Antti Puhakka

Weakest Congruences, Fairness and Compositional Process-Algebraic Verification

Tampere 2004

Note: the layout in this version is different from the printed version
Antti Puhakka

Weakest Congruences, Fairness and Compositional Process-Algebraic Verification

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotie Building, Auditorium TB109, at Tampere University of Technology, on the 27th of May 2004, at 12 o'clock noon.

Tampere 2004
Abstract

Traditionally, computer programs have been thought of as entities which take some input, execute a sequence of statements that process this input, and then return the result. More recently, concurrent and reactive computer systems have become increasingly important. Concurrency means that the system consists of several independent, interacting components, and reactivity means that the system is capable of continuous and simultaneous communication in several directions. Examples of concurrency and reactivity can be found in operating systems, embedded systems, computer networks and mobile communication.

Designing concurrent and reactive systems is difficult and error-prone. This is because the existence of concurrent components usually enables an enormous number of possible sequences of events in the system, and the complex communication patterns of reactive behaviour can be difficult to specify and analyse. It is hoped that concurrent and reactive systems can be made more reliable if we use precise mathematical models to represent these systems, and develop methods and computer tools for analysing their behaviour through these models.

Process algebra is one approach for modelling concurrent and reactive systems. In the process-algebraic model the system components make progress by executing actions, which are indivisible, atomic units with identity. Communication between concurrent entities is based on executing common actions synchronously. An important advantage offered by this approach is compositionality, which means that different components of a system can be considered in isolation and described by the behaviour that can be observed at their interface. This makes it possible to replace one component with another, simpler and smaller component that behaves in an equivalent way, which reduces the complexity of the overall system model.

However, we must first have a precisely defined notion of equivalent behaviour. A behavioural equivalence must support compositionality, that is, it must be a congruence, and it should not make unnecessary distinctions between systems, that is, it should be as weak as possible. Here we identify
weakest congruences related to the progress properties of systems, namely, to livelocks and deadlocks.

Another issue related to progress is fairness. This means that even though a system is allowed to make individual choices freely, it is not allowed to constantly favour some choices at the expense of others. We investigate how fairness can be used to guarantee progress in a compositional setting.

Sometimes compositionality makes it possible to handle arbitrarily large systems by using invariants that guarantee that the behaviour of a system stays the same when we add any number of certain components. As an example, we present a case study in which invariants are used to show that the external behaviour of a communication protocol is independent of the capacities of the underlying channels.
Preface

I am greatly indebted to my supervisor, Prof. Antti Valmari, for introducing me to the topic of concurrency theory, and for providing me with a unique opportunity to do research in this field, as well as for the many deep and insightful technical discussions we have had over the years. I also want express my thanks to Prof. Rob van Glabbeek and Prof. Juha Kortelainen for using their valuable time to review this thesis and for the useful and encouraging comments they have provided.

The research described in this thesis has been financially supported by the Tampere Graduate School in Information Science and Engineering (TISE) and the Academy of Finland, and this support is gratefully acknowledged.

The work leading to this thesis was carried out in the Verification Algorithm Research Group (VARG) at the Institute of Software Systems (formerly Software Systems Laboratory) of Tampere University of Technology. I would like to thank the many people who have been working with me in the VARG group. Special thanks to Jaco Geldenhuys for proof-reading parts of this thesis. I wish to thank the head of the Institute, Prof. Tommi Mikkonen, for his support during the preparation of this thesis. Furthermore, I wish to extend my thanks to all the people at the Institute. During my work I have greatly enjoyed the informal and encouraging atmosphere and the sense of community for which the Institute of Software Systems is known. Over the years I have had the privilege to get to know many of my co-workers well, and some have indeed become close friends, and they have remembered me even in the most difficult moments. My sincere thanks to you.

Finally, I want to thank all of my family and friends for their support of many kinds during this work, for providing me, from time to time, with a resting place from my work, and for their persistent optimism which has certainly been needed.
Contents

Abstract i
Preface iii
Contents v
List of Included Publications vii

1 Introduction 1
  1.1 Concurrent and Reactive Systems .................. 1
  1.2 Why Formal Models Are Needed .................... 3

2 Examples of Concurrent and Reactive Systems 5
  2.1 Parallel Processing ................................ 5
  2.2 Embedded Systems ................................ 6
  2.3 Operating Systems ................................ 7
  2.4 Digital Circuits ................................... 7
  2.5 Software Architecture .............................. 8
  2.6 Communication Protocols ........................... 10

3 Formal Verification of Concurrent Systems 13
  3.1 Models of Interprocess Communication ............. 13
  3.2 Model Checking and Temporal Logic ................ 15
  3.3 Petri Nets .......................................... 17
  3.4 Theorem Proving .................................... 19
  3.5 Action Systems ..................................... 20
  3.6 Finite-State Machines with Unbounded FIFO-Queues 21
  3.7 Process Algebras .................................. 22

4 Processes and Process Composition Operators 25
  4.1 LTSs .................................................. 25
  4.2 Process Operators .................................. 27
5 Behavioural Equivalences
  5.1 Equivalences ............................................. 31
  5.2 Congruences ............................................. 32
  5.3 Isomorphism and Tree-Equivalence ......................... 35
  5.4 Strong Bisimilarity ..................................... 37
  5.5 Weak Bisimilarity ....................................... 38
  5.6 Denotational Semantics .................................. 41
  5.7 Deadlocks and Stable Failures ............................. 43
  5.8 Divergences ............................................. 45
  5.9 Notes on Equivalences .................................... 47

6 Application to Protocol Verification ......................... 49
  6.1 Communication Protocols and the OSI
    Model ..................................................... 49
  6.2 Unreliable Channels ..................................... 51
  6.3 The Alternating Bit Protocol ............................... 53
  6.4 The Self-Synchronising Alternating Bit Protocol ......... 57

7 Conclusions ................................................. 59
  7.1 Introduction to the Included Publications ................. 59
  7.2 Future Work ............................................. 63
  7.3 Summary ................................................ 64

Bibliography .................................................. 65
List of Included Publications


Chapter 1

Introduction

Computer systems have been the subject of a great deal of research for a long time, and much theoretical work has been done in order to increase our understanding of the behaviour of these systems. In particular, models and techniques have been developed which allow us to prove that computer programs are correct, that is, satisfy their specification. Most of the early work concentrated on sequential computer programs. Later, concurrent and reactive computer systems have become more and more important, and these systems have become the focus of much of the theoretical work.

In this chapter we take a brief look at concurrent and reactive systems, and at how they differ from sequential programs. We also look at the problems that we face when building and modelling such systems.

1.1 Concurrent and Reactive Systems

A concurrent system consists of distinct, simultaneously active entities. The entities co-operate to accomplish their collective task, and therefore, they need to communicate with one another. Often, the entities have also some autonomy, so that between the communications they act independently of the other components.

It follows that in a concurrent system there are simultaneously several independent program states, for example, several programs counters. Therefore, a concurrent system differs from a sequential program, which has only one program counter, in other words, one thread of program execution.

It should be noted that we often view a sequential program as a composition of components, such as modules or objects. However, the total system has only one program counter, and the point of program execution moves from one component to another, as determined by the calling structure of
the procedures. In contrast, the different components in a concurrent system
can execute simultaneously.

On the other hand, we should also observe that even in a concurrent sys-
tem the different program threads do not necessarily run physically at the
same time. For example, a concurrent system may consist of processes that
are running on a single-processor computer. Since there is only one pro-
cessor available, the operating system has to interleave the execution of the
processes. For this purpose the operating system has a scheduler that gives
the processes execution turns, one at a time, according to some scheduling
policy. In reality, at any given moment only one process is running.

However, because there are several program counters, there can be two
or more processes simultaneously enabled for execution, and therefore the
scheduler has to make a choice among several candidates. Depending on the
scheduling policy this choice can be based on the times the processes have
spent in the queue waiting for their turn, process priorities, and random
selection.

Without knowing every detail about the scheduler and the current state of
the system, we cannot predict which process will be running next. Including
such information in our model of the system could make the model excessively
complicated. In any case, an outside observer that is communicating with
the processes does not have such knowledge. Instead, the observer sees that
the processes make progress but possibly in an unpredictable order. Thus,
it is sensible to consider the system of processes as concurrent rather than
sequential.

For this reason, a distinction is sometimes made between concurrent and
parallel systems. The former refers to communication between independent
program threads, and the latter means simultaneous execution of instructions
in hardware. Some systems are both of these at the same time. However, as
described above, a system may be concurrent even though it is not parallel,
and a system can also be parallel with little or no synchronisation between
the program threads.

A feature of systems often closely associated with concurrency is reac-
tiveness. Traditionally, programs are considered to be entities that first take
some input, then perform processing on the input, and finally return the
result of the processing. Therefore, a program is considered to implement
a function from some input domain to some output domain. This view has
been adopted, for example, in the theory of computability and computational
complexity.

However, reactive systems cannot be described simply as a mapping from
an input to an output. Instead, reactive systems are characterised by contin-
uous, never-ending interaction with their environment, and a reactive system
has to be constantly ready to accept stimuli arriving from two or more different directions and respond to these stimuli.

There is a close connection between concurrency and reactivity. On the one hand, the different components of a concurrent system may communicate independently with their environment, and this makes the behaviour of the total system reactive. On the other hand, implementing a system that is capable of simultaneous, reactive actions often requires using several components that act simultaneously. Furthermore, in many cases the different functions of a reactive system, such as an embedded control system or a web server, are physically distributed, which naturally introduces concurrency.

1.2 Why Formal Models Are Needed

Because the different subprocesses in a concurrent system may make progress at unknown relative speeds, there are potentially a very large number of different sequences of events that are possible, and a large number of states of the composite system that can potentially be reached. This is sometimes called the state-explosion problem. It is very difficult for human beings to think of all the possible ways that events in the system can take place. Rather, people tend to think in terms of sequential scenarios, where everything progresses in some pre-planned order. In this way one can fail to notice possible orders of events that have fatal consequences for the behaviour of the system.

One of the greatest problems in validating concurrent systems is that these systems are inherently nondeterministic. The interactions that take place in a concurrent system can depend on the relative speeds of the concurrently running processes, and this may change from one execution of the system to another. Therefore, even if we run the system with the same initial parameter values and give the same inputs during the execution, the outcome of the execution may be different.

A consequence of these problems is that it is very difficult to validate concurrent systems by testing. Firstly, when the number of potential execution orders in the system is very large, even thorough testing covers only a part of the executions. Typically, the system will be in real use for a much longer period of time, so it is likely that a far greater proportion of the potential executions are realised during the lifetime of the system. Therefore, it is possible that some of the executions that were not covered during testing cause fatal errors during operation.

Another problem is that if we encounter a fault during a test which, say, crashes the system, we would obviously like to be able to recreate the fault
in order to analyse more closely the events that caused the fault. However, with a nondeterministic system this is not necessarily possible, because the fault may not occur in the following test runs even if we give exactly the same inputs to the system.

What is perhaps even worse is that if we find an error in the system and make some modification in order to correct it, we cannot be sure that the problem has been fixed just by running the test case again. The behaviour of the system may now appear to be correct simply because the modification we have made has changed the delicate timing properties in such a way that the error does not show itself in the following test runs.

A problem with reactive systems is that it may be difficult even to specify what the correct behaviour of the system should be. As described above, the system does not just map an input to an output, but different inputs can arrive at the system and responses given in a number of different orders. Furthermore, the system has no end state where success or failure can be determined, but the interaction with the environment can continue indefinitely.

As an alternative or, as a complement, to testing, we can formulate mathematical models of concurrent systems, and design techniques that can be used to investigate the properties of these models. In this way we may be in a better position to make reliable conclusions about the behaviour of concurrent systems. Since computers are able to systematically explore a far greater number of different possibilities than human beings, we can use computers to run verification algorithms based on these techniques, and explore more complex system behaviours. In this way we are able to combine the benefits of human thinking and the raw processing power of computers to improve the reliability of concurrent and reactive system designs.
Chapter 2

Examples of Concurrent and Reactive Systems

There are many reasons why real systems exhibit concurrent and reactive behaviour. Here we take a brief look at some important examples of concurrent and reactive systems.

2.1 Parallel Processing

One advantage offered by concurrency is the possibility of obtaining more processing power, because several parallel processing units can perform more operations in a time unit than a single processing unit. Thus, we can try to solve laborious computational problems faster by dividing the processing burden over several processors.

In practice, this idea can be applied in different ways. At a higher level, a computer can have several processors, which either access the same system memory, or communicate with each other by using other mechanisms. Also, several computers can be connected together into a computer cluster, connected by a high-speed data network.

At a lower level, individual processors usually contain a varying number of parallel execution units for arithmetic operations, memory access, branching logic, and so on. The scheduling of instructions (operations) to the parallel units may be performed on the fly by a dedicated hardware unit, as in so-called “superscalar” architectures, or the units of parallel execution may be determined beforehand by the compiler and be part of the instruction stream, as in VLIW (Very Large Instruction Word) architectures.

In practice, computers with high processing power are needed for numerical simulations in many natural sciences. One of the earliest and most
important application areas is computational physics, which includes many-body problems, quantum field theory, fluid dynamics, and problems of technical mechanics. Many applications can also be found in chemistry, biology, cosmology and meteorology. High processing power is also needed for optimisation problems in many different fields. Yet another important application area is image processing and other types of signal processing.

However, it is important to notice that whether, and how much, benefit the use of multiple processing units can bring depends crucially on the problem that we are trying to solve. Some problems can be easily divided into independent subproblems that can be distributed over several processors, while for other problems this is difficult. It should also be noted that a restricting factor in distributed computation is organising the communication between the processing units, because the effort required for communication can reduce, and even exceed, the benefit obtained from having more processing power available.

An example of a problem that can be easily parallelised is an image processing task such as median filtering where the resulting pixel values depend only on the original values of the nearby pixels. Even for some problems, like a discrete cosine transform, where this does not hold, it is possible to find a reasonably parallelised order of computing. On the other hand, an example of an algorithm that is difficult to parallelise is a depth-first search in a graph, because the order in which states are encountered during the search is crucial.

## 2.2 Embedded Systems

One of the motivations for studying concurrent and reactive systems is the growing importance of *embedded computer systems*. These are computers that have been integrated into a mechanical or electrical system in order to help monitor and control these systems. Embedded computer systems have a wide variety of applications. Examples include process control systems in industry, cellular phones, cash register systems, control relays in power distribution as well as modern everyday consumer products such as washing machines, microwave ovens, and video and audio systems. An important application area is also transportation systems like aeroplanes, trains, ships and cars.

In many cases the concurrency in an embedded control system reflects the physical functions of the controlled system, because these are distributed and occur simultaneously. For example, an aeroplane may have sensors for measuring the speed of the air flowing past the aeroplane, the amount of
fuel in the fuel tanks and the temperature of the engine. On the other hand, dedicated actuators control the flaps, ailerons and landing gear of the airplane. All these components are connected to each other and to a central computer through a network.

In embedded systems, fault-tolerance is often an important issue. This means that the system should be able to withstand limited failure in some part or parts of the system without critical failure of the overall function. Distribution and isolation of components can be used to increase fault tolerance, because a malfunction in one component does not necessarily cause failure in other components. A critical component can also be replicated to improve fault-tolerance.

### 2.3 Operating Systems

Historically, one of the first initiatives for studying concurrent phenomena was created by the development of operating systems. Even in a single-user operating system concurrency is more or less unavoidable for the same reason as in embedded systems, namely because the operating system has to control several physically independent devices that operate simultaneously. Examples of devices controlled by an operating system could be a hard disk, display card, sound card and network card, as well as parallel ports which connect a keyboard, mouse and a printer. The devices send independently requests for services to the operating system. Typically, the operating system also controls queues of requests from programs for transferring data to and from the devices. These requests can arrive simultaneously from different sources and be partly overlapping. In practice, an operating system contains a number of device drivers, each controlling a particular device, and these are implemented as system processes or interrupt routines.

A multi-user operating system also interleaves the execution of user processes, and handles the execution of periodic tasks. Often the operating system provides mechanisms for communication between user processes, such as signalling, semaphore operations, or character queues.

### 2.4 Digital Circuits

Techniques that have been developed for the verification of reactive and concurrent systems can be applied also at the lowest level in digital information processing, namely, to digital circuits. A synchronous circuit can be described in terms of a basic model which has a number of input and output
lines, and which contains a register and logic, as shown in Figure 2.1. The operation of the circuit is controlled by a clock signal. The register stores the values at its input ports, and presents these values at its output ports during the next clock cycle (in reality, of course, the system may have several physical registers, but the combination of these can be considered as one large register). The system inputs and the register outputs are the input to the logic, which is a combination of memoryless logical ports. Finally, the outputs of the system of logical ports form the system output lines and inputs to the register.

We can take the values of the register outputs and the system inputs as the variables defining the system state. At each clock cycle the system changes its state as dictated by the logic. We notice that the operation of a synchronous circuit is entirely deterministic, that is, from each given state a unique new system state follows, and synchronous circuits are in this way different from most concurrent systems. In an asynchronous circuit there are more degrees of freedom, since the operations of the components are not dictated by a single clock, and synchronisation can be controlled at the local level between individual components. Model checking with the help of BDDs (binary decision diagrams) has been perhaps the most frequently used technique for verifying hardware circuits; see Section 3.2.

2.5 Software Architecture

One less obvious source of concurrency can be found in software design, in the methods and notations for specifying software architecture.
An example is the structured analysis (SA) approach, where a central notion are data flow diagrams (DFDs). The basic elements in these diagrams are processes (or transformations), drawn as “bubbles”, which represent the functional units in the system, data stores, which represent data storage and buffering, and data flows, drawn as arrows in the diagram, which represent the flow of information from one process to another and between processes and data stores. An example is shown in Figure 2.2. A process can hierarchically consist of a diagram of subprocesses, and so on.

In essence, the processes of a DFD can be viewed as concurrent entities that communicate with each other. Since DFDs have been regarded by many as a useful and easily understandable design approach, this would seem to suggest that concurrency is a natural way for humans to perceive a complex system. Another example could be the object-oriented approach, which has become perhaps the most important design paradigm. Here, a system is designed through a hierarchy of objects. The objects are entities which exist simultaneously and interact with each other. Again, this could be seen as a representation of concurrent behaviour.

In practice, however, systems that have been designed using data flow diagrams or the object-oriented approach are usually implemented in a way that contains concurrency only where this is unavoidable. Typically, data flow diagrams are sequentialised into a hierarchy of function calls, and objects call the methods (procedures) of other objects in a sequential manner (see Section 3.5 for an alternative object-oriented approach that supports concurrency). This is mainly because of the practical difficulties involved in developing concurrent software. In his 1982 book *Controlling Software Projects* [14] Tom DeMarco, one of the first software engineers to suggest the SA approach,
writes (notice that the terms “synchronous” and “asynchronous” refer to
what we call “sequential” and “concurrent”, respectively):

While an asynchronous implementation would be in many ways
the most natural one (that is, most closely related to the un-
derlying perception of the requirement), present-day technology
discourages it. Most developers have such a healthy respect for
the difficulty of building asynchronous systems that they would
never introduce asynchronism into any application that could be
built without it. I believe that this practice will change drastically
during the 1980s: By the end of the decade, we’ll be routinely
building systems with a maximum of asynchronism, rather than
minimum. But for the present, most systems are designed to run
as a single task, so the design activity entails creating a model of
a synchronous implementation.

Needless to say, the prediction in the above quotation turned out to be
hopelessly over-optimistic, and the design of concurrent systems has proven
a subtle problem that still remains largely unsolved.

2.6 Communication Protocols

One of the most promising uses for a mathematical theory of concurrency
can be found in solving problems related to the design of communication
networks. Communication networks are, of course, needed in the applica-
tion areas discussed above, especially in parallel processing and embedded
control systems. However, there are also other reasons why communication
networks have become vitally important. Nowadays most companies depend
on computer systems for controlling the storage and distribution of goods,
as well as for invoicing and accounting, and the local systems at different
towns or countries must be able to communicate with each other. Commu-
nication networks such as the Internet can also be used for presenting and
selling products and services to the customers. Computer networks are also
used for public administration, for everyday communication, as a forum for
exchanging opinions, and for many other purposes.

Digital communication is also needed in telephone networks for trans-
mitting telephone calls and other data. An especially important application
can be found in mobile phones and other wireless devices, which require both
communication over a radio signal between a basestation and nearby wireless
devices, as well as transmitting data in the fixed network that connects the
basestations to each other and to other networks.
What is common to all forms of communication over a network is the need to agree on conventions on how the interaction between the participants takes place. There are a great many issues that need to be solved before such communication can work correctly. For example, we have to define when each party can send and receive data, how data items are distinguished from each other in the communication stream, and how data items should be routed through the nodes of the communication network. The rules of communication should guarantee that the participants never become blocked by others, and they should enable recovery from the possible loss, corruption and duplication of data in the network, and other abnormal situations.

Since we cannot expect there to be human beings present to supervise the interaction between the communicating parties, intuitive “common sense” rules are not sufficient for solving these problems. Instead, we need to find simple, straightforward and unambiguous rules that can be encoded as an algorithm, and the rules should, ideally, guarantee correct operation when followed by each participant.

A collection of rules governing the actions of the communicating parties is called a protocol. In its widest sense, the concept of a protocol is very broad. For example, the definition of a file format for exchanging information could be classified as a protocol. However, the types of protocols for which a mathematical theory of concurrency can bring the most benefit are those that involve communication in two or more directions, and where simultaneous, independent actions by the participants are possible. When this is true there is usually a very large number of possible sequences of events that can take place, and it is therefore very difficult for humans to understand all aspects of the system behaviour and define correct rules for the interaction.

There will be further discussion of communication protocols, especially protocol layers and the OSI (Open Systems Interconnection) model, in Chapter 6.
Chapter 3

Formal Verification of Concurrent Systems

A formal approach to the design of concurrent systems requires a mathematical model that represents these systems. We also need a theory which allows us to reason about the behaviour of the model, and this includes the definition of what aspects of behaviour should be taken into consideration. We can then identify techniques that can be used to verify the presence or absence of such behaviours.

Often the conclusion we are seeking is that the behaviour of the model fulfills some specification of correct behaviour. Sometimes, we want to check the observed behaviour of the model against an informal specification. At other times, we transform the informal specification into a formal specification, which can be either a model of the same type as the system model, or expressed in some other mathematical formalism. In either case, we must also have a formal definition of what it means to satisfy a specification.

Ideally, we could maintain a distinction between the formal models of systems and the techniques that are used to reason about the models, but in practice some techniques are tied to specific models, and therefore we cannot entirely separate the description of these two topics.

In the rest of this chapter we give a brief overview of some influential models, theories and verification techniques of concurrent systems. We begin by considering some general questions in modelling concurrency.

3.1 Models of Interprocess Communication

One of the first requirements for a concurrent system is a mechanism that the concurrent entities can use to communicate with each other. In real
systems interprocess communication can take many forms. In a single computer different processes or program threads may communicate by writing to and reading from common memory locations in the system memory. On the other hand, communication between programs on two different computers ultimately requires communication over a physical communication link. In most cases this also requires various other phases where data is transferred between the application program and various parts of the operating system which manage the communication protocols and the communication hardware.

However, it is often useful to look at the situation at a higher level of abstraction, because giving consideration to everything that happens during interprocess communication would obscure our understanding of concurrent behaviour by unnecessary technical detail. We often find that technically different communication mechanisms produce similar patterns of concurrent behaviour.

In order to perform their tasks, concurrent processes need to exchange information with each other. In addition, the processes need to synchronise their actions, so that, for example, one process does not proceed until some information has been provided to it by another process. A special case of such synchronisation is mutual exclusion, which means that only one process at a time is allowed to perform some operation. For example, only one process may be allowed to perform a sequence of critical operations on a common variable, in order to avoid a confused state of the system.

In concurrent programming languages several mechanisms have been proposed for interprocess communication. One primitive mechanism is semaphores, which can be used for synchronisation, while common variables can be used for exchanging information. In conditional critical regions and monitors access to a group of common variables is protected by automatic mutual exclusion. In the former, other forms of synchronisation are implemented by evaluating boolean guards, and in the latter by interprocess signalling. Rendezvous communication and remote procedure calls (RPC) are based on calling a procedure in another process. In the call, parameters can be passed to the procedure and return values transmitted back. In rendezvous communication, only one procedure call is allowed at a time, while in RPC, several instances of the procedure can be invoked to handle simultaneous calls. Also, in the latter, communication can take place between programs on different computers. In some languages and operating systems communication is based on sending messages through first-in-first-out (FIFO) queues. In this case, both synchronisation and transmitting information are handled by the same mechanism. [3]

When developing a general model for concurrency and communication,
it is probably wise to try to find a simple and general communication mechanism. A useful way to characterise communication mechanisms is distinguishing between synchronous and asynchronous modes of communication.

In synchronous communication, the communicating parties have to commit themselves simultaneously to the communication action. If one party wants to communicate with another party, it has to wait until that other party is also willing to communicate. Only after the communication is over, the parties can continue with other tasks.

In asynchronous communication the communicating parties do not have to act simultaneously. One party can leave a message for the receiving party, and then continue with some other tasks. The receiving party can read the message when it is ready to do so.

Common variables are one form of asynchronous communication, because one process can write a value into the variable and continue with other tasks without waiting for the other process to read from the variable. A more general version of this type of communication is the FIFO message queue mentioned above. When the receiving process is ready, it can read the messages from the queue in the same order they were sent.

Finally, there is another way we can try to classify formal models of concurrency. A concurrent system, or any computer system, can be seen as having a state, and it changes its state by performing actions. Thus, it is possible to base a formal model either on the states or on the actions of the system or, to some degree, both.

### 3.2 Model Checking and Temporal Logic

In the classical model checking approach [12, 13, 39], the system model consists of a collection of states, and (invisible) moves by which it is possible to change the state. Typically, the properties of states are described by a collection of propositions, that is, assertions, that are either true or false in each state. This model is called a Kripke structure. In some cases, the model is considered to be the result of the asynchronous parallel composition of programs which communicate through common variables. Then each state consists of the values of the program variables, possibly including the program counter of each subprogram, and the assertions can be predicates on the values of the variables.

The correct behaviour of a system is specified by using a temporal logic which describes the temporal ordering of states. Temporal logic is distinguished from ordinary propositional logic by the use of modal temporal operators, such as “always”, denoted by “□” or “G”, and “eventually”, denoted...
Figure 3.1: An example Kripke structure, where the initial state is denoted by a small arrow, and propositions that hold are shown inside each state by “◊” or “□”. The purpose of model checking is to decide whether the Kripke model satisfies the temporal logic specification.

One of the most well-known temporal logics is linear temporal logic (LTL) [29, 36] which expresses properties which must hold of every (infinite) execution path in the Kripke model. As an example, if we have proposition request which states that a request for access to some resource is issued, and grant which states that access is granted, then we could make the requirement “always if a request is made, then eventually access is granted”. In LTL, this can be expressed with □(request ⇒ ◊grant) or, equivalently, G(request ⇒ Fgrant). For example, the simple Kripke structure in Figure 3.1 does not satisfy this property, because it is possible to loop forever in the topmost state at right. This is an example of a liveness property, because it states a requirement of the form “something good must eventually happen”.

In general, any property can be seen as a combination of a liveness property and a safety property, which states a requirement of the form “nothing bad must ever happen”. An example of the latter could be “no new request is made until the previous one has been served”.

Another class of temporal logics are branching time logics, which express not only properties of linear executions, but also, at every point during the execution, preserve the possible futures of the system. These logics are characterised by the presence of path operators, most commonly “for every execution path” (starting at a given state), denoted “A”, and “for some execution path”, denoted “E”. One of the most well-known of these logics is CTL. With this logic it is possible to express the requirement “always if a
request is made, then a sequence of events is possible where access is finally granted”. This is expressed with \( \text{AG}(\text{request} \Rightarrow \text{EF} \text{grant}) \). The example system in Figure 3.1 satisfies this property.

The logics CTL and LTL are incomparable in expressive power. However, the more general branching time logic CTL*, which allows arbitrary combination of quantifiers, contains both CTL and LTL as special cases.

The most important practical problem in applying model checking is the state-explosion problem. Often the state-space grows so large that it is impossible to construct it. Several techniques have been developed to alleviate this problem. One such technique is binary decision diagrams (BDDs), which are a potentially more compact representation for state sets and state-transition relations. In some cases BDDs allow handling a large number of system transitions in one step. In practice, BDDs have been frequently used in model checking of digital hardware circuits. E. Allen Emerson writes in [16]:

BDD-based model checkers have been remarkably effective and useful for debugging and verification of hardware circuits. For reasons not well understood, BDDs are able to exploit the regularity that is readily apparent even to the human eye in many hardware designs.

It should be noted that the ideas of temporal logic and model checking are not limited to a Kripke structure. They can be applied also in the action based setting of process algebras. An example of such application is in the enclosed publication [IV], where temporal logic is used to express fairness constraints for a process-algebraic system. A fairness constraint means that even though the system is allowed to make nondeterministic choices, we do not allow it to endlessly favour some choices at the expense of others.

### 3.3 Petri Nets

Petri nets [40, 41] are a class of automata-like systems that have been extensively studied over several decades. In the basic setting, a Petri net (or, a place/transition net) consists of a finite number of places and transitions. The places are intended as holding places for tokens. The places are connected to transitions, and transitions to places, by arcs. A simple example net from [17], representing a process control system, is illustrated in Figure 3.2.

Initially, there are some tokens situated in the places of the net. This is called the initial marking. The system changes its state when the transitions fire. When a transition fires, tokens are removed from the places which are
Figure 3.2: An example Petri net with initial marking

connected to this transition through arcs, and new tokens appear to the places to which this transition is connected. In our example system, the transition SendMessage is initially enabled, because there is a token in the only place, working, that is connected to it. When SendMessage fires, the token disappears from the place working, and one token appears in the place waiting, and another one appears in the place messages. In this way a new marking, or state of the system, has been reached. In the example system, the transition Control can fire next, removing the token from the place messages, and producing one in the place instructions. Now the transition Execute becomes enabled, and firing it returns the system back to the initial marking.

Questions we could ask about the behaviour of a Petri net include whether a particular marking is reachable from the initial marking through the firing of transitions, and whether some reachable marking is dead, meaning that no transitions are enabled in it. An individual transition can be, for example, live, meaning that from every reachable marking it is possible to make it enabled by possibly first firing some other transitions, or dead, meaning that it can never be fired. It is also important to notice that even though the Petri net formalism is not Turing-strong (that is, it cannot model an arbitrary computer), the state-space of a Petri net can be infinite. Therefore, an important question is whether the net is bounded, that is, whether there is some finite limit to the number of tokens that the places can contain. This is true precisely when the state-space is finite.

In addition to various analysis techniques that are specific to Petri nets (such as those based on the structure of the net), for bounded nets it is possible to form a reachability graph, whose nodes are the reachable markings of the net, and whose edges are labeled by the transitions that lead from one
marking to the other. In this way it is possible to apply model checking techniques similar to the ones described above to analyse the behaviour of a Petri net.

A characteristic of Petri nets is that, in contrast to many other approaches that concentrate either on the states or on the actions of a system, Petri nets treat both the places and the transitions as “first class citizens” in the sense that it is possible to express properties concerning both. Also, an appealing feature of Petri nets, especially for end users, is their visual appearance.

As a tool for modelling systems, it could be said that it is easy and natural to model some types of systems as Petri nets, while for some other systems this can be difficult. For example, Petri nets lend themselves naturally to modelling interprocess synchronisation semaphores. On the other hand, there is no straightforward way to model a message buffer as a Petri net. Also, from the fact that transitions in a Petri net represent linear operations on the markings, it follows that firing transitions “backwards” results in a unique marking. This means that it is difficult to model operations that lose information about the state of the system, especially assignment into a variable, for example “$x := 0$”.

The basic Petri net paradigm can be modified in various ways. Perhaps the most important extension are the so-called Coloured Petri nets, which allow including data to the tokens [23].

### 3.4 Theorem Proving

Theorem proving is a general approach that is not especially tied to concurrent systems, and refers to the construction of the proof of a mathematical theorem from a set of axioms and definitions with the help of a computer tool. There are computer tools that can verify the validity of proof steps presented to it, and some tools use advanced heuristic techniques to automatically construct proofs for theorems. In general, the degree of automation that is possible in proving theorems is related to the expressiveness of the underlying logic. Simple logics like propositional logic are decidable but make it difficult to express system properties. On the other hand, more expressive logics, especially higher order logics, allow easier formulation of system properties, but are more difficult to automate. See, e.g., [37, 18, 34, 35].

With theorem proving it is possible to take advantage of symbolic representation of systems. In many cases a correctness proof can be independent of the concrete values of some system parameters, such as the range of a variable, and this may allow verifying systems with arbitrarily large or infinite state-spaces. However, a drawback is that, in practice, constructing
a proof requires a great deal of assistance from the user, and therefore the
user has to have a very detailed understanding of the proof system and of
the behaviour of the model under investigation. It should also be noted that
even with finite-state techniques it is sometimes possible to verify arbitrarily
large state-spaces; an example can be found in the enclosed publication [V].

3.5 Action Systems

The Temporal Logic of Actions (TLA) [28] is an approach for modelling
concurrent systems where both the system implementation and the speci-
fication of its correct behaviour are expressed in same the logic, and the
“implements”-relation between them reduces to ordinary logical implication.
Similarly as in Section 3.2, a system state is assumed to consist of the current
values of system variables; the variables are allowed to have infinite ranges.
The actions of the system are defined as predicates on two copies of the sys-
tem variables, that is, on a pair of states. The idea is that the system may
make a step from one state to another precisely when some action predicate
holds for this pair of states.

A formula of TLA is applied to sequences of states (i.e., executions), and
may contain, in addition to ordinary logical operators, the temporal operator
“□” (always), which has a similar meaning as in LTL (see Section 3.2). If
\( A \) is an action, then \( □A \) holds of a state sequence when \( A \) holds of each
subsequent pair of states in the sequence. A concurrent program can be
defined in the canonical form \( \text{Init} \land □[\mathcal{M}]_V \land F \). Here, \( \text{Init} \) defines the initial
state of the program, \( \mathcal{M} \) is (the disjunct of) the actions of the program, and
the notation \([...]_V \) means that also so called \textit{stuttering actions}, which do not
change the system state, are allowed. Finally, \( F \) is a conjunction of fairness
constraints towards subsets of actions, to ensure progress in the system.

We can consider any predicate on states as an action that only refers
to the first state of the state pair. Thus, any property that a program
should satisfy can be expressed in the same logic. The fact that a program
satisfies a property means that the program logically implies the property.
The property can also be another program, which can be seen as a more
abstract specification.

\textit{DisCo} [4, 22, 30] is an object-oriented specification and verification ap-
proach that has been built on the formal basis of TLA. In DisCo, a system
consists of objects and actions. As in ordinary object-oriented languages, the
state of an object consists of the present values of its variables. However,
here objects have no methods. Instead, the actions provide \textit{roles} for objects,
and when there exist objects that fit into the given roles, and whose states
satisfy the guard of the action, the action is executed jointly by these objects. The execution of an action changes the states of the participating objects as defined in the body of the action.

In DisCo, it is possible to start from an abstract specification of a system, and use stepwise refinement, or superposition, to transform the system towards a more detailed and deterministic implementation. At each refinement step, new object classes and actions can be created, new variables can be introduced to objects, the guards of actions can be strengthened, and statements affecting the new variables can be added to the action bodies. Safety properties are automatically retained in each step, meaning that the new system implies the old system in the underlying TLA interpretation.

3.6 Finite-State Machines with Unbounded FIFO-Queues

One possible model of concurrent systems is a set of finite-state machines which communicate asynchronously by sending messages over unbounded FIFO-queues. Transitions of the machines are labelled with actions which either write to or read from the queues. At any moment, the state of the system consists of the states of the automata and of the contents of the queues. This approach is especially aimed at verifying communication protocols, with the queues an obvious representation for channels.

Automata with unbounded message queues can be used to simulate a Turing machine, and therefore most interesting verification questions for these systems are undecidable. However, it may be possible to represent an infinite set of queue contents by a finite means. One possibility is using finite automata, called Queue-content Decision Diagrams, or QDDs [8, 9]. In case the queues represent lossy channels, it is possible to use a simpler representation, called Simple Regular Expressions, or SREs [1, 2]. With these representations it is possible to use semi-algorithms which search for loops in the system behaviour and, if they terminate, produce a finite representation of the infinite state-space. In this way it is possible to check safety properties of these systems. However, checking liveness properties is more problematic since an unbounded number of messages can be input to an unbounded queue. Also, this model does not seem to lend itself very easily to compositional (piecewise) construction of systems, which is discussed in the next section and in Chapters 4–7.

Examples of the application of these models to communication protocols, some of which are similar to the protocols discussed in Chapter 6, can be
found, for example, in [1] and [8].

3.7 Process Algebras

In process algebras [32, 7, 5, 21, 42] processes change their state by executing actions, and the actions are either visible, representing external behaviour, or invisible, representing internal activity. Communication between processes is based on executing visible actions synchronously, although the details of the synchronisation differ slightly in different process algebras.

In a process language such as CSP [21, 42], CCS [32], ACP [7, 5] or LOTOS [10], processes are defined from primitive processes and actions by using process operators and recursive equations. An alternative approach is to give the operational semantics of processes directly as state-transition diagrams. Processes can be combined by using process operators, most importantly the parallel composition and hiding operators.

The primitive synchronous communication mechanism of process algebra is general enough to simulate other communication mechanisms, such as asynchronous communication through common variables or message queues. For example, we can model a common variable as a process which stores the value of the variable and offers actions for writing values to and reading from the variable. In this way the communication mechanism does not restrict the applicability of the model.

A characteristic of process algebras is that any composition of processes is also a process, and it is possible to construct a system from constituent processes in a number of different ways. Process algebras also place emphasis on the actions of the system rather than on the states. In a sense, the visible actions form the interface to a process. In this way we can concentrate on the externally observable behaviour, and develop semantic equivalences that determine whether or not the external behaviour of two processes is the same. This helps us tackle the state-explosion problem, because we can replace a subprocess in a composition by any equivalent, but possibly simpler and smaller process, thus simplifying the composite system. In order to find automatically simpler equivalent processes we can use reduction algorithms.

The approach described above is called compositional. Another benefit of the compositional approach is that it allows us to “calculate” with processes as mathematical entities using algebraic laws. This allows, among other things, the use of invariants in system composition, and thereby verification of arbitrarily large state-spaces; see Section 7.1 and the enclosed publication [V].

One benefit of the process-algebraic approach is that it allows the use
of visual verification. This means that the user does not have to make beforehand a specification and then check whether or not the system design fulfills it. Instead, the external behaviour of the system design is shown as a graphical diagram which the user can investigate. This is both user-friendly and allows one to find errors that might be difficult to think of during the specification phase. Examples of the use of visual verification can be found, for example, in [26, 27, 45, 48].

It should also be noted that it is possible to add state-based information to process algebras. Similarly as in a Kripke structure, we can have a set of propositions which are true or false in each state. Some of the propositions can have a special significance, such as forbidden state, cut state, and so on. Excluding the special propositions, it is possible to encode the state-based information in the actions of the system, allowing us to retain the action-oriented approach [19].

In this work we use process operators similar to those in CSP and LOTOS, and we model processes as state-transition diagrams. The model is described in more detail in the next chapter.
Chapter 4

Processes and Process Composition Operators

In this chapter we will define the formal model that we use for representing concurrent and reactive systems.

4.1 LTSs

We will from now on consider a process as an entity that has a state, and which changes its state by executing named actions. Therefore, we can represent the behaviour of a process as a labelled directed graph. The nodes (vertices) represent the states of the process and the arcs (edges) represent the possible transitions between the states. The label of each transition is the action that is executed. One of the nodes is a designated initial state where the process starts its execution.

Since we will later want to distinguish between internal (invisible) and external (visible) behaviour of a system, we will designate one action, called \( \tau \), as an invisible action. All other actions are visible actions. We attach to each process a set of visible actions called the alphabet of the process. A process can only execute actions in its alphabet and the \( \tau \)-action. In a sense, the alphabet describes the interface of the process to the external world.

We will call the model described above a labelled transition system, or LTS. An example LTS is shown in Figure 4.1, which is a simple representation of the behaviour of a cash point from the point of view of the customer. The initial state of the LTS is \( s_1 \), and it is denoted by the small arrow in the picture. From the initial state it is possible to execute only the action \textit{InsertCard} and thereby move into state \( s_2 \). From \( s_2 \) it is possible to execute the actions \textit{Withdraw} or \textit{Cancel}. After the former, the system may choose
between two courses of actions, shown as \( \tau \)-actions, which represent the decision to accept or deny the withdrawal request. The alphabet of this LTS is \{\text{InsertCard, Cancel, Withdraw, Grant, Refuse, CardOut}\}. We will return to this example later.

Formally, an LTS is defined as follows.

**Definition 1** A labelled transition system (LTS) is a four-tuple \((S, \Sigma, \Delta, \hat{s})\), where

- \( S \) is the set of states,
- \( \Sigma \) is the alphabet of the process; we assume that \( \tau \notin \Sigma \),
- \( \Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S \) is the set of transitions, and
- \( \hat{s} \in S \) is the initial state.

An LTS is finite if and only if the constituent sets \( S, \Sigma \) and \( \Delta \) are finite.

We will next define some notation that will make it easier to discuss the possible executions in an LTS. The \( s \rightarrow \eta \rightarrow s' \) notation means that from state \( s \) it is possible to execute the finite sequence of actions \( \eta \) and reach state \( s' \). By \( s \rightarrow \eta \rightarrow \) we mean that from \( s \) it is possible to execute the finite or infinite sequence of actions \( \eta \).
Definition 2  Let \((S, \Sigma, \Delta, \delta)\) be an LTS, let \(s, s' \in S\), and \(a, a_1, a_2, a_3, \ldots \in \Sigma \cup \{\tau\}\). We write

- \(s \xrightarrow{a} s'\) if and only if \((s, a, s') \in \Delta\),
- \(s \xrightarrow{a_1 a_2 \cdots a_n} s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0\), \(s_n = s'\) and \(s_{i-1} \xrightarrow{a_i} s_i\) when \(1 \leq i \leq n\),
- \(s \xrightarrow{a_1 a_2 \cdots a_n} \) if and only if there is \(s' \in S\) such that \(s \xrightarrow{a_1 a_2 \cdots a_n} s'\),
- \(s \xrightarrow{a_1 a_2 a_3 \cdots} \) if and only if there are \(s_0, s_1, s_2, \ldots\) such that \(s = s_0\) and \(s_{i-1} \xrightarrow{a_i} s_i\) when \(i \geq 1\).

In the example of Figure 4.1, we see that, among others, the following executions are possible:

- \(s_1 \xrightarrow{\text{InsertCard}} s_2\)
- \(s_1 \xrightarrow{\text{InsertCard Withdraw}} s_3\)
- \(s_3 \xrightarrow{\tau \text{ Refuse CardOut}} s_1\)
- \(s_1 \xrightarrow{\text{InsertCard Cancel CardOut InsertCard Withdraw \tau Grant CardOut}} s_1\)

4.2 Process Operators

Since we are developing a theory of concurrency, individual processes are usually not interesting as the objects of study. Rather, we would like to model the behaviour of a group of concurrent processes that communicate with one another. As described in Section 3.7, the process-algebraic approach is based on a primitive synchronous communication mechanism. Therefore, in our model processes communicate by executing actions synchronously. More precisely, every visible action is simultaneously executed by precisely those processes that have that action in their alphabet, while other processes retain their state during the execution of the action. This means that if even one process that has the action in its alphabet is not ready to execute it, then the action cannot be executed.

The \(\tau\)-actions do not synchronise, so every \(\tau\)-action is executed by exactly one process, while other processes retain their states. Of course, if a visible action is in the alphabet of only one process, it will also be executed only by that process.

Formally, we construct the state set of the parallel composition by using the Cartesian product of the state sets of the parameter processes. However, not all the combinations (tuples) of states may be reachable in the parallel composition. As the unreachable states do not affect the behaviour of the
system, it is technically more convenient if we do not have to later consider the unreachable states. Therefore, we choose to remove unreachable states from the parallel composition. (Not surprisingly, computer tools that implement parallel composition also remove unreachable states.) We will first define the product of LTSs, which has all the state combinations as its state set. The actual parallel composition is then the part of the product LTS that is reachable from the initial state.

**Definition 3** Let \( L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1) \) and \( L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2) \) be LTSs. Their product is the LTS \( \Delta' \) such that the following hold:

- \( S' = S_1 \times S_2 \)
- \( \Sigma = \Sigma_1 \cup \Sigma_2 \)
- \( ((s_1, s_2), a, (s'_1, s'_2)) \in \Delta' \) if and only if either
  - \( a \in (\Sigma_1 \cup \{\tau\}) \setminus \Sigma_2 \) and \((s_1, a, s'_1) \in \Delta_1 \) and \(s'_2 = s_2\), or
  - \( a \in (\Sigma_2 \cup \{\tau\}) \setminus \Sigma_1 \) and \((s_2, a, s'_2) \in \Delta_2 \) and \(s'_1 = s_1\), or
  - \( a \in \Sigma_1 \cap \Sigma_2 \) and \((s_1, a, s'_1) \in \Delta_1 \) and \((s_2, a, s'_2) \in \Delta_2 \)
- \( \hat{s} = (\hat{s}_1, \hat{s}_2) \)

The parallel composition \( L_1 \parallel L_2 \) is the LTS \((S, \Sigma, \Delta, \hat{s})\) such that

- \( S = \{ s \in S' \mid \exists \eta \in (\Sigma \cup \{\tau\})^* : \hat{s} - \eta \rightarrow s \} \)
- \( \Delta = \Delta' \cap (S \times (\Sigma \cup \{\tau\}) \times S) \)

One of the main themes in this work is compositionality, so it is important to notice that the result of parallel composition belongs to the same class as the parameters, that is to say, the parallel composition of LTSs is also an LTS. Therefore, the result can be used as a parameter process for further parallel composition or other operators.

Another mathematical property of parallel composition that supports compositionality is the fact that it is symmetric, so that \( L_1 \parallel L_2 \) is equivalent to \( L_2 \parallel L_1 \), and associative, so that \((L_1 \parallel L_2) \parallel L_3\) is equivalent to \(L_1 \parallel (L_2 \parallel L_3)\). This means, in essence, that it does not matter in what order we compose processes in parallel. However, before formalising this result we must define more precisely what we mean by “equivalent”, because in this case we cannot use identity of LTSs. For example, states of \((L_1 \parallel L_2) \parallel L_3\) are of the form \(((s_1, s_2), s_3)\), but states of \((L_1 \parallel L_2) \parallel L_3\) are of the form \((s_1, (s_2, s_3))\), where \(s_1, s_2\) and \(s_3\) are states of \(L_1, L_2\) and \(L_3\), respectively. However, apart from the
states, the transition structures of the processes are identical. We can capture this property by the notion of isomorphism, which requires that there is a 1-1 mapping (bijection) between the states of the two LTSs which respects the transition structure, and which maps the initial states to each other. Notice that we will only speak of isomorphism or equivalence between processes that have the same alphabet, so that they have, in a sense, the same interface to the external world.

**Definition 4** Let \( L_1 = (S_1, \Sigma, \Delta_1, \hat{s}_1) \) and \( L_2 = (S_2, \Sigma, \Delta_2, \hat{s}_2) \) be LTSs. \( L_1 \) and \( L_2 \) are isomorphic, denoted \( L_1 \cong L_2 \), if and only if there exists a bijection \( f \) from \( S_1 \) to \( S_2 \) such that

- \( f(\hat{s}_1) = \hat{s}_2 \)
- \((s, a, s') \in \Delta_1 \) if and only if \((f(s), a, f(s')) \in \Delta_2 \)

We can now formally state the symmetry and associativity of “\( || \)".

**Proposition 5** Let \( L_1, L_2 \) and \( L_3 \) be LTSs. Then

- \( L_1 || L_2 \cong L_2 || L_1 \)
- \((L_1 || L_2) || L_3 \cong L_1 || (L_2 || L_3) \)

**Proof** The required bijection \( f \) is in the first case \( f(s_1, s_2) = (s_2, s_1) \), and in the second case \( f((s_1, s_2), s_3) = (s_1, (s_2, s_3)) \). Showing that these bijections respect the transitions between states is straightforward but tedious. \( \square \)

Since we will never need to distinguish between isomorphic processes, this means that we can skip parenthesis in expressions like \((L_1 || L_2) || L_3 \), and write simply \( L_1 || L_2 || L_3 \). This generalises easily to any greater number of processes.

Parallel composition allows us to construct concurrent systems, but the resulting systems can be very complex and have a very large number of states and transitions. This makes the analysis of their behaviour and their further use as components difficult. On the other hand, we are usually not interested in all the details of the behaviour, but rather some specific aspects of it, such as the actions that are visible from the outside of the system. We were already anticipating this by including the invisible \( \tau \)-actions in the LTS model.

For this purpose we will next introduce the hiding operator which can be used to convert visible actions into \( \tau \)-actions. A typical application of the hiding operator is to conceal actions that are used for synchronisation
between subprocesses, but which are considered to be internal actions of the composite system and not visible to the external world (it should be noted that in some process algebras, especially CCS, synchronised actions are automatically hidden).

The \(\tau\)-actions cannot be distinguished from each other, and we usually take the view that it is not possible to see from the outside how many \(\tau\)-actions are executed. Therefore, declaring actions as internal usually simplifies the visible behaviour of the system, and may allow us to replace the original, complicated system with another, simpler and smaller system.

Formally, the hiding operator is defined as follows.

**Definition 6** Let \(L = (S, \Sigma, \Delta, \hat{s})\) be an LTS and \(X\) any set of action names. Then hide \(X\) in \(L\) is the LTS \((S, \Sigma', \Delta', \hat{s})\) such that the following holds:

- \(\Sigma' = \Sigma \setminus X\)
- \((s, a, s') \in \Delta'\) if and only if 
  \[a = \tau \land \exists b \in X : (s, b, s') \in \Delta, \text{ or } a \notin X \land (s, a, s') \in \Delta.\]

Notice that in the enclosed publication [V] we use an alternative postfix notation for hiding, \(L \setminus X\), because this makes the manipulation of complex terms easier.

It is possible to define other process operators, such as renaming, choice and interrupt. A version of the renaming operator is used in [V]. Definitions and examples of the use of the other operators can be found, for example, in [10, 42, 47].
Chapter 5

Behavioural Equivalences

5.1 Equivalences

Our objective is to have a notion of equivalence between processes, such that when processes $P$ and $Q$ are equivalent, denoted $P \simeq Q$, then no matter in what environment we place these processes, $P$ will always behave in the same way as $Q$.

Formally, an equivalence is a relation over the set of LTSs. A relation $\asymp$ over a target set $T$ can be defined as a subset of the set of pairs of elements of $T$, namely those pairs that are in the relation: $\asymp \subseteq T \times T$. We often use an infix notation, so that instead of $(P, Q) \in \asymp$ we write $P \asymp Q$.

An equivalence is a relation that has three special properties, each of which is intuitively more or less obvious. Namely, an equivalence is reflexive, so that every process is equivalent with itself, and symmetric, so that if $P$ is equivalent to $Q$, then $Q$ is equivalent to $P$. Finally, an equivalence is transitive, so that if $P$ is equivalent to $Q$ and $Q$ is equivalent to $R$, then $P$ is equivalent to $R$.

**Definition 7** A relation $\asymp$ over a set $T$ is an equivalence, if and only if for all $P, Q, R \in T$ the following hold

- $P \asymp P$ (reflexivity)
- If $P \asymp Q$ then $Q \asymp P$ (symmetry)
- If $P \asymp Q$ and $Q \asymp R$, then $P \asymp R$ (transitivity)
5.2 Congruences

Earlier we described how compositionality is supported at the operator level. Compositionality should also be supported at the level of behavioural semantics, and therefore we have to place an additional requirement for a behavioural equivalence, one that may not be as immediately obvious as the above three requirements.

Namely, assume that we have a system that is composed of processes $P$, $Q$, and $R$, and we find that process $P$ is equivalent to a simpler process $P'$, so we replace $P$ by $P'$ in order to simplify the construction. Then, we would obviously like that after the replacement the new composite system is still equivalent to the original system. This requirement is formally captured by the congruence property.

To define the congruence property formally, let us first investigate the notion of process contexts. A process context is a function that maps LTSs to an LTS, and represents one way of constructing a system from parameter processes by using process operators and other LTSs. If a process context $C$ has $n$ parameters, we let $L_1, \ldots, L_n$ represent the process parameter places, and write $C[L_1, \ldots, L_n]$. A context could involve just one operator, for example:

$$C_1[L] \equiv \text{hide}\{a, b\}\text{ in } L$$

$$C_2[L_1, L_2] \equiv L_1 || L_2$$

More complicated contexts can involve any combination of operators, for example:

$$C_3[L_1, L_2, L_3] \equiv \text{hide}\{a, b\}\text{ in } (L_1 || \text{hide}\{b, c, d\}\text{ in } (L_2 || L_3))$$

Formally, we can define a process context as follows:

**Definition 8** Assume

$$\Gamma = \{\text{op}_1(L_1, L_2, \ldots, L_{m_1}), \text{op}_2(L_1, L_2, \ldots, L_{m_2}), \ldots, \text{op}_k(L_1, L_2, \ldots, L_{m_k})\}$$

is a set of process operators. Syntactically, a process context with $n$ parameters, $C$, is any of the following:

1. $L_i$ for any $i = 1, \ldots, n$
2. any LTS $P$
3. $\text{op}_i(C_1, C_2, \ldots, C_m)$, where $i = 1, \ldots, k$, and $C_1, C_2, \ldots, C_m$ are process contexts with $n$ parameters
Let $P_1, \ldots, P_n$ represent any LTSs. Semantically, $C$ denotes a function that maps $n$ LTSs into an LTS such that, corresponding to the above forms, $C[P_1, \ldots, P_n]$ is

1. $P_i$

2. $P$

3. $\text{op}_i(C_1[P_1, \ldots, P_n], C_2[P_1, \ldots, P_n], \ldots, C_m[P_1, \ldots, P_n])$

When we are using a particular context $C[L_1, \ldots, L_n]$, we want to be able to replace any of the parameter processes with equivalent processes while retaining the equivalence of the result.

**Definition 9** An equivalence $\simeq$ is a congruence with respect to a process context $C[L_1, \ldots, L_n]$ if and only if for any LTSs $P_1, \ldots, P_n, P'_1, \ldots, P'_n$ such that $P_i \simeq P'_i$, $P_n \simeq P'_n$, it holds that $C[P_1, \ldots, P_n] \simeq C[P'_1, \ldots, P'_n]$.

We will say that the equivalence $\simeq$ is a congruence with respect to a process operator $\text{op}$ if and only if it is a congruence with respect to the corresponding context $C[L_1, \ldots, L_n] \equiv \text{op}(L_1, \ldots, L_n)$.

However, with any nontrivial operators there will be infinitely many possible process contexts, so we cannot prove the congruence property of an equivalence for each of them separately. Fortunately, this is not necessary, because it suffices to prove the congruence property for each of the operators we are using.

**Proposition 10** Let $\Gamma$ be a set of process operators. An equivalence $\simeq$ that is a congruence with respect to each operator $\text{op} \in \Gamma$, is a congruence with respect to any process context $C[\cdot]$ formed by using the operators in $\Gamma$.

**Proof** We use induction on depth $k$ of nested operators in the context. If $k = 1$, then the context $C$ contains a single operator, and $\simeq$ is a congruence for $C$ because it is a congruence for the operator. Then assume that the result holds for all contexts with at most $k$ levels of nested operators, and that context $C$ has $k + 1$ levels of nested operators. All the parameters of the topmost operator $\text{op}(L_1, \ldots, L_m)$ of $C$ are contexts with at most $k$ levels of operators. Thus, when the parameter LTSs $P_1, \ldots, P_n$ at the lowest level are replaced by equivalent LTSs $P'_1, \ldots, P'_n$, then the induction hypothesis implies that all the parameters of the topmost operator become replaced by equivalent LTSs. Since $\simeq$ is a congruence with respect to $\text{op}$, it follows that $C[P_1, \ldots, P_n] \simeq C[P'_1, \ldots, P'_n]$, and this concludes the proof. $\Box$
For this reason we will only discuss the congruence property of equivalences with respect to individual operators. However, it is also important to notice the difference to the axioms of equivalence. Fulfilling the axioms was a property of the relation itself, but whether or not an equivalence is a congruence potentially depends on the set of operators that we are allowed to use in constructing systems. We will usually require that any equivalence we use should be a congruence at least with respect to parallel composition and hiding. Fortunately, it often turns out that when an equivalence can be made a congruence for these operators, many other operators can be covered without major changes to the equivalence.

There is, however, one desired property of equivalences we have not discussed yet. Namely, if we would not equate any two (syntactically different) LTSs, then this notion of “equivalence” is trivially an equivalence relation and a congruence with respect to any operator. However, such an equivalence is not of much use, because it differentiates LTSs even if these would behave in the same way in all respects that we are interested in. Therefore, this is a too strong, or fine, notion of equivalence.

The desired property we were referring to is that our equivalence should not make unnecessary distinctions between processes, that is, distinctions that are not required by our concept of process behaviour or by the congruence requirement. In other words, we want our equivalence to be as weak, or coarse, as possible. Formally, equivalence \( \simeq_w \) is weaker than equivalence \( \simeq_s \) if \( \simeq_s \subseteq \simeq_w \), in other words, if \( P \simeq_s P' \) implies \( P \simeq_w P' \) for any LTSs \( P, P' \). We also say that the stronger equivalence preserves the weaker. Typically, we want to find the weakest congruence that preserves an equivalence which describes the process behaviour that we are interested in. The fact that a (unique) weakest preserving congruence always exists is proven formally in the following.

**Proposition 11** Let \( \Gamma \) be a set of process operators, and let \( \simeq \) be any equivalence. Define the relation \( \equiv \) by \( P \equiv P' \) if and only if for all one-parameter process contexts \( C \) formed from \( \Gamma \), \( C[P] \simeq C[P'] \). Then, \( \equiv \) is a congruence that preserves \( \simeq \). Furthermore, it is the weakest such congruence.

**Proof** It is easy to see that \( \equiv \) is reflexive and symmetric. For transitivity, if \( P_1 \equiv P_2 \land P_2 \equiv P_3 \) then \( \forall C : C[P_1] \simeq C[P_2] \land \forall C : C[P_2] \simeq C[P_3] \), so \( \forall C : (C[P_1] \simeq C[P_2] \land C[P_2] \simeq C[P_3]) \) and, since \( \simeq \) is an equivalence, \( \forall C : C[P_1] \simeq C[P_3] \), so \( P_1 \equiv P_3 \). Thus, \( \equiv \) is also an equivalence. Assume \( P \) and \( P' \) are any LTSs such that \( P \equiv P' \). Then, because any combination of contexts is also a context, it holds that \( \forall C \forall D : D[C[P]] \simeq D[C[P']] \), which implies that \( \forall C : C[P] \equiv C[P'] \). We have shown that \( \equiv \) is a congruence
for any one-parameter context. In an \(n\)-parameter context we can fix all parameters except one as constant LTSs, and replace the chosen parameter process \(P_i\) with an equivalent process \(P'_i\), and then by the previous result the system remains equivalent to the old one. In this way we can, one by one, replace all the parameter processes. Therefore, the result generalises to an arbitrary \(n\)-parameter context. To show that \(\cong\) implies \(\simeq\) it suffices to use the identity (empty) context. Finally, if \(\cong'\) is a congruence that preserves \(\simeq\), then \(P \cong' P'\) implies \(\forall C: C[P] \cong' C[P']\) and, since \(P \cong' P'\) implies \(P \simeq P'\), that \(\forall C: C[P] \simeq C[P']\). Thus, \(P \cong P'\).

These somewhat abstract ideas will be made more concrete in the following when we discuss different ways of looking at process behaviour.

A concept closely related to an equivalence is that of a preorder. A preorder between processes is usually intended to formalise the idea that one process behaves in a “better” or “more deterministic” way than the other process. This is useful when we do not require that two processes behave exactly in the same way but, say, that the one process is a specification and the other process must fulfill that specification. Formally, any relation \(\leq\) that is reflexive and transitive is a preorder, so that, compared to the definition of an equivalence in Definition 7, we drop the requirement of symmetry. In analogy with the congruence property defined above, a preorder that describes the behaviour of systems should be a precongruence, so that replacing parameter processes with smaller or equivalent processes produces a smaller or equivalent result.

Every preorder \(\leq\) defines a corresponding equivalence \(\simeq\), namely the one for which \(P \simeq Q\) if and only if \(P \leq Q\) and \(Q \leq P\). This also means that showing a relation to be a preorder in both ways can be used to show that it is the corresponding equivalence. It should be noted, however, that several preorders can correspond to the same equivalence. In practice, this means that if we have a behavioural preorder, then we automatically have also a behavioural equivalence. On the other hand, a preorder compatible with a given equivalence must be explicitly designed.

A preorder is used, for example, in the enclosed publication [V].

### 5.3 Isomorphism and Tree-Equivalence

In our model, the behaviour of a process consists of executing actions, and we cannot directly observe the state that the process is in. Therefore, as was already discussed in Section 4.2, it is not sensible to require identity between LTSs, because this differentiates LTSs which have an identical transition
structure but different internal construction of states. As was proposed there, the problem can be avoided by using isomorphism between LTSs, given in Definition 4, because isomorphism preserves only the transition structure and ignores the syntax of the states.

However, the notion of isomorphism is very restrictive. This can be seen by comparing processes \( P \) and \( Q \) in Figure 5.1. These processes cannot be isomorphic, because they have different numbers of states. However, if we look at the execution of actions, both processes clearly behave in the same way, because all they can do is execute endlessly \( a \)s.

We notice that the only difference between the two LTSs is that the latter starts repeating the same state one “\( a \)” later than the former. This suggests a different notion of equivalence. Namely, we can ignore the identity of states in an execution, and only look at the possibility, at each point, of executing further transitions. We start from the initial state of the LTS and “unfold” the LTS into a possibly infinite tree [50, 42]. The tree has no cycles, and there is a unique route from the root node (initial state) to every other node. We get the behavioural equivalence by requiring that the unfolded trees of the two LTSs must be isomorphic.

This solves the present problem, because both of the example processes unfold into a tree with the same structure, namely, an infinite sequence of \( a \)-transitions, as shown in Figure 5.2.

However, a problem with this idea is that a system may have several branches with the same behaviour, as exemplified by process \( R \) in Figure 5.3. Because of the extra branch, the unfolded tree of \( R \) is different from those of \( P \) and \( Q \), as shown in the picture, and therefore \( R \) cannot be equivalent to these processes according to the tree equivalence. Of course, it would be possible to take the view that the existence of several identical branches

\[
egin{array}{c}
\overset{a}{\longrightarrow} \quad \overset{a}{\longrightarrow} \quad \overset{a}{\longrightarrow} \quad \cdots
\end{array}
\]

Figure 5.2: The unfolded tree of processes \( P \) and \( Q \) of Figure 5.1
is a property of the system and should be preserved. However, since our main aim is to simplify systems by abstracting away unessential features of system behaviour, in most cases we do not wish to preserve different branches with identical behaviour. Therefore, we want to find a less strict notion of behavioural equivalence.

5.4 Strong Bisimilarity

One intuitive idea for defining behavioural equivalences is that equivalent processes should be able to “simulate” each other’s behaviour in every possible situation. This is the idea used in bisimulation-based equivalences.

Intuitively, if two processes are bisimilar, there exists a correspondence, or simulation, between their states; here we allow one state to simulate several states of the other process. The initial states of the processes must simulate each other, and always when the processes are in simulating states and one process can perform an action, the other process must be able to perform the same action in such a way that the two processes end up in simulating states.

Expressing this idea formally, we can define strong bisimilarity, or strong observation equivalence between two processes as follows.

Definition 12 Let $L_1 = (S_1, \Sigma, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, \Sigma, \Delta_2, \hat{s}_2)$ be LTSs. The binary relation $\sim \subseteq S_1 \times S_2$ is a strong bisimulation if and only if for every $a \in \Sigma \cup \{\tau\}$ and for all $s_1 \in S_1$, $s_2 \in S_2$ such that $s_1 \sim s_2$, the following hold:

- If $s_1 - a \rightarrow s'_1$ for some $s'_1$, then $s_2 - a \rightarrow s'_2$ for some $s'_2$ such that $s'_1 \sim s'_2$.
• If $s_2 \xrightarrow{a} s'_2$ for some $s'_2$, then $s_1 \xrightarrow{a} s'_1$ for some $s'_1$ such that $s'_1 \sim s'_2$.

$L_1$ and $L_2$ are strongly bisimilar (strongly observation equivalent) if and only if there exists a strong bisimulation $\sim$ between them such that $\hat{s}_1 \sim \hat{s}_2$.

It is easy to establish a strong bisimulation between process $P$ (or $Q$) in Figure 5.1 and process $R$ in Figure 5.3, and therefore these processes are strongly bisimilar.

It can be shown that strong bisimilarity is an equivalence and a congruence with respect to parallel composition and hiding. Strong bisimilarity is generally considered to be the strongest, i.e., the most discriminating, equivalence between processes that one may wish to use. This means that any two processes that are strongly bisimilar will also be equivalent according to any behavioural equivalence that we will discuss in the following.

### 5.5 Weak Bisimilarity

It is important to notice that strong bisimilarity treats invisible $\tau$-actions no differently from visible actions. For example, processes $P$ and $Q$ in Figure 5.4 are different by strong bisimilarity. However, we usually take the view that a finite number of $\tau$-actions is not directly observable from the outside, so that, for example, whether the system executes one, two or some other finite number of $\tau$-actions makes no difference.

Therefore, at the semantical level we usually want to abstract away from the internal invisible actions. As discussed earlier in connection with the hiding operator, this may allow us to replace the original, complicated system with another, simpler and smaller system. Thus, strong bisimilarity is usually too strong as a behavioural equivalence notion, although it will nevertheless be very useful for many purposes.

In order to make it easier to discuss the executions of systems only in terms of the visible actions, we will in the following define the “$=\sigma \Rightarrow$”-notation, which is analogous to the “$-\eta \rightarrow$”-notation, except that only the visible actions along the execution are listed.

![Figure 5.4: Two weakly bisimilar processes distinguished by strong bisimilarity](image)
Definition 13 Let \((S, \Sigma, \Delta, \hat{s})\) be an LTS, let \(s, s' \in S\), \(b, b_1, b_2, b_3, \ldots \in \Sigma\).

We write

- \(s = \varepsilon \Rightarrow s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0, s_n = s'\) and \(s_{i-1} \rightarrow \tau \rightarrow s_i\) when \(1 \leq i \leq n\) (notice that \(s = \varepsilon \Rightarrow s\) for all \(s \in S\)),

- \(s = b \Rightarrow s'\) if and only if there are \(s_1, s_2 \in S\) such that \(s = \varepsilon \Rightarrow s_1\) and \(b \rightarrow s_2 \Rightarrow s'\),

- \(s = b_1 b_2 \cdots b_n \Rightarrow s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0, s_n = s'\) and \(s_{i-1} = b_i \Rightarrow s_i\) when \(1 \leq i \leq n\),

- \(s = b_1 b_2 \cdots b_n \Rightarrow s'\) if and only if there is \(s_1, s_2, \ldots \in S\) such that \(s = s_0\) and \(s_{i-1} = b_i \Rightarrow s_i\) when \(i \geq 1\).

Let us again look at the example in Figure 4.1. The following holds of the system:

- \(s_1 = \varepsilon \Rightarrow s_1\)
- \(s_1 = \text{InsertCard} \Rightarrow s_2\)
- \(s_1 = \text{InsertCard Withdraw} \Rightarrow s_3\)
- \(s_3 = \text{Refuse CardOut} \Rightarrow s_1\)
- \(s_1 = \text{InsertCard Withdraw Grant CardOut InsertCard Cancel} \Rightarrow s_4\)
- \(s_1 = \varepsilon \Rightarrow s_1\)

By using a similar idea of mutual simulation as with strong bisimilarity, while only looking at the visible actions that the system executes, we can define weak bisimilarity, or weak observation equivalence between processes.

Definition 14 Let \(L_1 = (S_1, \Sigma, \Delta_1, \hat{s}_1)\) and \(L_2 = (S_2, \Sigma, \Delta_2, \hat{s}_2)\) be LTSs. The binary relation \(\approx \subseteq S_1 \times S_2\) is a weak bisimulation if and only if for every \(b \in \Sigma \cup \{\varepsilon\}\) and for all \(s_1 \in S_1, s_2 \in S_2\) such that \(s_1 \approx s_2\), the following hold:

- If \(s_1 = b \Rightarrow s'_1\) for some \(s'_1\), then \(s_2 = b \Rightarrow s'_2\) for some \(s'_2\) such that \(s'_1 \approx s'_2\)
- If \(s_2 = b \Rightarrow s'_2\) for some \(s'_2\), then \(s_1 = b \Rightarrow s'_1\) for some \(s'_1\) such that \(s'_1 \approx s'_2\)

\(L_1\) and \(L_2\) are weakly bisimilar (weakly observation equivalent) if and only if there exists a weak bisimulation \(\approx\) between them such that \(\hat{s}_1 \approx \hat{s}_2\).
Figure 5.5: Two processes distinguished by branching bisimilarity

Weak bisimilarity is a congruence with respect to parallel composition and hiding. However, it is not a congruence with respect to the choice operator which was mentioned in Section 4.2. Therefore, weak observation congruence \( \approx_c \) is defined to be the weakest equivalence that preserves weak bisimilarity and is a congruence with respect to the choice operator [32].

It should be noted that weak bisimilarity does not preserve the ability of systems to diverge (livelock), i.e., to perform an infinite sequence of invisible \( \tau \)-actions (see Section 5.8). However, it is possible to define a divergence-sensitive variant of weak bisimilarity, so that divergent states are not allowed to simulate non-divergent states [15, 49].

It has been argued that weak bisimilarity, unlike strong bisimilarity, does not preserve all of the branching behaviour of a process. A counterexample could be the processes \( P \) and \( Q \) in Figure 5.5. In \( P \) it is possible to execute the sequence of visible actions “\( ab \)” in such a way that between these two actions we do not pass through a state with the following property: it is still possible to execute \( c \), but not possible to execute \( d \). However, in \( Q \) the (only) execution of “\( ab \)” necessarily passes through such a state between \( a \) and \( b \). Therefore, it is argued, the branching structures of \( P \) and \( Q \) are different. However, \( P \) and \( Q \) are weakly bisimilar.

The proposed solution is branching bisimilarity [50]. We arrive at this equivalence by requiring that also the intermediate states of the two executions must simulate each other in a certain way. Branching bisimilarity is therefore able to differentiate between the example processes \( P \) and \( Q \) above.
5.6 Denotational Semantics

In defining behavioural equivalences it is possible to take an approach that is different from the idea of bisimulation presented above. In this approach, we consider a system as a kind of “black box”. The system communicates with the external world through the execution of visible actions. Therefore, the visible actions form the interface of the “box”. Looking at the “box” from the outside, we cannot directly observe the internal \( \tau \)-actions, or see the state the system is in. The question which follows is what kind of behaviour the system may exhibit when it is communicating with the external world in this way.

As our first example of such behaviour, we may want to know what visible actions and in which order the system can execute, in any finite amount of time, assuming the environment does not constrain the execution of actions. In other words, we would like our model to include the sequences of visible actions that the system can execute. We will call these sequences \textit{traces}.

\textbf{Definition 15} Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS. Then, \( \text{Tr}(L) = \{ \sigma \in \Sigma^* \mid \hat{s} = \sigma \Rightarrow \} \) is the set of traces of \( L \).

We can use the set of traces as our model of process behaviour, and the corresponding notion of equivalence is obtained by comparing the trace-sets of two processes:

\textbf{Definition 16} Let \( L \) and \( L' \) be LTSs with the same alphabet.

- The \textit{trace model} of \( L \) is the set \( \text{Tr}(L) \)
- \( L \) and \( L' \) are \textit{trace-equivalent}, \( L \simeq_T L' \), if and only if \( \text{Tr}(L) = \text{Tr}(L') \)

Before looking at other forms of behaviour that a system may exhibit, let us first generalise on the above reasoning that led to the trace-equivalence. Notice that the set of traces, \( \text{Tr} \) is a function that maps an LTS into a set of sequences of actions. Assume now that we have captured the interesting aspects of process behaviour by functions \( m_1, m_2, \ldots, m_k \), where each \( m_i \) maps an LTS into some mathematical structure that represents a particular aspect of the behaviour. Then, our model of the behaviour of a process \( L \) is the tuple \( (m_1(L), m_2(L), \ldots, m_k(L)) \). The corresponding equivalence is the obvious one that demands equal models of system behaviour, that is, \( L \simeq L' \) if and only if \( m_1(L) = m_1(L') \), \( m_2(L) = m_2(L') \), \ldots, and \( m_k(L) = m_k(L') \). We will call this equivalence “\textit{m}_1-\textit{m}_2-\ldots-\textit{m}_k\text{-equivalence}”. This approach is sometimes called \textit{denotational}, because the collection of items that model the process behaviour can be said to denote the process.
It is easy to see that any relation defined in this manner fulfills the three axioms in Definition 7, and is therefore an equivalence. Proving that a particular denotational equivalence is a congruence, on the other hand, can be significantly more difficult. However, the denotational representation offers a standard approach for proving the congruence property. Namely, the $m_1$-$m_2$-$\ldots$-$m_k$-equivalence is a congruence with respect to process operator $op(L_1, \ldots, L_n)$ if and only if the components in the denotational model of the resulting LTS can be obtained as functions of the components in the denotational models of the parameter processes. For example, $Tr(L_1 \parallel L_2)$ can be given as a function of $Tr(L_1)$ and $Tr(L_2)$, namely

$$Tr(L_1 \parallel L_2) = \{ \sigma \in (\Sigma_{L_1} \cup \Sigma_{L_2})^* \mid restr(\sigma, \Sigma_{L_1}) \in Tr(L_1) \land restr(\sigma, \Sigma_{L_2}) \in Tr(L_2) \}.$$ 

Here, $restr(\sigma, A)$ means the removal of all actions from the sequence $\sigma$ that are not in $A$. The existence of the above function shows that $Tr$-equivalence is a congruence with respect to the parallel composition operator $\parallel$. In the same manner we can show that $Tr$-equivalence is a congruence with respect to the hiding operator (and, in fact, with respect to more or less any other reasonable process operator one may want to consider; this means that $Tr$-equivalence is a very robust equivalence with respect to process operators).

The general idea is presented formally in the following result.

**Proposition 17** The $m_1$-$m_2$-$\ldots$-$m_k$-equivalence $\simeq$ is a congruence with respect to a process operator $op(L_1, \ldots, L_n)$ if and only if there exist functions $g_1, \ldots, g_k$ such that for all LTSs $P_1, \ldots, P_n$,

$$m_i(op(P_1, \ldots, P_n)) = g_i(m_1(P_1), \ldots, m_k(P_1), \ldots, m_1(P_n), \ldots, m_k(P_n))$$

for each $i = 1, \ldots, k$.

**Proof** For the $\Rightarrow$ part, assume that the equivalence is a congruence with respect to the operator. For any $m_i(op(P_1, \ldots, P_n))$, it holds that if the parameter processes are replaced by arbitrary processes with the same $m_1, \ldots, m_k$-values, then those processes are, by definition of the $m_1$-$\ldots$-$m_k$-equivalence, equivalent to the old ones. Thus, by the congruence property, the result remains equivalent and therefore the $m_i$-value of the result remains the same. Thus, the $m_i$-value has to be a function of the $m_1, \ldots, m_k$-values of the parameter processes.
For the “⇐” part, assume that the given functions $g_i$ exist. If we replace the parameter LTSs $P_1, \ldots, P_n$ with any equivalent LTSs $P'_1, \ldots, P'_n$, respectively, then, by definition, the new parameters have the same $m_1, \ldots, m_k$-values as the old parameters. The existence of the functions $g_i$ then implies that $m_i(op(P_1, \ldots, P_n)) = m_i(op(P'_1, \ldots, P'_n))$ for each $i = 1, \ldots, k$. Thus, $op(P_1, \ldots, P_n) \simeq op(P'_1, \ldots, P'_n)$. □

5.7 Deadlocks and Stable Failures

It is easy to see that trace-equivalence alone does not capture all aspects of behaviour that one may consider important. For example, the processes in Figure 5.6 (a) and (b), which could represent the behaviour of a simple communication protocol, are equivalent in the trace-semantics, because they can execute the same sequences of visible actions, namely $Tr(P_1) = Tr(P_2) = (send \ rec)^* \cup (send \ rec)^* \ send$. However, after executing $send$, process $P_2$ can execute an invisible $\tau$-action and enter a state where no actions are possible. We call such a state a deadlock state. Being in a deadlock state means that the process stops permanently.

We would usually like to know if the system can deadlock and, in most cases, we would also like to know the traces after which a deadlock can occur. Therefore, the semantics should preserve the following set.

**Definition 18** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS. Then, $Dltr(L) = \{ \sigma \in \Sigma^* \mid \exists s \in S : \hat{s} = \sigma \Rightarrow s \land \forall a \in \Sigma \cup \{\tau\} : \neg(s - a \rightarrow ) \}$ is the set of deadlock traces of $L$.

It may seem like a good idea to add the deadlock traces to our model of process behaviour, so that we would compare processes by using $Tr-Dltr$-

![Figure 5.6: A non-deadlocking and deadlockng process](image)
equivalence. However, it turns out that this equivalence is not a congruence with respect to the parallel composition operator. For example, if we consider the processes $P_1$ and $P_2$ shown in Figure 5.7, we notice that they have the same set of traces, and neither have any deadlock traces, so $P_1$ and $P_2$ are equivalent according to $Tr$-$Dl$tr-equivalence. However, when combined in parallel with the process $Q$, as shown in Figure 5.8, $P_2 || Q$ deadlocks but $P_1 || Q$ does not.

The problem is that after executing $aa$, process $P_1$ is in a state which offers both actions $a$ and $b$, but $P_2$ is either in a state which offers only $a$, or in a state which offers only $b$. Process $Q$ offers in the corresponding state only the action $b$, and $P_1$ will accept this. However, if $P_2$ happens to be in the former state, it cannot agree on any action with $Q$, and the system deadlocks. Therefore, we need to include in our model information about the ability of the process to refuse visible actions.

We also notice that if a state of a process has an outgoing invisible action, then the system cannot be deadlocked while the process is in this state, because irrespective of what visible actions are offered or refused by the environment, the process can always choose the invisible action. These observations lead to the definition of stable failures.

**Definition 19** Let $L = (S, \Sigma, \Delta, s)$ be an LTS. Then, $Sfail(L) = \{ (\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : \hat{s} = \sigma \Rightarrow s \wedge \forall a \in A \cup \{\tau\} : \neg(s \rightarrow a) \}$ is the set of stable
failures of $L$.

Clearly, for any LTS $L = (S, \Sigma, \Delta, \hat{s})$ it holds that

$$Dltr(L) = \{ \sigma \in \Sigma^* \mid (\sigma, \Sigma) \in S_{fail}(L) \}.$$ 

Therefore, any equivalence that preserves $S_{fail}$ preserves $Dltr$. Also, we can show by a similar technique as above that $S_{fail}$-equivalence is a congruence with respect to “||” and “hide”; the required functions can be found, for example, in [47] or in the enclosed publication [II].

## 5.8 Divergences

Apart from deadlocks, there is another reason why a system may stop responding to visible actions offered by the environment. Namely, if we look at the system in Figure 5.9, we see that after send this system may execute infinitely many invisible $\tau$-actions and thus stop doing anything visible. This phenomenon is called a livelock, or divergence. In analogy with deadlock traces, we define divergence traces as the traces that can lead to a divergence.

**Definition 20** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS. Then, $Divtr(L) = \{ \sigma \in \Sigma^* \mid \exists s : \hat{s} = \sigma \Rightarrow s \land s \not\rightarrow^\omega \}$, where $\tau^\omega$ denotes the infinite sequence of $\tau$-actions, is the set of divergence traces of $L$.

![Figure 5.8: Process Q and its parallel composition with $P_1$ and $P_2$](image)

Figure 5.8: Process Q and its parallel composition with $P_1$ and $P_2
It can be shown that the traces of the LTS $L = (S, \Sigma, \Delta, \hat{s})$ can be determined from its stable failures and divergence traces by the following formula [47]; intuitively, this is because after executing a particular trace the system has to either eventually stabilise or execute endlessly $\tau$-actions.

$$Tr(L) = Divtr(L) \cup \{ \sigma \in \Sigma^* \mid (\sigma, \emptyset) \in Sfail(L) \}$$

This shows that in the presence of $Sfail$ and $Divtr$ in the semantic model, the $Tr$ component becomes redundant. Thus, $Tr$-$Sfail$-$Divtr$-equivalence is the same as $Sfail$-$Divtr$-equivalence.

However, it again turns out that $Sfail$-$Divtr$-equivalence is not a congruence when we allow arbitrary (namely, infinite) LTSs, and this time the problem is not parallel composition but hiding. Namely, infinite execution sequences with infinitely many visible actions can turn into divergences when visible actions are hidden. In general, the $Sfail$ and $Divtr$ components do not preserve enough information about these execution sequences to allow us to always correctly determine the divergences that result from hiding [47]. To remedy this problem we can include in our model also the infinite sequences of visible actions that the system can execute.

**Definition 21** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS. Then, $Inftr(L) = \{ \xi \in \Sigma^\omega \mid \hat{s} = \xi \Rightarrow \}$ is the set of infinite traces of $L$.

Now our model of the behaviour of a system $L$ is the tuple

$$(Sfail(L), Divtr(L), Inftr(L))$$

and the corresponding equivalence is the $Sfail$-$Divtr$-$Inftr$-equivalence. These are called the Chaos-Free Failures Divergences (CFFD) model and equivalence [47]. CFFD-equivalence can be shown to be a congruence with respect to parallel composition and hiding, as well as certain other operators such as renaming. By including also the so-called initial stability bit to the model, which tells whether or not there are $\tau$-transitions from the initial state of the LTS, the equivalence becomes a congruence also for the choice operator.

Figure 5.9: A diverging process
5.9 Notes on Equivalences

When looking at the bisimulation-based equivalences and the denotational equivalences, we notice that these represent two somewhat different philosophical and practical approaches to behavioural semantics. In the first approach, we try to retain the internal structure of the behaviour as much as possible, given the division into visible and invisible actions. On the other hand, the view in the other approach is that since we are, in any case, discarding information about details of the behaviour, we may as well discard any information that does not affect the specific external behaviours that we are interested in.

The difference between these two approaches is related to the difference between the linear time and branching time logics discussed in Section 3.2. However, we should observe that the stable failures are not a purely “linear” property of systems, because they involve branching time information about the possible actions that can be executed next. On the other hand, such branching properties are of a very special and restricted form, and give just enough information to enable predicting deadlocks. Sometimes this type of a semantics is called a “decorated trace semantics”.

If we again look at Figure 5.7, we notice that trace equivalence equates all three LTSs, $P_1$, $P_2$ and $P_3$. The semantics based on stable failures that were discussed above distinguish between $P_1$ and $P_2$, because of their different failure behaviour after $a a$, but they equate $P_2$ and $P_3$ which have identical failure behaviour. The bisimulation based equivalences, however, distinguish between all three processes. This is because in $P_2$ both the executions $a a$ and $a b$ are still possible after the first $a$, while in $P_3$ this is not the case.

As another example, we can take a look at the behaviour of a cash point shown in Figure 5.10 and compare this with the original behaviour in Figure 4.1. We notice that the new system may choose to refuse the service already before the user has made a request. From the first viewpoint discussed above, we could say that the system makes the decision at a different point, and therefore the two behaviours are different. From the other point of view, we could say that the customer interacting with the cash point has no way of distinguishing the two behaviours. After the request has been made the system can, in any case, give either response, and therefore the external behaviours of the two systems are the same.

The benefit of preserving as little information as possible about the behaviour is that this allows greater reduction of systems. However, there are some situations where it can be useful to know what courses of actions are still possible. This is especially true of systems which involve stochastic choices. If the system has entered a livelock, we may want to know if an alternative
Figure 5.10: An alternative behaviour of a cash point

correct behaviour is also possible. This can give a hint as to whether the
livelock is caused by the same choice being repeated over and over again,
or whether there is a real error in the design of the system. However, some
progress is being made in this area through the use of fairness constraints
with denotational semantics; see Section 7.1 and the enclosed publication
[IV].

On the other hand, because of the coinductive way the bisimulation-based
equivalences are defined, it can be more difficult to obtain an intuitive under-
standing of all the behavioural properties that these equivalences preserve.
For partly the same reason, if two processes turn out to be inequivalent in a
denotational model based on elements such as traces, stable failures etc., a
counter-example can be expressed directly in terms of one of these elements,
while expressing a counter-example for a bisimulation-based equivalence can
be more complicated.

In any case, it is likely that no single equivalence can serve all purposes.
The equivalences form a hierarchy, where different equivalences preserve dif-
ferent properties and allow different types of reduction; see, for example,
Figure 7.1 in Section 7.1. Therefore, it is very useful if a verification tool is
able to support a hierarchy of different equivalences.
Chapter 6

Application to Protocol Verification

Perhaps the most important application of process algebra has been in the design and verification of communication protocols, which were introduced in Section 2.6. In the following we will discuss the concept of communication protocols with the help of the OSI reference model. We will then use the so called alternating bit protocol as an example of applying process-algebraic theory and visual verification to the design of protocols. The verification examples in the enclosed publications are also based on versions of this protocol. Examples of the application of closely related methods to the verification of communication protocols can be found, for example, in [20, 24, 26, 27, 45, 48]; see also the references given in the enclosed publications.

6.1 Communication Protocols and the OSI Model

If we take a detailed look at a communication action between two application programs over a computer network, we find that every exchange of information requires an enormous number of operations at various parts of the system. Therefore, without some kind of organisation and abstraction it would be more or less impossible to understand and design communication protocols.

A very useful way of understanding protocols is to arrange the communication actions into separate, independent layers. For this purpose the International Standards Organisation (ISO) has introduced the Open Systems Interconnection (OSI) reference model which divides the functions of a computer network into seven layers, each of which has been assigned a particular
task. An individual layer offers services to the layer above it, and performs its task by using the services offered by the layer below it. We will next briefly look at the functions of these seven layers.

The lowest layer in the OSI model is the **physical layer** which transmits a sequence of bits as a signal over a connection between two computers. The connection can be, for example, a coaxial cable, optic fibre, or a microwave radio link.

The task of the second layer, **data link layer** is to transmit frames of data over the connection from one computer to the other. The data link layer uses the raw, possibly unreliable, bit stream transportation provided by the physical layer for transferring the data. The data link layer takes care of inserting and detecting frame boundaries, calculating frame checksums, retransmitting frames in case of failures, and so on.

The task of the **network layer** (layer 3) is to route packets of data through a network of interconnected computers. Packets are transmitted between each pair of neighbouring, physically connected computers by using the services of the data link layer which controls that connection. At each node, the network layer has to choose for every received data packet a neighbouring node to which the packet is forwarded so that it will traverse in the right direction and, eventually, reach the correct recipient. This decision making is encoded as a routing algorithm.

The purpose of the **transport layer** (layer 4) is to provide end-to-end data transmission services between client programs. One form of service that is often provided by the transport layer is a reliable data stream in both directions between two client programs. Typically, the transport layer breaks the message stream into packets which it hands down to the network layer. The network layer then routes the packets to the destination computer where they are handed back to the transport layer, which puts them back into the correct order. The transport layer may use checksums for detecting transmission errors, and retransmit the data in case of errors or missing packets.

In practice, most applications use directly the transport layer for their communication needs. However, the OSI reference model defines further layers on top of the transport layer which provide advanced communication services. These are sometimes referred to as the **upper layers**.

The **session layer** (layer 5) controls the dialogue between the communicating parties. For example, it can keep track of whose turn it is to send data. The session layer also facilitates checkpoints in the communication stream, allowing a recovery from error situations by returning to a previous checkpoint. It also provides a mechanism for exception reporting.

It should be noted that the well-known interprocess communication para-
Remote Procedure Calls (RPC), which was mentioned in Section 3.1, is usually associated with this layer, even though it does not completely fit into the OSI model. RPC allows a computer program to call a procedure that resides on a different computer. From the point of view of the caller this looks much like an ordinary procedure call, because it is actually calling a local “client stub”, which encodes the parameters into a message that is sent to the receiving side. There, a “server stub” decodes the parameters and calls the target procedure with these parameters. Return values are handled similarly. The RPC mechanism also has to cope with network failures, server crashes and other problems. When this happens it can, for example, raise an exception in the calling program.

The presentation layer (layer 6) is, unlike the lower layers, concerned with the meaning of the data that is transmitted. Its task is to convert data and data structures from the local, internal representation at the sending machine to a machine-independent bit stream that can be transmitted through the network, and decoded at the receiving end. The presentation layer is also concerned with the compression and encryption of data.

The seventh and topmost layer is the application layer, which contains the actual applications that use the network for their communications needs. However, some applications are so common that they have been standardised as part of the application layer. This includes remote file access and transfer, and remote job control. There is also a service for coordinating multiparty interactions, called Commit, Concurrency and Recovery, which can be used to group a sequence of operations between different parties into an atomic action. All operations in the atomic action are either successfully completed, or otherwise the participants of the action return to their initial states.

6.2 Unreliable Channels

A typical problem we face when designing a communication protocol for either layer two (data link layer) or layer four (transport layer) of the OSI reference model is similar to the following. We have a sequence of data items that we want to send over a communication channel, but the channel is unreliable and can sometimes corrupt or lose messages.

In practice, we can identify two types of reasons why a channel may be unreliable. For protocols on layer two, the channel is typically a physical wire that connects two computers. There are many internal and external factors that can alter the signal that is used for transmitting the information, such as thermal noise in the wire and electromagnetic impulses from other lines and the environment. Furthermore, these errors tend to come as sporadic bursts.
It has long been known that, due to the nature of the errors, this problem cannot be solved solely by physical means, for example, by increasing transmission voltage, or by adding protective layering to the cables. Therefore, the only way to solve the problem is to use communication protocols that can cope with the errors.

On the other hand, our communication channel can also be a network of computers through which the data packets are routed. This is typical for protocols on layer four of the OSI model. Messages can disappear or become corrupted for the same reason as above, namely because of physical distortion of a signal in a channel between two computers — even though we would usually expect that the data link layer on the local nodes can correct errors of this type. However, when communicating through a network, messages can disappear for other reasons. Firstly, a computer that is a node in the network may malfunction or crash, and therefore lose the messages that were stored in its buffer at that moment. Secondly, even if all nodes were functioning flawlessly, at times of heavy congestion the nodes of the network may have to discard messages in order to avoid buffer overflow.

There are at least two ways in which we can try to cope with the problem of distorted messages. One possibility is to add redundancy to the message in the form of an error-correcting code, which allows the receiver to recover the data if only limited damage has been done to the message. However, this causes significant overhead to the communication, even though in a typical case errors are relatively infrequent. Furthermore, when errors occur as bursts they are likely to destroy the message beyond recovery. It is therefore sensible to use this method only in special circumstances. This is the case if the response time is too long, for example if we are transmitting data to a space probe, or if there is no possibility of feedback, such as when we are broadcasting a message to a very large number of receivers.

Thus, in most cases it is better to add just enough redundancy to a message to allow detecting that an error has occurred, and retransmit any damaged data. In practice, random errors can be detected with very high probability by using a cyclic redundancy checksum (polynomial checksum). This means that we interpret messages as binary polynomials, where the bits of the message represent the coefficients, drawn from a modulo 2 arithmetic. We add to each message a residual checksum which makes the polynomial formed from the message divisible by a predefined generator polynomial. The recipient can then simply check whether the received message is divisible by the known polynomial; if not, then the message must have been corrupted.

It is in many ways more difficult to cope with the possibility that messages disappear completely in the communication channel, than it is to cope with the possibility of messages becoming corrupted — assuming that the
corruption can be detected. In fact, the former problem subsumes the latter if we assume that the receiving side simply discards all corrupted messages. This shows that if we devise a protocol that can deal with disappearing messages, it can also deal with corrupted messages. Thus, in the following, we will for simplicity only consider disappearing messages.

### 6.3 The Alternating Bit Protocol

Let us now consider the problem of designing a protocol that can deliver messages correctly even when the communication channels are unreliable as described above. This presentation is similar to the one in [45].

The simplest protocol would, of course, be one where the sending side simply sends the sequence of data messages one by one through the communication channel to the receiver. The obvious problem here is that one or several of the messages can be lost in the channel, whereby the receiving side ends up with an incomplete message sequence.

Assuming that it is also possible to send messages in the reverse direction, a more or less obvious solution is to send an acknowledgement message back to the sender for each correctly received data message. After sending a data message the sender starts to wait for an acknowledgement. When it gets the acknowledgement the sender knows that the message has been correctly received, and it can send the next message. However, if no acknowledgement has arrived after some given time interval, we can suspect that the message has been lost or corrupted. The sender then has to send the same message again and wait for an acknowledgement. If the acknowledgement is still not received, this is repeated again, and so on.

A schematic drawing of the suggested protocol is shown in Figure 6.1. The processes of this initial protocol are shown in Figure 6.2. The sender (S) and receiver (R) function as described above. The channels (DC and...
AC) take a message and then choose nondeterministically between two \( \tau \)-transitions. One \( \tau \)-transition leads back to the initial state, meaning that the message has disappeared, and the other \( \tau \)-transition leads to a state which signifies that the message is kept and is offered to the receiving side.

These four processes are combined in parallel and actions other than the external actions, as shown in Figure 6.1 are hidden. Thus, we form the system

\[
P = \text{hide } \{ sd, rd, sa, ra \} \text{ in } (S || DC || AC || R)
\]

We use the TVT toolset [19] to construct and reduce the system with a CFFD-preserving reduction algorithm, and we get the external behaviour shown in Figure 6.3. We notice that the behaviour does not appear to be correct, because after one \textit{send}-action there can be any number of \textit{rec}-actions. If we leave more actions visible, we will see the reason for this. Namely, even if an acknowledgement has not been received by the sender, this does not necessarily imply that the data message has actually been lost. It is also possible that it was the acknowledgement that was lost, or that the data message or the acknowledgement were delayed. Therefore, the sender may retransmit a message even if the original message has been correctly received, which can cause the message to be \textit{duplicated} in the received sequence. It is important to notice that in this scenario the receiving side has no way of knowing whether a message is a duplication or not, because two consecutive messages in the sequence are allowed to have identical content.

We can avoid this problem by attaching a \textit{sequence number} to each message, so that the receiver can distinguish between a new message and a retransmission. Of course, we cannot grow the sequence number forever, because otherwise an unlimited space would be required for encoding the se-

![Figure 6.2: The processes of the initial version of the protocol](image-url)
sequence number. Fortunately, it will suffice to have a finite range of sequence numbers which will rotate back to the beginning when we reach the upper limit of the range. In fact, we will find that because we are dealing with a stop-and-wait protocol, meaning that the sender waits until the previous message has been delivered before sending a new message, it suffices to have just two different sequence numbers, which is enough to distinguish between two consecutive messages. Typically, the bits 0 and 1, with the usual modulo 2 addition, are used as the sequence numbers.

Now the receiver can immediately check if the received message contains the same sequence number as the previous one. If it does, the message is a duplicate, so the receiver sends an acknowledgement but does not deliver the message to the receiving client. The new protocol components are shown in Figure 6.4.

At first sight this arrangement might appear to solve the problem, but when we construct the external behaviour of the system we get the result shown in Figure 6.5. We notice from the right part of the picture that the system can make any number of send-actions without intermediate rec-
actions. Let us consider the following scenario. The sender first sends a data message with the sequence bit 0. The receiver receives the message and sends an acknowledgement, but this is delayed, so the sender retransmits the message. After that the acknowledgement reaches the sender, and the sender knows that the message has been delivered. Therefore, the sender sends a new data message, with bit 1. However, this message is lost in the channel. At the same time, the receiver has sent an acknowledgement for the retransmission of the previous message, and this acknowledgement reaches the sender. The sender believes that the latest message has been successfully received, and it sends the next data message. Thus, a data message has gone permanently missing.

The problem here is that the sender erroneously identifies an acknowledgement for one message as an acknowledgement for another message. Therefore, we need a way of distinguishing the acknowledgements from each other. The solution is to add the one bit sequence number also to the acknowledgements. We add the sequence number to the actions $sa$ and $ra$ in $S$, $AC$ and $R$. In $S$ we also include a loop which reads an acknowledgement with the wrong sequence number away from the channel. The protocol we get in this way is the so-called alternating bit protocol [6]. The processes of the protocol and the resulting external behaviour are shown in Section 5 of [IV]. The external behaviour consists of alternating $send$ and $rec$-actions, and is therefore what we would expect. Also, the protocol cannot deadlock. However, there are two $\tau$-loops, or divergences, in the behaviour. We return to this problem in Section 7.1.
6.4 The Self-Synchronising Alternating Bit Protocol

If the channels of the alternating bit protocol become broken so that they stop delivering messages, the sender will keep on sending retransmissions without limit. However, most realistic protocols do not behave in this way. Instead, they would have an error handling mode, where they refrain from further retransmissions and notify the client about the problem.

We can therefore make our protocol more realistic, if we limit the number of message retransmissions to some finite number, and add an action `err`, by which the protocol declares to the client of the sending side that it is uncertain whether the packet reached the recipient. However, the problem with a straightforward implementation of this idea turns out to be essentially the same that above forced us to use sequence numbers in the messages. Namely, in case of a failure to receive an acknowledgement, the actual message may or may not have disappeared. In this case this means that after a transmission failure the sender and receiver may have a different perception as to what the sequence number of the next message should be.

To remedy this problem, we can add a special synchronisation message to the protocol which is used to synchronise the two sides. When the last transmission has ended in a failure, and the sender gets a new sending request, it first sends a synchronisation message, which has the same sequence number as the failed data message. If it receives an acknowledgement with the same sequence number, the sender knows that it is in agreement with the receiving side about the sequence number that should follow. This protocol, which we can call the `self-synchronising alternating bit protocol`, was first developed in [45]; a brief introduction can also be found in [III]. In [38] it was shown how the protocol can be extended to handle a more general type of channels, namely channels that may keep a message forever if the receiving side never accepts it.
Chapter 7

Conclusions

7.1 Introduction to the Included Publications

In Section 5.2 we stated that an equivalence should not make unnecessary distinctions between processes, that is, distinctions that are not required by the properties that we want to preserve or by the congruence requirement. In other words, we want our equivalence to be as weak (coarse) as possible.

We stated as our goal to preserve traces, deadlock traces and divergence traces. As mentioned in Section 5.8, CFFD-equivalence preserves these elements and is a congruence with respect to the parallel composition and hiding operators. The interesting question now emerges whether CFFD-equivalence is the weakest congruence that preserves these behaviours. The answer to this question turns out to be that it is not. More precisely, not all infinite traces need to be preserved to preserve divergence traces.

In [I] the weakest congruence that preserves divergence traces is identified, and this is found to be the $Tr$-$Divtr$-$Enditr$-equivalence, where $Enditr$ stands for the so called eventually nondivergent infinite traces (see [I] for a formal definition). An implication of this is that the weakest congruence preserving traces, deadlock traces and divergence traces is the $Sfail$-$Divtr$-$Enditr$-equivalence. In [I], also the weakest congruence that preserves the existence of (any) divergences is identified.

It should be noted, however, that CFFD-equivalence has another interesting property. Namely, by using a suitable mapping from the state-based setting of a Kripke structure to process algebra, CFFD can be shown to be the weakest congruence that preserves the LTL logic without the “nextstate”-operator, and preserves the presence of deadlocks ([25, 44]; see also Section 3.2).

Above we have made a distinction between a deadlock and a livelock,
i.e., divergence, although in both cases the system stops executing visible actions. This distinction can be necessary in a situation where one process that is locked in an infinite internal execution can steal system resources and thus prevent other processes from making progress. This is true, for example, when parallel processes are scheduled by a non-pre-emptive operating system. On the other hand, if we do not need to make this distinction, we can define the concept of “anylock”, which means that the system can either deadlock or livelock. In [II] the weakest congruence is identified which preserves anylock, that is, tells whether or not the system can stop executing visible actions. Also the weakest congruence which preserves anylock-traces is identified, that is to say, traces after which the system can stop executing visible actions. It should be noted that the “should-testing”-equivalence presented in [11] is the weakest congruence that preserves anylock for finite-state systems under the fairness assumption that if a state is encountered infinitely many times, then all of its outgoing transitions are eventually taken.

Figure 7.1 shows the relative strength of the congruences developed in [I] and [II] and a number of previously known congruences. The upper part of each box defines the equivalence, and the lower part, if present, gives a characterisation for the equivalence as a weakest congruence. If other operators apart from parallel composition and hiding are required for the weakest congruence result, this is also indicated.

Here we assume the infinitary version of CSP [42, Chapter 10]. The predicate dfdl, “divergence-free deadlocks”, is a characterisation for CSP developed in [II]. The predicate “immediate-any-lock” is a characterisation for CSP given in [42, Chapters 9–10] (see the errata on the www-page of the book http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency). The proof of the latter characterisation requires a more powerful operator than parallel composition and hiding, namely the multiple (relational) renaming operator of CSP. This is explained in more detail in Section 5 of [II]. We should mention that some languages, such as SCCS [31] and ACP [7, 5], use versions of the parallel composition operator that are powerful enough to implement multiple renaming. It should also be noted that in [42] the preserved predicates are expressed as “tests” on processes. Assuming the finitely nondeterministic case, it is possible to arrive at essentially the same equivalence by using a different and more general notion of applying tests on processes, namely the “must-testing” of [33].

Sfail-equivalence is a congruence for parallel composition, hiding, as well as certain other operators. However, the interrupt operator, denoted by “[]” in LOTOS, requires that the Tr-component is added to the equivalence [43].

In Section 6.4 we were discussing the self-synchronising alternating bit protocol. A bidirectional version of this protocol, which allows sending mes-
Figure 7.1: The relative strength of different congruences

sages simultaneously in both directions while still only using two channels between the peers, has been developed in [38] and [III]. When looking at the behaviour of a bidirectional protocol from the point of view of one transmission direction, we should expect it to behave similarly as a unidirectional protocol. However, a problem is that since the actions of the other transmission direction are hidden, these appear as $\tau$-actions in the resulting diagram,
and since in this protocol the other direction can operate independently of
the visible direction, the \( \tau \)-actions form cycles, that is, divergences. In [III]
this problem is solved by using a suitable infinite “fairness LTS” to restrict
the operations of the invisible side, so that the divergences disappear. This
is achieved by using a finite lower and upper bound in place of the infinite
fairness LTS, and by a proof which is based on changing the set of visible
actions to reveal information about the internal structure of the divergences.

The problem described above relates to a more general problem in system
modelling. If we want to build a simple model of a system that involves
stochastic decisions, such as which side of a two-way protocol makes progress,
or whether a channel delivers or loses a message, we have to present these
decisions as unrestricted nondeterministic choices. The problem is that be-
cause the same selection can be repeated forever, the system may lack desired
progress (liveness) properties.

A standard solution to this problem is to use fairness constraints. As
explained in Section 3.2, this means that we restrict the nondeterministic
choices in infinite executions so that the system is not allowed to constantly
favour some choices at the expense of others. However, with process algebras
it has been difficult to combine fairness constraints with the compositional
approach. A problem is also the potentially infinite and infinitely nondeter-
nomistic behaviour that results from using fairness constraints.

The enclosed publication [IV] is devoted to the problem of using fairness in
process algebra. The first goal in [IV] is to provide a careful analysis of what
it means to have fairness constraints in a compositional process-algebraic
setting. We identify potential problems that can, and have, emerged when
one tries to use fairness constraints in such a setting. Secondly, we define a
class of fairness constraints that can be used in a consistent way, and which
are compatible with existing semantics. In addition, we present an algorithm
that allows us to construct a finite representation of a fair system in every
case that one exists.

In Section 6.3 we mentioned that a problem in the external behaviour of
the alternating bit protocol are divergences, or \( \tau \)-loops. We can guess that
the loops are caused by endless repetition of message loss in the channels and
retransmission of the same message. Clearly, we can hope that this protocol
works as intended only if the channels are guaranteed to deliver messages
correctly at least from time to time. This requirement can be expressed as
fairness constraints of the class mentioned above. In [IV] this protocol is
used as an example, and it is shown that by using such fairness constraints
the behaviour of the protocol is corrected.

In our discussion of the alternating bit protocol and self-synchronising
alternating bit protocol in Chapter 6, a limitation was that we used models
of communication channels which have capacity one, that is, which can hold at most one message at a time. In reality, communication channels can have a greater capacity, and the ability of the channels to store messages could, in principle, affect the behaviour of a protocol, although we would not expect this to be the case for the protocols discussed here. Also, we have for simplicity assumed that the self-synchronising protocol transmits each message only once. Since the number of retransmission attempts should be only a question of performance optimisation, we would expect that the external behaviour is independent of the particular number of retransmissions.

The problem in proving that the external behaviour of a system is independent of the values of such parameters is that we would actually have to verify an infinite family of different systems, where each system has greater and greater parameter values and thereby a larger and larger state-space. However, by virtue of the compositionality of the process-algebraic approach we can sometimes use invariants to verify such families of systems. A process that is used as an invariant has an “interface” for connecting new processes, and the invariant property means that the behaviour of the system stays the same (or alternatively, that no new unwanted behaviours are introduced) with the addition of these processes.

In [46] such invariants were used to show that the behaviour of the self-synchronising alternating bit protocol, with channels of capacity one, is independent of the number of retransmissions. In [V] it is shown by using suitable invariants and order of system construction that the external behaviour of the protocol, with one retransmission, is the same with any finite capacities of the channels. Furthermore, these two proofs are combined in [V] to show that the behaviour is, at the same time, independent of the channel capacities and the number of retransmissions. A similar proof can be carried out for the classical alternating bit protocol presented above. However, if we use reliable channels with the classical protocol, it turns out that a finite invariant does not exist for proving that the external behaviour is independent of channel capacities.

7.2 Future Work

There are several ways in which it is possible to continue the work on fairness in process algebra. First of all, as described in [IV], we can build support for our method into verification tools. One possible theoretical extension is to loosen the compatibility requirement in [IV] by allowing certain restrictions on the process context. In this way it would be possible to apply the method in a wider variety of situations. For example, the choice between two or more
visible actions can be made fair, if there is a guarantee that after applying the fairness operator these actions are hidden.

Another possibility is, as discussed in [IV], to find finite representations for the fair infinite LTSs, and to prove behavioural properties of the finite representations through the properties of the underlying ordinary infinite LTSs. In this way it might not only be possible to remove livelocks, that is, infinite $\tau$-executions, but also to represent fair choices between visible actions.

Finally, it may be possible to develop stronger semantic congruences that preserve fairness-related information, allowing us to use different types of fairness constraints. It should also be investigated what types of fairness constraints are most useful in practice.

In [V] it turned out that there does not exist a finite invariant for showing that the behaviour of the classical alternating bit protocol with reliable channels is independent of channel capacities. An interesting question is whether it would be possible to use infinite invariants in an analogous manner to the way that infinite processes are used by finite-state techniques in connection with fairness properties.

It would appear that the congruences related to simple behavioural properties like traces, deadlocks and livelocks have for the most part been identified. The question of finding characterisations for known congruences may still need some investigation. Also, as described above, it may be possible to identify the weakest congruences that preserve more complex properties of systems, such as those relating to fairness.

### 7.3 Summary

The process-algebraic approach is one way to specify and verify concurrent and reactive systems. This work has dealt with issues related to compositionality, which is one of the most important benefits of process algebra. We have identified new weakest compositional semantics, or congruences, that preserve the ability of systems to livelock and/or to deadlock. We showed how to use a class of fairness constraints in the compositional setting of process algebra to guarantee system progress. We have also made a case study where we showed by using compositional construction techniques that the external behaviour of a communication protocol is independent of the capacities of the underlying channels.
Bibliography


69
Weakest-Congruence Results for Livelock-Preserving Equivalences

Antti Puhakka & Antti Valmari

Tampere University of Technology, Software Systems Laboratory, PO Box 553, FIN-33101 Tampere, FINLAND, email: anpu@cs.tut.fi, ava@cs.tut.fi

Abstract. A behavioural equivalence is a congruence, if a system is guaranteed to remain equivalent when any one of its component processes is replaced by an equivalent component processes. An equivalence is weaker than another equivalence if the latter makes at least the same distinctions between systems as the former. An equivalence preserves a property, if no equivalence class contains one system that has that property and another system that lacks the property. Congruences that preserve such properties as deadlocks or livelocks are important in automatic verification of systems, and knowledge of the weakest such congruences is useful for designing verification algorithms. A simple denotational characterisation of the weakest deadlock-preserving congruence has been published in 1995. In this article simple characterisations are given to the weakest livelock-preserving congruence, and to the weakest congruence that preserves all livelocking traces. The results are compared to Hoare’s failures-divergences equivalence in the CSP theory.

1 Introduction

In this article we investigate weakest congruences for process-algebraic systems. A process algebra consists of a language for defining systems, and a semantic theory that defines one or more equivalences for the behaviours of systems. The language contains operators with which processes can be constructed and combined to form larger processes. An equivalence is a congruence, if and only if the replacement of a component process of a larger process with an equivalent component process always yields a result that is equivalent with the original larger process. Whether or not an equivalence is a congruence may depend on the set of operators that are allowed when constructing processes. An equivalence “≃₁” is weaker than another equivalence “≃₂” if and only if P ≃₂ Q implies P ≃₁ Q.

The research on weakest congruence results may have its origin in Robin Milner’s remark in p. 206 of [8]: “Hoare’s failures equivalence . . . is important, because it appears to be the weakest equivalence which never equates a deadlocking agent with one which does not deadlock.” Milner probably required that the equivalence must be a congruence, because otherwise the weakest equivalence would be the trivial one that has precisely two equivalence classes: the processes
(that is, Milner’s agents) that deadlock, and those that do not. In [10] it was proven that Milner’s guess was not precisely correct. The weakest deadlock-preserving congruence depends on the set of allowed process composition operators. Furthermore, assuming a reasonable choice of operators, it is the same as Hoare’s failures equivalence only in the absence of so-called divergence. From now on we will call Hoare’s equivalence CSP-equivalence to avoid confusion with some other important types of “failures” and “failures equivalences” that have appeared in the literature.

Another interesting weakest congruence result was proven in [5], where the so-called nondivergent failures divergences equivalence (NDFD-equivalence) was shown to be the weakest congruence that preserves the validity of formulae written in classic Manna-Pnueli linear time temporal logic [7] from which the “next state” operator “☐” has been removed. This logic is extremely important in verification of concurrent systems. Furthermore, if the congruence has to preserve also deadlocks, then the weakest congruence is the Chaos-free failures divergences (CFFD) equivalence. Because the Manna-Pnueli logic is state-based and process-algebraic equivalences are action-based, these results required an interpretation of the logic in an action-based setting. This can be done in more than one way. An alternative interpretation that is perhaps more relevant for practical verification than the original one was given in [11] (more easily found in [12] pp. 498–499).

Some researchers have tried to find the weakest congruence that preserves the results of certain kinds of tests. The solution with a fair way of testing was given by Brinksm, Rensink and Vogler in [2], and Leduc came to the conclusion that with another view to testing, the NDFD-equivalence is the solution [6].

Some equivalences investigated in weakest congruence research have their origin in [1].

In this article we are interested in weakest congruences that distinguish between diverging and non-diverging systems. Divergence is an important phenomenon, because it corresponds to livelock, and has perhaps been the biggest stumbling block in the quest of natural deadlock-preserving congruences. We also compare our results to the well-known CSP-equivalence.

Although the motivation of this article is mostly theoretical, weakest congruence results have also practical significance for automatic verification. One powerful way of fighting the well-known state explosion problem in automatic verification is compositional LTS construction, in which some reduction algorithm is applied to an LTS before using it as a component of a larger system. One way of guaranteeing that this approach produces correct results is to ensure that the reduction algorithm preserves some equivalence that is a congruence and that preserves the property in question. For instance, any reduction algorithm that preserves the weakest deadlock-preserving congruence can be used in compositional analysis of deadlocks.

Section 2 gives the earlier definitions, etc. that we will rely on in this article. The weakest congruence that preserves divergence traces is given in Section 3, and the weakest congruence that distinguishes between a diverging and
Let \( A^* \) denote the set of finite and \( A^\omega \) infinite strings of elements of a set \( A \). The empty string is denoted with \( \varepsilon \), and it is an element of \( A^* \), but not of \( A^\omega \). That a (finite or infinite) string \( \sigma \) is a prefix of a string \( \rho \) is denoted with \( \sigma \leq \rho \), and \( \sigma < \rho \) means that \( \sigma \leq \rho \wedge \sigma \neq \rho \). The length of the string \( \sigma \) is denoted with \( |\sigma| \).

The behaviour of a process consists of executing actions. There are two kinds of actions: visible and invisible. The invisible actions are denoted with a special symbol \( \tau \). The behaviour of a process is often represented as a labelled transition system. It is a directed graph whose edges are labelled with action names, with one state distinguished as the initial state of the process.

**Definition 1.** A labelled transition system, abbreviated LTS, is a four-tuple \((S, \Sigma, \Delta, \hat{s})\), where

- \( S \) is the set of states,
- \( \Sigma \), the alphabet, is the set of the visible actions of the process; we assume that \( \tau \notin \Sigma \),
- \( \Delta \subseteq S \times ((\Sigma \cup \{\tau\}) \times S \) is the set of transitions, and
- \( \hat{s} \in S \) is the initial state.

An LTS is finite if and only if its \( S \) and \( \Sigma \) are finite.

The following notation is useful for talking about the execution of a process starting at some given state. The \( \sim \)-notation requires that all actions along the execution path are listed, while the \( \tau \)-actions are skipped in the \( \Rightarrow \)-notation.

**Definition 2.** Let \((S, \Sigma, \Delta, \hat{s})\) be an LTS, \( s, s' \in S \), \( a, a_1, a_2, \ldots, a_n, \ldots \in \Sigma \cup \{\tau\} \), and \( b_1, b_2, \ldots, b_n, \ldots \in \Sigma \).

- \( s \rightarrow a \rightarrow s' \) is an abbreviation for \((s, a, s') \in \Delta \).
- \( s \rightarrow a_1 a_2 \cdots a_n \rightarrow s' \) means that there are \( s_0, s_1, \ldots, s_n \in S \) such that \( s_0 = s \), \( s_n = s' \) and \( s_{i-1} \rightarrow a_i \rightarrow s_i \) whenever \( 1 \leq i \leq n \).
- \( s \rightarrow a_1 a_2 \cdots a_n \rightarrow s' \) means that there is \( s' \) such that \( s \rightarrow a_1 a_2 \cdots a_n \rightarrow s' \).
- \( s \rightarrow a_1 a_2 a_3 \cdots \rightarrow s' \) means that there are \( s_0, s_1, s_2, \ldots \) such that \( s_0 = s \) and \( s_{i-1} \rightarrow a_i \rightarrow s_i \) whenever \( 1 \leq i \).
- \( \text{restr}(a_1 a_2 \cdots a_n, A) \), the restriction of \( a_1 a_2 \cdots a_n \) to \( A \), is the result of the removal of those \( a_i \) from \( a_1 a_2 \cdots a_n \) that are not in \( A \). The restriction of an infinite string is defined similarly.
- \( s \equiv \varepsilon \Rightarrow s' \) means that \( s \rightarrow \tau^n \rightarrow s' \) for some \( n \geq 0 \), where \( \tau^n \) denotes the sequence of \( n \) \( \tau \)-symbols.
− \( s = b_1 b_2 \cdots b_n \Rightarrow s' \) means that there are \( s_0, s_1, \ldots, s_n \in S \) such that \( s_0 = s, s_n = s' \) and \( s_{i-1} = b_i \Rightarrow s_i \) whenever \( 1 \leq i \leq n \). That is, \( s = b_1 b_2 \cdots b_n \Rightarrow s' \) if and only if there is \( \sigma \in (\Sigma \cup \{\tau\})^* \) such that \( s \sigma \Rightarrow s' \) and \( \text{restr}(\sigma, \Sigma) = b_1 b_2 \cdots b_n \).

− \( s = b_1 b_2 b_3 \cdots \Rightarrow \) means that there are \( s_0, s_1, s_2, \ldots \) such that \( s_0 = s \) and \( s_{i-1} = b_i \Rightarrow s_i \) whenever \( 1 \leq i \).

The semantic equivalences that we will discuss will use the following abstract sets extracted from an LTS. The traces of an LTS are the sequences of visible actions generated by any finite execution that starts in the initial state. An infinite execution that starts in the initial state generates either an infinite trace or a divergence trace, depending on whether the number of visible actions in the execution is infinite.

**Definition 3.** Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS.

− \( \text{Tr}(L) = \{ \sigma \in \Sigma^* \mid \hat{s} = \sigma \Rightarrow \} \) is the set of the traces of \( L \).

− \( \text{Inftr}(L) = \{ \xi \in \Sigma^* \mid \hat{s} = \xi \Rightarrow \} \) is the set of the infinite traces of \( L \).

− \( \text{Divtr}(L) = \{ \sigma \in \Sigma^* \mid \exists \xi : \hat{s} = \sigma \Rightarrow s \land s = \tau^\omega \Rightarrow \} \), where \( \tau^\omega \) denotes an infinite sequence of \( \tau \)-actions, is the set of the divergence traces of \( L \).

It is obvious that \( \text{Divtr}(L) \subseteq \text{Tr}(L) \) and, furthermore, if \( \xi \in \text{Inftr}(L) \) and \( \sigma < \xi \), then \( \sigma \in \text{Tr}(L) \). If an LTS (or just its set of states) is finite, then its infinite traces are determined by its ordinary traces, as was shown in [13], for instance.

**Proposition 1.** Let \( (S, \Sigma, \Delta, \hat{s}) \) be an LTS. If \( S \) is finite, then

\[ \text{Inftr}(L) = \{ \xi \in \Sigma^* \mid \forall \sigma : (\sigma < \xi \Rightarrow \sigma \in \text{Tr}(L)) \}. \]

We will later define some additional abstract sets. \( \text{Tr}, \text{Divtr} \) and \( \text{Inftr} \) are actually functions that take an LTS as input. Any collection of such functions can be used to define a semantic model of, and an equivalence between, LTSs as is shown below. Please notice that we will talk about an equivalence between two LTSs only if the LTSs have the same alphabet.

**Definition 4.** Let \( f_1, f_2, \ldots, f_k \) be any unary functions that take an LTS as their arguments.

− The semantic model of an LTS \( L \) induced by \( f_1, f_2, \ldots, f_k \) is the \( k \)-tuple \( (f_1(L), f_2(L), \ldots, f_k(L)) \).

− Assume that the LTSs \( L \) and \( L' \) have the same alphabet. The equivalence induced by \( f_1, f_2, \ldots, f_k \) is the equivalence “\( \equiv \)” defined as

\[ L \equiv L' \iff f_1(L) = f_1(L') \land f_2(L) = f_2(L') \land \cdots \land f_k(L) = f_k(L'). \]

We will call it the \( f_1\cdots f_k \)-equivalence.

Almost every process algebra contains some parallel composition operator. In this article we use the version which forces precisely those component processes to participate in the execution of a visible action that have that action in their
alphabets. The invisible action is always executed by one component process at a time. We first define the product of LTSs as the LTS that satisfies the above description and has the Cartesian product of component state sets as its set of states, and then define parallel composition by picking the part of the product that is reachable from the initial state of the product.

**Definition 5.** Let \( L_1 = (S_1, \Sigma_1, \Delta_1, s_1) \) and \( L_2 = (S_2, \Sigma_2, \Delta_2, s_2) \) be LTSs. Their product is the LTS \((S', \Sigma', \Delta', \bar{s})\) such that the following hold:
- \( S' = S_1 \times S_2 \)
- \( \Sigma = \Sigma_1 \cup \Sigma_2 \)
- \((s_1, s_2), a, (s'_1, s'_2)\) \(\in \Delta'\) if and only if either
  - \( a \in (\Sigma_1 \cup \{\tau\}) - \Sigma_2 \) and \((s_1, a, s'_1) \in \Delta_1 \land s'_2 = s_2\), or
  - \( a \in (\Sigma_2 \cup \{\tau\}) - \Sigma_1 \) and \((s_2, a, s'_2) \in \Delta_2 \land s'_1 = s_1\), or
  - \( a \in \Sigma_1 \cap \Sigma_2 \) and \((s_1, a, s'_1) \in \Delta_1 \) and \((s_2, a, s'_2) \in \Delta_2\).
- \( \bar{s} = (\bar{s}_1, \bar{s}_2) \)

The parallel composition \( L_1 || L_2 \) is the LTS \((S, \Sigma, \Delta, \bar{s})\) such that
- \( S = \{ s \in S' \mid \exists \sigma \in \Sigma^* : \bar{s} = \sigma \Rightarrow s \} \)
- \( \Delta = \Delta' \cap (S \times (\Sigma \cup \{\tau\}) \times S) \)

The following formulae describe the traces, etc. of a parallel composition as functions of the traces, etc. of its component processes. Their proofs are omitted because they basically consist of dull systematic checking against the definitions given above. Similar formulae can be found in the literature, for instance in [13].

**Proposition 2.** Let \( L_1 = (S_1, \Sigma_1, \Delta_1, s_1) \) and \( L_2 = (S_2, \Sigma_2, \Delta_2, s_2) \) be LTSs.
- \( \text{Tr}(L_1 || L_2) = \{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid \text{restr}(\sigma, \Sigma_1) \in \text{Tr}(L_1) \land \text{restr}(\sigma, \Sigma_2) \in \text{Tr}(L_2) \} \)
- \( \text{Divtr}(L_1 || L_2) = \{ \sigma \in \text{Tr}(L_1 || L_2) \mid \text{restr}(\sigma, \Sigma_1) \in \text{Divtr}(L_1) \lor \text{restr}(\sigma, \Sigma_2) \in \text{Divtr}(L_2) \} \)
- \( \text{Inftr}(L_1 || L_2) = \{ \xi \in (\Sigma_1 \cup \Sigma_2)^* \mid \text{restr}(\xi, \Sigma_1) \in \text{Inftr}(L_1) \land \text{restr}(\xi, \Sigma_2) \in \text{Tr}(L_2) \lor \text{restr}(\xi, \Sigma_1) \in \text{Tr}(L_1) \land \text{restr}(\xi, \Sigma_2) \in \text{Inftr}(L_2) \} \)

Another operator that is almost invariably found in process algebras in one form or another is hiding.

**Definition 6.** Let \( L = (S, \Sigma, \Delta, \bar{s}) \) be an LTS, and \( A \) any set of action names. The LTS hide \( A \) in \( L \) is the LTS \((S, \Sigma', \Delta', \bar{s}')\) such that the following hold:
- \( \Sigma' = \Sigma - A \)
- \( (s, a, s') \in \Delta' \) if and only if
  \( a = \tau \land \exists b \in A : (s, b, s') \in \Delta, \) or \( a \notin A \land (s, a, s') \in \Delta. \)

The traces, etc. of also hide \( A \) in \( L \) are functions of the traces, etc. of \( L \).
Proposition 3. Let \( L = (S, \Sigma, \Delta, s) \) be an LTS, and let \( \Sigma' \) be the alphabet of hide \( A \) in \( L \).

\[
\begin{align*}
\text{Tr}(\text{hide } A \text{ in } L) &= \{ \sigma \in \Sigma^* \mid \exists \rho \in \text{Tr}(L) : \sigma = \text{restr}(\rho, \Sigma') \} \\
\text{Divtr}(\text{hide } A \text{ in } L) &= \{ \sigma \in \Sigma^* \mid \exists \zeta \in \text{Divtr}(L) \cup \text{Inftr}(L) : \sigma = \text{restr}(\zeta, \Sigma') \} \\
\text{Inftr}(\text{hide } A \text{ in } L) &= \{ \xi \in \Sigma^\omega \mid \exists \zeta \in \text{Inftr}(L) : \xi = \text{restr}(\zeta, \Sigma') \}
\end{align*}
\]

An equivalence "\( \simeq \)" is a congruence with respect to a process operator \( op(L_1, \ldots, L_n) \) if and only if \( L_1 \simeq L_1' \land \cdots \land L_n \simeq L_n' \) implies \( op(L_1, \ldots, L_n) \simeq op(L_1', \ldots, L_n') \). We can reason from the above formulae that the \( \text{Tr-Divtr-Inftr} \)-equivalence is a congruence with respect to hiding and "\( || \)" [13]. Namely, if \( \text{Tr}(L) = \text{Tr}(L') \), \( \text{Divtr}(L) = \text{Divtr}(L') \), and \( \text{Inftr}(L) = \text{Inftr}(L') \), then

\[
\text{Divtr}(\text{hide } A \text{ in } L) = \{ \sigma \in \Sigma^* \mid \exists \zeta \in \text{Divtr}(L) \cup \text{Inftr}(L) : \sigma = \text{restr}(\zeta, \Sigma') \} \\
= \{ \sigma \in \Sigma^* \mid \exists \zeta \in \text{Divtr}(L') \cup \text{Inftr}(L') : \sigma = \text{restr}(\zeta, \Sigma') \} \\
= \text{divtr}(\text{hide } A \text{ in } L')
\]

where \( \Sigma \) is the common alphabet of \( L \) and \( L' \), and \( \Sigma' = \Sigma - A \). Similar reasoning applies to \( \text{Tr}(\text{hide } A \text{ in } L) \) and \( \text{Inftr}(\text{hide } A \text{ in } L) \). One can also immediately show with the same technique that \( \text{Tr}(L_1||L_2) = \text{Tr}(L_1'||L_2') \) etc., given that \( \text{Tr}(L_1) = \text{Tr}(L_1') \), etc.

In general, if \( f_1(op(L_1, \ldots, L_n)), f_2(op(L_1, \ldots, L_n)) \), \( \ldots \), \( f_k(op(L_1, \ldots, L_n)) \) can be represented as functions of \( f_1(L_1), f_2(L_1), \ldots, f_k(L_1), \ldots, f_1(L_n), f_2(L_n), \ldots, f_k(L_n) \), then the equivalence induced by \( f_1, f_2, \ldots, f_k \) is a congruence with respect to hiding to \( op \).

3 The Weakest Divergence-Trace-Preserving Congruence

The weakest divergence-trace-preserving congruence is the equivalence that preserves all divergence traces of a process, and is the weakest congruence with respect to hiding and "\( || \)" that has this property. In this section we will define eventually nondivergent infinite traces and then show that the equivalence induced by them together with traces and divergence traces is the weakest divergence-trace-preserving congruence. Eventually nondivergent infinite traces are those infinite traces, of whose prefixes only finitely many are divergence traces.

Definition 7. Let \( L = (S, \Sigma, \Delta, s) \) be an LTS. The set of the eventually nondivergent infinite traces of \( L \) is \( \text{Enditr}(L) = \text{Inftr}(L) \setminus \text{Divcl}(L) \), where

\[
\text{Divcl}(L) = \{ \xi \in \Sigma^\omega \mid \forall \sigma : (\sigma < \xi \Rightarrow \exists \sigma' : \sigma \leq \sigma' < \xi \land \sigma' \in \text{Divtr}(L)) \}.
\]

The abbreviation \( \text{Divcl} \) stands for “divergence closure”. The definition immediately implies that \( \text{Enditr}(L) \subseteq \text{Inftr}(L) \).

Proposition 4. Let \( L, L_1 \) and \( L_2 \) be LTSs, and let \( \Sigma_1, \Sigma_2 \) and \( \Sigma' \) be the alphabets of \( L_1, L_2 \) and hide \( A \) in \( L \).
1. \( \text{Divtr}(\text{hide } A \text{ in } L) = \{ \sigma \in \Sigma^* \mid \exists \zeta \in \text{Divtr}(L) \cup \text{Enditr}(L) : \sigma = \text{restr}(\zeta, \Sigma') \} \)

2. \( \text{Enditr}(\text{hide } A \text{ in } L) = \{ \xi \in \Sigma^* \mid \exists \zeta \in \text{Enditr}(L) : \xi = \text{restr}(\zeta, \Sigma') \} - \text{Divcl}(\text{hide } A \text{ in } L) \)

3. \( \text{Enditr}(L_1||L_2) = \{ \xi \in (\Sigma_1 \cup \Sigma_2)^* \mid \text{restr}(\xi, \Sigma_1) \in \text{Enditr}(L_1) \wedge \text{restr}(\xi, \Sigma_2) \in \text{Tr}(L_2) \cup \text{Enditr}(L_2) \vee \text{restr}(\xi, \Sigma_1) \in \text{Tr}(L_1) \wedge \text{restr}(\xi, \Sigma_2) \in \text{Enditr}(L_2) \} - \text{Divcl}(L_1||L_2) \)

4. The Tr-Divtr-Enditr-equivalence is a congruence with respect to hiding and \( || \).

Proof. Because Proposition 3 gives that \( \text{Divtr}(\text{hide } A \text{ in } L) = \{ \sigma \in \Sigma^* \mid \exists \zeta \in \text{Divtr}(L) \cup \text{Inftr}(L) : \sigma = \text{restr}(\zeta, \Sigma') \} \), and because \( \text{Enditr}(L) \subseteq \text{Inftr}(L) \), to prove 1 it suffices to show that whenever the strings in \( \text{Inftr}(L) - \text{Enditr}(L) \) contribute to \( \text{Divtr}(\text{hide } A \text{ in } L) \) would be in the latter set anyway. These strings have arbitrarily long prefixes that are divergence traces of \( L \). Let \( \zeta \in \text{Inftr}(L) - \text{Enditr}(L) \), and \( \sigma = \text{restr}(\zeta, \Sigma') \). If \( \sigma \) is infinite, then it is ruled out by the condition \( \sigma \in \Sigma^* \) in the right hand side of 1. Otherwise, \( \zeta \) has a finite prefix \( \zeta_1 \) such that \( \sigma = \text{restr}(\zeta_1, \Sigma') \). Because \( \zeta \in \text{Inftr}(L) - \text{Enditr}(L) \), \( \zeta \) has a prefix \( \zeta_2 \) such that \( \zeta_1 \leq \zeta_2 \) and \( \zeta_2 \in \text{Divtr}(L) \). We have \( \sigma = \text{restr}(\zeta_1, \Sigma') \leq \text{restr}(\zeta_2, \Sigma') \leq \text{restr}(\zeta, \Sigma') = \sigma \), so \( \sigma \) is included due to the part “\( \exists \zeta \in \text{Divtr}(L) : \ldots \)”. In a similar way one can show that if \( \zeta \in \text{Inftr}(L) - \text{Enditr}(L) \), then \( \text{restr}(\zeta, \Sigma') \in \text{Divcl}(\text{hide } A \text{ in } L) \) or \( \text{restr}(\zeta, \Sigma') \) is finite. This implies 2. The part 3 is proven similarly, and 4 follows from the previous parts and Propositions 2 and 3.

The next two propositions show that any equivalence that preserves the divergence traces and is a congruence with respect to “\( || \)” and hiding must preserve also ordinary traces and eventually nondivergent infinite traces.

Proposition 5. Let “\( \simeq \)” be a congruence with respect to “\( || \)” such that \( L \simeq L' \) implies \( \text{Divtr}(L) = \text{Divtr}(L') \). Then \( L \simeq L' \) implies \( \text{Tr}(L) = \text{Tr}(L') \).

Proof. Let \( L \simeq L' \), and let \( \Sigma \) be the common alphabet of \( L \) and \( L' \). Let \( \sigma = a_1a_2\ldots a_n \in \Sigma^* \), and let \( \text{Test}_1 \) be the LTS which has \( \Sigma \) as its alphabet, and whose states, transitions and initial state are as is shown in Figure 1. We have \( \sigma \in \text{Tr}(L) \iff \sigma \in \text{Divtr}(L||\text{Test}_1) \iff \sigma \in \text{Divtr}(L'||\text{Test}_1) \iff \sigma \in \text{Tr}(L') \), where the first and last logical equivalences are due to the structure of \( \text{Test}_1 \), and the middle one follows from the congruence requirement and that the equivalence preserves divergence traces.

Proposition 6. Let “\( \simeq \)” be a congruence with respect to “\( || \)” and hiding such that \( L \simeq L' \) implies \( \text{Divtr}(L) = \text{Divtr}(L') \). Then \( L \simeq L' \) implies \( \text{Enditr}(L) = \text{Enditr}(L') \).

Proof. Let \( L \simeq L' \), let \( \Sigma \) be the common alphabet of \( L \) and \( L' \), and \( \xi = a_1a_2a_3\ldots \in \text{Enditr}(L) \). Because \( \xi \in \text{Enditr}(L) \), it has a prefix \( a_1a_2\ldots a_n \) such
that every divergence trace of $L$ is either a prefix of $a_1a_2\cdots a_{n-1}$, or not a prefix of $\xi$. Let $a_{\text{new}}$ be a symbol that is not in $\Sigma$, and let $\text{Test}_2$ be the LTS that has $\Sigma \cup \{a_{\text{new}}\}$ as its alphabet, and whose states and transitions are as is shown in Figure 1. We see that $a_{\text{new}} \in \text{Divtr}(\text{hide } \Sigma \text{ in } (L||\text{Test}_2))$. The congruence requirement implies that $a_{\text{new}} \in \text{Divtr}(\text{hide } \Sigma \text{ in } (L'||\text{Test}_2))$. From this we can conclude that either $\xi \in \text{Inftr}(L')$, or $\text{Divtr}(L')$ contains some $\rho$ such that $a_1a_2\cdots a_n \leq \rho < \xi$. In the latter case $\text{Divtr}(L) = \text{Divtr}(L')$ would imply that $\rho$ is a divergence trace of $L$ that is a prefix of $\xi$ but not a prefix of $a_1a_2\cdots a_{n-1}$, which is in contradiction with the choice of $n$. So we see that $\xi \in \text{Inftr}(L')$ and, furthermore, $\xi \notin \text{Endtr}(L')$. These imply that $\xi \in \text{Endtr}(L')$. In conclusion, $\text{Endtr}(L) \subseteq \text{Endtr}(L')$. By replacing the roles of $L'$ and $L$ we see that also $\text{Endtr}(L') \subseteq \text{Endtr}(L)$.

Putting the results of this section together gives the following theorem.

**Theorem 1.** The Tr-Divtr-Enditr-equivalence is the weakest congruence with respect to “||” and hiding that preserves all divergence traces.

**Proof.** The two preceding propositions say that any congruence that preserves divergence traces implies the Tr-Divtr-Enditr-equivalence. On the other hand, because this equivalence is a congruence, it is the required weakest congruence. \hfill \square

The proof of Proposition 6 used an infinite LTS. Therefore, if we make the a priori assumption that all LTSs are finite, then the proposition cannot any more be used, at least not without a new proof. However, Proposition 5 remains valid in such a situation, because its proof did not assume infinite LTSs to be available. This fact allows us to show that Theorem 1 holds also if all LTSs are assumed to be finite, and even if hiding is removed from the set of operators with respect to which the equivalence must be a congruence.

**Theorem 2.** The Tr-Divtr-Enditr-equivalence is the weakest congruence between finite LTSs with respect to “||” and hiding that preserves all divergence traces. The claim remains valid if “and hiding” is removed.

**Proof.** Let “∞” preserve the divergence traces and be a congruence with respect to “||”. By Proposition 5 it preserves also the traces. Proposition 1 implies that if $L \simeq L'$, then $\text{Inftr}(L) = \{ \xi \in \Sigma^\omega \mid \forall \sigma : (\sigma < \xi \Rightarrow \sigma \in \text{Tr}(L)) \} = \{ \xi \in \Sigma^\omega \mid \forall \sigma : (\sigma < \xi \Rightarrow \sigma \in \text{Tr}(L')) \} = \text{Inftr}(L')$, so $\text{Endtr}(L) = \text{Endtr}(L')$.

80
The key message of the proof is that if the LTSs are finite, then the $Tr$-$Divtr$-$Enditr$-equivalence collapses to the $Tr$-$Divtr$-equivalence.

4 The Weakest Divergence-Preserving Congruence

In the previous section we started with the requirement that the congruence must preserve all divergence traces. In this section our starting point is weaker: we assume only that the congruence preserves the one bit of information that tells if the process can diverge or not. As a result, we will end up with a strictly weaker equivalence.

The equivalence will be built from those traces and infinite traces that do not have divergence traces as their prefixes, and from those divergence traces that do not have divergence traces as their proper prefixes.

**Definition 8.** Let $L$ be an LTS and $\Sigma$ its alphabet.

- $\text{diverges}(L) = \text{True}$ if and only if $Divtr(L) \neq \emptyset$. Otherwise $\text{diverges}(L) = \text{False}$.
- If $X \subseteq \Sigma^*$, then $\text{minimals}(X) = \{ \sigma \in X \mid \forall \rho : (\rho < \sigma \Rightarrow \rho \notin X) \}$
- The set of the minimal divergence traces of $L$ is $\text{Mindiv}(L) = \text{minimals}(Divtr(L))$
- The set of the extended divergence traces of $L$ is $Divext(L) = \{ \zeta \in \Sigma^* \cup \Sigma^\omega \mid \exists \rho : \rho \leq \zeta \land \rho \in Mindiv(L) \}$
- The set of the nondivergent traces of $L$ is $Ndtr(L) = Tr(L) - Divext(L)$
- The set of the nondivergent infinite traces of $L$ is $Ndinftr(L) = Inftr(L) - Divext(L)$

In analogy with the previous section, we need to show that the $Ndtr$-$Mindiv$-$Ndinftr$-equivalence is a congruence. We just present the formulae that give $Ndtr(\text{hide } A \text{ in } L)$, etc. as functions of $Ndtr(L)$, etc., and skip their (boring) proof.

**Proposition 7.** Let $L$ be an LTS, and $\Sigma' = \Sigma - A$.

- $\text{Ndtr}(\text{hide } A \text{ in } L) = \{ \sigma \in \Sigma'^* \mid \exists \rho \in Ndtr(L) : \sigma = \text{restr}(\rho, \Sigma') \} - \text{Divext}(L)$
- $\text{Mindiv(\text{hide } A \text{ in } L) = \text{minimals(\{ \sigma \in \Sigma'^* \mid \exists \zeta \in Mindiv(L) \cup Ndinftr(L) : \sigma = \text{restr}(\zeta, \Sigma') \})}}$
- $\text{Ndinftr(\text{hide } A \text{ in } L) = \{ \xi \in \Sigma^\omega \mid \exists \zeta \in Ndinftr(L) : \xi = \text{restr}(\zeta, \Sigma') \} - \text{Divext}(L)$
- $\text{Ndtr}(L_1 || L_2) = \{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid \text{restr}(\sigma, \Sigma_1) \in Ndtr(L_1) \land \text{restr}(\sigma, \Sigma_2) \in Ndtr(L_2) \}$
- $\text{Mindiv}(L_1 || L_2) = \text{minimals(\{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid \text{restr}(\sigma, \Sigma_1) \in Mindiv(L_1) \land \text{restr}(\sigma, \Sigma_2) \in Ndtr(L_2) \cup Mindiv(L_2) \lor \text{restr}(\sigma, \Sigma_1) \in Ndtr(L_1) \land \text{restr}(\sigma, \Sigma_2) \in Mindiv(L_2) \})}$
\[ Ndinftr(L_1) \cup Ndinftr(L_2) = \{ \xi \in (\Sigma_1 \cup \Sigma_2)^\omega \mid restr(\xi, \Sigma_1) \in Ndinftr(L_1) \land restr(\xi, \Sigma_2) \in Ndinftr(L_2) \cup Ndinftr(L_2) \pl intersections\}\]

That the \( Ndtr-Mindiv-Ndinftr \)-equivalence is the weakest diverges()-preserving congruence is a direct consequence of the following three propositions.

**Proposition 8.** Let \( \simeq \) be a congruence with respect to \( \| \| \) such that \( L \simeq L' \) implies that diverges\((L) = \) diverges\((L')\). Then \( L \simeq L' \) implies Mindiv\((L) = Mindiv(L')\).

**Proof.** Let \( \sigma = a_1a_2\ldots a_n \in Mindiv(L) \). Let Test\(_3\) have the same alphabet as \( L \), and let its other components be as is shown in Figure 2. Then diverges\((L||Test_3) = True\), so also diverges\((L'||Test_3) = True\), from which we can reason that \( \sigma \) has a prefix \( \rho \) such that \( \rho \in Mindiv(L') \). By repeating the argument with the roles of \( L \) and \( L' \) exchanged we see that \( \rho \) has a prefix \( \sigma' \) such that \( \sigma' \in Mindiv(L) \). Due to the definition of Mindiv we have \( \sigma' = \sigma \), so \( \rho = \sigma \) and we get Mindiv\((L) = Mindiv(L')\).

![Fig. 2. Two more LTSs used in proofs](image)

**Proposition 9.** Let \( \simeq \) be a congruence with respect to \( \| \| \) such that \( L \simeq L' \) implies that diverges\((L) = \) diverges\((L')\). Then \( L \simeq L' \) implies Ndtr\((L) = Ndtr(L')\).

**Proof.** Let \( \sigma = a_1a_2\ldots a_n \in Ndtr(L) \). We use again Test\(_1\) from Figure 1. We see that diverges\((L||Test_1) = True\), so also diverges\((L'||Test_1) = True\). This means that either \( \sigma \) has a prefix that is a divergence trace of \( L' \), or \( \sigma \in Ndtr(L') \). Proposition 8 and the definition of Ndtr\((L)\) rule out the former possibility. Thus \( \sigma \in Ndtr(L') \). So Ndtr\((L) \subseteq Ndtr(L') \). By symmetry also Ndtr\((L') \subseteq Ndtr(L) \).

**Proposition 10.** Let \( \simeq \) be a congruence with respect to \( \| \| \) and hiding such that \( L \simeq L' \) implies that diverges\((L) = \) diverges\((L')\). Then \( L \simeq L' \) implies Ndinftr\((L) = Ndinftr(L')\).

**Proof.** Let \( \xi = a_1a_2a_3\ldots \in Ndinftr(L) \). It is the time to use Test\(_4\) from Figure 2. Clearly hide \( \Sigma in \((L)||Test_4\) diverges, so also hide \( \Sigma in \((L'||Test_4\) must diverge. This is possible only if either some prefix of \( \xi \) is a divergence trace of \( L' \), or \( \xi \in Ndinftr(L') \). Like before, Proposition 8 and the definition of Ndinftr\((L)\) rule out the former possibility, so \( \xi \in Ndinftr(L') \). Like before, that \( \xi \in Ndinftr(L') \) implies \( \xi \in Ndinftr(L) \) follows now from symmetry.
Theorem 3. The Ndtr-Mindiv-Ndinftr-equivalence is the weakest congruence with respect to “||” and hiding that preserves the existence of divergence traces.

We again used an infinite LTS, namely in the proof of Proposition 10. Like in the previous section, this proposition can be replaced by another one if we restrict ourselves to finite LTSs. As a matter of fact, the weaker assumption that the LTSs are finitely branching can be used as well.

Definition 9. An LTS $(S, \Sigma, \Delta, \hat{s})$ is finitely branching, if and only if for each $s \in S$ and $a \in \Sigma \cup \{\tau\}$, the set $\{s' \mid (s, a, s') \in \Delta\}$ is finite.

Proposition 11. If $L$ is a finitely branching LTS, then
\[
\text{Ndinftr}(L) = \{\xi \in \Sigma^* \mid \forall \rho : (\rho < \xi \Rightarrow \rho \in \text{Ndtr}(L))\}.
\]

Proof. The direction “$\xi \in \text{Ndinftr}(L) \Rightarrow$” is obvious. For the opposite direction, let $\xi = a_1a_2a_3\ldots \in \Sigma^*$ such that $\forall \sigma : (\sigma < \xi \Rightarrow \sigma \in \text{Ndtr}(L))$. We show next that the system has an infinite execution $s_0 \xrightarrow{-b_1} \cdots \xrightarrow{-b_n} \cdots$, where $s_0 = \hat{s}$, such that $\text{restr}(b_1b_2\ldots, \Sigma) \leq \xi$. We do that by inductively demonstrating, for each $n \geq 0$, the existence of transitions $s_0 \xrightarrow{-b_1} \cdots \xrightarrow{-b_n} \cdots$, and an infinite set $E_n$ of arbitrarily long finite executions, such that the executions start with $s_0 \xrightarrow{-b_1} \cdots \xrightarrow{-b_n} s_n$, the traces of the executions are prefixes of $\xi$, and $s_0 = \hat{s}$.

A suitable $E_0$ is obtained by picking, for each $i \geq 0$, any execution that has $a_1a_2\ldots a_i$ as its trace. Of course, $s_0$ is chosen to be $\hat{s}$.

Because $E_n$ contains an infinite number of arbitrarily long executions, it contains infinitely many arbitrarily long executions that are longer than $n$. The $(n+1)$th transition of any such execution is labelled either with $\tau$, or with the $a_k$ such that $\text{restr}(b_1\ldots b_n, \Sigma) = a_1\ldots a_{k-1}$. Because the LTS is finitely branching, there are only finitely many $\tau$- and $a_k$-transitions that start in $s_n$. Thus infinitely many arbitrarily long members of $E_n$ must share the same $(n+1)$th transition $s_n \xrightarrow{-b_{n+1}} s_{n+1}$ (where $b_{n+1} = \tau$ or $b_{n+1} = a_k$). The set of those members can be chosen as $E_{n+1}$. This concludes the induction proof.

Our starting point included the assumption that all prefixes of $\xi$ are nondivergent traces. Therefore, $\text{restr}(b_1b_2\ldots, \Sigma)$ cannot be finite, because otherwise the infinite execution $s_0 \xrightarrow{-b_1} s_1 \xrightarrow{-b_2} \cdots$ would generate a divergence trace that is a prefix of $\xi$. As a consequence, $\text{restr}(b_1b_2\ldots, \Sigma) = \xi$, so $\xi \in \text{Ndinftr}(L)$.

Thus the Ndtr-Mindiv-Ndinftr-equivalence collapses to the Ndtr-Mindiv-equivalence if the LTSs are finitely branching.

Theorem 4. The Ndtr-Mindiv-equivalence is the weakest congruence between finite LTSs with respect to “||” and hiding that preserves the existence of divergence traces. The claim remains valid if “and hiding” is removed, and/or “finite” is replaced with “finitely branching.”
5 Comparison to CSP-Equivalence

Readers that are familiar with the CSP theory have certainly noticed that the $\text{Ndtr-Mindiv-Ndivtr}$-equivalence of the previous section has a striking similarity with the well-known failures-divergences equivalence of [3, 4, 9], and we will soon make this similarity explicit. As we hinted in the introduction, the meanings that this equivalence assigns to the terms “failure” and “divergence” are different from the meanings used elsewhere, so we prefer to call this equivalence CSP-equivalence to avoid confusion.

One important goal in the original definition of CSP-equivalence was to derive the meanings of recursive process equations directly — without first converting the processes to LTSs. In this way the need for an operational semantics was avoided. An analysis of the fixed points of process equations made such a definition possible. However, the definition had the consequence that no information of the behaviour of a process after it has executed a divergence trace is preserved by CSP-equivalence. A process that has executed a divergence trace is called Chaos in CSP literature. We assume in this section that all LTSs are finitely branching, because otherwise CSP-equivalence would not be a congruence ([9] p. 200).

CSP-equivalence can be defined in the LTS framework as follows ([9] p. 191). We use the additional concept of stable failure. A stable failure of an LTS is a pair consisting of a trace of that LTS and a subset of its alphabet. It is possible to execute that trace such that the LTS ends up in a state where it can execute neither invisible actions (the state is thus stable), nor any actions from the given subset. Stable failures or related concepts are important in equivalences that preserve deadlock information and are congruences with respect to “||”.

Definition 10. Let $L = (S, \Sigma, \Delta, \mathcal{s})$ be a finitely branching LTS.

- $\text{sfail}(L) = \{(\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : \mathcal{s} = \sigma \Rightarrow s \land \forall a \in A \cup \{\tau\} : \neg(s \xrightarrow{a})\}$ is the set of the stable failures of $L$.
- $\text{CSPdivtr}(L) = \{\sigma \in \Sigma^* \mid \exists \rho : \rho \leq \sigma \land \rho \in \text{Divtr}(L)\}$
- $\text{CSPfail}(L) = \text{sfail}(L) \cup (\text{CSPdivtr}(L) \times 2^\Sigma)$
- CSP-equivalence is the $\text{CSPfail}$-$\text{CSPdivtr}$-equivalence.

The following proposition implies that CSP-equivalence implies the $\text{Ndtr-Mindiv}$-equivalence.

Proposition 12. Let $L$ be a finitely branching LTS.

1. $\text{Ndtr}(L) = \{\sigma \mid (\sigma, \emptyset) \in \text{CSPfail}(L) \land \sigma \notin \text{CSPdivtr}(L)\}$
2. $\text{Mindiv}(L) = \text{minimals}(\text{CSPdivtr}(L))$

Proof. Let $\sigma \in \text{Ndtr}(L)$, and consider an arbitrary execution that produces $\sigma$. Because $\text{Ndtr}(L) \cap \text{Divtr}(L) = \emptyset$, any continuation of that execution with $\tau$-transitions eventually leads to a stable state. Thus $(\sigma, \emptyset) \in \text{sfail}(L) \subseteq \text{CSPfail}(L)$. The definitions of $\text{Ndtr}(L)$ and $\text{CSPdivtr}(L)$ give that $\sigma \notin \text{CSPdivtr}(L)$. On the other hand, if $(\sigma, \emptyset) \in \text{CSPfail}(L)$ and $\sigma \notin \text{CSPdivtr}(L)$, then $(\sigma, \emptyset) \in \text{sfail}(L)$, so $\sigma \in \text{Tr}(L)$ and $\sigma \in \text{Ndtr}(L)$. Part 1 has now been proven. Part 2 follows easily from the definitions. \qed
In the opposite direction, clearly $\text{CSPdivtr}(L) = \{ \sigma \in \Sigma^* \mid \exists \rho : \rho \leq \sigma \land \rho \in \text{Mindiv}(L) \}$. However, $\text{CSPfail}(L)$ cannot be obtained from $\text{Ndtr}(L)$, $\text{Mindiv}(L)$ and $\text{Ndinftr}(L)$, and the $\text{Ndtr-Mindiv-Ndinftr}$-equivalence does not imply CSP-equivalence. The first two LTSs in Figure 3 prove this.

![LTS Diagrams](image)

**Fig. 3.** Two pairs of $\text{Ndtr-Mindiv-Ndinftr}$-equivalent but not CSP-equivalent LTSs

It would be interesting to find a small strengthening to the starting point of the construction of the weakest divergence-preserving congruence such that the result would be precisely CSP-equivalence. Unfortunately, the task seems difficult. What $\text{Ndtr-Mindiv-Ndinftr}$-equivalence misses from CSP-equivalence is clearly related to the deadlock properties of LTSs. Unfortunately, it was shown in [10] that any equivalence that preserves the possibility of deadlocking and is a congruence with respect to "||" must preserve $\text{sfail}(L)$. CSP-equivalence preserves only those stable failures whose trace part has no divergence trace as a prefix. As a consequence, the requirement of deadlock-preservation would strengthen the equivalence too much. It seems that the requirement must somehow be formulated such that it, like CSP-semantics, does not say anything about the behaviour after executing a minimal divergence trace.

A seemingly promising possibility would be to seek for the weakest “any-lock”-preserving congruence, that is, the weakest congruence that distinguishes systems that can stop executing visible actions from those that cannot. There are two ways in which a system can stop executing visible actions: deadlock (the system cannot execute anything), and livelock or divergence (the system executes infinitely many invisible actions). The congruence needs not distinguish between these two reasons, unless the congruence requirement indirectly forces such a distinction to be possible. CSP-equivalence implies this congruence, because CSP-equivalence is itself a congruence, $\text{CSPdivtr}(L) = \emptyset$ if and only if $L$ has no divergences, and, in the case that $L$ has no divergences, $(\sigma, \Sigma) \in \text{CSPfail}(L)$ if and only if $\sigma$ is a trace that leads to a deadlock.

The proofs of Propositions 8, 9 and 10 can be carried through with the any-lock-preserving congruence by first adding a new action $a_{\text{new}}$ to the alphabets of $\text{Test}_3$, $\text{Test}_1$ and $\text{Test}_4$, and attaching a local $a_{\text{new}}$-loop (that is, the transition $s \xrightarrow{a_{\text{new}} \tau} s$) to each state of $\text{Test}_3$, $\text{Test}_1$ and $\text{Test}_4$. Furthermore, the LTSs $L_3$ and $L_4$ in Figure 3 are $\text{Ndtr-Mindiv-Ndinftr}$-equivalent but not any-lock-equivalent. As a consequence, the any-lock-preserving congruence is strictly stronger than the $\text{Ndtr-Mindiv-Ndinftr}$-equivalence. Indeed, with the $\text{Test}_5$ in Figure 4 one can prove that the any-lock-preserving congruence must preserve all stable failures $(a_1 a_2 \cdots a_n, \{b_1, \ldots, b_k\})$ such that no prefix of $a_1 a_2 \cdots a_n b$ is a divergence trace.
where $b$ is any element of $\{b_1, \ldots, b_k\}$. This is slightly less than what we want — we would like to allow $a_1a_2\cdots a_n b$ (although not $a_1a_2\cdots a_n$) to diverge.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{lts.png}
\caption{An LTS for testing failures}
\end{figure}

Unfortunately, the weakest any-lock-preserving congruence seems to be strictly weaker than CSP-equivalence. Namely, CSP-equivalence distinguishes between $L'_3 = \text{hide} \{a_{\text{new}}\} \text{ in } L_3$ and $L'_4 = \text{hide} \{a_{\text{new}}\} \text{ in } L_4$, but it seems that the weakest any-lock-preserving congruence does not. Both of these two LTSs can stop executing visible actions by first executing $a$ and then diverging. This remains true if the LTSs are put into an environment that hides $a$ or eventually offers $a$. If the environment refuses $a$ before offering it, then the systems can stop executing visible actions if and only if the environment can. It is thus difficult to think of an environment that would make it possible to distinguish between $L'_3$ and $L'_4$, when the only thing that can be observed is the ability of stopping executing visible actions.

Theorem 9.3.1 (iii) of [9] characterises CSP-equivalence as the weakest “immediate-any-lock-preserving” congruence, that is, as the weakest congruence that distinguishes systems that can deadlock or diverge before executing any visible actions from those that cannot. This result is not fully satisfactory from our point of view, because its proof uses a somewhat unusual “relational renaming” operator that can convert a transition to two transitions with different labels (see the errata that is in the www page of the book), whereas most of the other weakest congruence results rely only on ordinary parallel composition and hiding. Nevertheless, the result gives a characterisation of CSP-equivalence as the weakest congruence that satisfies a simple condition with respect to a well-known (large) set of operators.

6 Conclusions

We proved that the weakest livelock-preserving congruence is the $Ndtr-Mindiv-Ndinftr$-equivalence, and the weakest congruence that preserves all traces that lead to a livelock is the $Tr-Dieitr-Enditr$-equivalence. We proved that Hoare’s well-known CSP-equivalence implies the $Ndtr-Mindiv-Ndinftr$-equivalence but is not the same. As an attempt to give CSP-equivalence a characterisation as a weakest congruence we investigated the “any-lock”-preserving congruence, and
found it to be (apparently strictly) between the $Ndtr$-$Mindiv$-$Ndinftr$-equivalence and CSP-equivalence.

An interesting topic for future research would be to check how sensitive the results in this article are to the particular choice of operators. We state as a hypothesis that the $Tr$-$Divtr$-$Enditr$ and $Ndtr$-$Mindiv$-$Ndinftr$-equivalences remain congruences when other common process operators, such as choice, renaming, etc., are taken into account.

Acknowledgements  The comments by the anonymous referees were useful in finishing this paper. The work of A. Puhakka, the main author, was funded by the TISE graduate school and the Academy of Finland, project “Specification and Verification of Distributed Systems with Synchronous Actions”. The work of A. Valmari was funded by the Academy of Finland, project “Software Verification with CFFD-Semantics”.

References

Weakest Congruence Results Concerning “Any-Lock”

Antti Puhakka
Tampere University of Technology, Software Systems Laboratory,
PO Box 553, FIN-33101 Tampere, FINLAND,
email: anpu@cs.tut.fi

Abstract. In process algebras the weakest congruences that preserve interesting properties of systems are of theoretical and practical importance. A system can stop executing visible actions in two ways: by deadlocking or livelocking. The weakest deadlock-preserving congruence was published in [20]. The weakest livelock-preserving congruence and the weakest congruence that preserves all traces of visible actions leading to a livelock were published in [17]. In this paper we will equate deadlock and livelock. We introduce the weakest congruence that preserves the predicate “any-lock” which distinguishes those systems that can stop executing visible actions from those that cannot. We also present the weakest congruence that preserves all traces after which the system can stop executing visible actions. Finally, we give two simple weakest-congruence characterisations for the CSP failures-divergences equivalence, one of which is a minimal characterisation in a well-defined sense. However, we also show that there is no minimum (least) characterisation.

1 Introduction

Process algebras are one important approach for the specification and verification of concurrent systems. In a process algebra processes are constructed from other processes by using operators. A semantic equivalence tells whether the behaviour of two processes is the same with respect to the properties we are interested in. Such properties can be, for example, the presence of deadlocks or livelocks in the process. However, an equivalence is usually only deemed useful if it is a congruence. This means that when two equivalent processes, \( P \simeq Q \), are placed in any process context \( C[\cdot] \) constructed from operators and processes, the results have to be equivalent: \( C[P] \simeq C[Q] \). Whether or not an equivalence is a congruence may thus depend on the set of operators we are allowed to use.

An equivalence “\( \simeq_1 \)” is weaker (coarser) than another equivalence “\( \simeq_2 \)” if and only if \( P \simeq_2 Q \) implies \( P \simeq_1 Q \).

The weakest congruences that preserve certain interesting properties are of both theoretical and practical interest. These are the equivalences that contain just enough information, and nothing more, that is needed to deduce the given property of any system that can be constructed using our operators. For practical verification, congruences are important because they can be used in
compositional construction of LTSs (labelled transition systems). This means that a reduction algorithm which preserves the equivalence is applied to an LTS before using the LTS as a component in a larger system. The congruence property then guarantees that the resulting overall system is equivalent to the system without the reduction. That the equivalence is as weak as possible, on the other hand, means that it makes least distinctions between processes, thus giving us most opportunities for reducing processes.

A process communicates with its environment by executing visible actions, and it can stop executing visible actions in two ways. It can either deadlock, meaning that it cannot execute any actions, or livelock, meaning that it executes infinitely many internal actions. The weakest congruence that distinguishes deadlocking and non-deadlocking systems was identified in [20]. The weakest congruence that distinguishes livelocking and non-livelocking systems was found in [17]. Also, [17] presented the (strictly stronger than the former) weakest congruence that preserves all traces of visible actions leading to a livelock.

Based on these results on deadlock and livelock we know the weakest congruences that preserve both of these properties individually. However, sometimes the most important thing to know is whether or not the system can stop executing visible actions, not whether this happens through deadlock or livelock. In a typical situation, we want to make sure that our system has neither deadlocks nor livelocks.

In this paper we determine the weakest congruence that distinguishes systems that can stop executing visible actions from those that cannot, that is, preserves the predicate “deadlock-or-livelock”. This turns out to be strictly weaker than the congruence needed for preserving both of the above predicates. Also, in [17] it was conjectured that for finitely nondeterministic systems such a congruence would be strictly weaker than Hoare’s well-known CSP failures-divergences equivalence [19]. Here we prove that this is indeed the case. We also present the weakest congruence that preserves all traces of visible actions after which the system can stop executing further visible actions, i.e., deadlock or livelock. Finally, we give two simple models for which CSP-equivalence is the weakest preserving congruence. One of these is an optimal or, minimal, characterisation in an intuitively defined sense. However, we also show that there is no unique minimum characterisation.

In the next section we define our framework which is a CSP-like process algebra with LTSs as processes. The weakest any-lock-preserving congruence is given in Section 3, and the corresponding weakest congruence for any-lock-traces in Section 4. In Section 5 the CSP failures-divergences-equivalence is given the above-mentioned characterisations. In Section 6 we describe related work, and in Section 7 we present some conclusions.

2 Background

Let $A^*$ denote the set of finite and $A^\omega$ infinite strings of elements of a set $A$. The empty string is denoted with $\varepsilon$, and it is an element of $A^*$, but not of $A^\omega$. 
That a (finite) string $\sigma$ is a prefix of a (finite or infinite) string $\rho$ is denoted with $\sigma \leq \rho$, and $\sigma < \rho$ means that $\sigma \leq \rho \land \sigma \neq \rho$.

The behaviour of a process consists of executing actions. There are two kinds of actions: visible and invisible. The invisible actions are denoted with a special symbol $\tau$. Here the behaviour of a process is represented as a labelled transition system. It is a directed graph whose edges are labelled with action names, with one state distinguished as the initial state of the process.

Definition 1. A labelled transition system, abbreviated LTS, is a four-tuple $(S, \Sigma, \Delta, s)$, where

- $S$ is the set of states,
- $\Sigma$, the alphabet, is the set of the visible actions of the process; we assume that $\tau \notin \Sigma$,
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is the set of transitions, and
- $s \in S$ is the initial state.

Definition 2. An LTS $(S, \Sigma, \Delta, s)$ is finitely nondeterministic (finitely branching), if and only if for each $s \in S$ and $a \in \Sigma \cup \{\tau\}$, the set $\{s' \mid (s, a, s') \in \Delta\}$ is finite. An LTS $(S, \Sigma, \Delta, s)$ is finite if and only if $S$ and $\Sigma$ are finite.

We use $s \xrightarrow{a} s'$ as an abbreviation for $(s, a, s') \in \Delta$, and this is extended in the obvious way to $s \xrightarrow{\sigma} s'$ and $s \xrightarrow{\tau}$, where $\sigma$ is a finite and $\xi$ a finite or infinite sequence of actions. Let $\text{restr}(\sigma, A)$ denote the result of removal of all actions from $\sigma$ that are not in $A$. We write $s \xrightarrow{\rho} s'$ iff there is $\sigma$ such that $s \xrightarrow{\sigma} s'$ and $\rho = \text{restr}(\sigma, \Sigma)$. $s \xrightarrow{\rho}$ is defined similarly.

The semantic equivalences that we will discuss will use the following abstract sets extracted from an LTS. The traces of an LTS are the sequences of visible actions generated by any finite execution that starts in the initial state. An infinite execution that starts in the initial state generates either an infinite trace or a divergence trace, depending on whether the number of visible actions in the execution is infinite. The stable failures describe the ability of the LTS to refuse actions after executing a particular trace.

Definition 3. Let $L = (S, \Sigma, \Delta, s)$ be an LTS.

- $\text{Tr}(L) = \{ \sigma \in \Sigma^* \mid s \xrightarrow{\sigma} s' \}$ is the set of the traces of $L$.
- $\text{Inftr}(L) = \{ \xi \in \Sigma^* \mid s \xrightarrow{\xi} s' \}$ is the set of the infinite traces of $L$.
- $\text{Divtr}(L) = \{ \sigma \in \Sigma^* \mid \exists s : s \xrightarrow{\sigma} s \land s \xrightarrow{\tau^\omega} \}$, where $\tau^\omega$ denotes an infinite sequence of $\tau$-actions, is the set of the divergence traces of $L$.
- $\text{Sfail}(L) = \{ (\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : s \xrightarrow{\sigma} s \land \forall a \in A \cup \{\tau\} : \neg s \xrightarrow{a} \}$
  is the set of the stable failures of $L$.

It is obvious that $\text{Divtr}(L) \subseteq \text{Tr}(L)$ and, furthermore, if $\xi \in \text{Inftr}(L)$ and $\sigma < \xi$, then $\sigma \in \text{Tr}(L)$. If an LTS (or just its set of states) is finite, then its infinite traces are determined by its ordinary traces, as was shown in [23].

Proposition 1. Let $L = (S, \Sigma, \Delta, s)$ be an LTS. If $S$ is finite, then

$\text{Inftr}(L) = \{ \xi \in \Sigma^* \mid \forall \sigma < \xi : \sigma \in \text{Tr}(L) \}$. 
It is also easy to show that $\text{Tr}(L)$ can be determined if $\text{Divtr}(L)$ and $\text{Sfail}(L)$ are known:

**Proposition 2.** Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS. Then

$$\text{Tr}(L) = \text{Divtr}(L) \cup \{ \sigma \in \Sigma^* \mid (\sigma, \emptyset) \in \text{Sfail}(L) \}.$$ 

We will later define some additional abstract sets. $\text{Tr}$, $\text{Divtr}$, $\text{Inftr}$ and $\text{Sfail}$ are actually functions that take an LTS as input. Any collection of such functions can be used to define a semantic model of, and an equivalence between, LTSs as is shown below. Please notice that we will talk about an equivalence between two LTSs only if the LTSs have the same alphabet.

**Definition 4.** Let $f_1, f_2, \ldots, f_k$ be any unary functions that take an LTS as their arguments.

- The semantic model of an LTS $L$ induced by $f_1, f_2, \ldots, f_k$ is the $k$-tuple $(f_1(L), f_2(L), \ldots, f_k(L))$.
- Assume that the LTSs $L$ and $L'$ have the same alphabet. The equivalence induced by $f_1, f_2, \ldots, f_k$ is the equivalence “$\equiv$” defined as

$$L \equiv L' \iff f_1(L) = f_1(L') \land f_2(L) = f_2(L') \land \cdots \land f_k(L) = f_k(L').$$

We will call it the $f_1$-$f_2$-$\ldots$-$f_k$-equivalence.

Almost every process algebra contains some parallel composition operator. In this article we use the version which forces precisely those component processes to participate in the execution of a visible action that have that action in their alphabets. The invisible action is always executed by exactly one component process at a time. We first define the product of LTSs as the LTS that satisfies the above description and has the Cartesian product of component state sets as its set of states, and then define parallel composition by picking the part of the product that is reachable from the initial state of the product.

**Definition 5.** Let $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ be LTSs. Their product is the LTS $(S', \Sigma, \Delta', \hat{s})$ such that the following hold:

- $S' = S_1 \times S_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $((s_1, s_2), a, (s_1', s_2')) \in \Delta'$ if and only if either
  - $a \in (\Sigma_1 \cup \{\tau\}) \setminus \Sigma_2$ and $(s_1, a, s_1') \in \Delta_1 \land s_2' = s_2$, or
  - $a \in (\Sigma_2 \cup \{\tau\}) \setminus \Sigma_1$ and $(s_2, a, s_2') \in \Delta_2 \land s_1' = s_1$, or
  - $a \in \Sigma_1 \cap \Sigma_2$ and $(s_1, a, s_1') \in \Delta_1$ and $(s_2, a, s_2') \in \Delta_2$.
- $\hat{s} = (\hat{s}_1, \hat{s}_2)$

The parallel composition $L_1 || L_2$ is the LTS $(S, \Sigma, \Delta, \hat{s})$ such that

- $S = \{ s \in S' \mid \exists \alpha \in \Sigma^* : \hat{s} = \alpha \Rightarrow s \}$
- $\Delta = \Delta' \cap (S \times (\Sigma \cup \{\tau\}) \times S)$
The following formulae describe the traces, etc. of a parallel composition as functions of the traces, etc. of its component processes. Their proofs are omitted because they basically consist of tedious systematic checking against the definitions given above. Similar formulae can be found in the literature, for instance in [23].

**Proposition 3.** Let $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ be LTSs.

- $Tr(L_1 || L_2) = \{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid restr(\sigma, \Sigma_1) \in Tr(L_1) \land restr(\sigma, \Sigma_2) \in Tr(L_2) \}$
- $Divtr(L_1 || L_2) = \{ \sigma \in Tr(L_1) || L_2 \mid restr(\sigma, \Sigma_1) \in Divtr(L_1) \lor restr(\sigma, \Sigma_2) \in Divtr(L_2) \}$
- $Infr(L_1 || L_2) = \{ \xi \in (\Sigma_1 \cup \Sigma_2)^* \mid restr(\xi, \Sigma_1) \in Infr(L_1) \land restr(\xi, \Sigma_2) \in Infr(L_2) \} \lor restr(\xi, \Sigma_1) \in Infr(L_1) \land restr(\xi, \Sigma_2) \in Tr(L_2) \}$
- $Sfail(L_1 || L_2) = \{ (\sigma, A) \in (\Sigma_1 \cup \Sigma_2)^* \times 2^{\Sigma_1 \cup \Sigma_2} \mid \exists (b, B_1) \in Sfail(L_1), (b, B_2) \in Sfail(L_2) : restr(\sigma, \Sigma_1) = \rho_1 \land restr(\sigma, \Sigma_2) = \rho_2 \land A = B_1 \cup B_2 \}$

Another operator that is almost invariably found in process algebras in one form or another is **hiding**, which converts visible actions into $\tau$-actions and removes them from the alphabet.

**Definition 6.** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS, and $A$ any set of action names. Then **hide $A$ in $L$** is the LTS $(S, \Sigma', \Delta', \hat{s})$ such that the following hold:

- $\Sigma' = \Sigma \setminus A$
- $(s, a, s') \in \Delta' \text{ if and only if } a = \tau \land \exists b \in A : (s, b, s') \in \Delta, \text{ or } a \notin A \land (s, a, s') \in \Delta.$

The traces, etc. of also **hide $A$ in $L$** are functions of the traces, etc. of $L$.

**Proposition 4.** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS, and let $\Sigma'$ be the alphabet of hide $A$ in $L$.

- $Tr(hide A in L) = \{ \sigma \in \Sigma'^* \mid \exists \rho \in Tr(L) : \sigma = restr(\rho, \Sigma') \}$
- $Divtr(hide A in L) = \{ \sigma \in \Sigma'^* \mid \exists \zeta \in Divtr(L) \cup Infr(L) : \sigma = restr(\zeta, \Sigma') \}$
- $Infr(hide A in L) = \{ \xi \in \Sigma'^* \mid \exists \zeta \in Infr(L) : \xi = restr(\zeta, \Sigma') \}$
- $Sfail(hide A in L) = \{ (\sigma, B) \in \Sigma'^* \times 2^{\Sigma'} \mid \exists (\rho, C) \in Sfail(L) : \sigma = restr(\rho, \Sigma') \land C = B \cup A \}$

An equivalence “$\simeq$” preserves $f_1, \ldots, f_k$ if and only if “$L \simeq L'$” implies $f_i(L) = f_i(L') \land \cdots \land f_k(L) = f_k(L')$, that is, iff “$\simeq$” is stronger than the $f_1 \cdots f_k$-equivalence.

An equivalence “$\simeq$” is a **congruence** with respect to a process operator $op(L_1, \ldots, L_n)$ if and only if $L_1 \simeq L_1' \land \cdots \land L_n \simeq L_n'$ implies $op(L_1, \ldots, L_n) \simeq op(L_1', \ldots, L_n')$. As a simple example, we can reason from the above formulae that the $Tr$-equivalence is a congruence with respect to “$||$” and “hide”.
Namely, if \(\text{Tr}(L) = \text{Tr}(L')\), then because \(\text{Tr}(\text{hide } A \text{ in } L)\) is a function of \(A\) and \(\text{Tr}(L)\) only, then necessarily also \(\text{Tr}(\text{hide } A \text{ in } L) = \text{Tr}(\text{hide } A \text{ in } L')\). The same reasoning shows that \(\text{Tr}(L_1||L_2) = \text{Tr}(L'_1||L'_2)\) given that \(\text{Tr}(L_1) = \text{Tr}(L'_1),\ \text{Tr}(L_2) = \text{Tr}(L'_2)\).

In general, if \(f_1(op(L_1, \ldots, L_n)), \ldots, f_k(op(L_1, \ldots, L_n))\) can be represented as functions of \(f_1(L_1), \ldots, f_k(L_1), \ldots, f_1(L_n), \ldots, f_k(L_n)\), then the equivalence induced by \(f_1, \ldots, f_k\) is a congruence with respect to \(op\).

3 The Weakest Any-Lock-Preserving Congruence

There are two ways in which a process can stop executing visible actions: it can either deadlock or livelock. A deadlock means that the system is in a state where no actions are possible. For dealing with deadlocks we need the notion of stable failures. As a remainder, the set \(S\text{fail}(L)\) contains the pair \((\sigma, A)\) exactly when \(L\) can execute the trace \(\sigma\) and end up in a state that has neither invisible actions (the state is thus stable) nor any actions from the subset \(A\) of the alphabet as outgoing actions.

Deadlock traces are the traces of visible actions after which the system can deadlock.

**Definition 7.** Let \(L = (S, \Sigma, \Delta, \hat{s})\) be an LTS. The set of the deadlock traces of \(L\) is \(D\text{ltr}(L) = \{ \sigma \in \Sigma^* \mid (\sigma, \Sigma) \in S\text{fail}(L) \}\)

The following predicate tells whether or not there are any deadlocks in \(L\):

**Definition 8.** Let \(L\) be an LTS. Then

\[
\text{deadlock}(L) :\iff D\text{ltr}(L) \neq \emptyset.
\]

In [20] it was shown that the \(S\text{fail}\)-equivalence is the weakest equivalence that preserves deadlock and is a congruence with respect to “||” and “\text{hide}”. Since the \(S\text{fail}\)-equivalence preserves \(D\text{ltr}\), this means that it is also the weakest congruence that preserves all deadlock traces.

A livelock/divergence means that the system can execute infinitely many invisible \(\tau\)-actions, i.e., engage in an endless internal computation. For describing divergences we use the notion of divergence traces, defined in the previous section. The following predicate tells whether or not \(L\) can diverge:

**Definition 9.** Let \(L\) be an LTS. Then

\[
\text{diverge}(L) :\iff D\text{ivtr}(L) \neq \emptyset.
\]

For dealing with divergences, three new semantic sets were defined in [17]: the traces and infinite traces that do not have divergence traces as their prefixes, and the divergence traces that do not have divergence traces as their proper prefixes. These sets are defined formally as follows:

**Definition 10.** Let \(L\) be an LTS and \(\Sigma\) its alphabet.

\[-\text{ If } X \subseteq \Sigma^*, \text{ then } \text{minimals}(X) = \{ \sigma \in X \mid \forall \rho \in X : \rho \not< \sigma \}\}
\]

\[-\text{ The set of the minimal divergence traces of } L \text{ is } \text{Mindiv}(L) = \text{minimals}(D\text{ivtr}(L))\]

96
Fig. 1. Four example processes

- The set of the extended divergence traces of $L$ is
  \[ \text{Divext}(L) = \{ \zeta \in \Sigma^* \cup \Sigma^\omega \mid \exists \rho \in \text{Mindiv}(L) : \rho \leq \zeta \} \]
- The set of the nondivergent traces of $L$ is
  \[ \text{Ndtr}(L) = \text{Tr}(L) \setminus \text{Divext}(L) \]
- The set of the nondivergent infinite traces of $L$ is
  \[ \text{Ndinftr}(L) = \text{Inftr}(L) \setminus \text{Divext}(L) \]

In [17] it was shown that the $\text{Ndtr}$-$\text{Mindiv}$-$\text{Ndinftr}$-equivalence is the weakest congruence with respect to “||” and “hide” that preserves the existence of divergences, that is, preserves the predicate $\text{diverge}$.

In this article we will consider the combined effect of deadlocks and livelocks. It is easy to see that $\text{Ndtr}(L) = \{ \sigma \in \Sigma^* \mid (\sigma, \emptyset) \in S_{\text{fail}}(L) \} \setminus \text{Divext}(L)$, so the set $\text{Ndtr}(L)$ can be determined from $S_{\text{fail}}(L)$ and $\text{Mindiv}(L)$ and becomes redundant in the presence of these two sets. Combined with the above-mentioned result for the $S_{\text{fail}}$-equivalence this means that the $\text{Mindiv}$-$\text{Ndinftr}$-$S_{\text{fail}}$-equivalence is the weakest congruence with respect to “||” and “hide” that preserves both the deadlock- and $\text{diverge}$-predicates.

However, in some cases we only need to know whether or not the system can stop executing visible actions, not whether this happens by deadlocking or livelocking. Typically, we want to make sure that our system has neither deadlocks nor divergences. In such a situation the above requirement of preserving both predicates is unnecessarily strong. Instead, we would like to know the weakest congruence that preserves the information as to whether there are deadlocks or divergences in the system, i.e., the weakest congruence that distinguishes systems that can stop executing visible actions from those that cannot. Determining this equivalence is our goal in this section.

**Definition 11.** Let $L$ be an LTS. Then

\[ \text{anylock}(L) :\iff \text{deadlock}(L) \lor \text{diverge}(L). \]

**Example 1.** To illustrate the difference between the predicates, let us consider the simple example shown in Figure 1. Process $P_1$ could be, for example, the behaviour of a simple communication protocol. The other $P_i$ differ from $P_1$ by having deadlocks and/or divergences. It holds that $\text{deadlock}(P_1) = \text{deadlock}(P_3) = \text{False}$, but $\text{deadlock}(P_2) = \text{deadlock}(P_4) = \text{True}$. On the other hand, $\text{diverge}(P_1) = \text{diverge}(P_3) = \text{False}$, $\text{diverge}(P_2) = \text{diverge}(P_4) = \text{True}$. However, only $\text{anylock}(P_1) = \text{False}$, and $\text{anylock}(P_2) = \text{anylock}(P_3) = \text{anylock}(P_4) = \text{True}$. (Note, however, that the congruence developed below will need to distinguish all these LTSs except $P_3$ and $P_4$.)
It turns out, unsurprisingly, that we do not have to preserve all of $S_{\text{fail}}$ to preserve anylock. Let us define the following subset of $S_{\text{fail}}$:

**Definition 12.** Let $L = (S, \Sigma, \Delta, s)$ be an LTS. The set of the divergence-free failures of $L$ is $D_{\text{fail}}(L) = \{ (\sigma, A) \in S_{\text{fail}}(L) \mid \forall a \in A : \sigma a \notin Mindiv(L) \} \setminus D_{\text{ext}}(L) \times 2^{\Sigma}$.

Perhaps the most interesting point about this definition is that we not only reject those pairs where a prefix of $\sigma$ is a divergence trace, but we also do not include in $A$ actions that lead to a divergence after $\sigma$. Intuitively, the idea is that if the environment offers such an action, this automatically creates a divergence. Therefore, because we make no distinction between deadlock and divergence, it does not matter if refusing the action also creates a deadlock.

We next present the functions that give the resulting $D_{\text{fail}}$-set as a function of $Mindiv$ and the $D_{\text{fail}}$ sets of the parameters of the “$|$” and “$\text{hide}$” operators. The functions that give the $Mindiv$, $Ndtr$ and $Ndinfra$ sets as functions of the corresponding sets of the parameters have been presented in [17].

**Proposition 5.** Let $L_1$ and $L_2$ be LTSs with alphabets $\Sigma_1$ and $\Sigma_2$. Then $D_{\text{fail}}(L_1||L_2) = \{ (\sigma, A) \in (\Sigma_1 \cup \Sigma_2)^* \times 2^{\Sigma_1 \cup \Sigma_2} \mid \exists (\rho_1, B_1) \in D_{\text{fail}}(L_1), (\rho_2, B_2) \in D_{\text{fail}}(L_2) : \text{restrict}(\sigma, \Sigma_1) = \rho_1 \land \text{restrict}(\sigma, \Sigma_2) = \rho_2 \land \forall a \in A : \sigma a \notin Mindiv(L_1||L_2) \} \setminus D_{\text{ext}}(L_1||L_2) \times 2^{\Sigma_1 \cup \Sigma_2}$.

**Proof.** Assume $(\sigma, A) \in D_{\text{fail}}(L_1||L_2) \subseteq S_{\text{fail}}(L_1||L_2)$. By Proposition 3 there are $(\rho_1, C_1) \in S_{\text{fail}}(L_1)$ and $(\rho_2, C_2) \in S_{\text{fail}}(L_2)$ such that $\text{restrict}(\sigma, \Sigma_1) = \rho_1$, $\text{restrict}(\sigma, \Sigma_2) = \rho_2$ and $A = C_1 \cup C_2$. Next we will divide each of the refusal sets $C_1$, $C_2$ into two disjoint parts. Namely, let $C_1^i = \{ a \in C_1 \mid a \notin Mindiv(L_1) \}$ and $C_2^i$ similarly, and let $C_1^c = C_1 \setminus C_1^i$ and $C_2^c = C_2 \setminus C_2^i$. Firstly, if a prefix of $\rho_1$ were in $D_{\text{ext}}(L_1)$, then because $\rho_2 \in \operatorname{Tr}(L_2)$, we see by Proposition 3 that a prefix of $\sigma$ would be in $D_{\text{ext}}(L_1||L_2)$, which contradicts the assumption that $(\sigma, A) \in D_{\text{fail}}(L_1||L_2)$. Then, by the definition of $C_1^i$, we can conclude that $(\rho_1, C_1^i) \in D_{\text{fail}}(L_1)$. By symmetry, $(\rho_2, C_2^i) \in D_{\text{fail}}(L_2)$. Secondly, we will show that for all $a \in C_1^i$ (in process $L_1$) it holds that $a \in \Sigma_2$ but $\rho_2 a \notin \operatorname{Tr}(L_2)$ (in process $L_2$). If this were not true, then either $a \notin \Sigma_2$ or $\rho_2 a \in \operatorname{Tr}(L_2)$. By definition, $\rho_2 a \in Mindiv(L_1)$, so in both cases it would hold that $\rho a \in D_{\text{ext}}(L_1||L_2)$, which contradicts the original assumption. Thus, the claim has to hold, and $L_2$ can refuse $a$ in every stable state reachable with $\rho_2$. Because furthermore $\rho_2 a \notin Mindiv(L_2)$ (which is a subset of $\operatorname{Tr}(L_2)$), we can conclude that $(\rho_2, C_2^c \cup C_1^i) \in D_{\text{fail}}(L_2)$. By symmetry, $(\rho_1, C_1^c \cup C_2^i) \in D_{\text{fail}}(L_1)$. We can now use $(C_1^c \cup C_2^c)$ and $(C_1^i \cup C_2^i)$ as the sets $B_1$ and $B_2$ in the above formula, because $(C_1^c \cup C_2^c) \cup (C_1^i \cup C_2^i) = (C_1 \cup C_2) \cup (C_1^c \cup C_2^c) = C_1 \cup C_2 = A$. The other direction of the proof is a straightforward consequence of Proposition 3 and the definition of $D_{\text{fail}}$. \[\square\]

**Proposition 6.** Let $L$ be an LTS, and $\Sigma' = \Sigma \setminus A$. Then $D_{\text{fail}}(\text{hide } A \text{ in } L) = \{ (\sigma, B) \in \Sigma'^* \times 2^{\Sigma'} \mid \exists (\rho, C) \in D_{\text{fail}}(L) : \text{restrict}(\rho, \Sigma') = \sigma \land C = B \cup A \land \forall a \in B : \sigma a \notin Mindiv(\text{hide } A \text{ in } L) \} \setminus D_{\text{ext}}(\text{hide } A \text{ in } L)$.
Proof. By Proposition 4 and the definition of \( \text{Dfail} \), the above formula is clearly correct when “\( \text{Dfail}(L) \)” on the right hand side of the equation is replaced by “\( \text{Sfail}(L) \)”. It then remains to show that the set \( \text{Sfail}(L) \setminus \text{Dfail}(L) \) contributes nothing to the result. For each pair \( (\rho, C) \) in this set, a prefix of \( \rho \) or \( \rho a \), for some \( a \in C \), is in \( \text{Mindiv}(L) \). It follows that a prefix of \( \sigma \) or \( \sigma a \) is in \( \text{Mindiv}(\text{hide } A \text{ in } L) \). In the former case, the pair is removed by “\( \text{Divext}(\ldots) \)”.
In the latter case, \( a \notin A \) and because \( C = B \cup A \), \( a \in B \) and the pair is removed by the condition “\( \forall a \in B : \sigma a \notin \text{Mindiv}(\ldots) \)”.

The following propositions show that the congruence requirement forces us to preserve each of the sets \( \text{Mindiv} \), \( \text{Ndinftr} \), \( \text{Dfail} \). The technique we use is to devise contexts in which processes which differ with respect to any of the above sets would result in \( \text{anylock} \)-different results.

**Proposition 7.** Let “\( \simeq \)” be a congruence with respect to “\( \models \)” such that \( L \simeq L' \)
implies \( \text{anylock}(L) = \text{anylock}(L') \). Then \( L \simeq L' \) implies \( \text{Mindiv}(L) = \text{Mindiv}(L') \).

**Proof.** Let \( L \simeq L' \) and let \( \Sigma \) be the common alphabet of \( L \) and \( L' \). Assume \( \sigma \in \text{Mindiv}(L) \), where \( \sigma = a_1a_2\ldots a_n \). Let \( a_{\text{new}} \) be a novel action not in \( \Sigma \) and let \( \text{Test}_1 \) be the LTS with alphabet \( \Sigma \cup \{a_{\text{new}}\} \) and with other components as shown in Figure 2. Then \( \sigma \in \text{Dextr}(L\mid\text{Test}_1) \), so \( \text{anylock}(L\mid\text{Test}_1) = \text{True} \) and, by the congruence requirement, \( \text{anylock}(L'\mid\text{Test}_1) = \text{True} \). Because of the \( a_{\text{new}} \)-loops it clearly holds that \( \text{deadlock}(L'\mid\text{Test}_1) = \text{False} \), so there is a divergence in \( L'\mid\text{Test}_1 \). Thus, some prefix \( p \) of \( \sigma \) is in \( \text{Mindiv}(L') \). By repeating the argument with the roles of \( L \) and \( L' \) reversed we see that \( \rho \) has a prefix \( \sigma' \) such that \( \sigma' \in \text{Mindiv}(L) \). Due to the definition of \( \text{Mindiv} \) we have \( \sigma' = \sigma \), so \( \rho = \sigma \) and we get \( \text{Mindiv}(L) \subseteq \text{Mindiv}(L') \). The other direction follows from symmetry.

**Proposition 8.** Let “\( \simeq \)” be a congruence with respect to “\( \models \)” and “\( \text{hide} \)” such that \( L \simeq L' \) implies \( \text{anylock}(L) = \text{anylock}(L') \). Then \( L \simeq L' \) implies \( \text{Ndinftr}(L) = \text{Ndinftr}(L') \).

**Proof.** Let \( L \simeq L' \) and let \( \Sigma \) be the common alphabet of \( L \) and \( L' \). Assume \( \xi = a_{11}a_{21}a_{31}\ldots \in \text{Ndinftr}(L) \). Let \( a_{\text{new}} \) be a novel action not in \( \Sigma \), and let \( \text{Test}_2 \) be the LTS with alphabet \( \Sigma \cup \{a_{\text{new}}\} \) in Figure 2. Clearly \( \text{hide } \Sigma \in (L\mid\text{Test}_2) \) diverges, so \( \text{anylock}(\text{hide } \Sigma \in (L\mid\text{Test}_2)) = \text{True} \) and, by the congruence requirement, \( \text{anylock}(\text{hide } \Sigma \in (L'\mid\text{Test}_2)) = \text{True} \). Because of the \( a_{\text{new}} \)-loops there can be no deadlock in \( L'\mid\text{Test}_2 \), so there is none in \( \text{hide } \Sigma \in (L'\mid\text{Test}_2) \) either. Thus, there is a divergence. Because \( \text{Test}_2 \) (with alphabet \( \Sigma \cup \{a_{\text{new}}\} \)) allows only those executions of \( L' \) whose visible traces are prefixes of \( \xi \), the divergence can only
exist if either some prefix of \( \xi \) is a divergence trace of \( L' \), or \( \xi \in Ndinftr(L') \). However, the previous proposition and the definition of \( Ndinftr(L) \) rule out the former possibility. The other direction follows again from symmetry.

**Proposition 9.** Let \( \simeq \) be a congruence with respect to \( \|\| \) such that \( L \simeq L' \) implies anylock\((L) = \) anylock\((L') \). Then \( L \simeq L' \) implies Dffail\((L) = \) Dffail\((L') \).

**Proof.** Let \( \Sigma \) and \( a_{\text{new}} \) be as in the previous proofs. Assume \((\sigma, A) \in Dffail(L), \) where \( \sigma = a_1a_2\ldots a_n \) and \( A = \{b_1, b_2, \ldots, b_k\} \) is finite or infinite. Let Test\(_3\) be the LTS with alphabet \( \Sigma \oplus \{a_{\text{new}}\} \) in Figure 3. Clearly, \( \sigma \in Dltr(L\|\|\text{Test}_3) \). Thus, anylock\((L\|\|\text{Test}_3) = \text{True} \), so anylock\((L'\|\|\text{Test}_3) = \text{True} \). Because of Proposition 7 and the definition of Dffail, no prefix of \( \sigma \) or \( \sigma b_i \), where \( b_i \in A \), can be a divergence trace of \( L \) or \( L' \). It follows that there are no divergences in \( L'\|\|\text{Test}_3 \), so there has to be a deadlock. Because of the \( a_{\text{new}}\)-loops there can be no deadlock trace other than \( \sigma \), so \( \sigma \in Dltr(L'\|\|\text{Test}_3) \). Because of the structure of Test\(_3\), we see that \((\sigma, A) \in Sfail(L') \), and then by the above observations, \((\sigma, A) \in Dffail(L') \). The reverse follows again from symmetry. \qed

The following straightforward result shows that anylock\((L) \) can be determined from Mindiv\((L) \) and Dffail\((L) \).

**Proposition 10.** anylock\((L) \iff Mindiv(L) \neq \emptyset \lor Dffail(L) \cap (\Sigma^* \times \{\Sigma\}) \neq \emptyset \)


Fig. 3. An LTS for testing failures

We are now in a position to state the weakest-congruence result we were after:

**Theorem 1.** The weakest congruence with respect to \( \|\| \) and “hide” that preserves anylock (i.e. distinguishes processes that can stop executing visible actions from those that cannot) is the Mindiv-Ndinftr-Dffail-equivalence.

**Proof.** As mentioned above, the functions that give the Mindiv, Ndtr and Ndinftr sets as functions of the sets of the parameters of \( \|\| \) and “hide” can be found in [17]. Propositions 5 and 6 give the required additional functions for Dffail. Ndtr\((L) = \{\sigma \in \Sigma^* \mid (\sigma, \emptyset) \in Dffail(L)\} \), so the Ndtr component is redundant. These results show that the equivalence is a congruence, and Proposition 10 shows that it preserves anylock. Propositions 7, 8 and 9 show that any congruence that preserves anylock implies the Mindiv-Ndinftr-Dffail-equivalence. \qed

After dealing with the general case of all LTSs, we next consider two special cases, namely the subsets of finitely nondeterministic and finite LTSs. The following result for the Ndtr and Ndinftr sets of a finitely nondeterministic LTS
has been shown in [17]; essentially the same result, although in a different form, can also be found as Theorem 7.4.2 in [19]. It should be noted that the corresponding result for $\text{Tr}$ and $\text{Inftr}$, Proposition 1, holds only for finite LTSs but not for all finitely nondeterministic LTSs.

**Proposition 11.** If $L$ is a finitely nondeterministic LTS, then 
\[
\text{Ndinftr}(L) = \{ \xi \in \Sigma^\omega \mid \forall \rho < \xi : \rho \in \text{Ndtr}(L) \}.
\]

This shows that in the class of finitely nondeterministic (or finite) LTSs, $\text{Ndinftr}$ becomes redundant and the $\text{Mindiv}$-$\text{Ndinftr}$-$\text{Dffail}$-equivalence collapses to the $\text{Mindiv}$-$\text{Dffail}$-equivalence.

**Theorem 2.** The weakest congruence between finitely nondeterministic / finite LTSs with respect to "||" and "\text{hide}" that preserves anylock is the $\text{Mindiv}$-$\text{Dffail}$-equivalence. The claim remains valid if the part 'and "hide"' is removed.

**Proof.** Proposition 7, which uses a finite LTS, shows that every congruence with respect to "||" preserves $\text{Mindiv}$. As for Proposition 9, process $\text{Test}_3$ in Figure 3 is always finitely nondeterministic (each $b_i$ occurs only once). If the LTSs are finite, then $\Sigma$ is finite, and therefore $A$ and $\text{Test}_3$ are also finite. Thus, in both cases the proof of the proposition remains valid, and the equivalence has to preserve $\text{Dffail}$. We have already shown that the $\text{Mindiv}$-$\text{Ndinftr}$-$\text{Dffail}$-equivalence is a congruence with respect to both "||" and "\text{hide}", and this equivalence is now the same as the $\text{Mindiv}$-$\text{Dffail}$-equivalence. $\Box$

### 4 The Weakest Any-Lock-Trace-Preserving Congruence

After presenting the weakest anylock-preserving congruence, we will complete the spectrum of deadlock and/or livelock-preserving congruences by looking into the weakest congruence that preserves all traces of visible actions that lead either to a deadlock or a livelock. In other words, we want to preserve precisely the traces after which the system can stop executing visible actions.

**Definition 13.** Let $L$ be an LTS. The set of the any-lock-traces of $L$ is 
\[
\text{Anylocktr}(L) = \text{Dltr}(L) \cup \text{Dvitr}(L).
\]

**Example 2.** For process $P_5$ in Figure 4 it holds that $\text{anylock}(P_5) = \text{True}$, like it did for the processes $P_2$, $P_3$ and $P_4$ in Figure 1. However, in terms of $\text{Anylocktr}$, $P_5$ is different from every process in Figure 1. On the other hand, $P_5$ is $\text{Anylocktr}$-equivalent to $P_6$, because $\text{Anylocktr}(P_5) = \text{Anylocktr}(P_6) = (send\ rec)^* \cup (send\ rec)^*send$. Furthermore, $P_7$ is $\text{Anylocktr}$-different from $P_1, \ldots, P_6$, but $\text{Anylocktr}(P_7) = \text{Anylocktr}(P_5) = (send\ rec)^*$.

For preserving all of $\text{Anylocktr}$ we will need the following semantic set, which was defined in [17]:

**Definition 14.** Let $L = (S, \Sigma, \Delta, s)$ be an LTS. The set of the eventually non-divergent infinite traces of $L$ is $\text{Enditr}(L) = \text{Inftr}(L) \setminus \text{Dvitr}(L)$, where 
\[
\text{Dvitr}(L) = \{ \xi \in \Sigma^\omega \mid \forall \sigma < \xi : \exists \sigma' : \sigma \leq \sigma' < \xi \land \sigma' \in \text{Dvitr}(L) \}.
\]
The abbreviation \( \text{Divcl} \) stands for “divergence closure”. Eventually nondivergent infinite traces are those infinite traces, of whose prefixes only finitely many are divergence traces. In [17] it was shown that the \( \text{Tr}-\text{Divtr}-\text{Enditr} \)-equivalence is the weakest congruence with respect to “\( \| \)" and “\( \text{hide} \)” that preserves all divergence traces.

We also need a new failure-set definition. It turns out that we need to preserve precisely those stable failures for which the trace part is not a divergence trace.

This semantic set is taken from [10], where it was used as part of the weakest “\( \text{LTL}_{\vdash_X} \)”-preserving congruence.

**Definition 15.** Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS. The set of the nondivergent failures of \( L \) is \( \text{Ndfail}(L) = S_{\text{fail}}(L) \setminus \text{Divtr}(L) \times 2^\Sigma \).

The following properties of \( \text{Ndfail} \) are easy to establish:

**Proposition 12.** Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS. Then

\[
\text{Tr}(L) = \text{Divtr}(L) \cup \{ \sigma \in \Sigma^* \mid (\sigma, \emptyset) \in \text{Ndfail}(L) \}, \quad \text{and} \\
\text{Anylocktr}(L) = \text{Divtr}(L) \cup \{ \sigma \in \Sigma^* \mid (\sigma, \Sigma) \in \text{Ndfail}(L) \}.
\]

The functions which relate the \( \text{Ndfail} \) set resulting from “\( \| \)" or “\( \text{hide} \)” with the \( \text{Ndfail} \) sets of the parameter processes and the \( \text{Divtr} \) set of the result can be obtained from [23]. The functions for \( \text{Tr}, \text{Divtr} \) and \( \text{Enditr} \) can be found in [17]. Furthermore, the above proposition shows that \( \text{Tr} \) becomes redundant if \( \text{Divtr} \) and \( \text{Ndfail} \) are both included in the semantic model. Thus, we can conclude that the \( \text{Divtr}-\text{Enditr}-\text{Ndfail} \)-equivalence is a congruence with respect to “\( \| \)" and “\( \text{hide} \)”. Proposition 12 also shows that this equivalence preserves \( \text{Anylocktr} \). It then only remains to prove minimality.

**Proposition 13.** Let “\( \simeq \)" be a congruence with respect to “\( \| \)" such that \( L \simeq L' \) implies \( \text{Anylocktr}(L) = \text{Anylocktr}(L') \). Then \( L \simeq L' \) implies \( \text{Divtr}(L) = \text{Divtr}(L') \).

**Proof.** Let \( L \simeq L' \) and let \( \text{Test}_1 \) be similar as in the proof of Proposition 7. Then \( \sigma \in \text{Divtr}(L) \iff \sigma \in \text{Divtr}(L||\text{Test}_1) \iff \sigma \in \text{Anylocktr}(L||\text{Test}_1) \iff \sigma \in \text{Anylocktr}(L'||\text{Test}_1) \iff \sigma \in \text{Divtr}(L'||\text{Test}_1) \iff \sigma \in \text{Divtr}(L') \). The second and penultimate logical equivalences follow from the fact that, because of the structure of \( \text{Test}_1 \), \( \sigma \) cannot lead to a deadlock in \( L||\text{Test}_1 \) or \( L'||\text{Test}_1 \). \( \square \)

**Proposition 14.** Let “\( \simeq \)" be a congruence with respect to “\( \| \)" and “\( \text{hide} \)” such that \( L \simeq L' \) implies \( \text{Anylocktr}(L) = \text{Anylocktr}(L') \). Then \( L \simeq L' \) implies \( \text{Enditr}(L) = \text{Enditr}(L') \).
Proof. Let \( L \simeq L' \) and let \( \Sigma \) be the common alphabet of \( L \) and \( L' \). Assume \( \xi = a_1 a_2 a_3 \cdots \in \text{Enditr}(L) \). By definition, \( \xi \) has a prefix \( a_1 a_2 \cdots a_n \) such that there is no \( \sigma \in \text{Divtr}(L) \) with \( a_1 a_2 \cdots a_n \leq \sigma < \xi \). Let \( a_{\text{new}}, b_{\text{new}} \) be two novel actions not in \( \Sigma \), and let \( \text{Test}_4 \) be the LTS with alphabet \( \Sigma \cup \{ a_{\text{new}}, b_{\text{new}} \} \) in Figure 5. We see that \( b_{\text{new}} \in \text{Divtr}(\Sigma \in (L||\text{Test}_4)) \subseteq \text{Anylocktr}(\Sigma \in (L||\text{Test}_4)) \), so by the congruence requirement \( b_{\text{new}} \in \text{Anylocktr}(\Sigma \in (L'||\text{Test}_4)) \). Because of the \( a_{\text{new}} \)-loops in \( \text{Test}_4 \) there can be no deadlocks in \( L'||\text{Test}_4 \), so there are none in \( \Sigma \in (L'||\text{Test}_4) \) either. Thus, \( b_{\text{new}} \in \text{Divtr}(\Sigma \in (L'||\text{Test}_4)) \). From this we can conclude that either \( \xi \in \text{Inftr}(L') \), or \( \text{Divtr}(L') \) contains some \( \sigma \) such that \( a_1 a_2 \cdots a_n \leq \sigma < \xi \). The latter is a contradiction, because by the previous proposition \( \text{Divtr}(L') = \text{Divtr}(L) \). The same argument also rules out the possibility that \( \xi \in \text{Divcl}(L') \), so we can conclude that \( \xi \in \text{Enditr}(L') \). The other direction follows from symmetry. \( \square \)

The following result can be shown similarly as the earlier results by using \( \text{Test}_3 \) from Figure 3:

**Proposition 15.** Let \( \simeq \) be a congruence with respect to \( || \) such that \( L \simeq L' \) implies \( \text{Anylocktr}(L) = \text{Anylocktr}(L') \). Then \( L \simeq L' \) implies \( \text{Ndfail}(L) = \text{Ndfail}(L') \).

**Theorem 3.** The weakest congruence with respect to \( || \) and \( \text{hide} \) that preserves \( \text{Anylocktr} \) (the traces after which a process can stop executing visible actions) is the \( \text{Divtr} \)-\( \text{Enditr} \)-\( \text{Ndfail} \)-equivalence.

We now again consider the special cases of finitely nondeterministic and finite systems. The processes used in Propositions 13 to 15 are finitely nondeterministic. Thus, the claim in Theorem 3 holds also among finitely nondeterministic systems.

For finite systems we can show by using Proposition 1 and the definition of \( \text{Enditr} \) the following:

**Proposition 16.** If \( L = (S, \Sigma, \Delta, s) \) is a finite LTS, then
\[
\text{Enditr}(L) = \{ \xi \in \Sigma^\omega \mid \forall \sigma < \xi : \sigma \in \text{Tr}(L) \} \setminus \text{Divcl}(L).
\]

Thus, in the class of finite systems the \( \text{Divtr} \)-\( \text{Enditr} \)-\( \text{Ndfail} \)-equivalence collapses to the \( \text{Divtr} \)-\( \text{Ndfail} \)-equivalence. The processes used in Propositions 13 and 15 are finite when \( \Sigma \) is finite. Therefore we can conclude:

**Theorem 4.** The weakest congruence with respect to \( || \) and \( \text{hide} \) between finite systems that preserves \( \text{Anylocktr} \) is the \( \text{Divtr} \)-\( \text{Ndfail} \)-equivalence.
5 Weakest Congruence Characterisation of CSP-Equivalence

Readers who are familiar with the CSP theory have probably noticed a close resemblance between the Mindiv(-Ndinftr)-Dfail-equivalence of Section 3 and the CSP failures-divergences equivalence of [5, 9, 19]. As the latter is a very well-known and important equivalence, we would now like to make this connection explicit.

Because the CSP failures-divergences equivalence (or CSP-equivalence for short) is only a congruence among finitely nondeterministic systems ([19] p. 200), we will in this section consider only finitely nondeterministic L TSs. As shown in Section 3, in this class of systems the Mindiv-Ndinftr-Dfail-equivalence reduces to the Mindiv-Dfail-equivalence, and it is the weakest congruence among these systems that preserves anylock.

CSP-equivalence can be defined in the L TS framework as follows ([19] p. 191).

Definition 16. Let $L = (S, \Sigma, \Delta, \hat{s})$ be a finitely nondeterministic LTS.

- $CSPdivtr(L) = \{ \sigma \in \Sigma^* \mid \exists \rho \leq \sigma : \rho \in Divtr(L) \}$
- $CSPfail(L) = Sfail(L) \cup (CSPdivtr(L) \times 2^\Sigma)$
- CSP-equivalence is the CSPfail-CSPdivtr-equivalence.

The following proposition shows that CSP-equivalence implies the Mindiv-Dfail-equivalence.

Proposition 17. Let $L$ be a finitely nondeterministic LTS. Then

- Mindiv($L$) = $\text{minimals}(CSPdivtr(L))$, and
- $Dfail(L) = \{ (\sigma, A) \in CSPfail(L) \mid \forall a \in A : \sigma a \notin CSPdivtr(L) \} \backslash CSPdivtr(L) \times 2^\Sigma$.

In the opposite direction, clearly $CSPdivtr(L) = \{ \sigma \in \Sigma^* \mid \exists \rho \leq \sigma : \rho \in Mindiv(L) \}$. However, $CSPfail(L)$ cannot be obtained from $Mindiv(L)$ and $Dfail(L)$. Indeed, CSP-equivalence is strictly stronger than the Mindiv-Dfail-equivalence. This is shown by the processes in Figure 6, which are Mindiv-Dfail-equivalent but not CSP-equivalent: $(a, \{a\}) \in CSPfail(L_1)$, but $(a, \{a\}) \notin Dfail(L_1)$, because "aa" leads to a divergence.

These results mean that CSP-equivalence preserves anylock but is not the weakest congruence that preserves this predicate. This raises the question as to whether there is some similar characterisation for which CSP-equivalence is the
weakest preserving congruence. Of course, we could trivially characterise CSP-equivalence as the weakest congruence that preserves $CSPfail$ and $CSPdivtr$, but this is obviously not very useful. Rather, we would like to find a characterisation that is as weak (and, hopefully, simple) as possible while still requiring all the information preserved by CSP-equivalence to be a congruence.

The corrected form of Theorem 9.3.1 (iii) in [19] (see the errata on the www-page of the book http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency) characterises CSP-equivalence as the weakest “immediate-any-lock”-preserving congruence, that is, as the weakest congruence that distinguishes systems that can deadlock or diverge before executing any visible actions from those that cannot. However, this result depends crucially on the use of a multiple (double) renaming operator that can convert a transition into two transitions with different labels.

To illustrate the idea, we can again consider the CSP-different processes $L_1$ and $L_2$ in Figure 6. To show that they have to be differentiated, we should find a context $C[\cdot]$ such that $C[L_1]$ and $C[L_2]$ are different in terms of “immediate-any-lock”. $L_1$ has the failure $(a, \{a\})$, i.e. deadlock, that $L_2$ does not have (if the failure were something less than a deadlock, we could make it one by using parallel composition with a tester process similar to the one in Figure 3). We can make this an immediate deadlock by hiding $a$. However, the problem is that this also hides the second $a$-action, so that the divergence can be reached with $\tau$-actions. Thus, both systems become immediately divergent, which nullifies the effect of the immediate deadlock.

With a clever technique based on double renaming it is possible to hide the actions in the trace under investigation (the first $a$ in the example) without hiding the actions after the trace that correspond to the failure set (the second $a$ in the example), and therefore the proof of the above-mentioned theorem can nevertheless be carried through (see the above www-page for details).

However, we conjecture that the proof cannot be carried through without the double renaming operator, meaning that the weakest congruence with respect to “$\parallel$” and “hide” (and ordinary renaming) that preserves “immediate-any-lock” is strictly weaker than CSP-equivalence. It would therefore be interesting to find a characterisation for CSP-equivalence that does not depend on the multiple renaming operator, namely, for which the more common “$\parallel$” and “hide” would suffice.

In the following we define the set of deadlock traces that do not have divergence traces as prefixes. These are the sequences of visible actions after which the system can terminate without having had a chance to diverge earlier.

**Definition 17.** Let $L = (S, \Sigma, \Delta, \bar{s})$ be an LTS. Then the set of the divergence-free deadlock traces of $L$ is $Dfdlttr(L) = Dltr(L) \setminus Divext(L)$.

Starting from this definition it is straightforward to show that $Dfdlttr(L) = \{ \sigma \mid (\sigma, \Sigma) \in CSPfail(L) \} \setminus CSPdivtr(L)$. Thus, CSP-equivalence preserves $Dfdlttr$. CSP-equivalence is well known to be a congruence among finitely non-deterministic systems [19]. The following two propositions then show that it is the weakest congruence with respect to “$\parallel$” and “hide” that preserves $Dfdlttr$.

105
Proposition 18. Let \(\sim\) be a congruence with respect to \("\mid\mid\)" and \("\text{hide}\)" such that \(L \sim L'\) implies \(\text{Dfdltr}(L) = \text{Dfdltr}(L')\). Then \(L \sim L'\) implies \(\text{CSPdivtr}(L) = \text{CSPdivtr}(L')\).

Proof. Let \(L \sim L'\) and let \(\Sigma\) be the common alphabet of \(L\) and \(L'\). Let \(\text{Test}_3\) be an LTS with alphabet \(\Sigma\) that is like \(\text{Test}_3\) in Figure 2 but without the \(a_{\text{new}}\)-loops. Assume \(\sigma \in \text{CSPdivtr}(L)\). Then some prefix \(\rho\) of \(\sigma\) is a divergence trace of \(L\), and \(\rho \in \text{Dltr}(L|\text{Test}_3)\). Thus, \(\varepsilon \in \text{Ditr}(\text{hide} \Sigma \text{ in } (L|\text{Test}_3))\) and therefore \(\text{Dfdltr}(\text{hide} \Sigma \text{ in } (L|\text{Test}_3)) = \emptyset\), so \(\text{Dfdltr}(\text{hide} \Sigma \text{ in } (L'|\text{Test}_3)) = \emptyset\). Now some prefix of \(\sigma\) has to be a divergence trace of \(L'\), implying \(\sigma \in \text{CSPdivtr}(L')\).

Namely, if no prefix were a divergence trace, then \(L'|\text{Test}_3\) would have to deadlock after some prefix of \(\sigma\), because \(\text{Test}_3\) only allows executions whose visible traces are prefixes of \(\sigma\). This would create a divergence-free deadlock in \(\text{hide} \Sigma \text{ in } (L'|\text{Test}_3)\), which is a contradiction. \(\square\)

Proposition 19. Let \(\sim\) be a congruence with respect to \("\mid\mid\)" such that \(L \sim L'\) implies \(\text{Dfdltr}(L) = \text{Dfdltr}(L')\). Then \(L \sim L'\) implies \(\text{CSPfail}(L) = \text{CSPfail}(L')\).

Proof. Let \(\Sigma, \sigma, A, a_{\text{new}}\) and \(\text{Test}_3\) be as in the proof of Proposition 9, and let \((\sigma, A) \in \text{CSPfail}(L)\). Then either some prefix \(\rho\) of \(\sigma\) is a divergence trace of \(L\), or no prefix is a divergence trace and \((\sigma, A) \in \text{Sfail}(L)\). In the former case, \(\rho \in \text{CSPdivtr}(L)\), so by the previous proposition, \(\rho \in \text{CSPdivtr}(L')\), whereby \((\sigma, A) \in \text{CSPfail}(L')\). In the latter case, clearly \(\sigma \in \text{Dltr}(L|\text{Test}_3)\) and, furthermore, no prefix of \(\sigma\) is a divergence trace of \(L|\text{Test}_3\). Thus, \(\sigma \in \text{Dfdltr}(L|\text{Test}_3)\), and by the congruence requirement, \(\sigma \in \text{Dfdltr}(L'|\text{Test}_3) \subseteq \text{Dltr}(L'|\text{Test}_3)\). From the structure of \(\text{Test}_3\) we see that \((\sigma, A) \in \text{Sfail}(L') \subseteq \text{CSPfail}(L')\). \(\square\)

We have shown that \(\text{Dfdltr}\) is a weakest congruence characterisation for CSP-equivalence. However, as mentioned above, we would like to find a characterisation that is as weak as possible, and it quickly turns out that \(\text{Dfdltr}\) is not a weakest possible characterisation. Let us define a predicate that tells whether or not there are any divergence-free deadlocks in \(L\).

Definition 18. Let \(L\) be an LTS. Then \(\text{dfdl}(L) : \iff \text{Dfdltr}(L) \neq \emptyset\).

Since \(\text{dfdl}\) is (strictly) weaker than \(\text{Dfdltr}\), CSP-equivalence preserves this predicate. We can then show with a straightforward modification of the above two proofs that CSP-equivalence is the weakest congruence that preserves \(\text{dfdl}\).

Thus, \(\text{dfdl}\) is also a characterisation of CSP-equivalence.

Furthermore, it is easy to see that \(\text{dfdl}\) is a \emph{minimal} characterisation in the sense that no strictly weaker equivalence is a characterisation of CSP-equivalence. Namely, \(\text{dfdl}\) is a predicate, so as an equivalence it has only two equivalence classes (those processes for which it holds and those for which it does not).

Therefore, the only strictly weaker equivalence is the trivial one which equates all processes, for which CSP-equivalence is clearly not the weakest preserving congruence (since it is itself a congruence).

However, \(\text{dfdl}\) is not \emph{the weakest} characterisation in the sense that every other characterisation would imply it. As a trivial counterexample, we can define the
predicate \textit{non-\(\varepsilon\)-dfdl} which states that there are divergence-free deadlock traces \textit{other than \(\varepsilon\)}. This is also weaker than \textit{Dfdltr}, so CSP-equivalence preserves it, and we can again modify the earlier proofs (namely, by suitable \(b_{\text{new}}\)-actions) to show that CSP-equivalence is the weakest preserving congruence. So, as it is a predicate, \textit{non-\(\varepsilon\)-dfdl} is also a minimal characterisation of CSP-equivalence. Thus, there is no \textit{minimum} characterisation.

6 Related Work

Robin Milner’s remark on p. 206 of [13] can be said to be one of the origins of the research on weakest congruences: “Hoare’s failures equivalence . . . is important, because it appears to be the weakest equivalence which never equates a deadlocking agent with one which does not deadlock.” Milner no doubt also required that the equivalence must be a congruence, because otherwise the weakest equivalence would be the trivial one that has precisely two equivalence classes: the processes (that is, Milner’s agents) that deadlock, and those that do not. In [20] it was proven that Milner’s guess was not precisely correct. The weakest deadlock-preserving congruence depends on the set of allowed process composition operators. Furthermore, assuming a reasonable choice of operators, it is the same as Hoare’s failures equivalence only in the absence of divergence.

Vogler [24] identified the weakest congruence with respect to parallel composition of Petri Nets that preserves a predicate comprising deadlock and non-exitable divergence, as well as the weakest congruence that preserves the visible traces of all maximal occurrence sequences.

As already mentioned, the weakest congruence that preserves the traces leading to a livelock and the weakest congruence that preserves the existence of livelocks were identified in [17].

In [10] the so-called \textit{nondivergent failures divergences equivalence (NDFD-equivalence)} was shown to be the weakest congruence that preserves the validity of formulae written in classic Manna-Pnueli linear time temporal logic [12] from which the “next state” operator “\(\Box\)” has been removed. This logic is extremely important in verification of concurrent systems. Furthermore, if the congruence has to preserve also deadlocks, then the weakest congruence is the \textit{Chaos-free failures divergences (CFFD) equivalence}. A formulation that is perhaps more relevant for practical verification can be found in [21] and in ([22] pp. 498–499).

Some researchers have tried to find the weakest congruence that preserves the results of certain kinds of \textit{tests} on processes, an approach introduced by De Nicola and Hennessy [6, 8]. The solution with a fair way of testing was given by Brinksma, Rensink and Vogler in [4], and Leduc came to the conclusion that with another view to testing, the NDFD-equivalence is the solution [11]. Some equivalences investigated in weakest congruence research have their origin in [1].

Boreale, De Nicola and Pugliese [3] give the must-testing [6] and fair testing [4] equivalences new characterisations as the weakest congruences that preserve certain simple predicates in a CCS-like language. It is also shown in [3] (where only finitely nondeterministic systems are considered) that essentially the
same equivalence that in [17] was shown to be the weakest congruence among finitely nondeterministic systems that preserves the existence of divergences, is the weakest congruence that preserves immediate divergence/convergence. Indeed, it is possible to construct contexts which demonstrate that the weakest congruence preserving immediate and non-immediate divergence has to be the same (this is not, however, the case for “any-lock”; see the discussion in Section 5).

Furthermore, [3] proposes a new testing framework, which differs from must-testing in that the observed process is not allowed to diverge at the point of success. Interestingly, the resulting equivalence (preorder) resembles closely the finitely nondeterministic case for the “any-lock”-predicate at the end of Section 3 of this paper. This is because the above requirement forces one to consider also divergence after the refused/accepted actions. On the other hand, [3] shows that this is the weakest congruence that preserves a predicate which is very different from “any-lock”: it considers for each visible action \( \ell \) the ability to diverge before or after the first \( \ell \), as well as the reachability ("guarantee") of \( \ell \) from every state reachable from the initial state with invisible actions.

It is also interesting to compare Theorem 4.11 from [3], dealing with must-testing, and the proof discussed in Section 5 concerning CSP-equivalence and “immediate-any-lock”. The proof of the former theorem does not require multiple renaming, which is apparently because of the difference in the given predicate: instead of deadlock it considers the “guarantee” for each visible action. An interesting question is also whether the latter proof could be adapted to CCS. This does not appear to be the case, because in CCS communications turn immediately into invisible actions.

In [15] strong bisimulation [13] is characterised as the weakest congruence that preserves a barbed bisimulation, which observes invisible actions and at each step preserves the possibility to perform (any) visible actions. In [7] a corresponding result for the weak case is shown to hold in the asynchronous name-passing \( \pi \)-calculus [14].

7 Conclusions

We proved that the weakest congruence that distinguishes systems that can stop executing visible actions from those that cannot (i.e., preserves “any-lock”) is the Mindiv-Ndinftr-Dffail-equivalence. We also proved that the weakest congruence that preserves all traces after which the system can stop executing visible actions is the Divtr-Enditr-Ndfail-equivalence.

Hoare’s well-known CSP-equivalence was shown to be strictly (albeit only slightly) stronger than the Mindiv(-Ndinftr)-Dffail-equivalence. Finally, we gave CSP-equivalence two weakest congruence characterisations, Dfdltr and dfdl, the latter of which is a minimal characterisation.

Although we have omitted the less important operators in this article, the weakest-congruence results in Sections 3 and 4 hold also for larger sets of operators that include ordinary and multiple renaming as well as action prefix and
sequential composition (which requires a special termination action). This can be shown by giving the appropriate semantic functions for these operators. However, if we include a choice operator, defined as in e.g. LOTOS [2], we have to strengthen those equivalences by one bit of information, namely by the initial stability bit that tells whether or not there are \( \tau \)-actions from the initial state of the process. This is a well-known phenomenon for failure-based congruences (CSP, however, uses a different kind of choice operator). A similar observation holds for the interrupt operator of LOTOS.

As for Section 5, a characterisation (e.g. with respect to parallel composition and hiding) is also a characterisation with any larger set of operators for which the given congruence retains its congruence property. Therefore, the given characterisations of CSP-equivalence hold also with any larger set of operators for which CSP-equivalence is a congruence (including multiple renaming).

This and other work on weakest congruences can be said to advocate one particular approach for defining process equivalences: we first define a property of systems we are interested in, and then identify a congruence, preferably the weakest one, that preserves this property with respect to our process operators. It is clear that no single equivalence can be universally better than all others. Preserving information that is not needed for the case at hand can sometimes be harmful. Also, the state-explosion problem may limit the amount of information that we are able to preserve. Therefore, it is useful if our verification system supports a hierarchy of equivalences with different strengths.

One interesting research direction are congruences that preserve information related to different notions of fairness (e.g. weak and strong fairness). As pointed out in [18], most present equivalences either do not preserve enough of this type of information, or are too strong because they do not allow us to abstract away invisible actions. Such congruences have already been developed in [16], although for communicating imperative programs rather than “classical” process algebra.

Acknowledgements The author wishes to thank Antti Valmari for useful comments on this paper. This work has been supported by the Academy of Finland, project “Unifying Action-Based and State-Based Verification Techniques”, and the TISE Graduate School.

References


Abstract

This paper presents a process-algebraic verification case study in which progress properties have an important role. The most well-known process-algebraic theories, namely CCS and CSP, have limited capability for handling progress properties. A variant of CSP called CFFD that can express a wide range of progress properties was developed about ten years ago. However, progress of a system often depends on so-called fairness assumptions that are difficult to express even with CFFD. This paper presents one way of using CFFD-based tools for verification of a full-duplex communication protocol in the presence of fairness assumptions.

Keywords

verification, tool support, progress, fairness

1. Introduction

Parallel and distributed systems are notoriously difficult to design because they are prone to concurrency-related errors, such as deadlocks, that are difficult to find and analyse. Therefore, formal methods have been developed to aid in the design of such systems. However, formal verification methods suffer from the so-called state-explosion problem, which is caused by the enormous number of different executions that a typical parallel system has. Process-algebraic [Ros98, Mil89] methods are one class of formal methods that have been successfully used to cope with the state explosion. This can be achieved by compositionality, where behaviours of the individual parts of a system can be reduced in size before they are combined to form the behaviour of the total system. Furthermore, process-algebraic methods can provide a visual representation of the behaviour of the system to the user, thus making the design process easier [VKS96].

However, process-algebraic methods have difficulties in dealing with the progress (liveness) properties of systems. For example, the CSP model [Ros98] does not preserve any information on a system after it has performed a divergence (livelock) trace, and the weak bisimilarity of CCS [Mil89] ignores divergences completely. There are divergence-sensitive variants of bisimilarity, but they perform less well in reduction of systems, for reasons discussed in [Val95].

For this reason a model called CFFD (Chaos-Free Failures Divergences) [VaT91, VaT95] has been developed which resembles CSP but preserves information even after the execution of a divergence trace. CFFD covers the properties expressible in the popular linear-time temporal logic [MaP92] (without the ‘O’ operator) as well as deadlocks, and is well-suited for reduction [KaV92, Val95].

Even though CFFD allows the study of liveness properties, these often depend on so-called fairness assumptions [Fra86], which are difficult to combine with the compositional approach of process algebra.

In [BRV95] an equivalence is introduced that preserves information about the visible actions that are reachable after the execution of a given trace. However, it is not known how this equivalence can be used algorithmically, and it also does not allow us to make specific fairness assumptions to remove livelocks, as we will do in Section 6.

In this paper, we present a case study where our CFFD-based method is applied to the design of a parallel system in which progress properties play an important role. CFFD is briefly reviewed in Section 2. We build our example on an existing communication protocol, which is introduced in Section 3, and we design a fully bidirectional version of the protocol in Section 4. We also see how our method can provide immediate visual feedback about the correctness of the system. In Section 5 we show how assumptions about the environment of the system can be included in the model. In Section 6 we focus on the fairness problem.
and show how the user of our method can express and apply fairness assumptions to remove livelocks. Section 7 summarises our results.

2. Background

For completeness, we will now briefly review the most important parts of the CFFD-theory. However, most of this article will be understandable even without the detailed mathematical definitions – our goal is that our verification tools can be used without mathematical background.

In our approach processes are represented as labelled transition systems (LTS). These are directed graphs whose nodes represent states of the system and edges represent transitions between the states. The transitions are labelled by actions of the process. Actions can be either visible or invisible. Visible actions have names and they can be observed by the outside world, whereas invisible actions represent internal changes in the system. Invisible actions are denoted by the symbol $\tau$ (tau).

Definition 1 A labelled transition system (LTS) is a 4-tuple $(S, \Sigma, \Delta, is)$ where $S$ is the set of states, $\Sigma$ is a set of symbols called the alphabet such that $\tau \not\in \Sigma$, $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is the transition relation, and $is \in S$ is the initial state.

Processes can be combined with the parallel composition operator. Several variants of the operator have been defined. In the one used here, if $a$ is a visible action (that is, $a \neq \tau$), then $a$ is executed simultaneously by precisely those processes of the parallel composition that have $a$ in their alphabets. Synchronization of actions is thus determined by the alphabets of the component processes. The invisible action $\tau$ is executed by exactly one process at a time.

Definition 2 Let $L_1 = (S_1, \Sigma_1, \Delta_1, is_1), \ldots, L_n = (S_n, \Sigma_n, \Delta_n, is_n)$ be LTSs. Their parallel composition is the LTS $(S, \Sigma, \Delta, is) = \parallel L_1 \ldots \parallel L_n$ defined as follows:

- $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$, $S = \parallel S_1 \times \ldots \times S_n$ and $is = (is_1, \ldots, is_n)$.
- $((s_1, \ldots, s_n), \tau, (s'_1, \ldots, s'_n)) \in \Delta$ if and only if for every $1 \leq i \leq n$ such that $(s_i, \tau, s'_i) \in \Delta_i$ and $s'_i = s_j$ whenever $1 \leq j \leq n$ and $j \neq i$.
- If $a \in \Sigma$, then $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ if and only if for every $1 \leq i \leq n$ either $a \in \Sigma_i$ and $(s_i, a, s'_i) \in \Delta_i$ or $a \not\in \Sigma_i$ and $s'_i = s_j$.

In practice, the part of the parallel composition that is not reachable from the initial state is ignored, as it has no semantic significance (see Definition 4).

We can make uninteresting actions invisible with the hiding operator. $hide A$ in $L$ converts all transitions of $L$ labelled with an action in $A$ into $\tau$-transitions and subtracts $A$ from the alphabet.

The following notation is useful for abstracting $\tau$-transitions away from execution sequences.

Definition 3 Let $L = (S, \Sigma, \Delta, is)$ be an LTS. Let $s, s' \in S$, $a \in \Sigma \cup \{\tau\}$, and $a_1, \ldots, a_n \in \Sigma$.

- $s \xrightarrow{a} s'$ if and only if $(s, a, s') \in \Delta$.
- $s \xrightarrow{e} s'$ if and only if there are $n \geq 0$ and $s_0, s_1, \ldots, s_n$ such that $s = s_0$, $s_n = s'$ and $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n$, that is, $s'$ is reachable from $s$ by zero or more $\tau$-transitions.
- $s \xrightarrow{a} s'$ if and only if $a \neq \tau$ and there are $s_1$ and $s_2$ such that $s \xrightarrow{e} s_1 \xrightarrow{\tau} s_2 \xrightarrow{e} s'$.
- $s \xrightarrow{a_1 \cdot a_2 \cdots a_n} s'$ where $n > 1$ if and only if there are states $s_0, s_1, \ldots, s_n$ such that $s = s_0$, $s_n = s'$ and $s_0 = a_1 \cdot a_2 \cdots a_n \Rightarrow s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n$.
- $s \xrightarrow{\tau} s'$ if and only if there are $s'$ such that $s \xrightarrow{\tau} s'$, and similarly with $s = a_1 \cdot a_2 \cdots a_n \Rightarrow$.

In the following we define the sets that CFFD semantics use.

Definition 4 Let $L = (S, \Sigma, \Delta, is)$ be an LTS, $\sigma \in \Sigma^*$, and $A \subseteq \Sigma$.

- The set of traces of $L$ is $tr(L) = \{ \sigma \in \Sigma^* \mid is \xrightarrow{\sigma} \}$.
- The pair $(\sigma, A)$ is a stable failure of $L$, if and only if there is $s' \in S$ such that $\sigma \xrightarrow{\sigma} s'$, $s' = s \Rightarrow$, and $s' \xrightarrow{\tau} a$ for every $a \in A$. The set of stable failures of $L$ is denoted by $stfail(L)$.
- The sequence $\sigma \in \Sigma^*$ is a divergence trace of $L$, if and only if there are states $s_0, s_1, s_2 \ldots$ such that $is = \sigma \Rightarrow s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \ldots$. The set of divergence traces of $L$ is denoted by $divtr(L)$.
- The set of infinite traces of $L$ is $inftr(L) = \{ a_1 a_2 a_3 \ldots \in \Sigma^0 \mid \exists s_0, s_1, s_2, \ldots \in S: is = s_0 \wedge s_0 = a_1 \Rightarrow s_1 = a_2 \Rightarrow s_2 = a_3 \Rightarrow \ldots \}$.
- $stable(L)$ is a predicate that holds if and only if $is$ is stable, that is, $is \xrightarrow{\tau} \Rightarrow$.

We are now ready to define the CFFD semantics of an LTS. The CFFD semantics and the well-known CSP semantics [Ros98] are otherwise the same but they differ in the case of divergence traces. (Also the initial stability makes a difference, but it is not needed with the operators used in this article.) The CFFD semantics preserve information on traces, stable failures and divergence traces even after the process has executed a divergence trace, while the CSP semantics preserve no information in such a situation. Therefore the CFFD semantics are more suitable than the CSP semantics for analysing systems that contain divergences, such as the bidirectional protocol in this article. In the following we define the CFFD model and the corresponding CFFD-equivalence and CFFD-preorder.
Definition 5 Let $L$, $L_1$ and $L_2$ be LTSs, where $L_1$ and $L_2$ have the same alphabet.

- The \textit{CFFD model} of $L$ is the 4-tuple $(\text{stable}(L), \text{sfail}(L), \text{divtr}(L), \text{infr}(L))$.
- $L_1 \cong_{\text{CFFD}} L_2$ if and only if
  
  \[
  \text{stable}(L_1) = \text{stable}(L_2) \land \text{sfail}(L_1) = \text{sfail}(L_2) \land \text{divtr}(L_1) = \text{divtr}(L_2) \land \text{infr}(L_1) = \text{infr}(L_2).
  \]
- $L_1 \preceq_{\text{CFFD}} L_2$ if and only if
  
  \[
  (\text{stable}(L_1) \lor \neg \text{stable}(L_2)) \land \text{sfail}(L_1) \subseteq \text{sfail}(L_2) \land \text{divtr}(L_1) \subseteq \text{divtr}(L_2) \land \text{infr}(L_1) \subseteq \text{infr}(L_2).
  \]

The traces are not included in the model because they can be determined from the stable failures and divergence traces [VaT95]. Intuitively, CFFD-equivalence means that the behaviours of the two processes are the same regarding the properties that we are interested in. CFFD-preorder, on the other hand, means that the smaller process is ‘better’ than the larger process in the sense that the environment is more able to control its behaviour, and it is less likely to do anything forbidden.

CFFD-equivalence is congruent with respect to parallel composition and hiding (and many other operators [VaT95]). That is, if the component processes used in parallel composition and hiding are replaced by CFFD-equivalent ones, the result will be CFFD-equivalent. This helps us a great deal since we can choose a set of interesting (externally observable) actions in a system, and make all other actions invisible. Then we can process the system with a CFFD-preserving reduction algorithm and obtain an equivalent, but usually much smaller system that can replace the original system in any environment.

In the same way CFFD-preorder is a precongruence, so that if component processes are replaced by CFFD-smaller ones, the result will be CFFD-smaller or CFFD-equivalent [VaT95]. This means that if a system has been proven correct, then it remains correct when any of its component processes are replaced by CFFD-smaller processes. We will use this result in this article.

ARA, which is described in detail in [VKCL93, Sav95], is a group of computer tools that support the analysis of parallel systems on the basis of the CFFD semantics. Process descriptions can be written in a variant of the formal specification language LOTOS [BoB87], and an equivalent LTS is generated automatically. It is also possible to draw the LTS directly with a graphical tool. A reduction tool is available for producing a smaller but CFFD-equivalent LTS from a given LTS. After constructing and reducing the total behaviour of a system a dedicated visualization tool is used to show the result. The tool automatically tries to find a good layout for the graph, and different actions are denoted by different colours. The layouts of the global behaviour graphs shown in this article have almost entirely been produced by the tool, only the names of actions have been added manually.

3. Self-Synchronizing Alternating Bit Protocol

As the starting point of our development we used an earlier protocol, which has been introduced in [VKS96]. This protocol is an improved version of the famous alternating bit protocol [BSW69].

It is intended for sending messages over unreliable channels that can lose messages, but cannot reorder them. There are two one-way channels, one for the data from the sender to the receiver, and another for acknowledgements, as shown in Figure 1. The acknowledgements are needed because messages may get lost. If the acknowledgement for a message is not received in time, the protocol attempts retransmission. In order not to confuse new messages with retransmissions, all the messages and acknowledgements contain a sequence number. In the alternating bit protocol there are only sequence numbers 0 and 1.

The original alternating bit protocol may make an unbounded number of retransmissions. The [VKS96] version attempts only a finite number of transmissions.

\footnote{An improvement on the alternating bit protocol has also been presented in [HaS96]. That protocol is different from ours in that in it synchronization is based on timers, and the correctness relies on the assumption that timeouts do not occur prematurely.}
after which it gives up and declares an error to the client. However, since Sender does not know whether it was the message or the acknowledgement that disappeared – or whether they were just delayed – it cannot know what sequence number Receiver is expecting next. Therefore, before sending new data messages, it sends a special synchronization message, which conveys no data but which Receiver acknowledges in the usual manner. Only after receiving a correct acknowledgement does Sender send new data messages. In a sense, the synchronization procedure means forcing the earlier failed transmission to a completion, although without the data.

The LTSs of the data and acknowledgement channel are shown in Figure 2, and the Sender and Receiver processes in Figures 3 and 4. In these graphs send means a send data request from the client, err means declaring error and rec denotes receipt of the data on the other side. Action sd!0 means sending data with sequence number 0 to the channel, rd!0 reception of the data on the other side, ss!0 sending a synchronization, and so forth. The initial state is shown with a small arrow. For simplicity we have assumed that Sender attempts transmission only once. We have also not modelled the information content of data messages, as this does not directly affect the behaviour of the protocol. In order to make it more error-tolerant, the system has been made to perform a synchronization at startup.

The reduced global behaviour of this protocol produced by ARA when the actions send, rec and err have been left visible is shown in Figure 5. We note that after a send there is either a delivery with rec or the system may declare an error with err. An error may also be declared if the message has been delayed. We see that there cannot be more rec- than send-actions. There can sometimes be two successive rec-actions but, as was verified in [VKS96], this is just a consequence of the ability of the channels to hold messages: a new sending request may have been issued before the previous message was delivered, thus resulting in two subsequent deliveries.

4. Bidirectional Self-Synchronizing Protocol

We decided to develop a full-duplex version of the above one-way protocol with the following requirements. There would still be only two channels between the peers. Both sides should be able to send messages simultaneously, but a client on either side is also allowed to transmit messages when the other side is passive. Furthermore, for situations when there is both data and an acknowledgement to be transmitted at the same time, a new message type was added: a combined data and acknowledgement message.

The structure of the new protocol was based on the old one, so that each peer process consists of a com-
combined Sender and Receiver subprocess, as shown in Figure 6. Now there are two equal channels which both convey data and acknowledgements. The simplest definition of the new channel would be the old one augmented with each new message type. This is depicted in Figure 7 a), where \( m \) represents each of \( d_s, s_a \) and \( d_a \) (the combined message) with all different bit values, and the black state has a different copy for each \( m \).

However, at this point we decided to add just one more requirement for the protocol, because this channel definition is not the most general possible. Namely, when we tried the system with the reliable channel that cannot lose any messages, shown in Figure 7 b), we noted that the system deadlocks! This is because the reliable channel refuses to lose messages. If the message is not read and no other event is possible in the system, the old channel will choose the (invisible) lose-action. However, the reliable channel cannot do this, so the system deadlocks.

In the final version of our protocol we used the channel in Figure 7 c), which can both lose messages and refuse to lose them (by executing the \( \tau \)-transition to a state where only reception of the message is possible). In terms of the CFFD-preorder, \( \text{Channel}_{\text{rel}} \leq_{\text{CFFD}} \text{Channel}_{\text{new}} \) and \( \text{Channel}_{\text{old}} \leq_{\text{CFFD}} \text{Channel}_{\text{new}} \). As pointed out in Section 2, this means that if the protocol works correctly with the new channel, it also works correctly with the old channel and the reliable channel.

The Sender and Receiver subprocesses are shown in Figures 8 and 9. Here, \( sa?b \) denotes the actions \( sa!0 \) and \( sa!1 \), \( rd_a?b!0 \) denotes the actions \( rd_a!0!0 \) and \( rd_a!0!1 \), and so on. These subprocesses are based on the old Sender and Receiver processes, but there are many new transitions. These can be divided into four categories.

Firstly, we had to add the new combined data- and acknowledgement messages, which are sent and received jointly by Sender and Receiver. These messages contain two sequence numbers, so where Sender previously had a single \( sd!0 \)-action, for example, we now have also the actions \( sd_a!0!0 \) and \( sd_a!0!1 \) (state no. 4). When Sender is idle (waiting for send) it allows Receiver to receive all types of messages, including combined (states 1, 7, 8 and 14).

Secondly, we have forced the use of a combined message always when possible. Sender controls the send acknowledgement (\( sa \)) action of Receiver, refusing this action when it has data to send, and allowing it when it does not. The permission can be seen in the picture as an \( sa \) loop from a state to itself (from states 1, 3, 5, 6, 7, 8, 9, 10, 12, 14), while the absence of this loop means denial of the action. Receiver deals with the \( sd \) action of Sender in exactly the same way.

A third set of changes is needed because we can sometimes get unexpected acknowledgements caused by delayed messages, and the channels are now capable of keeping them forever. If we do not explicitly deal with them, the system can deadlock. Thus, we augment Sender with actions that read unexpected acknowledgements away from the channel (states 1, 3, 5, 6, 7, 8, 9, 10, 12, 14).

Figure 7: Three versions of the channel, with the branch corresponding to one message type shown.
However, even after the modification described above there is a deadlocking scenario, as was noted when a tentative version of the system was constructed. If both Receiver subprocesses are trying to send a response to the channels when they already contain messages from the Sender subprocesses, then the Receiver processes are unable to complete their own operation and cannot read the messages away from the channels; thus the system deadlocks. This problem is more difficult than the previous one, but in the final bidirectional protocol it was solved as follows: after receiving a message and before sending an acknowledgement the Receiver subprocess prevents Sender from sending anything to the channel. Therefore, there are sd- and ss-loops in states 1 and 6 of Receiver, but not in states 2, 3, 4 or 5. This might not appear to help very much, because Sender may have sent its message already. However, that cannot happen at both sides simultaneously. Thus, the arrangement in our final protocol ensures that there is at least one ‘free space’ somewhere in the system, and so the protocol cannot deadlock.

As the system is rather complicated, we emphasize that the reader does not have to understand every detail of the protocol. Indeed, it would be more or less impossible to be convinced of the correct behaviour of the protocol solely by looking at the structure of its components – or by running a limited number of test cases. The results obtained with the ARA tools, on the other hand, cover simultaneously every possible execution. Thus, we can see the complete picture of the behaviour of the system, and by using this picture we can either find errors or be convinced of the correctness of the system.

We will look at the protocol from the point of view of one transmission direction (AB-direction), while the other direction (BA) remains present in the system and its effects on the AB-direction are shown. The reduced global behaviour produced with ARA when all other actions except sendA, errA and recB are hidden is shown in Figure 10. We note immediately that the system does not deadlock. As we examine the picture in detail, we also note that there cannot be more recB- than sendA-actions. There can sometimes be two successive recB-actions, for the same reason as in the original protocol of Section 3.

A new feature in this picture when compared to Figure 5 is the presence of divergences, i.e. τ-loops from a state to itself. We will see in Section 6 that this is due to the fact that the transmission in the BA-direction can operate independently of the AB-direction. The actions of the BA-direction have been made invisible, and they appear as divergences in the reduced behaviour. It should be noted that according to CSP-semantics the entire LTS in Figure 10 would be equivalent to a single τ-loop, as the initial state contains a τ-loop. This shows why we need the CFFD-semantics.

**5. Modelling the Environment**

According to our preliminary analysis the behaviour of the protocol seems to be in order. What may not be immediately apparent, however, is that we have made assumptions about the environment (users) of the system. Namely, we hid the external actions correspond-
The idea is that with this model the user of the protocol is liberated from the burden of handling unexpected error messages, but it has permission to handle them if the designer wants that. On the other hand, the client process will always accept a recA-action; the receiving side cannot know when messages have been sent from the other side, so it should always be ready to receive. The following now holds: Client₁ ≤_{CFFD} Client₂ ≤_{CFFD} Client₃.

When we combine Client₃ with the protocol, we get the result shown in Figure 12. We see that the system still does not deadlock. When we compare this diagram with Figure 10, we immediately see that the left and upper parts are similar. In fact, the diagram has only changed with respect to sendA-actions. The three sendA-actions in the upper part of the old diagram have transformed into a single sendA starting from the second state in the third row. In this state the system can refuse all actions except sendA. This is a change that we would expect. If the other client has stopped sending messages and the previous transmission in the visible direction has been completed, then only a new sendA is possible. A similar change has taken place for the sendA-action on the lower right part of the old diagram.

6. Removing Divergences

Now the only remaining problem in our behaviour diagram is the presence of divergences. We cannot yet know with certainty that after entering a \(\tau\)-loop the system ever comes out of it. However, we guessed above that these \(\tau\)-loops reflect unending invisible flow of messages in the other direction. If the transmission in the other direction does not grab all the time in the system, then we can expect that the system will eventually move on from any of the \(\tau\)-loops.

We will now verify that this is the case. To do this, we must add to the system the assumption that B-side cannot make infinitely many sendB-actions without sendA-actions. Thus, we want to make a fairness assumption [Fra86],²

²In [BeK85] a manual process-algebraic verification is made with the help of the so-called Koomen’s fair abstraction rule. This means that when there is a cycle in the system consisting of actions to be hidden, then no \(\tau\)-cycle is formed but a step outside the cycle is assumed to be made ‘due to some fairness mechanism’. We do not want to do this, however, because a divergence can also be a real livelock caused by an error in the design. Instead, we intend to show that adding a specific fairness assumption leads to the disappearance of the divergences in the system.
We can make the assumption by augmenting the system with the process \textit{Fair} in Figure 13 b) before hiding the internal actions. This LTS will always accept a \textit{sendA}-action, but it accepts only a finite number of \textit{sendB}-actions between \textit{sendA}-actions. We do not want to assume anything else of the number of successive \textit{sendB}-actions than finiteness, and therefore we have to include in \textit{Fair} branches corresponding to all different maximum numbers. Unfortunately, this makes the LTS infinite, and we cannot directly handle such an LTS with our computer tools. However, we can prove what the result is by using the finite ‘lower limit’ \textit{Fairlow} (which is actually the first branch of \textit{Fair}) in Figure 13 a) and the ‘upper limit’ \textit{Fairhigh}, in Figure 13 c), and by applying results from the CFFD-theory.

It is a routine matter to check that \textit{Fairlow} \(\leq_{\text{CFFD}} \textit{Fair} \leq_{\text{CFFD}} \textit{Fairhigh}\) by comparing their LTS definitions against Definition 5. By the precongruence property of CFFD-preorder, the same relation holds for the total system with these LTSs: \(\text{Resultlow} \leq_{\text{CFFD}} \text{Result} \leq_{\text{CFFD}} \text{Resulthigh}\). The result with \textit{Fairhigh} as produced by ARA is (again) the LTS in Figure 12, and the result with \textit{Fairlow} is the LTS shown in Figure 14. We note that these are otherwise identical except that \textit{Resulthigh} has a number of divergences. Since the result with \textit{Fair} lies between these two, we conclude that it is the same as \textit{Resultlow} except that it may or may not have some of the divergence traces of \textit{Resulthigh}.

To complete our verification, we leave only the \textit{send}-actions on both sides of the protocol visible and construct the global behaviour diagram. The result is shown in Figure 15. We see that any infinite execution that does not contain an infinite number of \textit{sendA}-actions has to contain an infinite number of \textit{sendB}-actions. Since the \(\tau\)-loops in Figure 12 do not contain \textit{sendA} (\textit{sendA} is visible), they have to contain infinitely many \textit{sendB}-actions. \textit{Fair} does not allow executions that contain an infinite number of \textit{sendB}-actions without intermediate \textit{sendA}-actions, and therefore none of the divergences can exist in \textit{Result}. Thus, we conclude that the global behaviour of our protocol augmented with the fairness process \textit{Fair} is the LTS in Figure 14.

7. Conclusions

In this article a communication protocol was developed by using a set of computer tools that are based on process algebra and CFFD. We used an existing one-way self-synchronizing alternating bit protocol as a starting point and developed a fully bidirectional version of this protocol. During the development we noted that parallel systems like this can easily contain errors such as deadlocks that are very difficult to predict. However, our verification tools quickly pointed
out the errors. Then we were able to make changes to the system and check that the problems have been corrected and no new errors have been introduced.

It may seem that the deadlocks mentioned in Section 4 arise from the use of synchronous communication and would not have been a problem with, say, FIFO-based communication. In reality, however, the problem would not disappear with FIFOs, but only change shape. With FIFOs there would be the problem of an unbounded number of messages in the channel.

During our analysis we considered issues relating to the environment of the system. We added a client process that represents the environment in the invisible transmission direction. In terms of the CFFD model, we replaced the environment with a CFFD-larger one, and showed that the system still works correctly. In particular, this means that there will be no deadlocks in the system even when the other side stops transmitting.

The progress of a system can be hindered not only by deadlocks but also by livelocks (divergences). Proving liveness of a system is a subtle task that causes problems or is impossible with many formalisms such as CSP. In essence, we added a fairness assumption to the verification model ensuring that the reverse direction of the protocol cannot infinitely overtake the forward direction. With this assumption we proved that there are no \( \tau \)-loops, and therefore every sending request \((sendA)\) will eventually be handled in one way \((recB)\) or another \((errA)\).

Of course, this is by no means an ideal way to deal with fairness. We had to use the rather complicated ‘fairness’ processes to prove what we wanted and, even though most of our verification was automated, the final steps required some simple mathematical reasoning. In the future there may be better ways to deal with fairness. However, for the time being, this example gives us a method that can be used in practice.

Also, with our method the user can add fairness assumptions as needed to get rid of divergences. This is a great advantage, because guessing and formulating suitable fairness assumptions beforehand requires more intuition than can reasonably be assumed to be available when designing a new protocol.

Acknowledgements

This work has been supported by the TISE Graduate School and Academy of Finland, project ‘Specification and Verification of Distributed Systems with Synchronous Actions’.

References


Using Fairness in Process-Algebraic Verification

Antti Puhakka
Tampere University of Technology, Institute of Software Systems,
P.O. Box 553, FIN-33101 Tampere, Finland,
email: anpu@cs.tut.fi

Abstract. Although liveness and fairness have been used for a long time in classical model checking, with process-algebraic methods they have seen far less use. One problem is combining fairness with the compositionality of process algebra. In this article we analyse this problem, and then present an approach for using a class of fairness constraints. The approach fulfills all the requirements of compositionality and is compatible with an existing semantics. It is based on the standard LTS model and does not require new fairness-related constructs or rules for the process algebra. Therefore, it avoids potential conflicts between the fairness requirements and the underlying transition system. Although adding fairness can create an infinite subsystem, a larger system in which the subsystem is placed can still be finite. We present an algorithm for constructing a finite LTS which is equivalent to the larger system in every case that an exact finite representation exists, and which otherwise is a conservative estimate of it. However, checking whether an exact finite representation exists is costlier than building the representation, namely, it is PSPACE-complete in the size of an intermediate parameter system.

1 Introduction

In the verification of concurrent systems it is often important to show that the system eventually performs some desired task. Such properties are called liveness properties [2, 19]. For proving liveness properties some fairness assumptions [3, 12, 20, 21] usually have to be added to the system, meaning that the system is not allowed to continually favour some choices at the expense of others.

Within classical model checking [7, 30] liveness and fairness have been used in one form or another for quite some time. However, in the context of process-algebraic methods [17, 23, 31] they have seen relatively little use. One reason for this is that with most well-known process-algebraic semantic models it is difficult to express liveness properties. For example, the weak bisimilarity of CCS [23] ignores divergences (livelocks) completely, and the CSP model [17, 31] does not preserve any information on a system after it has performed a divergence trace.

A variant of CSP called CFFD (Chaos-Free Failures Divergences) [39] preserves information even after the execution of a divergence trace, and is suitable for expressing liveness properties. CFFD covers the properties expressible in linear-time temporal logic [22, 27] without the nextstate-operator (where the state-based logic is interpreted in an action-based setting) as well as deadlocks [18, 36]. It is also well-suited for typical process-algebraic verification methods [35, 37].
However, it is not clear how fairness assumptions should be used with the process-algebraic approach. One of the main problems is combining liveness and fairness with bottom-up type of compositionality, which is an important advantage offered by process-algebraic approaches for attacking the state-explosion problem (see, for instance, [37]). In this article we will discuss some of these problems and illustrate them with examples. Our emphasis is on “specific” fairness assumptions (such as “communication channel must eventually pass a message through”) rather than on “global” assumptions made automatically about every process or action (such as “every parallel process must eventually make progress”).

One solution that has been suggested previously is adding some finite fairness information to a process that constrains the infinite executions. However, a problem with this “finite representation” approach is the potential conflict between the underlying transition system and the additional fairness information, because the latter can demand actions that the former is unable to carry out. The problem is made especially difficult by the compositional setting. This issue is illustrated by an example in Section 3 and discussed further in Section 6.

In this article we use a CSP-like process algebra with LTSs (Labelled Transition Systems) as models of processes. LTSs are a simple, well-understood and widely accepted formalism for describing the behaviour of concurrent processes. Therefore, we take the view here that ordinary LTSs should be sufficient for describing process behaviour, whether that behaviour is “fair” or not. In this way there is no separation or conflict between the transition system and the fairness information. It suffices to prove that adding an allowed fairness constraint never creates new, unwanted behaviour in the resulting LTS. In this article we will do this for a useful class of fairness constraints that relate the infinite occurrence of different actions.

From the point of view of verification, a potential problem is that adding a fairness assumption to a finite LTS typically produces an infinite LTS. However, in many cases the infinite LTS is only a subsystem and, when it is placed in a larger context, the result can again be represented as a finite LTS. We will present an algorithm which constructs a finite LTS that is equivalent to the original system in every case that an exact finite representation exists, and otherwise it is a conservative estimate of it. However, it turns out that the complexity of deciding whether an exact finite representation exists is higher than of building the representation. More precisely, the problem is PSPACE-complete in the size of an intermediate parameter system. Fortunately, the parameter system can be significantly smaller than the full state-space of the system.

In the next section we will review the basic definitions concerning LTSs, process operators and behavioural equivalences. In Section 3 we will consider a (hypothetical) operator that modifies systems according to given fairness constraints. We describe the properties we believe such an operator should have in order to be meaningful in a compositional approach. We then show that most well-known semantic models are partly incompatible with these requirements, and also that the requirements impose limitations on the allowed fairness properties and process contexts. In spite of these problems, we are able to establish positive results for a useful class of fairness constraints in Section 4. The approach is based on ordinary LTSs and is compatible with the CFFD semantic model. In the beginning of Section 5 we illustrate the approach with a small verification example, and then we describe a general method for constructing a finite model of the infinite system. In Section 6
we describe related work and in Section 7 we present our conclusions. This article extends
the work in [29].

2 Background

The behaviour of a process consists of executing actions. There are two kinds of actions:
visible and invisible. Invisible actions are denoted with a special symbol ρ. The behaviour
of a process is represented as a labelled transition system. This is a directed graph whose
edges are labelled with action names, with one state distinguished as the initial state.

Definition 1 A labelled transition system, abbreviated LTS, is a four-tuple \((S, \Sigma, \Delta, \hat{s})\), where
\- \(S\) is the set of states,
\- \(\Sigma\), the alphabet, is the set of the visible actions of the process; we assume that
\(\tau \notin \Sigma\),
\- \(\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S\) is the set of transitions, and
\- \(\hat{s} \in S\) is the initial state.

We also use \(\Sigma(L)\) to denote the alphabet of the LTS \(L\). Let \(A^\omega\) denote the set of finite and
\(A^\omega\) the set of infinite sequences of elements of a set \(A\). The empty sequence is denoted with
\(\varepsilon\), and \(a^\omega\) denotes the infinite sequence of the symbol \(a\). For a finite or infinite sequence
\(\eta\), the restriction of \(\eta\) to \(B\), denoted \(\text{restr}(\eta, B)\), means the result of removing all actions
from \(\eta\) that are not in \(B\).

The following notation is useful for talking about the execution of a process. The
“\(\tau \mapsto \)"-notation requires that all actions along the execution path are listed, while the
\(\tau\)-actions are skipped in the “\(=\tau\Rightarrow \)"-notation.

Definition 2 Let \((S, \Sigma, \Delta, \hat{s})\) be an LTS, let \(s, s' \in S, a, a_1, a_2, a_3, \ldots \in \Sigma \cup \{\tau\}\), and \(b, b_1, b_2, b_3, \ldots \in \Sigma\). We write
\- \(s - a \mapsto s'\) if and only if \((s, a, s') \in \Delta\),
\- \(s - a_1 a_2 \cdots a_n \mapsto s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0, s_n = s'\) and \(s_i - a_i \mapsto s_{i-1}\) when \(1 \leq i \leq n\),
\- \(s - a_1 a_2 \cdots a_n \mapsto s'\) if and only if there is \(s' \in S\) such that \(s - a_1 a_2 \cdots a_n \mapsto s'\),
\- \(s - a_1 a_2 a_3 \cdots \mapsto s'\) if and only if there are \(s_0, s_1, s_2, \ldots \in S\) such that \(\hat{s} = s_0\) and \(s_i - a_i \mapsto s_{i+1}\) when \(i \geq 0\),
\- \(s = \varepsilon \Rightarrow s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0, s_n = s'\) and \(s_i - \tau \Rightarrow s_{i+1}\) when \(1 \leq i \leq n\),
\- \(s = b \Rightarrow s'\) if and only if there are \(s_1, s_2 \in S\) such that \(s = \varepsilon \Rightarrow s_1 - b \Rightarrow s_2 = \varepsilon \Rightarrow s'\),
\- \(s = b_1 b_2 \cdots b_n \Rightarrow s'\) if and only if there are \(s_0, s_1, \ldots, s_n \in S\) such that \(s = s_0, s_n = s'\) and \(s_i - b_i \Rightarrow s_{i+1}\) when \(1 \leq i \leq n\),
\- \(s = b_1 b_2 \cdots b_n \Rightarrow s'\) if and only if there is \(s' \in S\) such that \(s = b_1 b_2 \cdots b_n \Rightarrow s'\),
\- \(s = b_1 b_2 b_3 \cdots \mapsto s'\) if and only if there are \(s_0, s_1, s_2, \ldots \in S\) such that \(s = s_0\) and \(s_i = b_i \Rightarrow s_{i+1}\) when \(i \geq 0\).

We need the following semantic sets extracted from an LTS. A trace of an LTS is the
sequence of visible actions generated by any finite execution that starts in the initial state.
An infinite execution that starts in the initial state generates either an infinite trace or a divergence trace, depending on whether the number of visible actions in the execution is infinite or finite. The stable failures describe the ability of the LTS to refuse actions after executing a particular trace.

**Definition 3** Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS.
- $Tr(L) = \{ \sigma \in \Sigma^* \mid \hat{s} = \sigma \Rightarrow \}$ is the set of traces of $L$.
- $Inftr(L) = \{ \xi \in \Sigma^* \mid \hat{s} = \xi \Rightarrow \}$ is the set of infinite traces of $L$.
- $Divtr(L) = \{ \sigma \in \Sigma^* \mid \exists s \in S : \hat{s} = \sigma \Rightarrow s \land s - \tau^\omega \Rightarrow \}$ is the set of divergence traces of $L$.
- $Sfail(L) = \{ (\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : \hat{s} = \sigma \Rightarrow s \land \forall a \in A \cup \{ \tau \} : \neg(s - a \Rightarrow) \}$ is the set of stable failures of $L$.

The parallel composition operator defined below forces precisely those component processes to participate in the execution of a visible action that have the action in their alphabets. The invisible action is always executed by exactly one component process at a time. We first define the product of LTSs as the LTS that satisfies the above description and as its set of states the Cartesian product of the component state sets. We then define parallel composition by picking the part of the product that is reachable from the initial state of the product.

**Definition 4** Let $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1), \ldots, L_n = (S_n, \Sigma_n, \Delta_n, \hat{s}_n)$ be LTSs. Their product is the LTS $(S', \Sigma, \Delta', \hat{s})$ such that the following hold:
- $S' = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$
- $((s_1, \ldots, s_n), (a, (s'_1, \ldots, s'_n))) \in \Delta'$ if and only if either
  - $a = \tau$, and $(s_i, \tau, s'_i) \in \Delta_i$ for some $1 \leq i \leq n$,
  - and $s_j = s'_j$ for all $1 \leq j \leq n, j \neq i$
  - $a \in \Sigma_i$ and for each $1 \leq i \leq n$ either $a \in \Sigma_i$ and $(s_i, a, s'_i) \in \Delta_i$, or
  - $a \notin \Sigma_i$ and $s_i = s'_i$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$

The parallel composition $L_1 \parallel \cdots \parallel L_2$ is the LTS $(S, \Sigma, \Delta, \hat{s})$ such that
- $S = \{ s \in S' \mid \exists \sigma \in \Sigma^* : \hat{s} = \sigma \Rightarrow s \}$
- $\Delta = \Delta' \cap (S \times (\Sigma \cup \{ \tau \}) \times S)$

It is straightforward to show that “$\parallel$” is symmetric and associative, so that $L_1 \parallel L_2 \equiv L_2 \parallel L_1$ and $(L_1 \parallel L_2) \parallel L_3 \equiv L_1 \parallel (L_2 \parallel L_3)$, where “$\equiv$” denotes isomorphism. Therefore, if we wish, we can discard the parentheses and write $L_1 \parallel L_2 | L_3$, and similarly with any number of processes.

In one of the proofs we will need the following result, which gives the semantic sets of the parallel composition as functions of the sets of the parameter processes; see e.g. [39].

**Proposition 5** Let $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ be LTSs.
- $Tr(L_1 | L_2) = \{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid restr(\sigma, \Sigma_1) \in Tr(L_1) \land restr(\sigma, \Sigma_2) \in Tr(L_2) \}$
- $Divtr(L_1 | L_2) = \{ \sigma \in Tr(L_1 | L_2) \mid restr(\sigma, \Sigma_1) \in Divtr(L_1) \lor restr(\sigma, \Sigma_2) \in Divtr(L_2) \}$
\[ \text{Inftr}(L_1 \parallel L_2) = \{ \xi \in (\Sigma_1 \cup \Sigma_2)^\omega \mid \begin{array}{l}
\text{ restr}(\xi, \Sigma_1) \in \text{Inftr}(L_1) \land \text{ restr}(\xi, \Sigma_2) \in \text{Inftr}(L_2)
\end{array} \} \]

\[ \begin{array}{l}
\text{ restr}(\xi, \Sigma_1) \in \text{Inftr}(L_1) \land \text{ restr}(\xi, \Sigma_2) \in \text{Tr}(L_2) \\
\text{ restr}(\xi, \Sigma_1) \in \text{Tr}(L_1) \land \text{ restr}(\xi, \Sigma_2) \in \text{Inftr}(L_2)
\end{array} \}

\[ \text{Sfail}(L_1 \parallel L_2) = \{ (\sigma, A) \in (\Sigma_1 \cup \Sigma_2)^* \times 2^{\Sigma_1 \cup \Sigma_2} \mid \exists (\rho_1, B_1) \in \text{Sfail}(L_1), (\rho_2, B_2) \in \text{Sfail}(L_2) \mid \text{ restr}(\sigma, \Sigma_1) = \rho_1 \land \text{ restr}(\sigma, \Sigma_2) = \rho_2 \land A = B_1 \cup B_2 \} \]

The hiding operator converts given visible actions into \( \tau \)-actions and removes them from the alphabet.

**Definition 6** Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS and \( X \) any set of action names. Then hide \( X \) in \( L \) is the LTS \( (S, \Sigma', \Delta', \hat{s}) \) such that the following hold:

- \( \Sigma' = \Sigma - X \)
- \( (s, a, s') \in \Delta' \) if and only if \( a = \tau \land \exists b \in X : (s, b, s') \in \Delta, \) or \( a \notin X \land (s, a, s') \in \Delta. \)

Parallel composition and hiding have the following commutativity property which we will later need. When applied to LTSs, “\( = \)” means identity of LTSs.

**Proposition 7** Let \( L \) and \( L' \) be any LTSs and \( X \) any set of action names such that \( X \cap \Sigma(L') = \emptyset. \) Then (hide \( X \) in \( L \)) \( \parallel L' = \text{hide} \) \( X \) in \( L \parallel L'. \)

We now define the CFFD-model and CFFD-equivalence, which will be our main equivalence notion in this article. We also define CFFD-preorder. Intuitively, preorder means that the smaller process is “better” or “more deterministic” than the larger one. This can also be seen as an implementation relation, so that the larger process is a specification and the smaller process is an implementation of that specification.

**Definition 8** Let \( L \) and \( L' \) be LTSs with the same alphabet.

- The CFFD model of \( L \) is the 3-tuple \((\text{Sfail}(L), \text{Divtr}(L), \text{Inftr}(L))\)
- \( L \preceq_{\text{CFFD}} L' \iff \begin{array}{l}
\text{Sfail}(L) = \text{Sfail}(L') \land \text{Divtr}(L) = \text{Divtr}(L') \land \text{Inftr}(L) = \text{Inftr}(L')
\end{array} \)
- \( L \preceq_{\text{CFFD}} L' \iff \begin{array}{l}
\text{Sfail}(L) \subseteq \text{Sfail}(L') \land \text{Divtr}(L) \subseteq \text{Divtr}(L') \land \text{Inftr}(L) \subseteq \text{Inftr}(L')
\end{array} \)

The traces are not included in the CFFD model because they can be determined from \( \text{Sfail} \) and \( \text{Divtr} \) by the equation \( \text{Tr}(L) = \text{Divtr}(L) \cup \{ \sigma \in \Sigma^* \mid (\sigma, \emptyset) \in \text{Sfail}(L) \} \) [39].

It should be noted that when certain process-algebraic operators are used, a component called stability must be included in the CFFD model. This one bit of information tells whether or not there are \( \tau \)-transitions from the initial state of the LTS. However, with parallel composition and hiding this component is not needed, so we will not use it here.

An important property of an equivalence is that when a component process in a system is replaced by an equivalent process, the system should remain equivalent to the original one. This is formally captured by the congruence property. Similarly, a preorder should be a precongruence (monotonic), meaning that when a component process is replaced by a smaller or equivalent process the system will be smaller or equivalent.
Our process-algebraic framework. We next present a straightforward adaptation of the logic to
The desired properties of reactive and concurrent systems are often expressed by using
\[22, 27\]. We next present a straightforward adaptation of the logic to our process-algebraic framework.

3 LTSs, Temporal Logic and Fairness Operators

The desired properties of reactive and concurrent systems are often expressed by using linear temporal logic [22, 27]. We next present a straightforward adaptation of the logic to our process-algebraic framework.

Definition 9  An equivalence “\(\simeq\)” is a congruence with respect to a process operator \(op(L_1, \ldots, L_n)\) if and only if \(L_1 \simeq L'_1 \land \cdots \land L_n \simeq L'_n\) implies \(op(L_1, \ldots, L_n) \simeq op(L'_1, \ldots, L'_n)\).

Definition 10  A preorder “\(\leq\)” is a precongruence with respect to a process operator \(op(L_1, \ldots, L_n)\) if and only if \(L_1 \leq L'_1 \land \cdots \land L_n \leq L'_n\) implies \(op(L_1, \ldots, L_n) \leq op(L'_1, \ldots, L'_n)\).

CFFD-equivalence/preorder is a congruence/precongruence with respect to parallel composition and hiding, as shown in [39], for example.

The violation of a liveness property can only be detected in an infinite execution. Properties express requirements of the form “something good must eventually happen”.

Safety properties express requirements of the form “nothing bad must ever happen”.

Below we will use the following notation: if \(a\) then \(\psi\) else \(\phi\).

The semantics of formulas are defined on the infinite executions of systems.

Definition 11  A formula is generated by the grammar

\[\psi ::= \text{true} \mid a \mid En(a) \mid \neg \psi \mid \psi \lor \phi \mid \psi \land \psi,\]

where \(a\) is an action name. We also use the following denotations: \(false \equiv \neg true\), \(\psi \land \phi \equiv \neg (\neg \psi \lor \neg \phi)\), \(\psi \Rightarrow \phi \equiv \neg \psi \lor \phi\), \(\diamond \psi \equiv true \lor \psi\), \(\square \psi \equiv \neg \neg \psi\).

The semantics of formulas are defined on the infinite executions of systems.

Definition 12  Let \(L = (S, \Sigma, \Delta, s)\) be an LTS. The set of the infinite executions of \(L\) is \(\text{infex}(L) = \{s_0a_1s_1a_2s_2a_3 \cdots \mid s = s_0 \land \forall i \geq 1 : s_i-1 -a_i \rightarrow s_i\}\).

Below we will use the following notation: if \(\eta = s_0a_1s_1a_2s_2a_3 \cdots\) is an infinite execution, then \(\text{acts}(\eta)\) is the sequence of actions \(a_1a_2a_3 \cdots\) and \(\eta^j\) is the \(j\)th suffix \(s_0a_{i+1}s_{i+1}a_{i+2} \cdots\).

Definition 13  Let \(L = (S, \Sigma, \Delta, s)\) be an LTS and \(\eta = s_0a_1s_1a_2s_2a_3 \cdots\) an infinite execution of \(L\). Then

\(L, \eta) \models true\)
\(L, \eta) \models a\) iff \(a_1 = a\)
\(L, \eta) \models En(a)\) iff \(s_0 -a \rightarrow\) (that is, \(\exists s \in S : (s_0, a, s) \in \Delta)\)
\(L, \eta) \models \neg \psi\) iff not \((L, \eta) \models \psi\)
\(L, \eta) \models \psi \lor \phi\) iff \((L, \eta) \models \psi\) or \((L, \eta) \models \phi\)
\(L, \eta) \models \psi \land \phi\) iff \(\exists j \geq 0 : (L, \eta^j) \models \phi\) and \(\forall k, 0 \leq k < j : (L, \eta^k) \models \psi\)

The properties of reactive systems are usually divided into safety and liveness properties [2, 19]. Safety properties express requirements of the form “nothing bad must ever happen”. The violation of a safety property can always be detected in a finite execution. Liveness properties express requirements of the form “something good must eventually happen”. The violation of a liveness property can only be detected in an infinite execution.
Nondeterministic behaviour is characteristic of concurrent systems. In a typical situation we do not wish to, and cannot, limit the individual nondeterministic choices. However, we may still want that our system behaves “fairly” in the sense that it does not infinitely favour certain choices at the expense of others. Liveness properties that express this type of “fair” behaviour are called fairness properties.

Two well-known classes of fairness properties are weak fairness and strong fairness. Weak fairness with respect to action $a$ means that if $a$ is, from some point on, continuously enabled, then it must be executed infinitely often. This can be expressed with the formula

$$\Diamond \Box \text{En}(a) \Rightarrow \Box \Diamond a.$$  

Strong fairness means that if the action is enabled infinitely often, then it must be executed infinitely often. This can be expressed with $\Box \Diamond \text{En}(a) \Rightarrow \Box \Diamond a$.

It is customary in the verification of liveness properties to assume that the system satisfies some fairness constraint. A fairness constraint is a fairness property that is assumed, rather than proved, of the system. It formalises the idea that the underlying system behaves fairly with respect to the choices that we are interested in. The benefit of first constructing an “unfair” system model and then adding a fairness constraint is that we can usually make the system model much simpler if we do not attempt to build fairness-related details about the system. It formalises the idea that the underlying system behaves “fairly” in the sense that it does not infinitely violate certain choices at the expense of others. Liveness properties that express this type of “fair” behaviour are called fairness properties.

Let us assume that there is some fairness constraint $\phi$ that we would like to express in our process-algebraic framework. We would like to have a corresponding “fairness operator” $\Phi_\phi$ that, given an LTS $L$, produces a new LTS $L'$ whose finite behaviour (safety properties) is like that of $L$, but whose infinite executions fulfill the given fairness constraint. More precisely, we want all traces, $\text{Tr}(L)$, and stable failures, $\text{Sfail}(L)$, to stay the same (so that deadlocks are not affected), and exactly those infinite traces, $\text{Inftr}(L)$, and divergence traces, $\text{Divtr}(L)$, to remain that are created by some infinite execution in compliance with $\phi$. These requirements are stated formally in the following.

**Definition 14** An operator $\Phi_\phi$ is a fairness operator for the formula $\phi$ if and only if for every LTS $L = (S, \Sigma, \Delta, \hat{s})$ each of the following holds:

- $\text{Tr}(\Phi_\phi(L)) = \text{Tr}(L)$
- $\text{Sfail}(\Phi_\phi(L)) = \text{Sfail}(L)$
- $\text{Divtr}(\Phi_\phi(L)) = \{ \sigma \in \Sigma^* \mid \exists \eta \in \text{infex}(L) : ((L, \eta) \models \phi) \land \text{restr}(\text{acts}(\eta), \Sigma) = \sigma \}$
- $\text{Inftr}(\Phi_\phi(L)) = \{ \xi \in \Sigma^* \mid \exists \eta \in \text{infex}(L) : ((L, \eta) \models \phi) \land \text{restr}(\text{acts}(\eta), \Sigma) = \xi \}$

It is important to notice that we have not yet constructed a fairness operator, we have just stated desired properties of a (hypothetical) operator. An obvious requirement is also that any equivalence we use should be a congruence with respect to the fairness operator.

However, it turns out that the above properties are not easy to achieve. Consider the three LTSs $L$, $L'$ and $L''$ in Figure 1. These are all CFFD-equivalent. The same holds for most process-algebraic semantic models. If we apply weak fairness towards $a$, expressed with $\phi(a) \equiv \Diamond \Box \text{En}(a) \Rightarrow \Box \Diamond a$, the divergence in the initial state of $L$ disappears. However, $a$ is not continuously enabled in $L'$, so the fairness assumption does not remove the divergence there. Thus, $\varepsilon \notin \text{Divtr}(\Phi_\phi(a)(L))$, but $\varepsilon \in \text{Divtr}(\Phi_\phi(a)(L'))$, and the results are not equivalent. We can try using strong fairness instead, because this forces execution
of $a$ even in $L'$. However, by comparing $L$ and $L''$ we can similarly see that strong fairness leads to non-equivalent results. The conclusion from this counter-example is thus the following:

**Proposition 15** “$\equiv_{\text{CFFD}}$” cannot be a congruence with respect to any fairness operator $\Phi_{\phi(a)}$ for formula $\phi(a) \equiv \Diamond \Box E n(a) \Rightarrow \Box \Diamond a$ or $\Diamond \Diamond E n(a) \Rightarrow \Box \Diamond a$.

This result could equally well be formulated in terms of the failures-divergences model of CSP and other similar equivalences. For instance, $L \equiv_{\text{CSP}} L'' \equiv_{\text{CSP}} \Phi_{\phi(a)}(L'') \equiv_{\text{CSP}} \text{div} \not\equiv_{\text{CSP}} \Phi_{\phi(a)}(L)$, where $\text{div}$ is the CSP-least process [31].

Clearly the reason why most models are not congruences with respect to the fairness operator is that they do not preserve enough information on the enabledness of actions during infinite executions. A notable exception is the strong bisimilarity of [23], but as is well known, it treats invisible actions no differently from visible actions, and thus does not allow us to abstract them away.

Furthermore, we would like to make one more “soundness” requirement for the hypothetical fairness operator. This is because the fairness operator would typically be applied to some process $L$ (e.g., a communication channel) which can be placed in a larger context $C[\cdot]$ (e.g., a protocol system). The property of the underlying system expressed by the fairness constraint should remain the same in the larger context. Therefore, within some reasonable limits, it should make no difference whether the same fairness constraint is assumed of $L$ or of the composition $C[L]$. (Practical application of this principle is demonstrated in Section 5.) Thus, the fairness operator should ideally have the following property of context-independence, which essentially means (limited) commutativity with parallel composition and hiding.

**Definition 16** Let $\Phi_{\phi}$ be a fairness operator for formula $\phi$ which is expressed in terms of the actions $\mathcal{F}$. We say that $\Phi_{\phi}$ is context-independent with respect to “$\equiv$”, if and only if “$\equiv$” is a congruence with respect to $\Phi_{\phi}$, and for every LTS $L$ with $\mathcal{F} \subseteq \Sigma(L)$ it holds that

- $\Phi_{\phi}(L) \equiv L' \equiv L''$ for any LTS $L'$ (notice that “$\equiv$” is commutative), and
- hide $X$ in $\Phi_{\phi}(L) \equiv \Phi_{\phi}(\text{hide} X \text{ in } L)$ for any $X$ such that $X \cap \mathcal{F} = \emptyset$.

However, as readers familiar with process algebra may already expect, the property of context-independence cannot be generally true for an equivalence that preserves liveness properties, even if it were as detailed as strong bisimulation. This is essentially because other processes can interfere with the actions that we use in the fairness constraint.

As a simple counter-example, if we want process $A$ in Figure 2 to always get a chance to eventually execute $a$ in the combination $A || B$, we can declare either weak or strong fairness

![Figure 1. The processes L, L' and L''](attachment://image.png)
with respect to $a$. The resulting process has no executions ending in an infinite sequence of $b$’s, so there are none even when this process is combined with $C$. Consequently, there are no divergences after we hide $b$ from the result: $\text{Divtr}(\text{hide } b \text{ in } (\Phi_{\phi(a)}(A \parallel B) \parallel C)) = \emptyset$.

However, if we make the parallel composition $A \parallel B \parallel C$ first and then add the fairness constraint, the resulting system is able to execute $b^\omega$ because $a$ is not enabled in the combination $A \parallel B \parallel C$. This turns into a divergence when we hide $b$ from the result: $\varepsilon \in \text{Divtr}(\text{hide } b \text{ in } \Phi_{\phi(a)}(A \parallel B \parallel C))$. This shows that if an equivalence distinguishes whether or not there are infinite $\tau$-executions in a process, the fairness operator $\Phi_{\phi(a)}$ cannot be context-independent with respect to it. (If we, instead, had a “fair parallel composition” operator, we could use similar reasoning to show that such parallel composition would not be associative.)

**Proposition 17** Let “$\simeq$” be an equivalence such that $L \simeq L'$ implies $\text{Divtr}(L) = \emptyset$ if and only if $\text{Divtr}(L') = \emptyset$, and let $\phi(a)$ be either $\bigtriangleup \Box En(a) \Rightarrow \Box \bigtriangleup a$ or $\Box \bigtriangleup En(a) \Rightarrow \bigtriangleup \Box a$. Then, no fairness operator $\Phi_{\phi(a)}$ for $\phi(a)$ is context-independent with respect to “$\simeq$”.

The same example can also be used to illustrate the problem with finite representation of fairness constraints that was mentioned in Section 1 and which will be discussed further in Section 6. For example, if we were to use “infinitary restriction” (as in [26]) and, say, restricted the infinite executions of $A \parallel B$ to the set $(b^* a)^\omega$, that is, to the set $\{ \xi \in \{a,b\}^\omega \mid \text{restr} (\xi, \{a\}) = a^\omega \}$, this would not cause any immediate problems. However, the result of combining this process with $C$ would be equivalent to $A \parallel B \parallel C$ with an empty set of allowed infinite execution sequences. This is obviously an “impossible” situation, because all the system can do is execute $b$’s, and it is not allowed to continue in this way forever, nor terminate and refuse further $b$’s.

Also, this example can be used to illustrate the difficulties involved in developing semantic models for the kind of fair operational rules introduced in [9, 10] (see Section 6). Consider the system $A \parallel B$. Under the fair operational rules, this process does not have the infinite execution $b^\omega$ (which is unfair for $A$). However, when placed in a context that does not even involve $b$ (namely $C$), the result is capable of executing $b^\omega$.

In the next section we will present a partial solution that does not suffer from any of the above-mentioned problems. The approach is compatible with the CFFD-semantics, and therefore allows us to use existing CFFD-based verification tools. We achieve this by restricting our scope in two ways. Firstly, we consider only a specific class of fairness constraints. Secondly, we restrict the set of processes to which we apply the constraints.
4 Fairness LTSs as Fairness Operators

4.1 A Class of Fairness Constraints

In this section we will consider a class of fairness constraints that relate the infinite occurrence of different actions. Typical constraints in this class are of the form “if something happens infinitely many times then something else also has to happen infinitely many times”.

The class of fairness properties we will deal with is the set of formulas of the form

\[ \alpha \Rightarrow \beta \]

where \( \alpha \) and \( \beta \) are any formulas constructed from action names by using the operators “\( \lor \)”, “\( \land \)” and “\( \Box \Box \)” (“infinitely often”), with the restriction that every action name must reside within the scope of at least one “\( \Box \Box \)”-operator (because we are interested in fairness properties, not individual actions). More precisely, \( \alpha \) and \( \beta \) are any formulas generated by the grammar

\[ \phi ::= \text{false} | \phi \land \phi | \phi \lor \phi | \Box \Box \phi_1 \]

\[ \phi_1 ::= \phi_1 \land \phi_1 | \phi_1 \lor \phi_1 | \Box \Box \phi_1 | a \]

where \( a \) is any visible action name. Because we allow \( \beta \) to be \( \text{false} \), the complete formula may become \( \neg \alpha \).

Let us denote the set of formulas of this form by \( F \).

We next transform the fairness formulas into a normal form where, for technical convenience, the left side is in a conjunctive form and the right side in a disjunctive form.

**Theorem 18** Every formula in \( F \) that is not trivially \( \text{true} \) can be given in the form

\[ A_1 \land A_2 \land \cdots \land A_m \Rightarrow B_1 \lor B_2 \lor \cdots \lor B_n \]

where \( m > 0, n \geq 0 \), \( A_i = \Box \Box a_i^1 \lor \Box \Box a_i^2 \lor \cdots \lor \Box \Box a_i^u_i \), and \( B_j = \Box \Box b_j^1 \land \Box \Box b_j^2 \land \cdots \land \Box \Box b_j^v_j \), where \( u_i > 0 \), \( v_j > 0 \) and \( a_i^k \) and \( b_j^l \) are action names (notice that the same action name may occur several times). If \( n = 0 \) then the formula is

\[ \neg(A_1 \land A_2 \land \cdots \land A_m). \]

**Proof** The manipulation is otherwise standard, but the operator “\( \Box \Box \)” needs special attention. We always get rid of subformulas of the form “\( \text{false} \)” using “\( \Box \Box \text{false} \equiv \text{false}, \text{false} \land \psi \equiv \text{false}, \text{false} \lor \psi \equiv \psi \).” In the sequel, we will also use the fact that

\[ \Box \Box (\Box \Box \psi_1 \land \Box \Box \psi_2 \land \cdots \land \Box \Box \psi_k) \equiv \Box \Box \psi_1 \land \Box \Box \psi_2 \land \cdots \land \Box \Box \psi_k. \]

Assume we have any subformula of the form \( \Box \Box \omega \). We first identify subformulas of \( \omega \) that are either single action names or of the form \( \Box \Box (\cdots) \), and we transform \( \omega \) into disjunctive normal form by using these subformulas as propositions. Denote the disjuncts in the result by \( \omega_1, \omega_2, \ldots, \omega_k \). It clearly holds that

\[ \Box \Box (\omega_1 \lor \omega_2 \lor \cdots \lor \omega_k) \equiv \Box \Box \omega_1 \lor \Box \Box \omega_2 \lor \cdots \lor \Box \Box \omega_k. \]

Each subformula \( \omega_i \) is of the form \( \omega_i^1 \land \cdots \land \omega_i^j \), where \( \omega_j^i \) is either an action name or a formula beginning
with “□ ◻”. If, for a particular \( i \), all the \( \omega_j \) begin with “□ ◻” then, by the above, we can dispose of the “□ ◻” in front of the disjunct. Otherwise, if more than one of the \( \omega_j \) are action names and these contain different actions, then the subformula is identically false, since obviously no action in an execution can be two different actions at the same time. If the same action name appears many times, we can remove all but one. Therefore, we can assume without loss of generality that only one, say, \( \omega_1 \) is an action name and all the others are formulas that begin with “□ ◻”. Then, it is easy to see that

\[
\square ◻ (\omega_1 \land \omega_2 \land \cdots \land \omega_l) \equiv (\square ◻ \omega_1) \land \omega_2 \land \cdots \land \omega_l.
\]

Our formula is now either false or consists of subformulas of the form “□ ◻(…)” that are connected with “∨” and “∧”. We can repeat the above operation on these subformulas, until all “□ ◻”-operators have been pushed down to the level of individual actions.

If \( \alpha = false \), then the fairness property is trivially true. Otherwise, we can transform \( \alpha \) into conjunctive normal form where the propositions are of the form “□ ◻a”, with \( a \) an action name. If \( \beta = false \), we are left with \( \neg(A_1 \land A_2 \land \cdots \land A_m) \).

If \( \beta \neq false \), we can transform it into disjunctive normal form.

From now on we will denote a formula of the above form by

\[
\psi(A_1, \ldots, A_m; B_1, \ldots, B_n).
\]

In the following, we will also use \( A_i \) and \( B_j \) to denote the sets of actions \( \{a_{i1}^1, a_{i2}^1, \ldots, a_{iu_i}^1\} \) and \( \{b_{j1}^1, b_{j2}^1, \ldots, b_{jv_j}^1\} \), respectively. Further, we write \( A \) for \( A_1 \cup \cdots \cup A_m \) and \( B \) for \( B_1 \cup \cdots \cup B_n \).

### 4.2 Fairness LTS

Before we define a fairness operator for our class of fairness constraints, we will first define a “fairness LTS” that reflects the fairness property in its structure. The actual fairness operator works by placing the parameter system in parallel with the fairness LTS.

Because we want to restrict only infinite executions, the fairness LTS must have all possible finite traces. On the other hand, it should have only those infinite traces that model the fairness property. Obviously, it should not have any divergences. As for the stable failures, we have a choice to make. Basically, we would prefer to have as few stable failures as possible, that is, only the ones that are needed to restrict the infinite executions. However, it is straightforward to show that among the possible candidates for a fairness LTS there does not exist a minimum element with respect to stable failures, because it is always possible to remove some further stable failures. Fortunately, it suffices for our purposes if we allow a fairness LTS to always refuse any subset of \( A \).

**Definition 19** \( L = (S, \Sigma, \Delta, \hat{s}) \) is a fairness LTS for \( \psi(A_1, \ldots, A_m; B_1, \ldots, B_n) \) if and only if \( \Sigma = A \cup B \) and

- \( Tr(L) = \Sigma^* \)
- \( Divtr(L) = \emptyset \)
- \( Sfail(L) \subseteq \Sigma^* \times 2^A \)
- \( Inftr(L) = \{ \eta \in \Sigma^* | \eta \models \psi \} \)

135
We need to show that an LTS with these properties exists. As the actual construction of the LTS is somewhat involved, we will first give some illustrating examples of fairness constraints and the corresponding fairness LTSs. It should be noted that the fairness LTS has to be infinitely branching, because there does not exist a finitely branching divergence-free LTS with all finite traces but only the allowed infinite traces. This does not cause any problems on the denotational level if we are using a semantics that is a congruence for infinitely branching systems, and CFFD is such a semantics. We would also like to emphasise that the fairness LTS is a theoretical tool by which we can add a fairness assumption to a system in a consistent way by using existing theoretical machinery. When applying fairness in verification we do not have to construct the actual infinite LTS, as will be explained in the next section.

Figure 3 (a) illustrates a fairness LTS corresponding to the fairness constraint \( \neg \Box a \). It has \( a \)-branches of length 1, 2, 3, … from the initial state, so it can execute any finite number of \( a \)-actions, but not infinitely many. Figure 3 (b) illustrates a fairness LTS corresponding to \( \Box a \Rightarrow \Box b \). The structure is the same as in (a) but now from every state (including the initial state) it is possible to return to the initial state with \( b \). Therefore the LTS can now execute even an infinite number of \( a \)-actions, but these cannot be executed consecutively, because there have to be intermediate \( b \)-actions.

We can use the same basic structure also when we add more than one action in a disjunctive manner. For example, for the fairness constraint \( \Box a \lor \Box c \Rightarrow \Box b \lor \Box d \) we can take the LTS in Figure 3 (b) and simply add one \( c \)-transition in parallel with each \( a \)-transition and one \( d \)-transition in parallel with each \( b \)-transition.

Next we will illustrate the type of construct we use for conjunctive formulas. Figure 4 shows a fairness LTS that corresponds to the fairness constraint \( \Box a \land \Box c \Rightarrow \Box b \land \Box d \). Like before, there are branches of length 1, 2, 3, etc. However, now there are two branches of each length, one of which limits action \( a \) to the given finite number and the other which limits action \( c \). Furthermore, each branch has an internal structure which keeps track of which of the two actions \( b \) and \( d \) have been detected so far. Once both \( b \) and \( d \) have been detected we return to the initial state.
After these examples we will next present the general definition of a fairness LTS. We would again like to emphasise that for the present discussion it is not the detailed structure of the fairness LTS that is important, but we need the construction to establish that for each formula in our class of fairness constraints there exists a corresponding LTS with the properties given in Definition 19 above.

Intuitively, in the construction below the set of states of the fairness LTS consists of the initial state (an empty tuple) and tuples of the form \((A, l, r, B)\). Each “branch” of the fairness LTS is constructed by selecting one of the sets \(A_1, A_2, \ldots, A_m\) as \(A\). The execution of the actions in \(A\) is restricted to a finite number in the branch. In the examples in Figure 3 (a) and (b), \(A\) is always \(\{a\}\). In Figure 4, \(A\) is alternately \(\{a\}\) or \(\{c\}\). Notice that the two shortest branches in Figure 4 represent the case when the restricted action is executed first and refused immediately after that.

The number \(l\) (starting from 0) in the tuple is the maximum finite number of these
actions after we have entered the branch with the first action, so that \( l + 1 \) is the “length” of the branch. The number \( r \) is used to count how much of this finite number remains, that is, how many of the given actions can still be executed. Therefore, it also denotes the position of the state from the final state of the branch.

We can return to the initial state once all the actions from some set \( \mathcal{B}_j \) have been executed. The set \( B \) in the tuple records the \( \mathcal{B} \)-actions. When we execute an action \( b \) that would make one of the sets \( \mathcal{B}_j \) full, i.e., \( \mathcal{B}_j \subseteq B \cup \{ b \} \), we return to the initial state. In Figure 3 (a) there are no \( \mathcal{B}_j \)-sets. In Figure 3 (b) we have one singleton set \( \mathcal{B}_1 = \{ b \} \), and \( B \) is always the empty set. In Figure 4 we have the set \( \mathcal{B}_1 = \{ b, d \} \), and \( B \) is either \( \emptyset \), \( \{ b \} \) or \( \{ d \} \).

It should be noted that this is not the only possible LTS that fulfills the requirements of Definition 19. Here we have mainly aimed at a straightforward construction that corresponds as closely as possible to the intuitive idea of a fairness requirement.

**Definition 20** For a formula \( \psi(A_1, \ldots, A_m; \mathcal{B}_1, \ldots, \mathcal{B}_n) \), let \( L_\psi \) be the LTS \((S, \Sigma, \Delta, s)\), where

- \( S = \{ () \} \cup \{ (A, l, r, B) \mid \exists i \in \{ 1, \ldots, m \} : A = A_i \land 0 \leq r \leq l \land B \subseteq \mathcal{B} \land \forall j \in \{ 1, \ldots, n \} : \mathcal{B}_j \not\subseteq B \} \)
- \( s = () \)
- \( \Sigma = A \cup \mathcal{B} \)
- \( \Delta = \{ s \} \times \mathcal{B} \times \{ s \} \cup \{ (s, a, (A, l, l, \emptyset)) \in \{ s \} \times A \times S \mid l \geq 1 \lor a \in A \} \)
- \( \cup \{ ((A, l, r, B), a, (A, l, r-1, B)) \in S \times A \times S \mid a \in A \} \)
- \( \cup \{ ((A, l, r, B), a, (A, l, r, B)) \in S \times A \times S \mid a \not\in A \} \)
- \( \cup \{ ((A, l, r, B), b, (A, l, r, B \cup \{ b \})) \in S \times \mathcal{B} \times S \mid \} \)
- \( \cup \{ ((A, l, r, B), b, s) \in S \times \mathcal{B} \times \{ s \} \mid \exists j : \mathcal{B}_j \subseteq B \cup \{ b \} \} \)

The following proposition states that the LTS \( L_\psi \) defined above has the desired properties of a fairness LTS.

**Theorem 21** \( L_\psi \) is a fairness LTS for \( \psi(A_1, \ldots, A_m; \mathcal{B}_1, \ldots, \mathcal{B}_n) \).

**Proof** \( L_\psi \) has all possible finite traces from its alphabet, because we can execute any finite trace \( \sigma \) by selecting a branch that is longer than the number of \( A \)-actions in \( \sigma \). Clearly, \( L_\psi \) has no divergences. It is also easy to see that all states except those where \( r = 0 \) have outgoing transitions for every action of the alphabet, and the states with \( r = 0 \) can only refuse actions in \( A \) (which is allowed). Thus, the Sfail-condition holds.

It remains to show that \( L_\psi \) has precisely the infinite traces allowed by \( \psi \). Assume first that \( \psi \models \xi \). We need to identify an infinite execution of \( L_\psi \) whose sequence of visible actions is \( \xi \). Either of the following has to hold of \( \xi \):

a) \( \neg (A_1 \land \cdots \land A_m) \). Thus, there is at least one \( A_i \) such that all actions \( a^i_k \in A_i \) occur only finitely many times in \( \xi \). Therefore, there are altogether only some finite number \( q \) of the actions of \( A_i \) in \( \xi \). Let us choose a branch of \( L_\psi \) where \( A = A_i \) and whose length is greater than \( q \), that is, where the states are of the form \( (A_i, l, r, B) \) with \( l \geq q \). In the initial state we can freely execute any \( \mathcal{B} \)-actions. If an \( A \)-action
occurs in $\xi$ we can jump to the above-mentioned branch. We can proceed in this way also if we have returned to the initial state after all actions from some $B_j$-set have been executed. The fairness LTS never refuses any $B$-actions, and the branch we have chosen cannot refuse any $A$-actions before all $A$-actions in $\xi$ have been executed.

b) $B_1 \lor \cdots \lor B_n$. Thus, there is at least one $B_j$ such that every action $b_{ij} \in B_j$ occurs infinitely many times in $\xi$. We can construct the execution of $\xi$ in $L_\psi$ by repeating the following procedure. We stay in the initial state until we come across the next $A$-action; if there are none, we can stay in the initial state forever. We identify the shortest subsequence $\chi$ of $\xi$ following this $A$-action which contains all the actions of some $B_j$; this exists because all the actions of the above-mentioned $B_j$ occur infinitely many times in $\xi$. We then choose a branch whose length is greater than the total number of $A$-actions in the subsequence $\chi$. Therefore, we can execute $\chi$ in this branch, and then with the last action one $B_j$ set would be completed and we return to the initial state. Then, we can repeat the procedure.

Finally, assume $\psi \not\models \xi$. We need to show that $L_\psi$ cannot execute the infinite trace $\xi$. It now holds that $A_1 \land \cdots \land A_m \land \neg B_1 \land \cdots \land \neg B_n$. Therefore, in each $A_i$ there is at least one $a_{ik} \in A_i$ which occurs infinitely many times in $\xi$. On the other hand, in every $B_j$ there is at least one $b_{lj} \in B_j$ which occurs only finitely many times in $\xi$. Because of the latter, there is an end part $\eta$ of $\xi$ such that no $b_{lj}$ occurs in it. Since there are infinitely many $A$-actions, also $\eta$ contains $A$-actions. Thus, if we are not already in some branch (i.e., we are in the initial state) of the fairness LTS when we start to execute $\eta$, we have to enter some branch when we encounter the first $A$-action of $\eta$. However, we can never leave the branch while executing $\eta$, because no $B_j$-set can ever be completed. On the other hand, no branch allows infinitely many actions from every $A_i$.

4.3 Fairness Operator

As indicated earlier, before we define the fairness operator, we have to restrict the set of LTSs to which it can be applied. Intuitively, we could require that in these LTSs the actions $a_{ik}$ always start at unstable states (states with outgoing $\tau$-transitions). An unstable state can nondeterministically choose a $\tau$-transition instead of the $a_{ik}$-transition. Therefore, constraining these actions according to the formula $\psi$ only limits (in an infinitary way) the nondeterministic choices the system makes, precisely as a fairness constraint should. In mathematical terms, this guarantees that no new stable failures are created by refusing the $a_{ik}$-actions. In fact, it turns out that a closely related but weaker requirement suffices:

**Definition 22** LTS $L = (S, \Sigma, \Delta, s)$ is *compatible* with the fairness constraint $\psi(A_1, \ldots, A_m; B_1, \ldots, B_n)$ if and only if $A \cup B \subseteq \Sigma$ and $\forall (\sigma, X) \in S_{fail}(L): (\sigma, X \cup A) \in S_{fail}(L)$. The set of LTSs compatible with $\psi$ is denoted $COMP_\psi$.

Note that the given condition can be determined from the CFFD-model of an LTS, so its validity is preserved when a system is replaced by a CFFD-equivalent one. It is also straightforward to show that compatibility is insensitive to parallel composition and the hiding of unrelated actions, as stated in the following.
Proposition 23  Let $L$ and $L'$ be LTSs, and let $Y$ be any set of action names such that $Y \cap (A \cup B) = \emptyset$. If $L$ is in $COMP_{\psi}$, then $L \parallel L'$ and hide $Y$ in $L$ are also in $COMP_{\psi}$.

The fairness operator can now be defined simply as a parallel composition with a fairness LTS. We will use the LTS $L_{\psi}$ defined above, but it is easy to see that we could equally well use any LTS with the properties of a fairness LTS, as given in Definition 19, and the operator would produce an equivalent result in terms of CFFD.

Definition 24  For the formula $\psi(A_1, \ldots, A_m; B_1, \ldots, B_n)$, $\Psi_{\psi}$ is the following mapping from LTSs to LTSs: $\Psi_{\psi}(L) = L \parallel L_{\psi}$.

The following result states that $\Psi_{\psi}$ really is a fairness operator in the sense of Definition 14.

Theorem 25  When applied to LTSs from $COMP_{\psi}$, $\Psi_{\psi}$ is a fairness operator for $\psi(A_1, \ldots, A_m; B_1, \ldots, B_n)$.

Proof  Consider $\Psi_{\psi}(L) = L \parallel L_{\psi}$, where $L \in COMP_{\psi}$. We first show that the infinite traces of $\Psi_{\psi}(L)$ are precisely the infinite traces of $L$ that model $\psi$. For this class of formulas, an infinite trace $\xi$ models $\psi$ precisely when $restr(\xi, A \cup B)$ is either finite or models $\psi$. On the other hand, because $\Sigma(L_{\psi}) \subseteq \Sigma(L)$, the infinite traces of the parallel composition $L \parallel L_{\psi}$ are precisely those $\xi \in Inftr(L)$ where $restr(\xi, \Sigma(L_{\psi})) = restr(\xi, A \cup B)$ is either finite or is in $Inftr(L_{\psi})$. But $Inftr(L_{\psi})$ is the set of infinite traces from $(A \cup B)^{\omega}$ that model $\psi$ (Theorem 21), and this establishes the claim.

As for finite traces, $Tr(L_{\psi})$ contains all finite sequences of actions from $\Sigma(L_{\psi})$, which implies that $Tr(L \parallel L_{\psi}) = Tr(L)$. From this and $Divtr(L_{\psi}) = \emptyset$ it also follows that $Divtr(L \parallel L_{\psi}) = Divtr(L)$. This complies with the requirement because $\psi$ does not rule out executions with only finitely many visible actions. (Notice, however, that infinite traces can turn into divergences through hiding and therefore in a larger context divergences can be removed.)

Finally, assume $(\sigma, X) \in Sfail(L \parallel L_{\psi})$. Because $\Sigma(L_{\psi}) \subseteq \Sigma(L)$, $L$ has to execute the trace $\sigma$ when the parallel composition executes $\sigma$. Furthermore, because $L_{\psi}$ is only able to refuse actions from $A$, $L$ must have a stable failure $(\sigma, Y)$ such that $Y \cup A \supseteq X$. However, from the compatibility of $L$ it follows (Definition 22) that $(\sigma, Y \cup A) \in Sfail(L)$, and therefore also $(\sigma, X) \in Sfail(L)$. For the other direction, assume $(\sigma, X) \in Sfail(L)$. $L_{\psi}$ has all possible finite traces, and therefore it has also the trace $\rho = restr(\sigma, \Sigma(L_{\psi}))$. Since it cannot diverge, a stable state is reached after this trace. Therefore, $(\rho, \emptyset) \in Sfail(L_{\psi})$, and thus $(\sigma, X) \in Sfail(L \parallel L_{\psi})$. In conclusion, $Sfail(L \parallel L_{\psi}) = Sfail(L)$.

It should be noted that since our approach uses only ordinary LTSs as the model of a process, there can be no conflict between the underlying transition system and the additional fairness information like there can be in a “finite representation” approach (see the discussion in Sections 1 and 6 and the example at the end of Section 3). In our approach, if a process were unable to execute within the allowed infinite execution sequences, this
would present itself as new deadlocks. Therefore, the above result is especially important, because it shows that this will never happen. Furthermore, unlike in some other approaches, the fact that subprocesses can stop executing (for example, when blocked by others) does not cause any problems in our approach.

The congruence property for CFFD and the fairness operator follows directly from the fact that \( \simeq_{\text{CFFD}} \) is a congruence with respect to \( \| \| \).

**Proposition 26** \( \simeq_{\text{CFFD}} \) is a congruence with respect to \( \Psi \| \| \).

The important property of context-independence also holds for \( \Psi \| \| \) and \( \simeq_{\text{CFFD}} \).

**Proposition 27** \( \Psi \| \| \) is context-independent with respect to \( \simeq_{\text{CFFD}} \).

**Proof** Let \( L \) and \( L' \) be any LTSs. From the associativity and commutativity of \( \| \| \) it follows that \( \Psi \| \| (L \| L') = (L \| L\| L') \| L' \| L = \Psi \| \| (L \| L') \| L \| L' = \Psi \| \| (L \| L') \| L \), where \( \| \| \) denotes isomorphism and \( \simeq \) identity of LTSs. Secondly, assume that \( X \) is any set of actions with \( X \cap (A \cup B) = \emptyset \). Because \( A \cup B \) is the alphabet of \( L\psi \), the commutativity property of \( \| \| \) and \( \text{hide} \) (Proposition 7) shows that \( \text{hide} X \in \Psi \| \| (L) = \text{hide} X \in (L \| L\| L\psi) = (\text{hide} X \in L \| L\psi) \| L\psi = \Psi \| \| (\text{hide} X \in L) \).

It is also straightforward to show (Proposition 23 and the associativity and commutativity of \( \| \| \)) that compatibility is preserved by our fairness operators and that our fairness operators commute among themselves:

**Proposition 28** Let \( \Psi \| \| \) and \( \Phi \| \| \) be the fairness operators for the formulas \( \psi \) and \( \phi \), respectively, and let \( L \) be an LTS. Then
- if \( L \) is in \( \text{COMP}_\psi \), then \( \Phi \| \| (L) \) is also in \( \text{COMP}_\psi \),
- \( \Psi \| \| (\Phi \| \| (L)) \| L\psi = \Phi \| \| \Psi \| \| (L) \).

## 5 Verification

From the point of view of automated verification, a potential problem with the fairness operators is that they create infinite subprocesses. Fortunately, it often holds that a larger system, where the subprocesses are composed with other processes and actions are hidden, can be represented as a finite LTS. In this section we will show that in every case when this is true we can effectively construct a finite representation of the larger system. We avoid constructing the intermediate infinite processes by applying the context-independence property, because it allows us to move the fairness operators to a higher level of the system structure, where we can check their effect on the finite target system.

However, before we describe the general verification method in detail, we will illustrate the basic idea of our approach by removing livelocks from a simple protocol system.
5.1 Verification Example

Our example system is the well-known alternating bit protocol [4]. This protocol is intended for sending messages over channels that can lose messages, but cannot reorder them. There are two one-way channels, one for the data from the sender to the receiver and another for acknowledgements from the receiver to the sender, as depicted in Figure 5. Acknowledgements are needed because messages can be lost. If the acknowledgement for a message is not received in time, the protocol attempts retransmission. In order not to confuse new messages with retransmissions, each message and acknowledgement contains a sequence number, which is either 0 or 1.

Our LTS definitions of the sender ($S$), data channel ($DC$), acknowledgement channel ($AC$) and receiver ($R$) are shown in Figure 6. For simplicity we do not model the data content of messages, as it does not directly affect the behaviour of the protocol. After a sending request from the user, the sender puts a data message to the data channel with the appropriate bit value. If a correct acknowledgement is not received in time, it can send the same message any number of times. The sender can also read unexpected acknowledgements from the channel. The data channel gets a message from the sender and then chooses either to lose it or pass it through and give it to the receiver. The “pass data”-action is an invisible $\tau$-action, but we have made the “lose data”-action temporarily visible as $ld$ so that we can later define a fairness constraint for it. The acknowledgement channel works similarly. When the receiver gets a data message with a new bit value, it declares it with $rec$ and sends an acknowledgement. For repeated messages it only sends an acknowledgement.

We can construct and reduce the system by using the TVT toolset [14]. We hide
the internal actions \( I = \{sd0, sd1, ld, rd0, rd1, sa0, sa1, la, ra0, ra1\} \) from the system and leave only the external actions \{send, rec\} visible. Then the complete system \( P' = \text{hide} \ I \ \text{in} \ (S \| DC \| AC \| R) \) is reduced with a CFFD-preserving reduction algorithm. The result is shown in Figure 7. We note that the behaviour is otherwise acceptable, but there are two \( \tau \)-loops, or divergences, in the system. We cannot know with certainty that after entering one of these loops the system ever executes any more visible actions.

The channels \( DC \) and \( AC \) can lose all messages that are given to them, and we can therefore guess that the divergences are the result of an infinite sequence of retransmissions and losses of messages in the channels. Thus, we would expect that if we do not allow the channels to lose an infinite number of messages, the divergences should disappear.

With the fairness operators at our disposal, we can formally verify that this is the case. We add the fairness constraint \( \neg \square \diamond ld \) to channel \( DC \) and the constraint \( \neg \square \diamond la \) to channel \( AC \). We are allowed to use these constraints because the states where \( ld \) and \( la \) start are unstable, and thus \( DC \) and \( AC \) are in \( COMP_{\neg \square \diamond ld} \) and \( COMP_{\neg \square \diamond la} \), respectively. Thus, let

\[
P' = \text{hide} \ I \ \text{in} \ (S \| \Psi_{\neg \square \diamond ld}(DC) \| \Psi_{\neg \square \diamond la}(AC) \| R).
\]

Let \( I' = I - \{ld, la\} \). Then, because of the context-independence property we have the following equivalences:

\[
P' = \text{hide} \ I \ \text{in} \ (S \| \Psi_{\neg \square \diamond ld}(DC) \| \Psi_{\neg \square \diamond la}(AC) \| R)
\]

\[
\simeq_{\text{CFFD}} \text{hide} \ I \ \text{in} \ \Psi_{\neg \square \diamond ld}((S \| \Psi_{\neg \square \diamond la}(DC \| AC \| R)))
\]

\[
\simeq_{\text{CFFD}} \text{hide} \ \{ld, la\} \ \text{in} \ \Psi_{\neg \square \diamond ld}((\text{hide} \ I' \ \text{in} \ (S \| DC \| AC \| R)))
\]

The LTS of the inside process \( P' = \text{hide} \ I' \ \text{in} \ (S \| DC \| AC \| R) \) after CFFD-reduction is shown in Figure 8.

We can next determine the CFFD-model of \( P' \). We will do this manually now, but we will later in this section give an algorithm for the purpose. By Theorem 25, \( S_{\text{fail}}(P') = \)

![Figure 8](image-url)
Figure 9. The behaviour with the fairness constraints

\[ S_{fail}(P') \] As for \( Inftr \), the only infinite trace of \( P \) (Figure 7) is \( (send \; rec)^\omega \), and from Figure 8 we see that this infinite trace can be executed without any \( ld \)- or \( la \)-actions, and therefore, without violating \( \neg \square \diamond ld \) or \( \neg \square \diamond la \). Thus, \( Inftr(P') = Inftr(P) \). As for \( Divtr \), there are no divergences in Figure 8, so any divergences would have to emerge in hiding from infinite traces ending in a sequence of the form \( \{ld, la\}^\omega \). However, the fairness operators remove all such infinite traces, so \( Divtr(P') = \emptyset \). We conclude that the behaviour of \( P' \) is as shown in Figure 9, and this is clearly acceptable.

However, we can obtain an even better result. We made the assumption that the channels can lose only a finite number of messages, during their entire operation. We will next try the weaker assumption that the channels can lose only a finite number of messages \textit{between} passing messages through. In other words, the channels cannot from some point on lose all messages. This can be expressed with the constraints \( \psi_d \equiv \square \diamond ld \Rightarrow \square \diamond rd0 \lor \square \diamond rd1 \) and \( \psi_a \equiv \square \diamond la \Rightarrow \square \diamond ra0 \lor \square \diamond ra1 \). Thus, let

\[ P'' = \text{hide } I \; \text{in} \; (S \parallel \Psi_d^\parallel (DC) \parallel \Psi_a^\parallel (AC) \parallel R) \]

By the same arguments as above, \( S_{fail}(P'') = S_{fail}(P) \) and \( Inftr(P'') = Inftr(P) \).

Because even an infinite number of \( ld \)- and \( la \)-actions is now possible, for \( Divtr \) we need a larger diagram where also \( rd \)- and \( ra \)-actions are visible. We do not show the diagram (with 34 states) here, but it is straightforward to check from it that all infinite traces that could turn into divergences violate one or both of the fairness formulas, so \( Divtr(P'') = \emptyset \). Thus, even with the weaker assumption, the behaviour of the protocol is the LTS in Figure 9.

5.2 A Verification Approach

After the verification example we will in the remaining subsections present a general approach for using fairness operators in verification. As a starting point we assume that we have a system which is composed through parallel composition and hiding from basic processes, that is, a system generated by the grammar

\[ P ::= P \parallel P \]
\[ \mid \text{hide } A \; \text{in} \; P \]
\[ \mid L \]

Here, \( A \) represents any set of visible actions and \( L \) any finite LTS.

Let us now augment the system \( P \) with fairness operators \( \Psi_1^\parallel, \ldots, \Psi_k^\parallel \), where each \( \Psi_i^\parallel, i = 1, \ldots, k \), is applied to a subsystem that is compatible with the corresponding
fairness formula \( \psi_i(A_i^1, \ldots, A_i^{m_i}; B_i^1, \ldots, B_i^{n_i}) \) (in other words, a subsystem which is in \( \text{COMP}_{\psi_i} \)). The new system is denoted by \( P' \).

For technical convenience we will from now on assume that any actions from the fairness formulas that are hidden in an expression of the form \( \text{hide } X \in R \) do not occur anywhere else in \( P' \) except in the subsystem \( R \) under that hiding operator. We do not lose any generality in this assumption; if it does not hold we can simply rename the hidden actions in \( R \) with new, unique names without affecting the end result.

In general, \( P' \) is infinite. Therefore, our aim is to construct a finite representation of \( P' \), that is, a finite LTS \( P^* \) such that

\[
P^* \simeq_{\text{CFFD}} P'.
\]

Since we now have to deal with several fairness constraints simultaneously, we will use the following notation for talking about the sets of the various formulas.

**Definition 29** Let \( \psi_i(A_i^1, \ldots, A_i^{m_i}; B_i^1, \ldots, B_i^{n_i}) \), for \( i = 1, \ldots, k \), be fairness formulas. We write

- \( A_i = A_i^1 \cup \cdots \cup A_i^{m_i} \)
- \( B_i = B_i^1 \cup \cdots \cup B_i^{n_i} \)
- \( F_i = A_i \cup B_i \) (the actions in fairness formula \( \psi_i \)),
- \( F'_i = \{ a \in F_i \mid a \in X \text{ for some subexpression } \text{hide } X \text{ in } R \} \) (the actions in \( \psi_i \) that are used in some hiding operator)

for \( i = 1, \ldots, k \). We also write

- \( F = F_1 \cup \cdots \cup F_k \) (all the actions in the fairness formulas)
- \( F' = F'_1 \cup \cdots \cup F'_k \) (all the actions in the fairness formulas that are used in some hiding operator)

In our construction we will use graph-theoretic results. Therefore, we will consider LTSs as labelled graphs.

**Definition 30** A labelled graph is a pair \( G = (S, \Delta) \), where \( \Delta \) consists of triples of the form \((s, b, s')\) where \( s, s' \in S \) and \( b \) is any symbol, or label. We call \( S \) the set of states and \( \Delta \) the set of transitions. We call any labelled graph \( C = (S_C, \Delta_C) \) a subgraph, or a component, of \( G \) if and only if \( S_C \subseteq S \) and \( \Delta_C \subseteq \Delta \). A component is nontrivial if and only if \( \Delta_C \neq \emptyset \).

It is important to notice that here a subgraph is defined not only by its set of states but also by its set of transitions. Therefore, two subgraphs can have the same set of states but a different set of transitions. It is obvious that we can consider any LTS as a labelled graph simply by ignoring the alphabet and the initial state.

Our algorithm for constructing the finite representation \( P^* \) works as described below.

We assume that we have a function \( \text{FindMSSCs}(C) \), which identifies the maximal strongly connected components from the given (not necessarily connected) graph \( C \), e.g., by using Tarjan’s algorithm [1]. We also assume that we have a function \( \text{RemoveActions}(C, R) \), which removes from \( C \) all transitions labelled with actions in the set \( R \). Both of these operations can be done in time linear in the number of states and transitions in \( C \).
1. We construct a system $P^\dagger$ that is the same as $P$ except that the actions $\mathcal{F}$ from the fairness formulas are left visible. Intuitively, the significance of $P^\dagger$ is that this is the system that remains if we move the fairness operators and the hiding of the related actions out of $P'$; the context independence property allows us to do this. The idea is that we can check the effect of these operators on the finite parameter system $P^\dagger$. The construction of $P^\dagger$ will be described in more detail in Section 5.3.

2. We temporarily consider only the actions $\mathcal{F}' \cup \{\tau\}$ in $P^\dagger$, that is, the actions of the fairness formulas that will be hidden, and the $\tau$-action. Then, we identify the maximal nontrivial strongly connected components from the remaining subgraph. We will call these components $\mathcal{F}'_\tau$-components. Intuitively, the reason we are interested in them is that these are the components that become divergences after the actions $\mathcal{F}'$ have been hidden. Notice that $\Sigma_P = \Sigma_{P^\dagger} \setminus \mathcal{F}'$. We let

$$[C_1, \ldots, C_t] := \text{FindMSSCs}(\text{RemoveActions}(P^\dagger, \Sigma_P))$$

3. We let $C$ be each nontrivial member of $C_1, \ldots, C_t$, in turn, and repeat the following steps

(a) We remove all transitions and states of $C$ except a single state, denoted by $s_C$. This can be any state of $C$, but if $C$ contains the initial state, this will be selected as $s_C$. We redirect the external transitions of $C$ (transitions which are not in $C$ but start or end in a state of $C$) into $s_C$. Notice that if there is a transition between states of $C$ that is not part of $C$, then the transition becomes a loop from $s_C$ to itself. We will show the correctness of this construction in Section 5.5.

$$\text{if } s_{P^\dagger} \in S_C \text{ then } s_C := s_{P^\dagger} \text{ else choose any } s_C \in S_C$$

$$S_{P^\dagger} := S_{P^\dagger} \setminus (S_C \setminus \{s_C\})$$

$$\Delta_{P^\dagger} := \{(s_1, b, s_2) \mid \exists (s'_1, b, s'_2) \in \Delta_{P^\dagger} \setminus \Delta_C :$$

$$s'_1 \notin S_C \land s_1 = s'_1 \lor s'_1 \in S_C \land s_1 = s_C) \land$$

$$s'_2 \notin S_C \land s_2 = s'_2 \lor s'_2 \in S_C \land s_2 = s_C) \}$$

(b) If $C$ contains an infinite execution (starting from any state) that is allowed by the fairness formulas, we add a $\tau$-loop from $s_C$ to itself. In Section 5.4 we will show how this can be checked.

$$\text{if } C \text{ contains an infinite execution } \eta \text{ such that } \eta \models \psi_1, \ldots, \psi_k \text{ then }$$

$$\Delta_{P^\dagger} := \Delta_{P^\dagger} \cup \{(s_C, \tau, s_C)\}$$

4. Let us denote by $P^{\ddagger}$ the parameter system $P^\dagger$ after the above operations. We hide the actions $\mathcal{F}'$ to obtain the finite model $P^\ast$. If we wish, we can also reduce the result according to CFFD.

$$P^\ast = \text{hide } \mathcal{F}' \text{ in } P^{\ddagger}$$

5. We check whether $P^\ast$ is an exact representation of $P'$. In Section 5.5 we will show that $P^\ast \simeq_{\text{CFFD}} P'$ precisely when $\text{Inftr}(P^\dagger) = \text{Inftr}(P)$, and otherwise a finite representation of $P'$ does not exist. We give an algorithm for checking this in Section 5.6. Nevertheless, even in the negative case $P^\ast$ is a conservative estimate of $P'$, that is, $P' \leq_{\text{CFFD}} P^\ast$. 

146
Returning to our verification example, \( P^\ddagger \) in Figure 8 has four \( \mathcal{F}'_{\tau} \)-components. The topmost state at right and the attached \( ld \) - and \( la \)-loops form one \( \mathcal{F}'_{\tau} \)-component. The other three \( \mathcal{F}'_{\tau} \)-components each consist of a single state with an \( ld \)-loop. The constraints \( \neg \Box \Diamond ld \) and \( \neg \Box \Diamond la \) disabled infinite executions in all the \( \mathcal{F}'_{\tau} \)-components, and therefore, all the components could be replaced by a state without a \( \tau \)-loop. Then the actions \( \{ld, la\} \) were hidden, and after CFFD-preserving reduction the result was the graph in Figure 9. However, if we used only the constraint \( \neg \Box \Diamond ld \), then the \( \mathcal{F}'_{\tau} \)-component mentioned first would contain an allowed infinite execution, and therefore there would be one \( \tau \)-loop in the resulting diagram.

In the remaining subsections we will describe the steps above in more detail and prove the correctness of the construction. We will also consider the complexity of the algorithms. It is important to notice that system \( P^\ddagger \) which is used as a parameter in the construction, is not the complete state-space of the original system, but an intermediate system where the actions of the fairness formulas have been left visible. \( P^\ddagger \) can be constructed by using any CFFD-preserving reduced LTS construction method. In our verification example, the complete state-space has 146 states, while the \( P^\ddagger \) shown in Figure 8 has 7 states.

It should also be noted that even though we use model checking techniques in the sequel, the answer to these checks is not the final aim of our approach. The checks are used to obtain a finite LTS \( P^* \) that represents the system \( P' \). This LTS not only allows us to verify all properties preserved by the equivalence, but it can also be used as a component in further compositional analysis.

This approach can obviously be supported by verification tools, such as the TVT tool framework [14]. First of all, we need a tool for adding a fairness constraint to an LTS. The tool checks whether adding the constraint is legal, and stores the fairness constraint in the file representing the LTS. Then, if hiding is applied to an LTS file containing fairness constraints, the actions that are used in the constraints are not hidden but are marked as temporarily visible in the file. If the LTS is used in parallel composition, the constraints and temporarily visible actions in the different files are combined into the result file. If the temporarily visible actions are used in other processes of the parallel composition, the tool should first rename these actions suitably to avoid conflict. When a new fairness constraint is applied to the file, then because of Propositions 23 and 28, it suffices to check the underlying LTS and the alphabet for compositionality. Finally, there would be a tool for instantiating the “open” fairness constraints in an LTS file. This tool constructs, if possible, a finite LTS representing the fair system by using the algorithm presented above. The result is an LTS file without any fairness constraints.

### 5.3 Applying Context-Independence

As described above, we move the fairness operators to a higher level of the system structure by using the context-independence property, thus producing the finite system \( P^\ddagger \). This is expressed formally in the following proposition. Although the result is stated in terms of CFFD, it holds up to isomorphism.

**Proposition 31** Let \( P^\ddagger \) be the same as \( P \) except that actions \( \mathcal{F}'_1 \cup \cdots \cup \mathcal{F}'_k \) are not hidden. Then

\[
P' \simeq_{\text{CFFD}} \text{hide } \mathcal{F}'_1 \cup \cdots \cup \mathcal{F}'_k \text{ in } \Psi^\ddagger_1(\cdots \Psi^\ddagger_k(P^\ddagger) \cdots).
\]
Proof  The proof works by transforming $P'$. First, the hiding of any actions that are used in the fairness formulas is postponed until the topmost system level. Consider the fairness operator $\Psi_i^\|$, which involves the set of actions $F_i$. We will use an auxiliary variable $F_i'$ and initially we assign $F_i' := \emptyset$. We can then obviously replace $\Psi_i^\| (S)$ by $\text{hide } F_i' \in \Psi_i^\| (S)$. We next move the operator “\text{hide } $F_i'$” upward in the system structure by repeating the following steps.

If the operator is inside another hiding operator, say $\text{hide } X$ in $(\text{hide } F_i' \in Q)$, we subtract from the hiding set $X$ those actions that are in $F_i'$, and include them in the hiding set $F_i'$. Therefore, we assign $F_i' := F_i' \cup (X \cap F_i)$ and switch the order of the two hiding operators, so that the expression becomes $\text{hide } F_i' \in (\text{hide } X \setminus F_i \in Q)$.

If the hiding operator is part of a parallel composition, then because of Proposition 7 and the assumption about hidden actions we made in Section 5.2, we can move the operator outside the parallel composition, because the actions $F_i'$ cannot occur in the other processes. Finally, by the context-independence property (Proposition 27) we can move the hiding operator outside any fairness operator, because by the same assumption the actions $F_i'$ cannot occur in the fairness operator.

When “\text{hide } $F_i'$” has been moved to the outermost level in the system, $F_i'$ will consist of precisely those actions from $F_i$ that are in the hiding set of some hiding operator. Notice that by the assumption about the hidden actions, all such hiding operators are above $F_i$, and just below the hiding of the $F_i'$-actions. This is a straightforward application of the context-independence property. A fairness operator can be moved outside any parallel composition, and it can also be moved outside any hiding operator in $P^+$, because after the above procedure the actions $F_i'$ do not occur in these operators. Finally, because of the commutativity of fairness operators, we can move $\Psi_i^\|$ outside any other fairness operator (or, if we wish, we can simply move the outermost fairness operators first).

The result of this manipulation is that the system $P'$ is equivalent to

$$\text{hide } F_i' \cup \cdots \cup F_k' \in \Psi_i^\| (\cdots \Psi_k^\| (P^+) \cdots),$$

where $F_i'$ is, as described above, those actions from $F_i$ that are in the hiding set of some hiding operator, and $P^+$ is the same as $P^+$, except that the fairness operators...
have been removed. The latter means that \( P^\dagger \) is also the same as the original system \( P \) except that the actions \( \mathcal{F}'_1 \cup \cdots \cup \mathcal{F}'_k \) are not hidden.

5.4 Checking Allowed Infinite Executions

As described in Section 5.2, the first thing that we need to do with the system \( P^\dagger \) is to identify the \( \mathcal{F}' \)-components, and from each of these check whether they contain an infinite execution that is allowed by the formulas \( \psi_1, \ldots, \psi_k \). In this subsection we describe one way to check this.

Infinite sequences of invisible actions are fair and they can easily be detected with a depth-first search. Detecting fair infinite sequences with infinitely many \( \mathcal{F}' \)-actions is more difficult. We solve the problem by using reduction to Büchi automata [34]. Recall that a Büchi automaton is structurally identical to a finite automaton, but takes an infinite sequence of actions as input. Notice that here we allow Büchi automata to have invisible \( \tau \)-actions.

**Definition 32** A Büchi automaton is a tuple \( (S, \Sigma, \Delta, \hat{s}, F) \), where \( S, \Sigma, \Delta \) and \( \hat{s} \) are as in the definition of an LTS, Definition 1, and \( F \subseteq S \) is the acceptance set. Furthermore, all these sets are assumed to be finite.

A Büchi automaton accepts those infinite sequences of visible actions that can be executed by visiting acceptance states infinitely many times, and the accepted sequences constitute the language of the automaton.

**Definition 33** A Büchi automaton \( A = (S, \Sigma, \Delta, \hat{s}, F) \) accepts an infinite sequence of visible actions \( \xi \in \Sigma^\omega \) if and only if there are \( s_1, s_2, s_3, \ldots \in S \) and \( a_1, a_2, a_3, \ldots \in \Sigma \cup \{\tau\} \) such that \( \hat{s} \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \cdots \), and \( \text{restr}(a_1 a_2 a_3 \cdots, \Sigma) = \xi \), and \( s_i \in F \) for infinitely many \( i > 0 \). The language of the automaton \( L(A) = \{ \xi \in \Sigma^\omega \mid A \text{ accepts } \xi \} \).

From now on we will use \( \Sigma_{P^\dagger} \) as the alphabet \( \Sigma \). We will construct a Büchi automaton that accepts precisely the infinite sequences of visible actions from \( \Sigma \) that fulfill the fairness formulas. The problem of converting state-based propositional LTL to Büchi automata has been extensively studied [13, 40]. On the other hand, our setting is action-based, and we are dealing with a specific subset of LTL. It turns out that it is quite easy to directly construct automata for formulas of this sublogic, and we can obtain a better complexity than in the general case. We start by considering a single fairness formula, and then we combine automata for different formulas by using parallel composition. The length of a formula, \( |\psi| \), is the number of lexical elements ("\( \Box \)", "\( \Diamond \)", "\( \lor \)", "\( \land \)", "\( \Rightarrow \)", "\( \neg \)" and action names) in the formula.

**Proposition 34** Let \( \psi(A_1, \ldots, A_m; \mathcal{B}_1, \ldots, \mathcal{B}_n) \) be a fairness formula with actions from the alphabet \( \Sigma \). There is a Büchi automaton \( A \) with at most \( |\psi| \) transitions for each action \( a \in \Sigma \), and with at most \( |\psi| \) states, such that

\[
L(A) = \{ \xi \in \Sigma^\omega \mid \xi \models \psi \}.
\]
Figure 10. The Büchi automaton $A_i$ corresponding to formula $\neg A_i$.

Proof. We will first transform the formula into a disjunctive normal form. Namely,

$$A_1 \land \cdots \land A_m \Rightarrow B_1 \lor \cdots \lor B_n \equiv \neg(A_1 \land \cdots \land A_m) \lor B_1 \lor \cdots \lor B_n \equiv \neg A_1 \lor \cdots \lor \neg A_m \lor B_1 \lor \cdots \lor B_n$$

and

$$\neg A_i \equiv \neg(\Diamond \Box a_i^1 \lor \cdots \lor \Diamond \Box a_i^{u_i})$$
$$\equiv \neg \Box a_i^1 \lor \cdots \lor \neg \Box a_i^{u_i}$$

for each $i = 1, \ldots, m$. Formula $\neg A_i$ states that none of the actions in $A_i$ occurs infinitely many times. It is easy to see that the language defined by $\neg A_i$ is accepted by the automaton $A_i$ illustrated in Figure 10. Formally, $A_i = (S, \Sigma, \Delta, \hat{s}, F)$ with $S = \{s_1, s_2\}$, $\Delta = (\{s_1\} \times \Sigma \times \{s_1\}) \cup (\{s_1\} \times \Sigma \times \{s_2\}) \cup (\{s_2\} \times (\Sigma \setminus A_i) \times \{s_2\})$, $\hat{s} = s_1$ and $F = \{s_2\}$. Similarly, $B_j = \Diamond \Box b_j^1 \land \cdots \land \Diamond \Box b_j^{v_j}$ means that all of the actions $\{b_j^1, \ldots, b_j^{v_j}\}$ occur infinitely many times, and therefore the language defined by $B_j$ is accepted by the automaton $B_j$ in Figure 11. Formally, $B_j = (S, \Sigma, \Delta, \hat{s}, F)$ where $S = \{s_0, s_1, \ldots, s_{v_j}\}$, $\Delta = \{(s_{k-1}, b_k^j, s_k) \mid 1 \leq k \leq v_j\} \cup \{(s_k, a, s_k) \mid 0 \leq k < v_j \land b_{k+1}^j \neq a \in \Sigma\} \cup \{(s_{v_j},) \times \Sigma \times \{s_0\}\}$, and $\hat{s} = s_0$, and $F = \{s_{v_j}\}$. Then, we obtain an automaton $A$ accepting the language defined by $\neg A_1 \lor \cdots \lor \neg A_m \lor B_1 \lor \cdots \lor B_n$ by adding a new initial state and transitions for each $a \in \Sigma$ from this state to each start state of $A_1, \ldots, A_m, B_1, \ldots, B_n$. This automaton has $m \cdot 2 + \sum_{j=1}^{n} (v_j + 1) + 1 = 2m + n + \sum_{j=1}^{n} v_j + 1 \leq |\psi|$ states. Furthermore, for each $a \in \Sigma$ there are at most

$$m \cdot 3 + \sum_{j=1}^{n} (v_j + 1) + (m + n) = 4m + 2n + \sum_{j=1}^{n} v_j \leq |\psi|$$

transitions. □

Figure 11. The Büchi automaton $B_j$ corresponding to formula $B_j$. 150
We next define parallel composition of Büchi automata. This works in the same way as parallel composition of LTSs, but we also have to take into consideration the acceptance condition. The states of the parallel composition have an additional index that points to one of the parameter automata. When the automaton that is pointed to by the index is in an acceptance state, the index moves to point to the next automaton. The acceptance states of the parallel composition are chosen to be the states where the index points to the first automaton and this is in an acceptance state.

**Definition 35** Let \( A_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1, F_1) \), \( \ldots \), \( A_n = (S_n, \Sigma_n, \Delta_n, \hat{s}_n, F_n) \) be Büchi automata. Their product is the Büchi automaton \( (S', \Sigma, \Delta', \hat{s}, F) \) such that the following holds: \( S' = S_1 \times \cdots \times S_n \times \{1, \ldots, n\} \), and \( ((s_1, \ldots, s_n, i), a, (s'_1, \ldots, s'_n, i')) \in \Delta' \) if and only if the same condition holds as for parallel composition of LTSs in Definition 4 and, in addition, either \( i' = i \) or \( s_i \notin F_i \), or \( i' = i \mod n + 1 \) and \( s_i \in F_i \). Finally, \( F = F_1 \times F_2 \times \cdots \times F_n \times \{1\} \), and \( F \) and \( \hat{s} \) are as in Definition 4. As in Definition 4, the parallel composition \( A_1||\cdots||A_n \) is the reachable part of the product.

**Proposition 36** Let \( A_1, \ldots, A_n \) be automata with the same alphabet. Then, \( \mathcal{L}(A_1||\cdots||A_n) = \mathcal{L}(A_1) \cap \cdots \cap \mathcal{L}(A_n) \).

**Proof** Assume \( \xi \in \mathcal{L}(A_1||\cdots||A_n) \). The execution of \( \xi \) visits acceptance states of \( A_1||\cdots||A_n \) infinitely many times, and these are the states where \( A_1 \) is in an acceptance state and the index is 1. However, every time the execution reaches such a state, the index starts to point to the next automaton. Since the index returns to \( A_1 \) infinitely many times, also \( A_2, \ldots, A_n \) visit acceptance states infinitely many times. Since \( A_1, \ldots, A_n \) all have the same alphabet, they all execute the same sequence of visible actions, and therefore \( \xi \in \mathcal{L}(A_1) \cap \cdots \cap \mathcal{L}(A_n) \). For the other direction, if \( \xi \in \mathcal{L}(A_1) \cap \cdots \cap \mathcal{L}(A_n) \) then \( A_1, \ldots, A_n \) can each execute \( \xi \) while visiting acceptance states infinitely many times. Clearly, these executions can be made in the parallel composition by executing the visible actions in synchrony and interleaving any \( \tau \)-actions. In the composite execution the index rotates infinitely through the processes because we always eventually meet an acceptance state in the indexed process. Therefore, the composite execution, which has the same sequence of visible actions \( \xi \) as the constituent executions, visits acceptance states of \( A_1||\cdots||A_n \) infinitely many times, and thus \( \xi \in \mathcal{L}(A_1||\cdots||A_n) \).

**Proposition 37** Let \( \psi_1, \ldots, \psi_k \) be fairness formulas with actions from the alphabet \( \Sigma \). There is a Büchi automaton \( A \) with \( O(k \prod_{i=1}^{k} |\psi_i|) \) transitions for each \( a \in \Sigma \), and with \( O(k \prod_{i=1}^{k} |\psi_i|) \) states, such that

\[
\mathcal{L}(A) = \{ \xi \in \Sigma^\omega \mid \xi \models \psi_1, \ldots, \psi_k \}.
\]

**Proof** By Proposition 34 there are automata \( A_1, \ldots, A_k \) that accept, respectively, the languages defined by the formulas \( \psi_1, \ldots, \psi_k \). Then, by the previous proposition

\[
\mathcal{L}(A_1||\cdots||A_k) = \mathcal{L}(A_1) \cap \cdots \cap \mathcal{L}(A_k)
\]
\[
= \bigcap_{i=1}^{k}\{ \xi \in \Sigma^\omega \mid \xi \models \psi_i \}
= \{ \xi \in \Sigma^\omega \mid \xi \models \psi_1, \ldots, \psi_k \}.
\]

The automaton \(A_1 \mid \cdots \mid A_k\) can have at most \(k \cdot \prod_{i=1}^{k} |S_i|\) states, which is in \(O(k \prod_{i=1}^{k} |\psi_i|)\) by Proposition 34. The number of transitions with action \(a \in \Sigma\) is at most \(k\) times the product of the number of \(a\)-transitions in \(A_1, \ldots, A_k\), and is therefore in \(O(k \prod_{i=1}^{k} |\psi_i|)\).

Our aim was to check from each \(\mathcal{F}^*\)-component \(C\) for an infinite execution (starting at any state) which fulfills every fairness formula \(\psi_1, \ldots, \psi_k\).

**Proposition 38** Whether an \(\mathcal{F}^*\)-component \(C\) contains an infinite execution that fulfills \(\psi_1, \ldots, \psi_k\) can be decided in time

\[
O(|\Delta_C| k \prod_{i=1}^{k} |\psi_i|)
\]

**Proof** \(C\) contains an allowed infinite execution if and only if it either has an infinite \(\tau\)-execution, or can execute an infinite sequence of visible actions \(\xi\) such that \(\xi \models \psi_1, \ldots, \psi_k\). The former can be checked in time linear in \(|\Delta_C|\) by using depth-first search to look for \(\tau\)-cycles in \(C\). If there are none, then we can check the latter by considering \(C\) as a Büchi automaton with alphabet \(\Sigma_{P^*}\), and with every state an acceptance state, that is, \(F = S_C\). We can choose an arbitrary state of \(C\) as the initial state, because \(C\) is strongly connected and we are only concerned with the infinitary behaviour. We use the previous proposition to construct a Büchi automaton \(A\) such that \(L(A) = \{ \xi \in \Sigma^\omega \mid \xi \models \psi_1, \ldots, \psi_k \}\). Then, \(L(C\mid A) = L(C) \cap L(A) = \{ \xi \in \text{Inftr}(C) \mid \xi \models \psi_1, \ldots, \psi_k \}\). We thus need to check whether the language of the parallel composition \(C\mid A\) is non-empty. This can be done by depth-first search in time linear in the number of transitions \(|\Delta_C\mid A\) by looking for an acceptance state that is reachable from itself. (Notice that in this case there are no \(\tau\)-cycles.) Each visible transition in \(C\) with action \(a \in \Sigma\) is synchronised with at most each \(a\)-transition in \(A\), of which there are at most \(O(k \prod_{i=1}^{k} |\psi_i|)\). Furthermore, each \(\tau\)-transition of \(C\) appears in \(C\mid A\) at most twice the number of times there are states in \(A\), which is also in \(O(k \prod_{i=1}^{k} |\psi_i|)\).

Thus, altogether the number of transitions \(|\Delta_C\mid A\) is at most \(O(|\Delta_C| k \prod_{i=1}^{k} |\psi_i|)\). □

### 5.5 The Finite Representation

We next show that if \(\text{Inftr}(P') = \text{Inftr}(P)\), then the system \(P^*\) we have constructed is the desired finite representation of \(P'\), and otherwise there does not exist an exact finite representation.

In our proof we will need the following result. When \(s\) is a state of \(C\), let us denote by \(\text{Out}_C(s)\) the set of labels of (visible or invisible) transitions of \(C\) that start from \(s\), that is, \(\text{Out}_C(s) = \{ a \mid \exists s' \in S_C : (s, a, s') \in \Delta_C \}\).
Proposition 39 Let $C$ be an $\mathcal{F}'_\tau$-component of $P^1$. If $C$ does not contain an infinite execution that fulfills $\psi_1, \ldots, \psi_k$, then $C$ contains at least one state $q$ such that $Out_C(q) \subseteq A_1 \cup \cdots \cup A_k$.

Proof Assume that there is no such state. Then, since $C$ is strongly connected and nontrivial, every state of $C$ has at least one outgoing transition with an action outside of $A_1 \cup \cdots \cup A_k$. Therefore, starting from any state of $C$, we can execute these actions infinitely. Clearly, an infinite execution that does not contain actions from $A_1 \cup \cdots \cup A_k$ cannot contain any of $\psi_1, \ldots, \psi_k$. 

Theorem 40 If $Inftr(P') = Inftr(P)$ then $P^* \simeq_{CFFD} P'$. If $Inftr(P') \neq Inftr(P)$, then there does not exist a finite process $Q$ with $Q \simeq_{CFFD} P'$.

Proof We first observe that the following invariant holds for step 3 (a) (and, trivially, for 3 (b)) in the construction of Section 5.2: if $s$ is a state that is retained in the step, then the same set of traces of $\Sigma_P$-actions lead from the initial state to $s$ before and after the step. From this invariant it follows that the set of $\Sigma_P$-traces leading to any state of $P^1$ (or $P^*$) is the same as it was in the original $P^1$. The invariant property follows from two facts. First, $C$ itself is strongly connected and contains no actions from $\Sigma_P$. Secondly, every transition between a state of $C$ and an external state $q$ is replaced by a transition in the same direction between $s_C$ and $q$, and an external transition between states of $C$ is replaced by a transition from $s_C$ to itself.

For the stable failures part of CFFD-equivalence, we need to show that $Sfail(P^*) = Sfail(P')$. Because the fairness operators preserve stable failures (by Definition 14) and because the stable failures resulting from parallel composition and hiding depend only on the stable failures of the parameter processes [39], it holds that $Sfail(P') = Sfail(P)$. Thus, it suffices to show $Sfail(P^*) = Sfail(P)$.

Clearly, all the states of an $\mathcal{F}'_\tau$-component become unstable in $\text{hide } \mathcal{F}'$ in $P^1$ (which is CFFD-equivalent to $P$). Therefore, only states that would have become unstable are removed or modified during the construction which, together with the above observation about the preservation of traces, means that no stable failures are removed.

We next show that no new stable failures are introduced. If $s_C$, which remains from the component $C$, is unstable after hiding $\mathcal{F}'$ (because of a $\tau$-loop or an outgoing invisible transition), we are done. Otherwise, let $(\sigma, Y) \in Sfail(P^*)$ be any stable failure caused by $s_C$. It holds that $Y \subseteq \Sigma_P \setminus X$, where $X$ is the set of outgoing actions from $s_C$ in $P^*$. Furthermore, since in $P^*$ state $s_C$ is reachable with the trace $\sigma$, then by the above observation about the preservation of traces, it was reachable in $P^1$ with a trace $\rho$ for which $\text{restr}(\rho, \Sigma_P) = \sigma$.

In $C$ there is no infinite execution in compliance with $\psi_1, \ldots, \psi_k$ (otherwise a $\tau$-loop would have been added to $s_C$), so the previous proposition guarantees that there is a state $q$ in $C$ such that $Out_C(q) \subseteq A_1 \cup \cdots \cup A_k$. Clearly, state $q$ is reachable in $P^1$ with a trace $\rho'$ that is a continuation of $\rho$ and for which it also holds that $\text{restr}(\rho', \Sigma_P) = \sigma$. The outgoing actions of $q$ in $P^1$ consist of the ones that are part of $C$, namely $Out_C(q)$, and some subset of the actions $X$ (all of which are
visible) which are not part of $C$. Therefore, $q$ is stable and refuses at least the set of actions $\Sigma_{P_1} \setminus (\text{Out}_C(q) \cup X)$. Thus,

$$ (p', \Sigma_{P_1} \setminus (\text{Out}_C(q) \cup X)) \in S\text{fail}(P') \setminus \Psi. $$

Since each fairness operator $\Psi_i$ was applied to a subsystem that is in $\text{COMP}_{\Psi_i}$, from Proposition 23 and the fact that the actions of the fairness formulas are not used in hiding in $P'$, it follows that $P'$ is also in $\text{COMP}_{\Psi_i}$ for each $i = 1, \ldots, k$. From applying the definition of compatibility $k$ times it then follows that

$$ (p', \Sigma_{P_1} \setminus (\text{Out}_C(q) \cup X)) \cup A_1 \cup \cdots \cup A_k \in S\text{fail}(P') \setminus \Psi. $$

From $\text{Out}_C(q) \subseteq A_1 \cup \cdots \cup A_k$ and $\Sigma_{P_1} = \Sigma_P \cup \psi$, and the fact that $X$ is disjoint from $\tau$ (which is hidden), it follows that

$$ (\Sigma_{P_1} \setminus (\text{Out}_C(q) \cup X)) \cup A_1 \cup \cdots \cup A_k = \begin{array}{ll}
(\Sigma_{P_1} \setminus X) \cup A_1 \cup \cdots \cup A_k & \\
((\Sigma_P \cup \psi) \setminus X) \cup A_1 \cup \cdots \cup A_k & \\
(\Sigma_P \setminus X) \cup \psi \cup A_1 \cup \cdots \cup A_k & \\
\end{array} $$

Therefore,

$$ (p', (\Sigma_P \setminus X) \cup \psi \cup A_1 \cup \cdots \cup A_k) \in S\text{fail}(P'), $$

which trivially implies

$$ (p', (\Sigma_P \setminus X) \cup \psi) \in S\text{fail}(P'). $$

By the properties of hiding this means that

$$ (\sigma, \Sigma_P \setminus X) \in S\text{fail}((\text{hide } \psi \text{ in } P') \setminus \Psi) = S\text{fail}(P'), $$

which further implies $(\sigma, \psi) \in S\text{fail}(P')$. We conclude that no new stable failures are introduced, and $S\text{fail}(P') = S\text{fail}(P) = S\text{fail}(P')$.

We next show that $\text{Divtr}(P') = \text{Divtr}(P)$. Let $\sigma \in \text{Divtr}(P')$. After replacing the $\tau'$-components there are no cycles consisting of $\tau' \cup \{\tau\}$ other than the $s_C$ states with $\tau$-loops. Thus, one such $s_C$ is reachable with a trace whose sequence of actions from $\Sigma_P$ is $\sigma$. By construction, $s_C$ was originally part of the $\tau'$-component $C$ of $P'$ which contains an infinite execution that fulfills $\psi_1, \ldots, \psi_k$. Furthermore, the preservation of $\Sigma_P$-traces means that $s_C$ was reachable with the sequence of $\Sigma_P$-actions $\sigma$. Therefore, in $P'$ it is possible to execute an infinite sequence of actions which is allowed by the fairness operators and whose sequence of actions from $\Sigma_P$ is $\sigma$. It follows that

$$ \sigma \in \text{Divtr}(\text{hide } \tau' \text{ in } \Psi_1(\cdots \Psi_k(P') \cdots)) = \text{Divtr}(P'). $$

For the other direction, let $\sigma \in \text{Divtr}(P')$. Then there is an infinite execution in $\text{hide } \tau' \text{ in } \Psi_1(\cdots \Psi_k(P') \cdots) \approx_{\text{CFD}} P'$ whose sequence of visible actions is $\sigma$. In $P'$ this execution stays, from some point on, within some strongly connected component that consists of the actions $\tau' \cup \{\tau\}$, and is reachable with the sequence of $\Sigma_P$-actions $\sigma$. This component is included in some $\tau'$-component $C$. 

154
Because $C$ contains an infinite execution that is allowed by the fairness operators, and is thus in compliance with $\psi_1, \ldots, \psi_k$, a $\tau$-loop is added to the state $s_C$ which replaces $C$. Again, by the preservation of $\Sigma_P$-traces, $s_C$ is reachable by $\sigma$ in $P^*$, so $\sigma \in Divtr(P^*)$. Thus, $Divtr(P^*) = Divtr(P')$.

It remains to show $Inftr(P^*) = Inftr(P')$. We first show that the construction of $P^*$ preserves the traces and infinite traces of $P$. Assume $\sigma \in Tr(P^*)$. If $s$ is the last state in the execution generating the trace $\sigma$, then the preservation of $\Sigma_P$-traces means that $s$ is reachable by the $\Sigma_P$-trace $\sigma$ also in $P^*$, so $\sigma \in Tr(\text{hide} C \text{in } P^*) = Tr(P)$. Now assume $\sigma \in Tr(P)$, and let $s$ again be the last state in the execution generating $\sigma$. If $s$ is retained in the construction, then the preservation of $\Sigma_P$-traces means that $s$ can be reached by $\sigma$ also in $P^*$. If $s$ is removed as part of some $F_i$-component $C$, then the state $s_C$ that was retained from $C$ can be reached by the same sequence of $\Sigma_P$-actions. Thus, in both cases $\sigma \in Tr(P^*)$, so $Tr(P^*) = Tr(P)$. Next, if $L$ is an LTS and $Tr(L)$ is its set of traces, let $\overline{Tr(L)}$ denote the infinitary closure of the traces, i.e., $\overline{Tr(L)} = \{ \xi \in \Sigma_L^* \mid \forall \sigma < \xi : \sigma \in Tr(L) \}$. As shown in [39], for a finite LTS $L$ it holds that $Inftr(L) = \overline{Tr(L)}$. Therefore, since $P^*$ and $P$ are both finite, $Inftr(P^*) = \overline{Tr(P^*)} = Tr(P) = Inftr(P)$.

The assumption is that $Inftr(P') = Inftr(P)$. Thus, $Inftr(P^*) = Inftr(P')$, which concludes our proof that $P^* \simeq CFFD P'$.

For the second claim, assume there exists a finite LTS $Q$ with $Q \simeq CFFD P'$. By the properties of CFFD-equivalence, $Tr(Q) = Tr(P')$ and $Inftr(Q) = Inftr(P')$. Because the fairness operators preserve traces, and because the traces that result from parallel composition and hiding depend only on the traces of the parameters [39], $Tr(P') = Tr(P)$ and, therefore, $Tr(Q) = Tr(P)$. Since $P$ and $Q$ are both finite, $Inftr(Q) = Tr(Q) = Tr(P) = Inftr(P)$, and thus $Inftr(P') = Inftr(P)$.

Fortunately, even if a finite representation of $P'$ does not exist, $P^*$ is always a conservative estimate of $P'$ in CFFD-preorder; in particular, $P^*$ can differ from $P'$ only by having some infinite traces that $P'$ does not have.

**Proposition 41** $P' \leq_{CFFD} P^*$.

**Proof** The proof of the previous theorem shows that $S\text{fail}(P') = S\text{fail}(P^*)$ and $Divtr(P') = Divtr(P^*)$. This implies that $Tr(P') = Tr(P^*)$. It is easy to see that for any LTS $L$, $Inftr(L) \subseteq Tr(L)$. Therefore, since $P^*$ is finite, $Inftr(P') \subseteq Tr(P^*) = Inftr(P^*)$.

### 5.6 Checking Infinite Traces

If we wish to have an exact representation of $P'$ then, as shown above, we must check whether $Inftr(P') = Inftr(P)$. We could, of course, make this check before constructing $P^*$. However, here we presented the construction first, because it also served as a proof of the existence of a finite representation. We can make the check by using the Büchi automata introduced in Section 5.4.
Theorem 42  Deciding whether $\text{Inftr}(P') = \text{Inftr}(P)$ is PSPACE-complete\(^1\) in

$$|S_{P'|}| \leq k \prod_{1 \leq i \leq k} |\psi_i|$$

Proof  We will first show hardness by reducing the well-known PSPACE-complete problem of nonuniversality of Büchi automata [33] to an instance of this problem where $|S_{P'}|$ is polynomial (in fact, linear) in the size of the automaton and the fairness formula is constant.

Given a Büchi automaton $A$ with $n$ states and with alphabet $\Sigma$, the nonuniversality problem is to determine whether $L(A) \not= \Sigma^\omega$, that is, whether there is some infinite word that the automaton does not accept. We can assume without loss of generality that the transition relation of $A$ is full; if not, add a new non-accepting state, redirect all missing transitions to this state and add a transition for every action in $\Sigma$ from this state to itself. Construct an LTS from $A$ as follows. Select two new actions, $a_{\text{new}}$ and $b_{\text{new}}$. Replace every state $s$ in $A$ by two new states $s_1$ and $s_2$, and redirect all incoming transitions of $s$ from $s_1$ and start all outgoing transitions of $s$ from $s_2$. If $s$ is the initial state, make $s_1$ the initial state. If $s$ is an accepting state, add a transition $s_1 - b_{\text{new}} \rightarrow s_2$, otherwise add a transition $s_1 - a_{\text{new}} \rightarrow s_2$. Finally, add a $\tau$-transition from every state to one more new state, and from this state to itself, in order to make all states unstable. Take as alphabet $\Sigma \cup \{a_{\text{new}}, b_{\text{new}}\}$, and call the resulting LTS $P'$. Clearly, the number of states $|S_{P'}|$ is in $O(n)$. Let $\psi \equiv \Box \Diamond a_{\text{new}} \Rightarrow \Box \Diamond b_{\text{new}}$. Then, $P' \in \text{COMP}_\psi$. Let $P = \text{hide } a_{\text{new}}, b_{\text{new}} \text{ in } P'$, and $P' = \text{hide } a_{\text{new}}, b_{\text{new}} \text{ in } \Psi_\psi(P')$. Notice that the fairness formula is constant for all automata. It is straightforward to show that $\text{Inftr}(P') = L(A)$. Since the transition relation of the original automaton was full, $P$ can execute all infinite sequences of visible actions from the alphabet $\Sigma$, that is, $\text{Inftr}(P) = \Sigma^\omega$. It now holds that $\text{Inftr}(P') = \text{Inftr}(P)$ if and only if $\text{Inftr}(P') = \Sigma^\omega$ if and only if $L(A) = \Sigma^\omega$.

We next show that the problem can be solved in space that is polynomial in

$$|S_{P'}| \leq k \prod_{1 \leq i \leq k} |\psi_i|$$

We need to check whether $\text{Inftr}(P') = \text{Inftr}(P)$. As noted in the proof of Proposition 40, $\text{Tr}(P') = \text{Tr}(P)$. From this and the finiteness of $P$ we can conclude, similarly as in the proof of Proposition 41, that $\text{Inftr}(P') \subseteq \text{Inftr}(P)$. Thus, it suffices to check $\text{Inftr}(P) \subseteq \text{Inftr}(P')$. We reduce this problem to language containment of Büchi automata. $P$ can be trivially interpreted as a Büchi automaton $A_P$ by considering all states as acceptance states, that is, $F_{A_P} = S_P$. Then, obviously, $L(A_P) = \text{Inftr}(P)$. The case of $P'$ requires more care. By Proposition 37 there is a Büchi automaton $A$ with $O(k \prod_{1 \leq i \leq k} |\psi_i|)$ states such that $L(A) = \{ \xi \in \Sigma^\omega | \xi \models \psi_1, \ldots, \psi_k \}$. We again interpret $P'$ as a Büchi automaton $A_{P'}$ where every state is an acceptance state. Then, by Proposition 36,

\(^1\)By this we mean that the problem can be solved by using space $(|S_{P'}| k \prod_{1 \leq i \leq k} |\psi_i|)^c$ for some constant $c$, and that any problem in PSPACE is reducible in logarithmic space to an instance of this problem such that the term $|S_{P'}| k \prod_{1 \leq i \leq k} |\psi_i|$ is polynomial in the size of the original problem.
\[ L(A_P || A) = L(A_P) \cap L(A) = \{ \xi \in \text{Inftr}(P^\dagger) \mid \xi \models \psi_1, \ldots, \psi_k \}. \]

It follows that \( L(\text{hide } F' \in \text{in } (A_P || A)) = \text{Inftr}(P') \), where “hide” is defined similarly as for LTSs in Definition 6. Thus, the desired automaton is \( A_{P'} = \text{hide } F' \in (A_P || A) \), which has \( O(|S_{P'}| k \prod_{i=1}^{k} |\psi_i|) \) states.

Now the problem is to check whether \( L(A_P) \subseteq L(A_{P'}) \). This is true precisely when \( L(A_P) \cap L(A_{P'}) = \emptyset \), where \( L(A_{P'}) \) is the complement language \( \Sigma_P^* \setminus L(A_{P'}) \). Assume that \( A_P \) is an automaton accepting the complement language, that is, \( L(A_P) = L(A_{P'}) \). Then, we need to check \( L(A_P) \cap L(A_{P'}) = \emptyset \), which is true precisely when \( L(A_P || A_{P'}) = \emptyset \). We have thus reduced the check to the language emptiness of a Büchi automaton. Checking this is easy; see below. The difficult part here is constructing the complement automaton \( A_{P'} \).

In [33] a construction of a complement automaton is presented which is based on a finite decomposition of the (complement) language in terms of regular congruence classes of finite words. For an automaton with \( n \) states this creates a complement automaton with \( 2^{O(n^2)} \) states. It should be noted that the automata in [33] do not have invisible actions. Therefore, in order to make Lemma 2.2 of [33] hold, we must modify slightly the definition of the transition relation of the automata \( A_1 \), in [33]. Namely, we must replace each requirement of the form \( u \in \rho(v, a) \) with the requirement that \( u \) is reachable from \( v \) by an execution with visible content \( a \), and each requirement of the form \( u \in \rho(v, a) \cap F \) with the requirement that \( u \) is reachable from \( v \) by an execution that visits some state in \( F \) and has visible content \( a \).

As an alternative approach for complementing a Büchi automaton, [32] presents a subset construction by which an automaton can first be determinised into a Rabin automaton. This creates a complement automaton of size \( 2^{O(n \log n)} \).

In terms of memory usage, we can circumvent the exponential blow-up in the complementation, because we only need to check the result for nonemptiness and therefore we do not have to construct the entire automaton \( A_P || A_{P'} \) at once. The language of this automaton is nonempty precisely when there is an acceptance state that is reachable from itself by a path including visible actions. Following the solution of the nonemptiness problem of Büchi automata in [41], we can nondeterministically traverse the state-space of the automaton while looking for such a path, and at any time keep in memory only the acceptance state and the states immediately at hand. As in [33, 41], each state of the complement automaton \( A_{P'} \) can be represented in polynomial space in the size of the original automaton \( A_{P'} \). Furthermore, whether a state is an initial state or an acceptance state, and whether there is a transition with a given label between two states, can each be decided in polynomial space. The same is obviously true of \( A_P \), and therefore also of the parallel composition \( A_P || A_{P'} \). Thus, the nondeterministic search in the state-space of the parallel composition can be done in polynomial space, and therefore the problem is in NPSPACE and, by Savitch’s theorem, in PSPACE.

\( \square \)

It should be noted that there exist simpler tests that can be used as sufficient but not necessary conditions for \( \text{Inftr}(P') = \text{Inftr}(P) \). As an example, this holds if all visible actions are distinct from the \( A \)-actions, and the reachability of the visible transitions from each other does not depend on the \( A \)-actions; this condition was used in the verification example.

157
6 Related Work

Most earlier work on fairness and process algebra has dealt with “global” fairness assumptions, which make a general assumption about every process or action in a system. These are often intended to capture the idea that all processes in a parallel system should get a share of the execution time.

In [3] a number of different notions of global weak and strong fairness assumptions are analysed for different parallel programming languages in terms of feasibility, meaning that it is possible to implement the fairness assumption through scheduling, equivalence robustness, meaning that the fairness assumption does not depend on the ordering of independent actions, and liveness-enhancement, meaning that the fairness assumption should be capable of inducing a change in the system behaviour. The authors conclude that few of the notions fulfill all the requirements.

It should be noted that [3] makes an assumption of uniform choice, meaning that internal actions are never alternatives to external actions, and of noninstantaneous readiness, meaning that immediately after the execution of an external action only internal actions are enabled. For the kind of compositional approach that we use, such assumptions are problematic, because even if all the original processes fulfill the assumptions, their compositions may fail to do so. For example, if we have a process that is itself a composition of parallel processes, then it can easily happen that during the execution of one external action another external action in another part of the composition remains continuously enabled.

In [15] two “finite delay operators” and bisimulations are presented where the system may execute only a finite number of delay actions in the admissible computations. However, the language does not include parallel composition. [16] presents a notion of testing for the full CCS where only those infinite computations are considered where each parallel process makes infinitely many moves. A fully abstract model for the resulting preorder is introduced, which consists of acceptance trees augmented with information on the allowed infinite traces. The drawback of this notion of fairness is that if some subprocess is dead-locked or if its actions are blocked, this leads to an “impossible” situation where there is no way of continuing fairly. Essentially the same problem has been discussed in [12] and [21] in connection with “unconditional fairness” and “impartiality”, respectively. The problem seems especially difficult for a compositional approach, because a system can be placed in any environment that can block actions in an unknown way.

Other notable work on semantics with global fairness assumptions include [25, 24]. The former presents a fully abstract semantics for strong process fairness, and the latter gives, in addition, a fully abstract semantics for strong channel fairness and a sound semantics for weak process fairness. It should be noted that the setting in [25, 24] is somewhat different from the one used in this article, for example. Rather than a “classical” process algebra, it considers imperative programs (with while- and if-statements) that communicate in a CCS-like fashion.

By using a notion of “fair testing”, [6] develops a failure-based congruence for a CSP-like process algebra which ignores those divergences that can be exited with some action. For finite-state systems this corresponds to a notion of fairness such that if a state is encountered infinitely often then all its outgoing transitions are eventually taken.

In [5] a simple version of the alternating bit protocol (where messages cannot disappear but may become corrupted) is verified by using axiomatic process-algebraic proof rules.
Here, divergences are avoided by applying Koomen’s fair abstraction rule. This means that when there is a cycle in the system consisting of actions that are to be hidden, but the cycle can be exited with some action, then no $\tau$-cycle is formed, because a step outside the cycle is assumed to be made “due to some fairness mechanism”.

One suggested approach is to use operational rules which generate just the fair execution sequences. [9] introduces a labelling system and operational rules for CCS which generate exactly those execution sequences in which no (labelled) action stays enabled forever without being executed. Of the notions of fairness in [3] this resembles weak communication fairness, although exact comparison is difficult (for example, [9] does not conform to the above-mentioned assumptions). However, these rules do not allow abstraction from internal events in a system. A similar approach in [10] uses two other notions of fairness (which resemble weak and strong process fairness from [3]) in a way that also allows abstraction. However, the operational rules in [9] are more appealing in the sense that they are based on interleaving sequences of actions, whereas the ones in [10] require explicit bookkeeping on the sets of labels of enabled processes. To our knowledge, no semantic equivalences for dealing with any of the fair operational rules have been presented.

Sometimes a global fairness assumption in the execution model is not enough to show a desired property. On the other hand, we usually do not like making many a priori assumptions about a system we are verifying. A concurrent system may consist of heterogeneous subsystems on different computers, and it is not always realistic to assume that they all conform to the same global fairness assumption. Also, it is possible that some divergences in our system will be exited with probability one because the choices are governed by some probabilistic process, while some other divergences may represent real livelocks. For these reasons, we often want to use specific fairness assumptions, like the ones used in the verification example of Section 5.

To this end it has been suggested that a system could be augmented with additional liveness/fairness information that constrains the allowed infinite executions. [26] presents a method where a CCS-process can be augmented with the $\omega$-regular set of infinite sequences of actions it is allowed to execute. Recursive rules are defined for obtaining the constraint set for a process expression from the sets of the subexpressions. Strong and weak bisimulation are modified to account for this information and proof techniques for showing the bisimulations are given. The approach is applied to some verification examples, including the alternating bit protocol. (See also the discussion below.)

Another interesting approach based on a finite representation of liveness/fairness constraints is introduced in [8]. LTSs in a CCS-like setting can be augmented with Büchi states, and only those infinite executions are considered where a Büchi state is visited infinitely many times. Parallel composition and hiding operators are redefined to account for the Büchi states of the parameter processes, and a testing-based preorder is developed which is a precongruence for these operators.

The method is also applied to a simple communication protocol (where the channel cannot lose messages). The sender process is specified as an LTL formula which states that after sending one or more data messages the sender must eventually receive an acknowledgement. The Büchi LTS corresponding to this formula is then used as the sender in the system. At present the approach is limited to “purely nondeterministic” Büchi processes, that is, a process must always commit itself to one visible action at a time. (This is apparently because two processes that satisfy the same LTL-formulas can have different refusal
behaviour and can therefore produce LTL-different results in parallel composition, that is, “LTL-equivalence” is not a congruence for parallel composition.

A feature of all approaches based on some finite representation of liveness/fairness constraints is that they involve simultaneously two distinct layers: the ordinary operational model and the additional infinitary constraint. As mentioned earlier, this can create the problem that the two layers conflict with each other: the fairness constraint rules out an infinite execution, while the operational model prevents the process from stopping or choosing other actions. For example, a process may not be able to execute within the given $\omega$-regular set of sequences, and a Büchi process can end up in a state from which no Büchi states can be reached. What makes the problem difficult is that this can happen as a result of composing “healthy” subprocesses that are able to execute within their allowed sequences. An example of this was shown at the end of Section 3 by using the processes in Figure 2. A similar example could also be constructed for Büchi-processes.

It should be noted, however, that it may also be possible to combine the benefits of a finite representation approach with the type of approach based on ordinary infinite LTSs that was used in this article (see the next section).

In [31, Chapter 10], a fair version of the nondeterministic (“internal”) choice operator of CSP is considered. This is implemented by prefixing the two alternatives with temporary action names, and placing the system in parallel with an “arbiter process” which determines in a fair way the choice between the alternative actions. [31] points out that it is difficult to achieve in this way the desired mathematical properties of a choice operator, such as distributivity.

In [11] a timed version of CSP, where actions and refusals of actions are augmented with time stamps, is used to specify the alternating bit protocol. By using proof rules, a safety and a liveness property are shown to hold if the communication channel eventually passes messages through.

[38] uses fixed point techniques to show that there will be no divergences in the alternating bit protocol if there exist (a priori) finite maximum values for the number of message losses between successful message deliveries in the communication channels.

The idea of a “fairness LTS” was used in an application-specific way in [28] to ensure that neither side of a bidirectional communication protocol is starved. There, the result was obtained by using a finite upper and lower limit of the LTS representing the property, together with some manual reasoning. In this article we have presented a general theory which allows using a class of fairness properties in all applicable systems.

This article extends the work in [29].

7 Conclusions

In this article we have been studying the use of fairness and liveness in process algebra. We proposed intuitively reasonable requirements for a hypothetical operator that would implement fairness constraints. However, we showed that with many fairness constraints there are incompatibilities between these requirements and most well-known semantic models. One problem is that the models do not preserve enough information about the enabledness of actions in the infinite executions of a system. We also demonstrated that the compositionality of process algebra imposes limitations on the use of fairness. This is because outside processes can interfere with the actions that are part of a fairness constraint.
However, we presented a class of fairness constraints and a corresponding fairness operator that are compatible with the ordinary LTS model and an existing compositional semantics. We showed that the operator fulfills all the stated requirements. To avoid the above inconsistency with compositionality there are two possible approaches: we can restrict the set of target processes for the fairness operator, and/or we can restrict the set of contexts within which the context-independence property is guaranteed. The approach we took here was the former.

Even though the fairness operator produces infinite LTSs, these are often part of a larger system which has a finite representation. We described a method by which we can construct a finite representation of the larger system in every case that one exists. This finite representation can be used as a component in further compositional analysis. However, checking whether a finite representation exists is more costly than constructing the representation, namely, it is PSPACE-complete in the size of an intermediate parameter system. Fortunately, the constructed finite model is a conservative estimate of the original system even when an exact finite representation does not exist.

An obvious theme for further work is the implementation of automated support for the verification approach, as discussed in Section 5.2. An interesting research topic is also the possibility of strengthening our semantic equivalence in order to make it a congruence for other types of fairness constraints.

It is also important to note that our approach does not rule out finite representation of liveness/fairness properties, for example by Büchi automata or -regular languages. These could be used as finite representations for certain well-defined infinite ordinary LTSs. Because the latter can be composed by using ordinary process composition operators, the behavioural properties of the finite representations could be proved, rather than defined, by us. In this way we would obtain the benefits of finite representations while avoiding any inconsistencies between the operational model and the fairness information.

Acknowledgements This work has been funded by the TISE Graduate School and the Academy of Finland, projects “Unifying Action-Based and State-Based Verification Techniques” and “Compositional Verification Algorithms”. The author is grateful to Antti Valmari and Jaco Geldenhuys for useful comments on this article.

References


Compositional Construction of Protocol Behaviours with Arbitrary Channel Capacities

Antti Puhakka
Tampere University of Technology, Software Systems Laboratory, PO Box 553, FIN-33101 Tampere, FINLAND
E-mail: anpu@cs.tut.fi

Abstract

Formal, automated verification methods for parallel systems suffer from the state explosion problem, which prevents analysis of large systems with many components. However, with compositional methods we can sometimes take advantage of regularities in the system structure, enabling verification of even arbitrarily large systems. In this paper we present a case study where we use compositional methods for communication protocol verification. The aim is to determine the externally observable behaviour of protocols that have arbitrarily large (but finite) channel capacities. We construct the behaviour of an example protocol by choosing suitable abstractions and by composing the system in an appropriate order. We also extend the result to protocol families with arbitrary finite numbers of retranmissions. The protocol behaviour thus obtained can be used as a component in any larger system, as well as for verification and visualisation purposes. However, we also show that a similar approach is not possible for the classical alternating bit protocol with reliable or fair channels. Namely, we show that no finite-state invariant can prove that the behaviour is independent of channel capacities.

Keywords: verification, parallel systems, protocols, compositionality, arbitrary channel capacities

1. Introduction

It is well known that the design of parallel and distributed software/hardware systems is difficult, time-consuming and expensive. This is partly because these systems are prone to elusive concurrency-related errors, such as deadlocks, livelocks and unexpected execution sequences. Therefore, formal design and verification methods have been developed to help designers avoid such errors.

Often the system we are studying contains a parameter whose range can be large or infinite. For a parallel system this parameter can be the number of certain subcomponents. Formal methods based on theorem proving are often capable of proving results that hold for arbitrary values of parameters. However, this usually requires a great deal of human assistance.

On the other hand, methods based on exhaustive search of the reachable state-space of the system, such as model checking and process algebras, can often be used with little or no human intervention. However, these methods suffer from the state explosion problem, which means that a system typically has an exponential number of states in the number of parallel components. Therefore, a straightforward search of the state-space can become intractable already with a relatively small number of components.

One approach for alleviating the state explosion problem are compositional methods of system construction (see e.g. [21]). This means that we can initially construct only a chosen part of the system and replace it by a smaller but equivalent system, which can be used in place of the original system in any environment. This can also be applied to the larger system, and so on. However, even with this approach there is a limit to how many components we can use, and whether our verification results hold for larger systems remains an educated guess. If the subcomponents of the system are sufficiently similar, however, we may be able to construct the system with the help of invariants, as demonstrated in [8, 12, 13, 26].

For example, suppose we want to show that the composition $P_n$, consisting of a process $P_0$ together with $n$ identical processes $P$, satisfies some specification $S$, i.e., that $P_n = P_0 ? P ? P ? \ldots ? P \leq S$. Here, “?” is some method of parallel composition and “$\leq$” is a relation expressing that the first process is “better than” or “an implementation of” the second process. Then, we try to find an invariant, expressed as a process $I$, which is chosen to represent the expected behaviour of the arbitrarily large system. We check that $P_0 \leq I$ and $I ? P \leq I$ (this is called a pre-fixed point). Then, assuming “$\leq$” is mathematically sound (monotonic with respect to “?”), it holds that $P_0 ? P ? P ? \ldots ? P \leq I$. 

167
If we have chosen \( I \) such that \( I \leq S \), then we have proven that \( P_n \leq S \) for arbitrary \( n \). The simplest instance of this is when we choose \( I = P_0 \) and establish that \( I \cap P = I \). Then we have shown that \( P_n = P_0 \). This is called a fixed point.

Communication protocols are an important example of parallel and distributed systems. One modelling formalism for protocols are finite-state machines which communicate by unbounded FIFO-queues (CFSMs). A global state of such a system consists of the states of the machines and the contents of the queues. Although it is known that verification questions for such systems are usually undecidable [7], several practical semi-algorithms that use this model have been developed. These include QDDs (Queue-content Decision Diagrams) [4, 5] and, if the channels are also lossy, SREs (Simple Regular Expressions) [1, 2]. Symbolic reachability graphs can be generated for checking safety properties, although this does not allow compositionality.

Our analysis is based on process algebras [15, 17]. We will demonstrate that it is possible to construct a finite, compositional model of a protocol system with arbitrarily large channel capacities by using only process-algebraic techniques. We construct a channel (=queue, buffer) from primitive processes representing channels of capacity 1 (see e.g. Chapter 5 in [17]).

In this article we use the CFFD semantic model (Chaos-Free Failures Divergences) [25], which resembles the CSP-semantics [17], but preserves information on the system behaviour even after the occurrence of divergences. CFFD covers the properties expressible in linear-time temporal logic [14] without the next-state operator [11], and is well suited for typical process-algebraic verification methods [20]. Most of the analyses presented here could also be carried through with the CSP-model. The only exception is the part dealing with the classical alternating bit protocol and unreliable channels, which requires CFFD because of the presence of divergences.

Compositional construction with CSP was used in [18] to show the presence/absence of deadlocks in rings of arbitrary numbers of dining philosophers. This was achieved by adding philosophers in chunks of 10, which after hiding reduced to a fixed number (4) of states.

In [10] it was verified that a unidirectional version of the sliding window protocol [19] with channels of arbitrary capacity fulfills one safety and three liveness properties expressed in linear temporal logic [14]. This was shown by attaching a hidden data source to the protocol and constructing the system compositionally by using an appropriate invariant (abstraction) for the property to be verified. Based on data-independence, it was shown that the same properties would hold with any data source.

In [24] it was shown that the behaviour of a self-synchronizing alternating bit protocol (also used in this paper) was independent of the finite maximum number of retransmissions. The number was represented by connecting an appropriate number of “counter-cells” to the system, whereby a fixed point was found. Furthermore, similar cells were used to represent the number of messages the channels are allowed to lose before delivering a message. In this way it was shown that the classical alternating bit protocol (with an infinite number of retransmissions) is independent of the finite maximum number of losses.

In [24], channels of capacity 1 were used. In this paper we will construct the behaviour of similar protocols with channels of arbitrary finite capacity. It should be noted that our aims are different from those in e.g. [10]. We do not merely intend to show that the protocol satisfies a given set of properties, but we will construct the externally observable behaviour (semantic model) of the arbitrarily large system. In addition to verifying properties preserved by the equivalence (e.g. LTL−x for CFFD), this allows us to use the result thus obtained as a component in any larger system. Furthermore, it can be used for visualisation of the system behaviour, as will be demonstrated by the behaviour graphs in this article.

The remainder of the paper is organized as follows. In the next section we give the basic definitions, and in Section 3 we describe the main protocol we use in this article. In Section 4 we consider channels with arbitrary capacities, and in Section 5 we construct the behaviour of the protocol with these channels. We also consider different numbers of message retransmissions. In Section 6 we give results for reliable channels, and in Section 7 we consider the classical alternating bit protocol. Finally, Section 8 concludes the paper.

2. Background

For completeness, we will next briefly review the theoretical background. However, most of the paper can be understood even without knowing all the details of the mathematical definitions. The most fundamental ideas in this approach are that we model a system and its components as processes which are combined with the parallel composition operator and other operators, and that we compare processes by using a well-behaved concept of equivalence (“≃”) and preorder (“≤”).

The behaviour of a process consists of executing actions. There are two kinds of actions: visible and invisible. Visible actions are used to communicate with the outside world. Invisible actions represent internal processing, and are denoted with a special symbol \( τ \). Here, the behaviour of a process is represented as a labelled transition system. It is a directed graph whose edges are labelled with action names and with one state distinguished as the initial state.
Definition 1 A labelled transition system, abbreviated LTS, is a four-tuple \((S, \Sigma, \Delta, \hat{s})\), where
- \(S\) is the set of states,
- \(\Sigma\), the alphabet, is the set of the visible actions of the process; we assume that \(\tau \notin \Sigma\).
- \(\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S\) is the set of transitions, and
- \(\hat{s} \in S\) is the initial state.

Let \(A^*\) denote the set of finite and \(A^\omega\) infinite strings of elements of a set \(A\). The empty string is denoted with \(\varepsilon\). We use \(s - a \rightarrow s'\) as an abbreviation for \((s, a, s') \in \Delta\), and this is extended in the obvious way to \(s - \tau \rightarrow\) and \(s - \xi \rightarrow\), where \(\sigma\) is a finite and \(\xi\) a finite or infinite sequence of actions. We write \(s = \rho \Rightarrow s'\) if and only if there is \(\sigma\) such that \(s - \sigma \rightarrow s'\) and \(\rho\) is the result of removing all \(\tau\)-actions from \(\sigma\). \(s = \rho \Rightarrow\) is defined similarly. We will need the following semantic sets extracted from an LTS. A trace of an LTS is the sequence of visible actions generated by any finite execution that starts in the initial state. An infinite execution that starts in the initial state generates either an infinite trace or a divergence trace, depending on whether the number of visible actions in the execution is infinite. The stable failures describe the ability of the LTS to refuse actions after executing a particular trace.

Definition 2 Let \(L = (S, \Sigma, \Delta, \hat{s})\) be an LTS.
- \(\text{Tr}(L) = \{ \sigma \in \Sigma^* \mid \hat{s} = \sigma \Rightarrow \}\) is the set of the traces of \(L\).
- \(\text{Inftr}(L) = \{ \xi \in \Sigma^\omega \mid \hat{s} = \xi \Rightarrow \}\) is the set of the infinite traces of \(L\).
- \(\text{Divtr}(L) = \{ \sigma \in \Sigma^* \mid \exists s : \hat{s} = \sigma \Rightarrow s \land s - \tau^\omega \Rightarrow \}\), where \(\tau^\omega\) denotes an infinite sequence of \(\tau\)-actions, is the set of the divergence traces of \(L\).
- \(\text{Sfail}(L) = \{ (\sigma, A) \in \Sigma^* \times 2^S \mid \exists s \in S : \hat{s} = \sigma \Rightarrow s \land \forall a \in A \cup \{\tau\} : \neg(s - a \rightarrow \)\}\) is the set of the stable failures of \(L\).

Processes interact by executing actions synchronously. The parallel composition operator used here forces precisely those component processes to participate in the execution of a visible action that have that action in their alphabets. Synchronization of actions is thus determined by the process alphabets. The invisible action is always executed by exactly one component process at a time. We first define the product of LTSs as the LTS that satisfies the above description and has the Cartesian product of component state sets as its set of states, and then define parallel composition by picking the part of the product that is reachable from the initial state of the product.

Definition 3 Let \(L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)\) and \(L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)\) be LTSs. Their product is the LTS \((S', \Sigma, \Delta', \hat{s})\) such that the following hold:
- \(S' = S_1 \times S_2\)
- \(\Sigma = \Sigma_1 \cup \Sigma_2\)
- \((s_1, s_2), a, (s'_1, s'_2)\) \(\in \Delta'\) if and only if either
  - \(a \in (\Sigma_1 \cup \{\tau\}) - \Delta_1 \land s'_1 = s_2\), or
  - \(a \in (\Sigma_2 \cup \{\tau\}) - \Delta_2 \land s'_1 = s_1\), or
  - \(a \in \Delta_1 \land s'_1 = (s_2, a, s'_2) \in \Delta_2\).
- \(\hat{s} = (\hat{s}_1, \hat{s}_2)\)

The parallel composition \(L_1 || L_2\) is the LTS \((S, \Sigma, \Delta, \hat{s})\) such that
- \(S = \{ s \in S' \mid \exists a \in \Sigma^* : \hat{s} = \sigma \Rightarrow s \}\)
- \(\Delta = \Delta' \cap (S \times (\Sigma \cup \{\tau\}) \times S)\)

The hiding operator converts visible actions into \(\tau\)-actions and removes them from the alphabet.

Definition 4 Let \(L = (S, \Sigma, \Delta, \hat{s})\) be an LTS, and \(A\) any set of action names. Then \(L \setminus A\) is the LTS \((S, \Sigma_1', \Delta', \hat{s})\) such that the following hold:
- \(\Sigma_1' = \Sigma - A\)
- \((s, a, s') \in \Delta'\) if and only if \(a = \tau \land \exists b \in A : (s, b, s') \in \Delta\) or \(a \notin A \land (s, a, s') \in \Delta\).

The renaming operator converts visible actions into other visible actions:

Definition 5 Let \(L = (S, \Sigma, \Delta, \hat{s})\) be an LTS, and let \(a_1, \ldots, a_n, b_1, \ldots, b_n\) be visible action names. Then \(L[b_1, \ldots, b_n/a_1, \ldots, a_n]\) is the LTS \((S, \Sigma', \Delta', \hat{s})\) such that the following hold:
- \(\Sigma' = \Sigma - \{a_1, \ldots, a_n\} \cup \{b_i \mid a_i \in \Sigma\}\)
- \((s, b, s') \in \Delta'\) if and only if there is \((s, a, s') \in \Delta\) such that either \(b = a \notin \{a_1, \ldots, a_n\}\) or there is \(1 \leq i \leq n\) such that \(b = b_i\) and \(a = a_i\).

This definition allows multiple renaming. For example, \([a, a' / a, a]\) converts edges labelled with \(a\) into two edges labelled with \(a\) and \(a'\), respectively. We then use the abbreviation \([\{a, a'\} / a]\).

We now define the CFFD semantic model and equivalence, which will be our main equivalence notion in this article. Intuitively, equivalence between processes means that their behaviour is the same regarding the properties we are interested in. Preorder, on the other hand, means that the smaller process is ‘better’ or ‘more deterministic’ than the larger one. This can also be seen as an implementation relation, so that the larger process is a specification and the smaller process is an implementation of that specification.
The CFFD-equivalence/preorder is a congruence/precongruence with respect to parallel composition, hiding and renaming this component is not needed, so we will not use it here.

An important property of an equivalence is that when a component process in a system is replaced by an equivalent one, the system remains equivalent to the original one. We will not use it here.

An important property of an equivalence is that when a component process in a system is replaced by an equivalent one, the system remains equivalent to the original one. We will not use it here.

A preorder \( \leq \) is a precongruence (monotonic) with respect to the operators we use, meaning that when a component process is replaced by a smaller or equivalent process the system will be smaller or equivalent.

**Definition 8** A preorder \( \leq \) is a precongruence with respect to a process operator \( \text{op} (L_1, \ldots, L_n) \) iff \( L_1 \leq L'_1 \land \cdots \land L_n \leq L'_n \) implies \( \text{op}(L_1, \ldots, L_n) \leq \text{op}(L'_1, \ldots, L'_n) \).

The CFFD-equivalence/preorder is a congruence/precongruence with respect to parallel composition, hiding and renaming (see e.g. [25]).

The experiments described in this paper have been carried out using the ARA-toolset [23], which has facilities for constructing LTSs from LOTOS [6] descriptions and for reducing and comparing LTSs according to CFFD-semantics. Furthermore, it has a visualisation tool that shows the behaviour of a system as a graph where different actions are denoted by different colours. The layouts of the behaviour graphs shown in this article have been automatically produced by the tool, only the names of actions have been added manually.

### 3. Self-Synchronizing Alternating Bit Protocol

The protocol that we will use in the following two sections is taken from [22]. It is an improved version of the well-known alternating bit protocol [3], with an additional error recovery/synchronization mechanism. The protocol is intended for sending messages over unreliable communication channels that can lose, but not reorder, messages. There are two channels, one for data from the sender to the receiver, and another for acknowledgements, as shown in Figure 1. The acknowledgements are needed because messages can be lost. If an acknowledgement for a message is not received in time, the protocol attempts retransmission. In order not to confuse new messages with retransmissions, all messages and acknowledgements contain a sequence number, which in the alternating bit protocol is either 0 or 1.

The original alternating bit protocol makes an unbounded number of retransmissions, until it receives a correct acknowledgement. The improved version in [22] attempts only a finite number of times after which it gives up and declares an error. However, since the sender does not know whether it was the data or the acknowledgement that disappeared – or whether they were just delayed – it cannot know what sequence number the receiver is expecting next. Therefore, before sending new data messages, it sends a special synchronization message, which conveys no data but which the receiver acknowledges in the usual manner. Only after receiving a correct acknowledgement does the sender send new data messages. In a sense, the synchronization procedure means that the earlier failed transmission is forced to a completion, although without the data.

The LTSs of the sender \((Sd)\), receiver \((Rc)\) and 1-capacity data \((DC)\) and acknowledgement channel \((AC)\) are shown in Figure 2. In these graphs \textit{send} means a send data request for the protocol, \textit{err} means declaring an error and \textit{rec} denotes receipt of the data on the other side. Action \textit{sd0} means sending data with sequence number 0 to the channel, \textit{rd0} reception of the data on the other side, \textit{ss0} sending a synchronization message, and so forth. The initial state is denoted with a small arrow. In order to make the protocol more error-tolerant, it has been made to perform a synchronization at startup. For simplicity, we have not modelled the information content of data messages. This is not a big omission, however, because the data does not directly affect the behaviour of the protocol. Also, we could easily add any finite set of possible data values to the protocol components and run the same analyses, provided, of course, that the tools can still handle the state-spaces.
tially, we assume that the sender attempts transmission only once, and after that we will add the two \( a \) ("again")-actions in Figure 2 and show how to deal with an arbitrary finite number of retransmissions.

The protocol with channels of capacity 1, when only the external actions \{send, err, rec\} are left visible is:

\[
P_{1,1} = (Sd || DC || AC || Rc) \setminus I
\]

where \( I = \{sd0, sd1, ss0, ss1, rd0, rd1, rs0, rs1, sa0, sa1, ra0, ra1\} \) is the set of internal actions. The CFFD-reduced global behaviour of \( P_{1,1} \) produced by the ARA toolset is shown in Figure 3. We note that after send there is either delivery with rec or the system may declare an error with err. Error may also be declared if the message has been delayed. We see that there cannot be more rec- than send-actions. There can sometimes be two successive rec-actions but, as was verified in [22], this is just a consequence of the ability of the system to hold messages: a new sending request may have been issued before the previous message was delivered, thus resulting in two subsequent deliveries.

4. Arbitrary Finite Channel Capacities

We would now like to see how the protocol behaves with data and acknowledgement channels of arbitrary finite capacities. We can construct an \( n \)-capacity data channel \( DC_n \) from \( n \) primitive channels \( DC \) of capacity 1 as follows:

\[
DC_1 = DC

DC_{n+1} = (DC_n[X/S] || DC[X/R]) \setminus X
\]

Here, \( S = \{sd0, sd1, ss0, ss1\} \) and \( R = \{rd0, rd1, rs0, rs1\} \). \( X \) is any unique set of temporary actions names by which the successive channel elements are made to communicate, and which is hidden after the parallel composition. \( AC_n \) is constructed similarly, with \( S = \{sa0, sa1\}, R = \{ra0, ra1\}: \)

\[
AC_1 = AC

AC_{n+1} = (AC_n[X/S] || AC[X/R]) \setminus X
\]

Let \( m, n \geq 1 \). The system we consider is the following:

\[
P_{m,n} = (Sd || DC_m || AC_n || Rc) \setminus I
\]

Obviously, the state space of the parallel construction grows unboundedly with \( m \) and \( n \). However, based on our intuitive understanding of the protocol we can conjecture that the externally observable behaviour of the protocol is actually independent of channel capacities. However, channels with arbitrarily large capacities \( n \) cannot directly be replaced by any channel of fixed capacity. This is essentially because we can input to a channel of capacity \( n \) a message sequence of length \( n \), after which it can be output on the other end. For example, \( DC_n \) has the following trace of actions that no channel \( DC_k \) with \( k < n \) can have:

\[
\text{sd0 sd0 \ldots sd0 rd0 rd0 \ldots rd0}
\]
If we try to construct the system in other ways, we will encounter similar problems. For example, if we add \( n \) DC-elements to \( Sd \), the resulting system can execute \( n \) times \( send \) \( err \) and then \( n \) times \( r/s0 \) (at the end of the channel), which is not possible with a lesser number of elements. However, we can guess that in the complete system repeated messages are unimportant for the behaviour of the protocol. In fact, we make the following intuitive conjecture:

Inputting repeated acknowledgement messages to \( Sd \) has no effect on its behaviour, and inputting repeated data and synchronization messages to the combination of \( Rc, AC_n \) and \( Sd \) has no effect on the behaviour of this combination.

The conjecture suggests that we actually have to construct the system in a direction opposite to the flow of messages. Therefore, in the first phase of our analysis we add \( AC \)-elements one by one to \( Sd \), and then add \( Rc \). In the second phase we will add \( DC \)-elements one by one, starting from \( Rc \) and working towards \( Sd \). This is illustrated in Figure 4. The (intuitively obvious) fact that we can construct the system in this way is shown by inserting the definitions (1) and (2) to the expression (3) for \( F_{m,n} \), and reordering the terms by using the properties of renaming, parallel composition and hiding. We omit the proof which is simple but tedious. The recursive construction we obtain for the first phase is the following:

\[
SA_1 = (Sd \parallel AC)\backslash R \\
SA_{n+1} = (SA_n[R/S] \parallel AC)\backslash R \\
SAR_n = (SA_n \parallel Rc)\backslash S
\]  

(4)

Then, for the second phase:

\[
SARD_{1,n} = (SAR_n \parallel DC[X/S])\backslash R[R/X] \\
SARD_{k+1,n} = (SARD_{k,n} \parallel DC[X/S])\backslash R[R/X]
\]  

(5)

Here, we used the temporary actions names \( X \) to avoid confusing the \( sd0 \), etc. actions of \( Sd \) with the \( sd0 \), etc. actions of \( DC \). After the recursive construction we have to “close” the system with one \( DC \), so from now on we will assume that \( m \geq 2 \) and deal with the (trivial) special case \( m = 1 \) separately. We have:

\[
P_{m,n} = (SARD_{m-1,n} \parallel DC)\backslash R, S.
\]  

(6)

However, even this construction does not contain a (pre)fixed point. Namely, if we add \( n \) \( AC \)-elements to the sender, then we can feed the system a sequence of length \( n \) of alternating \( a0 \) and \( a1 \) messages, and then execute the sequence \( send \) \( sd0 \) \( sd1 \) \( send \) \( sd0 \) \( send \) \( sd1 \) \ldots of corresponding length. This is obviously not possible with any fewer number of \( AC \)-elements. However, we may suspect that in the complete protocol system there can never be many consecutive messages with alternating bit values in a channel, and long message sequences only consist of repeated messages. Thus, the above counter-example represents behaviour that can only exist in an incomplete fragment of the system, but not in the complete system. We will state this idea more precisely as a second conjecture. In the following, \( b \) denotes any bit value and \( \bar{b} \) the opposite value.

At all times, the joint sequence of messages in the data channel and acknowledgement channel is such that either all messages have the same bit value (i.e. the sequence is of form \( bb \ldots b \)), or the bit value changes at no more than one point (i.e. the sequence is of form \( bb \ldots bb \bar{b} \ldots \bar{b} \)).

Guided by this intuitive idea, we will construct an “abstract channel” \( AC' \), shown in Figure 5. As before, this process accepts an acknowledgement message and then either loses it or delivers it. However, it also acts as a “watchdog” by keeping track of the bit values of the message stream passing through it. For example, if it has just delivered a message with bit 0 and after that, without delivering messages, receives messages with bit 1 and then with bit 0 (meaning that our conjecture has been violated), it moves to the subprocess \( ANY' \). Process \( ANY' \) can perform any sequence of actions, but at any moment it can also refuse all output actions\(^1\). Intuitively, the idea is that by entering \( ANY' \), \( AC' \) can simulate all those behaviours that are features of the system fragment alone. Therefore it provides us a fixed point during the system construction. In the complete system, however, it will never enter \( ANY' \) and will thus behave like an ordinary channel element.

We also construct a corresponding process \( DC' \) for the data channel. It works similarly as \( AC' \), except that it has to deal with both data and synchronization messages. A check with the computer tools reveals that \( AC' \leq_{CFFD} AC \) and \( DC' \leq_{CFFD} DC \), as we would expect. This means that the protocol system with the primed elements will be CFFD-larger or equivalent (i.e. “worse”), so that it provides an upper bound for the behaviour of the original protocol.

\(^1\)The \( \tau \)-loop in \( ANY \) is not necessarily needed but it can be useful for finding a suitable abstraction, because a divergence reveals that the \( AC' \) subprocess has entered \( ANY \)
5. Using Fixed Points to Determine the Behaviour

We are now in a position to establish the required fixed points. First, we consider the system $S_A_n$ which is constructed like $S_A_n$ except that $S_A'_n = (Sd || AC') \setminus R$. A check with the computer tools reveals that $S_A'_2 \simeq_{CFFD} S_A'_1$, whereby the congruence property of CFFD-equivalence with respect to our operators implies that $S_A'_2 \simeq_{CFFD} S_A'_1$ and so forth, i.e. $S_A'_n \simeq_{CFFD} S_A'_1$ for any $n \geq 1$.

Next, we consider the system $SARD'_k,n$ which is similarly constructed starting from $SARD'_1,n = (SAR'_n || DC'[X/S]) \setminus R[R/X]$, where $DC'$ is the data channel abstraction. The above result means that $SARD'_k,n \simeq_{CFFD} SARD'_k,1$. Then, a new check reveals that $SARD'_2,1 \simeq_{CFFD} SARD'_1,1$, implying $SARD'_k,1 \simeq_{CFFD} SARD'_1,1$. Therefore, $SARD'_k,n \simeq_{CFFD} SARD'_1,1$ for any $k, n \geq 1$.

With $P_m,n = (SARD'_m-1,n || DC) \setminus R[S_n]$, it now holds that $P_m,n \simeq_{CFFD} P_1,1$ for $m \geq 2, n \geq 1$. As for the special case $m = 1$, $P'_1,n = (SAR'_n || DC) \setminus S[R \simeq_{CFFD} (SAR'_1 || DC) \setminus S \simeq_{CFFD} P_1,1$. Combining these results, we have, for any $m, n \geq 1$:

$$P_m,n \simeq_{CFFD} P_1,1$$

The fact that $AC \leq_{CFFD} AC', DC \leq_{CFFD} DC'$ and the monotonicity of $\leq_{CFFD}$ immediately imply that $P_m,n \simeq_{CFFD} P'_m,n$. Since the latter is now equivalent to $P'_1,1$, we have a fixed-size upper bound for $P_m,n$. The result of constructing its behaviour turns out to be the same as that of $P_1,1$ (Figure 3). Thus

$$P_m,n \leq_{CFFD} P_1,1$$

To establish a lower bound for $P_m,n$, we show that by increasing the channel capacity the behaviour of the protocol does not become smaller in CFFD. We use the ordinary channel elements, and a check reveals that $S_A_1 \leq_{CFFD} S_A_2$, which implies $S_A_1 \leq_{CFFD} S_A_n$ for all $n$. Then by the monotonicity property $SARD'_k,1 \leq_{CFFD} SARD'_k,n$ for all $k, n$. A similar check reveals that $SARD'_1,1 \leq_{CFFD} SARD'_2,1$, implying $SARD'_1,1 \leq_{CFFD} SARD'_1,n$. Thus, $SARD'_1,1 \leq_{CFFD} SARD'_k,n$, and for all $m \geq 2, n \geq 1$:

$$P_1,1 \leq_{CFFD} P_m,n$$

By a similar argument as above the result holds also for the special case $m = 1$. Combining the upper (7) and lower bound (8) we have the exact result we have been looking for:

$$P_m,n \simeq_{CFFD} P_1,1$$

for any $m, n \geq 1$. Thus, the externally observable behaviour (according to CFFD) of $P_m,n$ is that shown in Figure 3.

It should be noted that the channel abstractions $AC'$ and $DC'$ are not themselves invariants with respect to adding channel elements, and if we tried to make them such e.g. by allowing unbounded repetition of messages, this would cause divergences (livelocks) in the system behaviour. Therefore, the processes $S_A'_1$ and $SARD'_1,1$ were the actual invariants of the proof. Typically, such invariants are constructed directly. However, given the nature of the system under consideration, we found it easier to first encode our intuitive understanding of the protocol behaviour into the channel abstractions, and to build the invariants from these. For example, $AC'$ and $DC'$ have 10 and 14 states, respectively, while the (CFFD-reduced) invariants $S_A'_1$ and $SARD'_1,1$ have 54 and 380 states, respectively.

Next we consider the protocol with an arbitrary maximum number of message retransmissions. Let $Sd'$ be obtained from $Sd$ by adding the two $a$ (“again”)-actions shown in Figure 2. To allow a specific number $n$ of retransmissions, we apply the “counter-cell” technique from [24]. The counter cell $C$ in Figure 6 allows (at most) one $a$-action after each send. We can add an arbitrary number $n$ of these cells by using multiple renaming. Formally, let $Sd'_0 = Sd'$, and $Sd'_{n+1} = (Sd'_n \{a,a'/a\} || C[a'/a]) \setminus \{a'\}$. Thus, the system and the new cell communicate by the new primed action which is then hidden. Finally, we put a “lid” on the system by using the process $STOP \{a\}$, also shown in Figure 6, which blocks the action $a$: $Sd_n = (Sd'_n || STOP \{a\}) \setminus \{a\}$.

A simple reorganization of terms shows that we can add the cells after constructing the protocol. Therefore, we replaced $Sd$ with $Sd'$ and ran the same tests as above. The results were found to be the same. After closing the system with the last $DC$, we started adding counter cells recursively to the system. A fixed point was immediately found. Then we finished the system by putting the “lid” on it, and
messages with \(b\). This is needed for simulating a channel of larger capacity.

Also, there is no fixed point, essentially because after receiving \(n\) messages a channel of capacity \(n\) can refuse new messages, but one of greater capacity cannot. However, this actually works for us, so that there is a *pre-fixed* point. For example, \(SA''_m \leq_{\mathrm{CFFD}} SA''_1\), which implies \(SA''_m \leq_{\mathrm{CFFD}} SA''_1\), and so on. We were therefore able to establish that the upper bound \(P''_{m,n}\) of the behaviour is the LTS in Figure 7 a). It is easy to see that no process is strictly smaller than this in CFFD-preorder, so this is also the exact behaviour of \(P''_{m,n}\). Furthermore, we were able to extend the result to the family of protocols with an arbitrary maximum number of retransmissions by using the same counter cell technique as above.

6. Reliable Channels

Intuitively, we might think that if we make the underlying communication media more reliable, this will be just an easy special case of the system with unreliable media. However, this is not always true. The correct behaviour of some protocols requires that the channels are capable of losing out-of-date or unnecessary messages. Also, some verification methods (e.g. [1, 2]) are based on an assumption of unreliable channels. Therefore, we will next investigate the applicability of the above method for reliable channels.

The reliable channels we use are like those in Figure 2 but without the \(\tau\)-actions (similar techniques could be used also for a channel that can lose some limited number of messages, e.g. at most every second message). For the above protocol (with some additional actions for consuming unexpected acknowledgements, to avoid deadlock) no fixed points could be found, for a very simple reason: the externally observable behaviour is *not* independent of the capacities of the reliable channels. For example, the capacity of the data channel determines how many failed \(\text{send}\)-actions there can be before the first reception of a message (\(\text{rec}\)).

We therefore removed the synchronisation mechanism, thus reducing the protocol to the alternating bit protocol with one message transmission. The behaviour with reliable channels of capacity 1 is shown in Figure 7 a). For arbitrarily large channel capacities we needed new abstractions \(AC''_m\) and \(DC''_m\), which are capable of refusing new messages when full. We also replaced process \(\text{ANY}\) with process \(\text{CHAOS}\), which is capable of refusing any actions and is the CFFD-maximal process for its alphabet. \(DC''_m\) and \(\text{CHAOS}\) are shown in Figure 8. An interesting feature in \(DC''_m\) is that when it has received the first message with bit value \(b\), it can output one message with \(b\) or two messages with \(b\) and \(b\), respectively. This is needed to simulate the situation when a channel of capacity \(1\) still contains messages with \(b\) when the new message arrives, and these have to be output first. Furthermore, even though the original channel is reliable, \(DC''_m\) can dispose of subsequent

[Figure 6. The counter cell \(C\) and \(\text{STOP}_{\{a\}}\), with alphabets shown]

[Figure 7. Two protocol behaviours]
contains $AC_n$, it is easy to see that there are essentially three possibilities:

a) $I$ models a composition in which $AC_n$ is connected to $Sd$, which is possibly further connected to the other processes $DC$, $Rc$. Initially, $I$ has to be able to receive at least $n$ acknowledgements with $i$ ($sa0$ in our example), because $AC_n$ is able to do so. As $n$ grows unboundedly, $I$ has to be able to receive an arbitrarily long sequence of such messages. However, from the well-known König’s lemma it then follows that if $I$ is finite-state (or even finitely branching), then it is necessarily also capable of receiving an infinite sequence. Because $Sd$ can send an infinite sequence of these messages, it follows that when $I$ is combined (with a finite segment of channel) to the rest of the system, and the reception of these events (e.g. $sa0$) is hidden, this infinite sequence of events turns into a divergence.

b) $I$ models only $AC_n$. We can argue as in a).

c) $I$ models $AC_n$ connected to $Rc$, which is possibly connected to the other processes. $AC_n$ can store $n$ messages with $i$ before the first one is received by $Sd$, so it can deliver $n-1$ of them after that. As $n$ grows unboundedly, by a similar argument as above, a finite-state $I$ has to be able to deliver an infinite sequence of such messages (e.g. $ra0$). $Sd$ can receive an infinite sequence of these messages after the next send, because otherwise, unless it already contained a divergence, it would have a refusal (stable failure) where after some $k$ such messages it refutes new messages with sequence number $i$. Then, with $n > k$ this refusal could block the flow of messages and the protocol would not be correct. Therefore, when the system is completed, the infinite sequence of message deliveries (e.g. $ra0$) turns into a divergence.

The essential difference between this protocol and the above family of protocols with some finite maximum number of retransmissions is that the ability of the former to send retransmissions grows unboundedly with channel capacities, while for each member of the latter this ability does not grow after the maximum is reached. It should also be noted that the above argument does not depend on an assumption of reliable channels. Therefore, it holds also for all channels that are not fully reliable but fair [9] in the sense that they cannot from some point on lose all messages (this includes e.g. the channel that can lose at most every second message). If the channels are unreliable then, as described above, a finite-state invariant can be used because the protocol behaviour contains the divergences anyway.

8. Conclusions

In this paper we have been dealing with communication protocols and channels with arbitrarily large capacities. This is an important example of a family of systems where the state-space is unboundedly large. We demonstrated that it is possible to construct the externally observable behaviour of such protocols by using compositional, process-algebraic methods. We encoded our intuitive ideas about the protocol behaviour into channel abstractions, and built invariants from the abstractions and protocol components. Then, by using a suitable order of construction, we were able to find (pre)fixed points, which were checked automatically by computer tools. An important benefit of this approach is that the obtained behaviour can be used as a component in any larger system, as well as for verification and visualisation. The results were also extended to families of protocols with arbitrary maximum numbers of retransmissions. We used examples for both lossy and reliable channels.

On the other hand, the capacity of reliable channels can sometimes cause subtle differences in the external behaviour of some protocols. Furthermore, we showed that with reliable/fair channels it is not possible to use a finite-state invariant to prove that the behaviour of the classical alternating bit protocol is independent of channel capacities in a semantics that can detect divergences. Such results are not entirely unexpected, as it was shown in [26] that finite-state invariants do not necessarily exist even when the property to be proved holds.

Of course, nothing in principle prohibits the use of infinite (infinitely branching) invariants in a proof, although achieving the same level of automation is obviously more difficult. However, we can expect a highly regular structure in such invariants, and techniques for dealing with regular infinite LTSs have been developed e.g. in [16] in connection with fairness. This obviously requires the use of a semantics that is a congruence for infinitely branching systems, such as CFFD.

It should also be noted that the absence of a finite
invariant was caused by the ability of the alternating bit protocol to send the same message infinitely many times. A realistic protocol seldom behaves in this way. Rather, at some point it gives up and enters an error handling procedure (like our self-synchronizing protocol). So, the ability to retransmit infinitely is essentially an abstraction which is used to make the protocol model simpler.

Acknowledgements

This work has been supported by the Academy of Finland, project “Unifying Action-Based and State-Based Verification Techniques”, and the TISE Graduate School.

References


