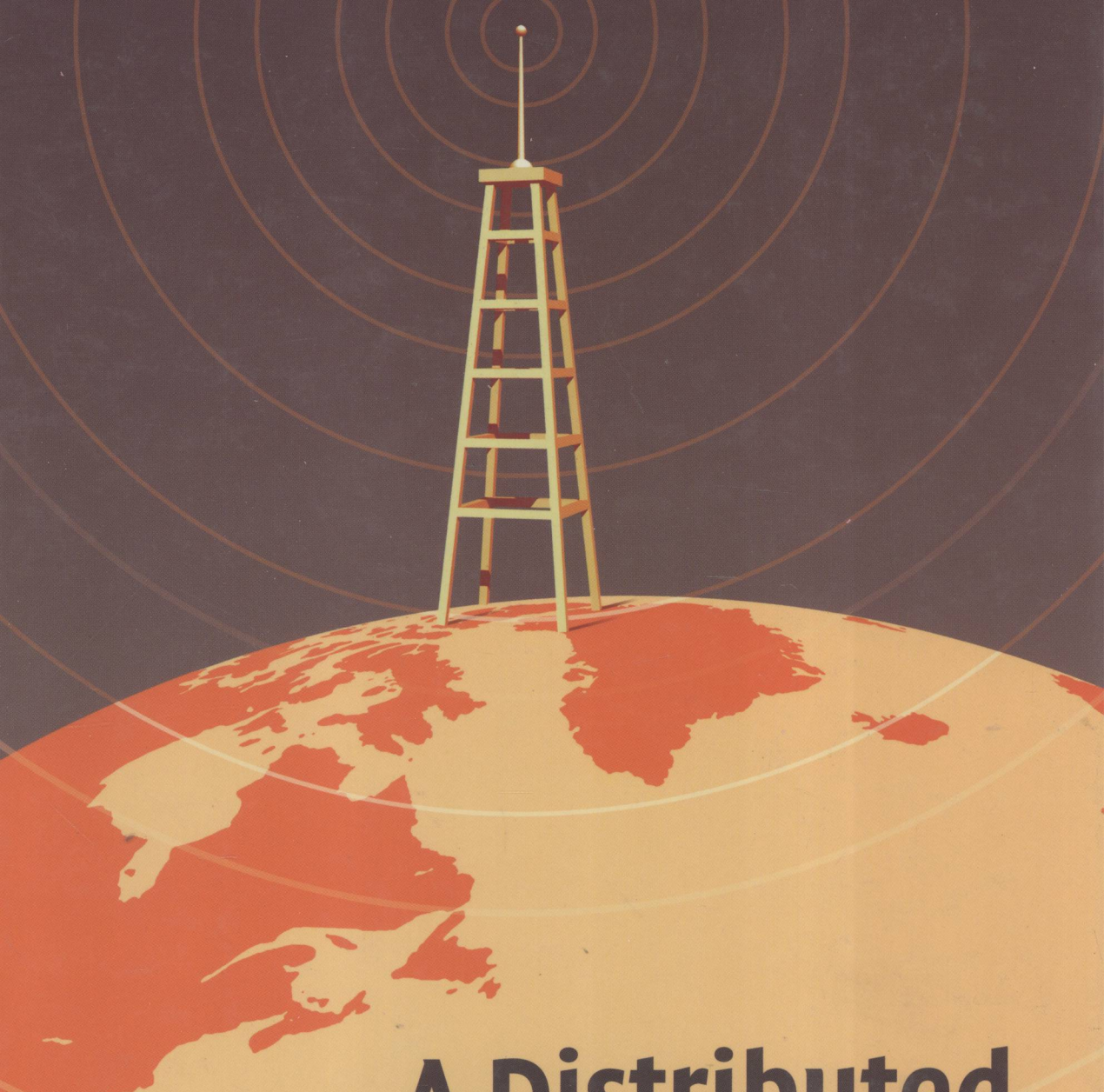


A Distributed Pi-Calculus

Matthew Hennessy

MBRIDGE

A DISTRIBUTED PI-CALCULUS

MATTHEW HENNESSY

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521873307

First published 2007

Printed in the United Kingdom at the University Press, Cambridge

A catalogue record for this publication is available from the British Library

ISBN-13 978-052-1-87330-7 hardback

A DISTRIBUTED PI-CALCULUS

Distributed systems are fast becoming the norm in computer science. Formal mathematical models and theories of distributed behaviour are needed in order to understand them. This book proposes a distributed PI-CALCULUS called ADPI, for describing the behaviour of mobile agents in a distributed world. It is based on an existing formal language, the PI-CALCULUS, to which it adds a network layer and a primitive migration construct.

A mathematical theory of the behaviour of these distributed systems is developed, in which the presence of types plays a major role. It is also shown how, in principle, this theory can be used to develop verification techniques for guaranteeing the behaviour of distributed agents.

The text is accessible to computer scientists with a minimal background in discrete mathematics. It contains an elementary account of the PI-CALCULUS, and the associated theory of bisimulations. It also develops the type theory required by ADPI from first principles.

To the memory of John and Ray

Preface

From ATM machines dispensing cash from our bank accounts, to online shopping websites, interactive systems permeate our everyday life. The underlying technology to support these systems, both hardware and software, is well advanced. However design principles and techniques for assuring their correct behaviour are at a much more primitive stage.

The provision of solid foundations for such activities, mathematical models of system behaviour and associated reasoning tools, has been a central theme of theoretical computer science over the last two decades. One approach has been the design of *formal calculi* in which the fundamental concepts underlying interactive systems can be described, and studied. The most obvious analogy is the use of the λ-calculus as a simple model for the study of sequential computation, or indeed the study of sequential programming languages. *CCS* (a *C*alculus for *C*ommunicating *S*ystems) [28] was perhaps the first calculus proposed for the study of interactive systems, and was followed by numerous variations. This calculus consists of:

- A simple formal language for describing systems in terms of their structure; how they are constructed from individual, but interconnected, components.
- A semantic theory that seeks to understand the behaviour of systems described in the language, in terms of their ability to interact with users.

Here a system consists of a finite number of independent processes that intercommunicate using a fixed set of named communication channels. This set of channels constitutes a connection topology through which all communication takes place; it includes both communication between system components, and between the system and its users.

Although successful, *CCS* can only describe a very limited range of systems. The most serious restriction is that for any particular system its connection topology is static. However modern interactive systems are highly dynamic, particularly when one considers the proliferation of wide area networks. Here computational

entities, or agents, are highly mobile, and as they roam the underlying network they forge new communication links with other entities, and perhaps relinquish existing links.

The PI-CALCULUS [9, 29] is a development from *CCS* that seeks to address at least some dynamic aspects of such agents. Specifically it includes the dynamic generation of communication channels and thus allows the underlying connection topology to vary as systems evolve. Just as importantly it allows private communication links to be established and maintained between agents, which adds considerably to its expressive power. Indeed the PI-CALCULUS very quickly became the focus of intensive research, both in providing for it a semantic understanding, and in its promotion as a suitable foundation for a theory of distributed systems; see [39] for a comprehensive account.

But many concepts fundamental to modern distributed systems, in particular those based on local area networks, are at most implicit in the PI-CALCULUS. Perhaps the most obvious is that of *domain*, to be understood quite generally as a locus for computational activity. Thus one could view a distributed system as consisting of a collection of domains, each capable of hosting computational processes, which in turn can migrate between domains.

The aim of this book is to develop an extension of the PI-CALCULUS in which these domains have an explicit representation. Of course when presented with such a prospect there is a bewildering number of concerns on which we may wish to focus. For example:

- What is the role of these domains?
- How are they to be structured?
- How is interprocess communication to be handled?
- How is agent migration to be described?
- Can agents be trusted upon entry to a *domain*?

Indeed the list is endless. Here our approach is conservative. We wish to develop a minimal extension of the PI-CALCULUS in which the concept of *domain* plays a meaningful, and non-trivial role. However their presence automatically brings a change of focus. The set of communication channels that in the PI-CALCULUS determines the interprocess communication topology now has to be reconciled with the *distribution topology*. Following our minimalistic approach we decide on a very simple distribution topology, namely a set of independent and non-overlapping domains, and only allow communication to happen within individual domains. This makes the communication channels of the PI-CALCULUS into local entities, in the sense that they only have significance relative to a particular domain. Indeed we will view them as a particularly simple form of *local resource*, to be used by

migrant agents. Thus we view our extension of the PI-CALCULUS, called ADPI – for ASynchronous Distributed PI-CALCULUS, as a calculus for distributed systems in which

- dynamically created *domains* are hosts to resources, which may be used by *agents*
- agents reside in *domains*, and may migrate between domains for the purpose of using locally defined resources.

Types and type inference systems now form an intrinsic part of computer science. They are traditionally used in programming languages as a form of static analysis to ensure that no runtime errors occur during program execution. Increasingly sophisticated type-theoretic concepts have emerged in order to handle modern programming constructs [36]. However the application of type theory is very diverse. For example types can be used to

- check the correctness of security protocols [12]
- detect deadlocks and livelocks in concurrent programs [26]
- analyse information flow in security systems [22].

In this book we also demonstrate how type systems can be developed to manage access control to resources in distributed systems. In ADPI we can view domains as offering resources, modelled as communication channels, to migrating agents. Moreover a domain may wish to restrict access to certain resources to selected agents. More generally we can think of resources having capabilities associated with them. In our case two natural capabilities spring to mind:

- the ability to *update* a resource, that is write to a communication channel
- the ability to *look up* a resource, that is read from a communication channel.

Then domains may wish to distribute selectively to agents such capabilities on its local resources.

We could develop a version of ADPI in which the principal values manipulated by agents are these capabilities. But this would be a rather complex language, having to explicitly track their generation, management, and distribution. Instead we show that these capabilities can be implicitly managed by using a typed version of ADPI. Moreover the required types are only a mild generalisation of those used in a type inference system for ensuring the absence of runtime errors, when ADPI systems are considered as distributed programs.

The behavioural theory of processes originally developed for *CCS* [28] based on *bisimulations*, has been extended to the PI-CALCULUS, and can be readily extended to ADPI. Indeed the framework is quite general. The behaviour of processes can be described, independently of the syntax, in terms of their ability to *interact* with other processes, or more generally with their computing environment. The form

these interactions take depend on the nature of the processes, and in general will depend on the process description language. They can be described mathematically as relations between processes, with

$$P \xrightarrow{l} Q$$

meaning the process P by interacting with its environment can be transformed into the process Q. The label l serves to record the kind of interaction involved, and perhaps some data used in the interaction. For example one can easily imagine the behaviour of an ATM machine being described in this manner, in terms of its internal states, and the evolution between these states depending of the different kinds of interactions with a customer. Such a behavioural description is formalised as a *labelled transition system*, or *lts*, and often refered to as an *operational semantics*.

The theory of bisimulations enables one to take such abstract behavioural descriptions of processes and generate a *behavioural equivalence* between processes. Intuitively

$$P \approx Q \tag{1}$$

will mean that no user, or computing environment, will be able to distinguish between P and Q using the interactions described in their behavioural descriptions.

However the use of types in ADPI has a serious impact on this general behavioural framework, particularly as these types implicitly represent the limited capabilities that agents have over resources. In other words these types limit the ways in which agents can interact with other agents. Consequently whether or not two agents are deemed equivalent will depend on the current distribution of the capabilities on the resources in the system.

The third and final aim of this book is to address this issue. We demonstrate that the general theory of bisimulations can be adapted to take the presence of types into account. We develop a relativised version of *behavioural equivalence* in which judgements such as (1) can never be made in absolute terms, but relative to a description of current capabilities. Moreover we will show that the proof techniques associated with standard bisimulations, based on *coinduction*, can be adapted to this more general framework, at least in principle.

A secondary aim of the book is didactic. Research into the use of formal calculi to model distributed or interactive systems, or even understanding the application of such calculi, requires detailed knowledge of a variety of mathematical concepts. We hope that the book will provide a good introduction to a range of these concepts, and equip the reader with sufficient understanding and familiarity to enable them to pursue independent research in the area. But we do not start from first principles. We assume the reader is at least familiar with elementary discrete mathematics,

and in particular *structural induction*. It would also be helpful to have a passing acquaintance with *bisimulations*, as in [28]. Chapter 1 reviews the knowledge assumed in these areas. But other than this, the aim is to be self-contained. In particular we give a detailed exposition of the PI-CALCULUS, and an elementary introduction to types and typing systems.

Structure

As already stated, **Chapter 1: Background** recalls elementary notions of *induction* and *coinduction*, which will be used extensively in all subsequent chapters. However the only form of coinduction to be used is that associated with *bisimulations*, the theory of which is also reviewed.

The PI-CALCULUS, or at least our version of it – called API, for ASYNCHRONOUS PI-CALCULUS, is explained in **Chapter 2: The asynchronous PI-CALCULUS**. This exposition starts from first principles, giving a detailed account of both syntactic and semantic concerns. We give two semantic accounts. The first, a so-called *reduction* semantics, may be viewed as a description, at a suitable level of abstraction, of a prototypical implementation of the language, explaining how API processes can be executed. The second gives an *lts* for API, explaining how processes can interact, by communicating along channels, with their peers. As we have already indicated this automatically gives us a bisimulation equivalence between processes, and moreover bisimulation theory supplies a very powerful coinductive proof principle for establishing equivalences.

However perhaps the most important topic in this chapter is a discussion of appropriate behavioural equivalences for process description languages in general. On the basis of some simple criteria we give a definition of a behavioural equivalence between processes, called *reduction barbed congruence*, $\cong$, which has the advantage of being applicable, or at least easily adapted to, most process description languages. We will also argue that it is the most natural semantic equivalence between processes; it relies essentially only on the process description language having a reduction semantics, and therefore is widely applicable. The chapter ends by showing how bisimulation equivalence needs to be adapted so as to coincide with $\cong$, thereby providing a complete coinductive proof principle for this natural behavioural equivalence.

In **Chapter 3: Types in API** we turn our attention to types, using a typed version of our language, TYPED API. By focusing on a straightforward notion of runtime error for API, we first explain the use of type systems, the general structure that the types need to take and the associated typechecking system. We then elaborate on

the kind of technical results one needs to establish about such a framework. These cumulate to the demonstration of:

- *Subject reduction:* the property of being well-typed is preserved under the reduction semantics.
- *Type safety:* well-typed processes do not give rise to runtime errors.

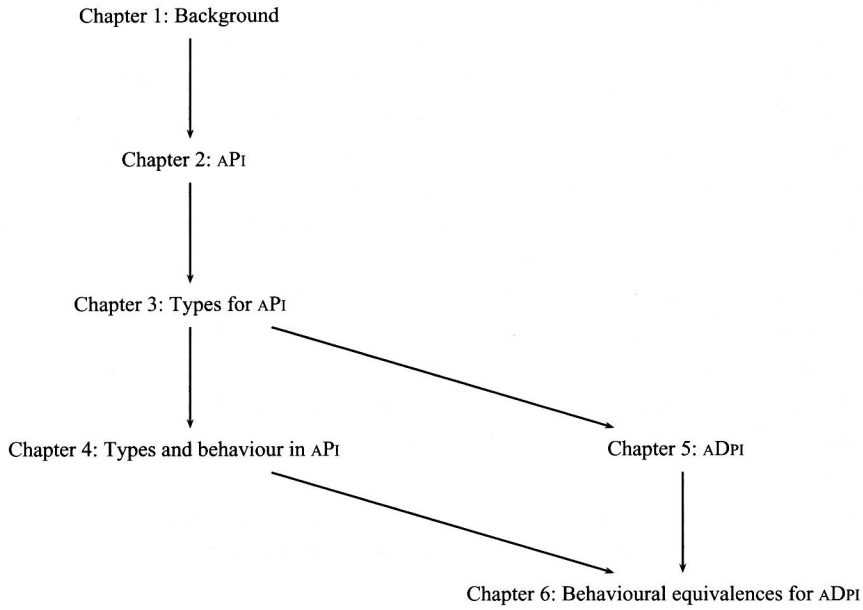
Next we introduce a more sophisticated capability-based type system, in which a type corresponds to a set of capabilities, or permissions, on a channel or *resource*. By means of examples we show how this type system can be used to manage, or control, access to these resources. Establishing the required technical results in this case is somewhat more challenging.

As we have already indicated, the presence of types affects the perceived behaviour of processes, and consequently the behavioural theories of Chapter 2 need to be adapted to TYPED API. This is the topic of **Chapter 4: Types and behaviour in** API, and is quite a challenge. We need to adapt the standard approach for producing an *lts* from a process description language, explained in detailed in Chapter 2, to generate a more descriptive *lts* in which the actions are parameterised by the environment's knowledge of the current capabilities concerned. Most of this chapter is concerned with technical results, which ensure that the resulting *lts* is, in some sense, self-consistent.

We then go on to show how this parameterised *lts* can be used to generate a parameterised version of bisimulation equivalence between processes. In fact this is relatively straightforward, but we also need to prove that the resulting parameterised equivalence satisfies a range of mathematical properties, which one would intuitively expect from such a framework.

At last, in **Chapter 5: A distributed asynchronous** PI-CALCULUS we give a detailed account of our distributed version of API, called ADPI. The syntax is obtained by adding a new syntactic category, for systems, to that of TYPED API. We obtain a description language in which systems consist of a collection of *domains*, hosting agents, which can autonomously migrate between domains. These domains also host local channels on which agents communicate; but more generally these may be used to model *local resources*, accessible to agents currently located at the host domain. The capability-based type system from Chapter 3 is also extended. One novelty is the use of record types for domains, but the fact that resources are purely local also requires a significant extension to the set of types. Essentially the name of a local resource is useless, without knowledge of its location; for this reason local resources are typed using a primitive kind of *existential* channel type.

We demonstrate the usefulness of the typing system via a sequence of examples, and of course we also prove that it satisfies the standard properties one would expect of any reasonable typing system. The chapter then ends with a detailed



formal account of the role of types in ADPI. This involves defining a version of ADPI, called TAGGED-ADPI, in which capabilities, rather than values, are the main entities manipulated by agents. This is a rather complicated language as the capabilities, and their propagation, have to be explicitly managed by the reduction semantics. However we demonstrate that in reality, there is no need for such a detailed language; in the presence of the capability-based type system ADPI can be considered as a more abstract, and therefore more manageable, version of TAGGED-ADPI.

In **Chapter 6: Behavioural equivalences for ADPI** we adapt the parameterised theory of bisimulation equivalence from Chapter 4 to ADPI. In fact we spare the reader most of the technical proofs, as they can easily be adapted from those for API. Instead we show, via a number of examples, that at least in theory many of the standard methodologies associated with bisimulation equivalence can be adapted to prove equivalences between ADPI systems.

We also revisit the principal topic of Chapter 2. We justify our parameterised version of bisimulation equivalence by showing that it coincides with a natural intuitively defined behavioural equivalence, a typed version of reduction barbed congruence.

We end with a brief section, entitled **Sources**, with references to the original research papers on which our material is based, together with some pointers to related work.

The book is essentially divided into two parts. After reviewing some required background material in Chapter 2, the following three chapters aim to give a coherent and detailed introduction to the PI-CALCULUS. These chapters could form the basis of a postgraduate course on the PI-CALCULUS. Although they pursue a particular viewpoint, students completing them should be competent in the mathematical techniques underlying the theory of the PI-CALCULUS, and therefore should have no problem absorbing supplementary material, taken for example from [39], [29], or the research literature.

The second part of the book, Chapter 5 and Chapter 6, give a detailed account of the language ADPI, and its behavioural theory. This could form the core of a postgraduate course on *process calculi for distributed systems*, for students familiar with the PI-CALCULUS. However once more this would need to be augmented with additional material from the research literature, to reflect the varied approaches to the subject.

Alternatively, a less theoretical course could be based upon the first four sections of Chapter 2, Chapter 3 and the first three sections of Chapter 5. This foregoes most of the material on behavioural equivalences, concentrating on the basic semantics of the languages (the *reduction semantics*), and typing.

Each chapter ends with a set of exercises, which the reader is encouraged to answer. For the most part these are related directly to the material of the chapter, perhaps extending it in certain directions, or illustrating certain consequences. By and large they should present no difficulties, although a few may be non-trivial.

Acknowledgements

Most of the research reported on here was carried out over the years with a large number of colleagues. The main language of the book, ADPI, was originally developed in conjunction with James Riely, while the behavioural theory, in Chapter 4 and Chapter 6, was developed jointly with Julian Rathke. Other colleagues whose contributions and collaborative efforts I wish to acknowledge include Alberto Ciaffaglione, Adrian Francalanza, Samuel Hym, Massimo Merro and Nobuko Yoshida.

The book was largely written while the author held a Royal Society/Leverhulme Trust Senior Fellowship during the academic year 2005/2006, although some preliminary material on which it was based was presented at the *Bertinoro International Spring School for Graduate Studies in Computer Science* in March 2004.

Finally a number of colleagues read parts of the book in draft form, and made numerous useful suggestions for improvements. These include Adrian Francalanza, Samuel Hym, Sergio Maffeis, Julian Rathke, James Riely and Nobuko Yoshida.

Contents

Preface ix
Acknowledgements xvii

1 Inductive principles 1
1.1 Induction 1
1.2 Coinduction 4
1.3 Bisimulation equivalence 6

2 The asynchronous PI-CALCULUS 10
2.1 The language APi 10
2.2 Reduction semantics for APi 16
2.3 An action semantics for APi 27
2.4 A coinductive behavioural equivalence for APi 34
2.5 Contextual equivalences 37
2.6 An observational lts for APi 43
2.7 Justifying bisimulation equivalence contextually 47
2.8 Questions 52

3 Types for API 55
3.1 Runtime errors 55
3.2 Typechecking with simple types 60
3.3 Properties of typechecking 65
3.4 Types as capabilities 72
3.5 Questions 93

4 Types and behaviour in APi 96
4.1 Actions-in-context for APi 98
4.2 Typed bisimulation equivalence 112
4.3 Questions 122

5 A distributed asynchronous PI-CALCULUS 124
5.1 The language ADPI 127
5.2 Access control types for ADPI 139
5.3 Subject reduction for ADPI 159
5.4 Type safety for ADPI 169
5.5 Distributed consistency of local channels 185
5.6 Questions 191

6 Behavioural equivalences for ADPI 194
6.1 Actions-in-context for ADPI 195
6.2 Typed bisimulation equivalence for ADPI 200
6.3 Describing bisimulations 202
6.4 Servers and clients 215
6.5 Modelling a firewall 221
6.6 Typed contextual equivalences 226
6.7 Justifying bisimulation equivalence contextually in ADPI 230
6.8 Questions 242

Sources 244
List of figures 248
Notation 250
Bibliography 254
Index 257

1

Inductive principles

Throughout the book we will make extensive use of both induction and coinduction, and their associated proof techniques. Here we give a brief review of these concepts, and an indication of how we intend to use them.

1.1 Induction

Figure 1.1 contains a definition of the (abstract) syntax of a simple language of machines. Here a ranges over some set of action labels Act, and intuitively a machine can carry out sequences of these actions, and periodically has a choice of which actions to perform. Let $\mathcal{M}$ be the set of all machines defined in Figure 1.1. Formally this is an *inductive* definition of a set, namely the least set S that satisfies

- $\mathsf{stop} \in S$
- $M \in S$ implies $a.M \in S$ for every action label a in Act
- $M_1, M_2 \in S$ implies $M_1 + M_2 \in S$.

The fact that $\mathcal{M}$ is the least set that satisfies these conditions gives us a proof technique for defining and proving properties of machines in $\mathcal{M}$; any other set satisfying the conditions is guaranteed to contain $\mathcal{M}$.

As an example consider the following definition of the *size* of a machine:

- $|\,\mathsf{stop}\,| = 0$
- $|a.M| = 1 + |M|$
- $|M_1 + M_2| = |M_1| + |M_2|$.

We know by induction that this function is now defined for every machine. Belabouring the point for emphasis let D be the domain of the size function $|\ \ |$, the set of elements for which it is defined. The three clauses in the definition of $|\ |$ above imply that D satisfies the three defining properties of $\mathcal{M}$. So we can conclude that $\mathcal{M} \subseteq D$; that is every machine is in the domain of $|\ \ |$. We refer to