since \( v \notin \text{CP} \), \( \{v\} \cap \text{CP} = \emptyset \). Hence, \( (\text{fn}(D) \cup \{v\}) \cap \text{CP} = \text{fn}(D) \cap \text{CP} \). Thus, \( \text{cp}(D') \subseteq \text{cp}(D) \).

Case 3: For any other \( \alpha \) not covered by previous cases. From Lemma 3.2.1 we have that \( \text{fn}(D') \subseteq \text{fn}(D) \). Hence, it follows that \( \text{fn}(D') \cap \text{CP} \subseteq \text{fn}(D) \cap \text{CP} \). Hence, \( \text{cp}(D') \subseteq \text{cp}(D) \). \( \square \)

Secondly, note that in the case \( \alpha = \overline{cp} \, z \) of Definition 5.1.2, the continuation mediator cannot refer to the \( z \) machine reference \( (z \, \text{mref}(D')) \). This constraint ensures that a mediator is allowed to output a machine reference only once (unless the mediator inputs the machine reference back in a later interaction).

**Example 5.1.3.** Consider the following process where \( \text{cp} \in \text{CP} \). The process inputs a machine reference \( b \) and then evolves to a concurrent system of two processes trying to output the machine reference.

\[
D = \text{cp}(b). (\overline{\text{cp}}(b).0 | \overline{\text{cp}}(b).0)
\]

After performing the trace \( (\text{cp} \, z, \overline{\text{cp}} \, z) \), \( D \) progresses to \( D_2 \equiv (0 | \overline{\text{cp}}(z).0) \) where \( z \in \text{mref}(D_2) \) and this invalidates Definition 5.1.2(.2). Therefore, \( D \) is not a mediator.

In the example above \( D \) can output a machine reference twice. If a mediator is permitted to output the \( z \) channel more than once then we cannot guarantee that only one mediator is communicating to the machine along \( z \) at any one time throughout the evolution of a system of mediators.

If no mediator is prepared to output a machine reference more than once then, in a system of mediators, no mediator can input a particular machine reference more than once. This justifies the assumption \( z \notin \text{mref}(D) \) in the case \( \alpha = \text{cp} \, z \) of Definition 5.1.2(.2). The assumption is formally justified when we consider constructing mediators in Section 5.2.
Example 5.1.4. Consider the following process, which inputs two machine references in sequence and then outputs them.

\[ D = \text{cp}(b).\text{cp}(c).\text{cp}(b).\text{cp}(c).0 \]

\(D\) is not a mediator if the following is permitted to occur,

\[ D \overset{\text{cp}(z).\text{cp}(z).0}{\longrightarrow} D' \overset{\text{cp}(z).\text{cp}(z).0}{\longrightarrow} \]

This is because \(\text{cp}(z).\text{cp}(z).0\) is not a mediator as discussed in the previous example. However, Definition 5.1.2 is vacuously true if the \(z\) channels are the same. Hence, we do consider \(D\) to be a mediator under the assumption that the two \(z\) channels are different.

Furthermore, the constraint in Definition 5.1.2(.2) ensures that mediators cannot make operation calls on a machine after outputting its reference.

Example 5.1.5. Consider the following process, which accepts any machine reference \(z\) over control point \(\text{cp}\) and then concurrently executes two sub-processes which interact on a hidden channel \(d\).

\[ D = \text{cp}(b).(\nu \quad d)(\overline{d} \quad \text{tick}-(*) \quad d.0 \mid d.\text{cp}(b).0) \]

Suppose \(D\) performs a \(\text{cp} \quad z\) action for some machine reference \(z\) to process \(D'\). Then the observable execution of \(D'\) is such that \(\overline{z} \quad \text{tick}-(*)\) always occurs before \(\text{cp} \quad z\) because of the synchronisation on the hidden \(d\). \(D\) is therefore, a valid mediator. Note however, if the process had been defined so that \(\overline{b} \quad \text{tick}-(*)\) is placed after the \(d\), then it would not be a valid mediator.

In the above example if \(D\) had been defined so that \(\overline{b} \quad \text{tick}-(*)\) is placed after the \(d\), during the evolution of \(D\), \(\text{cp} \quad z\) may occur before the \(\overline{z} \quad \text{tick}-(*)\) for some machine reference \(z\). If that happens then we cannot guarantee that the operation will be executed correctly because the environment has access to the \(z\) channel and may execute its own operations before \(D\).

All the examples above serve to highlight the constraints we place on mediators. We have placed a restriction on communicating control points so that it makes it easier to identify where machines come in contact with mediators. We also place a restriction on the communication of machine references. This constraint ensures that only one machine reference can be associated with a particular mediator at any one time. It enables us to identify if a given mediator \(D\) is consistent for executing B Machines without having to consider the environment. Considering mediators and their associated machines in this way means that we can prove consistency compositionally for large systems built from many mediators as we shall see in Chapter 6.

The following proposition summarises some basic results about the set of machine references throughout the evolution of a mediator.
Proposition 5.1.2. Given any mediator $D$, suppose $D \xrightarrow{\alpha} D'$,

1. if $\alpha = \overline{cp}z$ for some control point $cp$, $cp \in cp(D)$, and machine reference $z$, $z \in \text{mref}(D)$, then $\text{mref}(D') \subseteq (\text{mref}(D) \setminus \{z\})$,

2. if $\alpha = cp z$ for some control point $cp$, $cp \in cp(D)$, and machine reference $z$, $z \in \text{MR}$, then $\text{mref}(D') \subseteq (\text{mref}(D) \cup \{z\})$,

3. in all other cases $\text{mref}(D') \subseteq \text{mref}(D)$.

Proof. Consider proposition 5.1.2.(1). By Lemma 3.2.1 we have that $\text{fn}(D') \subseteq \text{fn}(D)$. Hence, $(\text{fn}(D') \cap \text{MR}) \setminus \{z\} \subseteq (\text{fn}(D) \cap \text{MR}) \setminus \{z\}$. From Definition 5.1.2.2 we have that $z \notin \text{mref}(D')$. Hence $z \notin \text{fn}(D') \cap \text{MR}$. Therefore, $(\text{fn}(D') \cap \text{MR}) \subseteq (\text{fn}(D) \cap \text{MR}) \setminus \{z\}$ from which it follows directly that $\text{mref}(D') \subseteq \text{mref}(D)$.

Consider proposition 5.1.2.(2). By Lemma 3.2.1 we have that $\text{fn}(D') \subseteq \text{fn}(D) \cup \{z\}$. Hence, $(\text{fn}(D') \cap \text{MR}) \subseteq ((\text{fn}(D) \cup \{z\}) \cap \text{MR})$ and $((\text{fn}(D) \cup \{z\}) \cap \text{MR}) = ((\text{fn}(D) \cap \text{MR}) \cup \{z\})$. Therefore, $(\text{fn}(D') \cap \text{MR}) \subseteq (\text{fn}(D) \cap \text{MR}) \cup \{z\}$ from which it follows directly that $\text{mref}(D') \subseteq \text{mref}(D) \cup \{z\}$.

Consider proposition 5.1.2.(3). By Lemma 3.2.1 we have that $\text{fn}(D') \subseteq \text{fn}(D)$ or $\text{fn}(D') \subseteq \text{fn}(D) \cup \{a\}$ for some $a \notin \text{MR}$. We check the second branch because it includes more names. $(\text{fn}(D') \cap \text{MR}) \subseteq ((\text{fn}(D) \cup \{z\}) \cap \text{MR}) \cup \{z\} \cap \text{MR}) = ((\text{fn}(D) \cap \text{MR}) \cup \{z\})$. Therefore, $(\text{fn}(D') \cap \text{MR}) \subseteq (\text{fn}(D) \cap \text{MR}) \cup \{z\}$ from which it follows directly that $\text{mref}(D') \subseteq \text{mref}(D) \cup \{z\}$.

The following Lemma establishes the basis for using structural congruence on mediators. We aim to show that given a mediator $D_1$, if $D_1 \equiv_\pi D_2$ then $D_2$ is a mediator. In proving this and other theorems about mediators we adopt the following general approach. $D$ is a mediator hence there is a mediator predicate $M$ such that $M(D)$ holds and from the definition of mediator predicates above we have that if $D_1 \xrightarrow{\alpha} D'_1$ then under certain conditions on $\alpha$ by Definition 5.1.2, $M(D'_1)$ holds. We are required to find a mediator predicate $M'$ for which $M'(D'_2)$ holds. An $M'$ is constructed by adding every process that is structurally congruent to a process for which $M$ gives true. We link between $M$ and $M'$ using results about structural congruence such as the Harmony Lemma 3.2.3 on page 36. In showing that $M'$ is indeed a mediator predicate, as Figure 5.3 shows, we essentially complete a square. If $M'(D'_2)$ holds then there is a $D_2$ such that $D_2 \equiv_\pi D'_2$ and $M(D_2)$ is true, if $D_2 \xrightarrow{\alpha} D'_2$ then $D_1 \xrightarrow{\alpha} D'_1$ for some $D'_1$ such that $D'_1 \equiv_\pi D'_2$. We know that $M(D'_1)$ is true, hence we can link back and say that $M'(D'_2)$ is true.

An important point to note is that in a mediator such as $[a = a] \pi . D_1$ for some prefix $\pi$ the name $a$ is not a machine reference (i.e. $a \notin \text{MR}$). This is because all machine references are fresh whenever they are received so processes of the above form are not useful. For example, consider the transitions of $cp(a).[a = b] \pi . D_1$, it is never going to be the case that the $a$ and $b$ match. Correspondingly whenever we derive $D_1 \equiv_\pi D_2$, 77
the rule \( SC - MAT \) is never used with machine references. For example we do not want a case \([a = a] \alpha \text{ operation}_{-}(\alpha) . D_1 \equiv_\pi \alpha \text{ operation}_{-}(\alpha) . D_1 \) where \( \alpha \in M\mathcal{R} \). As a result the following property holds which is used in the proof of Lemma 5.1.2.

**Lemma 5.1.1.** Given mediator \( D_1 \) if \( D_1 \equiv_\pi D_2 \) then \( \text{mref}(D_1) = \text{mref}(D_2) \).

**Proof.** By Proposition 3.2.1 if \( D_1 \equiv_\pi D_2 \) can be derived without \( SC - MAT \) then \( \text{fn}(D_1) = \text{fn}(D_2) \). \( \text{mref}(D_1) = \text{mref}(D_2) \) follows directly from \( \text{fn}(D_1) = \text{fn}(D_2) \) since \( \text{mref}(D_1) = (\text{fn}(D_1) \cap M\mathcal{R}) \) and \( \text{mref}(D_2) = (\text{fn}(D_2) \cap M\mathcal{R}) \).

In the case \([a = a] \alpha . D_1 \equiv_\pi D_1 \) where \( a \in M\mathcal{R} \) is omitted formally.

**Lemma 5.1.2.** Given mediator \( D_1 \) if \( D_1 \equiv_\pi D_2 \) then \( D_2 \) is a mediator.

**Proof.** \( D_1 \) is a mediator hence there exists a mediator predicate \( M \) such that \( M(D_1) \) holds. We need to show that there is a mediator predicate \( M' \) such that \( M'(D_2) \) holds.

Consider the following predicate defined for any \( D_2 \),

\[
M'(D_2) \equiv (\exists D_1 . D_1 \equiv_\pi D_2 \land M(D_1)) \tag{5.1}
\]

We need to show that \( M' \) is a mediator predicate.
Assume that $M'(D_2)$ is true and $D_2 \xrightarrow{\beta} D'_2$. Then since $M'(D_2)$ is true, by Equation 5.1, we have that $\exists D_1, D_1 \equiv \pi D_2 \land M(D_1)$. Then since $D_2 \xrightarrow{\beta} D'_2$ and $D_1 \equiv \pi D_2$ by Harmony Lemma 3.2.3 we have that $D_1 \xrightarrow{\alpha} D'_1$ such that $\alpha = \beta$ and $D'_1 \equiv \pi D'_2$ for any $\alpha$.

Thus, case analysis on $\beta$ for each of the categories of Definition 5.1.2 we identify the following:

Since we have that $\alpha = \beta$ and by Lemma 5.1.1, $mref(D_2) = mref(D_1)$ we conclude that $\alpha$ satisfies the same conditions as $\beta$.

Case 1:
Suppose $\beta = \pi \in \mathcal{CP}$ and $\pi \in \mathcal{MR}$ and $\pi \notin mref(D_3)$.

Then $\alpha = \pi \in \mathcal{CP}$ and since $mref(D_2) = mref(D_1)$, we have that $\pi \notin mref(D_1)$. Thus, $M(D_1)$ is true and $D_1 \xrightarrow{\pi \in \mathcal{CP}} D'_1$ where $\pi \in \mathcal{MR}$ and $\pi \notin mref(D_1)$ so by Definition 5.1.2, $M(D'_1)$ is true. Since, $M(D'_1)$ true and $D'_1 \equiv \pi D'_2$ by Equation 5.1, $M'(D'_2)$ is true. Hence, we have shown that if $M'(D_2)$ is true and $D_2 \xrightarrow{\pi \in \mathcal{CP}} D'_2$ where $\pi \in \mathcal{CP}$ and $\pi \in \mathcal{MR}$ and $\pi \notin mref(D_2)$ then $M'(D'_2)$ is true.

Case 2:
Suppose $\beta = \pi(x)$ where $\pi \in \mathcal{CP}$.

Then $\alpha = \pi(x)$ and by applying Definition 5.1.2 we have that $M$ is not a mediator predicate. This is a contradiction of the hypothesis. Hence it is not the case that $D_2 \xrightarrow{\beta} D'_2$ and the case is true vacuously.

The proof of every other case $\alpha$ which satisfies the conditions in Definition 5.1.2 is similar to the above cases so that $M'(D'_2)$ holds can be derived or the case is true vacuously.

Thus $M'$ is a mediator predicate. \[\square\]

5.2 Constructing a mediator

In the previous section we identified the behaviour that $\pi$-calculus agents must have in order to operate in parallel with machines. We referred to these agents as mediators. The following two sections offer the basis for constructing mediators from the $\pi$-calculus syntax.
5.2.1 Mediators with parallel, hiding and infinite replication

We present three results which ensure that mediator properties are preserved through the introduction of hiding, parallel, and infinite replication operators.

The first result of this section shows that, restricting names from \((SN \cup CP)\) in a mediator produces another mediator. Its proof follows a similar pattern to that of Lemma 5.1.2 in which we construct a mediator predicate \(M'\) from the mediator predicate \(M\) which determines that \(D\) is a mediator and try to show that \(M'\) is a mediator predicate. As Figure 5.4 shows, the set of agents which satisfy \(M'\) contains the set of agents which satisfy \(M\). In addition, \(M'\) contains the set of agents from \(M\) each constructed with \((\nu a)\) in front, for any \(a \in (SN \cup CP)\). This is because, as the diagram shows, an agent \(D_2\) from \(M'\) can execute a \(\beta\) action which essentially opens the bound name, so the resulting agent is \(D_1\), an agent for which \(M\) holds (rule OPEN 33). In all other transitions (resulting from rule RES) the name remains hidden thus the resulting agent \(D_2\) remains in \(M'\).

**Lemma 5.2.1.** If \(D\) is a mediator then for any \(a \in (SN \cup CP)\), \((\nu a)(D)\) is a mediator.

**Proof.** Assume that \(D\) is a mediator, and \(a \in (SN \cup CP)\). Then there exists a mediator predicate \(M\) such that \(M(D)\) holds.

We want to find a mediator predicate \(M'\) such that \(M'(\nu a)(D))\) holds.

Consider the following \(M'\) for any \(D_2\) and \(a \in (SN \cup CP)\).

\[
M'(D_2) \equiv (M(D_2)) \lor \exists D_1. M(D_1) \land (\nu a)(D_1) = D_2
\] (5.2)

We need to show that \(M'\) is a mediator predicate.
5.2. Constructing a mediator

Assume that \( D_2 \xrightarrow{\beta} D'_2 \land M'(D_2) \). Then, \( M(D_2) \) or \( \exists D_1. M(D_1) \land (\nu a)(D_1) = D_2 \) for \( a \in (SN \cup CP) \). We only need to consider the \( \exists D_1. M(D_1) \land (\nu a)(D_1) = D_2 \) branch because, we already know that if \( M(D_2) \) is true then \( D_2 \) is a mediator and \( M' \) is the same as \( M \).

Case 1:
Suppose \( \beta = \overline{c}(a) \) for some channel \( c \).

Then from transition rule \( OPEN \) we derive that if,

\[
D_2 \xrightarrow{\overline{c}(a)} D'_2 \\
\text{then} \\
D_1 \xrightarrow{\overline{a}} D'_1
\]

where \((\nu a)(D_1) = D_2 \) and \( D'_1 = D'_2 \).

Case 1.1:
Suppose \( c \in SN \) and \( a \in SN \).

Then, since \( M(D_1) \) is true and \( D_1 \xrightarrow{\overline{a}} D'_1 \) we have that \( M(D'_1) \) is true by Definition 5.1.2. Since, \( M(D'_1) \) is true and \( D'_1 = D'_2 \) we have that \( M'(D'_2) \) is true by Equation 5.2.

Thus, \( M'(D_2) \) is true and \( D_2 \xrightarrow{\overline{c}(a)} D'_2 \) for some \( c \in SN \) and \( a \in SN \) implies \( M'(D'_2) \) is true.

Case 1.2:
Suppose \( c \in SN \) and \( a \in CP \).

Then by applying Definition 5.1.2 we have that \( M \) is not a mediator predicate. This is a contradiction of the hypothesis. Hence it is not the case that \( D_2 \xrightarrow{\overline{c}(a)} D'_2 \) and the case is true vacuously.

Case 1.3:
Suppose \( c \in CP \) or \( c \in MR \).

follows a similar argument to case 1.2.

Case 2:
For any other \( \beta \) not covered by Case 1 above.
In all cases below, from transition rule $RES$, we have that if

$$D_2 \xrightarrow{\beta} D_2'$$

then

$$D_1 \xrightarrow{\beta} D_1'$$

where $(\nu \ a)(D_1) = D_2$ and $(\nu \ a)D_1' = D_2'$.

In addition we know that $\exists D_1. M(D_1) \land (\nu \ a)(D_1) = D_2$ from the hypothesis.

In all cases below by the side condition of $RES$ we have that $a \notin \{c, cp\}$.

Case 2.1:
Suppose $\beta = c \ v$ for some $c \in SN$ and $v \in SN$.

Then from Definition 5.1.2 applied to $D_1$ it follows that $M(D_1)$ is true. Since $(\nu \ a)D_1' = D_2'$ by Equation 5.2 we have that $M'(D_2')$ is true. Thus, we have that $M'(D_2)$ is true and $D_2 \xrightarrow{c \ u} D_2'$ for some $c \in SN$ and $v \in SN$ implies $M'(D_2')$ is true.

Case 2.2:
Suppose $\beta = \bar{c}v$ for some $c \in SN$.

Then from Definition 5.1.2 applied to $D_1$ follows that $v \in SN$ and $M(D_1)$ is true. Since $(\nu \ a)D_1' = D_2'$ by Equation 5.2 we have that $M'(D_2')$ is true. Thus, we have that $M'(D_2)$ is true and $D_2 \xrightarrow{\bar{c} \ u} D_2'$ for some $c \in SN$ implies $v \in SN$ and $M'(D_2')$ is true.

Case 2.3:
Suppose $\beta = \bar{c}(v)$ for some $c \in SN$ and $a \neq v$.

Similar to Case 2.2 above we can conclude that $M'(D_2)$ holds and $D_2 \xrightarrow{\bar{c} \ (v)} D_2'$ for some $c \in SN$ implies $v \in SN$ and $M'(D_2')$ holds.

Case 2.4:
Suppose $\beta = cp \ z$ for some machine reference $z \notin mref(D_2)$ and $cp \in CP$.

Then $z \notin mref(D_1)$ because $D_2 = (\nu \ a)(D_1)$. We have that, $M(D_1)$ is true and $D_1 \xrightarrow{cp \ z} D_1'$ hence from Definition 5.1.2, $M(D_1')$ is true. Since, $M(D_1')$ is true and $(\nu \ a)D_1' = D_2'$ by Equation 5.2 we have that $M'(D_2')$ is true. Thus, we have that $M'(D_2)$ is true and $D_2 \xrightarrow{cp \ z} D_2'$ for some machine reference $z \notin mref(D_2)$ and $cp \in CP$ implies $M'(D_2')$ is true.
5.2. Constructing a mediator

Case 2.5:
Suppose $\beta = \overline{c}\phi$ for some $c\phi \in CP$.

By Definition 5.1.2 we have that $M(D_1)$ is true and $D_1 \xrightarrow{\overline{c}\phi} D'_1$ implies $z \notin mref(D'_1)$ and $M(D'_1)$ is true. Then since, $(\nu a)D'_1 = D'_2$ and $z \notin mref(D'_1)$ we have that $z \notin mref(D'_2)$. Since, $M(D'_1)$ holds and $(\nu a)D'_1 = D'_2$ by Equation 5.2 we have that $M'(D'_2)$ holds. Thus, we have that $M'(D_2)$ is true and $D_2 \xrightarrow{\overline{c}\phi} D'_2$ implies $M'(D'_2)$ is true and $z \notin mref(D'_2)$.

Case 2.6:
Suppose $\beta = \overline{c}\phi(z)$ for any $z$.

Then from Definition 5.1.2 we have that $M(D_1)$ holds and $D_1 \xrightarrow{\overline{c}\phi(z)} D'_1$ implies false. So $M$ is not a mediator predicate which is a contradiction. Hence it is not the case that $\beta = \overline{c}\phi(z)$ in $D_2 \xrightarrow{\beta} D'_2$.

Case 2.7:
Suppose $\beta = z\ l$ for some machine reference $z$ for some valid label $l$.

Similar to Case 2.6.

Case 2.8:
Suppose $\beta = \overline{z}\ l$ for some machine reference $z$ for some valid label $l$.

Then from Definition 5.1.2 we have that $M(D_1)$ holds and $D_1 \xrightarrow{\overline{z}\ l} D'_1$ implies $M(D'_1)$ is true. Since, $M(D'_1)$ is true and $(\nu a)D'_1 = D'_2$ by Equation 5.2 we have that $M'(D'_2)$ is true. Thus, we have that $M'(D_2)$ is true and $D_2 \xrightarrow{\overline{z}\ l} D'_2$ implies $M'(D'_2)$ is true.

Case 2.9:
Suppose $\beta = \overline{z}(l)$ for some machine reference $z$ for some valid label $l$.

Similar to Case 2.6.

Case 2.10:
Suppose $\beta = \tau$.

From Definition 5.1.2 we have that $M(D_1)$ is true and $D_1 \xrightarrow{\tau} D'_1$ implies $M(D'_1)$ is true. We have that $M(D_1)$ is true and $D_1 \xrightarrow{\tau} D'_1$ hence $M(D'_1)$ is true. Since, $M(D'_1)$ is true and $(\nu a)D'_1 = D'_2$ by Equation 5.2 we have that $M'(D'_2)$ is true. Thus, we have shown that for any $D_2$ if $M'(D_2)$ is true and $D_2 \xrightarrow{\tau} D'_2$ then $M'(D'_2)$ holds.
Thus $M'$ is a mediator predicate. \qed

The second result shows that placing two mediators, that are disjoint on their machine references, in parallel yields a bigger mediator.

**Lemma 5.2.2.** For any mediator $D_1$ and mediator $D_2$ such that $\text{mref}(D_1) \cap \text{mref}(D_2) = \emptyset$ we have that $D_1 | D_2$ is a mediator.

**Proof.** Assume that $D_1$ is a mediator, $D_2$ is a mediator and $\text{mref}(D_1) \cap \text{mref}(D_2) = \emptyset$. Thus by Definition 5.1.3 we have that there are mediator predicates $M_1$ and $M_2$ such that, $M_1(D_1)$ and $M_2(D_2)$ hold. We need to find a mediator predicate $M'$ such that $M'(D_1 | D_2)$ holds.

Consider the following predicate for any $D_3$,

$$M'(D_3) \equiv \exists D_4, D_5. (M_1(D_4) \land M_2(D_5) \land 
\text{mref}(D_4) \cap \text{mref}(D_5) = \emptyset \land 
(D_3 = D_4 | D_5 \lor 
\exists a_1, \ldots, a_n. (a_1, \ldots, a_n \in SN \land 
D_3 = (\nu a_1, \ldots, a_n)(D_4 | D_5)))$$

(5.3)

Then the goal is to show that $M'$ is a mediator predicate.

Assume $M'(D_3)$ then,

$$\exists D_4, D_5. (M_1(D_4) \land M_2(D_5) \land 
\text{mref}(D_4) \cap \text{mref}(D_5) = \emptyset \land 
(D_3 = D_4 \lor D_5 \lor 
\exists a_1, \ldots, a_n. (a_1, \ldots, a_n \in SN \land 
D_3 = (\nu a_1, \ldots, a_n)(D_4 | D_5)))$$

Assume that $D_3 \xrightarrow{\beta} D_3'$ for any $\beta$.

Firstly, consider each $\beta$ with the $M_1(D_4) \land M_2(D_5) \land \text{mref}(D_4) \cap \text{mref}(D_5) = \emptyset \land D_3 = D_4 | D_5$ branch of 5.3.

(In the interest of brevity we consider only the transition rules $\text{PAR-L}$, $\text{COMM-L}$ and $\text{CLOSE-L}$ introduced in Table 3.2 on page 33 where the left agent initiates an interaction. The proof for $\text{PAR-R}$, $\text{COMM-R}$ and $\text{CLOSE-R}$ are symmetrical.)

**Case 1:**

Any $\beta$ such that $\beta \neq \tau$.  

5.2. Constructing a mediator

From rule $\text{PAR-L}$ we can infer that if,

$$D_4 \mid D_5 \xrightarrow{\beta} D'_4 \mid D_6$$

where $D'_4 = D'_4 \mid D_5$ then,

$$D_4 \xrightarrow{\beta} D'_4$$

Case 1.1:
Suppose $\beta = cp \ z$ for some control point $cp$ and $z \notin mref(D_3)$.

Since $z \notin mref(D_3)$ and $D_3 = D_4 \mid D_5$ we have that $z \notin mref(D_4)$ and $z \notin mref(D_5)$. We have that $mref(D_4) \cap mref(D_5) = \emptyset$ and $z \notin mref(D_5)$ and by Proposition 5.1.2 that $mref(D_4') \subseteq (mref(D_4) \cup \{z\})$ hence $mref(D_4') \cap mref(D_5) = \emptyset$. We have that $M_1(D_4)$ is true, and $D_4 \xrightarrow{cp \ z} D'_4$ for some $cp \in CP$ and $z \notin mref(D_4)$ hence by Definition 5.1.2, it is the case that $M_1(D_4')$ is true. Since $D'_4 = D'_4 \mid D_5$ by Equation 5.3 we have that $M'(D'_3)$ is true.

Thus, if $M'(D_3)$ is true and $D_3 \xrightarrow{cp \ z} D'_3$ or some $cp \in CP$ and $z \notin mref(D_3)$ then $M'(D'_3)$ is true.

Case 1.2:
Suppose $\beta = \overline{cp} \ z$ for some control point $cp$ and $z \in mref(D_3)$.

Since $mref(D_4) \cap mref(D_5) = \emptyset$ and $D_4 \xrightarrow{\overline{cp} \ z} D'_4$ we have that $z \notin mref(D_5)$. By Proposition 5.1.2 we have that $mref(D_4') \subseteq mref(D_4) \setminus \{z\}$, hence $mref(D_4') \cap mref(D_5) = \emptyset$. We have that $M_1(D_4)$ is true, and $D_4 \xrightarrow{\overline{cp} \ z} D'_4$ for some $cp \in CP$ hence by Definition 5.1.2, $M_1(D_4')$ is true and $z \notin mref(D_4')$. Since, $D'_4 = D'_4 \mid D_5$, $M_1(D_4')$ and $M_2(D_5)$ hold, and $mref(D_4') \cap mref(D_5) = \emptyset$ by Equation 5.3 we have that $M'(D'_3)$ holds. Since, $z \notin mref(D_4)$ and $z \notin mref(D_4')$ then $z \notin mref(D_5)$.

Thus if $M'(D_3)$ is true and $D_3 \xrightarrow{\overline{cp} \ z} D'_3$ or some $cp \in CP$ then $z \notin mref(D'_3)$ and $M'(D'_3)$ is true.

Case 1.3:
In all other cases (except $\tau, cp \ z, \overline{cp} \ z$) assume that in each case $\beta$ matches one of the clauses in Definition 5.1.2. $M_1$ is a mediator predicate hence by Definition 5.1.2 we have that, $M_1(D_4)$ is true, and $D_4 \xrightarrow{\beta} D'_4$, implies $M_1(D'_4)$. We have that $M_1(D_4)$ is true, and $D_4 \xrightarrow{\beta} D'_4$ hence $M_1(D'_4)$ holds.

By proposition 5.1.2 we have that $mref(D_4') \subseteq mref(D_4)$ and $mref(D_4') \cap mref(D_5) = \emptyset$ hence $mref(D_4') \cap mref(D_5) = \emptyset$. Since, $D'_4 = D'_4 \mid D_5$, $M_1(D'_4)$ and $M_2(D_5)$ hold, and $mref(D_4') \cap mref(D_5) = \emptyset$ by Equation 5.3 we have that $M'(D'_3)$ is true.
Thus if \( M'(D_3) \) is true and \( D_3 \xrightarrow{\beta} D'_3 \) then \( M'(D'_3) \) holds.

Case 2:
Suppose \( \beta = \tau \).

(This case represents a communication between two sub-agents.)

Case 2.1:
From rule \( \text{COMM-L} \) we can infer that if,

\[
D_4 \quad D_5 \xrightarrow{\tau} D'_4 \quad D'_5
\]

where \( D'_3 = D'_4 \quad D'_5 \) then,

\[
D_4 \xrightarrow{x} y \quad D'_4 \quad D_5 \xrightarrow{x} y \quad D'_5
\]

for some \( \pi \)-calculus names \( x \) and \( y \).

Case 2.1.1:
Let \( x \in CP \) and \( y \in mref(D_4) \). Then we have that \( y \in MR \) and since \( mref(D_4) \cap mref(D_5) = \emptyset \) we have that \( y \notin mref(D_5) \).

We have that \( M_1(D_4) \) is true and \( D_4 \xrightarrow{x} y \quad D'_4 \quad D_5 \xrightarrow{y} y \quad D'_5 \) for \( x \in CP \) and \( y \in mref(D_4) \) thus by Definition 5.1.2 \( y \notin mref(D'_4) \) and \( M_1(D'_4) \) is true.

We have that \( M_2(D_5) \) is true and \( D_5 \xrightarrow{x} y \quad D'_5 \quad D_4 \xrightarrow{y} y \quad D'_4 \) for \( x \in CP \) and \( y \in MR \) and \( y \notin mref(D_5) \) thus by Definition 5.1.2 \( M_2(D'_5) \) is true.

By Proposition 5.1.2 we have that \( mref(D'_4) \subseteq mref(D_4) \setminus \{y\} \) and \( mref(D'_5) \subseteq mref(D_5) \cup \{y\} \). Thus since \( mref(D_4) \cap mref(D_5) = \emptyset \) we can conclude that \( mref(D'_4) \cap mref(D'_5) = \emptyset \).

\( D'_3 = D'_4 \quad D'_5 \), thus by Equation 5.3 we have that \( M'(D'_3) \) is true. So \( M'(D_3) \) is true and \( D_3 \xrightarrow{\tau} D'_3 \) and \( M'(D'_3) \) is true.

Case 2.1.2:
If \( x \) and \( y \) match with one of the branches in Definition 5.1.2 such that \( x \notin CP \) and \( y \notin MR \). Let, \( D_4 \xrightarrow{\alpha_1}, D'_4 \) and \( D_5 \xrightarrow{\alpha_2}, D'_5 \) so the following combinations are possible.

1. \( \alpha_1 = \overline{x} \quad y \) and \( \alpha_2 = x \quad y \).
5.2. Constructing a mediator

2. \( \alpha_1 = \pi l \) and \( \alpha_2 = x l \) where \( x \in M_R \) and \( l \) is an operation label for some machine.

The latter combination falls out by contradiction because we already have that \( M_2(D_5) \) is true and if \( D_5 \xrightarrow{\pi l} D'_5 \) for \( x \in M_R \) and label \( l \) then \( M_2 \) is not a mediator predicate by Definition 5.1.2. We assumed that \( M_2 \) is a mediator predicate hence it is not the case that \( D_5 \xrightarrow{\pi l} D'_5 \) for \( x \in M_R \).

In the latter case. We have that \( M_1(D_4) \) holds and \( D_4 \xrightarrow{\pi y} D'_4 \) for \( x \in SN \) and \( y \in SN \) thus by Definition 5.1.2 \( M_1(D'_4) \) is true.

We also have that \( M_2(D_5) \) holds and \( D_5 \xrightarrow{\pi y} D'_5 \) for \( x \in SN \) and \( y \in SN \) thus by Definition 5.1.2 \( M_2(D'_5) \) is true.

From Proposition 5.1.2 we have that \( \text{mref}(D'_4) \subseteq \text{mref}(D_4) \) and \( \text{mref}(D'_5) \subseteq \text{mref}(D_5) \). Thus, since \( \text{mref}(D_4) \cap \text{mref}(D_5) = \emptyset \) we can conclude that \( \text{mref}(D'_4) \cap \text{mref}(D'_5) = \emptyset \).

\( D'_3 = D'_4 \mid D'_5 \), thus by the definition for \( M'(D'_3) \) (in Equation 5.3) we have that \( M'(D'_3) \) is true. So \( M'(D_3) \) is true and \( D_3 \xrightarrow{\pi} D'_3 \) and \( M'(D'_3) \) is true.

Case 2.2:

From rule \textit{CLOSE - L} we can infer that if,

\[
D_4 \mid D_5 \xrightarrow{\pi}(v \ a)(D'_4 \mid D'_5)
\]

where \( D'_3 = D'_4 \mid D'_5 \) then,

\[
D_4 \xrightarrow{\pi(a)} D'_4 \land
D_5 \xrightarrow{a} D'_5
\]

for the following combinations of \( \pi \)-calculus names:

1. \( x \in SN \) and \( a \in SN \), or
2. \( x \in CP \) and \( a \in M_R \), or
3. \( x \in M_R \) and \( a \) is an operation label.

The latter two combinations fall out vacuously.

We already have that \( M_1(D_4) \) is true and if \( D_4 \xrightarrow{\pi(a)} D'_4 \) for \( x \in CP \) and \( a \in M_R \) then \( M_2 \) is not a mediator predicate by Definition 5.1.2. We assumed that \( M_2 \) is a mediator predicate hence it is not the case that \( D_4 \xrightarrow{\pi(a)} D'_4 \) for \( x \in CP \) and \( a \in M_R \).
Similarly, we already have that \( M_2(D_5) \) is true and if \( D_5 \xrightarrow{a} D'_5 \) for \( x \in MR \) and operation label \( a \), then \( M_2 \) is not a mediator predicate by Definition 5.1.2. We assumed that \( M_2 \) is a mediator predicate hence it is not the case that \( D_5 \xrightarrow{a} D'_5 \) for \( x \in MR \) and operation label \( a \).

Thus we only have to check the foremost combination. We have that \( M_1(D_4) \) is true and \( D_4 \xrightarrow{(a)} D'_4 \) for \( x \in SN \) and \( a \in SN \) thus by Definition 5.1.2 \( M_1(D'_4) \) is true.

We also have that \( M_3(D_5) \) holds and \( D_5 \xrightarrow{a} D'_5 \) for \( x \in SN \) and \( a \in SN \) thus by Definition 5.1.2 \( M_2(D'_5) \) is true.

From Proposition 5.1.2 we have that \( mref(D'_4) \subseteq mref(D_4) \) and \( mref(D'_5) \subseteq mref(D_5) \). Thus, since \( mref(D_4) \cap mref(D_5) = \emptyset \) we can conclude that \( mref(D'_4) \cap mref(D'_5) = \emptyset \).

From Proposition 5.1.2, since \( M_1(D_4) \) and \( M_2(D_5) \) hold, and \( mref(D'_4) \cap mref(D'_5) = \emptyset \), \( a \in SN \) and \( D'_4 = (\nu a)(D'_4) \mid D'_5 \) we can conclude that \( \exists n \in \mathbb{N}, a_1 \ldots a_n \in SN \wedge D'_4 = (\nu a_1, \ldots, a_n)(D'_4 \mid D'_5) \) and correspondingly, by Equation 5.3, that \( M'(D'_3) \) holds.

Thus \( M'(D_3) \) is true and \( D_3 \xrightarrow{T} D'_3 \) implies that \( M'(D'_3) \).

Now we consider each \( \beta \) with the second branch of 5.3,

\[
M_1(D_4) \wedge M_2(D_5) \wedge mref(D_4) \cap mref(D_5) = \emptyset \wedge \exists a_1, \ldots, a_n(a_1, \ldots, a_n \in SN \wedge D_3 = (\nu a_1, \ldots, a_n)(D_4 \mid D_5))
\]

Case 3:

\( \beta = \overline{x}(a_1) \) for some channel \( x \in SN \). From transition rule OPEN we derive that if,

\[
D_3 \overrightarrow{\overline{x}(a_1)} D'_3
\]

then

\[
(\nu a_2, \ldots, a_n)(D_4 \mid D_5) \xrightarrow{a_1} D'_3
\]

where \( D'_3 = (\nu a_2, \ldots, a_n)(D'_4 \mid D_5) \).

Case 3.1: Let \( n = 1 \), so from above,

\[
(D_4 \mid D_5) \xrightarrow{a_1} D'_5
\]

then \( D'_3 = D'_4 \mid D_5 \).
5.2. Constructing a mediator

From PAR-L we have that $D_4 \xrightarrow{a_1} D'_4$.

Thus, $x \in SN$, $a_1 \in SN$ and since $M(D_4)$ is true and $D_4 \xrightarrow{a_1} D'_4$ we have that $M(D'_4)$ is true by Definition 5.1.2. From Proposition 5.1.2 we have that $\text{mref}(D'_4) \subseteq \text{mref}(D_4)$ and since $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$ we have that $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$. We already have that $M(D_5)$ is true. Thus $M(D'_4)$ and $M(D_5)$ hold, and $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$ and $D'_3 = D'_4 \mid D_5$. Hence by Equation 5.3, $M(D'_3)$ is true.

Case 3.2: For any $n > 1$, by applying rule $RES$, $n - 1$ times and then by PAR-L we can deduce that,

$$D_4 \xrightarrow{a_1} D'_4$$

Thus, $x \in SN$, $a_1 \in SN$ and since $M(D_4)$ is true and $D_4 \xrightarrow{a_1} D'_4$ we have that $M(D'_4)$ is true by Definition 5.1.2. From Proposition 5.1.2 we have that $\text{mref}(D'_4) \subseteq \text{mref}(D_4)$ and since $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$ we have that $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$. We already have that $M(D_5)$ is true. Thus $M(D'_4)$ and $M(D_5)$ hold, and $\text{mref}(D'_4) \cap \text{mref}(D_5) = \emptyset$ and $D'_3 = (\nu a_2, \ldots, a_n)(D'_4 \mid D_5)$. So we can conclude that $\exists m \in \mathbb{N}, a_1, \ldots, a_m \in SN \land D'_3 = (\nu a_1, \ldots, a_m)(D'_4 \mid D_5)$ and correspondingly, by Equation 5.3, we have that $M'(D'_3)$ is true.

Thus, $M'(D_4)$ is true and $D_3 \xrightarrow{a_1} D'_3$ for some $x \in SN$ and $a_1 \in SN$ implies that $M'(D'_3)$ is true.

Case 4:

For any other $\beta$ not covered by Case 3 above. Assume that in each case $\beta$ matches one of the clauses of Definition 5.1.2. By applying rule $RES$, we have that if

$$D_3 \xrightarrow{\beta} D'_3$$

then

$$(\nu a_2, \ldots, a_n)(D_4 \mid D_5) \xrightarrow{\beta} (\nu a_2, \ldots, a_n)(Q)$$

where $D'_3 = (\nu a_1, \ldots, a_n)(Q)$ with the following possibilities for $Q$,

1. $Q = D'_4 \mid D_5$ or,
2. $Q = D'_4 \mid D'_5$ or,
3. $Q = (\nu a_{n+1})(D'_4 \mid D'_5)$ for some $a_{n+1}$.

Then for an arbitrary $n$, by applying $RES$, $n - 1$ times we can infer that,

$$(D_4 \mid D_5) \xrightarrow{\beta} Q$$

Then by the same argument as in Cases 1 and 2 we have that $M(D'_4)$ is true and (in correspondence with each $Q$ above) either,
1. \( \text{mref}(D'_3) \cap \text{mref}(D'_5) = \emptyset \) or,
2. \( \text{mref}(D'_4) \cap \text{mref}(D'_6) = \emptyset \land M(D'_6) \) or,
3. \( a_{n+1} \in SN, \text{mref}(D'_4) \cap \text{mref}(D'_5) = \emptyset \land M(D'_6) \)

Since \( D'_4 = (\nu \ a_1, \ldots \ a_n)(Q) \), by Equation 5.3 applied to each case above we deduce
that \( M(D'_3) \) holds.

Thus we have shown that for any \( D_3 \) if \( M'(D_3) \) is true and \( D_3 \xrightarrow{\beta} D'_3 \) then \( M'(D'_3) \) is
true. In particular if \( \beta = c_\beta \ z \) then \( M'(D_3) \) is true and \( D_3 \xrightarrow{\beta} D'_3 \) implies \( z \notin \text{mref}(D'_4) \)
and \( M'(D'_3) \) is true. Hence by Definition 5.1.2 \( M' \) is a mediator predicate. □

The final result presented in Lemma 5.2.3 shows that if a mediator has no machine
references in its set of free names then we can replicate it infinitely and retain mediator
properties over the resulting agent.

**Lemma 5.2.3.** If \( \text{mref}(D) = \emptyset \) and \( D \) is a mediator then \( !D \) is a mediator.

**Proof.** Consider the predicate,

\[
M'(D_2) = \exists D_1. (\text{D}_1 \text{ is a mediator}) \land D_2 \equiv_n D_1 \ ||D
\]

We need to show that \( M' \) is a mediator predicate.

Assume that \( M'(D_2) \) is true and \( D_2 \xrightarrow{\beta} D'_2 \) then according to the definition of \( M' \) in
Equation 5.4,

\[
\exists D_1. (\text{D}_1 \text{ is a mediator}) \land D_2 \equiv_n D_1 \ ||D
\]

Case 1: Suppose by PAR-L we conclude that \( D_1 \ ||D \xrightarrow{\beta} D'_1 \ ||D \) in which case \( D'_1 \) is a mediator.

Since, \( D'_2 \equiv_n D'_1 \ ||D \) by definition of \( M' \) in Equation 5.4 we have that \( M'(D'_2) \) holds.

Case 2: Suppose by COMM-L and REP-ACT we conclude that \( \beta = r \) and \( D_1 \ ||D \xrightarrow{\beta} D'_1 \ ||D' \ ||D \).

Observe that if \( \beta \) is a communication of standard names, \( D_1 \xrightarrow{uv} D'_1 \) and \( D \xrightarrow{uv} D' \) where \( \{x, v\} \subseteq SN \), then from \( \text{mref}(D) = \emptyset \) we have that \( \text{mref}(D') = \emptyset \) so
\( \text{mref}(D'_1) \cap \text{mref}(D') = \emptyset \).

If \( \beta \) is a communication of machine references, \( D_1 \xrightarrow{c_\beta v} D'_1 \) and \( D \xrightarrow{c_\beta z} D' \) where \( c_\beta \in CP \)
and \( z \in \mathcal{MR} \), then from \( \text{mref}(D) = \emptyset \) and the fact that \( D_1 \) is a mediator we have that
5.2. Constructing a mediator

z \notin mref(D'_1) and \{z\} = mref(D') so again we have that mref(D'_1) \cap mref(D') = \emptyset.

D'_1 is a mediator, D' is a mediator and mref(D'_1) \cap mref(D') = \emptyset, so we can apply Lemma 5.2.2 and conclude that \( (D'_1 | D') \) is a mediator.

Then because \( D'_2 \equiv \pi (D'_1 | D') \vartriangledown D \) we can apply definition of \( M' \) in Equation 5.4 and conclude that \( M'(D'_2) \) holds.

Case 3:
We identify that \( \beta = \tau \) by application of \( REP-ACT \) and \( COMM-R \). Proof similar to Case 2.

Case 4:
Suppose by \( REP-ACT \) and \( CLOSE - L \) we conclude that \( \beta = \tau \), \( D_1 | D \vartriangledown \beta (\nu a)(D'_1 | D') \vartriangledown D \)

\( D_1 \) is a mediator so \( a \in SN \). \( D'_1 \) is a mediator and \( D' \) is a mediator. We have that \( mref(D) = \emptyset \) so \( mref(D') = \emptyset \). Therefore, by Lemma 5.2.2, \( (D'_1 | D') \) is a mediator. Since \( a \in SN \) by Lemma 5.2.1, \( (\nu a)(D'_1 | D') \) is also a mediator.

Since \( D'_2 \equiv \pi (\nu a)(D'_1 | D') \vartriangledown D \) we can apply definition of \( M' \) in Equation 5.4 and conclude that \( M'(D'_2) \) holds.

Case 5:
We identify that \( \beta = \tau \) by application of \( REP-ACT \) and \( CLOSE-R \). Proof similar to Case 4.

Case 6:
Suppose by \( PAR-R \) we conclude that \( D_1 \vartriangledown \beta D_1 | Q \), where \( Q \) can be in one of the following forms;

Case 6.1:
Suppose \( Q \equiv \pi D' \vartriangledown D \). Then the proof is similar to case 1 due to \( REP-ACT \).

Case 6.2:
Suppose \( Q \equiv \pi (D' | D'') \vartriangledown D \). Then proof is similar to Case 2 due to \( REP-COMM \).

Case 6.3:
Suppose \( Q \equiv \pi (\nu a)(D' | D'') \vartriangledown D \). Then proof is similar to Cases 4 due to \( REP-CLOSE \).

Thus \( M' \) is a mediator predicate.
The agent 0 is a mediator so $M'(0 \parallel D)$ holds so $0 \parallel D$ is a mediator. We have that $0 \parallel D \equiv^x \parallel D$ so by Lemma 5.1.2, $\parallel D$ is a mediator. □

5.2.2 Sequential finite controllers

In this section we define the building blocks for mediators called sequential finite controllers. Their definition is given in terms of a type system because by doing so we are better equipped to handle substitutions. This is important because of the separation between standard names, control points and machine references.

We begin by disallowing the following types. Any super type $\sharp o V_M$ is disallowed because we do not wish to consider channels which transmit control points for reason pointed out in Section 5.1. Furthermore, the types $\sharp V_M$ and $i V_M$ are disallowed because mediators only ever output on $V_M$ carrying channels.

**Definition 5.2.1.** Any type $T$ such that $\sharp o V_M \leq T$ or $T \leq i V_M$ is not an allowed type for channels in a mediator.

Note that by the definition above, the type $\sharp\sharp o V_M$ is not allowed as it is of the form $\sharp T$ for a $T$ that is not allowed.

We achieve the separation of the name space $N$ from the previous section using the following definition.

**Definition 5.2.2.** Given a typing environment $\Gamma$ and some channel $a$,

1. if $a : T$ for any $T$, such that $\sharp V_M \leq T$ for some machine $M$ then $a \in CP$,
2. if $a : T$ for any $T$, such that $\sharp V_M \leq T$ for some machine $M$ then $a \in MR$,
3. if $a : T$ for any $T$ such that it is not the case that $\sharp V_M \leq T$ or $\sharp V_M \leq T$ then $a \in SN$.

When typing a given process the type of a particular channel in the environment $\Gamma$ determines if the channel is a control point, a machine reference or a standard name.

A sequential finite controller is the smallest set of agents that satisfies the following:

**Definition 5.2.3.** Sequential finite controller (SFC)

Assume $SC$ is an SFC,

1. $0$ is an SFC,
2. $a(w).SC$ is an SFC,
3. $\exists (w).SC$ is an SFC,
5.2. Constructing a mediator

4. \( \tau .SC \) is an SFC,

5. \([a = b]\tau .SC \) is an SFC (for any prefix \( \pi \) from items 2, 3, 4 above),

6. \((\nu \ x : T)(SC)\) is an SFC where it is not the case that \( T = oV_M \) or \( T = V_M \),

7. if \( SC_1 \) and \( SC_2 \) are SFCs then \((SC_1 + SC_2)\) is an SFC.

Note that in item 5 of Definition 5.2.3 the names in a match prefix are not of types \( oV_M \), hence by Definition 5.2.2 they are not machine references. This requirement comes from the previous section (see Lemma 5.1.1). Furthermore, item 6 of Definition 5.2.3 prohibits the hiding of machine references to prevent the possibility for expressing bound output of such channels which is not intended behaviour for mediators. Finally note that the standard naming convention for \( \pi \) processes, introduced in Section 3.1.1, applies to SFCs.

We specialise the typing system from Section 3.4 in the specific case of SFC agents, to allow us to type SFCs that are mediators.

Firstly, we introduce a function \( \text{filter} \) below.

**Definition 5.2.4.** Given a typing environment \( \Gamma \) and a name \( x \) such that \( x \in \text{supp}(\Gamma) \), \( \Gamma\text{ filter }x \) denotes the typing environment

\[
\{(i,j) \mid (i,j) \in \Gamma \land (i = x \Rightarrow \neg(\Gamma(i) \leq oV_M))\}
\]

for any machine \( M \).

The operator removes a name from \( \Gamma \) if it is of a machine reference type. Note that it leaves the environment unchanged if the name had a different type. Thus, if \( \Gamma \vdash x : oV_M \) then \( \Gamma\text{ filter }x \) is not defined on \( x \) however, if \( \Gamma \vdash x : T \) for a \( T \neq oV_M \) then \( \Gamma\text{ filter }x = \Gamma \).

Using the operator we override the rules \( T\text{-OutS} \) and \( T\text{-Mat} \) specifically for use with SFCs. With these new rules we meet the constraint that a machine reference must not appear in the continuation of a mediator after output and must not appear in match prefixes.

**Definition 5.2.5.**

\[
\frac{\Gamma \vdash a : oT \quad \Gamma \vdash w : T \quad \Gamma \text{ filter }w \vdash SC}{\Gamma \vdash \overline{a(w)}.SC} \quad T\text{-OutS}
\]

\[
\frac{\Gamma \vdash a : T \quad \Gamma \vdash b : T \quad \Gamma \vdash SC \quad T \neq oV_M}{\Gamma \vdash [a = b]SC} \quad T\text{-Mat}
\]
The following two results, Lemma 5.2.4 and Theorem 5.2.1, work towards the proof of Theorem 5.2.2 which justifies that we have a correct implementation of the mediator requirements of Definition 5.1.3.

Essentially these are modified versions of the substitution and subject reduction results in Section 3.4. The main modification is that, whenever we substitute \( v \) for \( x \) and \( x \) is a machine reference then we assume \( v \) is fresh. Similarly in subject reduction we assume machine references received through input are fresh. This fact would not be formally proved until Section 5.3 where machine/mediator systems are considered.

**Lemma 5.2.4. Substitution Lemma**

*Given a sequential finite controller \( SC \) if,*

1. \( \Gamma \vdash SC \),
2. \( \Gamma(x) = T \),
3. \( \Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin fn(SC)) \),

*then \( \Gamma \vdash SC\{v/x\} \).*

*Proof.* By induction on the depth of the derivation of \( SC \). Assume true for \( SC' \).

**Case 1:**
Consider rule \( T-Nil \). We have that \( \Gamma \vdash 0 \) and \( \Gamma \vdash 0\{v/x\} \) is the same as \( \Gamma \vdash 0 \) hence the case is complete.

**Case 2:**
Suppose we have that \( \Gamma \vdash a(w : S).SC', \Gamma(x) = T \), and
\( \Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin fn(a(w : S).SC')) \). We need to show that
\( \Gamma \vdash (a(w : S).SC')\{v/w\} \).

Note by the naming convention, \( w \neq x \) and \( w \neq v \) and \( w \) is not defined in \( \Gamma \).

By rule \( T-InpS \), we derive that \( \Gamma \vdash a : iS \), and \( \Gamma, w : S \vdash SC' \). \( (\Gamma, w : S)(x) = T \) because we have that \( \Gamma(x) = T \). Since we have that \( \Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin fn(SC')) \) by Weakening 3.4.3 we also have that \( \Gamma, w : S \vdash v : T \land ((T = oV_M \land v \notin fn(SC')) \Rightarrow true) \). Hence, we can apply the inductive hypothesis deducing that \( \Gamma, w : S \vdash SC'\{v/w\} \).

If \( a = x \), then \( T = iS \) then from \( \Gamma \vdash a : iS \) and Weakening lemma 3.4.3, we have that \( \Gamma, w : S \vdash a : iS \). We have that \( \Gamma \vdash v : iS \land (T = oV_M \land v \notin fn(SC)) \).

Thus from \( \Gamma, w : S \vdash SC'\{v/w\} \) and \( \Gamma, w : S \vdash v : iS \) by applying rule \( T-InpS \) down we have a derivation of \( \Gamma, w : S \vdash v(w : S).SC'\{v/w\} \). Since \( w \) is
not free in \( v(w : S).SC'(v/x) \) we can apply the Strengthening 3.4.2 and derive that 
\[ \Gamma \vdash v(w : S).(SC'(v/x)) \] 
which is the same as \( \Gamma \vdash (a(w : S).SC'(v/x)) \) since \( a = x \).

Suppose \( a \neq x \), since we already have that \( \Gamma \vdash a : iS \) we apply Weakening 
lemma 3.4.3 to derive \( \Gamma, w : S \vdash a : iS \). Then from \( \Gamma, w : S \vdash SC'(v/x) \) 
and \( \Gamma, w : S \vdash a : iS \) by applying rule \( T-InpS \) down we have a derivation of 
\( \Gamma, w : S \vdash a(w : S).SC'(v/x) \). Since \( w \) is not free in \( a(w : S).SC'(v/x) \) we 
can apply Strengthening 3.4.2 and derive that 
\[ \Gamma \vdash a(w : S).SC'(v/x) \] 
which is the same as \( \Gamma \vdash (a(w : S).SC'(v/x)) \) since \( a \neq x \).

Case 3:
Suppose we have that \( \Gamma \vdash \alpha(w).SC' \), and \( \Gamma(x) = T \), and \( \Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin fn(\alpha(w).SC')) \).

We need to show that \( \Gamma \vdash \alpha(w).SC'(v/x) \).

By rule \( T-OutS \) we can conclude that for some \( S \), \( \Gamma \vdash a : oS \) and \( \Gamma \vdash w : S \) and 
\( \Gamma \) \filter \( w \vdash SC' \).

(Now we must consider different cases of \( S \).)

Case 3.1:
Suppose \( S = oV_M \) so \( \Gamma \vdash w : oV_M \). From \( \Gamma \) \filter \( w \vdash SC' \) and \( \Gamma \vdash w : oV_M \) we 
conclude that \( w \notin fn(SC') \) hence also from Weakening 3.4.3 \( (\Gamma \) \filter \( w), w : oV_M \vdash SC' \). Also note that \( (\Gamma \) \filter \( w), w : oV_M = \Gamma \).

Case 3.1.1:
Suppose \( z = w \) so we need to show that \( z \) is not in the free names of \( SC' \). From \( \Gamma(z) = T \) and \( \Gamma \vdash w : oV_M \) we conclude that \( T = oV_M \). Since \( T = oV_M \) and \( (T = oV_M \Rightarrow v \notin fn(\alpha(w).SC')) \) we have that \( v \notin fn(SC') \) and \( v \neq a \). Thus \( T = oV_M \) and \( v \notin fn(SC') \) hence \( T = oV_M \Rightarrow v \notin fn(SC') \).

Furthermore, \( \Gamma \vdash v : oV_M \) and we have that \( v \neq w \) and \( (\Gamma \) \filter \( w), w : oV_M = \Gamma \) hence we have that \( (\Gamma \) \filter \( w), w : oV_M \vdash v : oV_M \). \( \Gamma(z) = oV_M \) and \( x = w \) and 
\( (\Gamma \) \filter \( w), w : oV_M = \Gamma \) hence \( (\Gamma \) \filter \( w), w : oV_M(x) = oV_M \). Thus we can 
apply the inductive hypothesis and derive that \( (\Gamma \) \filter \( w), w : oV_M \vdash SC'(v/x) \).

Furthermore, \( w \notin fn(SC'), x = w, \) and \( v \notin fn(SC') \) hence \( w \notin fn(SC'(v/x)) \) and 
\( v \notin fn(SC'(v/x)) \).

Since \( ((\Gamma \) \filter \( w), w : oV_M) \vdash SC'(v/x) \) and \( (\Gamma \) \filter \( w), w : oV_M = \Gamma \) we have that 
\( \Gamma \vdash SC'(v/x) \). Since \( v \notin fn(SC'(v/x)) \) and \( \Gamma \vdash v : oV_M \), \( \Gamma \) \filter \( v \vdash SC'(v/x) \).

Hence \( \Gamma \vdash a : oV_M \), and \( \Gamma \vdash v : oV_M \), and \( \Gamma \) \filter \( v \vdash SC'(v/x) \) so by applying 
\( T-OutS \) down we have a derivation of \( \Gamma \vdash \alpha(v).SC'(v/x) \) which is the same as
$\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$ where $x = w$.

Case 3.1.2:
Suppose $x \neq w$.

(it is still possible that $v$ and $x$ are machine references.)

Case 3.1.2.1:
Suppose $T = oV_M$, then $v \neq w$, $v \neq a$ and $v \notin \text{fn}(SC')$ from the hypothesis $\Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin \text{fn}(\overline{a}(w).SC'))$.

Hence we have that $\Gamma$ filter $w \vdash v : T \land (T = oV_M \Rightarrow v \notin \text{fn}(SC'))$. We also have that $\Gamma$ filter $w \vdash SC'$ above and since $(\Gamma$ filter $w)(x) = T$ since $\Gamma(x) = T$ and $x \neq w$. Hence we can apply the inductive hypothesis and derive that $\Gamma$ filter $w \vdash SC'$\(^{\{v/x\}}\). Clearly $a \neq x$ because the types within this case are different ($a : oS$ and $x : S$), hence we can apply rule $T$-OutS down and derive $\Gamma \vdash \overline{a}(w).SC'\(^{\{v/x\}}\)$ which is the same as $\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$.

Case 3.1.2.2:
Suppose $T \neq oV_M$ and $S = oV_M$, then again $v \neq w$ since $\Gamma \vdash v : T$ and $\Gamma \vdash w : oV_M$. Hence, we still have that $\Gamma$ filter $w \vdash v : T \land (T = oV_M \Rightarrow v \notin \text{fn}(SC'))$.

Hence, as above we can apply the inductive hypothesis and derive that $\Gamma$ filter $w \vdash SC'$\(^{\{v/x\}}\).

If $x = a$, then $T = oS$. So we have that $\Gamma$ filter $w \vdash v : T$, so $\Gamma \vdash v : oS$, hence we can apply $T$-OutS down and derive that $\Gamma \vdash \overline{v}(w).SC'\(^{\{v/x\}}\)$. Since here $x = a$, we have a derivation of $\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$.

If $x \neq a$ then we can apply rule $T$-OutS down and derive that $\Gamma \vdash \overline{a}(w).SC'\(^{\{v/x\}}\)$ which is the same as $\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$ since $x \neq a$.

Case 3.1.3:
Suppose $T \neq oV_M$ and $S \neq oV_M$, then clearly $\Gamma$ filter $w = \Gamma$ by definition of filter 5.2.4. Hence, we have that $\Gamma \vdash SC'$, and $\Gamma(x) = T$, and $\Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin \text{fn}(SC'))$. So we can apply the inductive hypothesis and derive that $\Gamma \vdash SC'$\(^{\{v/x\}}\).

If $x = w$ then $S = T$, so $\Gamma \vdash v : S$, and applying $T$-OutS down we have the derivation $\Gamma \vdash \overline{a}(w).SC'\(^{\{v/x\}}\)$ which is the same as $\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$ since $x = w$.

If $x = a$ then $oS = T$, so $\Gamma \vdash v : oS$, and applying $T$-OutS down we have the derivation $\Gamma \vdash \overline{v}(w).SC'\(^{\{v/x\}}\)$ which is the same as $\Gamma \vdash (\overline{a}(w).SC')^{\{v/x\}}$ since
5.2. Constructing a mediator

If \( x \neq w \) and \( x \neq a \) then we can apply \( T{-}\text{OutS} \) down directly and obtain a derivation of \( \Gamma \vdash (\overline{a}(w).SC'(v/x)) \) which is the same as \( \Gamma \vdash (\overline{a}(w).SC')\{v/x\} \) since \( x \neq w \) and \( x \neq a \).

Case 4:
Suppose we have that \( \Gamma \vdash [a = b]SC' \), \( \Gamma(x) = T \), \( \Gamma \vdash v : T \wedge (T = oV_M \Rightarrow v \notin fn([a = b]SC')) \). We need to show that \( \Gamma \vdash ([a = b]SC')\{v/x\} \).

By applying rule \( T{-}\text{Mat} \) we have that \( \Gamma \vdash a : S \), and \( \Gamma \vdash b : S \), and \( \Gamma \vdash SC' \) for some type \( S \). We have that \( \Gamma \vdash v : T \wedge (T = oV_M \Rightarrow v \notin fn(SC')) \) from the hypothesis so we can apply the inductive hypothesis and derive that \( \Gamma \vdash SC'\{v/x\} \).

If \( a = x \) or \( b = x \) then \( T = \#S \). Thus we have that \( \Gamma \vdash v : \#S \) hence by applying \( T{-}\text{Mat} \) down we have a derivation of \( \Gamma \vdash [v = b](SC')\{v/x\} \), or \( \Gamma \vdash [a = v](SC')\{v/x\} \) respectively. In each case this is equivalent to \( \Gamma \vdash ([a = b]SC')\{v/x\} \).

Case 5:
Suppose we have that \( \Gamma \vdash \tau.SC' \), \( \Gamma(x) = T \), \( \Gamma \vdash v : T \wedge (T = oV_M \Rightarrow v \notin fn(\tau.SC')) \). We need to show that \( \Gamma \vdash (\tau.SC')\{v/x\} \).

By applying rule \( T{-}\text{TAU} \) we have that \( \Gamma \vdash SC' \). We have that \( \Gamma \vdash v : T \wedge (T = oV_M \Rightarrow v \notin fn(SC')) \) from the hypothesis so we can apply the inductive hypothesis and derive that \( \Gamma \vdash SC'\{v/x\} \).

Since we have \( \Gamma \vdash SC'\{v/x\} \) we can apply rule \( T{-}\text{TAU} \) down and get the derivation, \( \Gamma \vdash \tau.(SC')\{v/x\} \) which is the same as \( \Gamma \vdash (\tau.SC')\{v/x\} \).

Case 6:
Suppose we have that \( \Gamma \vdash (\nu y : S)(SC') \) such that it is not the case that \( S = oV_M \), \( \Gamma(x) = T \), \( \Gamma \vdash v : T \wedge (T = oV_M \Rightarrow v \notin fn((\nu y : S)(SC')))) \). We need to show that \( \Gamma \vdash ((\nu y : S)(SC'))\{v/x\} \).

Note by the standard naming convention it is the case that \( x \neq y \), and \( v \neq y \), and \( y \) is not defined in \( \Gamma \).

By rule \( T{-}\text{RES} \) we have that \( \Gamma, y : S \vdash SC' \). By Weakening 3.4.3 we derive that \( \Gamma, y : S \vdash v : T \) and since \( (T = oV_M \Rightarrow v \notin fn((\nu y : S)(SC')))) \) and \( v \neq y \) we also have that \( (T = oV_M \Rightarrow v \notin fn(SC')) \). Since \( \Gamma(x) = T \) we have that \( (\Gamma, y : S)(x) = T \). Hence we can apply the inductive hypothesis and derive that \( \Gamma, y : S \vdash SC'\{v/x\} \).
Since we have that \( \Gamma, y : S \vdash SC'(v/x) \), by applying rule \( T-RES \) down we have a derivation on \( \Gamma \vdash (\nu y : S)(SC'(v/x)) \) which is the same as \( \Gamma \vdash ((\nu y : S)(SC'))(v/x) \).

Case 7:
Suppose we have that \( \Gamma \vdash SC_1 + SC_2 \), \( \Gamma(x) = T \), \( \Gamma \vdash v : T \land (T = oV_M \Rightarrow v \notin fn(SC_1 + SC_2)) \). We need to show that \( \Gamma \vdash (SC_1 + SC_2)(v/x) \).

By applying rule \( T-SUM \) we have that \( \Gamma \vdash SC_1 \) and \( \Gamma \vdash SC_2 \). Hence we can apply the inductive hypothesis to both agents and derive that \( \Gamma \vdash SC_1(v/x) \) and \( \Gamma \vdash SC_2(v/x) \). Since we have that \( \Gamma \vdash SC_1(v/x) \) and \( \Gamma \vdash SC_2(v/x) \) we can apply \( T-SUM \) down and derive that \( \Gamma \vdash SC_1(v/x) + SC_2(v/x) \) which by definition of substitution is the same as \( \Gamma \vdash (SC_1 + SC_2)(v/x) \). \( \square \)

Lemma 4.4.1 shows that typing derivations are preserved by substitution. This property will be used in Theorem 5.2.2 which shows that a well typed \( SFC \) is a mediator. To obtain that result we also need to demonstrate the Subject Reduction Theorem which would give us the ability to reason about the types of processes as they evolve.

**Theorem 5.2.1.** Subject reduction for controllers If \( \Gamma \vdash SC \), with \( \Gamma \) closed, and \( SC \xrightarrow{\alpha} SC' \),

1. If \( \alpha = \tau \) then \( \Gamma \vdash SC' \),
2. If \( \alpha = a \cdot v \) then there is a \( T \) such that,
   (a) \( \Gamma \vdash a : iT \),
   (b) if \( \Gamma \vdash v : T \land T \neq oV_M \) then, \( \Gamma \vdash SC' \),
   (c) if \( \Gamma \vdash v : oV_M \land v \notin fn(SC) \) then \( \Gamma \vdash SC' \).
3. If \( \alpha = (\nu \bar{x} : \bar{S})\bar{a} \cdot v \) then there is a \( T \) such that,
   (a) \( \Gamma \vdash a : oT \),
   (b) \( \Gamma, \bar{x} : \bar{S} \vdash v : T \),
   (c) \( (\Gamma, \bar{x} : \bar{S}) \) filter \( v \vdash SC' \)
   (d) each component of \( \bar{S} \) is a link type.

**Proof.** By induction on \( SC \xrightarrow{\alpha} SC' \). Assume true for \( SC' \).

Case 1:
Suppose \( \Gamma \vdash \tau.SC' \) and \( \tau.SC' \xrightarrow{\tau} SC' \) (by rule \( Tau \)). By rule \( T-TAU \) if \( \Gamma \vdash \tau.SC' \) then \( \Gamma \vdash SC' \). We have \( \Gamma \vdash \tau.SC' \) hence we have \( \Gamma \vdash SC' \).

Case 2:
Suppose \( \Gamma \vdash a(w : T).SC' \), and \( a(w : T).SC' \xrightarrow{a,w} SC'(v/w) \) (by rule \( Inp \)) for some
5.2. Constructing a mediator

$v \neq w$ and $w \notin \Gamma$ by naming convention. By rule, $T.-\text{Inp}\mathcal{S}$ we infer that $\Gamma \vdash a : iT$ and $\Gamma, w : T \vdash SC'$.

We need to show that;

If $\Gamma \vdash v : T \land T \neq oV_M$ then $\Gamma \vdash SC'$ and,

if $\Gamma \vdash v : oV_M \land v \notin \text{fn}(SC)$ then $\Gamma \vdash SC'$. Since $\Gamma \vdash v : T$, by Weakening 3.4.3, we have that $\Gamma, w : T \vdash v : T$. We also have that $(\Gamma, w : T)(w) = T$.

Since $\Gamma, w : T \vdash SC'$, if $\Gamma \vdash v : T \land T \neq oV_M$ then by the Substitution lemma 5.2.4 we infer $\Gamma, w : T \vdash SC'(v/w)$. If $\Gamma \vdash v : oV_M \land v \notin \text{fn}(SC)$ then, we have that $T = oV_M \rightarrow v \notin \text{fn}(SC)$. Hence we can apply the Substitution lemma as above and again infer $\Gamma, w : T \vdash SC'(v/w)$. Then in both cases, since $w$ is no longer free in the names of $SC'(v/w)$, by Strengthening we can infer that $\Gamma \vdash SC'(v/w)$.

Case 3:

Suppose $\Gamma \vdash \overline{a}v.SC'$, and $\overline{a}v.SC' \xrightarrow{\nu \overline{\Sigma} ; \overline{S} ; \overline{v}} SC'$ (by rule Out) where $\overline{\Sigma}$ and $\overline{S}$ are empty lists.

By rule $T.-\text{Out}\mathcal{S}$ we derive that for some $T$, $\Gamma \vdash a : oT$, $\Gamma \vdash v : T$, and $\Gamma \text{filter} v \vdash SC'$ which is the same as $(\Gamma, \overline{\Sigma} : \overline{S}) \text{filter} v \vdash SC'$ for the empty lists $\overline{\Sigma}$ and $\overline{S}$.

Case 4:

Suppose, $\Gamma \vdash (\nu v : T)(SC)$ and $(\nu v : T)(SC) \xrightarrow{\nu \overline{\Sigma} ; \overline{S} ; \overline{v} ; T} SC'$ where by Definition 5.2.3, it is not the case that $oV_M$ is in the list $\overline{S}$ and $T \neq oV_M$. By typing rule $T.-\text{RES}$ we infer that $\Gamma, v : T \vdash SC$.

By rule OPEN we infer that $SC \xrightarrow{\nu \overline{\Sigma} ; \overline{S} ; \overline{v}} SC'$. We also have that $(\Gamma, v : T) \vdash SC$ so, by the inductive hypothesis we have $(\Gamma, v : T) \vdash a : oT$, and $(\Gamma, v : T), \overline{\Sigma} : \overline{S} \vdash v : T, ((\Gamma, v : T), \overline{\Sigma} : \overline{S}) \text{filter} v \vdash SC'$. By side condition of OPEN we have that $a \neq v$ hence if $\Gamma, v : T \vdash a : oT$ then it must be the case that $\Gamma \vdash a : oT$.

Hence we have that if $\Gamma \vdash (\nu v : T)(SC)$ and $(\nu v : T)(SC) \xrightarrow{\nu \overline{\Sigma} ; \overline{S} ; \overline{v} ; T} SC'$ there is a $T$ such that $\Gamma \vdash a : oT$, and $(\Gamma, v : T), \overline{\Sigma} : \overline{S} \vdash v : T$, and $(\Gamma, v : T), \overline{\Sigma} : \overline{S}) \text{filter} v \vdash SC'$.

Case 5:

Suppose we have that $\Gamma \vdash [a = a]SC$ and $[a = a]SC \xrightarrow{\alpha} SC'$ then by rule Match we infer that $SC \xrightarrow{\alpha} SC'$. By rule $T.-\text{Mat}$ we have that for some $T$, $\Gamma \vdash a : \sharp T$ and $\Gamma \vdash SC$. Hence we can apply the inductive hypothesis in each case of $\alpha$. 

Case 6:
Suppose we have that $\Gamma \vdash SC_1 + SC_2$, and $SC_1 + SC_2 \xrightarrow{\alpha} SC'_1$ then by rule SUM-\text{L}, we have that $SC_1 \xrightarrow{\alpha} SC'_1$. (For brevity we consider only SUM-\text{L} as the case for SUM-\text{R} is similar.) From rule T-SUM we have that $\Gamma \vdash SC_1$ hence case follows from inductive hypothesis in each case of $\alpha$.

Case 7:
Suppose, we have that $\Gamma \vdash (\nu x : T)(SC)$ and $(\nu x : T)(SC) \xrightarrow{\alpha} (\nu x : T)(SC')$ then by rule RES, $SC \xrightarrow{\alpha} SC'$. By side condition of RES, the subject of $\alpha$ is not equal to $z$, and $\Gamma$ is not defined on $x$ by naming convention, and it is not the case that $T = oV_M$ by Definition 5.2.3.

By typing rule T-RES we have that $\Gamma, x : T \vdash SC$. Hence case follows from inductive hypothesis in each case of $\alpha$ where as $z$ is not in the free names of $(\nu x : T)(SC')$ we can apply Strengthening 3.4.2. \hfill \Box

We use Theorem 5.2.1 in the following proof to show that a well typed sequential finite controller is a mediator.

**Theorem 5.2.2.** Given an SFC agent, SC, if there is a $\Gamma$ closed with respect to SC

$$ \Gamma \vdash SC $$

then SC is a mediator.

**Proof.** Suppose there is a $\Gamma$ closed with respect to SC, and $\Gamma \vdash SC$, and SC is an SFC. Then if SC is a mediator, by Definition 5.1.3 there must be a mediator predicate $M$ such that $M(SC)$ holds.

We construct a predicate $M$ as follows. For any $SC$

$$ M(SC) \triangleq \exists \Gamma. \Gamma \vdash SC $$

We need to show that $M$ is a mediator predicate.

Assume $M(SC)$ and $SC \xrightarrow{\beta} SC'$ then $\exists \Gamma. \Gamma \vdash SC$. By case analysis on $\beta$ we must show that in each case the relevant implication of Definition 5.1.2 holds.

Case 1:
If $\beta = a \ w$ then by Subject Reduction 5.2.1 there is a $T$ such that $\Gamma \vdash a : iT$, if $\Gamma \vdash w : T$ then $\Gamma \vdash SC'$ and if $\Gamma \vdash w : oV_M \land w \notin fn(SC)$ then $\Gamma \vdash SC'$.

For any $T$ such that it is not the case that $\|V_M \leq T$ or $\|V_M \leq T$ we have that $\Gamma \vdash a : iT$, if $\Gamma \vdash w : T$ then $\Gamma \vdash SC'$. Hence by Definition 5.2.2 we deduce $a \in SN$ and $w \in SN$. We also know $\Gamma \vdash SC'$ so $M(SC')$ is true. Thus we have that $M(SC)$ holds, and $SC \xrightarrow{\alpha} SC'$, and $a \in SN$, and $w \in SN$, and $M(SC')$ hence, the
5.2. Constructing a mediator

implication of Definition 5.1.2 holds.

We need to consider two cases for $T$.

If $T = oV_M$ such that $\Gamma a : ioV_M$ and if $\Gamma \vdash w : oV_M \land w \notin fn(SC)$ then $\Gamma \vdash SC'$.

By Definition 5.2.2 we have that $a \in CP$ and $w \in MR$. Since $w \notin fn(SC)$ we have that $w \notin mref(SC')$ by definition of $mref$. We have that $\Gamma \vdash SC'$ hence $M(SC')$ is true by Equation 5.5. Thus, we have that $M(SC)$ is true, and $SC \overset{a\rightarrow w}{\longrightarrow} SC'$, and $a \in CP$, and $w \in MR$, and $w \notin fn(SC)$, and $M(SC')$ is true hence, the implication of Definition 5.1.2 holds.

If $T = V_M$ then we violate Definition 5.2.1 with $\Gamma \vdash a : iV_M$ because $iV_M$ is not allowed. Hence, it is not the case that $\beta = a w$ for an $a$ such that $\Gamma \vdash a : iV_M$.

Hence, the implication from Definition 5.1.2 holds vacuously.

Case 2:

$\beta = \bar{a} w$ then by Subject Reduction 5.2.1 there is a $T$ such that, $\Gamma \vdash a : oT$, $\Gamma \vdash w : T$, $\Gamma \text{filter } w \vdash SC'$. (Here the lists $\bar{x}$ and $\bar{s}$ are empty hence we have omitted them for clarity.)

For any $T$ such that it is not the case that $\|V_M \leq T$ or $\|V_M \leq T$, by Definition 5.2.2 we have that $a \in SN$ and $w \in SN$. Since, $\Gamma \text{filter } w \vdash SC'$ by Equation 5.5 we have that $M(SC')$ is true. Thus, $M(SC)$ is true, and $SC \overset{a\rightarrow w}{\longrightarrow} SC'$, and $a \in SN$, and $w \in SN$, and $M(SC')$ is true hence, the implication of Definition 5.1.2 holds.

There are two cases to consider.

If $T = oV_M$ such that $\Gamma \vdash a : oV_M$, $\Gamma \vdash w : oV_M$, by Definition 5.2.2 we have that $a \in CP$ and $w \in MR$. Since $\Gamma \text{filter } w \vdash SC'$ and $\Gamma \vdash w : oV_M$ by definition of $\text{filter}$ and Lemma 5.2.4, $w \notin fn(SC')$. From definition of $mref$, since $w \notin fn(SC')$ we have that $w \notin mref(SC')$. We have that $\Gamma \text{filter } w \vdash SC'$ hence $M(SC')$. Thus, $M(SC)$ is true, and $SC \overset{a\rightarrow w}{\longrightarrow} SC'$, and $a \in CP$, and $w \in MR$, and $M(SC')$ is true hence, the implication of Definition 5.1.2 holds.

If $T = V_M$ such that $\Gamma \vdash a : V_M$, $\Gamma \vdash w : V_M$, then by Definition 5.2.2 we have that $a \in MR$. $\Gamma \vdash w : V_M$ hence $w$ is a valid operation label. We have that $\Gamma \text{filter } w \vdash SC'$ hence $M(SC')$. Thus, $M(SC)$ is true, and $SC \overset{a\rightarrow w}{\longrightarrow} SC'$, and $a \in MR$, and $w$ is a valid operation label, and $M(SC')$ is true hence, the implication of Definition 5.1.2 holds.

Case 3:

$\beta = (\nu w : S)\bar{a}w$, (which is equivalent to $\bar{a}(w)$). By Subject Reduction 5.2.1 there is
a $T$ such that, $\Gamma \vdash a : oT$, and $\Gamma, w : S \vdash w : T$; $(\Gamma, w : S)$ filter $w \vdash SC'$.

For any $T$ such that it is not the case that $\dual V_M \leq T$ or $\dual V_M \leq T$, by Definition 5.2.2 we have that $a \in SN$ and $w \in SN$. Since, $\Gamma$ filter $w \vdash SC'$ by Equation 5.5 we have that $M(SC')$ is true. Thus, $M(SC)$ is true, and $SC \xrightarrow{(w)} SC'$, and $a \in SN$, and $w \in SN$, and $M(SC')$ is true hence, the implication of Definition 5.1.2 holds.

By Definition 5.2.3 there is no $SC$ such that $SC \xrightarrow{(\nu w : V_M)\dual w SC'}$ or $SC \xrightarrow{(\nu w : V_M)\dual w SC'}$. Hence the implications of Definition 5.1.2 where $a \in CP$ and $w \in MR$ or $a \in MR$ and $w$ is an operation label, hold vacuously.

Case 4:
$\beta = \tau$, by Subject Reduction 5.2.1 $\Gamma \vdash SC'$. Hence by Equation 5.5 we have that $M(SC')$ is true. Hence the implications of Definition 5.1.2 holds.

Thus $M$ is a mediator predicate. 

This section gave the basis for building basic processes using a type system which are mediators. In the remainder of in this thesis if we say that $SC$ is a sequential finite controller we implicitly assume that there is a $T$ such that $T$ is closed with respect to $SC$ and $\Gamma \vdash SC$.

Using the results Lemma 5.2.1, Lemma 5.2.2 and Lemma 5.2.3 we can construct larger processes which are mediators. One drawback so far is that the rules do not allow the construction of mediators using prefixed parallel composition. There are, however, valid mediators (such as Example 5.1.5) of this form. In such cases one has to show that the process is a mediator before it can be used to construct larger systems.

5.3 Control systems

As mentioned above, mediators operate concurrently with B-Machines in an agent which we call a control system. As there are many ways to combine mediators and machines together in a complete system, this section is about defining a combination style that is suitable for the kind of verification undertaken in Chapter 6.

Firstly, consider the following agent from Definition 4.5.1

$$M\text{GENERATOR} = \dual \! ((\nu z : \dual V_M) (\text{create}M_1(z).\text{init}M_1.0 \mid [\text{BEGIN}]M_1(z)) \mid \ldots \mid \dual !(\nu z : \dual V_M) (\text{create}M_n(z).\text{init}M_n.0 \mid [\text{BEGIN}]M_n(z))$$
where $M_i$, $1 \leq i \leq n$ are the machines we wish to consider in a specification. In Chapter 4 we showed that this agent is always prepared to output a new machine $M_i$ instance on the corresponding $createM_i$ channel and that it is not responsible for causing machine divergence in the machines it can output. Therefore, this agent is a useful component in any specification because it generalises machine creation for any other component of the system.

In the systems we consider, an interaction over a $createM_i$ is to be the point at which a fresh machine $M_i$ is sent out in one of its initial states and received by a mediator (which could itself be a collection of smaller mediators). In that sense, the $createM_i$ channel is a control point as it sends machine references, however it is a special case, because it is the only one which is allowed to output bound names as machine references. This was the reason why in Definition 5.1.2 on page 73, bound output of machine references was not an allowable behaviour for mediators.

However, if within a given mediator we designate some of the control points to be $createM_i$ control points, we must ensure that all mediators of our specification use these channels only to input machine references. Otherwise a mediator could mimic $MGENERATOR$ and there is no way to guarantee that the machines received on $createMi$s are in their initial states.

Consider a mediator where all control points are hidden except the $createMi$ channels. Such mediators can be considered to be like a black box with respect to the internal machine movements. In some sense all components that handle machines in a specification have been defined hence the mediator is complete. $createM_i$ is the only way a new machine can link with a mediator. No machine is ever removed once created (due to structural congruence). In example 5.1.3, controller 2 is complete.

**Definition 5.3.1.** If $M_1, \ldots, M_n$ are machine definitions we wish to consider in a specification, mediator $D$ is complete if,

1. $\text{cp}(D) \subseteq \{createM_1, \ldots, createM_n\}$, and

2. there is a $\Gamma$ closed with respect to $D$, such that $\Gamma(createM_i) = ioV_{M_i}$ for $1 \leq i \leq n$ and $\Gamma \vdash D$.

Note above we assume that $D$ is already a mediator and we revert to the normal type-system outlined in Table 3.4. Also note that the process $D$ must be typeable with the types of $createM$ set to permit input of machine references only. If $D$ outputs on a $createM$ then it would not type check.

In following definitions we write $\text{cp} : \text{CP}$ to denote that the name $\text{cp}$ is of some control point type (see Definition 5.2.2). Correspondingly, we write $z : \text{MR}$ to denote that the name $z$ is of some machine reference type and $x : \text{SN}$ to denote that the name $x$ is of
some type, other than machine reference or control point. Then a definition of control system, is a typed process as follows:

Definition 5.3.2. Given a complete, mediator \( D \), a control system \( \text{CSYSTEM}_D \) is a process where, for some natural numbers \( n \) and \( m \),

1. \( \text{CSYSTEM}_D \equiv \pi 
   (\nu \text{create} M_1 : \mathcal{CP}, \ldots, \text{create} M_n : \mathcal{CP}, \ x_1 : \# V_{M_1}, \ldots, x_m : \# V_{M_m}) 
   (D \mid [x_1]_{M_1}(x_1) \mid \ldots \mid [x_m]_{M_m}(x_m) \mid \text{MGENERATOR}) 
\)

where \( M_1, \ldots, M_m \) are instances of machines we consider in the specification and each \( x_i \), for \( 1 \leq i \leq m \) is some machine state from \( ST^w_{M_i} \) and,

2. there is a \( \Gamma \) that is closed with respect to \( \text{CSYSTEM}_D \) such that

   (a) for all names \( a \in \text{supp}(\Gamma) \), \( \Gamma(a) \neq \mathcal{MR} \) and \( \Gamma(a) \neq \mathcal{CP} \) (i.e. there are no free channels that are machine references or control points),

   (b) \( \Gamma \vdash \text{CSYSTEM}_D \).

The first item gives the structure of a typical control system process. Using the structural congruence rules any process that is potentially a control system can be of this form by virtue of Proposition 3.2.2. The second item of the definition states that in a control system there are no visible machine references or control points and also that \( \text{CSYSTEM} \) is correctly typed and ensures that the types for the hidden \( \text{create} M \) channels and machine references \( x_1, \ldots, x_m \) have been specified correctly. Note that \( M_1, \ldots, M_n \) are machines we wish to consider in the specification whereas, \( M_1, \ldots, M_m \) are instances of those.

As the system above evolves, machines get emitted from \( \text{MGENERATOR} \) and are bound with the mediator. Thus, \( D \) evolves to other mediators which have more machine references within them. For example if \( D \xrightarrow{\text{create} M_1^k} D' \), where \( k \in \text{mref}(D') \), after an appropriate \( \tau \) transition, such that \( \text{CSYSTEM} \xrightarrow{\tau} \text{CSYSTEM}' \) then, the control system takes on the following form.

\[
\text{CSYSTEM}'_{D'} \equiv \pi 
(\nu \text{create} M_1 : \# V_{M_1}, \ldots, \text{create} M_n : \mathcal{CP}, 
 x_1 : \mathcal{MR}, \ldots, x_m : \mathcal{MR}, \ x_{m+1} : \# V_{M_1}) 
(D' \mid [x_1]_{M_1}(x_1) \mid \ldots \mid [x_m]_{M_m}(x_m) \mid [x_{m+1}]_{M_1}(x_{m+1}) \mid \text{MGENERATOR})
\]

for some initial state, \( x_{m+1} \in ST^w_{M_1} \) of \( M_1 \). \( D' \) can now execute operation calls on the machine instance at \( x_{m+1} \).

Finally, note that we can write mediators, that have no machines in contact with them in other words \( \text{mref}(D) = \emptyset \). We call such mediators \textit{machine closed}. 

Definition 5.3.3. A mediator $D$ is machine closed if, $\text{mref}(D) = \emptyset$.

Lemma 5.3.1. Given a mediator $D$, if $D$ is machine closed and $D =_{\equiv} D_1 \cup D_2$ then $D_1$ is machine closed and $D_2$ is machine closed.

Proof. We have that $\text{mref}(D) = \text{mref}(D_1) \cup \text{mref}(D_2)$, (from $\text{fn}(D) = \text{fn}(D_1) \cup \text{fn}(D_2)$). $\text{mref}(D) = \emptyset$ thus $0 = \text{mref}(D_1) \cup \text{mref}(D_2)$ hence $\text{mref}(D_1) = \emptyset$ and $\text{mref}(D_2) = \emptyset$. □

Control systems where $D$ is machine closed, complete mediator take the following form:

$$\text{CSYSTEM}_D = \nu (\text{createM} : \mathcal{C}P, \bar{x} : \mathcal{S}N)(D \mid \text{MGENERATOR})$$

Such control systems need not have any active machine processes and we refer to those systems as being in an initial state.

Executing a mediator in the context of a control system meets the requirement from Definition 5.1.2 that every machine reference received on a control point is fresh.

Theorem 5.3.1. Suppose a mediator $D$ is defined as follows,

$$(\nu \bar{c}p : \mathcal{C}P)(D_1 \mid D_2)$$

for any mediator $D_1$, $D_2$ and $\bar{c}p : \mathcal{C}P$ such that, $\text{mref}(D_1) \cap \text{mref}(D_2) = \emptyset$ and $D$ is complete.

Suppose also we have a $\text{CSYSTEM}_D$ such that $\text{CSYSTEM}_D \rightarrow^\tau \text{CSYSTEM}_{D'}$, where for any $cp \in \mathcal{C}P$ and $z \in \mathcal{M}R$, $D_1 \overset{cp}{\rightarrow} D_1'$, for some $D_1'$ then $z \notin \text{mref}(D_1)$ and $\text{mref}(D_1') \cap \text{mref}(D_2) = \emptyset$.

Proof. By considering the transition derivation of $\text{CSYSTEM}_D \rightarrow^\tau \text{CSYSTEM}_{D'}$, where one of the leaves is $D_1 \overset{cp}{\rightarrow} D_1'$.

Case 1: Suppose $cp = \text{createM}_i$ for some machine $M_i$ which we consider in the specification.

Then by using the transition rules we can conclude that the interaction took place between $\text{MGENERATOR}$ and $D_1$.

$$D_1 \mid \text{MGENERATOR} \equiv \rightarrow (\nu z)(D_1' \mid \text{initM}_i.0 \mid ([\text{BEGIN}])_{M_i}(z))$$

Then via the application of $\text{CLOSE-R}$ we can conclude that the above is true where $z \notin \text{fn}(D_1)$.

If $z \notin \text{fn}(D_1)$ then we that $z \notin \text{mref}(D_1)$. By Proposition 5.1.2 we have that $\text{mref}(D_1') \subseteq \text{mref}(D_1) \cup \{z\}$ and we have that $\text{mref}(D_1) \cap \text{mref}(D_2) = \emptyset$ hence we have that
Case 2: 
In all other cases by using the transition rules we can conclude that the interaction took place between $D_1$ and $D_2$.

$$D_1 \mid D_2 \xrightarrow{\tau} D'_1 \mid D'_2$$

for some $D'_2$.

By rule $COMM-R$ we conclude that $D_2 \xrightarrow{\pi, z} D'_2$ and $D_1 \xrightarrow{op, z} D'_1$ (confirming the hypothesis).

Since $D_2$ is a mediator we can conclude that $z \notin mref(D'_2)$. Also by Lemma 5.1.2, $mref(D'_2) \subseteq mref(D_2) \setminus \{z\}$ and $mref(D'_1) \subseteq mref(D_1) \cup \{z\}$. We have that $mref(D_1) \cap mref(D_2) = \emptyset$ so $z \notin mref(D_1)$ and $(mref(D_1) \cup \{z\}) \cap (mref(D_2) \setminus \{z\}) = \emptyset$. So we have that $z \notin mref(D_1)$ and $mref(D'_2) \cap mref(D'_2) = \emptyset$. □

This theorem illustrates that control systems always evolve to other control systems.

**Theorem 5.3.2.** For any complete $D$ and $\alpha$, if the control system, $CSYSTEM_D$, performs $CSYSTEM_D \xrightarrow{\alpha} P'$ then $P'$ is a control system.

**Proof.** By Definition 5.3.2 $CSYSTEM_D$ does not have any visible machine references or control points.

In the case where $\alpha$ is a bound output action, by Definition 5.1.2 $D$ cannot output a control point. Since $D$ is complete it cannot output a machine reference. Hence if $D'$ is the continuation of $D$ by Definition 5.3.1 we have that $D'$ is complete and by Lemma 5.1.2, $mref(D') \subseteq mref(D)$. Hence no machine references or control points are visible in $P'$. By Subject Reduction 3.4.1 we have that $P'$ is well typed. Thus $P'$ is a control system.

In the case where $\alpha$ is a free output action we can show that $\alpha$ does not involve machine references or control points. Hence if $D'$ is the continuation of $D$ we have that $D'$ is complete and by Lemma 5.1.2, $mref(D') \subseteq mref(D)$. Hence no machine references or control points are visible in $P'$. By Subject Reduction 3.4.1 we have that $P'$ is well typed. Thus $P'$ is a control system.

In the case where $\alpha$ is an input action, again we cannot observe any $\alpha$ that inputs a control point or a machine reference. The first is true by Definition 5.2.1 which states that none of the channels of $D$ have the type $\#V_M$ for any $M$. Thus if $D'$ is a continuation process of $D$, we have that $D'$ is complete and by Lemma 5.1.2, $mref(D') \subseteq mref(D)$.
5.4 Semantic definition of machine divergence freedom

We consider a control system to be a specification of a system. We want to be able to verify that the control systems we write are machine divergence-free so that no machine instance that is created or manipulated by a mediator enters into a divergent state. In Section 1.1.2 we noted that machine divergence freedom is an essential property of combined specifications, without which invariants of machine instances cannot be guaranteed. This section gives a semantic definition of this requirement using traces.

Recall that \( P \Downarrow \text{div} \) denotes that the transition derivation \( P \xrightarrow{\text{div}} P' \) holds, as in Section 4.5.

**Proposition 5.4.1.** For any machine \( M \) and operation of the machine \( op \) and machine reference \( z \),

\[
[(\text{BODY}_{op}, \bot)]_M(z) \Downarrow \text{div} = \text{true}
\]

**Proof.** Follows directly from the rules in Table 4.3. □

Using Proposition 5.4.1 as a base case, it is possible to give a definition of \( P \Downarrow \overline{x} = \text{true} \) inductively over the syntax of \( P \) as a set of barbs [36]. However, here we will just use the transition rules.

Control systems are machine divergence free if the following holds.

**Definition 5.4.1.** Given a control system \( \text{CSYSTEM}_D \), \( \text{CSYSTEM}_D \) is machine divergent free if,

\[
\text{CSYSTEM}_D \Downarrow \text{div} = \text{false}
\]

and there exists a \( \text{CSYSTEM}_{D_k} \) such that,

\[
\text{CSYSTEM}_D \xrightarrow{\text{tr}} \text{CSYSTEM}_{D_k}
\]

for any trace \( \text{tr} \) and,

\[
\text{CSYSTEM}_{D_k} \Downarrow \text{div} = \text{false}
\]
The above definition assumes that \textit{div} is considered to be a special channel in the specification. In the control systems that we consider \textit{div} must never be hidden. We will use this definition in the next chapter when we consider the verification of machine divergence freedom properties of control systems.
This chapter presents a technique for identifying a set of weakest precondition proof obligations on a mediator $D$. Discharging these proof obligations means that the specification $CSYSTEM_D$ is machine divergence-free. Machine divergence freedom is an essential property of combined specifications without which we cannot make any guarantees about the state of the machine instances controlled by mediators.

6.1 Machine consistent mediators

The technique that we propose focuses on a complete machine closed mediator $D$ and is analogous in application to a typing derivation. Here, instead of channels types, we attach predicates to control points that act as assumptions (in a rely-guarantee style) within $D$. The control point assumptions play an important role because when machines are passed around in a specification we will need to reason about their state when they leave a mediator, and when they are received by another. When a machine leaves a mediator’s control we need to provide a guarantee about its state whereas when a machine is received by a mediator then we need to rely on it being in a particular state. The overall outcome is a weakest pre-condition predicate which takes these assumptions into account and also ensures that $D$ executes without causing machine divergence in between the control points.

We assume that $D$ is constructed using the Lemma 5.2.1, Lemma 5.2.2 and the rules for constructing a sequential finite controller ($SFC$), Definition 5.2.3. Note this section does not consider mediators with infinite replication, constructed using Lemma 5.2.3. We consider such mediators in Section 6.4. Hence, the definitions that follow show how to decompose a $D$ with finite execution into corresponding $SFC$s, identifying the control point assertions and finally converting them to $wp$ predicates.
Firstly, in this section we make use of the B machine renaming mechanism. This mechanism allows the renaming of the signature of a library machine \( M \) and its variables list so that several instances can be included into a larger specification. The structure of the names, used to define a library machine, is \( \text{string}_\text{MACHINE}, \text{string}_\text{VARIABLES}, \text{string}_\text{OPERATIONS} \) for a given \( \text{string} \). To incorporate two different instances of a machine the specifier modifies \( \text{string} \) to be different from one to the other. We propose to use the \( \pi \)-calculus name used to reference the channel for communication with the machine instance in place of \( \text{string} \). This is only possible because the semantics of the \( \pi \)-calculus ensures that these names are different from each other for each machine instance in our specification.

Given a B predicate \( R \) where \( \text{fn}(R) \subseteq \text{set}_\text{of}(\text{VARIABLES}) \) for some finite list of \( \text{VARIABLES} \), use \( z\text{-(R)} \) to denote the predicate \( \text{wp(Variables := z\_VARIABLES, R)} \) where \( z\_\text{VARIABLES} \) is a list of variable names where each item in \( \text{VARIABLES} \) is prefixed with a \( z\_ \) string.

**Example 6.1.1.**

\[
z\text{-(}nn = 1) \equiv z\_nn = 1 \\
z\text{-(}nn > 1 \land mm \in \mathbb{N}) \equiv z\_nn > 1 \land z\_mm \in \mathbb{N} \\
z\text{-(}R1 \Rightarrow R2) \equiv z\text{-(}R1) \Rightarrow z\text{-(}R2) \text{ through distributivity of wp}
\]

In addition to renaming variables so that they are identified with the \( \pi \) channel used to communicate with the particular \( B \) machine we also rename the machine's operations. We denote with \( z\_\text{operation} \) the expression obtained from replacing every free name \( x \) in the operation of the machine called \( \text{operation} \) with \( z\_x \). For example,

**Example 6.1.2.** Consider the definition of operation tick of machine Clock 2.2, then we have that,

\[
z\text{\_tick} \equiv \text{PRE } z\_nn = 0 \text{ THEN } z\_nn := 1 \text{ END}
\]

Secondly, given the dynamic nature of being able to pass machines around a specification we need to define a mechanism for tagging machine state requirements at control points. This is because when we consider the disassembled SFCs we may need to make assumptions about the state of incoming machines in an SFC before they are manipulated. To illustrate, consider the following complete mediator \( D \),

**Example 6.1.3.**

\[
D = (\nu \ cp : \#oV_M)(SC_1 \mid SC_2)
\]

where \( SC_1 \) and \( SC_2 \) are SFC mediators sharing the control point \( cp \). In disassembling this expression we remove the hiding operator and then proceed to check whether \( SC_1 \) and \( SC_2 \) are machine divergence-free in isolation. However, neither SFC mediator has enough information to show machine consistency on its own. This is because through control point \( cp \) they might exchange machines between them, and would not know the state of the incoming machine. Thus, in some sense we must also show that the two are suitable to be in parallel with each other. \( SC_1 \) and \( SC_2 \) are suitable if whenever one sends a machine on \( cp \) it is in a state that the other expects and vice versa.
**Definition 6.1.1. Assertion tags**

1. The agent 0 is a tagged mediator;
2. Given a mediator D such that,
   \[ D \equiv \pi (\nu \ cp : \text{cp}_0V_M)(D') \]
   for some tagged mediator D'. The process
   \[ (\nu \ cp^R : \text{cp}_0V_M)(D') \]
   is a tagged mediator for some B predicate R such that,
3. the predicate R belongs to M.

The tags are expressed as B predicates and do not affect the name cp. Tags are supplied manually for each control point D in the agent except the createMi because for these channels the assertion is pre-set by the initialisation statement of M as we shall see below. In providing a tag the specifier must ensure that the predicate belongs to the machine M for which cp will transmit references. The machine is identified from the type given to cp, which is of the form \( \text{cp}_0V_M \) discussed in the previous section. Recall that "R belongs to a machine M" means that R is well-typed with respect to the signature of M.

While disassembling a process and checking for machine consistency, the predicate tags are recorded in a function called assert defined as follows,

**Definition 6.1.2.**

\[ \text{assert} \in \mathcal{CP} \rightarrow \mathcal{R} \]

where \( \mathcal{R} \) is the set of all B predicates.

The notation \( \text{assert}(cp) = R \) where \( R \in \mathcal{R} \), denotes that the control point cp is assigned the predicate R in assert.

**Definition 6.1.3.** A given function assert is closed with respect to a mediator D if for every cp such that \( cp \in \text{cp}(D) \) we have \( \text{assert}(cp) = (R) \) for some predicate R.

We provide the assertion function assert_init which is only defined on the createMi control points where the predicates are pre-set as follows.

**Definition 6.1.4.** If \( M_1, \ldots, M_n \) are the machines we wish to consider in the specification.

The function assert_init is defined as follows,

\[ \text{assert_init}(\text{createMi}) = R_i \]
for each $1 \leq i \leq n$ where $R_i$ is a predicate as follows,

$$R_i = \wp(VARIABLES'_M := VARIABLES_M, \text{ prd}_{VARIABLES_M} (T_M))$$

where $T_M$ is the initialisation statement of machine $M_i$, $VARIABLES_M$ is $M_i$ corresponding list of variables, and $VARIABLES'_M$ is a list where each item of $VARIABLES_M$ is decorated.

Recall from Section 4.1.2 that the effect of $\text{prd}_{VARIABLES}(T_M)$ on the initialisation statement $T_M$ gives a predicate where the free names are all decorated versions of the real variables. Thus, in the definition above we have applied a substitution to relate the primed variables with their unprimed counterparts. For example, in the case of machine Clock.

Example 6.1.4.

$$\text{prd}_{nn}(nn := 1) = (nn' = 1)$$

$$\wp(nn' := nn, (nn' = 1)) = (nn = 1)$$

We are now ready to define a function on mediators that maps them to a predicate which is the weakest precondition so that $D$ executes without machine divergence. We call this function $DIV_p(D, assert)$ where $p$ is a set of machine references, $D$ is a mediator, and $assert$ is an assertion function defined above. The set $p$ is used to constrain $DIV$ to a particular subset of machine references that we wish to examine. In all applications $p$ is a subset of the machine reference set $mref(D)$. When $p$ is the $\emptyset$ in particular, $DIV$ checks whether the mediator meets all control point assertions in a rely guarantee style. In all other cases $DIV_p$ does what $DIV_\emptyset$ does and in addition, outputs the weakest precondition so that the machines in $p$ would not diverge. For all practical purposes $p = mref(D)$, we have the extra functionality for proof purposes.
**Definition 6.1.5.** Given a tagged mediator \( D \) and a function assert closed with respect to \( D \),

\[
DIV_\rho(D, \text{ assert})
\]

is defined as follows,

1. if \( D \equiv_L (\nu \ v : S)(D') \) where \( S \) is not a control point type (see Definition 5.2.1) then

\[
DIV_\rho(D, \text{ assert}) = DIV_\rho(D', \text{ assert})
\]

2. if \( D \equiv_L (\nu \ cp^R : S)(D') \) and \( \forall \ o \ V_M \leq S \) (i.e. \( cp \) is a control point) then

\[
DIV_\rho(D, \text{ assert}) = DIV_\rho(D', \text{ assert} \cup \{(cp, R)\})
\]

3. \( D \equiv_L SC \mid D' \) where \( SC \) is a sequential finite controller then

\[
DIV_\rho(D, \text{ assert}) = DIV_{\rho \cap mref(SC)}(SC, \text{ assert}) \land
DIV_{\rho \cap mref(D')}(D', \text{ assert})
\]

4. \( D \equiv_L SC \) where \( SC \) is a sequential finite controller then

\[
DIV_\rho(D, \text{ assert}) = wp(\text{convert}(D)_\rho, \text{ true})
\]

with function assert, see Definition 6.1.6 for \( \text{convert}(D)_\rho \)

The first case allows us to ignore a hidden channel whenever the channel is not a control point. The second case identifies the control point \( cp \) with predicate \( R \) in the assert function and continues to check \( D' \) with the extra information. The third case requires that we check both agents of a parallel composition. Observe that we constrain \( \rho \) so that each agent of parallel combination is converted using those machine references that are applicable to it. In the case of infinite replication, we need only consider one execution of the replicated process.

The last case of Definition 6.1.5 is defined as follows.

**Definition 6.1.6.** Given a sequential finite controller \( SC \) and a function assert, closed with respect to \( SC \), \( \text{convert}(SC)_\rho \) is defined for \( \rho \subseteq mref(SC) \) as follows.
Assuming convert(SC₁)ᵢ, convert(SC₂)ᵢ are defined,

\[
\text{convert}(0)ᵢ = \text{skip}
\]

where \( cp \) is an input control point

\[
\text{convert}(\text{cp}(b).SC₁)ᵢ = \begin{cases} 
\text{SELECT } \text{b-assert}(cp) & b \notin \rho \\
\text{THEN } \text{skip END; convert}(SC₁)ᵢ \quad b \notin \rho
\end{cases}
\]

where \( z \) is a machine reference and \( l \) is an operation label

\[
\text{convert}(z.l,SC₁)ᵢ = \begin{cases} 
\text{convert}(SC₁)ᵢ \text{preref}(SC₁) & \text{if } z \notin \rho \\
(z.\text{op}(l); \text{convert}(SC₁)ᵢ \text{preref}(SC₁)) & z \in \rho
\end{cases}
\]

where \( cp \) is a control point

\[
\text{convert}(\text{cp}(z).SC₁)ᵢ = \begin{cases} 
\text{convert}(SC₁)ᵢ \text{preref}(SC₁) & \text{if } z \notin \rho \\
\text{PRE } z-\text{assert}(cp) & \text{z ∈ ρ} \\
\text{THEN } \text{skip END; convert}(SC₁)ᵢ \text{preref}(SC₁)
\end{cases}
\]

and for any prefix \( π \) which is not covered by the cases above

\[
\text{convert}(π.SC₁)ᵢ = \text{convert}(SC₁)ᵢ
\]

where \( v \) is a standard name

\[
\text{convert}((v.v:S)(SC₁))ᵢ = \text{convert}(SC₁)ᵢ
\]

\[
\text{convert}(SC₁ + SC₂)ᵢ = \text{CHOICE convert}(SC₁)ᵢ \text{preref}(SC₁) \text{ OR convert}(SC₂)ᵢ \text{preref}(SC₂) \text{ END}
\]

The agent 0 is converted to \text{skip} as it does not perform any actions. An operation call \( z.l \) is converted to the AMN \( z-\text{op}(l) \) which has the effect of pre-pending every variable used in the definition of \( \text{op}(l) \) with the string \( z- \) as discussed above. At an input control point we need to capture that a new machine has arrived with an expected state captured in \( \text{assert}(cp) \). This produces a guard that must discharge any subsequent state updates performed on the new machine. The name chosen to represent this new machine is different from other strings that have been used since the process conforms to the naming convention.

At an output control point we must check whether the machine is in a state that is compliant with the expectations of other mediators which might receive it. In other words its state meets the \( \text{assert}(cp) \) condition before output. This produces a pre-condition hence assertion violations are treated in a similar way to operation executions.
All other syntactic constructs are simply ignored by convert except the sum operator which is converted to an AMN choice. Note that at each stage the environment $\rho$ is constrained with the remaining machine references in the continuation process.

By obtaining $\text{DIV}_\emptyset(D, \text{assert}) = \text{true}$ for a machine closed $D$ we can conclude that $D$ is machine-consistent.

### Definition 6.1.7. Machine-Consistent Mediator

Given a machine closed, tagged mediator $D$, and there exists a function assert closed with respect to $D$ such that

$$\text{DIV}_\emptyset(D, \text{assert}) = \text{true}$$

then $D$ is machine-consistent.

In the rest of this chapter we prove that if $D$ is machine-consistent then a control system $\text{CSYSTEM}_D$ is machine divergence-free.

### 6.2 Example

In this section we present two examples which illustrate a drawback of the $\text{DIV}$ predicate and how this is resolved.

#### Example 6.2.1. Consider the controllers,

$$SC_1 \equiv \text{createCl}(w). w \text{ tick}_.(*). \text{cp}(w). \text{cp}(b). b \text{ tick}_.(*). \text{cp}(b). 0$$

$$SC_2 \equiv \text{cp}(m). m \text{ took}_.(*). \text{cp}(m). \text{cp}(n). n \text{ took}_.(*). 0$$

$$D \triangleq (\nu \text{ cp}^R : \|o[\text{tick}_.\text{unit}, \text{took}_.\text{unit}]) (SC_1 \mid SC_2)$$

which are responsible for controlling a single Clock machine from Figure 2.2. Controller $SC_1$ is responsible for creating an instance of the machine and controls the executions of the tick operation. Similarly, controller $SC_2$ communicates with $SC_1$ along control point $cp$ and is responsible for controlling the executions of the took operation.

The following shows that, $D$ is not machine-consistent because there is no assertion predicate $R$ for control point $cp$ that is strong enough to meet the pre-conditions of both tick and took.

We pre-set $\text{assert}_.\text{init}(\text{createCl}) = (nn = 1)$ as required by Definition 6.1.4. Then, we need to show that $\text{DIV}_\emptyset(D, \text{assert}_.\text{init}) = \text{true}$ by Definition 6.1.7. Applying Definition 6.1.5.

$$\text{DIV}_\emptyset(SC_1 \mid SC_2, \text{assert}_.\text{init} \cup \{(cp, R)\}) = \text{true}$$

for some B predicate $R$ which can be one of the following,

$$R1 \triangleq (nn = 1)$$

$$R2 \triangleq (nn = 2)$$

$$R3 \triangleq (nn = 1 \lor nn = 2)$$
Clearly, $nn = 1$ belongs to $Clock$ and $nn = 2$ belongs to $Clock$ since $nn$ is a variable of $Clock$ and the predicates are well typed with respect to the signature of the machine.

A further application of Definition 6.1.5 means that we have to show that,

$$DIV_0(SC_1, assert\_init \cup \{(cp, R)\}) = true,$$
$$DIV_0(SC_2, assert\_init \cup \{(cp, R)\}) = true$$

Let us check the first clause with $R = R1$ and by Definition 6.1.5 we must show that,

$$wp(convert(SC_1), true) = true.$$  
However, by Definition 6.1.6,

$$wp(convert(SC_1), true) = wp(SELECT \ w_{nn} = 1 THEN \ skip \ END; \ w_{tick}; \ PRE \ w_{nn} = 1 THEN \ skip \ END; \ SELECT \ b_{nn} = 1 THEN \ skip \ END; \ b_{tick}; \ PRE \ b_{nn} = 1 THEN \ skip \ END, true)$$

$$= wp(SELECT \ w_{nn} = 1 THEN \ skip \ END; \ w_{tick}; \ PRE \ w_{nn} = 1 THEN \ skip \ END; \ SELECT \ b_{nn} = 1 THEN \ skip \ END; \ PRE \ b_{nn} = 1 THEN \ b_{nn} := 2; \ b_{nn} = 1)$$

$$= wp(SELECT \ w_{nn} = 1 THEN \ skip \ END; \ w_{tick}; \ PRE \ w_{nn} = 1 THEN \ skip \ END; \ SELECT \ b_{nn} = 1 THEN \ skip \ END; \ b_{nn} = 1 \land \ 2 = 1)$$

$$= false$$

Hence it is not the case that $DIV_0(SC_1, assert\_init \cup \{(cp, R1)\}) = true.$
Now consider the predicate $R = R_2$.

$$\text{wp}(\text{convert}(SC_1)_g, \text{true}) = \text{wp}(\text{SELECT } w\_nn = 1 \text{ THEN skip END; } w\_\text{tick;} \text{ PRE } w\_nn = 2 \text{ THEN skip END; }$$
$$\text{SELECT } b\_nn = 2 \text{ THEN skip END; } b\_\text{tick;} \text{ PRE } b\_nn = 2 \text{ THEN skip END, true})$$
$$\vdots$$
$$= \text{wp}(\text{SELECT } w\_nn = 1 \text{ THEN skip END; } w\_\text{tick;} \text{ PRE } w\_nn = 2 \text{ THEN skip END; }$$
$$\text{SELECT } b\_nn = 2 \text{ THEN skip END, } b\_nn = 1) \tag{6.2}$$
$$= \text{wp}(\text{SELECT } w\_nn = 1 \text{ THEN skip END; } w\_\text{tick;} \text{ PRE } w\_nn = 2 \text{ THEN skip END; }$$
$$b\_nn = 2 \Rightarrow b\_nn = 1)$$
$$\vdots$$
$$= \text{false}$$

Hence it is not the case that $\text{DIV}_g(SC_1, \text{assert\_init} \cup \{(c_2, R_2)\}) = \text{true}$.

Now consider the predicate $R = R_3$.

$$\text{wp}(\text{convert}(SFC_1)_g, \text{true}) = \text{wp}(\text{SELECT } w\_nn = 1 \text{ THEN skip END; } w\_\text{tick;} \text{ PRE } w\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END; }$$
$$\text{SELECT } b\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END; } b\_\text{tick;} \text{ PRE } b\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END, true})$$
$$= \text{wp}(\text{SELECT } w\_nn = 1 \text{ THEN skip END; } w\_\text{tick;} \text{ PRE } w\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END; }$$
$$\text{SELECT } b\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END, } b\_nn = 1 \lor b\_nn = 2) \tag{6.3}$$
(cont)
(cont)

\[ \begin{align*}
\text{wp}( & \text{SELECT } w\_nn = 1 \text{ THEN skip END}; \\
 & w\_tick; \text{ PRE } w\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END}; \\
 & \text{SELECT } b\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END}, \\
 & b\_nn = 1) \\
\end{align*} \]

\[ \begin{align*}
\text{wp}( & \text{SELECT } w\_nn = 1 \text{ THEN skip END}; \\
 & w\_tick; \text{ PRE } w\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END}, \\
 & (b\_nn = 1 \lor b\_nn = 2) \Rightarrow b\_nn = 1) \\
\end{align*} \]  \hspace{1cm} (6.4)

\[ \begin{align*}
\text{wp}( & \text{SELECT } w\_nn = 1 \text{ THEN skip END}; \\
 & w\_tick; \text{ PRE } w\_nn = 1 \lor b\_nn = 2 \text{ THEN skip END}, \\
 & b\_nn = 2 \Rightarrow b\_nn = 1) \\
\end{align*} \]

\[ \begin{align*}
= & \text{false} \\
\end{align*} \]

From Equation 6.1 and Equation 6.3 we can deduce that there is no \( R \) such that
\[ \text{assert}(cp) = R \text{ and } \text{DIV}_0(SC_1, \text{assert}) = true. \]
Thus, \( SC_1 \) is not machine-consistent.

As we have already identified one of the participating controllers as being divergent
(by Definition 6.1.5 \( \text{DIV}_0(SC_1 | SC_2, \text{assert}) \) being a conjunction) we do not need
to consider \( SC_2 \) to deduce that \( D \) is not machine-consistent.

\( D \) is not machine-consistent, not because the operation calls are in the wrong order but
because only one control point is used and there is no predicate which can be assigned
to \( cp \) that simultaneously meets the pre-condition and post-condition of \( tick \). But in
fact, if we consider the sequence of operations called from within \( D \) their pre-conditions
will be met. Therefore, having just one control point is not enough in this example to
prove machine consistency. Consider adding an extra control point which would not
change the intended behaviour of \( D \).

**Example 6.2.2.** Consider the following controllers,
\[ SC_3 \triangleq \text{createCl}(w)\_\overline{w} \text{ tick\_(*).} c_{\overline{f}_2}(w).c_{\overline{p}_1}(b)\_\overline{b} \text{ tick\_(*).} c_{\overline{p}_2}(b)\_0 \]
\[ SC_4 \triangleq c_{p_2}(m)\_\overline{m} \text{ tock\_(*).} c_{\overline{f}_1}(m).c_{p_2}(n)\_\overline{n} \text{ tock\_(*).} 0 \]
\[ D_2 \triangleq (\nu \text{ cp}_1^{R_1}: \|o\_V_M, c_{p_2}^{R_2}: \|o\_V_M)(SC_3 | SC_4) \]

where \( \text{cp}_1 \) is used to pass control from \( SC_4 \) to \( SC_3 \) so that \( SC_3 \) can perform ticks and
similarly \( \text{cp}_2 \) used to pass control from \( SC_3 \) to \( SC_4 \) so that \( SC_4 \) can perform tocks.

Here by Definition 6.1.5, we need to define two assertion predicates before committing
to verify \( SC_3 \) and \( SC_4 \) individually. Let those be as follows:
\[ R1 \triangleq (nn = 1) \]
\[ R2 \triangleq (nn = 2) \]
Consider Definition 6.1.6 applied to $SC_3$ with $\text{assert}(cp_1) = R_1$ and $\text{assert}(cp_2) = R_2$.

$$wp(\text{convert}(SC_3)_B, \text{true}) = wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END};$$

$$w_{\text{tick}}; \text{ PRE } w_{nn} = 2 \text{ THEN skip END};$$

$$\text{SELECT } b_{nn} = 1 \text{ THEN skip END};$$

$$b_{\text{tick}}; \text{ PRE } b_{nn} = 2 \text{ THEN skip END, true})$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END};$$

$$w_{\text{tick}}; \text{ PRE } w_{nn} = 2 \text{ THEN skip END};$$

$$\text{SELECT } b_{nn} = 1 \text{ THEN skip END};$$

$$b_{\text{tick}}, b_{nn} = 2)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END};$$

$$w_{\text{tick}}; \text{ PRE } w_{nn} = 2 \text{ THEN skip END};$$

$$\text{SELECT } b_{nn} = 1 \text{ THEN skip END};$$

$$\text{ PRE } b_{nn} = 1 \text{ THEN } b_{nn} := 2, b_{nn} = 2)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END};$$

$$w_{\text{tick}}; \text{ PRE } w_{nn} = 2 \text{ THEN skip END};$$

$$\text{SELECT } b_{nn} = 1 \text{ THEN skip END},$$

$$b_{nn} = 1)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END};$$

$$w_{\text{tick}}; \text{ PRE } w_{nn} = 2 \text{ THEN skip END},$$

$$b_{nn} = 1 \text{ imps } b_{nn} = 1)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END},$$

$$w_{\text{tick}}, w_{nn} = 2)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END},$$

$$\text{ PRE } w_{nn} = 1 \text{ THEN } w_{nn} := 2; w_{nn} = 2)$$

$$= wp(\text{SELECT } w_{nn} = 1 \text{ THEN skip END}, w_{nn} = 1)$$

$$=(w_{nn} = 1 \Rightarrow w_{nn} = 1)$$

$$= \text{true}$$

(6.5)
Now consider Definition 6.1.6 applied to $SC_4$.

$$\text{wp(convert}(SC_4_0, \text{true}) = \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END; w_tock; PRE } w_{nn} = 1 \text{ THEN skip END; SELECT } b_{nn} = 2 \text{ THEN skip END; b_tock, true})$$

$$= \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END; w_tock; PRE } w_{nn} = 1 \text{ THEN skip END; SELECT } b_{nn} = 2 \text{ THEN skip END; PRE } b_{nn} = 2 \text{ THEN } b_{nn} := 1, \text{true})$$

$$= \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END; w_tock; PRE } w_{nn} = 1 \text{ THEN skip END; SELECT } b_{nn} = 2 \text{ THEN skip END, } b_{nn} = 2)$$

$$= \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END; w_tock; } w_{nn} = 1)$$

$$= \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END; PRE } w_{nn} = 2 \text{ THEN } w_{nn} := 1, w_{nn} = 1)$$

$$= \text{wp}(\text{SELECT } w_{nn} = 2 \text{ THEN skip END, } w_{nn} = 2)$$

$$=(w_{nn} = 2 \Rightarrow w_{nn} = 2)$$

$$= \text{true} \quad (6.6)$$

Thus, $DIV_0(SC_3, assert\_init \cup \{(cp_1, R1), (cp_2, R2)\}) = \text{true}$ and $DIV_0(SC_4, assert\_init \cup \{(cp_1, R1), (cp_2, R2)\}) = \text{true}$ hence by Definition 6.1.5 we can conclude that $DIV_0(D_2, assert\_init) = \text{true}$
Therefore, \( D_2 \) is \textit{machine-consistent}. From the examples above we must be clear about the purpose of the control points in a specification because the assertions reflect the rely guarantee conditions between the controllers.

### 6.3 Some useful results about \( \text{DIV} \)

We are working towards a consistency proof to show that if a mediator \( D \) is machine-consistent then the control system \( \text{CSYSTEM}_D \) is machine divergence-free. This is presented in Section 6.5. In this section we prove a number of supporting results about the \( \text{DIV} \) predicate which will be used in the proofs of the theorems in Section 6.5.

The following Lemma shows that the undecorated \( \text{DIV} \) predicate of a sequential finite controller is a conjunction of the decorated \( \text{DIV}_{\langle z \rangle} \) predicates for every machine reference \( z \) in the controller.

\textbf{Lemma 6.3.1.} For any sequential finite controller \( SC \), \( \rho \subseteq \text{mref} (SC) \) and some function \( \text{assert} \) closed with respect to \( SC \),

\begin{enumerate}
\item \( \text{DIV}_\rho (SC, \text{assert}) \Rightarrow \text{DIV}_\delta (SC, \text{assert}) \) for any \( \delta \subseteq \rho \).
\item \( \bigwedge_{z \in \rho} \text{DIV}_{\langle z \rangle} (SC, \text{assert}) \Leftrightarrow \text{DIV}_\rho (SC, \text{assert}) \)
\end{enumerate}

\textit{Proof.} By structural induction on sequential finite controllers. Case 1 shows that the first half of the theorem is correct and Case 2 shows the second half. Note that in proving Case 1 we have effectively proved the reverse implication of Case 2.

\textbf{Case 1:}

\textbf{Case 1.1 (Base Case):}

Suppose \( SC = 0 \),

Let \( SC \neq 0 \). We have that \( \text{mref}(0) = 0 \).

Then we need to show that,

\[ \text{DIV}_0 (0, \text{assert}) \Rightarrow \text{DIV}_0 (0, \text{assert}) \]

which is true.

\textbf{Case 1.1 (Inductive Case):}

Assume that for any sequential finite controller \( SC_1, \rho \subseteq \text{mref}(SC_1) \) and some function \( \text{assert} \) closed with respect to \( SC_1 \),

\[ \text{DIV}_\rho (SC_1, \text{assert}) \Rightarrow \text{DIV}_\delta (SC_1, \text{assert}) \]
for any $\delta \subseteq \rho$.

Case 1.1.1:
Let $SC = cp(b).SC_1$ where $cp$ is a control point.

Then we need to show that,

$$DIV_\rho(cp(b).SC_1, \text{assert}) \Rightarrow DIV_\delta(cp(b).SC_1, \text{assert})$$

where $\delta \subseteq \rho \subseteq \text{mref}(cp(b).SC_1)$.

By Definition 6.1.6,

$$DIV_\rho(cp(b).SC_1, \text{assert}) = \text{wp(SELECT b_assert(cp) THEN skip END; convert(SC_1)_{\rho\cup\{b\}}, true)}$$

$$= b_assert(cp) \Rightarrow DIV_{\rho\cup\{b\}}(SC_1, \text{assert})$$

$$\Rightarrow \text{by inductive hypothesis}$$

$$b_assert(cp) \Rightarrow DIV_{\delta\cup\{b\}}(SC_1, \text{assert})$$

where $b \notin \rho$.

By Definition 6.1.6,

$$DIV_\delta(cp(b).SC_1, \text{assert}) = \text{wp(SELECT b_assert(cp) THEN skip END; convert(SC_1)_{\delta\cup\{b\}}, true)}$$

$$= b_assert(cp) \Rightarrow DIV_{\delta\cup\{b\}}(SC_1, \text{assert})$$

where $b \notin \rho$.

Thus we have the case.

Case 1.1.2:
Let $SC = \overline{bl}.SC_1$ where $b$ is a machine reference.

Then we need to show that,

$$DIV_\rho(\overline{bl}.SC_1, \text{assert}) \Rightarrow DIV_\delta(\overline{bl}.SC_1, \text{assert})$$

where $\delta \subseteq \rho \subseteq \text{mref}(\overline{bl}.SC_1)$.

Case 1.1.2.1:
Suppose $b \notin \rho$. 
By Definition 6.1.6,

\[
\text{DIV}_\rho(b_1.SC_1, \text{assert})
\]
\[
= \text{because } b \notin \rho
\]
\[
wp(\text{convert}(SC_1)_{\rho \setminus \text{mref}(SC_1)}, \text{true})
\]
\[
= \text{DIV}_{\rho \setminus \text{mref}(SC_1)}(SC_1, \text{assert})
\]
\[
\Rightarrow \text{ by inductive hypothesis}
\]
\[
\text{DIV}_{\delta \setminus \text{mref}(SC_1)}(SC_1, \text{assert})
\]

By Definition 6.1.6,

\[
\text{DIV}_\delta(b_1.SC_1, \text{assert})
\]
because \( b \notin \rho \) and \( \delta \subseteq \rho \)
\[
= wp(\text{convert}(SC_1)_{\delta \setminus \text{mref}(SC_1)}, \text{true})
\]
\[
= \text{DIV}_{\delta \setminus \text{mref}(SC_1)}(SC_1, \text{assert})
\]

Thus we have the case.

Case 1.1.2.2:
Suppose \( b \in \rho \).

By Definition 6.1.6,

\[
\text{DIV}_\rho(b_1.SC_1, \text{assert})
\]
\[
= wp(b \_op(l); \text{convert}(SC_1)_{\rho \setminus \text{mref}(SC_1)}, \text{true})
\]
\[
= wp(b \_op(l), \text{DIV}_{\rho \setminus \text{mref}(SC_1)}(SC_1, \text{assert}))
\]
\[
\Rightarrow \text{ by inductive hypothesis}
\]
\[
wp(b \_op(l), \text{DIV}_{\delta \setminus \text{mref}(SC_1)}(SC_1, \text{assert}))
\]

Then either \( b \in \delta \) or \( b \notin \delta \).

Case 1.1.2.2.1:
Suppose \( b \notin \delta \).
By Definition 6.1.6,

\[
DIV_\delta(\bar{b}l.SC_1, \text{assert}) \\
= wp(convert(SC_1)_{\delta\cap mref(SC_1)}, \text{true}) \\
= DIV_{\delta\cap mref(SC_1)}(SC_1, \text{assert}) \\
= wp(skip, DIV_{\delta\cap mref(SC_1)}(SC_1, \text{assert})) \\
= \text{since the variables in } b\text{-op(l)} \\
do not affect any variables in } DIV_{\delta\cap mref(SC_1)}(SC_1, \text{assert}) \text{ since } b \notin \delta \\
wp(b\text{-op(l)}, DIV_{\delta\cap mref(SC_1)}(SC_1, \text{assert}))
\]

Thus we have the case.

Case 1.1.2.2.2:
Suppose \( b \in \delta \).

By Definition 6.1.6,

\[
DIV_\delta(\bar{b}l.SC_1, \text{assert}) \\
= wp(b\text{-op(l)}, DIV_{\delta\cap mref(SC_1)}(SC_1, \text{assert}))
\]

Thus we have the case.

Case 1.1.3:
Let \( SC' = \bar{cp}(b).SC_1 \) where \( cp \) is an output control point and \( b \) is a machine reference.

Then we need to show that,

\[
DIV_\rho(\bar{cp}(b).SC_1, \text{assert}) \Rightarrow DIV_\delta(\bar{cp}(b).SC_1, \text{assert})
\]

where \( \delta \subseteq \rho \subseteq mref(\bar{cp}(b).SC_1) \).

(Note that \( \bar{cp}(b).SC_1 \) is the same as executing an operation where the pre-condition is equivalent to \( b\text{-assert}(cp) \).)

Hence, the case is similar to Case 1.1.2.

Case 1.1.4:
For any prefix \( \pi \) not covered by the cases above suppose \( SC' = \pi.SC_1 \). The proof is similar to Case 1.1.2.1.
Case 1.1.5:
Where $v$ is a standard name $SC \equiv (\nu \; v : S)(SC_1)$. The proof is similar to Case 1.1.2.1

Case 1.1.6:
Assume that inductive hypothesis extends to agent $SC_2$.

Suppose $SC \equiv SC_1 + SC_2$.

Then we need to show that,

$$DIV_{\delta}(SC_1 + SC_2, \text{assert}) \Rightarrow DIV_\delta(SC_1 + SC_2, \text{assert})$$

where $\delta \subseteq \rho \subseteq mref(SC_1 + SC_2)$.

By Definition 6.1.6,

$$DIV_{\rho}(SC_1 + SC_2, \text{assert})$$

$$= DIV_{\rho \cap mref(SC_1)}(SC_1, \text{assert}) \land DIV_{\rho \cap mref(SC_2)}(SC_2, \text{assert})$$

$$\Rightarrow \text{ by inductive hypothesis}$$

$$DIV_{\delta \cap mref(SC_1)}(SC_1, \text{assert}) \land DIV_{\delta \cap mref(SC_2)}(SC_2, \text{assert})$$

By Definition 6.1.6,

$$DIV_{\delta}(SC_1 + SC_2, \text{assert})$$

$$= DIV_{\delta \cap mref(SC_1)}(SC_1, \text{assert}) \land DIV_{\delta \cap mref(SC_2)}(SC_2, \text{assert})$$

Thus we have the case.

Case 2:
Consider $\Rightarrow$.

Case 2.1 (Base Case):
Let $SC \equiv 0$. We have that $mref(0) = \emptyset$.

Then we need to show that,

$$\bigwedge_{x \in mref(0)} DIV_{\{x\}}(0, \text{assert}) \Rightarrow DIV_{\emptyset}(0, \text{assert})$$

which is the same as

$$DIV_{\emptyset}(0, \text{assert}) \Rightarrow DIV_{\emptyset}(0, \text{assert})$$
which is true.

Case 2.2 (Inductive Case):
Assume that for any sequential finite controller $SC_1$, $\rho \subseteq mref(SC_1)$, and function $assert$ closed with respect to $SC_1$,

$$\bigwedge_{z \in \rho} DIV_{\{z\}}(SC_1, \text{assert}) \Rightarrow DIV_{\rho}(SC_1, \text{assert})$$

Case 2.1.1:
Let $SC \triangleq cp(b).SC_1$ where $cp$ is an input control point.

Then we need to show that,

$$\bigwedge_{z \in \rho} DIV_{\{z\}}(cp(b).SC_1, \text{assert}) \Rightarrow DIV_{\rho}(cp(b).SC_1, \text{assert})$$

where $\rho \subseteq mref(cp(b).SC_1)$.

By Definition 6.1.6,

$$\bigwedge_{z \in \rho} DIV_{\{z\}}(cp(b).SC_1, \text{assert})$$

$$= \bigwedge_{z \in \rho} wp(\text{SELECT } b-\text{assert}(cp) \text{ THEN skip END}; \text{convert}(SC_1)_{\{z\} \cup \{b\}}, \text{true}) =$$

$$= \bigwedge_{z \in \rho} b-\text{assert}(cp) \Rightarrow DIV_{\{z\} \cup \{b\}}(SC_1, \text{assert})$$

$$= b-\text{assert}(cp) \Rightarrow \bigwedge_{z \in \rho} DIV_{\{z\} \cup \{b\}}(SC_1, \text{assert})$$

$$\Rightarrow \text{by Lemma 6.3.1(1)}$$

$$b-\text{assert}(cp) \Rightarrow \bigwedge_{z \in \rho} (DIV_{\{z\}}(SC_1, \text{assert}) \land DIV_{\{b\}}(SC_1, \text{assert}))$$

$$= b-\text{assert}(cp) \Rightarrow \bigwedge_{z \in (\rho \cup \{b\})} (DIV_{\{z\}}(SC_1, \text{assert}))$$

$$\Rightarrow \text{by inductive hypothesis}$$

$$b-\text{assert}(cp) \Rightarrow DIV_{\rho \cup \{b\}}(SC_1, \text{assert})$$

where $b \notin \rho$.

By Definition 6.1.6,

$$DIV_{\rho}(cp(b).SC_1, \text{assert})$$

$$= wp(\text{SELECT } b-\text{assert}(cp) \text{ THEN skip END}; \text{convert}(SC_1)_{\rho \cup \{b\}}, \text{true})$$

$$= b-\text{assert}(cp) \Rightarrow DIV_{\rho \cup \{b\}}(SC_1, \text{assert})$$
where \( b \notin \rho \).

Thus we have the case.

**Case 2.1.2:**
Let \( SC \triangleq \overline{b}.SC_1 \) where \( b \) is a machine reference.

Then we need to show that,

\[
\bigwedge_{z \in \rho} DIV_{\{z\}}(\overline{b}.SC_1, \text{assert}) \Rightarrow DIV_{\rho}(\overline{b}.SC_1, \text{assert})
\]

where \( \rho \subseteq mref(\overline{b}.SC_1) \).

**Case 2.1.2.1:**
In the case where \( b \notin \rho \).

By Definition 6.1.6,

\[
\bigwedge_{z \in \rho} DIV_{\{z\}}(\overline{b}.SC_1, \text{assert})
\]

because \( b \notin \rho \)

\[
= \bigwedge_{z \in \rho} DIV_{\{z\}\backslash mref(SC_1)}(SC_1, \text{assert})
\]

= the same as

\[
\bigwedge_{z \in (\rho \backslash mref(SC_1))} DIV_{\{z\}}(SC_1, \text{assert}) \land DIV_{\emptyset}(SC_1, \text{assert})
\]

\[
\Rightarrow \text{ by inductive hypothesis}
\]

\[
DIV_{\rho \backslash mref(SC_1)}(SC_1, \text{assert}) \land DIV_{\emptyset}(SC_1, \text{assert})
\]

By Definition 6.1.6,

\[
DIV_{\rho}(\overline{b}.SC_1, \text{assert}) = DIV_{\rho \backslash mref(SC_1)}(SC_1, \text{assert})
\]

Thus we have the case.

**Case 2.1.2.2:**
Suppose \( b \in \rho \).
By Definition 6.1.6,
\[
\bigwedge_{z \in \rho} \text{DIV}_{\{z\}}(\overline{b \cdot SC_1}, \text{assert})
\]
\[=\]
\[
\text{DIV}_{\{b\}}(\overline{b \cdot SC_1}, \text{assert}) \land \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(\overline{b \cdot SC_1}, \text{assert})
\]
\[=\]
\[
\text{trm}(b \cdot op(l)) \land \forall b \cdot \text{VARIABLES}'(\text{prd}_{b \cdot \text{VARIABLES}}(b \cdot op(l)) \Rightarrow \wp(b \cdot \text{VARIABLES} := b \cdot \text{VARIABLES}', \text{DIV}_{\{b\}}(SC_1, \text{assert}))
\]
\[
\land \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})
\]
\[=\]
\[
\text{trm}(b \cdot op(l)) \land \forall b \cdot \text{VARIABLES}'(\text{prd}_{b \cdot \text{VARIABLES}}(b \cdot op(l)) \Rightarrow \wp(b \cdot \text{VARIABLES} := b \cdot \text{VARIABLES}', \text{DIV}_{\{b\}}(SC_1, \text{assert}))
\]
\[
\land \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})
\]
\[\Rightarrow\]
\[
\text{trm}(b \cdot op(l)) \land \forall b \cdot \text{VARIABLES}'(\text{prd}_{b \cdot \text{VARIABLES}}(b \cdot op(l)) \Rightarrow \wp(b \cdot \text{VARIABLES} := b \cdot \text{VARIABLES}', \text{DIV}_{\{b\}}(SC_1, \text{assert}))
\]
\[
\land \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})
\]
\] since we can move the conjuncts under the \( \forall \) statement because
\[
(f_n( \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})) \cap \text{set-of}(b \cdot \text{VARIABLES}') = \emptyset
\]
\[
(f_n( \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})) \cap \text{set-of}(b \cdot \text{VARIABLES}) = \emptyset)
\]
\[=\]
\[
\text{trm}(b \cdot op(l)) \land \forall b \cdot \text{VARIABLES}'(\text{prd}_{b \cdot \text{VARIABLES}}(b \cdot op(l)) \Rightarrow \wp(b \cdot \text{VARIABLES} := b \cdot \text{VARIABLES}', \text{DIV}_{\{b\}}(SC_1, \text{assert}) \land \bigwedge_{z \in \rho \setminus \{b\}} \text{DIV}_{\{z\}}(SC_1, \text{assert})
\]
\] through distributivity of \( \wp \) and disjoint assignment,
\[=\]
\[
\text{trm}(b \cdot op(l)) \land \forall b \cdot \text{VARIABLES}'(\text{prd}_{b \cdot \text{VARIABLES}}(b \cdot op(l)) \Rightarrow \wp(b \cdot \text{VARIABLES} := b \cdot \text{VARIABLES}', \bigwedge_{z \in \rho} \text{DIV}_{\{z\}}(SC_1, \text{assert}))
\]
6.3. Some useful results about $DIV$

\[
\therefore \text{by inductive hypothesis} \\
\text{trm}(b_{op}(l)) \land \forall b_{VAR}\text{iables'}(\text{prd}_{b_{VAR}}(b_{op}(l)) \Rightarrow \wp(b_{VAR}\text{iables} := b_{VAR}\text{iables}', \ DIV_{\rho}(SC_1, assert)))
\]

\[
\therefore \text{Theorem 2.5.1} \\
\wp(b_{op}(l); convert(SC_1)_{\rho}, true)
\]

By Definition 6.1.6,

\[
DIV_{\rho}(\overline{b}.SC_1, assert) = \wp(b_{op}(l); convert(SC_1)_{\rho}, true)
\]

Thus we have the case.

Case 2.1.3:
Let $SC = \overline{c}(b).SC_1$ where $c$ is a control point and $b$ is a machine reference.

(Note that $\overline{c}(b).SC_1$ is equivalent to executing an operation where the pre-condition is equivalent to $b_{-assert}(c)$.)

Hence the case is similar to case 2.1.2.

Case 2.1.4:
For any prefix $\pi$ not covered by the cases above suppose $SC = \pi.SC_1$. The proof is similar to Case 2.1.2.1

Case 2.1.5:
Where $v$ is a standard name $SC = (v \ v : S)(SC_1)$. The proof is similar to Case 2.1.2.1

Case 2.1.6:
Assume that the inductive hypothesis extends over agent $SC_2$.

Suppose $SC = SC_1 + SC_2$.

Then we need to show that,

\[
\bigwedge_{x \in \rho} DIV_{\{x\}}(SC_1 + SC_2, assert) \Rightarrow DIV_{\rho}(SC_1 + SC_2, assert)
\]

where $\rho \subseteq mref(SC_1 + SC_2)$.
By Definition 6.1.6,
\[
\bigwedge_{z \in p} \text{DIV}_{\{z\}}(SC_1 + SC_2, \text{assert}) = \\
\bigwedge_{z \in p} (\text{DIV}_{\{z\}} \cap \text{mref}(SC_1)(SC_1, \text{assert}) \land \text{DIV}_{\{z\}} \cap \text{mref}(SC_1)(SC_2, \text{assert}))
\]
\[
= \\
\bigwedge_{z \in (\rho \cap \delta)(SC_1)} \text{DIV}_{\{z\}}(SC_1, \text{assert}) \land \text{DIV}_0(\text{SC}_1, \text{assert}) \land \\
\bigwedge_{z \in (\rho \cap \delta)(SC_2)} \text{DIV}_{\{z\}}(SC_2, \text{assert}) \land \text{DIV}_0(\text{SC}_2, \text{assert})
\]
\[
\Rightarrow \text{ by inductive hypothesis}
\]
\[
\text{DIV}_{\rho \cap \delta}(SC_1, \text{assert}) \land \text{DIV}_{\rho \cap \delta}(SC_2, \text{assert}) \land \\
\text{DIV}_0(SC_1, \text{assert}) \land \text{DIV}_0(SC_2, \text{assert})
\]

By Definition 6.1.6,
\[
\text{DIV}_0(SC_1 + SC_2, \text{assert}) = \text{DIV}_{\rho \cap \delta}(SC_1, \text{assert}) \land \\
\text{DIV}_{\rho \cap \delta}(SC_2, \text{assert})
\]
Thus we have the case.

(Note the case $\Rightarrow$ is true by virtue of Lemma 6.3.1. (1)). □

The following Lemma is used in the proof of Lemma 6.3.3.

Lemma 6.3.2. For any sequential finite controller $SC$, $(\rho \cap \delta) \subseteq \text{mref}(SC)$, and function assert closed with respect to $SC$.
\[
\bigwedge_{z \in \rho \cap \delta} \text{DIV}_{\{z\}}(SC, \text{assert}) \Rightarrow \bigwedge_{z \in \rho \cap \delta} \text{DIV}_{\{z\}}(SC, \text{assert})
\]

Proof. By contradiction.

Assume that,
\[
\bigwedge_{z \in (\rho \cap \delta)} \text{DIV}_{\{z\}}(SC, \text{assert}) \land \neg (\bigwedge_{z \in \rho \cap \delta} \text{DIV}_{\{z\}}(SC, \text{assert}))
\]
\[
\Rightarrow \\
\bigwedge_{z \in (\rho \cap \delta)} \text{DIV}_{\{z\}}(SC, \text{assert}) \land \neg (\bigwedge_{z \in (\rho \cap \delta)} \text{DIV}_{\{z\}}(SC, \text{assert}) \land \text{DIV}_0(SC, \text{assert}))
\]
\[
= \\
\bigwedge_{z \in (\rho \cap \delta)} \text{DIV}_{\{z\}}(SC, \text{assert}) \land \neg \text{DIV}_0(SC, \text{assert})
\]
Then either $\rho \cap \delta = \emptyset$ or $\rho \cap \delta \neq \emptyset$.

In the first case we obtain a contradiction directly,

$$\text{DIV}_\emptyset(SC, \text{assert}) \land \neg \text{DIV}_\emptyset(SC, \text{assert})$$

In the second case for some $p$,

$$\text{DIV}_{\{p\}}(SC, \text{assert}) \land \bigwedge_{z \in (p \cap \delta) \setminus \{p\}} \text{DIV}_z(SC, \text{assert}) \land \neg \text{DIV}_\emptyset(SC, \text{assert})$$

We have that $\emptyset \subseteq \{p\}$ thus by Lemma 6.3.1 the above implies,

$$\text{DIV}_\emptyset(SC, \text{assert}) \land \bigwedge_{z \in (p \cap \delta) \setminus \{p\}} \text{DIV}_z(SC, \text{assert}) \land \neg \text{DIV}_\emptyset(SC, \text{assert})$$

which is a contradiction.

Thus the result is true. \hfill \Box

We extend Lemma 6.3.1 which was restricted to sequential finite controllers, to all mediators. Using the Lemma below we can justify breaking up the weakest pre-condition proof for a combined system to smaller localised proof obligations.

**Lemma 6.3.3.** For any mediator $D$, $\rho \subseteq \text{mref}(D)$ and some function assert closed with respect to $D$,

1. $\text{DIV}_\rho(D, \text{assert}) \Rightarrow \text{DIV}_\delta(D, \text{assert})$ for any $\delta \subseteq \rho$,
2. $\bigwedge_{z \in \rho} \text{DIV}_z(D, \text{assert}) \iff \text{DIV}_\rho(D, \text{assert})$

**Proof.** By structural induction on mediator $D$,

Case 1:

Case 1.1 (Base Case):
Suppose $D = SC$ for some sequential finite controller $SC$.

Then the case is true by Lemma 6.3.1.
Case 1.2 (Inductive Case):
Assume that for mediator \( D_1, \rho \subseteq mref(D_1) \) and some function \( \text{assert} \) closed with respect to \( D_1, \)
\[
DIV_\rho(D_1, \text{assert}) \Rightarrow DIV_\delta(D_1, \text{assert})
\]
for any \( \delta \subseteq \rho. \)

Case 1.2.1:
Suppose \( D = SC \mid D_1 \) for some sequential finite controller.

Then by Definition 6.1.5,
\[
DIV_\rho(D, \text{assert}) = DIV_{\rho \cap mref(SC)}(SC, \text{assert}) \land DIV_{\rho \cap mref(D_1)}(D_1, \text{assert})
\]
\[
\Rightarrow \text{ by inductive hypothesis and Lemma 6.3.1}
\]
\[
DIV_{\delta \cap mref(SC)}(SC, \text{assert}) \land DIV_{\delta \cap mref(D_1)}(D_1, \text{assert})
\]

Then by Definition 6.1.5,
\[
DIV_\delta(D, \text{assert}) = DIV_{\delta \cap mref(SC)}(SC, \text{assert}) \land DIV_{\delta \cap mref(D_1)}(D_1, \text{assert})
\]

Thus we have the case.

Case 1.2.2:
Suppose \( D = (\nu \ cp^R : S)(D_1) \) for some control point type \( S. \)

Then by Definition 6.1.5,
\[
DIV_\rho(D, \text{assert}) = DIV_\rho(D', \text{assert} \cup \{(cp, A)\})
\]
\[
\Rightarrow \text{ by inductive hypothesis}
\]
\[
DIV_\delta(D', \text{assert} \cup \{(cp, A)\})
\]

Then by Definition 6.1.5,
\[
DIV_\delta(D, \text{assert}) = DIV_\delta(D', \text{assert} \cup \{(cp, A)\})
\]

Thus we have the case.
6.3. Some useful results about \( \text{DIV} \)

Case 1.2.3:
Suppose \( D = (\nu \; v : S)(D_1) \) for some \( S \) that is not a control point type. The proof is similar to Case 1.2.2.

Case 2:
Instead of performing structural induction on \( D \) note that, by Lemma 3.2.2 and Lemmas 5.2.1 and 5.2.2 we have that for every \( D \),

\[
D \equiv \pi_2 (\nu \; \vec{x})(SC_1 \mid \ldots \mid SC_n)
\]

for some list of names \( \vec{x} \) and sequential finite controllers \( SC_1, \ldots, SC_n \).

By applying Lemma 6.3.1 \( n \) times we have that for each \( i, 1 \leq i \leq n \),

\[
\bigwedge_{z \in (\rho \cap \text{mref}(SC_i))} \text{DIV}_{\{z\}}(SC_i, \text{assert}) \leftrightarrow \text{DIV}_{\rho \cap \text{mref}(SC_i)}(SC_i, \text{assert})
\]

Thus in the \( \Rightarrow \) case,

\[
\bigwedge_{z \in \rho} \text{DIV}_{\{z\}}(D, \text{assert})
= \bigwedge_{z \in \rho} \bigwedge_{1 \leq i \leq n} \text{DIV}_{\{z\} \cap \text{mref}(SC_i)}(SC_i, \text{assert})
= \bigwedge_{1 \leq i \leq n} (\bigwedge_{z \in (\rho \cap \text{mref}(SC_i))} \text{DIV}_{\{z\}}(SC_i, \text{assert})) \wedge \bigwedge_{1 \leq i \leq n} \text{DIV}_{\emptyset}(SC_i, \text{assert})
= \bigwedge_{1 \leq i \leq n} \text{DIV}_{\rho \cap \text{mref}(SC_i)}(SC_i, \text{assert}) \wedge \bigwedge_{1 \leq i \leq n} \text{DIV}_{\emptyset}(SC_i, \text{assert})
= \text{DIV}_{\rho}(D, \text{assert}) \wedge \bigwedge_{1 \leq i \leq n} \text{DIV}_{\emptyset}(SC_i, \text{assert})
\Rightarrow
\text{DIV}_{\rho}(D, \text{assert})
Chapter 6. Verification of a Combined Specification

In the $\leq$ case,
\[
DIV_{\rho}(D, \text{assert}) = \bigwedge_{1 \leq i \leq n} DIV_{\rho \setminus \text{mref}(SC_i)}(SC_i, \text{assert})
\]
\[
\Rightarrow \bigwedge_{1 \leq i \leq n} \bigwedge_{z \in \rho \setminus \text{mref}(SC_i)} DIV_{\{z\}}(SC_i, \text{assert})
\]
\[
\Rightarrow \text{by Lemma 6.3.2(1)}
\]
\[
\bigwedge_{1 \leq i \leq n} \bigwedge_{z \in \rho} DIV_{\{z\}}(D, \text{assert})
\]
\[
= \bigwedge_{z \in \rho} DIV_{\{z\}}(D, \text{assert})
\]

Thus the result is true. \(\square\)

The following result allows us to ignore incoming machine references in the $DIV$ predicate of continuation processes if we are not interested in keeping that information. It considers an input control point action of a mediator $D$ and the impact of the action on the environment variable $\rho$ in the $DIV$ function. This result will be used in the proof of Theorem 6.3.1 when dealing with input control point actions.

**Lemma 6.3.4.** Given a tagged mediator $D$ and function assert closed with respect to $D$, and $\rho \subseteq \text{mref}(D)$.

Suppose $D \overset{cp}{\rightarrow} D'$ for some control point $cp$ and machine reference $z \notin \text{mref}(D)$ then,
\[
DIV_{\rho}(D, \text{assert}) \Rightarrow DIV_{\rho \setminus \text{mref}(D')}(D', \text{assert})
\]

**Proof.** Instead of performing induction on $D \overset{cp}{\rightarrow} D'$ note that by Lemma 3.2.2 and Lemmas 5.2.1 and 5.2.2 we have that for every $D$ and $D'$ we consider,
\[
D \equiv_n (\nu \bar{x})(SC_1 | ... | SC_n)
\]
\[
D' \equiv_n (\nu \bar{y})(SC'_1 | ... | SC'_n)
\]
for some list of names $\bar{x}$, $\bar{y}$ and sequential finite controllers $SC_1, ..., SC_n$ and $SC'_1, ..., SC'_n$.

If $D \overset{cp}{\rightarrow} D'$ then it can be shown that for some $i$, $1 \leq i \leq n$, $SC_i \overset{cp}{\rightarrow} SC'_i$ and for all $j$, $1 \leq j \leq n$, and $j \neq i$, $SC_j =_n SC'_j$.

Thus by Definition 6.1.5, for some function $\text{assert2} \subseteq \text{assert}$
\[
DIV_{\rho}(D, \text{assert}) = \bigwedge_{1 \leq \sigma \leq n} DIV_{\rho \setminus \text{mref}(SC_{\sigma})}(SC_{\sigma}, \text{assert})
\]
and
\[
DIV_{\rho}(D', \text{assert}) = \bigwedge_{1 \leq \sigma \leq n} DIV_{\rho \setminus \text{mref}(SC'_{\sigma})}(SC'_{\sigma}, \text{assert})
\]
6.3. Some useful results about $DIV$

and for all $j$, $1 \leq j \leq n$ and $j \neq i$,

$$DIV_{\rho \cap \text{mref}(SC)_{j}}(SC_{j}, \text{assert2}) = DIV_{\rho \cap \text{mref}(SC)_{j}}(SC'_{j}, \text{assert2})$$

Thus to show that the proposition holds, it is sufficient to perform induction over $SC_{i} \leftarrow^{\rho, j} SC'_{i}$ and show that,

$$DIV_{\rho \cap \text{mref}(SC)_{i}}(SC_{i}, \text{assert2}) = DIV_{\rho \cap \text{mref}(SC)_{i}}(SC'_{i}, \text{assert2})$$

Case 1 (Base Case):
Suppose by rule $INP$ we can conclude that,

$$SC_{i} \equiv cp(b).SC_{1}$$

Thus $SC'_{i} \equiv SC_{1}\{z/b\}$.

We need to show that,

$$(DIV_{\rho \cap \text{mref}(SC_{i})}(SC_{i}, \text{assert2}) \land z_{-\text{assert}}(cp)) = DIV_{\rho \cap \text{mref}(SC_{i})}(SC'_{i}, \text{assert2})$$

By Definition 6.1.6,

$$DIV_{\rho \cap \text{mref}(SC_{i})}(SC_{i}, \text{assert2}) = b_{-\text{assert}}(cp) = DIV_{\rho \cap \text{mref}(SC_{i})}(SC_{1}, \text{assert2})$$

where $b \notin \rho$.

Since $z \notin \text{mref}(SC_{i})$ we can substitute $z$ for $b$ in the predicate without changing the result thus we have that,

$$DIV_{\rho \cap \text{mref}(SC_{i})}(SC_{i}, \text{assert2}) = z_{-\text{assert}}(cp) = DIV_{\rho \cap \text{mref}(SC_{1}\{z/b\})}(SC_{1}\{z/b\}, \text{assert2})$$

From Lemma 6.3.1 we have that,

$$DIV_{\rho \cap \text{mref}(SC_{1}\{z/b\})}(SC_{1}\{z/b\}, \text{assert2}) = (DIV_{\rho \cap \text{mref}(SC_{i})}(SC'_{i}, \text{assert2}) \land DIV_{\{z\} \cap \text{mref}(SC_{1}\{z/b\})}(SC_{i}, \text{assert2}))$$
Thus we have that,

\[(z\text{-assert}(cp) \land \text{true}) \Rightarrow \]

\[\left(\text{DIV}_{\rho \text{\neg mref}(SC_i)}(SC'_i, \text{assert}2) \land \text{DIV}_{\{z\}_\text{\neg mref}(SC'_i)}(SC'_i, \text{assert}2)\right)\]

\[\Rightarrow \text{as } z \notin \text{mref}(SC_i) \text{ hence } z \notin \rho \]

\[\left(\text{true} \Rightarrow \text{DIV}_{\rho \text{\neg mref}(SC_i)}(SC'_i, \text{assert}2)\right)\]

\[\land \]

\[\left(z\text{-assert}(cp) \Rightarrow \text{DIV}_{\{z\}_\text{\neg mref}(SC'_i)}(SC'_i, \text{assert}2)\right)\]

\[\Rightarrow \]

\[\left(\text{true} \Rightarrow \text{DIV}_{\rho \text{\neg mref}(SC'_i)}(SC'_i, \text{assert}2)\right)\]

\[\Rightarrow \]

\[\text{DIV}_{\rho \text{\neg mref}(SC'_i)}(SC'_i, \text{assert}2)\]

Thus the case holds.

Case 1.1 (Inductive Case):
Assume that for SC1 and function assert2 closed with respect to SC1, and \(\rho \subseteq \text{mref}(SC1)\).

If \(SC1 \xrightarrow{cp} SC1'\) for some control point cp and machine reference \(z \notin \text{mref}(SC1)\) then,

\[\text{DIV}_\rho(SC1, \text{assert}2) \Rightarrow \text{DIV}_{\rho \text{\neg mref}(SC1')}((SC1', \text{assert}2))\]

Case 1.1.1:
Suppose by rule MAT we conclude that,

\[SC_i \doteq \{w = w\}SC1\]

such that \(SC'_i \doteq SC1'\), for some standard name w.

Then we need to show that,

\[\text{DIV}_\rho(SC_i, \text{assert}2) \Rightarrow \text{DIV}_{\rho \text{\neg mref}(SC1')}((SC', \text{assert}2))\]

By Definition 6.1.6,

\[\text{DIV}_\rho(SC_i, \text{assert}2) = \text{DIV}_\rho(SC1, \text{assert}2)\]

\[\Rightarrow \text{by inductive hypothesis}\]

\[\text{DIV}_{\rho \text{\neg mref}(SC1')}((SC1', \text{assert})\]
Thus we have the case.

Case 1.1.2:
Suppose by rule $RES$ we conclude that,

$$SC_i \equiv (\nu \ v : S)SC1$$

such that $SC_i' \equiv SC1'$, for some standard name $v$.

Then the case is similar to 1.1.1.

Case 1.1.3:
Suppose the inductive hypothesis extends over agent $SC2$.

Suppose by rule $SUM-L$ we conclude that,

$$SC_i \equiv SC1 + SC2$$

such that $SC_i' \equiv SC1'$.

By Definition 6.1.6,

$$DIV_{\rho}(SC_i, \ asserts2) = DIV_{\rho_{\text{mref}(SC1)}}(SC1, \ asserts2) \land DIV_{\rho_{\text{mref}(SC2)}}(SC2, \ asserts2)$$

$\Rightarrow$ by inductive hypothesis

$$DIV_{\rho_{\text{mref}(SC1')}}(SC1', \ asserts) \land DIV_{\rho_{\text{mref}(SC2)}}(SC2, \ asserts2)$$

$\Rightarrow$

$$DIV_{\rho_{\text{mref}(SC1')}}(SC1', \ asserts)$$

Thus we have the case.

Case 1.1.4:
Suppose the inductive hypothesis extends over agent $SC2$.

Suppose by rule $SUM-R$ we conclude that,

$$SC_i \equiv SC1 + SC2$$

such that $SC_i' \equiv SC2'$.

Then the case is similar to 1.1.3.
The following theorem allows us to infer information about the \( \text{DIV} \) predicates of subsequent evolutions of a mediator in an analogous way to a subject reduction theorem. Essentially, it shows the relationship between actions of a mediator and the \( \text{DIV} \) function. It is an important result used when we consider actions of control systems in the proofs of the theorems in Section 6.5.

**Theorem 6.3.1.** Given a tagged mediator \( D \) and function assert closed with respect to \( D \), and \( \rho \subseteq \text{mref}(D) \).

Suppose \( D \xrightarrow{\alpha} D' \) then,

1. if \( \alpha = \text{cp} z \) for some control point \( \text{cp} \) and machine reference \( z \notin \text{mref}(D) \) then,

\[
( \text{DIV}_\rho(D, \text{assert}) \land z_\text{assert}(\text{cp})) \Rightarrow \text{DIV}_{(\rho \cup \{z\}) \cap \text{mref}(D')}(D', \text{assert})
\]

2. if \( \alpha = \text{cp} z \) for some control point \( \text{cp} \) and machine reference \( z \in \rho \) then,

\[
\text{DIV}_\rho(D, \text{assert}) \Rightarrow (\text{DIV}_{\rho \cap \text{mref}(D')}(D', \text{assert}) \land z_\text{assert}(\text{cp}))
\]

3. if \( \alpha = \neg z \) for some machine reference \( z \in \rho \) and operation label \( l \) then,

\[
\text{DIV}_\rho(D, \text{assert}) \Rightarrow \\
\text{trm}(z_\text{op}(l)) \land \\
\exists z_\text{VARIABLES}', (\text{prd}_z_\text{VARIABLES}(z_\text{op}(l)) \Rightarrow \\
\text{wp}(z_\text{VARIABLES} := z_\text{VARIABLES}', \text{DIV}_{\rho \cap \text{mref}(D')}(D', \text{assert})))
\]

where \( z_\text{VARIABLES}' \) denotes a list of variable names each prefixed with \( z_\text{ and primed} \).

4. in all other cases,

\[
\text{DIV}_\rho(D, \text{assert}) \Rightarrow \text{DIV}_{\rho \cap \text{mref}(D')}(D', \text{assert})
\]

**Proof.** By similar argument as in Lemma 6.3.4 we perform induction over \( SC_i \xrightarrow{\alpha} SC_i' \) for some sequential finite controller \( SC_i \) in \( D \) and \( SC_i' \) in \( D' \) for the first three cases. As the fourth case might involve a \( \tau \) which is the result of process communication, we need to consider that the induction hypothesis extends to an \( SC_j \) in \( D \) such that \( j \neq i \).

Case 1:
Suppose \( \alpha = \text{cp} z \) for some control point \( \text{cp} \) and machine reference \( z \notin \text{mref}(SC_i) \).

We need to show that,

\[
(\text{DIV}_{\rho \cap \text{mref}(SC_i)}(SC_i, \text{assert}_2) \land z_\text{assert}(\text{cp})) \Rightarrow \text{DIV}_{(\rho \cup \{z\}) \cap \text{mref}(SC_i')} (SC_i', \text{assert}_2)
\]
Case 1.1 (Base Case):
Suppose by rule INP we can conclude that,

$$SC_i \equiv cp(b).SC_i$$

Thus \(SC'_i \equiv SC_i \{^* / b\}\).

By Definition 6.1.6,

$$DIV_{\rho \cap mref(SC_i)}(SC_i, assert2) =$$

$$b \_ assert2(cp) \Rightarrow DIV_{(\rho \cup \{b\}) \cap mref(SC_i)}(SC_i, assert2)$$

where \(b \notin \rho\).

Since \(z \notin mref(SC_i)\) we can substitute \(z\) for \(b\) in the predicate without changing the result thus we have that,

$$DIV_{\rho \cap mref(SC_i)}(SC_i, assert2) =$$

$$z \_ assert2(cp) \Rightarrow DIV_{(\rho \cup \{z\}) \cap mref(SC_i)}(SC_i \{^* / b\}, assert2) =$$

$$z \_ assert2(cp) \Rightarrow DIV_{(\rho \cup \{z\}) \cap mref(SC_i)}(SC'_i, assert2)$$

Thus we need to show that

$$(z \_ assert2(cp) \Rightarrow DIV_{(\rho \cup \{z\}) \cap mref(SC'_i)(SC'_i, assert2)) \land z \_ assert(cp)$$

$$\Rightarrow$$

$$DIV_{(\rho \cup \{z\}) \cap mref(SC'_i)}(SC'_i, assert2)$$

We have that assert2 \(\subseteq\) assert hence the statement is logically true.

Thus the case holds.

Case 1.1 (Inductive Case):
Assume that for \(SC_1\) and function assert2 closed with respect to \(SC_1\), and \(\rho \subseteq mref(SC_1)\).

If \(SC_1 \stackrel{cp}{\rightarrow} SC'_1\) for some control point \(cp\) and machine reference \(z \notin mref(SC_1)\) then,

$$(DIV_{\rho}(SC_1, assert2) \land z \_ assert(cp)) \Rightarrow DIV_{(\rho \cup \{z\}) \cap mref(SC'_1)}(SC'_1, assert2)$$
Case 1.1.1:
Suppose by rule MAT we conclude that,
\[ SC_i \equiv [w = w]SC1 \]
such that \( SC'_i \equiv SC1' \), for some standard name \( w \).

By Definition 6.1.6,
\[ DIV_p(SC_1, \text{assert}2) = DIV_p(SC1, \text{assert}2) \]
Thus we have the case from the inductive hypothesis.

Case 1.1.2:
Suppose by rule RES we conclude that,
\[ SC_i \equiv (\nu v : S)SC1 \]
such that \( SC'_i \equiv SC1' \), for some standard name \( v \).

Then the case is similar to 1.1.1,

Case 1.1.3:
Suppose by rule SUM-L we conclude that,
\[ SC_i \equiv SC1 + SC2 \]
such that \( SC'_i \equiv SC1' \).

By Definition 6.1.6,
\[ DIV_p(SC_1, \text{assert}2) = DIV_p(SC1, \text{assert}2) \land DIV_p(SC2, \text{assert}2) \Rightarrow DIV_p(SC1, \text{assert}2) \]
Thus the case follows from the inductive hypothesis.

Case 1.1.4:
Suppose the inductive hypothesis extends over agent \( SC2 \).

Suppose by rule SUM-R we conclude that,
\[ SC_i \equiv SC1 + SC2 \]
such that $SC'_i \equiv SC2'$.

Then the case is similar to 1.1.3.

Case 2:
Suppose $\alpha = \overline{cp} z$ for some output control point $cp$ and machine reference $z \in \rho$.

We need to show that,
$$DIV_{\rho mref(SC_i)}(SC_i, assert2) \Rightarrow (DIV_{\rho mref(SC'_i)}(SC'_i, assert2) \land z\_assert(cp))$$

Case 2.1 (Base Case):
Suppose by rule OUT we infer that,
$$SC_i \equiv \overline{cp}(z) SC'_i$$

By Definition 6.1.6 we conclude that,
$$DIV_{\rho}(SC_i, assert2) = wp(convert(\overline{cp}(z) SC'_i)_{\rho}, true)$$
$$= wp(PRE z\_assert2(cp) THEN skip END; convert(SC'_i) \rho mref(SC'_i), true)$$
since $assert \subseteq assert2$
$$= wp(PRE z\_assert(cp) THEN skip END; convert(SC'_i) \rho mref(SC'_i), true)$$
by Definition 2.4.3
$$= z\_assert(cp) \land wp(convert(SC'_i) \rho mref(SC'_i), true)$$
by Definition 6.1.6
$$= z\_assert(cp) \land DIV_{\rho mref(SC'_i)}(SC'_i, assert2)$$

Thus we have the case.

Case 2.2 (Inductive Case):
Assume that for mediator $SC1$ and function $assert2$ closed with respect to $SC1$, and $\rho \subseteq mref(SC1)$, if $SC1 \overline{cp} z SC1'$ then,
$$DIV_{\rho mref(SC1)}(SC1, assert2) \Rightarrow (DIV_{\rho mref(SC1')} (SC1', assert2) \land z\_assert2(cp))$$

Case 2.2.1:
Suppose by rule MAT we conclude that,
$$SC_i \equiv [w = w]SC1$$
such that $SC_i' \triangleq SC1'$, for some standard name $w$.

By Definition 6.1.6,

$$DIV_{\rho}(SC_i, assert2) = DIV_{\rho}(SC1, assert2)$$

Thus we have the case from the inductive hypothesis.

Case 2.2.2:
Suppose by rule $RES$ we conclude that,

$$SC_i \triangleq (v : S)SC1$$

such that $SC_i' \triangleq SC1'$, for some standard name $v$.

Then the case is similar to 2.2.1,

Case 2.2.3:
Suppose by rule $SUM-L$ we conclude that,

$$SC_i \triangleq SC1 + SC2$$

such that $SC_i' \triangleq SC1'$.

By Definition 6.1.6,

$$DIV_{\rho}(SC_i, assert2) = DIV_{\rho \cap \text{mref}(SC1)}(SC1, assert2) \land DIV_{\rho \cap \text{mref}(SC2)}(SC2, assert2)$$

$$\Rightarrow DIV_{\rho \cap \text{mref}(SC1)}(SC1, assert2)$$

Thus the case follows from the inductive hypothesis.

Case 2.2.4:
Suppose the inductive hypothesis extends over agent $SC2$.

Suppose by rule $SUM-R$ we conclude that,

$$SC_i \triangleq SC1 + SC2$$

such that $SC_i' \triangleq SC2'$.

Then the case is similar to 2.2.3.
6.3. Some useful results about DIV

Case 3:
Suppose $\alpha = \overline{z} l$ for some machine reference $z \in \rho$ and operation label $l$.

Then we need to show that,

$$DIV_{\rho \setminus \text{mref}(SC_i)}(SC_i, \text{assert}2) \Rightarrow trm(z \_ op(l)) \land \exists z \_ \text{VARIABLES}' \ (\text{prd}_{z \_ \text{VARIABLES}}(z \_ op(l)))$$
$$\Rightarrow wp(z \_ \text{VARIABLES} := z \_ \text{VARIABLES}' \ , \ DIV_{\rho \setminus \text{mref}(SC_i)}(SC'_i, \text{assert}2))$$

where $z \_ \text{VARIABLES}'$ denotes a list of variables where each name is prefixed with $z_-$ and primed.

Case 3.1 (Base Case):
Suppose by application of $OUT$ we conclude that,

$$SC_i \equiv \overline{z}! . SC'_i$$

By Definition 6.1.6 we conclude that for some function $\text{assert}2$ such that $\text{assert} \subseteq \text{assert}2$ and it is closed with respect to $SC_i$,

$$DIV_{\rho \setminus \text{mref}(SC_i)}(SC_i, \text{assert}2) = wp(\text{convert}(\overline{z}! . SC'_i)_{\rho \setminus \text{mref}(SC_i)}, \text{true})$$
$$= wp(z \_ op(l); \ \text{convert}(SC'_i)_{\rho \setminus \text{mref}(SC_i)}, \text{true})$$
$$\Rightarrow \text{by Theorem 2.5.2}$$
$$trm(z \_ op(l)) \land \exists z \_ \text{VARIABLES}' \ (\text{prd}_{z \_ \text{VARIABLES}}(z \_ op(l)))$$
$$\Rightarrow wp(z \_ \text{VARIABLES} := z \_ \text{VARIABLES}' \ , \ trm(\text{convert}(SC'_i)_{\rho \setminus \text{mref}(SC_i)}))$$
$$= trm(z \_ op(l)) \land \exists z \_ \text{VARIABLES}' \ (\text{prd}_{z \_ \text{VARIABLES}}(z \_ op(l)))$$
$$\Rightarrow wp(z \_ \text{VARIABLES} := z \_ \text{VARIABLES}' \ , \ DIV_{\rho \setminus \text{mref}(SC_i)}(SC'_i, \text{assert}2))$$

Thus we have the case.

Case 3.2 (Inductive Case):
Assume that for mediator $SC_1$ and function $\text{assert}2$ closed with respect to $SC_1$, and

$$\rho \subseteq \text{mref}(SC_1) \text{ if } SC_1 \xrightarrow{\overline{z} l} SC'_1 \text{ for some machine reference } z \text{ and operation label } l$$

then,

$$DIV_{\rho \setminus \text{mref}(SC_1)}(SC_1, \text{assert}2)$$
$$\Rightarrow trm(z \_ op(l)) \land \exists z \_ \text{VARIABLES}' \ (\text{prd}_{z \_ \text{VARIABLES}}(z \_ op(l)))$$
$$\Rightarrow wp(z \_ \text{VARIABLES} := z \_ \text{VARIABLES}' \ , \ DIV_{\rho \setminus \text{mref}(SC_1)}(SC_1', \text{assert}2))$$
Case 3.2.1:
Suppose by rule MAT we conclude that,

\[ SC_i = [w = w]SC \]

such that \( SC_i' = SC_i1' \), for some standard name \( w \).

By Definition 6.1.6,

\[ DIV(\rho, SC_i, \text{assert}2) = DIV(\rho, SC1, \text{assert}2) \]

Thus we have the case from the inductive hypothesis.

Case 3.2.2:
Suppose by rule RES we conclude that,

\[ SC_i = (\nu v : S)SC \]

such that \( SC_i' = SC1' \), for some standard name \( v \).

Then the case is similar to 3.2.1,

Case 3.2.3:
Suppose by rule SUM-L we conclude that,

\[ SC_i = SC1 + SC2 \]

such that \( SC_i' = SC1' \).

By Definition 6.1.6,

\[ DIV(\rho, SC_i, \text{assert}2) = DIV(\rho, \text{mref}(SC1)(SC1, \text{assert}2) \wedge DIV(\rho, \text{mref}(SC2)(SC2, \text{assert}2) \Rightarrow \]

\[ DIV(\rho, \text{mref}(SC1)(SC1, \text{assert}2) \]

Thus the case follows from the inductive hypothesis.

Case 3.2.4:
Suppose the inductive hypothesis extends over agent \( SC2 \).
6.3. Some useful results about $DIV$

Suppose by rule $SUM-R$ we conclude that,

\[ SC_i \equiv SC1 + SC2 \]

such that $SC'_i \equiv SC2'$.

Then the case is similar to 3.2.3.

Case 4:
Any case of $\alpha$ not covered by cases 1 to 3.

We must show that,

\[ DIV_{\rho \cap mref(SC_i)}(SC_i, \text{assert}) \Rightarrow DIV_{\rho \cap mref(SC'_i)}(SC'_i, \text{assert}) \]

Case 4.1 (Base Case):
Suppose $\alpha = a \ w$ for some standard names $a$ and $w$. Suppose by application of $INP$ we conclude that,

\[ SC_i \equiv a(b).SC1_i \]

for some process $SC1_i$ and name $b$, such that $SC'_i \equiv SC1_i\{w/b\}$.

By Definition 6.1.6 we conclude that for some function $\text{assert2}$ such that $\text{assert} \subseteq \text{assert2}$ and it is closed with respect to $SC_i$,

\[ DIV_{\rho \cap mref(SC_i)}(SC_i, \text{assert2}) = wp(convert(a(b).SC1_i)_{\rho \cap mref(SC_i)}, \text{true}) \]
\[ = mref(SC_i) = mref(SC1_i) \]
\[ = wp(convert(SC1_i)_{\rho \cap mref(SC1_i)}, \text{true}) \]
\[ = DIV_{\rho \cap mref(SC1_i)}(SC1_i, \text{assert2}) \]

Since $w$ and $b$ are standard names we can substitute them in the above expression without affecting $DIV$ hence we have that,

\[ = DIV_{\rho \cap mref(SC1_i\{w/b\})}(SC1_i\{w/b\}, \text{assert2}) \]

Thus we have the case.

Case 4.2:
Suppose $\alpha = \overline{a} \ w$ for some standard names $a$ and $w$. Suppose by application of $OUT$ we conclude that,

\[ SC_i \equiv \overline{a}(w).SC'_i \]
By Definition 6.1.6 we conclude that for some function $assert2$ such that $assert \subseteq assert2$ and it is closed with respect to $SC_i$,

$$DIV_{\rho \setminus \text{mref}(SC_i)}(SC_i, assert2) = wp(convert(\bar{a}(w).SC_i, SC_i), true)$$

$$= mref(SC_i) = mref(SC'_i)$$

$$= wp(convert(SC'_i, SC_i), true)$$

by Definition 6.1.5

$$= DIV_{\rho \setminus \text{mref}(SC_i)}(SC'_i, assert2)$$

Thus we have the case.

Case 4.3:
Suppose $\alpha = \overline{cp} \ z$ for some output control point $cp$ and machine reference $z \notin \rho$.

Suppose by application of $OUT$ we conclude that,

$$SC_i \triangleq \overline{cp}(z).SC'_i$$

By Definition 6.1.6 we conclude that for some function $assert2$ such that $assert \subseteq assert2$ and it is closed with respect to $SC_i$,

$$DIV_{\rho \setminus \text{mref}(SC_i)}(SC_i, assert2) = wp(convert(\overline{cp}(z).SC'_i, SC_i), true)$$

$$= z \notin \rho$$

$$wp(convert(SC'_i, SC_i), true)$$

by Definition 6.1.5

$$= DIV_{\rho \setminus \text{mref}(SC_i)}(SC'_i, assert2)$$

Thus we have the case.

Case 4.4:
Suppose $\alpha = \overline{z} \ l$ for some machine reference $z \notin \rho$.

Suppose by application of $OUT$ we conclude that,

$$SC_i \triangleq \overline{z} \ l.SC'_i$$
By Definition 6.1.6 we conclude that for some function assert2 such that assert \subseteq assert2 and it is closed with respect to SC_i,

\[ DIV_{\rho \cap \text{mref}(SC_i)}(SC_i, assert2) = \wp(\text{convert}(z \cdot SC_i^{\rho \cap \text{mref}(SC_i)}) \cdot \text{true}) \]

\[ = \begin{cases} z \notin \rho & \wp(\text{convert}(SC_i^{\rho \cap \text{mref}(SC_i)}) \cdot \text{true}) \\ \text{by Definition 6.1.5} & = DIV_{\rho \cap \text{mref}(SC_i)}(SC_i, assert2) \end{cases} \]

Thus we have the case.

Case 4.5:
Suppose \( \alpha = \tau \),

Suppose by rule TAU we conclude that,

\[ SC_i = \tau \cdot SC_i' \]

By Definition 6.1.6 we conclude that for some function assert2 such that assert \subseteq assert2 and it is closed with respect to SC_i,

\[ DIV_{\rho \cap \text{mref}(SC_i)}(SC_i, assert2) = \wp(\text{convert}(\tau \cdot SC_i^{\rho \cap \text{mref}(SC_i)}) \cdot \text{true}) \]

\[ = \text{mref}(SC_i) = \text{mref}(SC_i') \]

\[ = \wp(\text{convert}(SC_i^{\rho \cap \text{mref}(SC_i)}) \cdot \text{true}) \]

\[ \text{by Definition 6.1.5} \]

\[ = DIV_{\rho \cap \text{mref}(SC_i)}(SC_i, assert2) \]

Thus we have the case.

Case 4.6 (Inductive case and \( \alpha = \bar{a}(\nu) \)):
Assume that for mediator SC1 and function assert2 closed with respect to SC1, and \( \rho \subseteq \text{mref}(SC1) \), if \( SC1 \xrightarrow{\alpha} SC1' \) then,

\[ DIV_{\rho \cap \text{mref}(SC1)}(SC1, assert2) \Rightarrow DIV_{\rho \cap \text{mref}(SC1')} (SC1', assert2) \]
Case 4.6.1:
Suppose $\alpha = \overline{a}(v)$ for some standard names $a$ and $v$ and by rule OPEN we conclude that,

$$SC_i \equiv (v \ v : S)SC1$$

such that $SC1 \xrightarrow{\alpha \ v} SC1'$ and $SC1' \equiv SC_i'$.

By Definition 6.1.6 we conclude that for some function assert2 such that $assert \subseteq assert2$ and it is closed with respect to $SC_i$,

$$DIV_{\rho \cdot mref(SC_i)}(SC_i, \text{assert2}) = wp(convert((v \ v)SC1)_{\rho \cdot mref(SC_i)}, \text{true})$$

$$mref(SC_i) = mref(SC1)$$

$$= wp(convert(SC1)_{\rho \cdot mref(SC1)}, \text{true})$$

by Definition 6.1.5

$$= DIV_{\rho \cdot mref(SC1)}(SC1, \text{assert2})$$

Thus we have the case from the inductive hypothesis.

Case 4.6.2:
Suppose by rule MAT we conclude that,

$$SC_i \equiv [w = w]SC1$$

such that $SC_i' \equiv SC1'$, for some standard name $w$.

By Definition 6.1.6,

$$DIV_{\rho}(SC_i, \text{assert2}) = DIV_{\rho}(SC1, \text{assert2})$$

Thus we have the case from the inductive hypothesis.

Case 4.6.3:
Suppose by rule RES we conclude that,

$$SC_i \equiv (v \ v : S)SC1$$

such that $SC_i' \equiv SC1'$, for some standard name $v$.

Then the case is similar to 3.2.1.

Case 4.6.4:
Suppose by rule SUM-L we conclude that,

$$SC_i \equiv SC1 + SC2$$
6.3. Some useful results about $\text{DIV}$

such that $SC'_i \equiv SC'$. 

By Definition 6.1.6,

$$DIV_{p(SC_i, \text{assert2})} = DIV_{p\text{mref}(SC_1)}(SC_1, \text{assert2}) \land DIV_{p\text{mref}(SC_2)}(SC_2, \text{assert2})$$

$$\Rightarrow$$

$$DIV_{p\text{mref}(SC_1)}(SC_1, \text{assert2})$$

Thus the case follows from the inductive hypothesis.

Case 4.6.5:
Suppose the inductive hypothesis extends over agent $SC_2$.

Suppose by rule $\text{SUM-R}$ we conclude that,

$$SC_i \equiv SC_1 + SC_2$$

such that $SC'_i \equiv SC_2'$.

Then the proof is similar to case 4.6.4.

Case 4.7:
Suppose $D \rightarrow D'$ such that for some $i$ and $j$, $1 \leq i, j \leq n$ and $i \neq j$ we have that,

$$SC_i \mid SC_j \rightarrow SC'_i \mid SC'_j$$

or

$$SC_i \mid SC_j \rightarrow (\nu \ v : S)(SC'_i \mid SC'_j)$$

for some standard name $v$.

Assume that for some function $\text{assert2}$ closed with respect to $SC_i$ and $SC_j$ Theorem 6.3.1 holds.

Case 4.7.1:
Suppose by $\text{COMM-L}$ we conclude that for some control point $cp$ and machine reference $z$, $SC_i \xrightarrow{cp \ z} SC'_i$ and $SC_j \xrightarrow{cp \ z} SC'_j$, $z \in \rho$.

From the inductive hypothesis we have that,

$$DIV_{p\text{mref}(SC_i)}(SC_i, \text{assert2}) \Rightarrow (DIV_{p\text{mref}(SC'_i)}(SC'_i, \text{assert2}) \land z_{-\text{assert2}}(cp))$$

$$\land$$

$$(DIV_{p\text{mref}(SC_j)}(SC_j, \text{assert2}) \land z_{-\text{assert2}}(cp)) \Rightarrow DIV_{(p\cup(z))\text{mref}(SC'_j)}(SC'_j, \text{assert2})$$
which implies,

\[ \text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]
\[ \land \]
\[ \text{DIV}_{(\mu \cup \{z\}) \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]

Thus we have the case.

Case 4.7.2:
Suppose by \textit{COMM-L} we conclude that for some control point \( cp \) and machine reference \( z \), \( SC_i \xrightarrow{cp, z} SC'_i \) and \( SC_j \xrightarrow{cp, z} SC'_j \), \( z \notin \rho \).

From the inductive hypothesis we have that,

\[ \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \]
\[ \land \]
\[ (\text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \land z \cdot \text{assert2}(cp)) \Rightarrow \text{DIV}_{(\mu \cup \{z\}) \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]

which implies by Lemma 6.3.4,

\[ \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \]
\[ \land \]
\[ (\text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \land z \cdot \text{assert2}(cp)) \Rightarrow \]
\[ (\text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \land \text{DIV}_{(\mu \cup \{z\}) \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]

which implies

\[ \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \]
\[ \land \]
\[ \text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]
\[ \land \]
\[ z \cdot \text{assert2}(cp) \Rightarrow \text{DIV}_{(\mu \cup \{z\}) \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]

which implies

\[ \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_i)}(SC'_i, \text{assert2}) \]
\[ \land \]
\[ \text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \Rightarrow \text{DIV}_{\mu \cap \text{mref}(SC'_j)}(SC'_j, \text{assert2}) \]

Thus we have the case.
6.3. Some useful results about DIV

Case 4.7.2:
Suppose by COMM-R we conclude that for some control point \( cp \) and machine reference \( x \), \( SC_i \xrightarrow{cp,x} SC'_i \) and \( SC_j \xrightarrow{cp,x} SC'_j \).

Case then follows similar to 4.7.1

Case 4.7.3:
Suppose by COMM-L we conclude that for some standard names \( a \) and \( w \), \( SC_i \xrightarrow{a,w} SC'_i \) and \( SC_j \xrightarrow{a,w} SC'_j \).

From the inductive hypothesis we have that,

\[
\begin{align*}
DIV(SC_i, assert_2) &\Rightarrow DIV(SC'_i, assert_2) \\
DIV(SC_j, assert_2) &\Rightarrow DIV(SC'_j, assert_2)
\end{align*}
\]

Thus we have the case.

Case 4.7.4:
Suppose by COMM-R we conclude that for some standard names \( a \) and \( w \), \( SC_i \xrightarrow{a,w} SC'_i \) and \( SC_j \xrightarrow{a,w} SC'_j \).

Case then follows similar to 4.7.3.

Case 4.7.5:
Suppose by CLOSE-L we conclude that for some standard names \( a \) and \( v \), \( SC_i \xrightarrow{a,(v)} SC'_i \) and \( SC_j \xrightarrow{a,(v)} SC'_j \).

From the inductive hypothesis we have that,

\[
\begin{align*}
DIV(SC_i, assert_2) &\Rightarrow DIV(SC'_i, assert_2) \\
DIV(SC_j, assert_2) &\Rightarrow DIV(SC'_j, assert_2)
\end{align*}
\]

Hence case follows similar to 4.7.3

Case 4.7.6:
Suppose by CLOSE-R we conclude that for some standard names \( a \) and \( v \), \( SC_i \xrightarrow{a,w} SC'_i \) and \( SC_j \xrightarrow{a,(v)} SC'_j \).

Case then follows similar to 4.7.3.
6.4 Machine consistent mediators with infinite replication

This section extends the notions discussed so far in this chapter to mediators containing the bang operator. We construct such mediators using Lemma 5.2.3. As a result, note that mediators such as $!D$ are always machine closed ($\text{mref}(D) = \emptyset$ so $\text{mref}(!D) = \emptyset$).

First we extend Definition 6.1.5 of the $\text{DIV}$ function to act over agents with bang. Since such mediators are always machine closed in this thesis, the environment variable $\rho$ will always equal $\emptyset$ whenever we apply $\text{DIV}$. This is reflected in the definition.

**Definition 6.4.1.** Given a tagged mediator $!D$ and a function assert closed with respect to $!D$,

$$\text{DIV}_{\emptyset}(!D, \text{assert}) = \text{DIV}_{\emptyset}(D, \text{assert})$$

We now proceed the extend some of the relevant results from the previous section.

**Lemma 6.4.1.** For any mediator $!D$, $\rho \subseteq \text{mref}(!D)$ and some function assert closed with respect to $!D$,

1. $\text{DIV}_{\rho}(!D, \text{assert}) \Rightarrow \text{DIV}_{\delta}(!D, \text{assert})$ for any $\delta \subseteq \rho$.

2. $\bigwedge_{z \in \rho} \text{DIV}_{\{z\}}(!D, \text{assert}) \Leftrightarrow \text{DIV}_{\rho}(!D, \text{assert})$

**Proof.** Observe that any $\rho = \emptyset$ so any $\delta = \emptyset$.

Thus in the first part of the lemma we get,

$$\text{DIV}_{\emptyset}(!D, \text{assert}) \Rightarrow \text{DIV}_{\emptyset}(!D, \text{assert})$$

which is true.

In the second part of the lemma we get,

$$\bigwedge_{z \in \emptyset} \text{DIV}_{\{z\}}(!D, \text{assert}) \Leftrightarrow \text{DIV}_{\emptyset}(!D, \text{assert})$$

which is the same as,

$$\text{DIV}_{\emptyset}(!D, \text{assert}) \Leftrightarrow \text{DIV}_{\emptyset}(!D, \text{assert})$$

which is true. $\square$

The following Lemma 6.4.2 extends the inductive argument of Lemma 6.3.4 to infinite mediators.

**Lemma 6.4.2.** Given a tagged mediator $D$ and function assert closed with respect to $D$, and $\rho \subseteq \text{mref}(D)$.

Suppose $D \xrightarrow{cp} D'$ for some control point $cp$ and machine reference $z \notin \text{mref}(D)$ then,

$$\text{DIV}_{\rho}(D, \text{assert}) \Rightarrow \text{DIV}_{\rho \cup \text{mref}(D')}(D', \text{assert})$$
Proof. This is an inductive argument that carries on from Lemma 6.3.4.

Assume that for any mediator $D_2$, function assert closed with respect to $D_2$ and $\rho \subseteq mref(D_2)$, if $D_2 \overset{cp}{\rightarrow} D'_2$ for some control point $cp$ and machine reference $z$ then,

$$DIV_{\rho}(D_2, \text{assert}) \Rightarrow DIV_{\rho \setminus mref(D'_2)}(D'_2, \text{assert})$$

Suppose, $D_2$ is a machine closed mediator and $D \overset{D}{=} !D_2$. Then by $REP\text{-}ACT$ we conclude that conclude that $D \overset{\rho}{\rightarrow} D'$ where $D' \overset{\rho}{=} D'_2 \parallel D_2$.

Note that $\rho = \emptyset$ and by Definition 6.1.5,

$$DIV_{\emptyset \setminus mref(D'_2)}(D'_2, \text{assert}) = DIV_{\emptyset \setminus mref(D'_2)}(D'_2, \text{assert}) \land DIV_{\emptyset \setminus mref(1D_2)}(1D_2, \text{assert})$$

Thus we need to show that,

$$DIV_{\emptyset}(D_2, \text{assert}) \Rightarrow (DIV_{\emptyset \setminus mref(D'_2)}(D'_2, \text{assert}) \land DIV_{\emptyset \setminus mref(1D_2)}(1D_2, \text{assert}))$$

which is the same as (due to $D \overset{\emptyset}{=} !D_2$),

$$DIV_{\emptyset}(1D_2, \text{assert}) \Rightarrow DIV_{\emptyset \setminus mref(D'_2)}(D'_2, \text{assert})$$

By Definition 6.4.1, $DIV_{\emptyset}(1D_2, \text{assert}) = DIV_{\emptyset}(D_2, \text{assert})$ so,

$$DIV_{\emptyset}(D_2, \text{assert}) \Rightarrow DIV_{\emptyset \setminus mref(D'_2)}(D'_2, \text{assert})$$

which is true from the inductive hypothesis. □

The following Theorem 6.4.1 extends the inductive argument of Therorem 6.3.1 to mediators with bang.

**Theorem 6.4.1.** Given a tagged mediator $D$ and function assert closed with respect to $D$, and $\rho \subseteq mref(D)$.

Suppose $D \overset{\alpha}{\rightarrow} D'$ then,

1. if $\alpha = cp z$ for some control point $cp$ and machine reference $z \notin mref(D)$ then,

$$DIV_{\rho}(\beta, \text{assert}) \land z_{-\text{assert}(cp)}) \Rightarrow DIV_{(\rho \cup \{z\}) \setminus mref(D')}(D', \text{assert})$$

2. if $\alpha = cp z$ for some control point $cp$ and machine reference $z \in \rho$ then,

$$DIV_{\rho}(D, \text{assert}) \Rightarrow (DIV_{\rho \setminus mref(D')}(D', \text{assert}) \land z_{-\text{assert}(cp)})$$
3. if $\alpha = \bar{l}$ for some machine reference $z \in \rho$ and operation label $l$ then,

$$DIV_\rho(D, assert) \Rightarrow$$

$$\text{trm}(z_{-op(l)}) \land$$

$$\exists z_{-\text{VARIABLES}'}, (\text{pr}d_{z_{-\text{VARIABLES}'}}(z_{-op(l)}) \Rightarrow$$

$$\text{wp}(z_{-\text{VARIABLES}'} := z_{-\text{VARIABLES}'}, DIV_{\rho \cap \text{mref}(D')} (D', assert)))$$

where $z_{-\text{VARIABLES}'}$ denotes a list of variable names each prefixed with $z_-$ and primed.

4. in all other cases,

$$DIV_\rho(D, assert) \Rightarrow DIV_{\rho \cap \text{mref}(D')} (D', assert)$$

Proof. Case 1:
Assume that for any mediator $D_2$, function $assert$ closed with respect to $D_2$ and $\rho \subseteq \text{mref}(D_2)$, if $D_2 \leftrightarrow z D_2$ for some control point $cp$ and machine reference $z$ then,

$$DIV_\rho(D_2, assert \land z_{-assert(cp)}) \Rightarrow DIV_{(\rho \cup \{z\}) \cap \text{mref}(D_2')} (D_2', assert)$$

Suppose, $D_2$ is a machine closed mediator and $D \equiv !D_2$. Then by $\text{REP-ACT}$ we conclude that conclude that $D \xrightarrow{cp} D'$ where $D' \equiv D_2' \parallel !D_2$. We need to show that,

$$(DIV_\rho(D, assert) \land z_{-assert(cp)}) \Rightarrow DIV_{(\rho \cup \{z\}) \cap \text{mref}(D_2')} (D_2', assert)$$

Note that $\rho = \emptyset$ and by Definition 6.1.5,

$$DIV_{\{z\} \cap \text{mref}(D')}(D_2', assert) = DIV_{\{z\} \cap \text{mref}(D_2')} (D_2', assert) \land DIV_{\emptyset} (\emptyset, assert)$$

Thus we need to show that,

$$(DIV_\emptyset(D, assert) \land z_{-assert(cp)}) \Rightarrow (DIV_{\{z\} \cap \text{mref}(D_2')} (D_2', assert) \land DIV_{\emptyset} (\emptyset, assert))$$

which is the same as (due to $D \equiv !D_2$),

$$(DIV_\emptyset (\emptyset, assert) \land z_{-assert(cp)}) \Rightarrow DIV_{\{z\} \cap \text{mref}(D_2')} (D_2', assert)$$

By Definition 6.4.1, $DIV_\emptyset (\emptyset, assert) = DIV_\emptyset (D_2, assert)$ so,

$$(DIV_\emptyset (D_2, assert) \land z_{-assert(cp)}) \Rightarrow DIV_{\emptyset \cap \text{mref}(D_2')} (D_2', assert)$$

which is true from the inductive hypothesis.

Case 2:
Since we are only interested when $D \equiv !D_2$ for some machine closed mediator $D_2$, it is not the case that $D_2 \xrightarrow{cp} D_2'$ for any control point $cp$. Hence, the case is vacuously true.
Case 3:
Since we are only interested when $D = \mathcal{D}_2$ for some machine closed mediator $D_2$, it is not the case that $D_2 \Rightarrow^{\mathcal{D}} D_2'$ for any machine reference $z$. Hence, the case is vacuously true.

Case 4:
In any case of $\alpha$ not covered by cases 1 to 3.

Assume that for any mediator $D_2$, function assert closed with respect to $D_2$ and $\rho \subseteq mref(D_2)$, if $D_2 \Rightarrow^{\alpha} D_2'$ for some control point $cp$ and machine reference $z$ then,

$$DIV_{\rho}(D_2, \text{assert}) \Rightarrow DIV_{\rho \cap mref(D_2')}(D_2', \text{assert})$$

Suppose, $D_2$ is a machine closed mediator and $D = \mathcal{D}_2$. Then there are three possibilities. Either, the transition was inferred using $REP$-ACT or using $REP$-COMM or $REP$-CLOSE.

Case 4.1:
The transition was deduced using $REP$-ACT so $D' = D_2' \mathcal{D}_2$.

Note that $\rho = \emptyset$ and by Definition 6.1.5,

$$DIV_{\emptyset}(D', \text{assert}) = DIV_{\emptyset}(D_2', \text{assert}) \wedge DIV_{\emptyset}(\mathcal{D}_2, \text{assert})$$

Thus we need to show that,

$$DIV_{\emptyset}(D, \text{assert}) \Rightarrow (DIV_{\emptyset}(D_2', \text{assert}) \wedge DIV_{\emptyset}(\mathcal{D}_2, \text{assert}))$$

which is the same as (due to $D = \mathcal{D}_2$),

$$DIV_{\emptyset}(\mathcal{D}_2, \text{assert}) \Rightarrow DIV_{\emptyset}(D_2', \text{assert})$$

By Definition 6.4.1, $DIV_{\emptyset}(\mathcal{D}_2, \text{assert}) = DIV_{\emptyset}(D_2, \text{assert})$ so,

$$DIV_{\emptyset}(D_2, \text{assert}) \Rightarrow DIV_{\emptyset}(D_2', \text{assert})$$

which is true from the inductive hypothesis.

Case 4.2:
The transition is a $\tau$ and was deduced using $REP$-COMM so $D' = D_2' \mathcal{D}_2'' \mathcal{D}_2$. Assume that the inductive hypothesis extends over agent $D_2''$ and for some $\alpha_1$ and $\alpha_2$, $D_2 \Rightarrow^{\alpha_1} D_2'$ and $D_2 \Rightarrow^{\alpha_2} D_2''$ such that $D \Rightarrow^{\tau} D'$.

Note that $\rho = \emptyset$ and by Definition 6.1.5,

$$DIV_{\emptyset}(D', \text{assert}) = DIV_{\emptyset}(D_2', \text{assert}) \wedge DIV_{\emptyset}(D_2'', \text{assert}) \wedge DIV_{\emptyset}(\mathcal{D}_2, \text{assert})$$
Then by similar argument as in case 4.1 we conclude that,

\[ \text{DIV}_\emptyset(D_2, \text{assert}) \rightarrow (\text{DIV}_\emptyset(D_2', \text{assert}) \land \text{DIV}_\emptyset(D_2'', \text{assert})) \]

which is the same as,

\[ (\text{DIV}_\emptyset(D_2, \text{assert}) \rightarrow (\text{DIV}_\emptyset(D_2', \text{assert})) \land (\text{DIV}_\emptyset(D_2, \text{assert}) \rightarrow \text{DIV}_\emptyset(D_2'', \text{assert})) \]

which is true from the inductive hypothesis.

Case 4.3:  

The transition is a \( \tau \) and was deduced using \texttt{REP-CLOSE} so \( D' \sim (\nu a)(D_2' | D_2'') \parallel D_2 \) for some \( a \in SN \). Assume that the inductive hypothesis extends over agent \( D_2'' \) and for some \( \alpha_1 \) and \( \alpha_2 \), \( D_2 \xrightarrow{\alpha_1} D_2' \) and \( D_2 \xrightarrow{\alpha_2} D_2'' \) such that \( D \xrightarrow{\tau} D' \).

Note that \( \rho = \emptyset \) and by Definition 6.1.5,

\[ \text{DIV}_\emptyset(D', \text{assert}) = \text{DIV}_\emptyset(D_2', \text{assert}) \land \text{DIV}_\emptyset(D_2'', \text{assert}) \land \text{DIV}_\emptyset(!D_2, \text{assert}) \]

Hence this case is similar to case 4.2. \( \square \)

The following corollary is a special case of Theorem 6.4.1 when \( \rho = \emptyset \).

**Corollary 1.** Suppose we have that for some \( D \), \( \text{DIV}_\emptyset(D, \text{assert}) = \text{true} \) and \( D \xrightarrow{\alpha} D' \) for any \( \alpha \) then,

\[ \text{DIV}_\emptyset(D', \text{assert}) = \text{true} \]

**Proof.** By case analysis on \( \alpha \).

Suppose \( \alpha = cp \ z \) for some control point \( cp \) and machine reference \( z \).

Then by Theorem 6.3.1 we have that (where \( z \notin \rho \)),

\[ (\text{DIV}_\emptyset(D, \text{assert}) \land z_{-\text{assert}}(cp)) \Rightarrow \text{DIV}_{(\emptyset \cup \{z\}) \cap \text{mref}(D')}(D', \text{assert}) \]

\[ (\text{true} \land z_{-\text{assert}}(cp)) \Rightarrow \text{DIV}_{(\emptyset \cup \{z\}) \cap \text{mref}(D')}(D', \text{assert}) \]

Then by Lemma 6.4.1 we can split the \( \text{DIV} \) in to two conjunctions so,

\( \text{true} \Rightarrow \text{DIV}_{(\emptyset \cap \text{mref}(D'))}(D', \text{assert}) \land (z_{-\text{assert}}(cp) \Rightarrow \text{DIV}_{(\{z\} \cap \text{mref}(D'))}(D', \text{assert})) \)

Thus we have that \( \text{true} \Rightarrow \text{DIV}_{(\emptyset \cap \text{mref}(D'))}(D', \text{assert}) \) hence,

\[ \text{DIV}_\emptyset(D', \text{assert}) = \text{true} \]
6.5 Consistency with semantic definition of machine divergence freedom

All other cases of $\alpha$ are trivially true by Theorem 6.4.1 as $\rho = \emptyset$,

$$DIV_{\emptyset}(D, \text{assert}) \Rightarrow DIV_{\emptyset}^{\text{mref}(D')}(D', \text{assert})$$

Thus we have that true $\Rightarrow$ $DIV_{\emptyset}(D', \text{assert})$ hence,

$$DIV_{\emptyset}(D', \text{assert}) = \text{true} \quad \Box$$

6.5 Consistency with semantic definition of machine divergence freedom

In this section we show that the DIV predicate defined in Section 6.1 and Section 6.4 is consistent with the definition of machine divergence in control systems from Definition 5.4.1 on page 5.4.1. The property we want to show is that a control system with a machine closed mediator $D$ for which we have obtained $DIV(D, \text{assert}_{-\text{init}}) = \text{true}$ is machine divergence-free. This result is stated in Theorem 6.5.4. Before we attempt to prove Theorem 6.5.4 we need several supporting definitions and intermediate results.

We, note that there is a mismatch between the mediator selecting an operation and the subsequent machine state update. In the definition of DIV in Section 6.1 an operation is considered as an atomic action whereas in the $\pi$-style of $LTS_M$ introduced in Section 4.3 an operation is split in to two transitions: the operation selection and the state update. In the results that follow we consider traces of a control system between points where all machines have completed the state update routines and this enables us to harmonise these two views.

**Definition 6.5.1.** We say a control system $\text{CSYSTEM}_D$ is in a ready state if for every state $x_i$ for $1 \leq i \leq n$ such that,

$$\text{CSYSTEM}_D \equiv_{\pi} (\nu \quad \bar{v})(D \mid [x_1](z_1) \mid \ldots \mid [x_n](z_n) \mid \text{MGENERATOR})$$

we have that $x_i$ is equivalent to the maplet $(\text{READY, val})$ for some machine state val.

Furthermore, we sometimes need to convert a machine state back to a predicate.

**Definition 6.5.2.** Given a machine state val,

1. $R^\text{val}$ denotes the least predicate satisfied by val.

Recall also from Section 6.1 that for a given machine reference $z$, $z_{-}(R)$ denotes the predicate where every free name is prefixed by $z_{-}$. Thus, $z_{-}(R^\text{val})$ denotes the least predicate that is satisfied by val and where each name is prefixed by $z_{-}$.

**Example 6.5.1.**

$$z_{-}(R_e^{(mn_{\frac{1}{1}})}) \equiv (z_{-}nn = 1)$$
The first result we obtain, considers the behaviour of the control system with respect to a single machine which is being initialised. The hypothesis assumes that the $DIV_0$ predicate of $D_0$ is true. This is to ensure that the control points have been tagged consistently. The theorem shows that after the initialisation is completed the final state of the machine is not equal to $\perp$ and implies the $DIV$ predicate of the final mediator which is constrained to the reference of the initialised machine. The theorem is true for all traces of a control system however we will check only one case. The result is combined into an inductive argument in Theorem 6.5.3.

**Theorem 6.5.1.** Given a control system $CSYSTEM_D$ where,

$$CSYSTEM_D = \pi(\nu \bar{v})(D_0 | [(\text{READY}, val_1)]_{M_1}(z_1) | \ldots | [(\text{READY}, val_t)]_{M_t}(z_t) | MGENERATOR)$$

and $t \in \mathbb{N}$ with $z_1, \ldots, z_t$ in the list $\bar{v}$ and

$$DIV_0(D_0, \text{assert}_{\text{-}init}) = \text{true}$$

and there is a control system $CSYSTEM_{D_k}$ where,

$$CSYSTEM_{D_k} = \pi(\nu \bar{y})(D_k | [(\text{READY}, val'_1)]_{M_1}(z_1) | \ldots | [(\text{READY}, val'_m)]_{M_m}(z_m) | MGENERATOR)$$

with $z_1, \ldots, z_m$ in the list $\bar{y}$ and $t < m$ and for some $z_0$ where $t < o \leq m$, $val'_o \in INIT_M$ and for any trace $tr$,

$$CSYSTEM_D \xrightarrow{tr} CSYSTEM_{D_k}$$

Then it is the case that $z_o \overset{R_{val}}{\not=} \perp$ and,

$$z_0 \overset{R_{val}}{\Rightarrow} DIV_{\{z_o\} \cap \text{mref}(D_k)}(D_k, \text{assert}_{\text{-}init})$$

**Proof.** Consider an arbitrary trace of $CSYSTEM_D$.

The trace by which $CSYSTEM_D$ can progress to $CSYSTEM_{D_k}$ in ready state is of the following form,

$$\langle \alpha_1, \ldots, \alpha_n, \tau_{\text{create}M} z_0, \alpha_{n+2}, \ldots, \alpha_s, \tau_{\text{init}M}, \alpha_{s+2}, \ldots, \alpha_k \rangle$$

where we have labelled the two $\tau$ responsible for creating the machine at $z_o$ and initialising it for clarity.

Here we only consider the case where any action from $\alpha_{s+2}, \ldots, \alpha_k$ does not involve communication with the machine at $z_o$. These cases are covered in Theorem 6.5.2 which follows.
In performing a transition derivation $CSYSTEM_{D_i} \xrightarrow{\alpha_{i+1}} CSYSTEM_{D_{i+1}}$ in the range $0 \leq i < n$, one of the following is true: $D_i \equiv D_{i+1}$ or $D_i \stackrel{\text{createM } z}{\rightarrow} D_{i+1}$ for some machine reference $z \neq z_0$ or $D_i \xrightarrow{\beta} D_{i+1}$ for some other action $\beta$ that is not equivalent to $\text{createM } z$.

In the first and the last case from Theorem 6.4.1 we can conclude that,

$$DIV_\rho(D_i, \text{assert_init}) \Rightarrow DIV_{\rho \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init})$$

for some $\rho \subseteq \text{mref}(D_i)$.

If $D_i \stackrel{\text{createM } z}{\rightarrow} D_{i+1}$ for some machine reference $z \neq z_0$ is true then we can conclude that,

$$(DIV_\rho(D_i, \text{assert_init}) \land z_{-\text{assert_init}(\text{createM})}) \Rightarrow
DIV_{(\rho \cup \{z\}) \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init})$$

for some $\rho \subseteq \text{mref}(D_i)$.

However, in the latter case we can distribute the predicate using Lemma 6.3.3 as follows,

$$((DIV_\rho(D_i, \text{assert_init}) \land z_{-\text{assert_init}(\text{createM})}) \Rightarrow
DIV_{(\rho \cup \{z\}) \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init}))$$

$$\Rightarrow (DIV_\rho(D_i, \text{assert_init}) \Rightarrow DIV_{\rho \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init})) \land
z_{-\text{assert_init}(\text{createM})} \Rightarrow DIV_{\{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init})$$

for some $\rho \subseteq \text{mref}(D_i)$.

In each case where this occurs we can throw away the conjunct,

$$z_{-\text{assert_init}(\text{createM})} \Rightarrow DIV_{\{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert_init})$$

as it involves a machine which we are not interested in.

We know that $DIV_\emptyset(D_n, \text{assert_init}) = \text{true}$ thus from the above we can conclude that $DIV_\emptyset(D_n, \text{assert_init}) = \text{true}$.

At $CSYSTEM_{D_n} \xrightarrow{\text{createM } z_0} CSYSTEM_{D_{n+1}}$ we conclude that $D_n \stackrel{\text{createM } z_0}{\rightarrow} D_{n+1}$. Thus by Theorem 6.4.1 we can conclude that,

$$DIV_\emptyset(D_n, \text{assert_init}) \land z_{0-\text{assert_init}(\text{createM})}) \Rightarrow
DIV_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert_init})$$

$$= (\text{true} \land z_{0-\text{assert_init}(\text{createM})}) \Rightarrow DIV_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert_init})$$

$$= z_{0-\text{assert_init}(\text{createM})} \Rightarrow DIV_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert_init})$$
For all remaining actions of $D_{n+1}$,

\[
\{\alpha_{n+2}, \ldots, \alpha_s, \tau_{\text{init}M}, \alpha_{s+2}, \ldots, \alpha_k\}
\]

by similar argument as in the range $0 \leq i < n$ above we can conclude that,

\[
DIV_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert\_init}) \Rightarrow DIV_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init})
\]

Thus at $k$ either $\{z_0\} \cap \text{mref}(D_k) = \emptyset$ or $\{z_0\} \cap \text{mref}(D_k) = \{z_0\}$.

If $\{z_0\} \cap \text{mref}(D_k) = \emptyset$ then $DIV_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init}) = \text{true}$ by Corollary 1. Thus for any $z_0 - R^{val'}$,

\[
z_0 - R^{val'} \Rightarrow z_0 - \text{assert\_init}(\text{create}M) \Rightarrow \text{true}
\]

If $\{z_0\} \cap \text{mref}(D_k) = \{z_0\}$ we need to show that there is a $val'_0 \in INIT_M$ such that,

\[
z_0 - R^{val'} \Rightarrow z_0 - \text{assert\_init}(\text{create}M)
\]

For both cases above we know that $CSYSTEM_{D_{n+1}} \xrightarrow{\tau_{\text{init}}M} CSYSTEM_{D_{n+1}}$ which corresponds to the machine at $z_0$ initialising.

\[
[(\text{BEGIN})]_M(z_0) \xrightarrow{\text{init}M} [(\text{READY}, \text{val}'_0)]_M(z_0)
\]

where $val'_0 \in INIT_M$.

By Definition 6.1.4 we have that,

\[
\text{assert\_init}(\text{create}M) = \text{wp}(\text{VARIABLES}'_M := \text{VARIABLES}_M, \text{pr} \text{d} \text{VARIABLES}_M(T_M))
\]

where $\text{VARIABLES}_M$ is the list of machine variables of $M$ and $\text{VARIABLES}'_M$ is an identical list where each variable is primed. This definition is analogous to Definition of $INIT_M$ (Definition 4.1.3). Thus the valuations that satisfy $\text{assert\_init}(\text{create}M)$ are exactly those that are in the set $INIT_M$. Thus $val'_0$ satisfies $\text{assert\_init}(\text{create}M)$. Thus if $R^{val'}$ is the least predicate that $val'_0$ satisfies, then $R^{val'} \Rightarrow \text{assert\_init}(\text{create}M)$. Thus it is also the case that $z_0 - (R^{val'}) \Rightarrow \text{assert\_init}(\text{create}M)$ is true which is equivalent to,

\[
z_0 - R^{val'} \Rightarrow z_0 - \text{assert\_init}(\text{create}M)
\]

Thus the theorem is true. \qed

The following theorem illustrates that, if the state of an active machine satisfies the $DIV$ predicate of the mediator $D_0$ which is constrained to the reference of that machine then the control system can progress by executing operations on the machine to a new control system where the new state of the machine satisfies the $DIV$ predicate of the new mediator constrained for that machine. As in the previous theorem we consider only one case even though the theorem works for all traces.
Theorem 6.5.2. Given a control system $\text{CSYSTEM}_D$ where,

$$\text{CSYSTEM}_{D_0} \equiv \pi(\nu \bar{v})(D_0 \ | \ [(\text{READY}, \ \text{val}_1)]_{M_1}(z_1) \ | \ \ldots \ | \ [(\text{READY}, \ \text{val}_t)]_{M_t}(z_t) \ | \ \text{MGENERATOR})$$

and $t \in \mathbb{N}$ with $z_1, \ldots, z_t$ in the list $\bar{v}$ and for some $z_0$ where $1 \leq o \leq t$,

$$z_0 - R^{v_0} \Rightarrow \text{DIV}_{\{z_o\}}(D_0, \ \text{assert\_init})$$

and $\text{val}_o \neq \bot$ and there is a control system $\text{CSYSTEM}_{D_k}$ where,

$$\text{CSYSTEM}_{D_k} \equiv \pi(\nu \bar{y})(D_k \ | \ [(\text{READY}, \ \text{val}_1')]_{M_1}(z_1) \ | \ \ldots \ | \ [(\text{READY}, \ \text{val}_m')]_{M_m}(z_m) \ | \ \text{MGENERATOR})$$

with $z_1, \ldots, z_m$ in the list $\bar{y}$ and $t < m$ and for any trace $\text{tr}$,

$$\text{CSYSTEM}_{D_0} \xrightarrow{\text{tr}} \text{CSYSTEM}_{D_k}$$

Then it is the case that $z_0 - R^{v_0} \neq \bot$ and,

$$z_0 - R^{v_0} \Rightarrow \text{DIV}_{\{z_o\} \cap \text{ref}(D_k)}(D_k, \ \text{assert\_init})$$

Proof. Consider an arbitrary trace of $\text{CSYSTEM}_{D_0}$.

The trace by which $\text{CSYSTEM}_{D_0}$ can progress to $\text{CSYSTEM}_{D_k}$ in a ready state is of the following form,

$$\langle \alpha_1, \ldots, \alpha_n, \tau_{\bar{z}_a} l_1, \alpha_{n+2}, \ldots, \alpha_s, \tau_{\text{op}(l)}, \alpha_{s+2}, \ldots, \alpha_k \rangle$$

or

$$\langle \alpha_1, \ldots, \alpha_n, \tau_{\bar{z}_a} l_1, \alpha_{n+2}, \ldots, \alpha_s, \overline{\text{div}}, \alpha_{s+2}, \ldots, \alpha_k \rangle$$

where we have labelled the two $\tau$ responsible for selecting the operation $l$ on $z_o$ and performing the state update.

Here we only consider the case where any action from $\alpha_{s+2}, \ldots, \alpha_k$ does not involve communication with the machine at $z_o$. These cases are covered by the fact that the Theorem 6.5.2 can be applied reapplied to the final control system. We also consider only good traces of the system where div does not occur.

In performing a transition derivation $\text{CSYSTEM}_{D_i \xrightarrow{\alpha_{i+1}} \text{CSYSTEM}_{D_{i+1}}}$ in the range $0 \leq i < n$, one of the following is true; $D_i \equiv \pi D_{i+1}$ or $D_i \overset{\text{create}_M}{\xrightarrow{\beta}} D_{i+1}$ for some machine reference $z \neq z_o$ or $D_i \overset{\beta}{\xrightarrow{\gamma}} D_{i+1}$ for some other action $\beta$ that is not equivalent to $\text{create}_M z$. 
In the first and the last case from Theorem 6.4.1 we can conclude that,

\[ \text{DIV}_\rho(D_i, \text{assert}_\text{-init}) \Rightarrow \text{DIV}_{\rho \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init}) \]

for some \( \rho \subseteq \text{mref}(D_i) \).

If \( D_i \xrightarrow{\text{create}_M} z D_{i+1} \) for some machine reference \( z \neq z_0 \) is true then we can conclude that,

\[
(DIV_\rho(D_i, \text{assert}_\text{-init}) \land z_\text{-assert}_\text{-init}(\text{create}_M)) \Rightarrow \\
\text{DIV}_{\rho \cup \{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init})
\]

for some \( \rho \subseteq \text{mref}(D_i) \).

However, we can distribute the predicate using Lemma 6.3.3 as follows,

\[
((DIV_\rho(D_i, \text{assert}_\text{-init}) \land z_\text{-assert}_\text{-init}(\text{create}_M)) \Rightarrow \\
\text{DIV}_{\rho \cup \{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init})) \\
\Rightarrow \\
(DIV_\rho(D_i, \text{assert}_\text{-init}) \Rightarrow \text{DIV}_{\rho \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init})) \land \\
(z_\text{-assert}_\text{-init}(\text{create}_M) \Rightarrow \text{DIV}_{\{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init})
\]

for some \( \rho \subseteq \text{mref}(D_i) \).

In each case where this occurs we can throw away the conjunct,

\[
(z_\text{-assert}_\text{-init}(\text{create}_M) \Rightarrow \text{DIV}_{\{z\} \cap \text{mref}(D_{i+1})}(D_{i+1}, \text{assert}_\text{-init}))
\]

as it involves a machine in which we are not interested.

We know that,

\[
z_0 R_{\text{val}} \Rightarrow \text{DIV}_{\{z_0\}}(D_0, \text{assert}_\text{-init})
\]

thus from the above we can conclude that,

\[
z_0 R_{\text{val}}' \Rightarrow \text{DIV}_{\{z_0\} \cap \text{mref}(D_n)}(D_n, \text{assert}_\text{-init})
\]

At \( \text{CSYSTEM}_{D_n} \xrightarrow{\tau_{z_0}} \text{CSYSTEM}_{D_{n+1}} \) we conclude that \( D_n \xrightarrow{\tau_{z_0}} D_{n+1} \). Then there are two cases either \( \{z_0\} \cap \text{mref}(D_n) = \emptyset \) or \( \{z_0\} \cap \text{mref}(D_n) = \{z_0\} \).

If \( \{z_0\} \cap \text{mref}(D_n) = \emptyset \) then \( \text{DIV}_{\{z_0\} \cap \text{mref}(D_n)}(D_n, \text{assert}_\text{-init}) = \text{true} \). Thus, for any \( z_0 R_{\text{val}}' \),

\[
z_0 R_{\text{val}}' \Rightarrow \text{DIV}_{\{z_0\} \cap \text{mref}(D_n)}(D_n, \text{assert}_\text{-init})
\]
If \( \{z_0\} \cap \text{mref}(D_n) = \{z_0\} \) by Theorem 6.4.1 we can conclude that,

\[
\text{DIV}_{\{z_0\} \cap \text{mref}(D_n)}(D_n, \text{assert}\_\text{init}) \Rightarrow G
\]

where \( G = \text{trm}(z_0\_\text{op}(l)) \land \)

\[
\exists z_o\_\text{VARIABLES}' , \ (\text{pr}_d z_o\_\text{VARIABLES}(z_0\_\text{op}(l)) \Rightarrow \wp(z_o\_\text{VARIABLES} := z_o\_\text{VARIABLES}' , \text{DIV}_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert}))
\]

For all remaining actions,

\[
\{\alpha_{n+2}, \ldots, \alpha_s, \tau_{op}, \alpha_{s+2}, \ldots, \alpha_k\}
\]

by similar argument as in the range \(0 \leq i < n\) above we can conclude that,

\[
\text{DIV}_{\{z_0\} \cap \text{mref}(D_{n+1})}(D_{n+1}, \text{assert}) \Rightarrow \text{DIV}_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert}\_\text{init})
\]

At \( k \) either \( \{z\} \cap \text{mref}(D_k) = \emptyset \) or \( \{z\} \cap \text{mref}(D_k) = \{z\} \).

If \( \{z_0\} \cap \text{mref}(D_k) = \emptyset \) then \( \text{DIV}_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert}\_\text{init}) = \text{true} \). Thus, for any \( z_0\_\text{val}' \),

\[
z_0\_\text{val}' \Rightarrow \text{DIV}_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert}\_\text{init})
\]

If \( \{z_0\} \cap \text{mref}(D_k) = \{z_0\} \) we need to show that there is a \( \text{val}'_o \) such that,

\[
z_0\_\text{val}'_o \Rightarrow \text{DIV}_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert}\_\text{init})
\]

From monotonicity of \( \wp \) we can substitute in the predicate \( G \) above thus we need to show that,

\[
z_0\_\text{val} \Rightarrow \text{trm}(z_0\_\text{op}(l)) \land \)

\[
\exists z_o\_\text{VARIABLES}' , \ (\text{pr}_d z_o\_\text{VARIABLES}(z_0\_\text{op}(l)) \Rightarrow \wp(z_o\_\text{VARIABLES} := z_o\_\text{VARIABLES}' , \text{DIV}_{\{z_0\} \cap \text{mref}(D_k)}(D_k, \text{assert}\_\text{init}))
\]
Thus we have that,
\[
(z_0 - R^{val_0} \Rightarrow trm(z_0 - op(l))) \\
\land \\
(z_0 - R^{val_0} \Rightarrow \\
\exists z_0 - VARIABLES', (prd_{z_0 - VARIABLES}(z_0 - op(l)) \Rightarrow \\
wp(z_0 - VARIABLES := z_0 - VARIABLES', DIV_{z_0 \cap \text{mref}(D_k)}(D_k, \text{assert_init}))))
\]

(6.7)

Thus for some predicate \( z_0 - R^{\text{primed}}(val_{2_0}) \),
\[
(z_0 - R^{val_0} \land z_0 - R^{\text{primed}}(val_{2_0}) \Rightarrow \\
(prd_{z_0 - VARIABLES}(z_0 - op(l)) \Rightarrow \\
w{p}(z_0 - VARIABLES := z_0 - VARIABLES', DIV_{z_0 \cap \text{mref}(D_k)}(D_k, \text{assert_init}))))
\]

Thus to show that
\[
z_0 - R^{val_{2_0}} \Rightarrow DIV_{z_0 \cap \text{mref}(D_k)}(D_k, \text{assert_init})
\]

we need to show that,
\[
(z_0 - R^{val_0} \land z_0 - R^{\text{primed}}(val_{2_0}) \Rightarrow prd_{z_0 - VARIABLES}(z_0 - op(l))
\]

We have that \( CSYSTEM_{D_e} \xrightarrow{\tau_{op(l)}} CSYSTEM_{D_{e+1}} \) which corresponds to the machine at \( z_{-o} \) performing the state update,
\[
\llbracket(BODY_{op(l)}, val_0)\rrbracket_M\langle z_0 \rangle \xrightarrow{\text{init}_M} \llbracket(READY, val_0')\rrbracket_M\langle z_0 \rangle
\]

where by Definition 4.1.1, \([val_0, \text{primed}(val_0')]\) satisfies \( prd_{ VARIABLES}(op(l)) \).

Then it follows that,
\[
(z_0 - R^{val_0} \land z_0 - R^{\text{primed}}(val_{2_0}) \Rightarrow prd_{z_0 - VARIABLES}(z_0 - op(l))
\]

Thus we obtain that,
\[
z_0 - R^{val_{20}} \Rightarrow DIV_{z_0 \cap \text{mref}(D_k)}(D_k, \text{assert_init})
\]

Furthermore, it is not the case that \( val_0' = 1 \) because,
\[
\llbracket(BODY_{op(l)}, val_0)\rrbracket_M\langle z_0 \rangle \xrightarrow{\text{div}} \llbracket(READY, 1)\rrbracket_M\langle z_0 \rangle
\]
6.5. Consistency with semantic definition of machine divergence freedom

if by Definition 4.1.1, $val_0 = \bot$ or $val_0$ satisfies $abt(op(l))$.

We have that $val_0 \neq \bot$ from the hypothesis.

Also, above (Equation 6.7) we obtained that $(z_0 - Rval_0 \Rightarrow trn(z_0 - op(l)))$ must be true, which is equivalent to $val_0$ satisfies $trn(op(l))$ which is equivalent to $val_0$ satisfies $-abt(op(l))$.

Thus the theorem is true. □

The following theorem combines the results of Theorem 6.5.1 and Theorem 6.5.2 into a single inductive argument about the behaviour of a control system with $t$ active machines. The theorem relies on the fact that communication between the mediator and all active machines is interleaved.

**Theorem 6.5.3.** Given a control system $CSYSTEM_D$ where,

$$CSYSTEM_{D_0} = \exists (\nu \bar{v})(D_0 | [\{(READY, val_i)\}]_{M_1}(z_1) | \ldots | [\{(READY, val_i)\}]_{M_t}(z_t) | MGENERATOR)$$

and $t \in \mathbb{N}$ with $z_1, \ldots, z_t$ in the list $\bar{v}$,

$$\bigwedge_{1\leq o\leq t} z_o - Rval_0 \Rightarrow DIV(D_0, assert\_init)$$

and $1 \leq o \leq t$, $val_0 \neq \bot$ and there is a control system $CSYSTEM_{D_k}$ where,

$$CSYSTEM_{D_k} = \exists (\nu \bar{y})(D_k | [\{(READY, val'_i)\}]_{M_1}(z_1) | \ldots | [\{(READY, val'_m)\}]_{M_m}(z_m) | MGENERATOR)$$

with $z_1, \ldots, z_m$ in the list $\bar{y}$ and $t < m$ and for any trace $tr$,

$$CSYSTEM_{D_0} \xrightarrow{tr} CSYSTEM_{D_k}$$

Then it is the case that for $1 \leq o \leq t$, $val'_o \neq \bot$ and,

$$\bigwedge_{1\leq o\leq m} z_o - Rval'_o \Rightarrow DIV(D_k, assert\_init)$$

**Proof.** Consider an arbitrary trace $tr$ of $CSYSTEM_{D_0}$.

By finite number of applications of Theorem 6.5.1 and Theorem 6.5.2 we can conclude that $1 \leq o \leq t$, $val'_o \neq \bot$ and,

$$z_o - Rval'_o \Rightarrow DIV(z_o) \cap ref(D_k)(D_k, assert\_init)$$
which is the same as
\[ \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow \bigwedge_{1 \leq o \leq m} \text{DIV}_{\{z_{o}\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init}) \]
which is the same as
\[ \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow \bigwedge_{z \in \{z_1, \ldots, z_m\}} \text{DIV}_{\{z\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init}) \]
\[ \Rightarrow \text{ by Lemma 6.3.3} \]
\[ \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow (\bigwedge_{z \in \{z_1, \ldots, z_m\} \cap \text{mref}(D_k)} \text{DIV}_{\{z\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init}) \land \text{DIV}_0(D_k, \text{assert\_init})) \]
\[ = \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow \bigwedge_{z \in \text{mref}(D_k)} \text{DIV}_{\{z\} \cap \text{mref}(D_k)}(D_k, \text{assert\_init}) \land \text{DIV}_0(D_k, \text{assert\_init}) \]
\[ \Rightarrow \text{ by Lemma 6.3.3} \]
\[ \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow \text{DIV}(D_k, \text{assert\_init}) \land \text{DIV}_0(D_k, \text{assert\_init}) \]
\[ \Rightarrow \bigwedge_{1 \leq o \leq m} z_{o - R \text{val}^e_{o}} \Rightarrow \text{DIV}(D_k, \text{assert\_init}) \]

\[ \square \]

**Theorem 6.5.4.** Given a control system $CSYSTEM_D$ in an initial state if
\[ \text{DIV}(D, \text{assert\_init}) = \text{true} \]
for some assert\_init then $CSYSTEM_D$ is machine divergence-free.

**Proof.** If $CSYSTEM_D$ is in an initial state then there are no active $B$ machines hence,
\[ CSYSTEM_D \downarrow \overline{\text{div}} = \text{false} \]

Suppose,
\[ CSYSTEM_D \overset{\text{tr}}{\rightarrow} CSYSTEM_{D_k} \]
for any $CSYSTEM_{D_k}$ and tr.

Then from $\text{DIV}(D, \text{assert\_init}) = \text{true}$ and Theorem 6.5.3, we have that none of the active machines have machine state equal $\bot$. Thus,
\[ CSYSTEM_{D_k} \downarrow \overline{\text{div}} = \text{false} \]

Thus $CSYSTEM_D$ is machine divergence-free. \[ \square \]
This result justifies our formal framework in which we can specify a system of π-calculus agents and machines and show that the agents do not cause divergence in those machines. It mirrors the results and ideas in [62, 55, 56] but significantly extends them to include mobility and dynamic instantiation.
Chapter 7

Resource Allocation System

7.1 System overview

To illustrate the framework presented in this thesis, we consider an example of allocating resources within a network. This example is published in [28]. The system is called a Resource Allocation Service (RAS). It offers clients the opportunity to request increases and decreases in the quantity of resource currently allocated. Resources are passed around a network, to service areas of high demand. Servers in the network autonomously decide how best to respond to a request for more resources, either by creating a fresh resource, by allocating one from a local pool of free resources, or by passing a request to another server. In the example we focus on the range of possibilities for action that is available to the servers and we abstract from this decision making process.

We begin by providing a full specification of the RAS system and in Section 7.3 we verify that the system is machine divergence-free.

7.2 Specification

We model resources in a network as B 'Node' machines which in an implementation of a real system can be components that are allocated to particular tasks such as querying a database. Figure 7.1 gives the description of the machine we consider which we call 'Node'. The machine has three possible states, and operations for switching between them. It is initially in state Fresh, and once it is activated it alternates between Busy and Idle, by means of the operations GetBusy and GetFree. The preconditions of these operations introduce the requirement that GetBusy should be executed when the Node is not busy and GetFree when the Node is busy. It will be important to ensure that the RAS does not violate these requirements when activating nodes.

The dynamic part of the system consists of a collection of SERVER mediators. These mediators provide an external interface inc and dec for clients of the system to request
MACHINE Node
SETS STATUS = {Fresh, Busy, Free}
VARIABLES status
INITIALISATION
status ∈ STATUS
OPERATIONS
   GetBusy ≡ PRE status = Fresh ∨ status = Free
       THEN status := Busy
       END;
   GetFree ≡ PRE status = Busy
       THEN status := Free
       END
END

Figure 7.1: The B description of a Node

an increase of a resource, or a decrease of a resource, at a particular area, respectively. They also activate, manage and transfer resources in order to meet areas of high demand. The architecture of SERVER mediators is presented in Figure 7.2 and their π-calculus description is given in Figure 7.3.

In the π-specification of a server components we use $V_{Node}$ to denote the variant type $[\text{GetFree\_unit}, \text{GetBusy\_unit}]$ that corresponds to the signature of a Node machine. Thus, a particular Node machine with machine reference $z$ will have its operations called through occurrences of $z \text{GetBusy}\_\langle * \rangle$ and $z \text{GetFree}\_\langle * \rangle$. This notation was first introduced in Section 3.4 on page 37.

The mediator $SERVER_i$ is able to handle requests for a resource from an external client, through the particular channel $inc_i$. Requests can also arrive along $req$ channels from other servers; the channel $req\_i,i$ is used to pass a request from $SERVER_j$ to $SERVER_i$. When a request has been received (from either of these sources), there are three ways of obtaining the resource required. The first is through the creation of a new resource, provided by a machine generator on channel $create_i$; the second is by identifying a free resource currently in the local pool of available resources, and this is done through the server's internal channel $p_i$; and the third is by passing the request to another server $k$, along channel $req\_i,k$, and then receiving the response along channel $c_{k,i}$. These three
7.2. Specification

\[
SERVER_i = (\nu p_i : \| o V_{Node} )
\]

\[ (! (inc_i . (create_i(z) . WORK_i(z))
+ p_i(z) . WORK_i(z))
+ \Sigma_{k \in S_i \mid req_{i,k}} . c_{k,i}(z) . WORK_i(z))
+ \Sigma_{j \in C_i \mid req_{j,i}} . (create_i(z) . SEND_i(j,z)
+ p_i(z) . SEND_i(j,z)
+ \Sigma_{k \in S_i \mid req_{i,k}} . c_{k,i}(z) . SEND_i(j,z))))
\]

\[
WORK_i(z) = \exists GetBusy_{-}(z) . \ exists . \exists GetFree_{-}(z) . p_i(z) . 0
\]

\[
SEND_i(j,z) = c_{i,j}(z) . 0
\]

Figure 7.3: Description of a server controller

Figure 7.4: Possible responses to \(inc_i\) (and also to \(req_{j,i}\)) from \(SERVER_i\)

possible reactions to \(inc_i\) (and also to \(req_{j,i}\)) are illustrated in Figure 7.4. Each of the channels \(create_i\), \(p_i\), and \(c_{k,i}\) are used to communicate machine references, and so they are all control points of the \(SERVER_i\) mediator. Observe that the description of \(SERVER_i\) is consistent with the requirements on a mediator: most importantly that when a machine reference is output on a control point then the subsequent description should not contain any free occurrence of the machine reference. This requirement on mediators was discussed in Section 5.1.1. In the case of \(SERVER_i\) we see that when machine reference \(z\) is output along control point \(p_i\) or \(c_{i,j}\), then the subsequent description on that thread of control is in fact 0, which indeed does not contain \(z\).

When \(SERVER_i\) has control of a Node through a link \(z\), it is able to activate it and shut it down by use of \(z\ GetBusy_{-}()\) and \(z\ GetFree_{-}()\). Requests to reduce resource usage along the \(dec_i\) channel result in the closing down of node activity, and the release of the node into the local pool of available resources along the channel \(p_i\).

The use of replication in the server description indicates that any number of \(inc_i\) or \(req_{j,i}\) requests can be handled. However, observe that \(dec_i\) is possible only when there
\[ MGEN_i = !(\nu z_i : V_{\text{Node}})(\text{create}_i(z_i), \text{initNode}_i.0 | \text{BEGIN}_i(\text{Node}(z_i)) \]

\[ RESRCER_i = (\nu create_i : V_{\text{Node}})(\text{SERVER}_i | MGEN_i) \]

\[ RAS = (\nu C : V_{\text{Node}}, R : \text{unit})(\text{RESRCER}_1 | \ldots | \text{RESRCER}_n) \]

where

\[ C = \bigcup_{i \in I} \{c_{i,j} \mid j \in C_i\} \]

\[ R = \bigcup_{i \in I} \{\text{req}_i,j \mid j \in S_i\} \]

Figure 7.5: The architecture of RAS

are active nodes, and it will be blocked otherwise.

In the specification we let \( C_i \) denote the set of servers which can make a resource request to \( \text{SERVER}_i \): it is those \( j \)'s for which a \( \text{req}_j,i \) will be allowed. Conversely, the set \( S_i \) denotes the servers from which \( \text{SERVER}_i \) can request a resource, and for consistency we require that \( j \in C_i \Leftrightarrow i \in S_j \) for any \( i \) and \( j \). If the set \( S_i \) is empty then it will not be possible for \( \text{SERVER}_i \) to pass the request on, and it will have to be serviced either by recycling a resource, or by creating a new one. Conversely, if \( C_i \) is empty then \( \text{SERVER}_i \) will not receive requests from any other servers.

The sets \( C_i \) or \( S_i \) will correspond to a network structure or hierarchy of resource allocators, which will vary according to the considerations of the RAS design. In our example we do not allow cycles in the graph of request links so that a request cannot be passed indefinitely round the loop of servers.

### 7.2.1 Putting the system together

The \( \text{SERVER}_i \) mediator is combined with a mechanism for generating Node machines. This process is called \( MGEN_i \) which is short for the process \( \text{MGENERATOR} \) (Definition 4.5.1), in Chapter 4. The description of \( MGEN_i \) is given in Figure 7.5. Essentially, it is used to generate and initialise ‘Node’ machines, raising a fresh machine reference \( z \) for that node, and passing that machine reference to \( \text{SERVER}_i \) along their joint channel \( \text{create}_i \). Figure 7.6 illustrates the combination of \( \text{SERVER}_i \) and a newly generated Node.

Observe that we include an \( MGEN_i \) process with every \( \text{SERVER}_i \) process. This is to illustrate that \( MGEN \) does not have to be a central component of the system as was discussed in previous chapters. In this case, each \( MGEN_i \) outputs a node reference along its own \( \text{create}_i \) channel however, had those channels been the same then it can be shown that \( MGEN \) and \( MGEN \mid MGEN \mid \ldots \) are equivalent processes.

Using structural congruence, Definition 3.2.1, we can show that \( RAS \) is congruent to \( RAS2 \) shown in Figure 7.8. Observe that the process \( RAS2 \) is consistent with
7.2. Specification

Definition 5.3.2 and the process,

\[ CM = (\nu \ C : \|_0 V_{Node}, R : \|unit)(SERVER_1 | \ldots | SERVER_n) \]

is a complete mediator (Definition 5.3.1).

7.2.2 Demonstrating dynamic behaviour

From the above description RAS is a dynamic system allowing the creation of Nodes which can move around. Figure 7.7 illustrates a scenario of a Node instance being passed from one server to another. The first event in this scenario is a request \( inc_i \) for another resource at \( SERVER_i \). In this case the request results in a request to a neighbouring server \( SERVER_k \) along \( req_i,k \). That server picks up a machine reference \( z \) from the pool of local free machines, and then passes \( z \) along channel \( c_{k,i} \) in response to \( SERVER_i \)'s request \( req_i,k \). Once this last communication has occurred, \( SERVER_k \) no longer has access to \( z \). Thus the node becomes wholly under the control of \( SERVER_i \), which is now able to issue the instruction \( z \ GetBusy \) and make use of this resource.
$RAS \equiv \pi RAS2$

where

\[
RAS2 = (\nu \ C : \#o V_{Node}, R : \#unit, E : \#o V_{Node})(SERVER_1 | \ldots | SERVER_n | MGEN_1 | \ldots | MGEN_n)
\]

and

\[
E = \{create_i | i \in I\}
\]

Figure 7.8: Identifying a controller

<table>
<thead>
<tr>
<th>Control Point</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>status = Free</td>
</tr>
<tr>
<td>$c_{i,j}$</td>
<td>status \neq Busy</td>
</tr>
</tbody>
</table>

Figure 7.9: Assertions associated with the control points within RAS

### 7.3 Machine divergence freedom verification

The aim of this section is to show that $CM$, defined above, is consistent with respect to controlling 'Node' machines. It is necessary to ensure that $z \ GetBusy\_\{\ast\}$ is invoked only when the Node instance referenced by $z$ is not already busy, since this requirement is encapsulated by the precondition of the operation. In order to guarantee this, we identify assertions on the states of the machine instances whose references are passed across control points. Any machine whose unique reference $z$ is passed on $p_i$ must have $status = Free$, since it can only appear on $p_i$ following $z \ GetFree\_\{\ast\}$. Finally, we can associate the assertion $status \neq Busy$ with machine references passed along the $c_{i,j}$ control points. The collection of assertions on the control points is given in Figure 7.9.

We do not decorate the syntax of the mediators with these predicates as was proposed in the previous chapter for simplicity. We can also see that any machine reference passed along $create$, must have $status = Fresh$ for the associated machine instance, since such an instance will still be in its initial state. These assertions are included in the initial relation $assert\_init$ proposed in Definition 6.1.4.

\[
assert\_init = \{(create_i, (status = Fresh)) | i \in I\}
\]

In general, we identify assertions on the states of machine instances at control points within the mediators, i.e. when the instances were created, when they were passed from one mediator to another, and during internal communication. When a mediator receives a reference to a newly created machine instance then we can assume that the instance is in its initial state. When a machine instance is received by a mediator we may assume the instance will be a particular state. It is the responsibility of the mediator relinquishing control of the instance to guarantee that this assumption is met.
7.3. Machine divergence freedom verification

7.3.1 Establishing consistency using DIV

In this section we apply the DIV function (Definition 6.1.5) to identify a weakest precondition style proof obligation for CM. We illustrate how the proof obligation can be broken down into smaller proof obligations. Discharging all of the individual obligations ensures that the RAS system is machine divergence-free.

Initially, we set \( \rho \) to \( \emptyset \) because CM does not have any active machine references and we are interested in the full behaviour with regards to any machine references it may receive as it evolves.

\[
DIV_0(CM, \text{assert\_init}) = \]
\[
DIV_0(\text{SERVER}_1, \text{assert\_init} \cup \text{assert\_new}) \land \ldots \land
DIV_0(\text{SERVER}_n, \text{assert\_init} \cup \text{assert\_new})
\]
where
\[
\text{assert\_new} = \bigcup_{i \in I} \{(c_{i,j}, (\text{status} \neq \text{Busy})) | j \in C_i\}
\]

by Definition 6.1.5

We will consider only one of the conjuncts of the above as all of them are quite similar. Consider \( DIV_0(\text{SERVER}_1, \text{assert\_init} \cup \text{assert\_new}) \) and by applying Definition 6.1.5 we obtain the following:

\[
DIV_0(\text{SERVER}_1, \text{assert\_init} \cup \text{assert\_new})
= DIV_0(\{(\text{inc}_1 \cdot (\text{create}_1(x) \cdot \text{WORK}_i(x))
+ p_1(x) \cdot \text{WORK}_i(x)
+ \Sigma_{k \in S_1 \text{req}_{i,k} \cdot c_{k,1}(z) \cdot \text{WORK}_i(z))
+ \Sigma_{j \in C_1} (\text{req}_{j,1} \cdot (\text{create}(x) \cdot \text{SEND}_1(j,z)
+ p_1(z) \cdot \text{SEND}_1(j,z)
+ \Sigma_{k \in S_1 \text{req}_{j,k} \cdot c_{k,1}(z) \cdot \text{SEND}_1(j,z))\},
\text{assert\_init} \cup \text{assert\_new} \cup \{(p_1, \text{(status} \neq \text{Busy}))\})
= wp(\text{convert}(\text{inc}_1 \cdot (\text{create}_1(x) \cdot \text{WORK}_i(z)
+ p_1(x) \cdot \text{WORK}_i(x)
+ \Sigma_{k \in S_1 \text{req}_{i,k} \cdot c_{k,1}(z) \cdot \text{WORK}_i(z))
+ \Sigma_{j \in C_1} (\text{req}_{j,1} \cdot (\text{create}(x) \cdot \text{SEND}_1(j,z)
+ p_1(z) \cdot \text{SEND}_1(j,z)
+ \Sigma_{k \in S_1 \text{req}_{j,k} \cdot c_{k,1}(z) \cdot \text{SEND}_1(j,z)))\}
\]

where the assertion function is given by
\[
\text{assert\_init} \cup \text{assert\_new} \cup \{(p_1, \text{(status} \neq \text{Busy}))\}
\]
By applying Definition 6.1.6 we obtain

\[
= wp(CHOOSE \ convert(inc_1 \cdot (create_1(z) \cdot WORK_i(z))
+ p_1(z) \cdot WORK_i(z)
+ \Sigma_{k \in S_i \overline{req_i,k}} \cdot c_k(1(z) \cdot WORK_i(z)))_0
\]

\[
OR \ convert((\overline{req_i,1} \cdot (create(z) \cdot SEND_i(j_1, z))
+ p_i(z) \cdot SEND_i(j_1, z)
+ \Sigma_{k \in S_i \overline{req_i,k}} \cdot c_k(1(z) \cdot SEND_i(j_1, z))))_0
\]

\[
\text{OR} \ldots
\]

\[
\text{OR} \ convert((\overline{req_i,1} \cdot (create(z) \cdot SEND_i(j_h, z))
+ p_i(z) \cdot SEND_i(j_h, z)
+ \Sigma_{k \in S_i \overline{req_i,k}} \cdot c_k(1(z) \cdot SEND_i(j_h, z))))_0
\]

\[
\text{END}, \ true
\]

where \( \{j_1, \ldots, j_h\} = C_1 \)

According to Definition 2.4.3, the weakest precondition of a \textit{CHOICE} statement to establish a predicate is the conjunction of the weakest preconditions of each separate branch statement from which the choice is formed to establish the predicate. Therefore, we now separate the statement above and check each branch separately. We consider only two branches as all of them are quite similar. These are shown in Equations 7.2 and 7.6 below.

\[
wp(\text{convert}(inc_1 \cdot (create_1(z) \cdot WORK_i(z))
+ p_1(z) \cdot WORK_i(z)
+ \Sigma_{k \in S_i \overline{req_i,k}} \cdot c_k(1(z) \cdot WORK_i(z)))_0
\]

\[
= wp(\text{convert}((create_1(z) \cdot WORK_i(z))
+ p_1(z) \cdot WORK_i(z)
+ \Sigma_{k \in S_i \overline{req_i,k}} \cdot c_k(1(z) \cdot WORK_i(z)))_0
\]

\[
= wp(CHOOSE \ convert(create_1(z) \cdot WORK_i(z)))_0
\]

\[
\text{OR} \ convert(p_1(z) \cdot WORK_i(z))_0
\]

\[
\text{OR} \ convert(\overline{req_i,k_1} \cdot c_{k_1,1}(z) \cdot WORK_i(z))_0
\]

\[
\text{OR} \ldots
\]

\[
\text{OR} \ convert(\overline{req_i,k_y} \cdot c_{k_y,1}(z) \cdot WORK_i(z))_0
\]

\[
\text{END}, \ true
\]

where \( \{k_1, \ldots, k_y\} = S_i \)

For the same reasons as above we illustrate the derivation using the first three branches of Equation 7.2 as all of them are quite similar. These branches are considered in
Equations 7.3, 7.4 and 7.5 respectively.

\[ wp(\text{convert}(\text{create}_1(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Fresh} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree};
\text{PRE } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{skip, true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]

\[ wp(\text{convert}(p_1(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree};
\text{PRE } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{skip, true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]

\[ wp(\text{convert}(p_2(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]

\[ (7.3) wp(\text{convert}(\text{create}_1(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Fresh} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree};
\text{PRE } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{skip, true}) = z\_\text{status} = \text{Fresh} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]

\[ (7.4) wp(\text{convert}(p_1(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]

\[ (7.5) wp(\text{convert}(p_2(z) \cdot \text{WORK}_1(z))_0, \text{true}) = wp(\text{SELECT } z\_\text{status} = \text{Free} \text{ THEN skip END}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_1(z))_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}; \text{convert}(p_1(z) \cdot 0)_{\{z\}, \text{true}) = z\_\text{status} = \text{Free} \Rightarrow wp(\text{skip}; z\_\text{GetBusy}; z\_\text{GetFree}, z\_\text{status} = \text{Free})
\]
\[ wp(\text{convert}(\overline{\text{req}_{k_1}} \cdot c_{k_1,1}(z) \cdot \text{WORK}_t(z))_0, \text{true}) \]
\[ = wp(\text{convert}(c_{k_1,1}(z) \cdot \text{WORK}_t(z))_0, \text{true}) \]
\[ = wp(\text{SELECT } z\_status \neq \text{Busy} \text{ THEN } \text{skip END; convert}(\text{WORK}_t(z))_{\{z\}}, \text{true}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; \text{convert}(\text{WORK}_t(z))_{\{z\}}, \text{true}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; z\_GetBusy; z\_GetFree; \text{convert}(p_1(z) \cdot 0)_{\{z\}}, \text{true}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; z\_GetBusy; z\_GetFree; \]
\[ \text{PRE } z\_status = \text{Free} \text{ THEN } \text{skip END; skip, true} \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; z\_GetBusy; z\_GetFree, z\_status = \text{Free}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; \]
\[ \text{PRE } z\_status = \text{Busy} \text{ THEN } z\_status := \text{Free END}, \]
\[ z\_status = \text{Free}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; z\_GetBusy, z\_status = \text{Busy}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}; \text{PRE } z\_status \neq \text{Busy} \text{ THEN } z\_status := \text{Busy END}, \]
\[ z\_status = \text{Busy}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow wp(\text{skip}, z\_status \neq \text{Busy}) \]
\[ = z\_status \neq \text{Busy} \Rightarrow z\_status \neq \text{Busy} \]
\[ = \text{true} \hspace{1cm} (7.5) \]

We also need to backtrack to Equation 7.1 and show that one of the other execution branches is valid by Equation 7.6.

\[
\begin{align*}
wp(\text{convert}(\text{create}(z) \cdot \text{SEND}_1(j_1, z)) \\
&+ p_1(z) \cdot \text{SEND}_1(j_1, z) \\
&+ \Sigma_{k \in S_1} \text{req}_{k_1} \cdot c_{k_1,1}(z) \cdot \text{SEND}_1(j_1, z)))_0,
\end{align*}
\]
\[ = wp(\text{convert}(\text{create}(z) \cdot \text{SEND}_1(j_1, z)) \\
&+ p_1(z) \cdot \text{SEND}_1(j_1, z) \\
&+ \Sigma_{k \in S_1} \text{req}_{k_1} \cdot c_{k_1,1}(z) \cdot \text{SEND}_1(j_1, z)))_0,
\]
\[ = wp(CHOOSE \text{convert}(\text{create}(z) \cdot \text{SEND}_1(j_1, z))_0 \]
\[ OR \text{convert}(p_1(z) \cdot \text{SEND}_1(j_1, z))_0 \]
\[ OR \text{convert}(\overline{\text{req}_{k_1}}, c_{k_1,1}(z) \cdot \text{SEND}_1(j_1, z)))_0 \]
\[ OR \ldots \]
\[ OR \text{convert}(\overline{\text{req}_{k_g}}, c_{k_g,1}(z) \cdot \text{SEND}_1(j_1, z)))_0 \]
\[ , \text{true}) \]
\[ \text{where } \{k_1, \ldots, k_g\} = S_1 \hspace{1cm} (7.6) \]

Again we check only three branches of Equation 7.6 presented in Equations 7.7, 7.8,
and 7.9 below.

\[
\begin{align*}
wp(\text{convert}(\text{create}(z) \cdot \text{SEND}_1(j_1, z)))_q, \text{true}) \\
= wp(\text{SELECT } z\text{-status} = \text{Fresh} \; \text{THEN} \; \text{skip END}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Fresh} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Fresh} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{create}(z) \cdot 0)_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Fresh} \Rightarrow wp(\text{skip}; \; \text{PRE } z\text{-status} \neq \text{Busy} \; \text{THEN} \; \text{skip END}; \; \text{skip, true}) \\
= z\text{-status} = \text{Fresh} \Rightarrow z\text{-status} \neq \text{Busy} \\
= \text{true} & \quad (7.7)
\end{align*}
\]

\[
\begin{align*}
wp(\text{convert}(p_1(z) \cdot \text{SEND}_1(j_1, z)))_q, \text{true}) \\
= wp(\text{SELECT } z\text{-status} = \text{Free} \; \text{THEN} \; \text{skip END}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Free} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Free} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{create}(z) \cdot 0)_{\{z\}}, \text{true}) \\
= z\text{-status} = \text{Free} \Rightarrow wp(\text{skip}; \; \text{PRE } z\text{-status} \neq \text{Busy} \; \text{THEN} \; \text{skip END}; \; \text{skip, true}) \\
= z\text{-status} = \text{Free} \Rightarrow z\text{-status} \neq \text{Busy} \\
= \text{true} & \quad (7.8)
\end{align*}
\]

\[
\begin{align*}
wp(\text{convert}(\text{req}_{1,k_1} \cdot c_{k_1}(z) \cdot \text{SEND}_1(j_1, z)))_q, \text{true}) \\
= wp(\text{convert}(c_{k_1}(z) \cdot \text{SEND}_1(j_1, z)))_q, \text{true}) \\
= wp(\text{SELECT } z\text{-status} \neq \text{Busy} \; \text{THEN} \; \text{skip END}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} \neq \text{Busy} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{SEND}_1(j_1, z))_{\{z\}}, \text{true}) \\
= z\text{-status} \neq \text{Busy} \Rightarrow wp(\text{skip}; \; \text{convert}(\text{create}(z) \cdot 0)_{\{z\}}, \text{true}) \\
= z\text{-status} \neq \text{Busy} \Rightarrow wp(\text{skip}; \; \text{PRE } z\text{-status} \neq \text{Busy} \; \text{THEN} \; \text{skip END}; \; \text{skip, true}) \\
= z\text{-status} \neq \text{Busy} \Rightarrow z\text{-status} \neq \text{Busy} \\
= \text{true} & \quad (7.9)
\end{align*}
\]

By discharging the above obligations we have shown that \(\text{DIV}_q(\text{CM, assert_init})\) is true and hence \(\text{CM}\) is a consistent mediator for controlling Node machines. Hence, by Theorem 6.5.4 in Chapter 6 we can deduce that the RAS control system is machine divergence-free.

### 7.4 Discussion

In the above example we have shown that state and the operations which update and query the state can be described in conjunction with a mobile paradigm. We demonstrated how 'Node' machines were associated
with unique references so that they can be instantiated at run-time by a \( \pi \) \textit{SERVER} process. We used our syntactic framework for \( \pi \) processes in order to define the RAS system in order to control the execution of B operations. We achieved this by first constructing mediators:

- using parallel composition (of several mediators which are disjoint of the sets of active machine references),
- using hiding of non-machine reference channels in a mediator,
- using the sequential parts of the \( \pi \) language, and
- using infinite replication.

We also used a behavioural type system within the \( \pi \)-calculus to provide guarantees on the way machine instances and processes should interact. We converted the signature of a machine instance into a variant type which specifies the operations a \( \pi \) process can execute. Without such a type system it would be difficult to specify the interface of a machine instance and ensure that \( \pi \) processes do not call operations which are not in that interface. The typing system provides the guarantee that any operation call will always be serviced appropriately by a machine instance and not by another \( \pi \) processes (pretending to be a machine). Using this typing system we were able to define an \textit{MGENERATOR} process which created instances of B machines.

We showed that discharging weakest-precondition proof obligations ensures that the 'Node' machines instances do not diverge when controlled by \textit{SERVER} processes. The steps we followed to achieve this were the following:

- identify the control points in all mediators,
- assign assertions to each control point,
- verification using wp-style proof.

The rely/guarantee style reasoning ensures the control point assertions are respected, and that operations are never called out of their preconditions. Whenever a machine reference is input at a control point we may assume that the machine instance satisfies the corresponding assertion. This is then enough to ensure (1) that the mediator receiving the machine reference calls operations appropriately; and (2) that it can guarantee the assertion is true at any control point where it outputs the machine reference. The nature of mediators ensures that a machine reference is always located at no more than one mediator, ensuring that state updates are strictly controlled, and are the responsibility of a single mediator from the point the machine reference is received on the control point. Hence, there can be no interference on the updates of the machine's state.
The example presented above highlighted dynamic instantiation and control passing communication. We could extend the notion of dynamic instantiation so that the servers, as well as workers, could also be generated dynamically, possibly by some kind of server controller, and they could even dynamically change the network by adding new links allowing them to pass requests for resource to new servers. Furthermore, we could consider descriptions of a P2P network overlay, as in [2], where the complex topology of the servers was stored in the B and this information passed to the π processes.
Chapter 8

Conclusion

8.1 Contribution

In this thesis we presented a framework which gives machines, whose operations are defined without input and output parameters, a π-calculus operational semantics. Our main contribution is that data structures with complex state can be studied in conjunction with the mobility paradigms implemented by the π-calculus. We will refer to some emerging alternative approaches in the next section but our research is novel with respect to modelling a specification using a state-based method and a mobile process calculus.

One aspect of our work extends B-Machines with unique references so that they can be instantiated as objects in a π-specification at runtime, and this was discussed in Chapter 4. A reference can be used in a π-specification to select the execution of an operation on the machine instance in an object oriented style. Machines execute operations sequentially as they receive operation requests. More importantly the references can be communicated between π-calculus agents so that machine instances can be connected to the π-specification in a flexible way. All of the above is facilitated by the semantics of the π-calculus.

We identified an important behavioural requirement on the π-calculus specification which is necessary to ensure internal machine instance consistency. After a process outputs a machine reference to another process it cannot subsequently refer to that machine reference (i.e. output again or execute another operation). This is because no guarantees can be given about the state of the machine past such a point. In general this requirement cannot be relaxed except in certain cases which we detail out below.

To propose our new formal framework we had to appreciate the contribution that behavioural type-systems provide to the modelling of distributed sys-
tems. In the \( \pi \)-calculus specifically, a type-system is the clearest mechanism by which the use of channels in various parts of the system can be rigorously specified. Using a type-system in Chapter 5, we defined a syntactic framework for agents that meet the above mentioned behavioural requirement. This type-system also provides essential guarantees about the use of machine reference channels in a \( \pi \)-specification. For example, the signature of an operation is converted to a special type, called a variant, which enforces what operations \( \pi \)-agents are permitted to select. Furthermore, because of the type-system, agents cannot hijack a machine reference and then appear to other agents as a machine.

In Chapter 6 we provided a syntactic weakest precondition based method which can be applied to verify machine divergence freedom of a combined \( \pi \) and B specification. This method can be used on any \( \pi \) mediator from the above mentioned framework to ensure that the operations calls made by the mediators are always done when the preconditions of the operations hold. The framework breaks the proofs into smaller weakest precondition proofs and uses a rely-guarantee style of reasoning.

In Chapter 7 we described a case study of a resource allocation system which demonstrated the mobile and dynamic aspects of our new \( \pi | B \) framework, together with detailed workings of how the weakest precondition verification is applied in practice.

Currently the CSP\( ||B \) approach does not include mobility. The principal motivation for this work was to draw out the complex issues related to dynamic interactions so that this could guide future research to include mobility in the CSP\( ||B \) framework. We have identified the most important behavioural requirement that must be met in that respect. We believe that it would be possible to migrate our mobile verification framework to CSP\( ||B \). In the framework identified in this thesis it is possible to weaken the behavioural requirement and allow \( \pi \)-agents to delegate the capability to execute certain operations to other agents in a dynamic manner. This could include allowing other agents to execute query operations on the machine instance while retaining overall control of the machine. Such operations can be bundled in a variant type, which for a particular machine, is a sub-type of all available operations. A controlling \( \pi \)-agent can output the machine reference with this constrained variant to other agents, which would ensure that those agents which receive it can execute only query operations. Even though in [17] Evans et al. have discussed allowing more than one controller in CSP access to a B machine this was done in a restricted way. Whether CSP\( ||B \) could be extended to include the delegation capability described above so that it is as flexible as the approach presented in this thesis, remains an open issue.
8.2 Related work

Our work is related to an integration of Object-Z and the π-calculus presented in [27]. In their framework π-calculus channels are semantically identified with state variables in Object-Z schemas and both process abstractions and input prefix are identified with operations which may perform state updates. For example, input prefix \( a(x).P \) is equivalent to the Object-Z schema which modifies \( x \) such that \( x' = y \) for some \( y \) and then continues as \( P \). A process abstraction with \( n \) parameters, \( X(a_1, \ldots, a_n) \) relates to executing schema named \( X \) with inputs \( a_1, \ldots, a_n \). The processes syntax allowed in their schemas does not include the \((v \nu)\) operator. Instead the framework relies on channel generation from the Object-Z semantics. Reductions of the language are defined in terms of tuples containing processes and state. We are cautious about this level of integration between the π-calculus and a state oriented language. The semantics of a name in the π-calculus is not to be confused with that of a state holder. Intuitively, a π-name is similar to a pointer whereas a variable has a pointer and resides on an allocated space in the memory of a computer.

Another closely related investigation is [61] where the operators of the mobile version of Occam, Occam \( - \pi \) (previously known as Occam \( - \mathcal{M} \)) have been studied. The aim is to develop a theory of mobile processes for the Circus [68] framework. In [61], the operators of Occam \( - \mathcal{M} \) are given denotational semantics in Hoare and He's Unifying Theories of Programming [25]. Thus, the main benefit is that a set of refinement laws are defined so that they can be integrated with Circus' refinement calculus. This may give a mature formal framework for developing distributed systems. The syntax of Occam \( - \pi \) is essentially higher order CSP processes extended with variable definition and assignments. A variable can be a program variable, a channel, or a process. While the operators for input and output are similar to standard CSP this framework contains a new output prefix which is specifically used for outputting process variables. The object in the prefix becomes unspecified after the interaction hence any attempt to use it becomes unpredictable. This means that if a process attempts to output the same mobile component twice then the second time the receiving process does not have any guarantee about the value of the variable it has received. This bears some resemblance with the way mediators move machine references in our framework. As stated above it is important that mediators do not use machine references after outputting them because there is no guarantee about the state of the machine after the interaction. In this respect mediators and non-divergent Occam \( - \pi \) processes receive only fresh machines and components respectively.

In [49] the mobility of the π-calculus is separated into two mechanisms: internal mobility and external mobility. Internal-mobility, is akin to the way mediators move machine references, and external mobility is akin to the way
mediators can move names other than machine references. One result of the paper shows that when considering internal mobility, infinite replication is less expressive than recursion. For example the behaviour of the process \( P \triangleq (x).x(y).P(y) \) cannot be matched with a process containing a bang operator \(!\) instead. Throughout our work we have used infinite replication, however the above has no major impact. This is because only machine references are communicated with internal mobility and we cannot think of an application where the process above would be useful. In our framework it is not the case that \( x \) and \( y \) have the same type - \( x \) is a control point, \( y \) is a machine reference.

A more remarkable aspect in [49] is that it defines a hierarchy of calculi based on structural types. The hierarchy is applied to each of the name-passing and higher order frameworks of the \( \pi \)-calculus, to grade various orders of mobility. This has greatly clarified the correspondence between the two frameworks. In general, the \( \pi \)-calculus and higher-order processes are equivalent in terms of expressiveness [48]. However, it would be interesting to see if Occam – \( \pi \) mobility mechanisms can be encoded in the above mentioned framework because then the expressiveness of higher order CSP can be put in context with the expressiveness of the \( \pi \)-calculus with internal mobility.

In general our concerns regarding mobility and integrity of data, expressed in this thesis, are strongly reflected in Peter Welch's work on Occam-\( \pi \) where mobile channels have recently been included in the language. Welch has proposed a CSP model for mobile channels [67]. The motivation for this was that Occam-\( \pi \)'s classical design was founded on CSP. We also noted in the introductory chapter that we could encode the notion of dynamically acquiring channels using infinite sets and recursion. Welch notes that this gives flexibility when modelling but in order to produce simulations a clearer model of mobility is needed. In Occam-\( \pi \) communication via channels has a notion of its originator and recipient and they can change during run-time. CSP does not explicitly define which process has the responsibility for its channel ends. Therefore, Welch has proposed that mobile channels be modelled as CSP processes, and each process is produced on demand. Each of these processes has an unique identifier number and the mobility in a formal model comes from communicating the index. The formal model presented in [67] is very low level and reflects how it can be implemented successfully in Occam-\( \pi \). This is exemplified by some impressive simulations of biological systems using this technology [41], including a 3D simulation [45].

### 8.3 Future work

This section outlines some interesting ways in which this work can be extended.
8.3. Future work

8.3.1 Extension to machines with input and output

An important extension to this thesis is to include machines where the operations can have input and output parameters.

We would require a preliminary labelled transition system similar to our \( \text{LTS}_M \), which takes into account the input and output parameters of an operation. This is important because when ensuring that an operation is called within its precondition we may need to refer to the input values in addition to \( M \)'s state. The same are also necessary for selecting appropriate output values.

There are examples in the literature which could provide an insight on how to extend the labelled transition system. For example, in [10] we can find relational semantics for \( Z \) and Object-\( Z \), where inputs and outputs are modelled as sequences of parameters. The B relational model has also been implemented in PVS by [17]. In that work inputs and outputs were considered as temporary state in which the main state space was expanded to model the call of an operation with I/O. Following the execution of an operation the I/O state was discarded thus making sure that it was relevant in the frame of just that call.

It is also possible to continue in the spirit of this thesis and extend the valuation functions from Section 2.2 to include the inputs and outputs. Given an operation with inputs and outputs \( \text{out} \leftarrow \text{operation}(\text{in}) \) where \( \text{in} \) and \( \text{out} \) are lists of variable names, it can be shown that,

\[
\text{fn}(\text{abt}(\text{out} \leftarrow \text{operation}(\text{in}))) \subseteq (\text{set}_{-}\text{of}(\text{VARIABLES}) \cup \text{set}_{-}\text{of}(\text{out}) \cup \text{set}_{-}\text{of}(\text{in}))
\]

\[
\text{fn}(\text{prdVARIABLES}, \text{out}(\text{out} \leftarrow \text{operation}(\text{in}))) \subseteq
(\text{set}_{-}\text{of}(\text{VARIABLES}) \cup \text{set}_{-}\text{of}(\text{VARIABLES'}) \cup \text{set}_{-}\text{of}(\text{out'}) \cup \text{set}_{-}\text{of}(\text{in}))
\]

where each \( \text{set}_{-}\text{of}(\text{VARIABLES}), \text{set}_{-}\text{of}(\text{VARIABLES'}), \text{set}_{-}\text{of}(\text{out}), \text{set}_{-}\text{of}(\text{in}) \) are pairwise disjoint.

This means that for the particular operation \( \text{out1} \leftarrow \text{operation1}(\text{in1}) \) we can provide valuations of the following form,

\[
\text{val}_1 \in \text{set}_{-}\text{of}(\text{VARIABLES}) \rightarrow \mathcal{D}_B
\]

\[
\text{primed}(\text{val}_2) \in \text{set}_{-}\text{of}(\text{VARIABLES'}) \rightarrow \mathcal{D}_B
\]

\[
\text{val}_{\text{in1}} \in \text{set}_{-}\text{of}(\text{in1}) \rightarrow \mathcal{D}_B
\]

\[
\text{val}_{\text{out1}} \in \text{set}_{-}\text{of}(\text{out1}) \rightarrow \mathcal{D}_B
\]
with which we can test whether

\[ [val_1, val_{in1}] \text{ satisfies } abt(out1 \leftarrow operation1(in1)) \]

\[ [val_1, val_{in1}, primed(val_2), primed(val_{out1})] \text{ satisfies } \]

\[ \text{prdVARIABLES, out1(out1 \leftarrow operation1(in1))} \]

This means that a definition can be provided, extending Definition 4.1.1 in a natural way, where for some relation \( R \) we have tuples of the form,

\[(val_1, val_{in1}, operation, val_2, val_{out1}) \in R\]

All other forms of an operation are special cases of the above. For example, if we have an operation with output parameters but no input parameters \( out2 \leftarrow operation2 \) we can have the following tuple,

\[(val_1, \emptyset, operation2, val_2, val_{out2}) \in R\]

where \( \emptyset \) denotes the empty valuation noting that \( operation2 \) has no input parameters.

Similarly an operation with input parameters but without output parameters such as \( operation3(in3) \) becomes,

\[(val_1, val_{in3}, operation2, val_2, \emptyset) \in R\]

Operations without input and output parameters such as the ones we have considered in this thesis become,

\[(val_1, \emptyset, operation4, val_2, \emptyset) \in R\]

Once we have identified a way to represent input and output states then we need to re-examine how divergence is modelled. We need to be careful because the design decisions at this stage might influence the \( \pi \)-calculus operational semantics.

For example, suppose that we have the operation \( out1 \leftarrow operation1(in1) \) and \([val_1, val_{in1}]\) is the common extension of the machine state before the execution of \( operation1 \) and the values of \( in1 \). Suppose that \([val_1, val_{in1}]\) can cause \( operation1 \) to diverge. Then the tuples containing \( val_1 \) and \( val_{in1} \) as the before state could come from the evaluation of the \( prdVARIABLES, out(out \leftarrow operation(in)) \) predicate or from an evaluation of \( abt(out \leftarrow operation(in)) \) or \( val_1 = \bot \).

The first case would require that we add a tuple with any possible after state \( val_2 \) and valuation \( val_{out1} \) for any possible output except the \( \emptyset \) valuation. This is because the variables \( out1' \) appear in the \( prd \) predicate and we
would be required to assign a value to check whether 

\[ [\text{val}_1, \text{val}_{\text{in}1}, \text{primed}(\text{val}_2), \text{primed}(\text{val}_{\text{out}1})] \] satisfies it.

In the second and last case we do not have such a requirement. Variables \( \text{out}1 \) do not appear in the \( \text{abt} \) predicate hence we can just add one tuple where \( \text{val}_{\text{out}1} = \emptyset \). This would underline the notion of the possibility of non-termination. However, in the context of a combination with the \( \pi \)-calculus where input and outputs happen sequentially it could have a serious impact. This would mean that the machine agent may block and not provide an output after divergence. This in turn may have an impact on those proofs that rely on a machine reaching a \emph{READY} state.

Another design decision is to have some \( \text{val}_{\text{out}} \) that is not \( \emptyset \) in all tuples that can diverge. This means that the machine can provide some output and reach a \emph{READY} state. This however raises questions as to whether we consider a divergent machine as well-typed in the context of the \( \pi \)-calculus as it could provide any output that may not even reflect types given to appropriate channels.

On the side of the \( \pi \)-calculus one would definitely need to upgrade to the polyadic \( \pi \)-calculus [34] so that transmissions of multiple names can be handled on a single channel. We could also revisit the type-system proposed by Gay and Hole in [21] so that we can consider a machine reference channel as a session channel. This would mean that we would need only one channel to represent all communications between a machine instance and the \( \pi \)-agents. This eliminates the need to have one channel to select an operation and another to collect the output from the operation. In comparison we could use a type-system with linear-receptive channels such as the one expressed in [51] \( (\text{rec}) \). If a process creates an \( \text{rec} \) channel then the input end of the channel must be available immediately afterwards and also the link can be used just once. This means that the processes must be of the form

\[
(\nu \ q : \text{trec}_\text{link}) (\text{output}(q).P_1 | q(f).P_2)
\]

where \( q \notin \text{fn}(P_1) \cup \text{fn}(P_2) \).

Consider adding a channel \( q \) to Figure 4.3 on page 59 of type \( \text{trec} \). To execute an operation with output the mediator would create a new \( \text{trec} \) channel such as \( q \) and then pass the parameters of the operation along the new channel as an extra parameter. This establishes a private channel between a mediator and a machine instance for the duration of the execution. Then the machine would execute the operation and return the outputs on the provided link. Then the machine and process would close the link. Syntactically, to execute \( \text{out} \leftarrow \text{operation}(\text{in}) \) a mediator would have to perform the following sequence of actions,

\[
(\nu \ q : \text{trec}_\text{link}) (\exists \ \text{operation}_\text{-}(\text{in, } q)).0 | q(\text{out}).D' )
\]
The importance of using linearly receptive channels for output carrier links is that we ensure mediators do not forget to collect an output from the machine and thus inadvertently blocking it. Furthermore, using a type-system in general ensures that only appropriate values are passed as input parameters to the machine.

Correspondingly, we would need to extend the set $IS$ from Definition 4.3.1 with a special $OUTPUT$ state to specify that a machine is in a state that is ready to provide output. Thus, we would need to have an operational semantics for the process $\llbracket (OUTPUT, x) \rrbracket_M(z, q)$ where $q$ is the output carrier link and $s$ is some augmented state space containing outputs.

### 8.3.2 Polyadic control points

By upgrading to the polyadic $\pi$-calculus we can express control points that transmit more than one machine reference with a single interaction.

An interesting approach is to tag predicates to control points that express some relationship between the states of the machines received at that point. The predicate would be similar to those encountered when two machines are linked with a $USES$ statement [1].

Such predicates could be useful in specifying complicated state component relationships that must hold before and after the execution of some control sequence.

### 8.3.3 Dynamic control points

In this thesis control points were static channels. This was done in the interest of clarity rather than to avoid some important technical problem.

To include control point mobility we can tag every name in a given process, similar to the way types are assigned to channels. The tag that we apply to a name can be a $B$ predicate prefixed with $\$\$ to indicate that this is a link that carries a control point which can input a machine instance in state satisfying the predicate. Thus, $\$$ $A$ is a link that carries a link that carries a control point etc. Links that do not input a machine reference could be tagged with $true$. If such a link inputs another link then the second link is also tagged with $true$. Thus, control points that are permitted to input a machine in any state and channels that do not input machines will be tagged with the same predicate.

The rules of $DIV$ could then be modified to take into account these assertions. Currently $assert$ only maps control points to some predicate $R$,
here \textit{assert} would resemble a typing environment where every free name of the process that is under investigation is mapped to some predicate of the kind described above. This extension would extend the expressiveness of the method.
Bibliography


