Department of Computer Science
Series of Publications A
Report A-1996-1

Equivalences, Preorders and Compositional Verification for Linear Time Temporal Logic and Concurrent Systems

Roope Kaivola

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium IV, Porthania, on March 12th, 1996, at 2 p.m.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 0 708 51

Telefax: +358 0 708 44441

ISSN 1238-8645
ISBN 951-45-7338-2
Computing Reviews (1991) Classification: F.3.1, F.1.2, D.2.4, C.2.2
Helsinki 1996
Yliopistopaino

Equivalences, Preorders and Compositional Verification for Linear Time Temporal Logic and Concurrent Systems

Roope Kaivola

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland

Series of Publications A, Report A-1996-1
Helsinki, March 1996, 176+9 pages
ISSN 1238-8645, ISBN 951-45-7338-2

Abstract

A promising approach to formal specification and verification of finite-state concurrent systems is using propositional temporal logic as a specification language and applying automated model-checking algorithms for the verification task. However, the so-called state explosion-problem caused by representing concurrency by interleaving makes model-checking in many cases practically intractable. One way of attacking this problem is replacing the modules of a system by smaller ones so that the visible properties of the system are not affected. This approach leads to notions of compositional property-preserving equivalences and preorders between modules.

This work examines the uses of equivalences and preorders in verifying nexttime-less linear temporal logic properties. Concurrent systems are modelled here by labelled transition systems augmented with state information, and both synchronisation and read-shared variables are considered as methods of communication. We examine a novel equivalence notion, called the non-divergent failures divergences or NDFD-equivalence. It is shown that NDFD preserves all nexttime-less linear temporal logic properties, and that it is compositional with respect to parallel composition and abstraction by hiding and encapsulation. Conversely, it is shown that NDFD is the weakest such equivalence, i.e. that it is fully abstract with respect to preserving nexttime-less linear temporal logic properties in an arbitrary context. The computational complexity of determining NDFD-equivalence is also examined and the problem is shown to be PSPACE-complete.

Analogous to NDFD-equivalence, the theory of NDFD-preorder is developed and the compositionality, full abstractness and computational complexity results extended to it. Moreover, it is shown that the verification technique of modular validity introduced by Manna and Pnueli corresponds to an extremal case of NDFD-preorder. As a larger case study illustrating the practical applications of the results, we use the NDFD-preorder to verify semi-automatically both safety and liveness properties of the sliding window communication protocol for arbitrary channel lengths and realistic parameter values. In this process we locate a previously undiscovered fault leading to lack of liveness in a version of the protocol.

Computing Reviews (1991) categories and subject descriptors:

- F.3.1 [Specifying and verifying and reasoning about programs]: logics of programs, mechanical verification, specification techniques
- F.1.2 [Modes of computation]: concurrency
- D.2.4 [Program verification]
- C.2.2 [Network protocols]: protocol verification

General terms: theory, verification

Additional key words: concurrency, equivalences, preorders, specification, temporal logic

Acknowledgments

I would like to express my gratitude to my supervisor, Martti Tienari, for advising and supporting my work and encouraging me to bring it to a successful conclusion. I am also greatly indebted to Antti Valmari, who originally suggested the research problem, was very active and helpful in the joint research effort, and carefully commented on successive versions of the text. Furthermore, I would like to thank my other colleagues at the department, in particular Jaana Eloranta, Timo Karvi and Pekka Orponen for helpful discussions, and the departmental technical support staff and Jukka Kemppainen from VTT Electronics for their assistance in the experimental part of the research.

Most of the work leading to this monograph was carried out in the Department of Computer Science of the University of Helsinki between 1990 and 1992. Financial support from the Academy of Finland and the University of Helsinki during this period is gratefully acknowledged. From autumn 1992 I had the opportunity to pursue research in the Laboratory for Foundations of Computer Science in the University of Edinburgh, thanks to funding from the Academy of Finland, the British Council, the CVCP Overseas Research Students Awards Scheme and the Finnish Cultural Foundation. Although not directly related to my work in Edinburgh, the final form of the current text has benefited from exposure to a different research culture and perspective.

Finally, thanks are due to my family and friends, both in Helsinki and in Edinburgh, for their support and for occasionally dragging me away from science. Above all, I would like to thank Marjukka Barron for bearing with me despite the many days and nights this work has kept me away.

Contents

Chapter 1

Introduction

1.1 Overview

The basic aim of verification methods is trying to ascertain that an actual system or program works according to its requirements. In formal verification methods one tries to achieve this aim by describing the system by a mathematical model, expressing the requirements as properties of this model and by showing through rigorous mathematical reasoning that the model of the system indeed has the required properties.

This work concentrates on some aspects of formal verification of concurrent systems. There are several competing approaches to formal verification of such systems, and no general consensus on the respective merits and applicability of the various methods has been reached yet. A strong research trend in the field is using *temporal logics* as specification languages. These are logic-based formal languages containing temporal operators such as *sometimes* or *always*. For surveys of temporal logics in computer science see [Eme90, Sti92], and for temporal logics in a more traditional philosohical setting see [Bur84, RU71].

A common approach to using temporal logic in verification is based on modelling the behaviour of a system by a graph which can be directly interpreted as a temporal logic model, and expressing the required properties of the system as temporal logic formulae. This means that the issue of the system having the required properties can be reduced to verifying that the temporal logic formulae corresponding to the required properties are true in the model corresponding to the system. This so-called *model-checking* problem can usually be decided mechanically, if the execution graph of the program is finite and the required properties can be expressed by propositional temporal logic. In this case an automated model-checking algorithm can be applied in order to find out whether the system has the required properties.

The main obstacle to this approach is that the execution graphs of realistic concurrent systems are often too large for feasible model-checking. An essential reason for this *state-explosion problem* is the modelling of concurrency by arbitrary interleavings of the atomic actions of the concurrent processes. This means that in the general case the size of the execution graph is exponential in the number of concurrent processes.

One approach to tackling this problem is applying notions of semantic equivalence between models. Suppose such an equivalence preserves every property to be verified.

Due to the symmetricity of an equivalence relation this means that it also preserves the lack of any property to be verified. Given a model and a property we may then try to find an equivalent model that is smaller in size or otherwise easier to handle than the original one. Checking whether this equivalent model has the required property gives an immediate answer to whether the original model had the required property or not.

To take advantage of the real power of equivalences, they should be congruences with respect to the syntactic operations used for constructing systems from their components. If this is the case, the components of a system can be replaced with smaller equivalent ones before composition. As this means that the sizes of the components are reduced before the composition step causing the state-explosion problem, the result can be a dramatic reduction in the size of the system.

The current work concentrates on developing this approach for a particular framework. Let us outline this and discuss the results for the equivalence-based approach first.

The formalism used here for expressing required properties of systems is standard propositional linear-time temporal logic from which the nexttime operator has been excluded. Linear-time temporal logic is perhaps the most traditional temporal-logic-based formalism used for describing properties of systems, and the literature concerning its use in specification and verification is extensive. Removing the nexttime operator from the language decreases its expressive power. However, it seems that most of the properties considered usually interesting are expressible in this less expressive fragment. What is more, having less expressive power should not be seen only as a negative feature: With a very expressive specification language it is possible to express specifications that are too rigid and restrict the potential implementations in an unreasonable way. For example, in full linear-time temporal logic it is possible to express the requirement *an operation should take 27 steps*, which is not a reasonable requirement if a system is examined on different levels of abstraction or in the context of an environment. In this respect the nexttime-less linear-time temporal logic seems to provide a good balance between the ability to express properties we would like to express, and the inability to express properties that should not be allowed to be expressed.

The way of modelling systems discussed here is based on transition systems both the states and the transitions of which carry labels. Both complete systems and components of them are modelled by similar structures. To build systems from components we provide a set of syntactic operators. On the one hand, we look at a scenario where components of a system interact by synchronisation. The syntactic operators for this approach are based on those of the process algebra Basic LOTOS [BB89]. On the other hand, we look at a scenario where components interact by shared variables, and provide syntactic operators for modelling this [MP81, MP91]. Although these two scenarios are distinct and not usually considered together, one of the interesting observations of this work is that many of the results here are applicable to both of them.

With this framework in mind, we examine the properties of a particular equivalence called the *non-divergent failures divergences equivalence* or the *NDFD-equivalence*. This is a semantic model consisting of infinite traces, divergence traces and non-divergent failures. It is shown that this equivalence preserves all properties expressible in the nexttime-less linear-time temporal logic. Furthermore, it is also shown that the equivalence is a congruence with respect to all the syntactic operations above. Consequently, in the verification process any component of any system can be replaced with an NDFD-equivalent one without affecting the outcome of the verification.

As a converse result, it is shown that in the class of all equivalences that preserve the nexttime-less linear-time temporal logic and are congruences with respect to the syntactic operations, the NDFD-equivalence holds a special position: it is the weakest or most liberal equivalence having these properties. This means that if any equivalence is a congruence and preserves the nexttime-less linear-time temporal logic, then it must be a refinement of the NDFD-equivalence. Another way of expressing this result using terminology often found in the literature, is saying that the NDFD-equivalence is *fully abstract* with respect to preserving nexttime-less linear temporal logic properties in an arbitrary context.

These results express the necessary and sufficient conditions for being allowed to replace a component of a system with another one in the verification task. Remembering the intuitive aims of the whole approach, we should also answer another question: Given a component of a system, how to find a smaller one that can be substituted for it. In the present framework this is equivalent to asking: Given a component of a system, how to find a smaller NDFD-equivalent one.

Finding the smallest NDFD-equivalent replacement for a given component is not easy, as this problem turns out to be PSPACE-complete. However, this should not be seen as too great an obstacle, since what is required is a small replacement, not necessarily a small*est* one. Various heuristics can be used for finding such a replacement. One, discussed later in this work, is essentially based on determinisation and minimisation of a non-deterministic finite automaton. However, any other method, for example minimisation or reduction algorithms for any stronger equivalence could be used equally well.

Another approach to the state-explosion problem, closely related to the use of equivalences, is using a weaker semantic relation preserving each property to be verified but not necessarily the lack of it. This leads to the notion of semantic preorders preserving the class of properties in one direction, say downwards. Given a model and a property we may now try to find another model that is smaller in size but greater in the preorder than the original one, and check whether the new model has the required property. If this turns out to be the case, the original model had the property, as well. However, if the new model lacks the property, it is not possible to conclude anything about whether the original model lacked it or not. Consequently, in contrast with equivalences, here the result of checking whether the new model has a property does not necessarily yield a definite answer to the question whether the original model had it.

To use such preorders to show that a model lacks a property, they can be applied symmetrically in the other direction. This time, given a model and a property, we try to find another model that is lower in the preorder, and check whether it has the required property. If the new model lacks the property, then the original one lacked it, as well. Naturally, if the new model now turns out to have the property instead, it is not possible to say whether the original model had it or not.

As with equivalences, these preorders should be congruences with respect to the syntactic operations used for constructing systems from their components.

Analogously with the NDFD-equivalence, we define the NDFD-preorder. We show that it preserves all properties expressible in the nexttime-less linear-time temporal logic, and is a congruence with respect to all the syntactic operators discussed. Furthermore, we also extend the full abstractness result to the preorder.

The process of finding a suitable replacement for a given component of a system is slightly different in flavour for the preorders than for the equivalences. With equivalences any component could be replaced with any equivalent one. However, with preorders two issues have to be borne in mind when choosing a replacement. First, the replacement should naturally be greater in the preorder and smaller in size than the original component. Secondly, it should still be possible to verify that the resulting system fulfils the required property. The second issue is automatically guaranteed when using equivalences. What can happen with preorders, however, is that the replacement is in a sense too general or too far away in the preorder, and the resulting system does not fulfil the required property even if the original one did. In this sense choosing a good replacement is a balancing act between the two requirements.

When using preorders the range of possible replacements for a given component is usually much larger than when using equivalences. This can be seen both as a negative and as a positive feature. On the one hand, it means that the task of selecting a replacement is unlikely to be automatisable, although various heuristics can naturally be developed. On the other hand, it means that it may be possible to find replacements and verify systems that defy all attempts using the equivalence-based method. One good example of this is the verification of the Sliding Window communication protocol later in this work.

Another way of looking at the contrast with equivalences is that an equivalence preserves all the characteristics that are important for verifying any property in any environment. When using a preorder, however, it is possible to choose which characteristics to preserve and which not to. If we manage to single out just those characteristics which are relevant to the current property in the current environment, it may be possible to describe these very concisely, thus potentially providing great savings, and still verify that the system fulfils the required property. However, to do this we need to understand which characteristics are important for the currently verified property in the current environment and which are not.

There is one aspect of the whole approach we would like to stress. The aim of the methods is to make the verification easier than using straightforward model checking, and to make it possible to verify systems that would be beyond the reach of conventional model-checking methods. In some cases the savings achieved will be large. Equally well, it is easy to see that in some other cases the methods will provide no help at all; for example, if there are no smaller NDFD-equivalent replacements for any components of a system, then NDFD-equivalence is not going to help in the verification. Consequently, the approach cannot provide a universal cure for the state-explosion problem, and the real criterion for the practical usefulness of the method is to what extent it can be applied to realistic systems.

What is more, inherent in the approach is that it is not necessarily possible to tell in advance whether it is going to provide savings with a given system or not. Intuitively, the process of finding a small NDFD-equivalent replacement for a given component is that of abstracting the relevant information from it and representing this information in as concise a way as possible. As far as we can see, there is no easy way of determining how concisely the relevant information can be represented, other than actually determining what it is and trying to represent it. Consequently, given a component of a system, there seems to be no easy way to predict a priori how great savings the method is going to give. What can be naturally done is to give some rules of thumb about situations in which the method is likely to be succesfully applicable. However, ultimately the only way to find out whether the method can be succesfully applied to a given system is by trying it out and seeing whether it helps.

As stated above, the real criterion for the usefulness of the methods discussed in this work is their applicability to realistic systems. To illustrate this, we have examined as a case study the verification of the Sliding Window or SW communication protocol [Ste76], which forms the basis of the data transfer function of the high-level data link control or HDLC-protocol [ISO79a, ISO79b, ISO80]. An extreme simplification of the SW-protocol is the so-called Alternating Bit Protocol, which is one of the stock examples in the literature on formal verification of concurrent systems. Although almost any verification method seems to be able to cope with the Alternating Bit Protocol, the situation becomes much more difficult when moving to the more general SW-protocol. Even with relatively small window sizes and channel lengths the total state space of the protocol becomes far too large for normal model-checking or related methods.

In this work we have used the compositional verification technique based on the NDFD-preorder to verify that the SW-protocol meets its safety and liveness properties for a variety of window sizes and for *arbitrary* channel lengths. A more surprising observation in this case study, however, was that a variant of the SW-protocol actually *fails* to fulfil the standard liveness requirements. This variant of the original protocol arises from a modification that is seemingly small and harmless enough that it has occasionally been unwittingly made in the literature, causing the faulty variant to be used instead of the correct protocol. In our opinion this result is important in illustrating the advantages of rigorous formal verification, as the error caused by the modification is so subtle that it would be very easy to overlook in an informal analysis.

The issues introduced above are discussed in much more detail in later chapters. In Chapter 2 the mathematical structures modelling concurrent systems and the syntactic operators modelling process composition methods are introduced. In Chapter 3 the linear temporal logic is introduced and its uses in program verification are briefly discussed.

The main body of the work is in Chapter 4. We first examine equivalences that preserve nexttime-less linear temporal logic in Section 4.1, and look at compositionality issues and define the NDFD-equivalence in Section 4.2. The compositionality of the NDFD-equivalence is shown in Section 4.3, and the full abstractness or optimality result in Section 4.4. The algorithmic aspects of the equivalence are discussed in Section 4.5. In Section 4.6 various extensions of the previous results are considered. These include adding new composition operators in Subsection 4.6.1, refining the NDFD-equivalence to cope with a choice-operator in Subsection 4.6.2, and extending the full abstractness result to infinite systems in Subsection 4.6.3.

Chapter 5 discusses the preorder-based approach and shows that most results for the NDFD-equivalence also apply to the NDFD-preorder. In Section 5.2 we look at the relation of the NDFD-preorder and the concept of *modular validity* introduced in [MP91]. Chapter 6 contains some examples of using the approach in practice, most notably the SW-protocol. Finally, some comparison is made to related research in Chapter 7.

1.2 Research history

Many of the ideas and results appearing in this monograph were originally presented, at least implicitly, in the research articles [KV91, KV92, Kai92]. However, the monograph forms a more polished and coherent whole than these articles, and aims to be more approachable by containing more introductory material. An early version of this monograph was submitted as the author's Lic. Phil. thesis [Kai93].

The first research article [KV91] discusses the use of a failure-based process algebraic equivalence called the CFFD or *chaos-free failures divergences* equivalence in verifying nexttime-less linear temporal logic properties of systems interacting by synchronisation. The paper also examines modifications to an algorithm for CFFD-preserving reductions which would enable the algorithm to be used in the temporal logic context, as well. This research direction is continued in the second article [KV92], which shows that a close variant of CFFD, the NDFD-equivalence, is fully abstract with respect to preserving all nexttime-less linear temporal logic properties in an arbitrary context, and that the CFFD-equivalence itself is fully abstract with respect to a slightly extended version of the nexttime-less linear temporal logic in the same way.

The third research article [Kai92] examines methods for deciding whether a nexttime-less linear temporal logic formula holds of a shared variable module in all environments. As will be shown in Section 5.2, this can be seen as one extremity in modular verification by the NDFD-preorder. The paper also discusses fairness issues, which are not addressed in this monograph.

An essential difference between this monograph and [KV91, KV92] is that the process model considered here is more general, having both state and transition labels, whereas in the articles the process model conformed to the process-algebraic tradition and had only transition labels. As temporal logic formulae are traditionally used to describe how situations or momentary states evolve, the articles had to introduce special conventions on how state labels could be reconstructed from transition labels. This meant that transition labels specified both how the state labelling of a process changes and how the process interacts with its environment. In order to cater for this dual function, [KV91] used a somewhat inelegant dual labelling of transitions, whereas [KV92] used single transition labels but had to impose some restrictions on when composition operators could be applied. In the current work the state labelling and transition labelling are completely independent of each other and no such special solutions are necessary. Furthermore, the model used here corresponds more closely to the model most usually used in the temporal logic community.

Chapter 2

Concurrent systems

2.1 Background

In this chapter we first describe in general terms the kinds of systems discussed in this work, then define the mathematical structures used in modelling them, and finally present some examples of such systems. These example systems will be returned to in the subsequent chapters of the work.

Probably the easiest way to describe concurrent systems in general is by contrasting them with traditional sequential ones. The task of an archetypal sequential program is to compute a function or a relation between the input and the output. Such a program can be seen as a single agent operating for a finite time on a given input and producing some output. Concurrent systems, in contrast, are characterised by multiple computing agents each of which may operate sequentially, but instead of computing a simple function, interacts intermittently with its environment. The computation of a concurrent system is not necessarily terminating, either.

A fundamental issue in which various approaches to concurrency differ is the method of interaction between different components of a system. Three common methods are *shared variables* and either *synchronous* or *asynchronous message-passing* [Pnu86, LL90]. The first two will be discussed here directly. Asynchronous message-passing can be reduced to synchronous message-passing by explicit modelling of channels, so the current work applies to that as well, albeit more indirectly.

We do not present formally any language for writing concurrent programs. Instead, in the examples such systems will be presented by using generic programming notation for the individual processes and by combining processes by composition operators that will be introduced in the following.

Concurrent systems are widely used in practice. Unfortunately, the methods that are conventionally applied in order to guarantee that a sequential program meets its requirements are often not generalisable to concurrent programs. It is also the case that concurrent systems are much more prone to intricate errors caused by unforeseen interferences than sequential ones are. Therefore, rigorous techniques for guaranteeing that a concurrent program meets its specification are essential in the design of concurrent systems.

The requirements that a system should meet are naturally closely tied to the intended purpose of that particular system. However, there are some general categories into which many required properties fall. The simplest classification of properties is into the so-called *safety* or *invariance* properties, which intuitively express that the system does not do anything bad, and to *liveness* properties, which express that in the long run the system does something useful. More involved classification schemes have been presented e.g. in [MP90] [MP91].

2.2 Modelling concurrent systems

In order to analyse a system rigorously, we first need to construct a mathematical model describing the relevant aspects of the system. An often used approach with sequential programs is modelling them as functions or relations between the input and the output values. The formal model of a concurrent system needs to reflect not only the initial and final configurations but the gradual progress of computation and the existence of interaction between the agents. In this work both the execution of the component processes of a system and the execution of a complete system are modelled by an *augmented labelled transition system* or *alts*.

Intuitively, an augmented labelled transition system is a directed graph modelling the execution of a system or a component of a system. The nodes of the graph correspond to momentary execution states of the system and the arcs to the execution of atomic actions or atomic commands of the system.

An execution state of a system has various observable characteristics, e.g. the current values of the program variables and the program counter, the messages that are currently transmitted along communication channels etc. In order to avoid unnecessarily restricting the kinds of systems that can be modelled, we do not specify what the observable characteristics actually are. Instead, we only require that they can be described by a set of truth-valued statements. It is easy to see that e.g. the value of an integer variable x can be fully described by the set of statements *the value of x is 0*, *the value of x is 1* etc.

Following this idea, when modelling the execution of a system, we associate with it a set of statements or *atomic propositions* which are relevant to describing the observable characteristics of the execution states of the system. With each individual execution state, i.e. each node of the alts-graph, we associate the set of relevant atomic propositions that happen to be true in that execution state.

The value of a boolean variable can be described by a single truth-valued proposition. In the following we adopt the convention that the values of a boolean variable x are described by a proposition having the same name x. The situation where the variable x has the value *true* is described by the proposition x being true in that situation, and vice versa. By this convention the distinction between a boolean variable and an atomic proposition can often be ignored and both terms used interchangeably.

To be slightly more formal, we assume the existence of a set AP of all atomic propositions or all possible statements relevant to the observable characteristics of the execution states of any system. We use the standard notations such as $\cap$, $\cup$, $\setminus$ etc. for various set operations, and write $P(X)$ for the powerset of a set X, i.e. the set of subsets of X.

Definition 2.2.1 We denote by AP a fixed, countably infinite set of *atomic propositions*. The subsets of AP are called *truth sets.* $\square$

When modelling a system, we associate with it some subset A of AP, the propositions relevant to that system.

The transitions of an alts-graph correspond to the execution of atomic program steps of the system the alts is modelling. The labels of the transitions specify aspects of interaction between a system or a component of a system and its environment. If the method of interaction is synchronisation, the transition labels are names on which components of a system can synchronise. If, on the other hand, the interaction takes place by references to shared variables, the transition labels are truth-valued conditions. However, for the basic definitions it does not matter what the exact nature of transition labels is, and we postpone a full treatment of them until later.

Definition 2.2.2 Let Σ be a countably infinite set of *non-empty transition labels*, not containing the unique *empty transition label* τ. We write Σ_τ for $\Sigma \cup \{\tau\}$, and call the elements of Σ_τ *transition labels.* $\square$

Definition 2.2.3 An *augmented labelled transition system (alts)* is a 6-tuple

$$L = (S, I, \Gamma, \Delta, A, V)$$

where

- S is a non-empty set, the elements of which are called *nodes of* L,
- $I \subseteq S$ is a non-empty set of *initial nodes*,
- $\Gamma \subseteq \Sigma$ is a set of non-empty transition labels,
- $\Delta \subseteq S \times (\Gamma \cup \{\tau\}) \times S$ is the *transition relation*, the elements of which are called *transitions of* L,
- $A \subseteq AP$ is a set of atomic propositions, and
- $V : S \to P(A)$ is a *truth-valuation* giving for each node the set of relevant atomic propositions true in it.

For an alts L, we use $\Sigma(L)$ to denote the set of non-empty transition labels of L, and $AP(L)$ to denote the set of atomic propositions of L, i.e. if L is as above, $\Sigma(L) = \Gamma$ and $AP(L) = A$. $\square$

We do not formally require that all the states of an alts are reachable from some initial state, as this would complicate some later definitions. However, for any practical purpose unreachable states are meaningless and will be routinely discarded in the examples.

Definition 2.2.4 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts. We say that L is *finite* iff the sets S, Γ and A are finite. If L is a finite alts, the *size of* L is $|L| = |S| + |\Delta|$. $\square$

If we omit the last two components, A and V, from the definition of an alts, we obtain the definition of a structure (S, I, Γ, Δ) commonly known in the process algebra literature as a *labelled transition system* or *lts*. The concept of an augmented labelled transition system extends in a natural way the concept of a standard lts by introducing a state labelling in addition to a transition labelling. Essentially the same concept occurs frequently in the literature under various names, e.g. as *parametrised labelled transition systems* [Arn94], *modal models* [Sti92] and as a subclass of *general structures* in [Eme90].

Usually the execution state of a program is uniquely defined by the value of the program counter and the values of all its variables. Consequently, when the execution of such a program is modelled by an alts, and the set A of atomic propositions of that alts is chosen so that it can fully characterise the value of the program counter and the values of all the variables, no two execution states, i.e. no two nodes of the alts, have the same set of atomic propositions true in them. In other words, the state-labelling function V is injective and the partial inverse V^{-1} to it well-defined. This means that having $s \in S$ and $V(s)$ as separate entities would be superfluous, and we might instead identify each s with $V(s)$ and take the set of nodes S simply to be a subset of $P(A)$. This idea, that a state of an execution graph is uniquely determined by the values of variables in that state, and that states can therefore be equated with valuations, would lead to model structures similar to that used by Lamport in [Lam85] and Shankar and Lam in [SL92].

However, although the execution state of a program is usually uniquely defined by the values of the program counter and the variables, we are not necessarily interested in all aspects of the execution state. For example, it is often the case that only the values of some variables are of interest, whereas those of some others will be irrelevant with respect to the properties of the system we are interested in. If we choose to abstract away from the irrelevant features and choose the set A of atomic propositions so that they are not recorded, it can well be the case that different execution states, i.e. nodes $s, s' \in S$ have the same set $V(s) = V(s')$ of atomic propositions true in them. In this case the distinction between the node s and the associated set $V(s)$ of true atomic propositions is no longer superfluous. Intuitively, a good way to understand the relation between s and $V(s)$ is to take s as the underlying, not directly observable execution state, and $V(s)$ as the observable or interesting characteristics of this state.

2.3 Definitions and notation

A single transition $(s, l, s') \in \Delta$ of an alts corresponds intuitively to the execution of a single atomic statement of the program the execution of which is being modelled. The results of executing such an atomic action are twofold: First, it may interact directly with the environment, and secondly, it may cause the execution to enter a new execution state. The first of these facets is reflected by the transition label l and the second by the execution state becoming s'. However, bearing in mind that the execution state s' is not directly observable, the latter effect corresponds to the observable characteristics of the execution state becoming $V(s')$. As this notion of the dual observable results of a transition will often be needed in the following, we give it a name of its own.

Definition 2.3.1 Let $l \in \Sigma_\tau$ and $A \subseteq AP$. The ordered pair (l, A) is called an *augmented transition label*. The set of all augmented transition labels is denoted by $\Sigma_{aug} = \Sigma_\tau \times P(AP)$. □

A single execution sequence of a system consists of a sequence of executions of atomic actions starting from an initial execution state. In an alts modelling the execution of a system this corresponds to a connected sequence of transitions starting from some initial node of the alts. As we noted above, the observable characteristics of a node s is the set $V(s)$, and the observable characteristics of a transition (s, l, s') are the transition label l and the change to $V(s')$ in the observable characteristics of the execution state. This means that the observable characteristics of an execution sequence of the system correspond to a set of propositions true in an initial execution state and a sequence of augmented transition labels. Such sequences will be frequently needed in the following and are given a separate name.

Definition 2.3.2 A finite sequence $A_0 l_1 A_1 \ldots l_n A_n$, where $n \geq 0$, $A_0 \subseteq AP$ and $(l_i, A_i) \in \Sigma_{aug}$, for all $0 < i \leq n$, is called a *finite behaviour*. The set of all finite behaviours is denoted by B_*. If a finite behaviour consists of a single element A, i.e. $n = 0$ above, we use the special symbol ε_A to denote it.

An infinite sequence $A_0 l_1 A_1 \ldots l_n A_n \ldots$, where $A_0 \subseteq AP$ and $(l_i, A_i) \in \Sigma_{aug}$, for all $i > 0$, is called an *infinite behaviour*. The set of all infinite behaviours is denoted by B_ω.

A sequence $A_0 l_1 A_1 \ldots$ is a *behaviour* iff it is either a finite or an infinite behaviour. The set of all behaviours is denoted by $B_\infty = B_* \cup B_\omega$. □

The term *behaviour* was adopted from [LT89]. Although used in a different technical framework there, the intuition behind the concept is the same. The notations B_*, B_ω and B_∞ are modelled after the standard constructs A^*, A^ω and A^∞ used for ω-regular expressions (see e.g. [Tho90, p. 136]). However, although the intuitions behind the notations are the same, technically we are dealing with different concepts. To remind of this, we write the $*$, ω and ∞ in B_*, B_ω and B_∞ as subscripts instead of superscripts. Let us introduce some concepts and notation related to behaviours.

Definition 2.3.3 For each behaviour $\rho \in B_\infty$, the *length of* ρ, denoted by $|\rho|$, is an element of $\mathbb{N} \cup \{\omega\}$, where ω is an element not in $\mathbb{N}$, defined by:

- if $\rho \in B_*$ is a finite behaviour, $\rho = A_0 l_1 A_1 \ldots l_n A_n$, then $|\rho| = n$.
- if $\rho \in B_\omega$ is an infinite behaviour, then $|\rho| = \omega$.

The standard ordering $<$ of $\mathbb{N}$ is extended to $\mathbb{N} \cup \{\omega\}$ by defining $n < \omega$ for all $n \in \mathbb{N}$. The standard addition operation $+$ of $\mathbb{N}$ is extended to $\mathbb{N} \cup \{\omega\}$ by defining $n + \omega = \omega + n = \omega$ for all $n \in \mathbb{N} \cup \{\omega\}$. □

Notice that $|\varepsilon_A| = 0$ for all $A \subseteq AP$.

Definition 2.3.4 Let $\rho = A_0 l_1 A_1 \ldots l_n A_n \ldots \in B_\infty$ be a behaviour.

- The *initial truth set of* ρ, denoted by ρ_{init}, is A_0.

- If ρ is a finite behaviour, the *final truth set of* ρ, denoted by ρ_{fin}, is $A_{|\rho|}$.
- For each $0 < i < |\rho| + 1$, the *i-th element of* ρ, denoted by ρ_i, is the augmented transition label $\rho_i = (l_i, A_i) \in \Sigma_{aug}$.
- For each $0 \leq i < |\rho| + 1$, the *i-suffix of* ρ, denoted by $\rho[i \ldots]$, is the behaviour $\rho[i \ldots] = A_i l_{i+1} A_{i+1} \ldots \in B_\infty$.
- For each $0 \leq i < |\rho|+1$, the *i-prefix of* ρ, denoted by $\rho[0 \ldots i]$, is the finite behaviour $\rho[0 \ldots i] = A_0 l_1 A_1 \ldots A_{i-1} l_{i-1} A_i \in B_*$.
- The *set of atomic propositions occurring in* ρ, denoted by $AP(\rho)$, is the set $AP(\rho) = \bigcup_{0 \leq i < |\rho|+1} A_i$.
- The *set of non-empty transition labels occurring in* ρ, denoted by $\Sigma(\rho)$, is the set $\Sigma(\rho) = \bigcup_{1 \leq i < |\rho|+1} \{l_i\} \setminus \{\tau\}$. □

Definition 2.3.5 Let $\rho \in B_*$ be a finite behaviour and $\rho' \in B_\infty$ a behaviour such that $\rho_{fin} = \rho'_{init}$. The *concatenation of* ρ *and* ρ', denoted by $\rho \cdot \rho'$, is the behaviour $\rho'' \in B_\infty$ defined by $|\rho''| = |\rho| + |\rho'|$, $\rho''_{init} = \rho_{init}$, for all $0 < i < |\rho| + 1$, $\rho''_i = \rho_i$, and for all $0 < i < |\rho'| + 1$, $\rho''_{i+|\rho|} = \rho'_i$.

Let $\rho \in B_\omega$ be an infinite behaviour and $\rho' \in B_\infty$ a behaviour. Then the concatenation of ρ and ρ' is defined by $\rho \cdot \rho' = \rho$.

Let $\rho \in B_\infty$ be a behaviour such that $\rho_{fin} = \rho_{init}$, and let $n \geq 0$. Then ρ^n is defined recursively by $\rho^0 = \rho_{init}$, and $\rho^{i+1} = \rho^i \cdot \rho$. □

Definition 2.3.6 Let $\rho, \rho' \in B_\infty$ be behaviours. We say that ρ *is a prefix of* ρ' and write $\rho \preceq \rho'$ iff there is a $\rho'' \in B_\infty$ such that $\rho \cdot \rho'' = \rho'$.

We say that ρ is a *proper prefix of* ρ' and write $\rho \prec \rho'$ iff $\rho \preceq \rho'$ and $|\rho| < |\rho'|$. □

Definition 2.3.7 Let $\rho = A_0 l_1 A_1 \ldots l_n A_n \ldots \in B_\infty$ be a behaviour and let f be a function $f : P(AP) \to P(AP)$. The result of applying f to ρ is defined as the behaviour $f(\rho) = f(A_0) l_1 f(A_1) \ldots l_n f(A_n) \ldots$, i.e. the behaviour $f(\rho) \in B_\infty$ for which $|f(\rho)| = |\rho|$, $f(\rho)_{init} = f(\rho_{init})$, and for all $0 < i < |\rho| + 1$, if $\rho_i = (l, A)$ then $f(\rho)_i = (l, f(A))$.

Similarly, let f be a function $f : \Sigma_\tau \to \Sigma_\tau$. The result of applying f to ρ is defined as the behaviour $f(\rho) = A_0 f(l_1) A_1 \ldots f(l_n) A_n \ldots$, i.e. the behaviour $f(\rho) \in B_\infty$ for which $|f(\rho)| = |\rho|$, $f(\rho)_{init} = \rho_{init}$, and for all $0 < i < |\rho|+1$, if $\rho_i = (l, A)$ then $f(\rho)_i = (f(l), A)$. □

In an alts a connected sequence of transitions which starts from an initial state corresponds to one possible partial execution sequence of the system the alts is modelling. The following concept of a behaviour of an alts captures the idea of the observable characteristics of such a sequence.

Definition 2.3.8 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts.

Let $\rho \in B_*$ be a finite behaviour, and let $s, s' \in S$. We say that *there is a path from* s *to* s' *with behaviour* ρ and write $s \xrightarrow{\rho} s'$ iff $\rho_{init} = V(s)$ and there is a sequence $s_0, s_1, \ldots, s_{|\rho|}$ such that $s = s_0$, $s' = s_{|\rho|}$ and for each $0 < i < |\rho| + 1$, $s_i \in S$ and if $\rho_i = (l_i, A_i)$, then $(s_{i-1}, l_i, s_i) \in \Delta$ and $V(s_i) = A_i$.

Let $\rho \in B_\infty$ be a behaviour, and let $s \in S$. We say that *there is a path from s with behaviour* ρ and write $s \xrightarrow{\rho}$ iff $\rho_{init} = V(s)$ and either ρ is finite and there is a sequence $s_0, s_1, \ldots, s_{|\rho|}$ or ρ is infinite and there is a sequence $s_0, s_1, \ldots$ such that $s = s_0$ and for each $0 < i < |\rho| + 1$, $s_i \in S$ and if $\rho_i = (l_i, A_i)$, then $(s_{i-1}, l_i, s_i) \in \Delta$ and $V(s_i) = A_i$. □

Notice that for all $A \subseteq AP$, $s \xrightarrow{\varepsilon_A} s$ iff $V(s) = A$.

Definition 2.3.9 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts.

A behaviour $\rho \in B_\infty$ is a *behaviour of* L iff there is an $s \in I$ such that $s \xrightarrow{\rho}$.

The set of all behaviours of L is denoted by $beh(L)$. The set of all *finite behaviours of* L, denoted by $fbeh(L)$, is defined by $fbeh(L) = beh(L) \cap B_*$. The set of all *infinite behaviours of* L, denoted by $infbeh(L)$, is defined by $infbeh(L) = beh(L) \cap B_\omega$. □

A connected sequence of transitions which starts from an initial node and is either infinite or leads to a node from which no transition can be taken corresponds to one possible complete execution sequence of the system the alts is modelling. This intuition of the observable characteristics of such a complete execution sequence is captured by the concept of a maximal behaviour of an alts.

Definition 2.3.10 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts.

A finite behaviour $\rho \in B_*$ is a *deadlock behaviour of* L iff there are $s \in I$ and $s' \in S$ such that $s \xrightarrow{\rho} s'$ and for all $l \in \Sigma_\tau$ and all $s'' \in S$, $(s', l, s'') \notin \Delta$, i.e. there are no outgoing transitions from s'. The set of all deadlock behaviours of L is denoted by $dlbeh(L)$.

A behaviour $\rho \in B_\infty$ is a *maximal behaviour of* L iff it is either a deadlock behaviour of L or an infinite behaviour of L. The set of all maximal behaviours of L is denoted by $maxbeh(L) = dlbeh(L) \cup infbeh(L)$. □

2.4 Composition operators

Now we turn to methods that can be used to build larger systems from components. These are represented by operators that can be used to combine two altss into a new one or to manipulate a single alts. Only a fairly limited set of operators is provided. However, the selected operators are fundamental and well-established, and allow us to illustrate both the process of composition and that of abstraction.

As already mentioned, we will discuss two scenarios for interaction between a system or a component of a system and its environment.

In the first case the interaction takes place by synchronising some actions so that a synchronised action can be executed only if some action with the same synchronisation name is also executed simultaneously in the environment. The transition labels of an alts are here just considered to be names on which synchronisation can take place. The syntactic operators catering for this method of interaction are:

- Parallel composition $L_1 \;|[\Pi]|\; L_2$, which takes two altss, L_1 and L_2, and combines them in parallel so that transitions with labels in the set Π are synchronised, and all other transitions executed asynchronously.
- Synchronisation name hiding **hide** Π **in** L, which takes an alts L and makes the transition labels in the set Π unavailable for further synchronisation by replacing them with the empty transition label τ.
- Synchronisation name renaming $L[f]$, which takes an alts L and changes each transition label l in it to $f(l)$.
- Proposition hiding or encapsulation **encaps** B **in** L, which takes an alts L and removes from the state labelling the information concerning propositions in the set B.

The parallel composition, hiding and renaming have close counterparts in most process algebras. The particular variants adopted here have been closely influenced by the operators in LOTOS [BB89, ISO89]. Intuitively, the encapsulation operator corresponds to scoping mechanisms available in many programming languages. It allows us to make some characteristics of the local state invisible to the outside world.

In the second scenario of interaction, components of a system interact by making reference to characteristics of the local state of other components. This relies on interpreting transition labels as truth-valued expressions, which can hold or fail to hold in a given environment. A transition can be taken in a given environment only if the expression labelling it evaluates to true in that environment. The approach can be applied to model interaction by shared variables which are restricted in the sense that each variable can be modified only by a single component of a system. Such variables have also been called *read-shared*, *local shared* [LL90] or *distributed* [Pnu85]. The syntactic operators catering for this method of interaction are:

- Parallel composition $L_1 \parallel_v L_2$, which takes two altss, L_1 and L_2, and combines them in parallel, evaluating the transition conditions to the extent that they can be evaluated on the basis of the local state characteristics of the other component, and disallowing transitions with conditions evaluating to false. Again, parallel execution is represented by interleaving atomic actions. The subscript v is intended to refer to interaction by variables.
- Variable hiding or encapsulation **encaps** B **in** L, the same as above.

Occasionally we abuse our notation slightly, and use the symbol $\parallel$ without a subscript to refer to both forms of parallel composition above.

It may be wondered why the interaction by shared variables has been presented as its own case. After all, it is well-known that shared variables can easily be simulated in the approach where interaction is by synchronisation, at the cost of introducing a new parallel component to the system for each shared variable. The reasons for this are twofold. First, in our judgement it is interesting to see how the same theory can be applied with very few modifications to cater directly for both the interaction by synchronisation and the interaction by shared variables scenario, two at the face value fairly dissimilar approaches. Secondly, the shared variables model is probably the most widely used in the temporal logic literature, so it is preferable to have results concerning it directly rather than via some encoding.

If we would like to have a model where components of a system can interact with some components by synchronisation and with some other components by shared variables, the two scenarios could easily be mixed in a single model. In this case the only necessary change to the work here would be dividing the set of transition labels into two disjoint parts, one for synchronisation labels and one for conditions, and taking care that in the parallel composition with synchronisation $|[\ldots]|$ the condition labels are not used in any way, and vice versa, that in the parallel composition with shared variables $\|_v$ the synchronisation labels are not used.

The most important operator omitted in both the scenarios is the sequential composition. The reason for this is that in this work we are primarily interested in systems which consist of a group of sequential processes running concurrently, as e.g. in [MP91]. In other words, we expect concurrency to take place only at the top level of a system. Since we would like to concentrate on the problems of verification caused by concurrency, we do not include in the syntactic machinery the means to build the basic sequential components of a system. Rather, we expect them to be given directly as altss and start from that point. Some forms of sequential composition can be incorporated into the framework of this work without any modifications. This issue will be discussed further in Section 4.6.1.

The various syntactic operators are defined formally as follows.

Definition 2.4.1 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be augmented labelled transition systems such that $A_1 \cap A_2 = \emptyset$, and let $\Pi \subseteq \Sigma$ be a set of non-empty transition labels.

Then $L_1 \,|[\Pi]|\, L_2$ is the alts $L = (S, I, \Gamma, \Delta, A, V)$ such that

$$\begin{array}{rcl}
S & = & S_1 \times S_2 \\
I & = & I_1 \times I_2 \\
\Gamma & = & \Gamma_1 \cup \Gamma_2 \\
\Delta & = & \{((s,t), l, (s',t')) \mid l \in \Pi \text{ and } (s,l,s') \in \Delta_1 \text{ and } (t,l,t') \in \Delta_2\} \cup \\
 & & \{((s,t), l, (s',t)) \mid l \in \Sigma_\tau \setminus \Pi \text{ and } (s,l,s') \in \Delta_1 \text{ and } t \in S_2\} \cup \\
 & & \{((s,t), l, (s,t')) \mid l \in \Sigma_\tau \setminus \Pi \text{ and } (t,l,t') \in \Delta_2 \text{ and } s \in S_1\} \\
A & = & A_1 \cup A_2 \\
V & : & S \to P(A); (s_1, s_2) \mapsto V_1(s_1) \cup V_2(s_2)
\end{array}$$

□

Above, for all practical purposes we can always remove the part of the resulting alts-graph which is not reachable from any initial node. This will be done automatically in all the examples in this work.

Definition 2.4.2 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts and $\Pi \subseteq \Sigma$ a set of non-empty transition labels. Then **hide** Π **in** L is the alts $L' = (S', I', \Gamma', \Delta', A', V')$ defined by

$$\begin{array}{rcl} S' & = & S \\ I' & = & I \\ \Gamma' & = & \Gamma \setminus \Pi \\ \Delta' & = & \{(s, l, s') \mid l \notin \Pi \text{ and } (s, l, s') \in \Delta\} \cup \\ & & \{(s, \tau, s') \mid \text{ there is an } \; l \in \Pi \text{ such that } \; (s, l, s') \in \Delta\} \\ A' & = & A \\ V' & = & V \end{array}$$

□

Definition 2.4.3 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts and $f : \Sigma \to \Sigma$ a function mapping non-empty transition labels to non-empty transition labels. Then $L[f]$ is the alts $L' = (S', I', \Gamma', \Delta', A', V')$ defined by

$$\begin{array}{rcl} S' & = & S \\ I' & = & I \\ \Gamma' & = & \{l' \in \Sigma \mid \exists l \in \Gamma : l' = f(l)\} \\ \Delta' & = & \{(s, \tau, s') \mid (s, \tau, s') \in \Delta\} \cup \\ & & \{(s, l, s') \mid \text{ there is an } \; l' \in \Sigma \text{ such that } \; (s, l', s') \in \Delta \text{ and } f(l') = l\} \\ A' & = & A \\ V' & = & V \end{array}$$

□

Definition 2.4.4 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts and let $B \subseteq AP$ a set of atomic propositions. Then **encaps** B **in** L is the alts $L' = (S', I', \Gamma', \Delta', A', V')$ defined by

$$\begin{array}{rcl} S' & = & S \\ I' & = & I \\ \Gamma' & = & \Gamma \\ \Delta' & = & \Delta \\ A' & = & A \setminus B \\ V' & : & S' \to P(A'); s \mapsto V(s) \setminus B \end{array}$$

□

To define the meaning of the parallel composition operator $\|_v$, we need to specify first how transition labels can be seen as truth-valued expressions that refer to some characteristics of the momentary state of the environment. This is achieved by assuming that the transition labels have a particular internal structure; that they are condition expressions as defined in the following. The particular way of expressing transition conditions is adopted from [Dam92], and is related to the transition predicates of [AS89].

Definition 2.4.5 A *condition expression* is an ordered pair (B^+, B^-), where

- $B^+ \subseteq AP$,
- $B^- \subseteq AP$, and
- $B^+ \cap B^- = \emptyset$.

The unique *empty condition expression* is the pair $(\emptyset, \emptyset)$. All other condition expressions are *non-empty*. We equate the empty condition expression $(\emptyset, \emptyset)$ with the empty transition label τ.

The set of all non-empty condition expressions is denoted by Σ_{cond}.

Furthermore, if $A \subseteq AP$, we denote by $\Sigma_{cond}(A)$ the set of all non-empty condition expressions (B^+, B^-) for which $B^+ \cup B^- \subseteq A$. □

Definition 2.4.6 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts. We say that L is *condition-labelled* iff $\Gamma \subseteq \Sigma_{cond}(AP \setminus A)$, i.e. iff all transition labels occurring in L are condition expressions containing only propositions external to L. □

A condition expression is a pair of sets of atomic propositions (B^+, B^-), intuitively specifying that the propositions in B^+ should be true, and the propositions in B^- false. The third requirement in the above definition says that a condition expression cannot be self-contradictory, i.e. specifying that some proposition should be both true and false. The empty condition expression $(\emptyset, \emptyset)$ requires neither that any proposition is true, nor that any proposition is false. Therefore, the possibility of executing a transition labelled with the empty condition expression $(\emptyset, \emptyset)$ cannot be affected by the environment in any way, and the empty condition expression can be equated with the empty transition label τ.

Any condition that can be represented as a conjunction of atomic propositions and their negations can clearly be represented directly as a condition expression, as by the intended interpretation of condition expressions the formula $\bigwedge_{i \in I} a_i \wedge \bigwedge_{j \in J} \neg b_j$ corresponds to the condition expression $(\bigcup_{i \in I}\{a_i\}, \bigcup_{j \in J}\{b_j\})$. We use this correspondence to write transition conditions in examples as conjunctions of atomic propositions and their negations.

Suppose then that we have an arbitrary boolean formula C and that we would like to express the condition *if C holds in the current environment, then the transition $s \longrightarrow s'$ is possible.* Without loss of generality we can assume that C is in disjunctive normal form, i.e. a disjunction $\bigvee_{k \in K} C_k$ of conjunctions C_k of atomic propositions and their negations, as any boolean formula can be mechanically transformed into such a form. But a condition in disjunctive normal form can be easily represented by introducing a separate transition $s \xrightarrow{C_k} s'$ for each distinct disjunct C_k. If some such transition is possible in a given environment, then some C_k holds which means that C holds as well, and vice versa, if C holds then some C_k must hold and some transition $s \xrightarrow{C_k} s'$ must be possible. This means that we have the means to represent any boolean transition condition in an alts. Furthermore, since in the general case nothing prevents these disjunctions and conjunctions from being infinitary, we have a powerful machinery for representing almost any conceivable transition condition in an alts.

Notice that the condition expressions labelling the transitions of an alts are not allowed to contain atomic propositions of the alts itself, since we require that $\Sigma(L) \subseteq \Sigma_{cond}(AP \setminus A)$

for any condition-labelled alts L. Intuitively, this reflects the fact that the transition conditions are supposed to express a requirement that the *environment* must fulfil, if the transition is to be taken. If we have a condition that makes reference to characteristics of both the environment and the local state of the system, this is easy to transform into a condition for the environment only, since in any local state of a system we already know fully what the local characteristics of it are.

To see whether a particular condition holds in a given momentary state of an environment, we need a notion of evaluating a condition in an environment. The observable features of the momentary states of the environment of an alts $L = (S, I, \Gamma, \Delta, A, V)$ are described by the atomic propositions in the set $AP \setminus A$. For any momentary state of the environment of L, there is a particular subset A^+ of propositions in $AP \setminus A$, those that are true at that state, characterising the observable features of that state of the environment. A condition (B^+, B^-) is satisfied in a momentary state of the environment corresponding to A^+ iff every proposition in B^+ holds, i.e. $B^+ \subseteq A^+$, and every proposition in B^- fails to hold, i.e. $B^- \cap A^+ = \emptyset$.

When we are defining the effect of the parallel composition operator $\|_v$, we need to combine the above intuition of evaluating a condition in an environment with the fact that in $L_1 \|_v L_2$, the node labels of L_2 provide only partial information about the environment of L_1 and vice versa. A momentary state of the environment of L_1 in $L_1 \|_v L_2$ consists of a momentary state of L_2 and a momentary state of the rest of the environment. The observable features of the former are described by propositions in A_2 and those of the latter by propositions in $AP \setminus (A_1 \cup A_2)$. Suppose that we know the momentary state s of L_2. The observable features of s are characterised by the set $A_2^+ = V_2(s) \subseteq A_2$. Write A_2^- for $A_2 \setminus A_2^+$, i.e. for the set of propositions describing observable features of L_2 that happen to be false at s. Suppose then that (B_1^+, B_1^-) is a condition expression labelling a transition of L_1. Given that we know the momentary state s of L_2, there are two possible situations.

- It can be the case that $B_1^+ \cap A_2^- \neq \emptyset$ or $B_1^- \cap A_2^+ \neq \emptyset$, i.e. that either there is a proposition that B_1^+ requires to be true but that is a proposition of L_2 and false in s, or there is a proposition that B_1^- requires to be false but that is a proposition of L_2 and true in s. In this case we know that the condition cannot be satisfied, no matter what the state of the rest of the environment is.
- This is not the case, i.e. $B_1^+ \cap A_2 \subseteq A_2^+$ and $B_1^- \cap A_2 \subseteq A_2^-$. In this case all the propositions of L_2 that are required to be true by the condition expression are actually true in s, and all the propositions of L_2 that are required to be false by it are actually false in s. Consequently, we know that the condition is satisfied iff those propositions in B_1^+ that are not propositions of L_2 , i.e. the propositions in $B_1^+ \setminus A_2$, are true in the current state of the rest of the environment, and the propositions in $B_1^- \setminus A_2$ similarly false.
  A particular subcase here is when $B_1^+ \setminus A_2 = B_1^- \setminus A_2 = \emptyset$. Then we know that the condition expression will be satisfied, no matter what the state of the rest of the environment is.

This intuition is captured by the following definitions.

Definition 2.4.7 Let $A, A^+ \subseteq AP$ be sets of atomic propositions such that $A^+ \subseteq A$, and let $\perp$ be a new element not in $\Sigma_{cond} \cup \{\tau\}$. The *partial evaluation function* $eval(A, A^+)$ is the mapping

$$eval(A, A^+) : \Sigma_{cond} \cup \{\tau\} \rightarrow \Sigma_{cond}(AP \setminus A) \cup \{\tau, \perp\}$$

defined by the following rules:

- $eval(A, A^+)((B^+, B^-)) = \perp$, if $B^+ \cap (A \setminus A^+) \neq \emptyset$ or $B^- \cap A^+ \neq \emptyset$
- $eval(A, A^+)((B^+, B^-)) = (B^+ \setminus A, B^- \setminus A)$ otherwise. □

Notice that $eval(A, A^+)(\tau) = eval(A, A^+)((\emptyset, \emptyset)) = (\emptyset, \emptyset) = \tau$ for all $A^+ \subseteq A \subseteq AP$, reflecting the vacuousness of the empty condition expression.

Definition 2.4.8 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be condition-labelled altss such that $A_1 \cap A_2 = \emptyset$.

Then $L_1 \parallel_v L_2$ is the alts $L = (S, I, \Gamma, \Delta, A, V)$ such that

$$\begin{array}{rcl} S &=& S_1 \times S_2 \\ I &=& I_1 \times I_2 \\ \Gamma &=& \{l \in \Sigma_{cond} \mid \exists l_1 \in \Gamma_1 \text{ and } B_2 \subseteq A_2 : l = eval(A_2, B_2)(l_1)\} \cup \\ && \{l \in \Sigma_{cond} \mid \exists l_2 \in \Gamma_2 \text{ and } B_1 \subseteq A_1 : l = eval(A_1, B_1)(l_2)\} \\ \Delta &=& \{((s,t), l, (s', t)) \mid \exists l' : (s, l', s') \in \Delta_1 \ \wedge \ eval(A_2, V_2(t))(l') = l \neq \perp\} \cup \\ && \{((s,t), l, (s, t')) \mid \exists l' : (t, l', t') \in \Delta_2 \ \wedge \ eval(A_1, V_1(s))(l') = l \neq \perp\} \\ A &=& A_1 \cup A_2 \\ V &:& S \rightarrow P(A); (s_1, s_2) \mapsto V_1(s_1) \cup V_2(s_2) \end{array}$$

□

Let us see next that $\parallel_v$ is associative up to isomorphism of altss. This allows us to omit parenthesis from expressions like $(L_1 \parallel_v L_2) \parallel_v L_3$ and write just $L_1 \parallel_v L_2 \parallel_v L_3$.

Definition 2.4.9 Two altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ are *isomorphic* iff $\Gamma = \Gamma'$, $A = A'$ and there is a bijection $f : S \rightarrow S'$ such that

- for all $s \in S$, $s \in I$ iff $f(s) \in I'$,
- for all $s, s' \in S$ and all $l \in \Sigma_\tau$, $(s, l, s') \in \Delta$ iff $(f(s), l, f(s')) \in \Delta'$, and
- for all $s \in S$, $V(s) = V'(f(s))$. □

Proposition 2.4.10 Let L_1, L_2 and L_3 be altss. The alts $L_1 \parallel_v (L_2 \parallel_v L_3)$ is defined iff $(L_1 \parallel_v L_2) \parallel_v L_3$ is defined, and if both are defined, they are isomorphic.

Proof: Notice first that both $L_1 \parallel_v (L_2 \parallel_v L_3)$ and $(L_1 \parallel_v L_2) \parallel_v L_3$ are defined iff $AP(L_1) \cap AP(L_2) = AP(L_1) \cap AP(L_3) = AP(L_2) \cap AP(L_3) = \emptyset$. Define $L = L_1 \parallel_v (L_2 \parallel_v L_3)$ and $L' = (L_1 \parallel_v L_2) \parallel_v L_3$, and write $L = (S, I, \Gamma, \Delta, A, V)$, $L' = (S', I', \Gamma', \Delta', A', V')$ and $L_i = (S_i, I_i, \Gamma_i, \Delta_i, A_i, V_i)$ for $i = 1, 2, 3$. From the definition of $\parallel_v$, we have $S = S_1 \times (S_2 \times S_3)$, $I = I_1 \times (I_2 \times I_3)$, $S' = (S_1 \times S_2) \times S_3$ and $I' = (I_1 \times I_2) \times I_3$.

Define a bijection $f : S \to S'$ by $f((s_1,(s_2,s_3))) = ((s_1,s_2),s_3)$. For every $s \in S$, $s \in I$ iff $f(s) \in I'$, since $(s_1,(s_2,s_3)) \in I$ iff $((s_1,s_2),s_3) \in I'$, and $V(s) = V'(f(s))$, since $V((s_1,(s_2,s_3))) = V_1(s_1) \cup V_2(s_2) \cup V_3(s_3) = V'(((s_1,s_2),s_3))$. For notational convenience, extend the $eval(\cdot,\cdot)$ function for argument value $\perp$ by: $eval(A,A^+)(\perp) = \perp$ for all A, A^+.

Let us take any $s, s' \in S$ and $l \in \Sigma_\tau$, and write $s = (s_1,(s_2,s_3))$, $s' = (s'_1,(s'_2,s'_3))$. We want to show that $(s,l,s') \in \Delta$ iff $(f(s),l,f(s')) \in \Delta'$. Notice that for all $i,j = 1,2,3$ and all $l' \in \Sigma_\tau$, if $i \neq j$ then

$$\begin{aligned} eval(A_i,V_i(s_i))(eval(A_j,V_j(s_j))(l')) &= eval(A_i \cup A_j, V_i(s_i) \cup V_j(s_j))(l') \\ &= eval(A_j,V_j(s_j))(eval(A_i,V_i(s_i))(l')) \end{aligned}$$

This means that $(s,l,s') = ((s_1,(s_2,s_3)),l,(s'_1,(s'_2,s'_3))) \in \Delta$ iff one of the following three conditions holds iff $(f(s),l,f(s')) = (((s_1,s_2),s_3),l,((s'_1,s'_2),s'_3)) \in \Delta'$.

- $s_1 = s'_1$, $s_2 = s'_2$ and there is an l' such that $(s_3,l',s'_3) \in \Delta_3$ and $eval(A_1 \cup A_2, V_1(s_1) \cup V_2(s_2))(l') = l \neq \perp$,
- $s_1 = s'_1$, $s_3 = s'_3$ and there is an l' such that $(s_2,l',s'_2) \in \Delta_2$ and $eval(A_1 \cup A_3, V_1(s_1) \cup V_3(s_3))(l') = l \neq \perp$, or
- $s_2 = s'_2$, $s_3 = s'_3$ and there is an l' such that $(s_1,l',s'_1) \in \Delta_1$ and $eval(A_2 \cup A_3, V_2(s_2) \cup V_3(s_3))(l') = l \neq \perp$. □

Often two altss are combined by synchronising them with respect to all their common non-empty synchronisation labels. Let us introduce abbreviated notation for this case. The subscript s in the following notation is intended to remind of the interaction by synchronisation.

Definition 2.4.11 The expression $L_1 \parallel_s L_2$ is an abbreviation for $L_1 |[\Sigma(L_1) \cap \Sigma(L_2)]| L_2$. □

Notice that the operator $\parallel_s$ here is slightly different from the $\parallel$-operator of LOTOS; the former synchronises systems with respect to all their common non-empty synchronisation labels, and the latter with respect to all non-empty synchronisation labels.

Neither the parallel composition operator $|[\Pi]|$, nor the derived operator $\parallel_s$ is associative in general. However, in the commonly occurring restricted case described by the following proposition, $\parallel_s$ is associative.

Proposition 2.4.12 Let L_1, L_2 and L_3 be altss such that $\Sigma(L_1) \cap \Sigma(L_3) = \emptyset$. Then the alts $L_1 \parallel_s (L_2 \parallel_s L_3)$ is defined iff $(L_1 \parallel_s L_2) \parallel_s L_3$ is defined, and if both are defined, they are isomorphic.

Proof: The claim concerning the definedness of the two expressions, and the bijection f witnessing their isomorphism are defined exactly as in the proof of 2.4.10. Let us also use the notation of that proof.

Take any $s, s' \in S$ and $l \in \Sigma_\tau$, and write $s = (s_1,(s_2,s_3))$, $s' = (s'_1,(s'_2,s'_3))$. We want to show that $(s,l,s') \in \Delta$ iff $(f(s),l,f(s')) \in \Delta'$. This holds, since $\Sigma(L_1) \cap \Sigma(L_3) = \emptyset$ implies that $(s,l,s') = ((s_1,(s_2,s_3)),l,(s'_1,(s'_2,s'_3))) \in \Delta$ iff one of the following conditions holds iff $(f(s),l,f(s')) = (((s_1,s_2),s_3),l,((s'_1,s'_2),s'_3)) \in \Delta'$.

- $l \in \Sigma(L_1) \cup \{\tau\}$, $l \notin \Sigma(L_2)$, $l \notin \Sigma(L_3)$, $(s_1, l, s_1') \in \Delta_1$, $s_2 = s_2'$ and $s_3 = s_3'$
- $l \notin \Sigma(L_1)$, $l \in \Sigma(L_2) \cup \{\tau\}$, $l \notin \Sigma(L_3)$, $s_1 = s_1'$, $(s_2, l, s_2') \in \Delta_2$ and $s_3 = s_3'$
- $l \notin \Sigma(L_1)$, $l \notin \Sigma(L_2)$, $l \in \Sigma(L_3) \cup \{\tau\}$, $s_1 = s_1'$, $s_2 = s_2'$ and $(s_3, l, s_3') \in \Delta_3$
- $l \in \Sigma(L_1)$, $l \in \Sigma(L_2)$, $l \notin \Sigma(L_3)$, $(s_1, l, s_1') \in \Delta_1$, $(s_2, l, s_2') \in \Delta_2$ and $s_3 = s_3'$
- $l \notin \Sigma(L_1)$, $l \in \Sigma(L_2)$, $l \in \Sigma(L_3)$, $s_1 = s_1'$, $(s_2, l, s_2') \in \Delta_2$ and $(s_3, l, s_3') \in \Delta_3$ □

Finally, let us define some useful syntactic abbreviations for commonly used forms of parallel composition, hiding, renaming and encapsulation. Some of the abbreviations rely on the ability to treat transition and proposition names as strings and to subject them to standard operations on strings, such as concatenation $(\cdot)$ or testing for prefixes.

Definition 2.4.13 Let L be an alts. Let us define

- **hide** $l_1, \ldots, l_n$ **in** $L =$ **hide** $\{l_1, \ldots, l_n\}$ **in** L
- **hide** $l_1*, \ldots, l_n*$ **in** $L =$ **hide** Π **in** L,
  where $\Pi = \{l \in \Sigma \mid \text{there is an } i \text{ such that } l_i \text{ is a prefix of } l\}$
- $L[l_1'/l_1, \ldots, l_n'/l_n] = L[f]$,
  where $f(l) = l$, if $l \notin \{l_1, \ldots, l_n\}$, and $f(l_i) = l_i'$ for all l_i.
- $L[l_1'*/l_1*, \ldots, l_n'*/l_n*] = L[f]$,
  where
  $$f(l) = \begin{cases} l_i' \cdot x & \text{if there are } i \text{ and } x \text{ such that } l = l_i \cdot x \\ l & \text{otherwise} \end{cases}$$
- **encaps** $l_1, \ldots, l_n$ **in** $L =$ **encaps** $\{l_1, \ldots, l_n\}$ **in** L
- **encaps** $a_1*, \ldots, a_n*$ **in** $L =$ **encaps** B **in** L,
  where $B = \{a \in AP \mid \text{there is an } i \text{ such that } a_i \text{ is a prefix of } a\}$

Let L_1 and L_2 be altss. Then

- $L_1 \mathrel{|[} l_1*, \ldots, l_n* \mathrel{]|} L_2 = L_1 \mathrel{|[} \Pi \mathrel{]|} L_2$,
  where $\Pi = \{l \in \Sigma \mid \text{there is an } i \text{ such that } l_i \text{ is a prefix of } l\}$ □

2.5 Examples

In this section we look at two concurrent systems and the way they can be modelled by altss. These examples will be returned to later to illustrate practical applications of the theory.

2.5.1 Mutual exclusion

A common problem to be solved in concurrent systems is mutual exclusion: Several processes are competing for a limited resource such as a printer or some other device, which can be used by only one process at a time. In order to use such a resource a

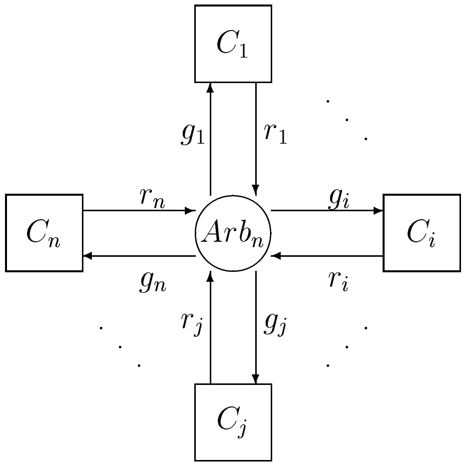


Figure 2.1: Mutual exclusion: Interaction structure of the system

Process Arb_n:
loop forever
    for $i := 1$ to n
        if r_i then
            $g_i := \top$
            wait until $\neg r_i$
            $g_i := \bot$
        end if
    end for
end loop

Process C_i:
loop forever
    /* Noncritical */
    $\vdots$
    $r_i := \top$
    wait until g_i
    $c_i := \top$
    /* Critical section begins */
    $\vdots$
    /* Critical section ends */
    $c_i, r_i := \bot$
    wait until $\neg g_i$
    /* Noncritical */
    $\vdots$
end loop

Figure 2.2: Mutual exclusion: Processes

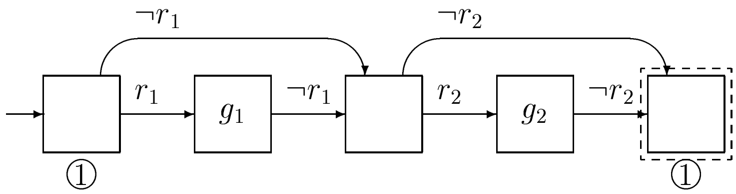


Figure 2.3: Mutual exclusion: Arbiter alts Arb_2

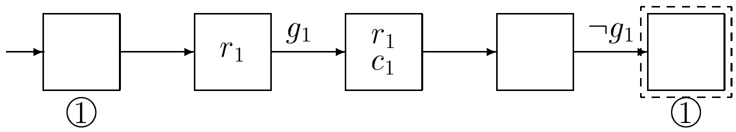


Figure 2.4: Mutual exclusion: Customer alts C_1

process has to negotiate with other processes according to some fixed protocol so that no more than one process uses the resource at any one time.

In Figure 2.1 a shared variable solution to the mutual exclusion problem is outlined: The customer processes $C_1, \ldots, C_n$ all ask permission to use the resource from an arbiter process Arb_n. A customer C_i signals its wish to use the resource by turning a flag r_i to true. When the arbiter decides that C_i is allowed to use the resource, it signals its decision to C_i by turning a flag g_i to true. After C_i has finished using the resource, it turns r_i back to false and waits until the arbiter has noticed this and turned g_i to false. The pseudo-code for the customers and the arbiter is in Figure 2.2.

Properties that the system is required to have are:

- No two processes use the resource at the same time.
- If a process requests the resource, its request is eventually granted.
- If customer process C_i has requested access to the resource, no other customer process is granted access to the resource more than once before the request of C_i has been granted. This property is known as *one-bounded overtaking*.

Examples of the augmented labelled transition systems modelling the mutual exclusion system with $n = 2$ are presented in Figures 2.3 and 2.4. The atomic propositions corresponding to the boolean variables owned by the arbiter alts are g_1 and g_2, and the atomic propositions of each C_i are r_i expressing that C_i has requested the resource and c_i expressing that C_i is using the resource.

The graphs contain some conventions that will be used throughout this work:

- Nodes are represented by boxes and transitions by arrows between them.
- The elements of the set of atomic propositions labelling a node are listed inside the box representing the node. In particular, if a node is labelled by the empty set, then the box representing the node is empty.
- Transition labels are written next to the transition arrows. However, empty transition labels τ are omitted.

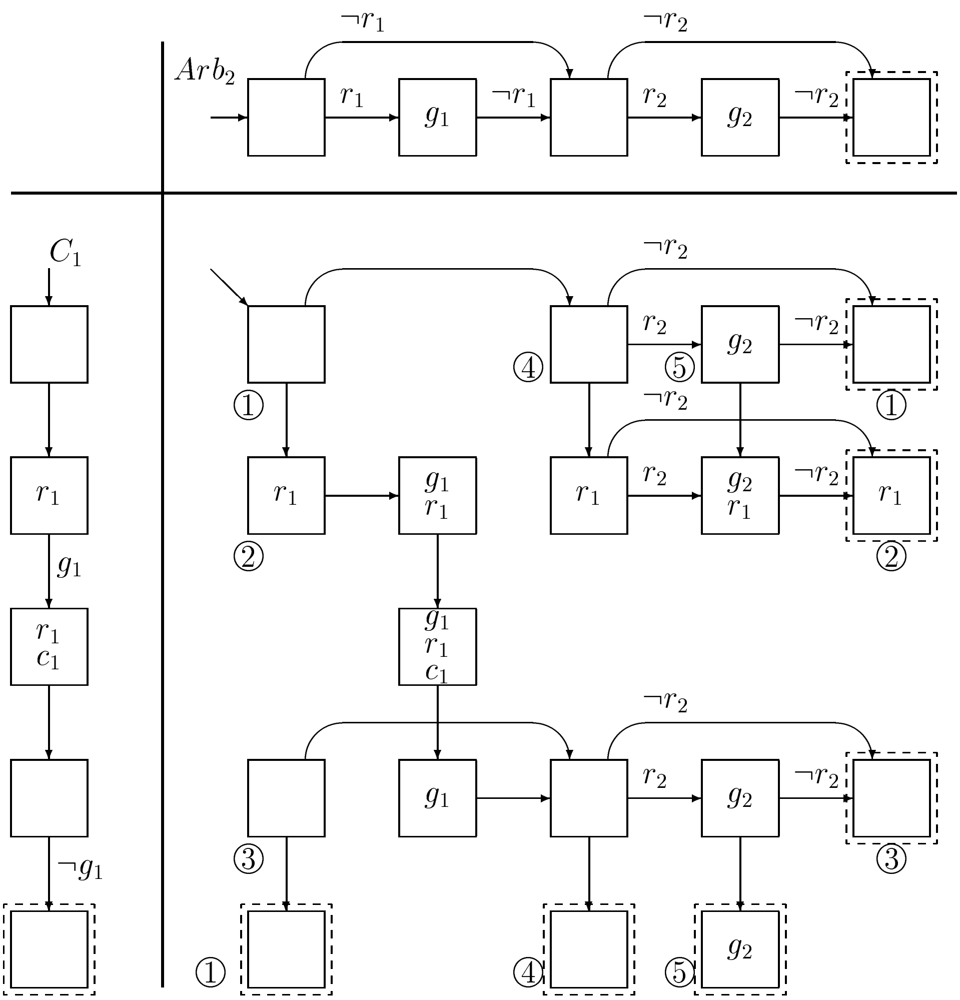


Figure 2.5: Mutual exclusion: Alts $Arb_2 \parallel_v C_1$

- Initial nodes are marked by short unlabelled arrows that do not originate from any node.
- A single node can have multiple occurrences in the picture for reasons of drawing topology. In this case each occurrence will have a circled identification number next to it, the same number for all the occurrences of the same node, and all but one occurrence of the node will have dashed boxes surrounding them.

For an n-customer system, the complete system is modelled by the alts

$$L = Arb_n \parallel_v C_1 \parallel_v C_2 \parallel_v \ldots \parallel_v C_n$$

The alts corresponding to $Arb_2 \parallel_v C_1$ is presented in Figure 2.5. The alts corresponding to the complete system for $n = 2$, $Arb_n \parallel_v C_1 \parallel_v C_2$, has 36 nodes and does not unfortunately lend itself into a visually pleasing presentation. From the graph it is relatively easy to see that the system manages to guarantee mutual exclusion, since no node is labelled with a set containing both c_1 and c_2. However, deciding whether the other two properties hold is more difficult, if one just examines the graph informally.

Specifying the behaviour of a mutual exclusion protocol like the one presented here is examined in detail in [MP91].

2.5.2 Sliding window protocol

The second verification example considered in this work comes from the realm of communication protocols. The particular protocol considered is a unidirectional version of the so-called *sliding window* or *SW-protocol* [Ste76], which forms the basis of the data transfer function of the HDLC (high level data link control) protocol [ISO79a, ISO79b, ISO80]. Variants of the protocol have been studied extensively from the specification and verification point of view (see Section 7.3).

2.5.2.1 Protocol

The aim of the protocol is to provide reliable transmission of data over an unreliable communication medium. The communication structure of the system is described in Figure 2.6.

The SW-protocol itself consists of four components. These are:

- The *sender*, which receives data items from an external *data source*, sends messages consisting of a sequence number and a data item to the channel, receives acknowledgement messages from the acknowledgement channel, and receives timeout signals from an unspecified external source.
- The *channel*, which receives messages consisting of a sequence number and a data item from the sender and transmits them to the receiver. The channel may distort or lose messages but may not change their order.
- The *acknowledgement channel*, which receives acknowledgement messages consisting of a sequence number from the receiver and transmits them to the sender. The

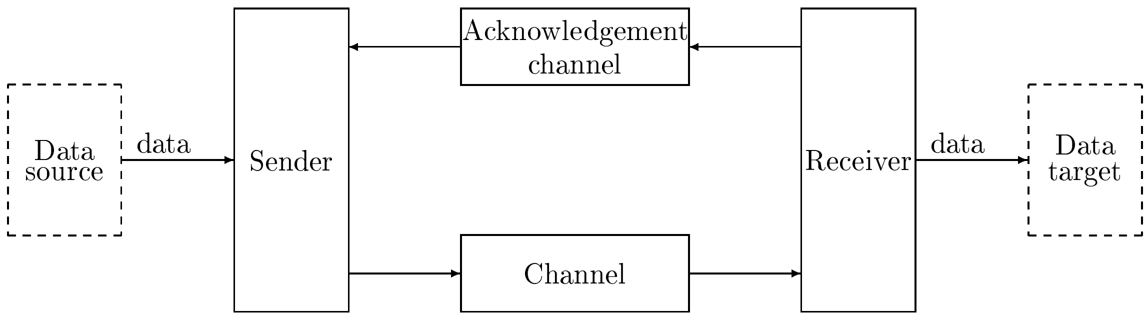


Figure 2.6: SW-protocol: Structure of the system

acknowledgement channel may distort or lose messages but may not change their order.

- The *receiver*, which reads messages from the channel, forwards correctly received data items to an external *data target*, and sends acknowledgement messages consisting of a sequence number to the acknowledgement channel.

Three key parameters to the protocol are

- **tw**, the size of the *transmission window*, which specifies the maximum number of data items that the sender may send without receiving an acknowledgement to any of them
- **rw**, the size of the *reception window*, which specifies the maximum number of data items that may be received by the receiver but not yet passed on to the data target
- **w**, which specifies the maximum value for the sequence numbers used. The sequence numbers are used cyclically modulo $w + 1$.

The values of these parameters must fulfil the constraint $tw + rw \leq w + 1$ [Ste76].

The structure of the sender process is described in Figure 2.7 and that of the receiver in Figure 2.8. For more discussion on the logic and the intuitions behind the protocol, see [Ste76].

An issue that requires some further comment is the role of timeouts in the sender process. If some messages or acknowledgements are lost, the sender may enter a state where it has sent all the messages in the transmission window and is waiting for an acknowledgement that is not going to arrive. To let the system recover from such a situation a timeout mechanism is introduced. In a simplified scenario, after sending all the data items in d, the sender sets a timer. If no valid acknowledgements arrive and no new data items are received from the data source within a specified time, the timer goes off and sends a timeout-signal to the sender. Having received the timeout signal, the sender starts retransmitting all the data items in d. If a timer is set and the sender receives a valid acknowledgement or a new data item from the data source, the timer is cancelled. In practice, the timeout mechanism is likely to be more complicated; for example, Stenning [Ste76] uses a separate timer for each message.

```
SENDER

Notation
    x ⊕ y ≡df (x + y) mod (w + 1)
    x ⊖ y ≡df (x − y) mod (w + 1)
Variables
    d[0 ... (tw − 1)] : table of data items    /* Data items that have been received from the
              data source, but have not been acknowledged yet. */
    i : int /* The index of the first empty place in d. If i = 0, d is empty. If i = tw, d is full.
              i ∈ {0, ..., tw} */
    j : int /* The index of the element of d to be sent next. j ∈ {0, ..., tw} */
    k : int /* The sequence number of the data item in d[0]. k ∈ {0, ..., w} */
    /* x: a temporary variable for the sequence number in an acknowledgement. */
    /* a: a temporary variable for (x ⊖ k) + 1, i.e. for the number of ack'd data items. */
Initially
    i = 0, j = 0, k = 0

loop infinitely
    choose nondeterministically
                         /* If the data table d is not full, receive new data from source */
          i < tw       ⟶ receive( data source, d[i] );
                         i := i + 1;
    []
                         /* If not all elements of d have been sent, send next one. */
          j < i        ⟶ send( channel, (j ⊕ k, d[j]) );
                         j := j + 1;
    []
                         /* A timeout causes retransmission of messages in d. */
          j = i ∧ i > 0 ⟶ receive( timeout );
                         j := 0;
    []
                         /* Receive an acknowledgement message. If it acknowledges
                            a data item in the transmission window, remove all data items
                            up to the ack'd one from d and update i, j and k. */
          ⊤            ⟶ receive( ack channel, x );
                         a := (x ⊖ k) + 1;
                         if (a ≤ tw) then
                              d[0 ... (tw − 1 − a)] := d[a ... (tw − 1)]; /* Shift d left by a entries.*/
                              i := max(0, i − a);
                              j := max(0, j − a);
                              k := x ⊕ 1 /* i.e. = k ⊕ a */
                         end if;
    end choose;
end loop;
```

Figure 2.7: SW-protocol: Sender

RECEIVER

Notation
$x \oplus y \equiv_{df} (x + y) \bmod (w + 1)$
$x \ominus y \equiv_{df} (x - y) \bmod (w + 1)$
Variables
$t[0 \ldots (rw - 1)]$: table of data items /* Data items that have been received from the channel but have not been forwarded to the data target yet. */
$r[0 \ldots (rw - 1)]$: table of booleans /* Which data items in the reception window have already been received. $r[j] = \top$ iff the corresponding item has been received. */
e : int /* The sequence number of the first data item that has not been received yet and that is to be stored in $t[0]$. $e \in \{0, \ldots, w\}$ */
/* x: a temporary variable for the sequence number in a message. */
/* y: a temporary variable for the data item in a message. */
/* s: a temporary variable for $(x \ominus e)$. */
/* i.e. for the index in t the received data item belongs to. */
Initially
$e = 0$, $r[0 \ldots (rw - 1)] = (\bot, \ldots, \bot)$

loop infinitely
/* Receive a message from the channel. */
receive(channel, (x, y));
/* If the message has a sequence number that belongs to the current reception window, store the data item in it to t. */
$s := x \ominus e$;
if $(s < rw)$ then
$r[s] := \top$;
$t[s] := y$;
end if;
/* Forward the sequence of data items from the beginning of t up to the first unreceived data item to the data target. */
while $(r[0] \neq \bot)$ do
send(data target, $t[0]$);
$t[0 \ldots (rw - 2)] := t[1 \ldots (rw - 1)]$;
$r[0 \ldots (rw - 2)] := r[1 \ldots (rw - 1)]$;
$r[rw - 1] := \bot$;
$e := e \oplus 1$;
end while;
/* Send an acknowledgement message containing the sequence number of the last data item that has been forwarded to data target. */
send(ack channel, $(e \ominus 1)$);
end loop;

Figure 2.8: SW-protocol: Receiver

To keep the description simple, we have not presented explicitly the setting and cancelling of timers in the algorithm of Figure 2.7. Instead, we have only specified that the sender can receive a timeout-signal in situations where it cannot send any further data items to the channel. Furthermore, it is also specified that these are the only situations when a timeout-signal can be received. The latter of these assumptions is not essential for the safety properties of the protocol, only for liveness.

A further feature of the timeout mechanism is that the delay between setting the timer and a timeout should be long enough so that the timeout does not occur prematurely, when an acknowledgement is still on its way to the sender. A way to guarantee this is choosing the delay to be longer than the maximum time it may take for a message to travel from the sender to the receiver plus the maximum time for an acknowledgement to return. If such maximum times cannot be established, we can mark all messages with a time stamp and decree an expiry period so that messages past their expiry time are discarded. In this way we can view the expiry period as the maximum time a message or an acknowledgement may take to travel, even when no actual maximum time exists.

Again, the methods for avoiding premature timeouts are not presented explicitly in the algorithm of Figure 2.7. However, when modelling the system we would like to represent somehow the fact that timeouts do not occur prematurely. Furthermore, we would like to do this without introducing an explicit notion of timing constraints or real time. This will be done by requiring that there are no messages in either the channel or the acknowledgement channel and that the receiver is not about to send any acknowledgements, if a timeout is to occur.

2.5.2.2 Properties

The aim of the protocol is to transmit a sequence of data items from the data source to the data target in the correct order. Reflecting this, the following requirements should be fulfilled.

SAFE The data delivered to the target should be correct. More exactly, take any moment in an execution sequence of the protocol, and let s_r denote the sequence of data items that the sender has received from the source, and s_t the sequence of data items transmitted to the target before the given moment. Then s_t should be a prefix of s_r.

- As many data items should be delivered to the target as are read from the source. This can be divided into two parts, for finite and infinite sequences of data:

  LIVE-1 Take any execution sequence of the protocol such that n data items are read from the data source along it, and the execution is complete under the assumption that no more data items are available for reading from the source. Then n data items should be delivered to the target along the execution sequence.

  LIVE-2 Take any execution sequence of the protocol such that infinitely many data items are read from the data source along it. Then infinitely many data items should be delivered to the target along the execution sequence.

LIVE-3 The sender should accept data from the data source. More exactly, for any moment in a complete execution sequence of the protocol, there should be a future moment when the sender reads a data item from the data source, unless no more data is available for reading from the data source.

The first of these is a safety and the others liveness requirements.

However, we cannot expect the liveness properties to hold unconditionally. For example, if the channel systematically loses all the messages sent to it, clearly the liveness requirement LIVE-1 cannot hold. Therefore, we require only that the liveness properties hold under the following fairness condition:

FAIR Let $i \in \{0, \ldots, w\}$ be any sequence number. If messages with sequence number i are sent to the channel infinitely often, then messages with that sequence number will also be transmitted successfully to the receiver infinitely often. The same holds for the acknowledgement channel.

Although the requirements stated above are intuitively clear and easy to express, we can make the verification task substantially easier by appealing to a syntactic property of the system called *data-independence* [Wol86, Sab88], which allows us to verify the requirements of the system just for certain particular kinds of sequences of data items, and conclude that the system works correctly with all possible sequences of data items. As the theory of data-independence is not really a topic of the current work, full discussion on this has been postponed to Appendix A. There it is shown that the protocol fulfils the safety property SAFE iff it fulfils the safety property SAFE' below, and given that the protocol satisfies the safety properties, it fulfils the liveness properties LIVE-1, LIVE-2 and LIVE-3, respectively, iff it fulfils the liveness properties LIVE-1', LIVE-2' and LIVE-3' below.

Let 0, 1 and 2 be three distinct data items, and let x^*, x^ω, $\cup$, $\cdot$ and ϵ have their standard meaning in building ω-regular expressions [Tho90, p. 136]. Furthermore, let $x^\infty = x^* \cup x^\omega$. The safety property SAFE' is:

SAFE' Take any moment in an execution sequence of the protocol and let s_r denote the sequence of data items that the sender has received from the source, and s_t the sequence of data items transmitted to the target before the given moment. If s_r belongs to the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, then s_t belongs to the same language.

The liveness properties are:

LIVE-1' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it belongs to the language $0^* \cdot 1$, and the execution is complete under the assumption that no more data items are available for reading from the source. Then the data item 1 should be delivered to the target along the execution sequence.

LIVE-2' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it is 0^ω. Then the data item 0 should be delivered infinitely often to the target along the execution sequence.

LIVE-3' Take any moment in a complete execution sequence of the protocol such that the sequence of data items read from the data source along the execution belongs to the language 0^*. Then there should be a future moment when the sender reads a data item from the data source, unless no more data is available for reading from the data source.

In the following, we shall take SAFE', LIVE-1', LIVE-2' and LIVE-3' as the properties to be verified of the protocol.

2.5.2.3 Modelling

Having presented the system and the requirements semi-informally, we can turn to modelling the system by altss. There are some aspects of the description that need to be made more precise before constructing the altss. First, the informal description did not make any reference to the range or type of the data items. As we saw above that having three distinct data items is sufficient for our purposes, we fix the set of data items as $\{0, 1, 2\}$. Secondly, we expect the channel and the acknowledgement channel to have some upper bounds for the numbers of messages they can hold at a time. These bounds, denoted by n_c and n_a respectively, are independent of each other and of the window sizes.

In this example it is natural to use altss interacting by synchronisation. The synchronisation labels and their intuitive interpretation are as follows. The letters x and y should be understood as variables ranging over the sets $\{0, \ldots, w\}$ and $\{0, 1, 2\}$, respectively.

- **ds**y – transmission of data item y from data source to sender,
- **sc**xy – messages from the sender to the channel (x specifies the sequence number of the data item and y the data item),
- **cr**xy – messages from the channel to the receiver,
- **ra**x – acknowledgement messages from the receiver to the acknowledgement channel,
- **as**x – acknowledgement messages from the acknowledgement channel to the sender,
- **rt**y – transmission of data items from the receiver to the target, and
- **tim** – the timeout-signal.

Having fixed the values of w, tw, rw, n_c and n_a, we build a model of the system from six basic altss:

- **S**, an alts modelling the sender process,
- **R**, an alts modelling the receiver process,
- **C**, an alts modelling a channel with a capacity of one message,
- **A**, an alts modelling an acknowledgement channel with a capacity of one message,
- $\mathbf{C_F}$, an alts modelling a faulty channel with a capacity of one message, and
- $\mathbf{A_F}$, an alts modelling a faulty acknowledgement channel with a capacity of one message.

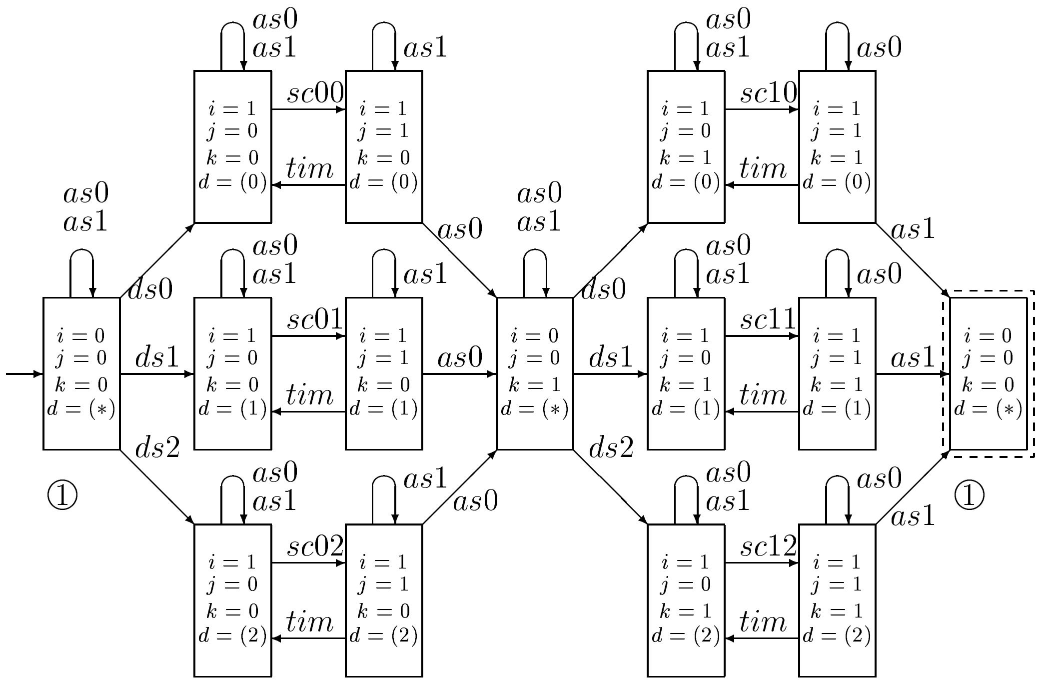


Figure 2.9: SW-protocol: Sender alts S for $w = 1$, $tw = 1$

The altss S and R can be constructed straightforwardly from the descriptions in Figures 2.7 and 2.8. The alts S modelling the sender process for $w = 1$ and $tw = 1$ is presented in Figure 2.9 and the alts R modelling the receiver process for $w = 1$ and $rw = 1$ in Figure 2.10. The dashed transitions labelled with tim in Figure 2.10 and the following ones are not part of the alts and will be explained later. The variables of the sender and receiver processes are described by propositions of the form *variable* = *value*. To keep the graphs more concise, we have not specified the values stored in those elements of the tables d and t that are irrelevant to the execution of the system, i.e. the value of $d[i']$ is not recorded if $i \leq i'$, and the value of $t[i]$ is not recorded if $r[i] = \perp$. In these cases an asterisk marks the unspecified value. Furthermore, in the graphs we have adopted the convention of attaching several labels to one arrow when an alts has several transitions with different labels between the same nodes.

Instead of directly defining an alts modelling a channel with capacity n_c, we will build it recursively, using n_c altss each modelling a channel with capacity one. Such a simple channel is essentially a one-place buffer and is easy to model. For $w = 1$ the alts C modelling a one-place channel is described in Figure 2.11. To represent the unreliability of the channel, we also need an alts modelling a faulty channel with capacity one. This is modelled as a one-place buffer that may fail to output the message it has received. For $w = 1$ the alts C_F modelling a faulty one-place channel is described in Figure 2.11. Distortion of messages in the channel is not modelled explicitly. Instead, it is assumed that distorted messages are detected by some checksum mechanism and discarded, which means that distorted messages can be treated as lost.

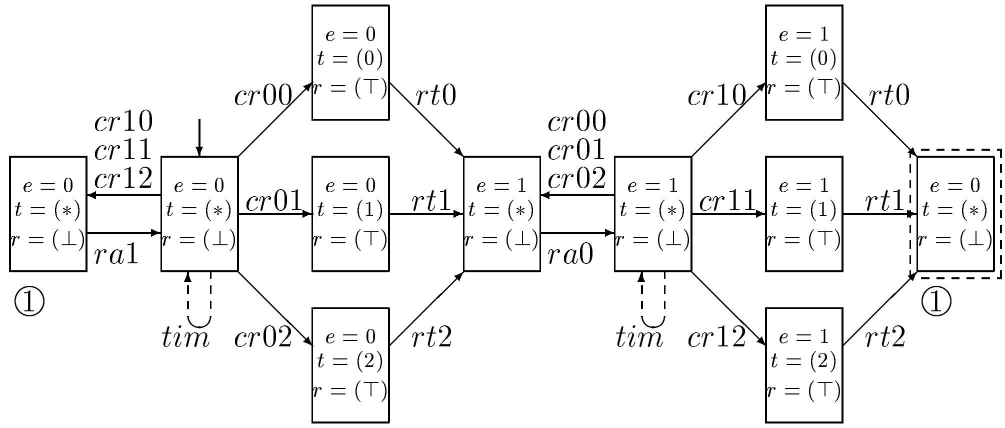


Figure 2.10: SW-protocol: Receiver alts R for $w = 1$, $rw = 1$

In a similar way, the acknowledgement channel will be built from n_a altss each modelling an acknowledgement channel with capacity one. For $w = 1$ the corresponding altss A and A_F are presented in Figure 2.11.

Having defined the basic building blocks, we can now construct an alts modelling the whole system. The structure of the construction is described in Figure 2.12. Let us start by defining inductively an alts SC corresponding to the composition of the sender and a reliable channel.

$$\begin{aligned} SC_0 &\equiv_{df} S \\ SC_{i+1} &\equiv_{df} \textbf{hide } sc* \textbf{ in } (SC_i[sc*/cr*] \parallel_s C) \\ SC &\equiv_{df} SC_{n_c-1} \end{aligned}$$

Remembering the sets of transition labels of S and C, we can state explicitly the labels on which synchronisation takes place:

$$SC_{i+1} = \textbf{hide } sc* \textbf{ in } (SC_i[sc*/cr*] \,|[sc*]|\, C)$$

In a similar way, we define inductively an alts SCA corresponding to the composition of the sender, a reliable channel and a reliable acknowledgement channel.

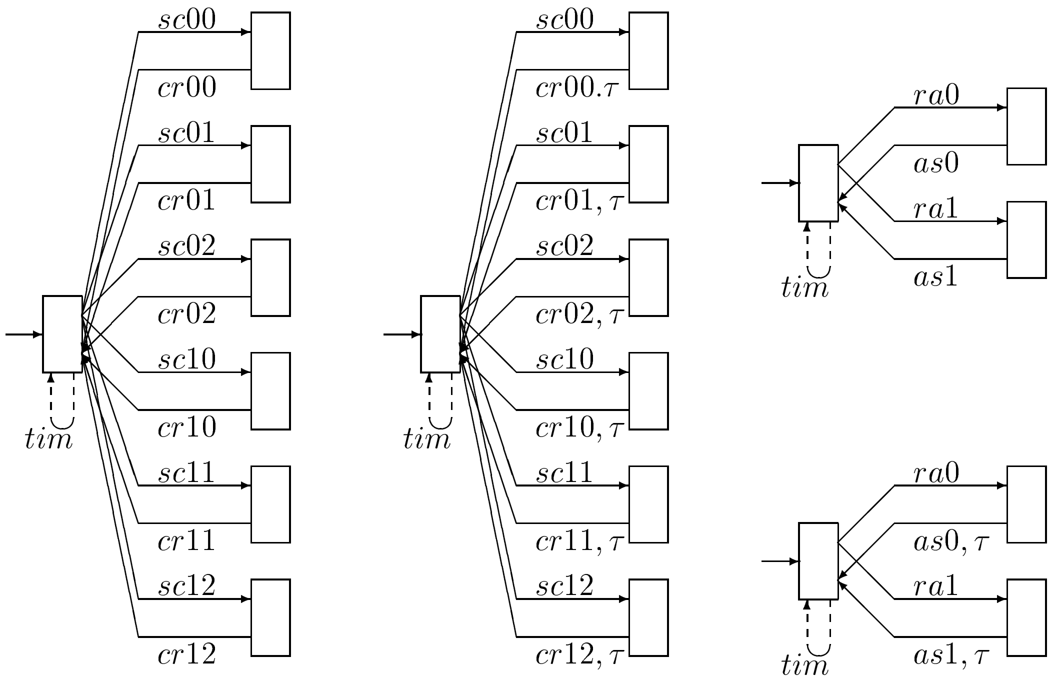
$$\begin{aligned} SCA_0 &\equiv_{df} SC \\ SCA_{i+1} &\equiv_{df} \textbf{hide } as* \textbf{ in } (SCA_i[as*/ra*] \parallel_s A) \\ SCA &\equiv_{df} SCA_{n_a-1} \end{aligned}$$

Taking into account the sets of transition labels of SC and A, we can again make explicit the labels on which synchronisation takes place:

$$SCA_{i+1} = \textbf{hide } as* \textbf{ in } (SCA_i[as*/ra*] \,|[as*]|\, A)$$

To represent the unreliability of the channel, we add one faulty one-place buffer between SCA and the receiver, and similarly for the acknowledgement channel.

$$\begin{aligned} SC_FA &\equiv_{df} SCA[sc*/cr*] \parallel_s C_F \\ SC_FA_F &\equiv_{df} SC_FA[as*/ra*] \parallel_s A_F \end{aligned}$$



Altss C (left), C_F (middle), A (top right) and A_F (bottom right).

Figure 2.11: SW-protocol: Channel and acknowledgement channel altss for $w = 1$

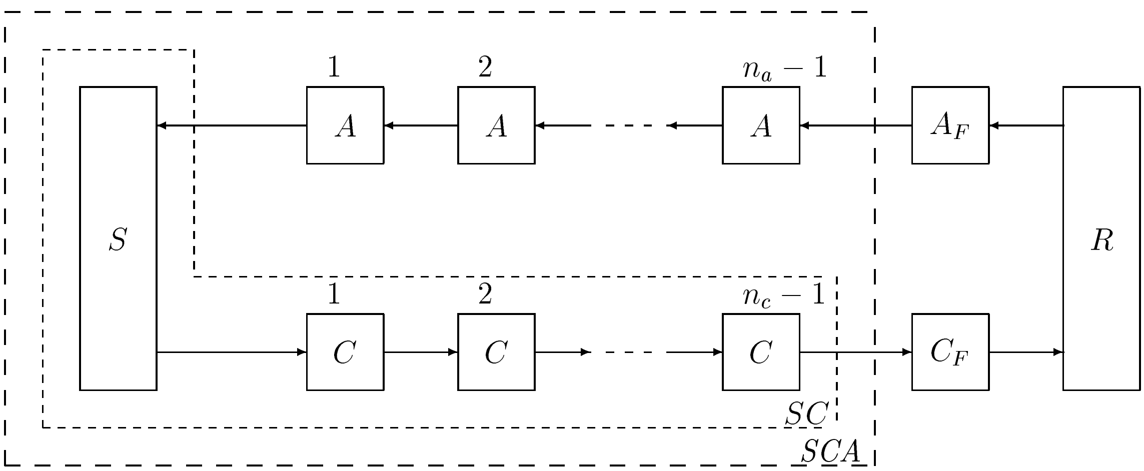


Figure 2.12: SW-protocol: Structure of the model

Notice that as far as the possible sequences of messages passing though the channel and the acknowledgement channel are concerned, it does not matter in which position the faulty element is. Another way of representing the unreliability would have been to take all the one-place buffers and acknowledgement buffers as faulty. As far as the verification of safety properties is concerned, the choice between these two is immaterial. For the verification of liveness properties, the current choice facilitates the task, as will be seen later.

Notice also that above we first build the alts SCA corresponding to the composition of the sender and a reliable channel and acknowledgement channel, and only then add the faulty buffers to it. An alternative way would have been to start from SC as above, but compose it first with the faulty buffer C_F, resulting in an alts corresponding to the sender and a faulty channel, and only after this add the reliable and faulty parts of the acknowledgement channel. The difference between these two approaches is only in the order of defining equations; the resulting altss are in both cases isomorphic.

Now the alts P corresponding to the whole protocol can be obtained by composing SC_FA_F with the receiver R:

$$P \;\equiv_{df}\; SC_FA_F \,\|_s\, R$$

Making explicit the labels on which synchronisation takes place this means:

$$P \;=\; SC_FA_F \,|[cr*, ra*]|\, R$$

However, there is still one aspect of the system that the model does not account for: the fact that timeouts do not occur prematurely. As already mentioned, this will be modelled by requiring that a timeout can occur only when the channel and the acknowledgement channel are empty, and the receiver is not about to send any acknowledgements. This is achieved by synchronising the sender S with all the other components of the system, i.e. C, C_F, A, A_F and R, with respect to the timeout-action tim so that a timeout can occur in the sender only if all the other components agree on that. Then we have to modify the other components so that they allow a timeout transition iff they are in an appropriate state.

In the altss C, C_F, A and A_F the modification is straightforward. As a timeout should occur only when the channel and the acknowledgement channel are empty, these altss modelling one-place buffers should allow timeout transitions only when they do not contain any messages, i.e. in their initial states. This means that we just need to add a new transition labelled with tim from the initial node to itself in each of C, C_F, A and A_F. These new transitions are marked in Figure 2.11 with dashed arcs.

The alts R should agree to a timeout-action only when the receiver is not about to send any acknowledgements without receiving some message from the channel first. Looking at the description of the receiver process and the way it is modelled by the alts R, we see that such situations correspond precisely to the nodes from which there are no transitions labelled with either rax for some $x \in \{0, \ldots, w\}$ or rty for $y \in \{0, 1, 2\}$. Consequently, for each such node we add a new transition labelled with tim from the node to itself. These new transitions are marked in Figure 2.10 with dashed arcs.

The way we build a model of the complete system from the basic processes stays the same as before, but we need to take into account the synchronisation on tim-labels. The

```
SENDER'
...
loop infinitely
        choose nondeterministically
                ...
        []
                                /* Receive an acknowledgement message. If it acknowledges */
                                /* a data item that has been sent, remove all data items */
                                /* up to the ack'd one from d and update i, j and k. */
                ⊤           ⟶  receive( ack channel, x );
                                a := (x ⊖ k) + 1;
                                if (a ≤ j) then
                                        ...
                                end if;
        end choose;
end loop;
```

Figure 2.13: SW-protocol: Alternative Sender

definitions for SC, SCA and P are exactly the same as above. However, the addition of tim to the sets of transition labels of C, C_F, A, A_F and R means that the forms with explicit listing of the labels to be synchronised on are slightly different:

$$\begin{aligned} SC_{i+1} &= \textbf{hide } sc{*} \textbf{ in } (SC_i[sc{*}/cr{*}] \; |[sc{*}, tim]| \; C) \\ SCA_{i+1} &= \textbf{hide } as{*} \textbf{ in } (SCA_i[as{*}/ra{*}] \; |[as{*}, tim]| \; A) \\ P &= SC_F A_F \; |[cr{*}, ra{*}, tim]| \; R \end{aligned}$$

The data source is in principle external to the definition of the protocol. However, if we have an alts D modelling some particular data source, the whole system can be modelled by

$$D \;\|_s\; P$$

2.5.3 Sliding window protocol: variations

2.5.3.1 Alternative sender

A slight variation of the SW protocol discussed in the previous section can be obtained by modifying the sender process in the way presented in Figure 2.13. The only difference between the original sender and this variant, which is sometimes found in the literature, e.g. in [RRSV87], is in the decision when to treat an acknowledgement as valid. In the original sender an acknowledgement is treated as valid, if it contains a sequence number anywhere in the current transmission window. In the variant an acknowledgement is treated as valid, if it contains a sequence number belonging to a data item that is between the beginning of the transmission window and the data item to be sent next. If an acknowledgement contains a sequence number belonging to a data item in the remaining part of the transmission window, it does not cause any effect.

```
SENDER
...
Variables
        ...
        ts : boolean      /* Timer has been set. */
Initially
        ... ts = ⊥
loop infinitely
        choose nondeterministically
                i < tw                  ⟶ receive( data source, d[i] );
                                          ts := ⊥;
                                          ...
        []
                j < i                   ⟶ ...
        []
                                          /* All messages sent and possible previous timeout cleared:
                                             Set timer. */
                j = i ∧ i > 0 ∧ ¬to     ⟶ ts := ⊤;
        []
                ⊤                       ⟶ receive( ack channel, x );
                                          ts := ⊥;
                                          ...
        []
                                          /* If timer set and timeout occurs, start retransmission. */
                ts ∧ to                 ⟶ j := 0;
                                          ts := ⊥;
        end choose;
end loop;
```

Figure 2.14: SW-protocol: Sender with timer

Adding this 'sanity check' to the protocol does not seem to be a substantial change at first sight: How could there be an acknowledgement to any of the data items in the remaining part of the transmission window, anyway? After all, the messages corresponding to these data items are only just to be sent. However, we shall see later that the difference affects liveness properties of the protocol in quite a subtle way.

The properties that the alternative version of the protocol should fulfil are the same as for the original one, and modelling the alternative version of the sender by an alts S_a is done exactly in the same way as for the original sender.

2.5.3.2 Timer process

To illustrate the use of both synchronisation and variable-based interaction in the same system, let us examine further the timeout mechanism of the SW-protocol. In the approach taken in Subsection 2.5.2, the mechanism for providing timeouts to break deadlocks was left open and it was only specified when timeouts can occur. A more faithful way of modelling can be obtained by introducing a special timer process. The timer can be set and reset by the sender process, and if set, it can signal a timeout to the sender.

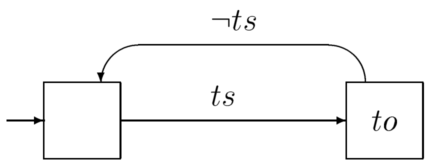


Figure 2.15: SW-protocol: Timer alts T

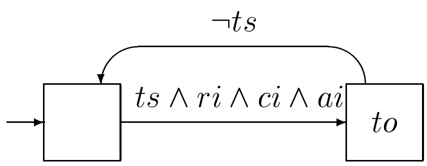


Figure 2.16: SW-protocol: Timer alts T'

Using variables for interaction, this can be done by introducing a boolean variable ts owned by the sender process, expressing whether timer is set, and a boolean variable to owned by the timer process, expressing whether a timeout has occurred. The required modifications to the sender process are described in Figure 2.14, and an alts for the timer process in Figure 2.15. The alts P for the protocol can be built on the basis of the modified sender S as in Section 2.5.2, and the complete system with the timer modelled by the alts $T \parallel_v P$.

The assumption that timeouts do not occur prematurely can also be modelled naturally using variables. Let us introduce for the receiver, the channel and the acknowledgement channel each a boolean variable, say ri, ci and ai, intuitively expressing that the corresponding component is empty. The modifications required for the receiver, channel and acknowledgement channel altss are straightforward. Now the timer process can be modified as in Figure 2.16. It is clear from T' that a timeout can be signalled only if timer is set and all components of the system are idle, so no premature timeouts can occur.

Chapter 3

Linear time temporal logic

3.1 Background

In this chapter we discuss a formal language, linear time temporal logic, which can be used to describe properties of concurrent systems. The presentation is not intended to be comprehensive, but only to cover those aspects of temporal logics that will be needed in this work. For a more thorough presentation please see [Eme90, MP91, Sti92].

Originally a branch of philosophical modal logic, temporal logic was first proposed as a specification language for concurrent programs by Pnueli in [Pnu77]. Temporal logic extends the classical propositional or first-order logic by adding a set of temporal operators such as *always*, or *sometime in the future*. Instead of being interpreted over a single, static situation, temporal logic formulae are interpreted over structures consisting of several situations that are related to each other by a temporal ordering.

There are several ways of selecting the temporal operators. The most traditional one is having dual *always* and *sometime* -operators, one pair referring to the future, the other to the past. As often in temporal logics of programs, we leave out all past operators and consider only those referring to the future, as this does not cause any loss of expressive power [GPSS80].

Since there are simple temporal properties that cannot be expressed by the *always in the future* ($\Box$) and the *sometime in the future* ($\Diamond$) operators, the temporal logic applied here is based on a more powerful *until*-operator ($\mathcal{U}$). The intuitive meaning of $\phi_1 \mathcal{U} \phi_2$ is *sometime in the future* ϕ_2 *holds and from now up to (but not necessarily including) that moment* ϕ_1 *holds*. The $\Box$ and $\Diamond$-operators are easily definable in terms of $\mathcal{U}$.

An operator usually available in linear temporal logic but not included in the logic used here is the *at the next moment of time* or $\bigcirc$-operator. As this operator is not definable in terms of the until-operator, the nexttime-less logic used here is inferior in expressive power to the standard linear temporal logic. However, it has been strongly argued that using the nexttime-operator in a specification language allows one to express distinctions between systems that should be considered equivalent according to all the observable effects they have [Lam83]. It is also the case that generally properties expressed using the nexttime-operator are not preserved in concurrent combination of processes. In our opinion all this, together with the fact that the main power of the compositional reduction techniques of Chapter 4 arises from not having to record non-state-changing transitions, more than makes up for the loss in expressiveness. What can easily be defined on the basis of the until-operator is a variant of the nexttime-operator for a given process expressing intuitively the idea *the next time something changes with respect to the local state information, the situation will be such-and-such.*

The temporal logic used here is interpreted primarily over linear discrete sequences of situations. An augmented labelled transition system induces a set of maximal execution sequences in a natural fashion. A temporal logic formula is said to hold of a program if it holds of all its maximal execution sequences. A feature not available in the logic used here is the possibility to express properties of the branching structure of the system. This qualifies the logic as a linear temporal logic in contrast with the branching-time temporal logics that give one the power of explicit quantification over possible execution branches.

The logic used here is propositional in character. Temporal operators could be added to first-order logic, as well.

3.2 Definitions

In this section we define formally our specification language, the nexttime-less linear temporal logic.

Definition 3.2.1 The *alphabet of propositional nexttime-less linear temporal logic* LTL' consists of the set AP of atomic propositions and of the symbols $(,), \neg, \vee, \mathcal{U}$. The set of *well-formed formulae* (wffs) of LTL' is the minimal set closed under the rules below:

1 if $\phi \in AP$, then ϕ is a wff,

2 if ϕ is a wff, then $(\neg\phi)$ is a wff

3 if ϕ_1 and ϕ_2 are wffs, then $(\phi_1 \vee \phi_2)$ is a wff

4 if ϕ_1 and ϕ_2 are wffs, then $(\phi_1 \mathcal{U} \phi_2)$ is a wff

If ϕ is a wff, $|\phi|$ denotes the length of ϕ, understood as a string. □

The formulae of the temporal logic are interpreted over sequences of situations. In our terminology these correspond to behaviours.

Definition 3.2.2 Let ϕ be an LTL'-formula and $\sigma \in B_\infty$ a behaviour. We say that ϕ *is true of* σ and write $\sigma \models \phi$ according to the following rules:

1 If $\phi \in AP$, then $\sigma \models \phi$ iff $\phi \in \sigma_{init}$.

2 $\sigma \models (\neg\phi)$ iff not $\sigma \models \phi$.

3 $\sigma \models (\phi_1 \vee \phi_2)$ iff $\sigma \models \phi_1$ or $\sigma \models \phi_2$.

4 $\sigma \models (\phi_1 \mathcal{U} \phi_2)$ iff there is some $0 \leq i < |\sigma| + 1$, such that

- $\sigma[i\ldots] \models \phi_2$, and
- for all $0 \leq j < i$, $\sigma[j\ldots] \models \phi_1$.

If ϕ is not true of σ, we write $\sigma \not\models \phi$. □

The interpretations of the operators are the conventional ones. Notice that if $\sigma \models \phi_2$, then $\sigma \models \phi_1 \mathcal{U} \phi_2$ for any ϕ_1. Remembering the intuitive meaning of the until-operator, this can be seen as saying that it regards the present moment as a part of the future. This qualifies our until-operator as what is often called a *reflexive until*, as opposed to an *irreflexive until*-operator, which does not take the present moment to be a part of the future.

As mentioned before, when considering the validity of an LTL'-formula over a concurrent system, every possible maximal execution sequence of the system induces a behaviour on which the formula can be interpreted. A formula holds of a system iff it holds of every behaviour corresponding to a possible maximal execution sequence of the system.

Definition 3.2.3 Let ϕ be an LTL'-formula and L an alts. We say that ϕ *is true of* L and write $L \models \phi$ iff

$$\text{for all } \sigma \in \mathit{maxbeh}(L), \sigma \models \phi$$

If ϕ is not true of L, we write $L \not\models \phi$.

We say that ϕ is *universally valid* and write $\models \phi$ iff $L \models \phi$ for all altss L. $\square$

To make it easier to express required properties in the formalism, we introduce some additional operators as syntactic abbreviations.

Definition 3.2.4 Let us define the syntactic abbreviations

- $\top = (p \vee (\neg p))$ for some fixed $p \in AP$,
- $\bot = \neg\top$,
- $(\phi_1 \wedge \phi_2) = (\neg((\neg\phi_1) \vee (\neg\phi_2)))$,
- $(\phi_1 \Rightarrow \phi_2) = ((\neg\phi_1) \vee \phi_2)$,
- $(\Diamond\phi) = (\top \mathcal{U} \phi)$,
- $(\Box\phi) = (\neg(\Diamond(\neg\phi)))$, and
- $(\phi_1 \mathcal{U}_w \phi_2) = ((\phi_1 \mathcal{U} \phi_2) \vee (\Box\phi_1))$.

and the following precedence rules to reduce the number of parentheses:

$$\Box, \Diamond, \neg, \mathcal{U}, \mathcal{U}_w, \wedge, \vee, \Rightarrow$$

where $\Box$ has the highest precedence and $\Rightarrow$ the lowest, i.e. $\Box\phi_1 \Rightarrow \phi_2$ abbreviates $((\Box\phi_1) \Rightarrow \phi_2)$. $\square$

Notice the difference between the $\phi_1 \mathcal{U} \phi_2$ and $\phi_1 \mathcal{U}_w \phi_2$ constructions. The former says that ϕ_2 holds at some future time and ϕ_1 holds until ϕ_2 holds. The latter, on the other hand, says only that ϕ_1 holds until ϕ_2 holds: in particular, if ϕ_2 never holds, then ϕ_1 holds forever. Because of this, $\mathcal{U}_w$ is often called in the literature the *weak until* or the *unless* operator.

The logic LTL', like most temporal logics used in program specification, is essentially state-based: it allows us to describe properties of execution states and sequences of them.

However, sometimes we would also like to make reference to transitions and their labels, for example when expressing properties like *such and such action will happen.* To this purpose we introduce a slight extension of LTL'. Similar extensions from state-based to transition-based logics are discussed in [DV90a].

Definition 3.2.5 The alphabet of the logic $tLTL'$ is that of LTL' with an additional symbol $\mathcal{U}^a$ for each $a \in \Sigma$. The set of well-formed formulae (wffs) of $tLTL'$ is defined as the minimal set closed under the rules 1-4 of Definition 3.2.1 and the following rule:

5 if ϕ_1 and ϕ_2 are wffs and $a \in \Sigma$, then $(\phi_1 \mathcal{U}^a \phi_2)$ is a wff. □

Definition 3.2.6 If ϕ is a $tLTL'$-formula and $\sigma \in B_\infty$ a behaviour, $\sigma \models \phi$ is defined by the rules 1-4 of Definition 3.2.2 and the following rule:

5 $\sigma \models \phi_1 \mathcal{U}^a \phi_2$ iff there is some $1 \leq i < |\sigma| + 1$ such that

- $\sigma[i \ldots] \models \phi_2$,
- $\sigma_i = (a, A)$ for some $A \subseteq AP$,
- for all $0 \leq j < i$, $\sigma[j \ldots] \models \phi_1$.
- for all $1 \leq j < i$, $\sigma_j = (l, A)$ for some $l \neq a$ and $A \subseteq AP$. □

Intuitively $\phi_1 \mathcal{U}^a \phi_2$ says that

- some future transition is labelled with a,
- immediately after the next transition labelled with a, ϕ_2 holds, and
- until that moment ϕ_1 holds.

Let us introduce some syntactic abbreviations and notation.

Definition 3.2.7 Let us define the syntactic abbreviations:

- $\boxed{a}\phi = \top \mathcal{U}^a \phi$
- $\boxed{a} = \top \mathcal{U}^a \top$. □

Intuitively, $\boxed{a}\phi$ says that ϕ *holds after the next transition labelled with a, and there is such a transition*, and $\boxed{a}$ that *some future transition is labelled with a.*

Definition 3.2.8 Let ϕ be an LTL' or $tLTL'$-formula. Then $AP(\phi)$ denotes the set of atomic propositions occurring in ϕ.

Let ϕ is a $tLTL'$-formula. Then $\Sigma(\phi)$ denotes the set of transition labels $a \in \Sigma$ occurring in operators $\mathcal{U}^a$ in ϕ. For an LTL'-formula ϕ, we define $\Sigma(\phi) = \emptyset$. □

3.3 Expressing properties

Having defined the specification language and its interpretation, we can turn to the issues of expressing interesting properties of systems in the language. The rough overall classification of system properties into safety and liveness properties is reflected in the structure of the formulae expressing these properties. For example, safety properties saying ϕ *never happens* can be formulated in LTL' as $\Box\neg\phi$, and simple liveness properties of the type *eventually* ϕ *happens* typically result in formulae like $\Diamond\phi$ or $\Box\Diamond\phi$. More complex formulae are needed for other types of properties. The issue of expressing interesting system properties in linear temporal logic is discussed in great detail in [MP91]. Here we just illustrate this by some examples.

Example 3.3.1 The required properties of the mutual exclusion protocol presented in Subsection 2.5.1 can be expressed by the following formulae:

- The program establishes mutual exclusion:

  $$\bigwedge_{1\leq i,j\leq n\ \ i\neq j} \Box\neg(c_i \wedge c_j)$$

  This can be divided into $n \cdot (n-1)/2$ individual properties, each establishing the mutual exclusion between two processes i and j for $i \neq j$:

  $$\Box\neg(c_i \wedge c_j)$$

- Each request is eventually granted:

  $$\bigwedge_{1\leq i\leq n} \Box(r_i \Rightarrow \Diamond c_i)$$

  which can be divided into n properties

  $$\Box(r_i \Rightarrow \Diamond c_i)$$

- One-bounded overtaking:

  $$\bigwedge_{1\leq i,j\leq n\ \ i\neq j} \Box(r_i \Rightarrow (\neg g_j)\,\mathcal{U}_w(g_j\,\mathcal{U}_w((\neg g_j)\,\mathcal{U}_w g_i)))$$

  which can be divided into

  $$\Box(r_i \Rightarrow (\neg g_j)\,\mathcal{U}_w(g_j\,\mathcal{U}_w((\neg g_j)\,\mathcal{U}_w g_i)))$$

  Notice here the use of $\mathcal{U}_w$ instead of the $\mathcal{U}$-operator. Since we are not claiming here that the request will ever be granted, we do not require that g_i will eventually become true. Instead, we only require that *if* g_i ever becomes true, then g_j has changed from false to true at most once between now and that time.

The first and third are safety properties and the second is a liveness property. $\Box$

Example 3.3.2 To verify the required properties of the SW-protocol introduced in Subsection 2.5.2, we first have to show that it meets the safety requirement SAFE':

SAFE' Take any moment in an execution sequence of the protocol and let s_r denote the sequence of data items that the sender has received from the source, and s_t the sequence of data items transmitted to the target before the given moment. If s_r belongs to the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, then s_t belongs to the same language.

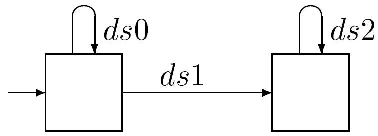


Figure 3.1: SW-protocol: Data source alts D_{safe}

To do this, we plug into the protocol a data source producing precisely the language $0^{\infty} \cdot (1 \cdot 2^{\infty} \cup \epsilon)$ and verify that on any possible maximal execution of this system the sequence of data items transmitted to the target belongs to the same language. The alts D_{safe} modelling such a data source is in Figure 3.1.

An execution of the system is guaranteed to fulfil the property *the sequence of data items delivered to the target belongs to the language* $0^{\infty} \cdot (1 \cdot 2^{\infty} \cup \epsilon)$ iff it satisfies the $tLTL'$-formula ϕ_{safe} below:

$$\begin{aligned}
\phi_{S1} &\equiv_{df} & ((rt1) \Rightarrow (rt1)(\neg (rt0) \wedge \neg (rt1))) \\
\phi_{S2} &\equiv_{df} & ((rt2) \Rightarrow (rt2)(\neg (rt0) \wedge \neg (rt1))) \\
\phi_{S3} &\equiv_{df} & ((rt2) \Rightarrow (rt1)) \\
\phi_{safe} &\equiv_{df} & \phi_{S1} \wedge \phi_{S2} \wedge \phi_{S3}
\end{aligned}$$

Intuitively, the first formula says that after a 1 is delivered to the data target, no 0:s or 1:s are going to be delivered to it. The second formula says that after a 2 is delivered to the data target, no 0:s or 1:s are going to be delivered to it. Finally, the third formula says that a 2 cannot be delivered to the data target unless a 1 is delivered, as well.

Summarising, to see that the protocol satisfies the safety requirement SAFE', we try to verify that:

$$D_{safe} \parallel_s P \quad \models \quad \phi_{safe}$$

For the definition of P, see page 35.

Secondly, assuming that we manage to verify the safety properties, we would also like to verify that the protocol meets the liveness requirements LIVE-1', LIVE-2' and LIVE-3':

LIVE-1' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it belongs to the language $0^* \cdot 1$, and the execution is complete under the assumption that no more data items are available for reading from the source. Then the data item 1 should be delivered to the target along the execution sequence.

LIVE-2' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it is 0^{ω}. Then the data item 0 should be delivered infinitely often to the target along the execution sequence.

LIVE-3' Take any moment in a complete execution sequence of the protocol such that the sequence of data items read from the data source along the execution belongs

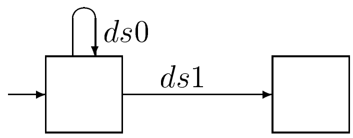


Figure 3.2: SW-protocol: Data source alts D_{live}

to the language 0^*. Then there should be a future moment when the sender reads a data item from the data source, unless no more data is available for reading from the data source.

To verify the liveness properties, we connect into the protocol the data source D_{live} in Figure 3.2, producing the language $0^\omega \cup 0^* \cdot 1$, and verify that the following property LIVE' is true of the resulting system:

LIVE' on any possible maximal execution either the data item 1 or infinitely many 0:s are transmitted to the target.

Notice that whereas LIVE-1' etc. are properties of the protocol, LIVE' is a property of the system consisting of the protocol and the particular data source D_{live}. Let us see why verifying LIVE' suffices.

Suppose first that LIVE-1', LIVE-2' and LIVE-3' all hold. Since LIVE-3' holds, in any complete execution of the system consisting of the protocol and D_{live}, the sequence of data items read from the source is either 0^ω or in the language $0^* \cdot 1$. As LIVE-1' and LIVE-2' hold, then either the data item 1 or infinitely many 0:s are delivered to the target along any complete execution of the system, and LIVE' holds.

Conversely, suppose that LIVE-3' fails. In this case the system consisting of the protocol and D_{live} has a complete execution for which the sequence of data items read from the data source belongs to the language 0^*. By SAFE', the sequence of data items transmitted to the target belongs also to 0^*, and LIVE' fails. If LIVE-2' fails, the protocol has an execution sequence along which the sequence of data items read from the source is 0^ω, but 0 is delivered only finitely often to the target. By SAFE', only 0:s may be delivered to the target in this execution sequence, implying that the system consisting of the protocol and D_{live} has an infinite execution sequence along which no 1:s and only finitely many 0:s are delivered to the target, i.e. LIVE' fails. If LIVE-1' fails, the protocol has an execution sequence such that the sequence of data items read from the data source along it belongs to $0^* \cdot 1$, the execution is complete under the assumption that no more data items are available from the source, but the data item 1 is not delivered to the target along the execution sequence. By SAFE', only finitely many 0:s can be delivered to the target along this execution sequence, implying that the system consisting of the protocol and D_{live} has a maximal execution sequence along which no 1:s and only finitely many 0:s are delivered to the target, and LIVE' fails.

Consequently, to verify LIVE-1', LIVE-2' and LIVE-3', it is enough to verify that LIVE' is true of the system consisting of the protocol connected to the data source D_{live}.

We can express the required property of maximal executions by the following formula:

$$\Box\Diamond \textit{rt0} \vee \Diamond \textit{rt1}$$

However, we expect the liveness properties to hold only under the fairness constraint FAIR:

FAIR Let $i \in \{0, \ldots, w\}$ be any sequence number. If messages with sequence number i are sent to the channel infinitely often, then messages with that sequence number will also be transmitted successfully to the receiver infinitely often. The same holds for the acknowledgement channel.

This constraint can be expressed by the following $tLTL'$-formula.

$$\phi_{fair} \equiv_{df} \bigwedge_{0\le i\le w} (\Box\Diamond(\bigvee_{0\le j\le 2} \textit{scij}) \Rightarrow \Box\Diamond(\bigvee_{0\le j\le 2} \textit{crij})) \wedge \bigwedge_{0\le i\le w} (\Box\Diamond \textit{rai} \Rightarrow \Box\Diamond \textit{asi})$$

Let us define

$$\phi_{live} \equiv_{df} (\phi_{fair} \Rightarrow (\Box\Diamond \textit{rt0} \vee \Diamond \textit{rt1}))$$

Summarising, to see that the protocol satisfies the liveness requirements LIVE-1', LIVE-2' and LIVE-3' under the fairness constraint FAIR, we have to verify that

$$D_{live} \parallel_s P \models \phi_{live}$$

$\Box$

3.4 Verifying properties

This far we have established how to model concurrent systems by altss and how to express their properties by temporal logic formulae. Now we turn to the issue of finding out whether a system has a required property. The methods for this can for historical reasons be roughly classified into two categories.

In the so-called *theorem-proving* approach, one tries to prove in some formal deductive system that a system satisfies a formula describing the required properties. Although by no means a necessary ingredient of the approach, often the main burden of the verification task is on the human verifier: even though the proof can be and often is checked by a mechanical proof assistant, the task of constructing the proof lies on the verifier.

In the so-called *model-checking* approach, on the other hand, one tries to apply reasoning on the basis of the underlying model in order to establish the required properties. This time, although again by no means a necessary ingredient of the approach, the trend is often to automate the task to as large an extent as possible. If the model of the system is finite, the question of whether a given formula holds of it or not can often be decided by reasonably effective algorithms [CES86, LP85, EL87], in which case the requirements to the human verifier are minimal. In addition to such fully automated methods there are also methods which allow and require the human verifier to guide the verification task, e.g. the tableau-based model-checkers in [SW91, BS92].

In this work we follow the latter track, and concentrate on establishing required properties by automated model-checking. Without going into the details of the actual algorithm deciding the validity of an LTL'-formula ϕ over a system L, we list some results related to this.

Proposition 3.4.1 Determining whether $L \models \phi$ for given alts L and LTL'-formula ϕ is PSPACE-complete. The same holds for determining whether $\models \phi$ for a given ϕ.

Proof: See [SC85, Theorem 4.1]. It should be pointed out that this theorem of Sistla and Clarke is stated precisely for the linear time temporal logic without a nexttime-operator, i.e. LTL'. □

Proposition 3.4.2 Determining whether $L \models \phi$ for given alts L and LTL'-formula ϕ can be done in deterministic $O(|L| \cdot 2^{O(|\phi|)})$ time. The same holds for $tLTL'$.

Determining whether $\models \phi$ for a given ϕ can be done in deterministic $2^{O(|\phi|)}$ time.

Proof: An algorithm for model-checking the full linear temporal logic is presented in [LP85]. This algorithm satisfies the time bound above. Since LTL' is a sublanguage of the full linear temporal logic, the algorithm can be used for model-checking LTL' without any changes. The changes required for accommodating the $\mathcal{U}^a$ operator of $tLTL'$ are straightforward.

The statement concerning $\models \phi$ is shown in [Wol83]. See also [LPZ85, KMMP93] for other validity-checking algorithms for linear time temporal logic. □

Although the worst-case time-complexity of the algorithm referred to in Proposition 3.4.2 is singly exponential in the length of the formula, this factor is usually not a major problem. Since many required properties can be expressed rather concisely, the exponential factor is often relatively low in practice [LP85].

On the other hand, although the algorithm is only linear in the size of the alts L, the model-checking of realistic systems is often impossible because of the sheer size of L. This may seem counter-intuitive. However, in concurrent systems L is typically a composition of several individual components: $L = (P_1 \parallel \ldots \parallel P_n)$. Due to representing concurrency by interleaving, $|L|$ is of the order of $|P_1| \cdot \ldots \cdot |P_n|$, i.e. exponential in the number of components. Although not all the nodes of L are necessarily reachable, usually even the number of reachable nodes grows rapidly enough so as to make L infeasibly large. In the rest of this work we look at some ways of alleviating this so-called *state explosion* problem.

Notice that no such phenomenon arises with sequential composition. In this case the size of a complete system is linear in the number of components. Therefore, sequential composition does not seem to pose an obstacle to automated model checking in the same way as parallel composition does.

Example 3.4.3 To illustrate the state explosion problem, the table in Figure 3.3 lists the number of reachable nodes of the alts $L = (S, I, \Gamma, \Delta, A, V)$ modelling the mutual exclusion system for some values of n. It is clear that straightforward model-checking is feasible only for very small n.

n	$\lvert S\rvert = 2 \cdot n \cdot 3^n$
2	36
5	2430
10	1180980
100	$> 2^{166}$
500	$> 2^{800}$

Figure 3.3: Mutual exclusion: Sizes of altss (reachable nodes)

The general formula for the size of S can be justified as follows. For a system with n customers, the arbiter has $2n$ states. In each state it either grants access to the resource for one customer or denies access to all customers. In the states in which arbiter denies access to all customers, every customer can be in any state except its critical section, i.e. there are 3^n choices. In the states in which arbiter grants access to one customer, this customer can be in in any state but its initial one, and every other customer can be in any state except in the critical section, so again there are 3^n choices. Consequently, there are $2 \cdot n \cdot 3^n$ choices for the global state. $\square$

Example 3.4.4 The table in Figure 3.4 lists the number of reachable nodes in some altss used for modelling the SW-protocol for some small values of the parameters w, tw, rw, n_c and n_a and the data source D_{safe}. The choice of values was dictated by the tools used, which did not allow constructing altss with more than c. 75000 nodes.

Using the particular data source D_{safe} restricts the number of combinations of data values that the protocol is processing at any moment. Despite this, the size of the system grows rapidly, especially when the length of the channel n_c and the size of the transmission window tw are increased. To make the analysis realistic, we should not use a channel length n_c that is smaller than the transmission window size tw. Bearing this in mind, it can be seen from the table that straightforward model-checking becomes quite hard when $tw > 2$. $\square$

Parameters				Reachable nodes		
w	tw	rw	$n_a = n_c$	S	R	$D_{safe} \parallel_s P$
1	1	1	1	14	10	87
2	1	1	1	21	15	124
			2			178
	2	1	1	102	15	1096
			2			2299
3	2	2	2	136	80	3479
			3			6164
	3	1	2	568	20	14828
			3			35649
4	2	2	2	170	100	4320
			3			7659
	3	2	2	710	100	23276
			3			54223
	4	1	1	2735	25	16300
			2			55638
5	2	2	2	204	120	5161
	3	3	1	852	480	11702
			2			31567
	4	2	1	3282	120	28868
	5	1	1	12030	30	41475
6	2	2	2	238	140	6002
	3	3	1	994	560	13615
			2			36747
	4	3	1	3829	560	43562
	5	2	1	14035	140	75437
7	2	2	2	272	160	6843
	3	3	1	1136	640	15528
			2			41927
	4	4	1	4376	2560	56947
	5	3	1	16040	640	a)
Note: a) too large to be constructed						

Figure 3.4: SW-protocol: Sizes of altss (reachable nodes)

Chapter 4

Equivalences

4.1 Preserving temporal logic properties

The basic task this and the following chapters concentrate on can be briefly paraphrased as follows: Given a system $L = (P_1 \parallel \ldots \parallel P_n)$ and an LTL'-formula ϕ, try to check whether $L \models \phi$ using less effort than straightforward model-checking would take. On a very general level the method presented in this chapter is based on replacing L with an alts L' so that for all LTL'-formulae ϕ, $L \models \phi$ iff $L' \models \phi$, and it is easier to check whether $L' \models \phi$ holds than whether $L \models \phi$ holds. In the following we usually equate the ease of model-checking with the size of a system, so the aim can be rephrased as trying to find a smaller L' such that $L \models \phi$ iff $L' \models \phi$, and trying to check that $L' \models \phi$ holds. However, it needs to be pointed out that the primary concern is making the verification task easier, and the reason the size of the replacement L' is of interest is as a means to achieving this aim.

As the relation *validating the same LTL'-formulae* is an equivalence relation between altss, we introduce the following notation.

Definition 4.1.1 Let X be a set and $R \subseteq X \times X$ a relation on X. We say that R is an *equivalence* iff it is

- reflexive, i.e. $R(x, x)$ holds for all $x \in X$,
- symmetric, i.e. $R(x, y)$ implies $R(y, x)$ for all $x, y \in X$, and
- transitive, i.e. $R(x, y)$ and $R(y, z)$ imply $R(x, z)$ for all $x, y, z \in X$. □

In the following, whenever a relation is called an equivalence, we implicitly imply that it has the above three properties. In all the cases it is obvious that the alleged equivalences actually satisfy these requirements.

Definition 4.1.2 Let L and L' be altss. We say that L and L' are *LTL'-equivalent* and write $L \overset{ltl'}{\approx} L'$ iff for all LTL'-formulae ϕ, $L \models \phi$ iff $L' \models \phi$.

Similarly, we say that L and L' are *$tLTL'$-equivalent* and write $L \overset{tltl'}{\approx} L'$ iff for all $tLTL'$-formulae ϕ, $L \models \phi$ iff $L' \models \phi$. □

This, of course, does not yet give us any clue on how to find a smaller model equivalent to a given model. The first step in this direction is providing a more constructive characterisation of the LTL'-equivalence. For this we need some additional definitions.

Definition 4.1.3 Let $A \in P(AP)$ be a truth set and $(l, A') \in \Sigma_{aug}$ an augmented transition label. We say that (l, A') *is invisible in situation* A iff

- $l = \tau$, and
- $A = A'$.

Otherwise (l, A') is said to be *visible in situation* A. □

The intuition behind the previous definition is that if an action cannot interact with the environment and does not affect the observable characteristics of the local state in any way, we cannot directly observe the action taking place.

A characteristic feature of the logic LTL' is that it is unable to distinguish a behaviour ρ from another one that has been obtained from ρ by adding or removing invisible transitions. This feature is sometimes referred to as the *stuttering-immunity* of LTL' [Lam83].

Definition 4.1.4 Let $\rho \in A_0 l_1 A_1 \ldots \in B_\infty$ be a behaviour. We say that ρ is a *visible behaviour* iff for all $1 \leq i < |\rho| + 1$, (l_i, A_i) is visible in situation A_{i-1}. The set of all visible behaviours is denoted by VB_∞.

The set of all finite visible behaviours is denoted by $VB_* = VB_\infty \cap B_*$, and the set of all infinite visible behaviours by $VB_\omega = VB_\infty \cap B_\omega$. □

Definition 4.1.5 Let $\rho = A_0 l_1 A_1 \ldots \in B_\infty$ be a behaviour. The *visible subsequence of* ρ is

$$vis(\rho) = A_0 l_{i_1} A_{i_1} l_{i_2} A_{i_2} \ldots l_{i_m} A_{i_m} \ldots$$

where $i_1, i_2, \ldots$ is a strictly increasing enumeration of the set

$$\{0 < i < |\rho| + 1 \mid (l_i, A_i) \text{ is visible in situation } A_{i-1}\}$$

□

In other words, $vis(\rho)$ is the sequence obtained from ρ by removing from it every augmented transition label (l_i, A_i) that is invisible in the previous situation A_{i-1}. Notice that the visible subsequence of an infinite behaviour may be finite.

Lemma 4.1.6 Let ϕ be an LTL'-formula. Then for all behaviours $\rho \in B_\infty$:

$$\rho \models \phi \text{ iff } vis(\rho) \models \phi$$

Proof: The claim is proved by induction on the structure of the formula ϕ.

Suppose that ϕ is atomic. Then for any $\rho \in B_\infty$, $\rho \models \phi$ iff $\phi \in \rho_{init}$. But clearly $\rho_{init} = vis(\rho)_{init}$, which means that $\rho \models \phi$ iff $vis(\rho) \models \phi$. The induction step is trivial, when $\phi = \neg\phi_1$ or $\phi = \phi_1 \vee \phi_2$.

Suppose then that $\phi = \phi_1 \mathcal{U} \phi_2$, and that the claim holds for ϕ_1 and ϕ_2. Take any behaviour $\rho \in B_\infty$. Let us define a function f from the states of ρ to the states of $vis(\rho)$,

$$f : \{i \in \mathbb{N} \mid i < |\rho| + 1\} \to \{j \in \mathbb{N} \mid j < |vis(\rho)| + 1\}$$

recursively by

$$\begin{array}{rcl} f(0) & = & 0 \\ f(i+1) & = & \begin{cases} f(i)+1 & \text{if } (l_{i+1}, A_{i+1}) \text{ is visible in situation } A_i \\ f(i) & \text{otherwise} \end{cases} \end{array}$$

Intuitively, $f(i)$ expresses how many visible transitions occur in ρ before point i.

It should be easy to see that

1 $vis(\rho[n\ldots]) = vis(\rho)[f(n)\ldots]$ for all $n < |\rho| + 1$, and

2 f is surjective and monotonic.

By definition, $\rho \models \phi_1 \mathcal{U} \phi_2$ iff there is some $0 \le i < |\rho| + 1$ such that $\rho[i\ldots] \models \phi_2$ and for all $0 \le j < i$, $\rho[j\ldots] \models \phi_1$. By induction hypothesis, this holds iff there is some $0 \le i < |\rho| + 1$ such that $vis(\rho[i\ldots]) \models \phi_2$ and for all $0 \le j < i$, $vis(\rho[j\ldots]) \models \phi_1$. By property 1 of f, this holds iff there is some $0 \le i < |\rho| + 1$ such that $vis(\rho)[f(i)\ldots]) \models \phi_2$ and for all $0 \le j < i$, $vis(\rho)[f(j)\ldots]) \models \phi_1$. Since f is surjective and monotonic, this holds iff there is some $0 \le i' < |vis(\rho)| + 1$ such that $vis(\rho)[i'\ldots]) \models \phi_2$ and for all $0 \le j' < i'$, $vis(\rho)[j'\ldots]) \models \phi_1$, i.e. iff $vis(\rho) \models \phi_1 \mathcal{U} \phi_2$ This concludes the induction step for $\phi = \phi_1 \mathcal{U} \phi_2$. □

Analogously to the concepts and notation used to discuss behaviours, we introduce some concepts and notation for visible behaviours.

Definition 4.1.7 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts.

Let $\sigma \in VB_*$ be a finite visible behaviour, and let $s, s' \in S$. We say that *there is a path from s to s' with visible behaviour σ* and write $s \stackrel{\sigma}{\Longrightarrow} s'$ iff there is a $\rho \in B_*$ such that $s \stackrel{\rho}{\longrightarrow} s'$ and $vis(\rho) = \sigma$, i.e. there is a path from s to s' with behaviour ρ and the visible subsequence of ρ is σ.

Let $\sigma \in VB_\infty$ be a visible behaviour, and let $s \in S$. We say that *there is a path from s with visible behaviour σ* and write $s \stackrel{\sigma}{\Longrightarrow}$ iff there is a $\rho \in B_\infty$ such that $s \stackrel{\rho}{\longrightarrow}$ and $vis(\rho) = \sigma$, i.e. there is a path from s with behaviour ρ and the visible subsequence of ρ is σ. □

In the following definition we digress from the systematic way of naming concepts, and call finite visible behaviours of an alts *traces*. This corresponds to the standard meaning of the term in process algebraic terminology [Mil89].

Definition 4.1.8 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts. A visible behaviour $\sigma \in VB_\infty$ is a *visible behaviour of L* iff there is an $s \in I$ such that $s \stackrel{\sigma}{\Longrightarrow}$. The set of all visible behaviours of L is denoted by $vbeh(L)$.

The set of all *finite visible behaviours of L*, also called *traces of L* and denoted by $tr(L)$, is defined by $tr(L) = vbeh(L) \cap VB_*$.

The set of all *infinite visible behaviours of L*, also called *infinite traces of L* and denoted by $inftr(L)$, is defined by $inftr(L) = vbeh(L) \cap VB_\omega$. □

The maximal behaviours of an alts can be divided into two sets: finite ones, corresponding to finite executions leading to a deadlock state, and infinite ones, corresponding to infinite executions. When discussing maximal visible behaviours, the picture is slightly more complex, since there is a third alternative: finite visible behaviours corresponding to infinite executions. These are called *divergences*.

Definition 4.1.9 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts.

A finite visible behaviour $\sigma \in VB_*$ is a *deadlock trace of* L iff there is a deadlock behaviour of L, $\rho \in \mathit{dlbeh}(L)$, such that $\mathit{vis}(\rho) = \sigma$. The set of all deadlock traces of L is denoted by $\mathit{dltr}(L)$.

A finite visible behaviour $\sigma \in VB_*$ is a *divergence trace of* L iff there is an infinite behaviour of L, $\rho \in \mathit{infbeh}(L)$, such that $\mathit{vis}(\rho) = \sigma$. The set of all divergence traces of L is denoted by $\mathit{divtr}(L)$.

A finite visible behaviour $\sigma \in VB_*$ is a *non-divergent deadlock trace of* L iff it is a deadlock trace of L but not a divergence trace of L. The set of all non-divergent deadlock traces of L is denoted by $\mathit{nddltr}(L)$.

A finite visible behaviour $\sigma \in VB_*$ is a *finite maximal trace of* L iff it is either a deadlock trace of L or a divergence trace of L. The set of all finite maximal traces of L is denoted by $\mathit{fmaxtr}(L) = \mathit{dltr}(L) \cup \mathit{divtr}(L) = \mathit{nddltr}(L) \cup \mathit{divtr}(L)$.

A visible behaviour $\sigma \in VB_\infty$ is a *maximal trace of* L iff it is either a finite maximal trace of L or an infinite trace of L. The set of all maximal traces of L is denoted by $\mathit{maxtr}(L) = \mathit{fmaxtr}(L) \cup \mathit{inftr}(L)$. □

Lemma 4.1.10 Let L be an alts. Then

$$\begin{aligned}
\mathit{tr}(L) &= \{\sigma \in VB_* \mid \text{there is a } \rho \in \mathit{maxbeh}(L) \text{ such that } \sigma \preceq \mathit{vis}(\rho)\} \\
&= \{\sigma \in VB_* \mid \text{there is a } \sigma' \in \mathit{maxtr}(L) \text{ such that } \sigma \preceq \sigma'\} \\
\mathit{inftr}(L) &= \{\sigma \in VB_\omega \mid \text{there is a } \rho \in \mathit{maxbeh}(L) \text{ such that } \sigma = \mathit{vis}(\rho)\} \\
\mathit{maxtr}(L) &= \{\sigma \in VB_\infty \mid \text{there is a } \rho \in \mathit{maxbeh}(L) \text{ such that } \sigma = \mathit{vis}(\rho)\} \\
\mathit{fmaxtr}(L) &= \mathit{maxtr}(L) \cap VB_* \\
\mathit{inftr}(L) &= \mathit{maxtr}(L) \cap VB_\omega
\end{aligned}$$

Proof: Immediate from the definitions of *tr*, *inftr* (4.1.8), *maxtr* (4.1.9) and *maxbeh* (2.3.10). □

Definition 4.1.11 Let L and L' be altss. We say that L and L' are *maximal trace equivalent* and write $L \overset{maxtr}{\approx} L'$ iff $\mathit{maxtr}(L) = \mathit{maxtr}(L')$ □

Theorem 4.1.12 Let L and L' be altss such that $L \overset{maxtr}{\approx} L'$. Then $L \overset{ltl'}{\approx} L'$.

Proof: Suppose that L and L' are altss such that $L \overset{maxtr}{\approx} L'$, and take any LTL'-formula ϕ. Then:

$L \models \phi$

iff $\rho \models \phi$ for all $\rho \in \mathit{maxbeh}(L)$ (by Definition 3.2.3)

iff $\mathit{vis}(\rho) \models \phi$ for all $\rho \in \mathit{maxbeh}(L)$ (by Lemma 4.1.6)

iff $\sigma \models \phi$ for all $\sigma \in maxtr(L)$ (by Lemma 4.1.10)
iff $\sigma \models \phi$ for all $\sigma \in maxtr(L')$ (by assumption)
iff $vis(\rho) \models \phi$ for all $\rho \in maxbeh(L')$ (by Lemma 4.1.10)
iff $\rho \models \phi$ for all $\rho \in maxbeh(L')$ (by Definition 4.1.6)
iff $L' \models \phi$. □

A similar result holds for the logic $tLTL'$.

Proposition 4.1.13 Let L and L' be altss such that $L \overset{maxtr}{\approx} L'$. Then $L \overset{tltl'}{\approx} L'$.

Proof: Essentially as the proof of 4.1.12. □

We now know that $L \overset{maxtr}{\approx} L'$ is a sufficient condition for $L \overset{ltl'}{\approx} L$. In the following we shall examine to what extent it is also a necessary one. For this we need first some definitions and lemmas.

Definition 4.1.14 Let L be an alts. We say that L is *closed* iff $\Sigma(L) = \emptyset$, i.e. iff all the transitions of L are labelled with τ:s. □

Intuitively, if an alts is closed, the environment cannot affect its behaviour in any way. Therefore, in a sense a closed alts corresponds to a model of a complete system.

Lemma 4.1.15 Let L and L' be finite closed altss such that $L \overset{ltl'}{\approx} L'$. Then $tr(L) = tr(L')$.

Proof: Suppose that $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ are closed altss such that $L \overset{ltl'}{\approx} L'$. Define $A'' = A \cup A'$. Notice that A'' is finite. Take any finite visible behaviour $\sigma = B_0 l_1 B_1 \ldots l_n B_n \in VB_*$. We have to show that $\sigma \in tr(L)$ iff $\sigma \in tr(L')$.

If σ contains a non-empty transition label $l_i \neq \tau$, then $\sigma \notin tr(L)$ and $\sigma \notin tr(L')$, since both L and L' are closed. Therefore, we can assume in the following that $l_i = \tau$ for all $1 \leq i \leq |\sigma|$.

We shall construct a particular LTL'-formula $ch(\sigma)$ such that for both $L^\dagger = L$ and $L^\dagger = L'$, the following holds:

$$L^\dagger \not\models \neg ch(\sigma) \;\text{ iff }\; \sigma \in tr(L^\dagger)$$

Once this has been shown, the result follows easily, as
$\sigma \in tr(L)$
iff $L \not\models \neg ch(\sigma)$
iff $L' \not\models \neg ch(\sigma)$ (since by assumption $L \overset{ltl'}{\approx} L'$)
iff $\sigma \in tr(L')$.

For every $B \subseteq A''$, define the characteristic formula of B by:

$$ch(B) \;=\; \bigwedge_{a \in B} a \;\wedge\; \bigwedge_{a \in A'' \setminus B} \neg a$$

Define the operator $\mathcal{U}'$ as a syntactic abbreviation: $\phi_1 \mathcal{U}' \phi_2 \;=\; \phi_1 \wedge (\phi_1 \mathcal{U} \phi_2)$. Now the characteristic formula of σ is

$$ch(\sigma) \;=\; ch(B_0)\,\mathcal{U}'(ch(B_1)\,\mathcal{U}'(\ldots(ch(B_{n-1})\,\mathcal{U}'ch(B_n))\ldots))$$

It should be easy to see that for all behaviours $\rho = B'_0 l'_1 B'_1 \ldots \in B_\infty$ for which each $l'_i = \tau$ and $B'_i \subseteq A''$, $\rho \models ch(\sigma)$ iff $\sigma \preceq vis(\rho)$. Notice also that any $\rho \in maxbeh(L) \cup maxbeh(L')$ fulfils these requirements.

This means that for both $L^\dagger = L$ and $L^\dagger = L'$,
$L^\dagger \not\models \neg ch(\sigma)$
iff there is a $\rho \in maxbeh(L^\dagger)$ such that $\rho \models ch(\sigma)$ (by Definition 3.2.3)
iff there is a $\rho \in maxbeh(L^\dagger)$ such that $\sigma \preceq vis(\rho)$ (by the observations above)
iff $\sigma \in tr(L^\dagger)$ (by Lemma 4.1.10).
This concludes the proof. □

Lemma 4.1.16 Let L and L' be finite closed altss such that $L \overset{ltl'}{\approx} L'$. Then $fmaxtr(L) = fmaxtr(L')$.

Proof: Up to the definition of $ch(\sigma)$, the proof proceeds word to word as in Lemma 4.1.15, by substituting *fmaxtr* for *tr* everywhere. This time the characteristic formula of σ is

$$ch(\sigma) \;=\; ch(B_0)\,\mathcal{U}'(ch(B_1)\,\mathcal{U}'(\ldots(ch(B_{n-1})\,\mathcal{U}'\Box ch(B_n))\ldots))$$

It should be easy to see that for all behaviours $\rho = B'_0 l'_1 B'_1 \ldots \in B_\infty$ for which each $l'_i = \tau$ and $B'_i \subseteq A''$, $\rho \models ch(\sigma)$ iff $\sigma = vis(\rho)$. Notice also that any $\rho \in maxbeh(L) \cup maxbeh(L')$ fulfils these requirements.

This means that for both $L^\dagger = L$ and $L^\dagger = L'$,
$L^\dagger \not\models \neg ch(\sigma)$
iff there is a $\rho \in maxbeh(L^\dagger)$ such that $\rho \models ch(\sigma)$ (by Definition 3.2.3)
iff there is a $\rho \in maxbeh(L^\dagger)$ such that $\sigma = vis(\rho)$ (by the observations above)
iff $\sigma \in fmaxtr(L^\dagger)$ (by Lemma 4.1.10 and the finiteness of σ).
As in the proof of 4.1.15, this proves the claim. □

Lemma 4.1.17 Let L and L' be finite altss such that $tr(L) = tr(L')$. Then $inftr(L) = inftr(L')$.

Proof: For any finite alts L, an infinite visible behaviour $\sigma \in VB_\omega$ is an infinite trace of L iff all its finite prefixes $\sigma' \prec \sigma$ are traces of L. In other words,

$$inftr(L) = \{\sigma \in VB_\omega \mid \text{for all } \sigma' \in VB_*, \text{ if } \sigma' \prec \sigma \text{ then } \sigma' \in tr(L)\}$$

Since by assumption $tr(L) = tr(L')$, this implies $inftr(L) = inftr(L')$. □

Theorem 4.1.18 Let L and L' be finite closed altss such that $L \overset{ltl'}{\approx} L'$. Then $L \overset{maxtr}{\approx} L'$.

Proof: Immediate from Lemmas 4.1.15, 4.1.16 and 4.1.17. □

Corollary 4.1.19 Let L and L' be finite closed altss. Then $L \overset{ltl'}{\approx} L'$ iff $L \overset{maxtr}{\approx} L'$.

Proof: Immediate from 4.1.12 and 4.1.18. □

It may be wondered where the requirement of the altss being closed is needed in this result. This has an easy explanation: As the logic LTL' can only describe properties of sequences of states, it has no way of distinguishing between altss that differ only with respect to the labels of some transitions.

We know now that for all altss $\stackrel{maxtr}{\approx}$ implies $\stackrel{ltl'}{\approx}$, and that for finite closed altss the converse holds as well. An interesting question not addressed yet is whether the converse holds for infinite closed altss. This issue will be discussed in Subsection 4.6.3.

4.2 NDFD-equivalence

On the basis of the previous section, we know that if we want to replace an alts L by a smaller L' in order to verify that $L \models \phi$, it is enough to find a smaller alts L' so that $L \stackrel{maxtr}{\approx} L'$. However, where the problem still lies in the whole approach is that it requires first constructing the alts $L = (P_1 \parallel \ldots \parallel P_n)$ to be used as the basis of constructing the smaller alts L'. But this is just what we are trying to avoid, since the size of L is prohibitively large. A far more effective way could be obtained if we could somehow take advantage of the structure of L, by operating directly on $P_1, \ldots, P_n$ when constructing L'. This leads to looking at equivalences that allow us to replace the components P_i by smaller ones before constructing the complete system from them, while preserving the validity or the invalidity of the required properties of the complete system. In other words, we would like to have an equivalence $\stackrel{x}{\approx}$ that implies $\stackrel{ltl'}{\approx}$ and is *compositional* in the sense of the following definition.

Definition 4.2.1 Let $\stackrel{x}{\approx}$ be an equivalence relation on the class of altss, and f an n-place syntactic operator constructing an alts from n component altss. We say that $\stackrel{x}{\approx}$ *is compositional with respect to* f iff for all altss $L_1, \ldots, L_n$, $L'_1, \ldots, L'_n$ such that for all $1 \leq i \leq n$, $L_i \stackrel{x}{\approx} L'_i$:

- $f(L_1, \ldots, L_n)$ is defined iff $f(L'_1, \ldots, L'_n)$ is defined, and
- if both are defined, $f(L_1, \ldots, L_n) \stackrel{x}{\approx} f(L'_1, \ldots, L'_n)$. □

The first candidate for $\stackrel{x}{\approx}$ would naturally be $\stackrel{maxtr}{\approx}$ itself. However, as the following example shows, this does not work, since $\stackrel{maxtr}{\approx}$ is not compositional with respect to the parallel composition operator $|[\ldots]|$. In a similar way we could see that it is not compositional with respect to $\|_v$, either.

Example 4.2.2 Let L_1, L_2 and L_3 be altss for which $AP(L_1) = AP(L_2) = AP(L_3) = \emptyset$, and for which S, I and Δ are described by the graphs in Figure 4.1.

Notice that $L_1 \stackrel{maxtr}{\approx} L_2$, since

$$maxtr(L_1) = maxtr(L_2) = \{\emptyset k \emptyset, \emptyset l \emptyset\}$$

However, $L_1 \,|[k,l]|\, L_3 \stackrel{maxtr}{\not\approx} L_2 \,|[k,l]|\, L_3$, since

$$\begin{array}{rcl} maxtr(L_1 \,|[k,l]|\, L_3) & = & \{\emptyset k \emptyset\} \\ maxtr(L_2 \,|[k,l]|\, L_3) & = & \{\emptyset k \emptyset, \emptyset\} \end{array}$$

□

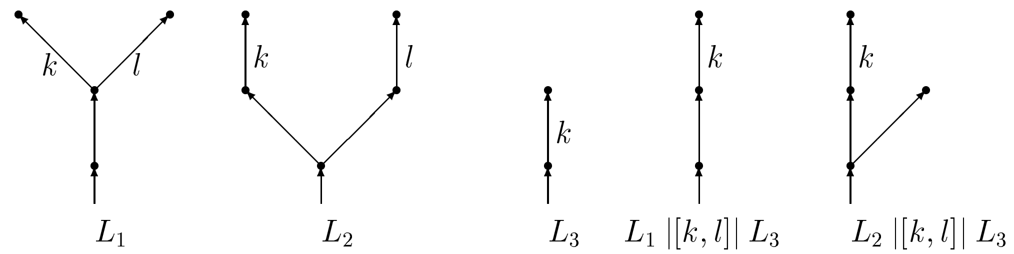


Figure 4.1: Noncompositionality of maximal traces

The reason for this noncompositionality is that the maximal traces do not alone convey information about the ways in which the execution of a program may be blocked. In the previous example, L_1 may be blocked only if the environment is willing to synchronise with neither k nor l, whereas L_2 may be blocked also in an environment that is willing to synchronise with only one of k or l. Adding this information to traces leads to the introduction of so-called *failures*.

A failure consists of a trace σ and a set of non-empty transition labels Π. The intuitive meaning of a failure is that the process can proceed from an initial state to a state s so that the visible behaviour corresponding to this execution path is σ, and the execution cannot proceed from s if the environment only allows the process to take transitions labelled with elements of Π. This intuition also naturally contains the idea that we do not need to be concerned about executions that cannot be blocked, i.e. leading to an *unstable* node that has τ-transition out from it. The intuition is captured by the following definition.

Definition 4.2.3 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts, $\sigma \in VB_*$ a finite visible behaviour, and $\Pi \subseteq \Sigma$ a set of non-empty transition labels. The ordered pair $(\sigma, \Pi) \in VB_* \times P(\Sigma)$ is *a stable failure of* L iff there is an $s \in I$ and an $s' \in S$ such that

- $s \stackrel{\sigma}{\Longrightarrow} s'$ and
- there is no $s'' \in S$ and $l \in \Pi \cup \{\tau\}$ such that $(s', l, s'') \in \Delta$.

The set of all stable failures of L is denoted by *sfail*(L). $\square$

The next candidate for the compositional semantic equivalence $\stackrel{x}{\approx}$ could consist of maximal traces and stable failures. Unfortunately, this equivalence is not compositional with respect to $|[\ldots]|$ either, as the following example shows.

Example 4.2.4 Let L_1, L_2 and L_3 be altss for which $AP(L_1) = AP(L_2) = AP(L_3) = \emptyset$, and for which S, I and Δ are described by graphs in Figure 4.2.

Notice that

$$\begin{aligned} maxtr(L_1) &= maxtr(L_2) &&= \{\emptyset k \emptyset\} \\ sfail(L_1) &= sfail(L_2) &&= \{(\emptyset, \Pi) \mid \Pi \subseteq \Sigma \setminus \{k\}\} \cup \{(\emptyset k \emptyset, \Pi) \mid \Pi \subseteq \Sigma\} \end{aligned}$$

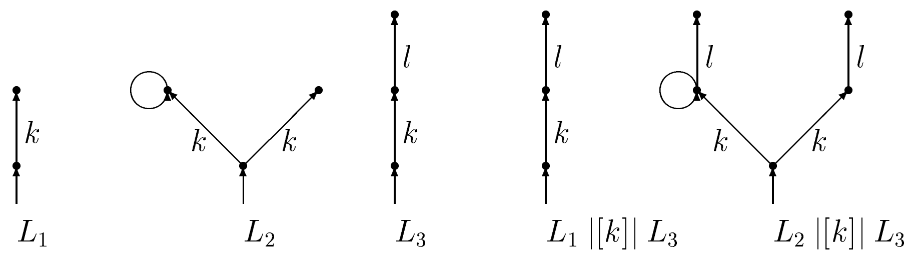


Figure 4.2: Noncompositionality of maximal traces and stable failures

However,

$$\begin{aligned} \mathit{maxtr}(L_1 \,|[k]|\, L_3) &= \{\emptyset k \emptyset l \emptyset\} \\ \mathit{maxtr}(L_2 \,|[k]|\, L_3) &= \{\emptyset k \emptyset l \emptyset, \emptyset k \emptyset\} \end{aligned}$$

i.e. $\mathit{maxtr}(L_1 \,|[k]|\, L_3) \neq \mathit{maxtr}(L_2 \,|[k]|\, L_3)$. □

The deficiency can be corrected by adding information about divergence traces into the model. This leads to a semantic model consisting of maximal traces, stable failures and divergence traces. However, after adding divergence traces to the model, there is no need to retain information about all stable failures any longer. Intuitively, a stable failure (σ, Π) tells that it is possible for the system to execute σ leading to a state from which the system does not take any visible actions in a certain kind of environment. But if σ is a divergence of the system, we already know that it is possible for the system to execute σ so that after σ the system does not take any visible actions, no matter what the environment is like then. In other words, if the trace part σ of a potential stable failure (σ, Π) is already a divergence trace, there is no need to record whether (σ, Π) is a stable failure of the alts or not. This leads to the notion of *non-divergent failures.*

Definition 4.2.5 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts, $\sigma \in VB_*$ a finite visible behaviour, and $\Pi \subseteq \Sigma$ a set of non-empty transition labels. The ordered pair $(\sigma, \Pi) \in VB_* \times P(\Sigma)$ is *a non-divergent failure of* L iff

- (σ, Π) is a stable failure of L, $(\sigma, \Pi) \in \mathit{sfail}(L)$, and
- σ is not a divergence trace of L, $\sigma \notin \mathit{divtr}(L)$.

The set of all non-divergent failures of L is denoted by $\mathit{ndfail}(L)$. □

This leads to a semantic model consisting of maximal traces, non-divergent failures and divergence traces. However, finite maximal traces turn out to be actually superfluous, as they are already determined by non-divergent failures and divergences, so the model only needs to record infinite maximal traces, i.e. infinite traces.

Lemma 4.2.6 $\mathit{fmaxtr}(L) = \{\sigma \in VB_* \mid (\sigma, \Sigma) \in \mathit{ndfail}(L)\} \cup \mathit{divtr}(L)$

Proof: Immediate from the definitions of *fmaxtr* (4.1.9), *ndfail* (4.2.5) and *divtr* (4.1.9). Notice that $\mathit{dltr}(L) = \{\sigma \in VB_* \mid (\sigma, \Sigma) \in \mathit{sfail}(L)\}$, and consequently $\mathit{dltr}(L) \setminus \mathit{divtr}(L) = \{\sigma \in VB_* \mid (\sigma, \Sigma) \in \mathit{ndfail}(L)\}$. □

All this results in a semantic model consisting of infinite traces, divergence traces and nondivergent failures. We give this a particular name: *non-divergent failures divergences equivalence* or *NDFD-equivalence*. For the relation of the following definition of NDFD-equivalence with the definitions in [KV92] and [VT95], see discussion in Subsection 4.6.2.

Definition 4.2.7 Let L_1 and L_2 be altss. We say that L_1 *and* L_2 *are non-divergent failures divergences equivalent* or *NDFD-equivalent* and write $L_1 \overset{\mathrm{ndfd}}{\approx} L_2$ iff

- $\Sigma(L_1) = \Sigma(L_2)$,
- $AP(L_1) = AP(L_2)$,
- $\mathit{inftr}(L_1) = \mathit{inftr}(L_2)$,
- $\mathit{divtr}(L_1) = \mathit{divtr}(L_2)$, and
- $\mathit{ndfail}(L_1) = \mathit{ndfail}(L_2)$. □

The only conditions in this definition that have not already been intuitively justified are the first two, $\Sigma(L_1) = \Sigma(L_2)$ and $AP(L_1) = AP(L_2)$. Essentially $\Sigma(L)$ and $AP(L)$, the sets of all transition labels and all atomic propositions that can possibly occur in the labels of L, can be seen as syntactic parameters constraining the possible behaviours that L can exhibit. The idea behind requiring $\Sigma(L_1) = \Sigma(L_2)$ and $AP(L_1) = AP(L_2)$ is that having the same syntactic parameters is a precondition to a meaningful comparison between two systems with respect to their behaviours.

When working towards the definition of NDFD-equivalence, we took care to preserve maximal traces at all stages. This means that all LTL'-properties are preserved by $\overset{\mathrm{ndfd}}{\approx}$, as well.

Lemma 4.2.8 Let L_1 and L_2 be altss such that $L_1 \overset{\mathrm{ndfd}}{\approx} L_2$. Then $L_1 \overset{\mathit{maxtr}}{\approx} L_2$.

Proof: Let $L_1 \overset{\mathrm{ndfd}}{\approx} L_2$. Then

$\mathit{fmaxtr}(L_1) =$

$\{\sigma \in VB_* \mid (\sigma, \Sigma) \in \mathit{ndfail}(L_1)\} \cup \mathit{divtr}(L_1) =$ (by Lemma 4.2.6)

$\{\sigma \in VB_* \mid (\sigma, \Sigma) \in \mathit{ndfail}(L_2)\} \cup \mathit{divtr}(L_2) =$ (by assumption)

$\mathit{fmaxtr}(L_2)$. (as above)

Consequently,
$\mathit{maxtr}(L_1) = \mathit{inftr}(L_1) \cup \mathit{fmaxtr}(L_1) = \mathit{inftr}(L_2) \cup \mathit{fmaxtr}(L_2) = \mathit{maxtr}(L_2)$. □

Proposition 4.2.9 Let L_1 and L_2 be altss such that $L_1 \overset{\mathrm{ndfd}}{\approx} L_2$. Then $L_1 \overset{\mathit{ltl}'}{\approx} L_2$ and $L_1 \overset{\mathit{tltl}'}{\approx} L_2$.

Proof: Immediate from Lemma 4.2.8 and Propositions 4.1.12 and 4.1.13. □

4.3 Compositionality

4.3.1 Parallel composition $|[\Pi]|$

In order to show that we have indeed reached our aim of compositionality, some technical definitions and lemmas are needed. The proof of compositionality here is based on the same approach and ideas as that in [VT91].

Let us first notice a very useful result: $\overset{\text{ndfd}}{\approx}$ preserves all traces.

Lemma 4.3.1 Let L_1 and L_2 be altss such that $L_1 \overset{\text{ndfd}}{\approx} L_2$. Then $tr(L_1) = tr(L_2)$.

Proof: Let $L_1 \overset{\text{ndfd}}{\approx} L_2$. Then
$tr(L_1) =$
$\{\sigma \in VB_* \mid \exists \sigma' \in maxtr(L_1) \text{ such that } \sigma \preceq \sigma'\} =$ (by Lemma 4.1.10)
$\{\sigma \in VB_* \mid \exists \sigma' \in maxtr(L_2) \text{ such that } \sigma \preceq \sigma'\} =$ (by Lemma 4.2.8)
$tr(L_2)$ (as above). □

Definition 4.3.2 Let $\sigma, \pi, \pi' \in VB_\infty$ be visible behaviours, $\Pi \subseteq \Sigma$, and write $\sigma = A_0 l_1 A_1 l_2 \ldots$, $\pi = B_0 k_1 B_1 k_2 \ldots$ and $\pi' = B'_0 k'_1 B'_1 k'_2 \ldots$ We say that σ *is a parallel composition of* π *and* π' *synchronised by* Π and write $par_s(\sigma, \pi, \pi', \Pi)$ iff there are surjective projection functions $p : \{i \in \mathbb{N} \mid i \leq |\sigma|\} \to \{i \in \mathbb{N} \mid i \leq |\pi|\}$ and $p' : \{i \in \mathbb{N} \mid i \leq |\sigma|\} \to \{i \in \mathbb{N} \mid i \leq |\pi'|\}$ such that

- $p(0) = p'(0) = 0$,
- for all $1 \leq i \leq |\sigma|$,
  - if $l_i \in \Pi$ then $p(i) = p(i-1) + 1$ and $p'(i) = p'(i-1) + 1$,
  - otherwise either $p(i) = p(i-1) + 1$ and $p'(i) = p'(i-1)$, or $p(i) = p(i-1)$ and $p'(i) = p'(i-1) + 1$,
- for all $0 \leq i \leq |\sigma|$, $A_i = B_{p(i)} \cup B'_{p'(i)}$, and
- for all $1 \leq i \leq |\sigma|$,
  - if $p(i) = p(i-1) + 1$ then $l_i = k_{p(i)}$, and
  - if $p'(i) = p'(i-1) + 1$ then $l_i = k'_{p'(i)}$. □

Lemma 4.3.3 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $A_1 \cap A_2 = \emptyset$, let $\Pi \subseteq \Sigma$, and let $s, s' \in S_1$, $t, t' \in S_2$. Notice that the pairs (s, t) and (s', t') are states of $L_1 \,|[\Pi]|\, L_2$. Then for all finite behaviours $\sigma \in VB_*$,

$$(s, t) \overset{\sigma}{\Longrightarrow} (s', t') \quad \text{in } L_1 \,|[\Pi]|\, L_2$$

iff there are $\pi, \pi' \in VB_*$ such that

- $s \overset{\pi}{\Longrightarrow} s'$ in L_1,
- $t \overset{\pi'}{\Longrightarrow} t'$ in L_2, and
- $par_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Since every execution path of $L_1|[\Pi]|L_2$ can be projected to execution paths of the components L_1 and L_2, it is easy to see that for any $s, s' \in S_1$, $t, t' \in S_2$ and any finite behaviour $\rho \in B_*$, $(s,t) \xrightarrow{\rho} (s',t')$ iff there are behaviours $\gamma, \gamma' \in B_*$ such that $s \xrightarrow{\gamma} s'$ in L_1, $t \xrightarrow{\gamma'} t'$ in L_2, and $par_s(\rho, \gamma, \gamma', \Pi)$ holds.

As by definition $(s,t) \stackrel{\sigma}{\Longrightarrow} (s',t')$ iff there is a $\rho \in B_*$ such that $(s,t) \xrightarrow{\rho} (s',t')$ and $vis(\rho) = \sigma$, this implies that $(s,t) \stackrel{\sigma}{\Longrightarrow} (s',t')$ iff there are $\rho, \gamma, \gamma' \in B_*$ such that $\sigma = vis(\rho)$, $s \xrightarrow{\gamma} s'$ in L_1, $t \xrightarrow{\gamma'} t'$ in L_2, and $par_s(\rho, \gamma, \gamma', \Pi)$ holds.

If such ρ, γ, γ' exist, then we have the required π and π' by $\pi = vis(\gamma)$, $\pi' = vis(\gamma')$. Notice that $par_s(\sigma, \pi, \pi', \Pi)$ holds, since a transition in ρ is visible iff the corresponding transition of γ or γ' is visible. On the other hand, if we have the required π and π', then there are some $\gamma, \gamma' \in B_*$ such that $s \xrightarrow{\gamma} s'$ in L_1, $t \xrightarrow{\gamma'} t'$ in L_2, $\pi = vis(\gamma)$ and $\pi' = vis(\gamma')$. Furthermore, as $par_s(\sigma, \pi, \pi', \Pi)$ holds, there is some ρ obtained by adding invisible transitions to σ for which $par_s(\rho, \gamma, \gamma', \Pi)$ holds. $\square$

Lemma 4.3.4 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $A_1 \cap A_2 = \emptyset$, let $\Pi \subseteq \Sigma$, and let $s \in S_1$, $t \in S_2$. Then for all behaviours $\sigma \in VB_\infty$,

$$(s,t) \stackrel{\sigma}{\Longrightarrow} \quad \text{in } L_1 \,|[\Pi]|\, L_2$$

iff there are $\pi, \pi' \in VB_\infty$ such that

- $s \stackrel{\pi}{\Longrightarrow}$ in L_1,
- $t \stackrel{\pi'}{\Longrightarrow}$ in L_2, and
- $par_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: As the proof of 4.3.3. $\square$

Lemma 4.3.5 Let L_1 and L_2 be altss such that $AP(L_1) \cap AP(L_2) = \emptyset$, and let $\Pi \subseteq \Sigma$. Then for all infinite behaviours $\sigma \in VB_\omega$,

$$\sigma \in inftr(L_1 \,|[\Pi]|\, L_2)$$

iff there are $\pi, \pi' \in VB_\infty$ such that

- $\pi \in tr(L_1) \cup inftr(L_1)$,
- $\pi' \in tr(L_2) \cup inftr(L_2)$, and
- $par_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Immediate from the definition of *inftr* (4.1.9) and Lemma 4.3.4. $\square$

Lemma 4.3.6 Let L_1 and L_2 be altss such that $AP(L_1) \cap AP(L_2) = \emptyset$, and let $\Pi \subseteq \Sigma$. Then for all finite behaviours $\sigma \in VB_*$,

$$\sigma \in divtr(L_1 \,|[\Pi]|\, L_2)$$

iff there are $\pi, \pi' \in VB_*$ such that

- $\pi \in tr(L_1)$,
- $\pi' \in tr(L_2)$,
- either $\pi \in divtr(L_1)$ or $\pi' \in divtr(L_2)$, and
- $par_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Suppose that there are π and π' fulfilling the above conditions. Since we know that $par_s(\sigma, \pi, \pi', \Pi)$ holds, the execution paths corresponding to π in L_1 and π' in L_2 can be combined in $L_1 \,|[\Pi]|\, L_2$ to produce a behaviour ρ such that $vis(\rho) = \sigma$. Since either the path corresponding to π or that corresponding to π' is infinite, ρ is infinite as well. But this means by definition of *divtr* (4.1.9) that $\sigma \in divtr(L_1 \,|[\Pi]|\, L_2)$.

Suppose then that $\sigma \in divtr(L_1 \,|[\Pi]|\, L_2)$. This means that there is an infinite path in $L_1 \,|[\Pi]|\, L_2$ corresponding to an infinite behaviour ρ such that $vis(\rho) = \sigma$. This path can be projected to a path of L_1 and a path of L_2. Define ρ_1 as the behaviour corresponding to the former path and ρ_2 to the latter, and define $\pi = vis(\rho_1)$ and $\pi' = vis(\rho_2)$. It is easy to see that $par_s(\sigma, \pi, \pi', \Pi)$ holds. Since an augmented transition in ρ is visible iff its projection in ρ_1 and ρ_2 is visible, and since ρ has only finitely many visible augmented transitions, we know that both π and π' are finite. Consequently, $\pi \in tr(L_1)$ and $\pi' \in tr(L_2)$. Furthermore, since ρ is infinite, either ρ_1 or ρ_2 is infinite, which means that either $\pi \in divtr(L_1)$ or $\pi' \in divtr(L_2)$. $\square$

Lemma 4.3.7 Let L_1 and L_2 be altss such that $AP(L_1) \cap AP(L_2) = \emptyset$, and let $\Pi \subseteq \Sigma$. Then for all finite behaviours $\sigma \in VB_*$ and transition sets $\Pi_f \subseteq \Sigma$,

$$(\sigma, \Pi_f) \in sfail(L_1 \,|[\Pi]|\, L_2)$$

iff there are $\pi, \pi' \in VB_*$ and $\Pi_{f1}, \Pi_{f2} \subseteq \Sigma$ such that

- $(\pi, \Pi_{f1}) \in sfail(L_1)$,
- $(\pi', \Pi_{f2}) \in sfail(L_2)$,
- $\Pi_f = ((\Pi_{f1} \cup \Pi_{f2}) \cap \Pi) \cup ((\Pi_{f1} \cap \Pi_{f2}) \setminus \Pi)$, and
- $par_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Let us write $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$, $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ and $L = (S, I, \Gamma, \Delta, A, V) = L_1 \,|[\Pi]|\, L_2$. Suppose that $(\sigma, \Pi_f) \in sfail(L)$. Then by definition of *sfail* (4.2.3), there are $(s,t) \in I$ and $(s',t') \in S$ such that $(s,t) \stackrel{\sigma}{\Longrightarrow} (s',t')$, and there are no $l \in \Pi_f \cup \{\tau\}$ and $(s'',t'') \in S$ such that $((s',t'), l, (s'',t'')) \in \Delta$ would hold. By Lemma 4.3.3 this means that there are $\pi, \pi' \in VB_*$ such that $s \stackrel{\pi}{\Longrightarrow} s'$ in L_1, $t \stackrel{\pi'}{\Longrightarrow} t'$ in L_2, and $par_s(\sigma, \pi, \pi', \Pi)$ holds. Since there are no $l \in \Pi_f \cup \{\tau\}$ and $(s'',t'') \in S$ such that $((s',t'), l, (s'',t'')) \in \Delta$ would hold, for every $l \in \Pi_f \cup \{\tau\}$ at least one of the following conditions must hold:

1 $l \in \Pi$ and there is no $s'' \in S_1$ such that $(s', l, s'') \in \Delta_1$

2 $l \in \Pi$ and there is no $t'' \in S_2$ such that $(t', l, t'') \in \Delta_2$

3 $l \notin \Pi$ and there is neither any $s'' \in S_1$ such that $(s', l, s'') \in \Delta_1$, nor any $t'' \in S_2$ such that $(t', l, t'') \in \Delta_2$.

Define now

$$\begin{aligned}\Pi_{f1} &= \{l \in \Pi_f \mid 1 \text{ or } 3 \text{ holds above}\}\\ \Pi_{f2} &= \{l \in \Pi_f \mid 2 \text{ or } 3 \text{ holds above}\}\end{aligned}$$

It is easy to see that $(\pi, \Pi_{f1}) \in \mathit{sfail}(L_1)$ and $(\pi', \Pi_{f2}) \in \mathit{sfail}(L_2)$. What is more

$$\begin{aligned}\Pi_f &= \{l \in \Pi_f \mid 1, 2 \text{ or } 3 \text{ holds above}\}\\ &= \{l \in \Pi_f \mid 1 \text{ or } 2 \text{ holds above}\} \cup \{l \in \Pi_f \mid 3 \text{ holds above}\}\\ &= ((\Pi_{f1} \cup \Pi_{f2}) \cap \Pi) \cup ((\Pi_{f1} \cap \Pi_{f2}) \setminus \Pi)\end{aligned}$$

Suppose then that the required $\pi, \pi', \Pi_{f1}, \Pi_{f2}$ exist. By definition of *sfail* (4.2.3), there are $s \in I_1$, $s' \in S_1$ such that $s \stackrel{\pi}{\Longrightarrow} s'$ and for all $l \in \Pi_{f1} \cup \{\tau\}$ and $s'' \in S_1$, $(s', l, s'') \notin \Delta_1$ Similarly, there are $t \in I_2$, $t' \in S_2$ such that $t \stackrel{\pi'}{\Longrightarrow} t'$ and for all $l \in \Pi_{f2} \cup \{\tau\}$ and $t'' \in S_2$, $(t', l, t'') \notin \Delta_2$. By Lemma 4.3.3 this means that $(s, t) \stackrel{\sigma}{\Longrightarrow} (s', t')$ in L.

By assumption, for every $l \in \Pi_f$, if $l \in \Pi$, then $l \in \Pi_{f1} \cup \Pi_{f2}$, and if $l \notin \Pi$, then $l \in \Pi_{f1} \cap \Pi_{f2}$. If $l \in \Pi_f$ and $l \in \Pi$, this means that there is either no $s'' \in S_1$ such that $(s', l, s'') \in \Delta_1$, or no $t'' \in S_2$ such that $(t', l, t'') \in \Delta_2$, which implies that there is no $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$. If $l \in \Pi_f$ but $l \notin \Pi$, this means that there is neither any $s'' \in S_1$ such that $(s', l, s'') \in \Delta_1$, nor any $t'' \in S_2$ such that $(t', l, t'') \in \Delta_2$, which implies that there is no $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$. The same holds also if $l = \tau$. Consequently, there are no $l \in \Pi_f \cup \{\tau\}$ and $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$ would hold, i.e. $(\sigma, \Pi_f) \in \mathit{sfail}(L)$. □

Lemma 4.3.8 Let L_1 and L_2 be altss such that $AP(L_1) \cap AP(L_2) = \emptyset$, and let $\Pi \subseteq \Sigma$. Then for all finite behaviours $\sigma \in VB_*$ and transition sets $\Pi_f \subseteq \Sigma$,

$$(\sigma, \Pi_f) \in \mathit{ndfail}(L_1 \mid[\Pi]\mid L_2)$$

iff $\sigma \notin \mathit{divtr}(L_1 \mid[\Pi]\mid L_2)$ and there are $\pi, \pi' \in VB_*$ and $\Pi_{f1}, \Pi_{f2} \subseteq \Sigma$ such that

- $(\pi, \Pi_{f1}) \in \mathit{ndfail}(L_1)$,
- $(\pi', \Pi_{f2}) \in \mathit{ndfail}(L_2)$,
- $\Pi_f = ((\Pi_{f1} \cup \Pi_{f2}) \cap \Pi) \cup ((\Pi_{f1} \cap \Pi_{f2}) \setminus \Pi)$, and
- $\mathit{par}_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Let us write $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$, $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ and $L = (S, I, \Gamma, \Delta, A, V) = L_1 \mid[\Pi]\mid L_2$. Suppose that $(\sigma, \Pi_f) \in \mathit{ndfail}(L)$. By definition this means that $\sigma \notin \mathit{divtr}(L)$ and $(\sigma, \Pi_f) \in \mathit{sfail}(L)$. The latter implies by Lemma 4.3.7 the existence of $\pi, \pi', \Pi_{f1}, \Pi_{f2}$ fulfilling the requirements in all the other respects, but $(\pi, \Pi_{f1}) \in \mathit{sfail}(L_1)$ instead of $\mathit{ndfail}(L_1)$ and similarly for (π', Π_{f2}). To justify $(\pi, \Pi_{f1}) \in \mathit{ndfail}(L_1)$ and $(\pi', \Pi_{f2}) \in \mathit{ndfail}(L_2)$, we need to show that $\pi \notin \mathit{divtr}(L_1)$

and $\pi' \notin \mathit{divtr}(L_2)$. Supposing the contrary, Lemma 4.3.6 implies that $\sigma \in \mathit{divtr}(L)$. But this is a contradiction, so the claim holds.

Suppose then that $\sigma \notin \mathit{divtr}(L)$ and that the required $\pi, \pi', \Pi_{f1}, \Pi_{f2}$ exist. Since $(\pi, \Pi_{f1}) \in \mathit{ndfail}(L_1)$ implies $(\pi, \Pi_{f1}) \in \mathit{sfail}(L_1)$, and similarly for (π', Π_{f2}), we know by Lemma 4.3.7 that $(\sigma, \Pi_f) \in \mathit{sfail}(L)$. But then by definition $(\sigma, \Pi_f) \in \mathit{ndfail}(L)$. □

Proposition 4.3.9 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \stackrel{\text{ndfd}}{\approx} L_1'$ and $L_2 \stackrel{\text{ndfd}}{\approx} L_2'$, and let $\Pi \subseteq \Sigma$. Then

- $L_1 \,\|[\Pi]\|\, L_2$ is defined iff $L_1' \,\|[\Pi]\|\, L_2'$ is defined, and
- if both are defined, then $L_1 \,\|[\Pi]\|\, L_2 \stackrel{\text{ndfd}}{\approx} L_1' \,\|[\Pi]\|\, L_2'$.

Proof: Since $L_1 \stackrel{\text{ndfd}}{\approx} L_1'$ and $L_2 \stackrel{\text{ndfd}}{\approx} L_2'$, clearly $AP(L_1) = AP(L_1')$ and $AP(L_2) = AP(L_2')$. Now $L_1 \,\|[\Pi]\|\, L_2$ is defined iff $AP(L_1) \cap AP(L_2) = \emptyset$ iff $AP(L_1') \cap AP(L_2') = \emptyset$ iff $L_1' \,\|[\Pi]\|\, L_2'$ is defined. Suppose then that both are defined. Notice first that

$$\Sigma(L_1 \,\|[\Pi]\|\, L_2) = \Sigma(L_1) \cup \Sigma(L_2) = \Sigma(L_1') \cup \Sigma(L_2') = \Sigma(L_1' \,\|[\Pi]\, L_2')$$

and

$$AP(L_1 \,\|[\Pi]\|\, L_2) = AP(L_1) \cup AP(L_2) = AP(L_1') \cup AP(L_2') = AP(L_1' \,\|[\Pi]\, L_2')$$

Since by the definition of $\stackrel{\text{ndfd}}{\approx}$ and Lemma 4.3.1

$$\begin{aligned} \mathit{tr}(L_1) &= \mathit{tr}(L_1') \\ \mathit{inftr}(L_1) &= \mathit{inftr}(L_1') \\ \mathit{divtr}(L_1) &= \mathit{divtr}(L_1') \end{aligned}$$

and similarly for L_2 and L_2', Lemma 4.3.5 implies that

$$\mathit{inftr}(L_1 \,\|[\Pi]\|\, L_2) = \mathit{inftr}(L_1' \,\|[\Pi]\|\, L_2')$$

and Lemma 4.3.6 that

$$\mathit{divtr}(L_1 \,\|[\Pi]\|\, L_2) = \mathit{divtr}(L_1' \,\|[\Pi]\|\, L_2')$$

Together with $\mathit{ndfail}(L_1) = \mathit{ndfail}(L_1')$ and $\mathit{ndfail}(L_2) = \mathit{ndfail}(L_2')$ this implies by Lemma 4.3.8 that

$$\mathit{ndfail}(L_1 \,\|[\Pi]\|\, L_2) = \mathit{ndfail}(L_1' \,\|[\Pi]\|\, L_2')$$

Consequently, $L_1 \,\|[\Pi]\|\, L_2 \stackrel{\text{ndfd}}{\approx} L_1' \,\|[\Pi]\|\, L_2'$. □

This result also implies easily the compositionality of $\stackrel{\text{ndfd}}{\approx}$ with respect to the derived operator $\|_s$.

Proposition 4.3.10 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \stackrel{\text{ndfd}}{\approx} L_1'$ and $L_2 \stackrel{\text{ndfd}}{\approx} L_2'$. Then

- $L_1 \|_s L_2$ is defined iff $L_1' \|_s L_2'$ is defined, and
- if both are defined, then $L_1 \|_s L_2 \stackrel{\text{ndfd}}{\approx} L_1' \|_s L_2'$.

Proof: Since $L_1 \stackrel{\text{ndfd}}{\approx} L_1'$ and $L_2 \stackrel{\text{ndfd}}{\approx} L_2'$ imply $\Sigma(L_1) = \Sigma(L_1')$ and $\Sigma(L_2) = \Sigma(L_2')$, the claim follows from Proposition 4.3.9, $L_1 \|_s L_2 = L_1 \,\|[\Sigma(L_1) \cap \Sigma(L_2)]\|\, L_2$ and $L_1' \|_s L_2' = L_1' \,\|[\Sigma(L_1') \cap \Sigma(L_2')]\|\, L_2'$. □

4.3.2 Parallel composition $\|_v$

In this subsection we show that the NDFD-equivalence is compositional with respect to the parallel composition $\|_v$ in which interaction takes place via shared variables. The proof generally follows the same lines as for $|[\Pi]|$, although many details are different.

Definition 4.3.11 Let $\sigma, \pi, \pi' \in VB_\infty$ be visible behaviours, and let $A_1, A_2 \subseteq AP$ such that $AP(\pi) \subseteq A_1$ and $AP(\pi') \subseteq A_2$. Let us write $\pi = B_0 k_1 B_1 k_2 \ldots$ and $\pi' = B'_0 k'_1 B'_1 k'_2 \ldots$ We say that σ *is a parallel composition of* π *and* π' *with respective local proposition sets* A_1 *and* A_2 and write $par_v(\sigma, \pi, \pi', A_1, A_2)$ iff there is a behaviour $\rho = A_0 l_1 A_1 l_2 \ldots$ and surjective projection functions $p : \{i \in \mathbb{N} \mid i \leq |\rho|\} \to \{i \in \mathbb{N} \mid i \leq |\pi|\}$ and $p' : \{i \in \mathbb{N} \mid i \leq |\rho|\} \to \{i \in \mathbb{N} \mid i \leq |\pi'|\}$ such that

- $\sigma = vis(\rho)$,
- $p(0) = p'(0) = 0$,
- for all $1 \leq i \leq |\rho|$,
  - either $p(i) = p(i-1) + 1$ and $p'(i) = p'(i-1)$,
  - or $p(i) = p(i-1)$ and $p'(i) = p'(i-1) + 1$,
- for all $0 \leq i \leq |\sigma|$, $A_i = B_{p(i)} \cup B'_{p'(i)}$, and
- for all $1 \leq i \leq |\sigma|$,
  - if $p(i) = p(i-1) + 1$ then $l_i = eval(A_2, B'_{p'(i-1)})(k_{p(i)}) \neq \bot$, and
  - if $p'(i) = p'(i-1) + 1$ then $l_i = eval(A_1, B_{p(i-1)})(k'_{p'(i)}) \neq \bot$. □

Lemma 4.3.12 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be condition-labelled altss such that $A_1 \cap A_2 = \emptyset$, and let $s, s' \in S_1$, $t, t' \in S_2$. Then for all finite behaviours $\sigma \in VB_*$,

$$(s,t) \stackrel{\sigma}{\Longrightarrow} (s',t') \quad \text{in } L_1 \|_v L_2$$

iff there are $\pi, \pi' \in VB_*$ such that

- $s \stackrel{\pi}{\Longrightarrow} s'$ in L_1,
- $t \stackrel{\pi'}{\Longrightarrow} t'$ in L_2, and
- $par_v(\sigma, \pi, \pi', A_1, A_2)$ holds.

Proof: As the proof of 4.3.3, except that this time (using the notation of that proof) a transition of ρ may be invisible, although the underlying transition of γ of γ' would be visible. See Example 4.3.16 for such a case. □

Lemma 4.3.13 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be condition-labelled altss such that $A_1 \cap A_2 = \emptyset$, and let $s \in S_1$, $t \in S_2$. Then for all behaviours $\sigma \in VB_\infty$,

$$(s,t) \stackrel{\sigma}{\Longrightarrow} \quad \text{in } L_1 \|_v L_2$$

iff there are $\pi, \pi' \in VB_\infty$ such that

- $s \stackrel{\pi}{\Longrightarrow}$ in L_1,
- $t \stackrel{\pi'}{\Longrightarrow}$ in L_2, and
- $par_v(\sigma, \pi, \pi', A_1, A_2)$ holds.

Proof: As the proof of 4.3.12. □

Lemma 4.3.14 Let L_1 and L_2 be condition-labelled altss s.t. $AP(L_1) \cap AP(L_2) = \emptyset$. Then for all infinite behaviours $\sigma \in VB_\omega$,

$$\sigma \in \mathit{inftr}(L_1 \parallel_v L_2)$$

iff there are $\pi, \pi' \in VB_\infty$ such that

- $\pi \in tr(L_1) \cup \mathit{inftr}(L_1)$,
- $\pi' \in tr(L_2) \cup \mathit{inftr}(L_2)$, and
- $par_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds.

Proof: Immediate from the definition of *inftr* (4.1.9) and Lemma 4.3.13. □

Lemma 4.3.15 Let L_1 and L_2 be condition-labelled altss s.t. $AP(L_1) \cap AP(L_2) = \emptyset$. Then for all finite behaviours $\sigma \in VB_*$,

$$\sigma \in \mathit{divtr}(L_1 \parallel_v L_2)$$

iff there are $\pi, \pi' \in VB_\infty$ such that

- $\pi \in tr(L_1) \cup \mathit{inftr}(L_1)$,
- $\pi' \in tr(L_2) \cup \mathit{inftr}(L_2)$,
- either $\pi \in \mathit{divtr}(L_1) \cup \mathit{inftr}(L_1)$ or $\pi' \in \mathit{divtr}(L_2) \cup \mathit{inftr}(L_2)$, and
- $par_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds.

Proof: Supposing that the required π and π' exist, we can show that $\sigma \in \mathit{divtr}(L_1 \parallel_v L_2)$ exactly as in the proof of Lemma 4.3.6. Suppose then that $\sigma \in \mathit{divtr}(L_1 \parallel_v L_2)$. This means that there is an infinite path in $L_1 \parallel_v L_2$ corresponding to an infinite behaviour ρ such that $vis(\rho) = \sigma$. This path can be projected to a path of L_1 and a path of L_2. Define ρ' as the behaviour corresponding to the former path and ρ'' to the latter, and define $\pi = vis(\rho')$ and $\pi' = vis(\rho'')$. It is easy to see that $par_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds, and that $\pi \in tr(L_1) \cup \mathit{inftr}(L_1)$, $\pi' \in tr(L_2) \cup \mathit{inftr}(L_2)$. Furthermore, since ρ is infinite, either ρ' or ρ'' is infinite, which means that either $\pi \in \mathit{divtr}(L_1) \cup \mathit{inftr}(L_1)$ or $\pi' \in \mathit{divtr}(L_2) \cup \mathit{inftr}(L_2)$. □

Notice that the alts $L_1 \parallel_v L_2$ may have a divergence trace even though neither L_1 nor L_2 has any. This is not possible in the case of the parallel composition by $|[\ldots]|$: either the projection of a divergence trace in $L_1 \,|[\Pi]|\, L_2$ to a trace of L_1 is a divergence of L_1 or that to a trace of L_2 is a divergence of L_2.

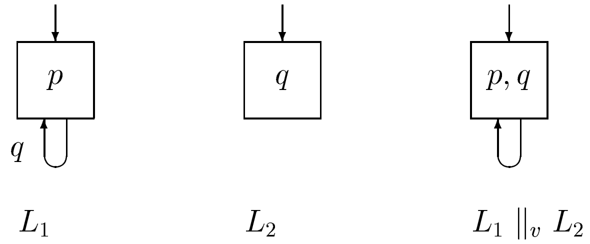


Figure 4.3: Divergences and parallel composition $\parallel_v$

Example 4.3.16 Let L_1 and L_2 be the altss for which $AP(L_1) = \{p\}$, $AP(L_2) = \{q\}$, and the transition graphs are described in Figure 4.3.

Now $\mathit{divtr}(L_1) = \mathit{divtr}(L_2) = \emptyset$, but $\mathit{divtr}(L_1 \parallel_v L_2) = \{\epsilon_{\{p,q\}}\}$ (i.e. a visible behaviour of length 0 consisting of just the initial truth set $\{p, q\}$). $\square$

Lemma 4.3.17 Let L_1 and L_2 be condition-labelled altss s.t. $AP(L_1) \cap AP(L_2) = \emptyset$. Then for all finite behaviours $\sigma \in VB_*$ and transition sets $\Pi \subseteq \Sigma$,

$$(\sigma, \Pi) \in \mathit{sfail}(L_1 \parallel_v L_2)$$

iff there are $\pi, \pi' \in VB_*$ such that

- $(\pi, \Pi_1) \in \mathit{sfail}(L_1)$,
- $(\pi', \Pi_2) \in \mathit{sfail}(L_2)$, and
- $\mathit{par}_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds,

where

- $\Pi_1 = \{l \in \Sigma_{cond}(AP \setminus AP(L_1)) \mid \mathit{eval}(AP(L_2), \pi'_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$, and
- $\Pi_2 = \{l \in \Sigma_{cond}(AP \setminus AP(L_2)) \mid \mathit{eval}(AP(L_1), \pi_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$.

Proof: Let us write $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$, $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ and $L = (S, I, \Gamma, \Delta, A, V) = L_1 \parallel_v L_2$. Suppose that $(\sigma, \Pi) \in \mathit{sfail}(L)$. By definition of sfail (4.2.3), there are $(s, t) \in I$ and $(s', t') \in S$ such that $(s, t) \stackrel{\sigma}{\Longrightarrow} (s', t')$, and there are no $l \in \Pi \cup \{\tau\}$ and $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$ would hold. By Lemma 4.3.12 this means that there are $\pi, \pi' \in VB_*$ such that $s \stackrel{\pi}{\Longrightarrow} s'$ in L_1, $t \stackrel{\pi'}{\Longrightarrow} t'$ in L_2, and $\mathit{par}_v(\sigma, \pi, \pi', A_1, A_2)$ holds. Notice that $V_1(s') = \pi_{\mathit{fin}}$ and $V_2(t') = \pi'_{\mathit{fin}}$. Since there are no $l \in \Pi \cup \{\tau\}$ and $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$ would hold, the following conditions must be true:

- If $l \in \Sigma_{cond}(AP \setminus A_1) \cup \{\tau\}$, i.e. if l is a potential transition condition of L_1, and if $\mathit{eval}(A_2, V_2(t'))(l) \in \Pi \cup \{\tau\}$, then for all $s'' \in S_1$, $(s', l, s'') \notin \Delta_1$.
- If $l \in \Sigma_{cond}(AP \setminus A_2) \cup \{\tau\}$, i.e. if l is a potential transition condition of L_2, and if $\mathit{eval}(A_1, V_1(s'))(l) \in \Pi \cup \{\tau\}$, then for all $t'' \in S_2$, $(t', l, t'') \notin \Delta_2$.

The first condition implies that $(\pi, \Pi_1) \in \mathit{sfail}(L_1)$, and the second condition that $(\pi', \Pi_2) \in \mathit{sfail}(L_2)$. Consequently, we have the required π and π'.

Suppose then that the required π and π' exist. By definition of *sfail* (4.2.3), there are $s \in I_1$, $s' \in S_1$ such that $s \stackrel{\pi}{\Longrightarrow} s'$ and for all $l \in \Pi_1 \cup \{\tau\}$ and $s'' \in S_1$, $(s', l, s'') \notin \Delta_1$. Similarly, there are $t \in I_2$, $t' \in S_2$ such that $t \stackrel{\pi'}{\Longrightarrow} t'$ and for all $l \in \Pi_2 \cup \{\tau\}$ and $t'' \in S_2$, $(t', l, t'') \notin \Delta_2$. By Lemma 4.3.12 this means that that $(s,t) \stackrel{\sigma}{\Longrightarrow} (s', t')$ in L. Notice again that $V_1(s') = \pi_{\mathit{fin}}$ and $V_2(t') = \pi'_{\mathit{fin}}$.

Take now any $l \in \Pi \cup \{\tau\}$. By the relation between Π and Π_1 and the observations in the previous paragraph, for every $l' \in \Sigma_{\mathit{cond}}(AP \setminus A_1) \cup \{\tau\}$ such that $\mathit{eval}(A_2, V_2(t'))(l') = l$ and every $s'' \in S_1$, we have $(s', l', s'') \notin \Delta_1$. Similarly, for every $l' \in \Sigma_{\mathit{cond}}(AP \setminus A_2) \cup \{\tau\}$ such that $\mathit{eval}(A_1, V_1(s'))(l') = l$ and every $t'' \in S_2$, we have $(t', l', t'') \notin \Delta_2$. Consequently, there is no $(s'', t'') \in S$ such that $((s', t'), l, (s'', t'')) \in \Delta$, implying that $(\sigma, \Pi) \in \mathit{sfail}(L)$. $\square$

Lemma 4.3.18 Let L_1 and L_2 be condition-labelled altss s.t. $AP(L_1) \cap AP(L_2) = \emptyset$. Then for all finite behaviours $\sigma \in VB_*$ and transition sets $\Pi \subseteq \Sigma$,

$$(\sigma, \Pi) \in \mathit{ndfail}(L_1 \parallel_v L_2)$$

iff $\sigma \notin \mathit{divtr}(L_1 \parallel_v L_2)$ and there are $\pi, \pi' \in VB_*$ such that

- $(\pi, \Pi_1) \in \mathit{ndfail}(L_1)$,
- $(\pi', \Pi_2) \in \mathit{ndfail}(L_2)$, and
- $\mathit{par}_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds,

where

- $\Pi_1 = \{l \in \Sigma_{\mathit{cond}}(AP \setminus AP(L_1)) \mid \mathit{eval}(AP(L_2), \pi'_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$, and
- $\Pi_2 = \{l \in \Sigma_{\mathit{cond}}(AP \setminus AP(L_2)) \mid \mathit{eval}(AP(L_1), \pi_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$.

Proof: Similar to the proof of Lemma 4.3.8, based on Lemmas 4.3.17 and 4.3.15. $\square$

Proposition 4.3.19 Let L_1, L'_1, L_2 and L'_2 be condition-labelled altss such that $L_1 \stackrel{\text{ndfd}}{\approx} L'_1$ and $L_2 \stackrel{\text{ndfd}}{\approx} L'_2$. Then

- $L_1 \parallel_v L_2$ is defined iff $L'_1 \parallel_v L'_2$ is defined, and
- if both are defined, then $L_1 \parallel_v L_2 \stackrel{\text{ndfd}}{\approx} L'_1 \parallel_v L'_2$.

Proof: With the exception of verifying that $\Sigma(L_1 \parallel_v L_2) = \Sigma(L'_1 \parallel_v L'_2)$, the proof is like that of Proposition 4.3.9, but instead of relying on Lemmas 4.3.5, 4.3.6 and 4.3.8, it relies on Lemmas 4.3.14, 4.3.15 and 4.3.18, respectively. To see $\Sigma(L_1 \parallel_v L_2) = \Sigma(L'_1 \parallel_v L'_2)$, notice that

$$\begin{aligned}
\Sigma(L_1 \parallel_v L_2) &= \{l \in \Sigma_{\mathit{cond}} \mid \exists l_1 \in \Sigma(L_1) \text{ and } B_2 \subseteq AP(L_2) : l = \mathit{eval}(AP(L_2), B_2)(l_1)\} \cup \\
&\quad\ \{l \in \Sigma_{\mathit{cond}} \mid \exists l_2 \in \Sigma(L_2) \text{ and } B_1 \subseteq AP(L_1) : l = \mathit{eval}(AP(L_1), B_1)(l_2)\} \\
&= \{l \in \Sigma_{\mathit{cond}} \mid \exists l_1 \in \Sigma(L'_1) \text{ and } B_2 \subseteq AP(L'_2) : l = \mathit{eval}(AP(L'_2), B_2)(l_1)\} \cup \\
&\quad\ \{l \in \Sigma_{\mathit{cond}} \mid \exists l_2 \in \Sigma(L'_2) \text{ and } B_1 \subseteq AP(L'_1) : l = \mathit{eval}(AP(L'_1), B_1)(l_2)\} \\
&= \Sigma(L'_1 \parallel_v L'_2)
\end{aligned}$$

$\square$

4.3.3 Other operators

This far, we have shown that $\overset{\text{ndfd}}{\approx}$ is compositional with respect to both forms of parallel composition. Showing that it is compositional with respect to the other operations, hiding, encapsulation and renaming, is more straightforward.

Proposition 4.3.20 Let L_1 and L_1' be altss such that $L_1 \overset{\text{ndfd}}{\approx} L_1'$, and let $\Pi \subseteq \Sigma$. Then

- both $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1$ and $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1'$ are defined, and
- $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1 \overset{\text{ndfd}}{\approx} \mathbf{hide}\ \Pi\ \mathbf{in}\ L_1'$.

Proof: The claim that $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1$ and $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1'$ are both defined is trivially true. It is also easy to see that $AP(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) = AP(L_1) = AP(L_1') = AP(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1')$, and $\Sigma(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) = \Sigma(L_1)\setminus\Pi = \Sigma(L_1')\setminus\Pi = \Sigma(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1')$. To see that the other requirements of $\overset{\text{ndfd}}{\approx}$ are fulfilled, let us define two auxiliary functions:

$$\begin{array}{rll} h: & \Sigma_\tau \to \Sigma_\tau; & l \mapsto \begin{cases} \tau & \text{if } l \in \Pi \\ l & \text{otherwise} \end{cases} \\ g: & VB_\infty \to VB_\infty; & \sigma \mapsto \mathit{vis}(h(\sigma)) \end{array}$$

where $h(\sigma)$ in the latter definition denotes the application of h to σ as defined in 2.3.7. Notice that

$$\begin{array}{rcl} \mathit{inftr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) & = & \{\sigma \in VB_\omega \mid \exists \sigma' \in \mathit{inftr}(L_1) : g(\sigma') = \sigma\} \\ \mathit{divtr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) & = & \{\sigma \in VB_* \mid \exists \sigma' \in \mathit{divtr}(L_1) \cup \mathit{inftr}(L_1) : g(\sigma') = \sigma\} \\ \mathit{ndfail}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) & = & \{(\sigma, \Pi_f) \in VB_* \times P(\Sigma) \mid \sigma \notin \mathit{divtr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) \text{ and} \\ & & \qquad \exists \sigma' \in VB_* : g(\sigma') = \sigma \text{ and} \\ & & \qquad (\sigma', \Pi_f \cup \Pi) \in \mathit{ndfail}(L_1)\} \end{array}$$

and that similar equations hold for $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1'$. Since $\mathit{inftr}(L_1) = \mathit{inftr}(L_1')$ and $\mathit{divtr}(L_1) = \mathit{divtr}(L_1')$ by assumption, the first two equations imply that

$$\begin{array}{rcl} \mathit{inftr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) & = & \mathit{inftr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1') \\ \mathit{divtr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) & = & \mathit{divtr}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1') \end{array}$$

This, together with the last equation and the assumption $\mathit{ndfail}(L_1) = \mathit{ndfail}(L_1')$, imply

$$\mathit{ndfail}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1) = \mathit{ndfail}(\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1')$$

Consequently, $\mathbf{hide}\ \Pi\ \mathbf{in}\ L_1 \overset{\text{ndfd}}{\approx} \mathbf{hide}\ \Pi\ \mathbf{in}\ L_1'$. □

Proposition 4.3.21 Let L_1 and L_1' be altss such that $L_1 \overset{\text{ndfd}}{\approx} L_1'$, and let $f : \Sigma \to \Sigma$ be a renaming function. Then

- both $L_1[f]$ and $L_1'[f]$ are defined, and
- $L_1[f] \overset{\text{ndfd}}{\approx} L_1'[f]$.

Proof: Like the proof of Lemma 4.3.20, except that this time

$$\begin{aligned}\Sigma(L_1[f]) &= \{l' \in \Sigma \mid \exists l \in \Sigma(L_1) : l' = f(l)\} \\ &= \{l' \in \Sigma \mid \exists l \in \Sigma(L_1') : l' = f(l)\} \ = \ \Sigma(L_1'[f])\end{aligned}$$

and the definition of the auxiliary function h and the equation for *ndfail* are:

$$\begin{aligned} h &: \Sigma_\tau \to \Sigma_\tau;\ l \mapsto \begin{cases} f(l) & \text{if } l \neq \tau \\ \tau & \text{if } l = \tau \end{cases} \\ \mathit{ndfail}(L_1[f]) &= \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \sigma \notin \mathit{divtr}(L_1[f]) \text{ and} \\ &\qquad \exists \sigma' \in VB_* \text{ such that } g(\sigma') = \sigma \text{ and} \\ &\qquad (\sigma, \{l \in \Sigma \mid f(l) \in \Pi\}) \in \mathit{ndfail}(L_1)\} \end{aligned}$$

□

Proposition 4.3.22 Let L_1 and L_1' be altss such that $L_1 \overset{\text{ndfd}}{\approx} L_1'$, and let $B \subseteq AP$. Then

- both **encaps** B **in** L_1 and **encaps** B **in** L_1' are defined, and
- **encaps** B **in** $L_1 \overset{\text{ndfd}}{\approx}$ **encaps** B **in** L_1'.

Proof: Like the proof of Lemma 4.3.20, except this time the definition for the auxiliary function h and the equation for *ndfail* are:

$$\begin{aligned} h &: P(AP) \to P(AP);\ A \mapsto A \setminus B \\ \mathit{ndfail}(\textbf{encaps } B \textbf{ in } L_1) &= \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \sigma \notin \mathit{divtr}(\textbf{encaps } B \textbf{ in } L_1) \text{ and} \\ &\qquad \exists \sigma' \in VB_* \text{ such that} \\ &\qquad g(\sigma') = \sigma \text{ and} \\ &\qquad (\sigma, \Pi) \in \mathit{ndfail}(L_1)\} \end{aligned}$$

□

We can now state the main result about the compositionality of NDFD-equivalence.

Theorem 4.3.23 The NDFD-equivalence $\overset{\text{ndfd}}{\approx}$ is compositional with respect to the operations:

- $\ldots |[\Pi]| \ldots$
- $\ldots \|_s \ldots$
- $\ldots \|_v \ldots$
- **hide** Π **in** $\ldots$
- $\ldots [f]$ and
- **encaps** B **in** $\ldots$

for all $\Pi \subseteq \Sigma$, $f : \Sigma \to \Sigma$ and $B \subseteq AP$.

Proof: This just summarises 4.3.9, 4.3.10, 4.3.19, 4.3.20, 4.3.21 and 4.3.22. □

This means that when constructing a system L from individual altss $L_1, \ldots, L_n$ by means of the above operators, we may freely replace any L_i or any intermediate alts L' by any NDFD-equivalent alts without affecting the validity of any LTL'-formula over the resulting system.

4.4 Full abstractness

What has been shown above is that the NDFD-equivalence can be used for compositional reductions when verifying LTL'-properties. However, there are other equivalences, e.g. the strong bisimulation equivalence [Mil89] (see Definition 7.1.1) that have the same property. These equivalences can be compared with each other by considering how much or how little superfluous information they preserve. The less extra information an equivalence pays attention to, the more models it considers equal and the better the chance of finding a small model equivalent to a given one is.

It turns out that the NDFD-equivalence is, in fact, optimal in this sense. On the one hand, it preserves all the characteristics that are needed for preserving LTL' and for compositionality, and on the other hand, all the characteristics it preserves are either required for preserving LTL' or for compositionality. Another way of expressing this is that $\stackrel{\text{ndfd}}{\approx}$ is the *weakest* equivalence that preserves LTL' and is compositional with respect to the syntactic operators, or that $\stackrel{\text{ndfd}}{\approx}$ is *fully abstract* with respect to preserving LTL'-properties in any environment. This means that for any other equivalence $\stackrel{x}{\approx}$ that has these properties, $L \stackrel{x}{\approx} L'$ implies $L \stackrel{\text{ndfd}}{\approx} L'$.

For this result we first need a small technical restriction. As explained before, in the definition of an alts the set of propositions of the alts is intuitively intended to correspond to a set of statements that are relevant to describing the observable characteristics of the system the alts is modelling. However, the definition currently overlooks one aspect, that for each system there are always some statements that are not relevant to it. For example, assuming that statements are used to describe the values of variables, and that there are an infinite number of potential variable names, then for any given system there are always variables that the system does not refer to, and consequently, propositions, viz. those referring to the unused variables, that cannot be meaningfully applied to describing the execution states of the system.

To capture this aspect we require that the set A of propositions relevant to a given system is a proper subset of AP. What is more, in order to guarantee that this property is preserved by the composition operators, we also require that the union of two such sets is still a proper subset of AP. We do not define exactly which subsets of AP could be relevant to some system. Given a fixed programming language and a syntax of atomic proposition names, such a definition would be straightforward. Instead, we just assume that the class of such subsets of AP is given and only require that this class fulfils some simple properties.

Definition 4.4.1 Fix some collection $\mathcal{A}$ of subsets of AP, $\mathcal{A} \subseteq P(AP)$, fulfilling the following properties

- if $A \subseteq AP$ is finite, then $A \in \mathcal{A}$,
- if $A \in \mathcal{A}$, then $A \neq AP$,
- if $A \in \mathcal{A}$ and $A' \subseteq A$, then $A' \in \mathcal{A}$, and
- if $A \in \mathcal{A}$ and $A' \in \mathcal{A}$, then $A \cup A' \in \mathcal{A}$.

We call a set $A \subseteq AP$ *admissible* iff $A \in \mathcal{A}$. □

For the rest of this section, we require that the set of atomic propositions of an alts is an admissible one.

Proving the full abstractness result is done by a series of lemmas, each of them pointing out some particular characteristics that a suitable equivalence should have. The proof here is based on and follows closely that presented in [KV92]. For several lemmas in the following, there are two versions of essentially the same result; one concerning equivalences having some characteristics when restricted to the class of *finite* altss, and another concerning equivalences having these characteristics generally, for all altss. To avoid unnecessarily re-stating the same results, this is indicated by inserting the parenthesised qualifier *finite* to the relevant parts of the claims and proofs. Each such claim contains actually two statements; one where all parenthesised occurrences of the word *finite* are included in the statement, and another where all these occurrences are ignored.

Lemma 4.4.2 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$ and $\mathbf{hide}\ \Sigma\ \mathbf{in}$, and
- for all (finite) closed altss L and L', $L \overset{x}{\approx} L'$ implies $tr(L) = tr(L')$.

Then for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $tr(L) = tr(L')$

Proof: Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be (finite) altss such that $L \overset{x}{\approx} L'$. We need to show that $tr(L) = tr(L')$.

Take any finite visible behaviour $\sigma \in VB_*$. If $AP(\sigma) \nsubseteq A \cup A'$, then clearly $\sigma \notin tr(L)$ and $\sigma \notin tr(L')$. Suppose then that $AP(\sigma) \subseteq A \cup A'$. We shall construct a finite alts L_σ and a finite visible behaviour $\sigma' \in VB_*$ such that for both $L^\dagger = L$ and $L^\dagger = L'$:

$$\sigma \in tr(L^\dagger) \text{ iff } \sigma' \in tr(\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Sigma]|\ L^\dagger))$$

Since $L \overset{x}{\approx} L'$, we have by compositionality of $\overset{x}{\approx}$ with respect to $|[\Sigma]|$ and $\mathbf{hide}\ \Sigma\ \mathbf{in}$:

$$\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Pi]|\ L) \overset{x}{\approx} \mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Pi]|\ L')$$

What is more, both $\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Pi]|\ L)$ and $\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Pi]|\ L')$ are closed (and finite, if L and L' are finite). Therefore, by assumption

$$\begin{aligned}
&\sigma \in tr(L) \text{ iff} \\
&\sigma' \in tr(\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Sigma]|\ L)) \text{ iff} \\
&\sigma' \in tr(\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma\ |[\Sigma]|\ L')) \text{ iff} \\
&\sigma \in tr(L')
\end{aligned}$$

This means that for all finite behaviours we know that $\sigma \in tr(L)$ iff $\sigma \in tr(L')$, i.e. $tr(L) = tr(L')$.

Let us now describe the alts L_σ and the behaviour $\sigma' \in VB_*$. Write first $\sigma = A_0 l_1 A_1 \ldots l_n A_n$. Let $i_1, \ldots, i_m$ be the subsequence of $1, \ldots, n$ such that $l_{i_1}, \ldots, l_{i_m}$ is the subsequence of $l_1, \ldots, l_n$ containing all the non-empty transition labels in it.

Take now some $a \in AP$ such that $a \notin A \cup A'$. Such an a exists, since both A and A' are admissible, therefore $A \cup A'$ is admissible, and as such $A \cup A' \neq AP$. Define

- for every $0 \leq k \leq m$, $B_k = \emptyset$ if k is even, and $B_k = \{a\}$, if k is odd,
- $\pi = B_0 l_{i_1} B_1 l_{i_2} B_2 \ldots l_{i_m} B_m$,
- $L_\sigma = (\{0, \ldots, m\}, \{0\}, \Sigma(\sigma), \Delta_\sigma, \{a\}, V_\sigma)$, where $\Delta_\sigma = \{(j-1, l_{i_j}, j) \mid 1 \leq j \leq m\}$, and $V_\sigma(j) = B_j$, for all $0 \leq j \leq m$,
- for all $0 \leq k \leq n$, $C_k = A_k \cup B_j$, where $j \in \{0, \ldots, m\}$ is defined by: either $j < m$ and $i_j \leq k < i_{j+1}$, or $j = m$ and $i_j \leq k$. Intuitively, C_k is similar to A_k, except for the atomic proposition a, which changes value from true to false or false to true each time the transition label l_k in not τ. This means that for all $1 \leq k \leq n$, $l_k = \tau$ iff $C_{k-1} \cap \{a\} = C_k \cap \{a\}$.
- $\sigma' = C_0 \tau C_1 \tau C_2 \ldots \tau C_n \in VB_*$.
  Notice that for every $1 \leq k \leq n$, either $l_k = \tau$, in which case $A_{k-1} \neq A_k$ since (l_k, A_k) is visible in A_{k-1}, or $l_k \neq \tau$, in which case $C_{k-1} \cap \{a\} \neq C_k \cap \{a\}$. In both cases $C_{k-1} \neq C_k$. This means that $\sigma' \in VB_*$, as stated.
- $\sigma'' = C_0 l_1 C_1 l_2 C_2 \ldots l_n C_n$.

We shall show next that for both $L^\dagger = L$ and $L^\dagger = L'$:

1 $\sigma \in \mathit{tr}(L^\dagger)$ iff $\sigma'' \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$

2 $\sigma'' \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$ iff $\sigma' \in \mathit{tr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L^\dagger))$.

These clearly imply the required $\sigma \in \mathit{tr}(L^\dagger)$ iff $\sigma' \in \mathit{tr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L^\dagger))$.

Suppose that $\sigma \in \mathit{tr}(L^\dagger)$. Combining the transition sequence corresponding to σ in $L^\dagger$ with the transition sequence corresponding to π in L_σ, and noticing that $L^\dagger$ and L_σ are synchronised with respect to all non-empty transition labels in $L_\sigma \,|[\Sigma]|\, L^\dagger$, we get a transition sequence corresponding to σ'' in $L_\sigma \,|[\Sigma]|\, L^\dagger$, i.e. $\sigma'' \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$.

Suppose then that $\sigma'' \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$. Projecting the transition sequence corresponding to σ'' in $L_\sigma |[\Sigma]| L^\dagger$ to the component alts $L^\dagger$, we get a transition sequence corresponding to $\sigma \in L^\dagger$, i.e. $\sigma \in \mathit{tr}(L^\dagger)$. Consequently, claim 1 above holds.

It is easy to see that if $\sigma'' \in \mathit{tr}(L_\sigma |[\Sigma]| L^\dagger)$, $\sigma' \in \mathit{tr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma |[\Sigma]| L^\dagger))$, i.e. claim 2 holds in one direction. Suppose finally that $\sigma' \in \mathit{tr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L^\dagger))$. This means that there is a $\rho \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$ such that when all transition labels in ρ are replaced by τ:s, the visible subsequence of the result is σ'. We know that the value of the proposition a changes between C_{k-1} and C_k in σ' iff the corresponding transition in ρ has arisen as a result of synchronisation between L_σ and $L^\dagger$, i.e. iff $l_k \neq \tau$. What is more, due to the structure of L_σ we know that all the non-empty transition labels in ρ form the same sequence as in π. But this means that there is only one possible ρ, that is σ''. Therefore, $\sigma'' \in \mathit{tr}(L_\sigma \,|[\Sigma]|\, L^\dagger)$, and the claim 2 above holds in the other direction, as well. □

Lemma 4.4.3 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$ and **hide** Σ **in** , and
- for all closed altss L and L', $L \overset{x}{\approx} L'$ implies $\mathit{inftr}(L) = \mathit{inftr}(L')$.

Then for all altss L and L', $L \overset{x}{\approx} L'$ implies $\mathit{inftr}(L) = \mathit{inftr}(L')$

Proof: As the proof of Lemma 4.4.2, except this time σ and L_σ are infinite. □

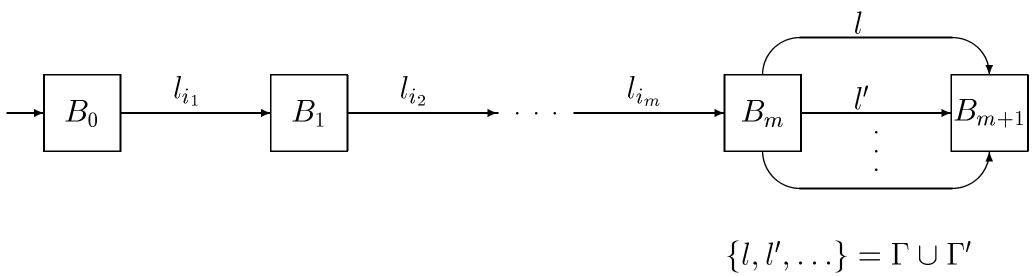


Figure 4.4: Alts L_σ

Notice that unlike Lemma 4.4.2, Lemma 4.4.3 does not say anything about the case where we know only that $L \overset{x}{\approx} L'$ implies $\mathit{inftr}(L) = \mathit{inftr}(L')$ for all *finite* closed altss L and L'. The reason for this is that the alts L_σ required in the proof is now infinite, and consequently we cannot make the implication from $\sigma' \in \mathit{inftr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L))$ to $\sigma' \in \mathit{inftr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L'))$ and back, if we only know that $L \overset{x}{\approx} L'$ implies $\mathit{inftr}(L) = \mathit{inftr}(L')$ for finite altss.

Lemma 4.4.4 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$ and **hide** Σ **in** , and
- for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$.

Then for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$

Proof: Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be (finite) altss such that $L \overset{x}{\approx} L'$. We need to show that $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$.

Take any finite visible behaviour $\sigma \in VB_*$. If $AP(\sigma) \not\subseteq A \cup A'$, then clearly $\sigma \notin \mathit{fmaxtr}(L)$ and $\sigma \notin \mathit{fmaxtr}(L')$. Suppose then that $AP(\sigma) \subseteq A \cup A'$. We shall construct an alts L_σ, which is finite if both L and L' are, and a finite visible behaviour $\sigma' \in VB_*$ such that for both $L^\dagger = L$ and $L^\dagger = L'$:

$$\sigma \in \mathit{fmaxtr}(L^\dagger) \text{ iff } \sigma' \in \mathit{fmaxtr}(\textbf{hide } \Sigma \textbf{ in } (L_\sigma \,|[\Sigma]|\, L^\dagger))$$

The main claim follows from this as in the proof of Lemma 4.4.2.

Write now $\sigma = A_0 l_1 A_1 \ldots l_n A_n$, and let $i_1, \ldots, i_m$ denote the subsequence of $1, \ldots, n$ such that $l_{i_1}, \ldots, l_{i_m}$ is the subsequence of $l_1, \ldots, l_n$ containing all the non-empty transition labels in it. Take some $a \in AP$ such that $a \notin A \cup A'$. Such an a exists as in the proof of Lemma 4.4.2. Define

- for every $0 \leq k \leq m+1$, $B_k = \emptyset$ if k is even, and $B_k = \{a\}$, if k is odd,
- $L_\sigma = (\{0, \ldots, m+1\}, \{0\}, \Sigma(\sigma) \cup \Gamma \cup \Gamma', \Delta_\sigma, \{a\}, V_\sigma)$, where

$$\begin{array}{rcl} \Delta_\sigma & = & \{(j-1, l_{i_j}, j) \mid 1 \leq j \leq m\} \cup (\{m\} \times (\Gamma \cup \Gamma') \times \{m+1\}) \\ V_\sigma & : & S \to P(A); i \mapsto B_i \end{array}$$

This is described by the graph in Figure 4.4. Notice that if L and L' are finite, then L_σ is finite, as well.

- for all $0 \leq k \leq n$, $C_k = A_k \cup B_j$, where $j \in \{0, \ldots, m\}$ is defined by: either $j < m$ and $i_j \leq k < i_{j+1}$, or $j = m$ and $i_j \leq k$.
- $\sigma' = C_0 \tau C_1 \tau C_2 \ldots \tau C_n \in VB_*$.
- $\sigma'' = C_0 l_1 C_1 l_2 C_2 \ldots l_n C_n$.

We shall show next that for both $L^\dagger = L$ and $L^\dagger = L'$:

1 $\sigma \in \mathit{fmaxtr}(L^\dagger)$ iff $\sigma'' \in \mathit{fmaxtr}(L_\sigma \;|[\Sigma]|\; L^\dagger)$

2 $\sigma'' \in \mathit{fmaxtr}(L_\sigma \;|[\Sigma]|\; L^\dagger)$ iff $\sigma' \in \mathit{fmaxtr}(\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma \;|[\Sigma]|\; L^\dagger))$.

From 1 and 2 the required

$$\sigma \in \mathit{fmaxtr}(L^\dagger) \text{ iff } \sigma' \in \mathit{fmaxtr}(\mathbf{hide}\ \Sigma\ \mathbf{in}\ (L_\sigma \;|[\Sigma]|\; L^\dagger))$$

follows easily.

Claim 2 is shown exactly as in the proof of Lemma 4.4.2. Let us concentrate on claim 1 here.

Suppose first that $\sigma \in \mathit{fmaxtr}(L^\dagger)$. This means that there is a transition sequence ts_1 corresponding to σ in $L^\dagger$, leading either to a deadlocking or a divergent node s of $L^\dagger$. Now L_σ has a transition sequence ts_2, leading to the node m from which all outgoing transitions are labelled with some $l \neq \tau$. Combining now ts_1 and ts_2 in $L_\sigma \;|[\Sigma]|\; L^\dagger$ yields a transition sequence ts leading to (m, s) and corresponding to σ''.

If s is divergent, then (m, s) is divergent, as well, and $\sigma'' \in \mathit{divtr}(L_\sigma|[\Sigma]|L^\dagger)$, implying that $\sigma'' \in \mathit{fmaxtr}(L_\sigma|[\Sigma]|L^\dagger)$. If s is deadlocking, then no transitions of $L^\dagger$ are possible from the node (m, s) of $L_\sigma \;|[\Sigma]|\; L^\dagger$. What is more, no transitions of L_σ are possible from (m, s) either, since all the transitions of L_σ going out from m are labelled with some $l \neq \tau$, and cannot be taken in $L_\sigma \;|[\Sigma]|\; L^\dagger$ without $L^\dagger$ taking a transition labelled with the same l at the same time. Consequently, in this case (m, s) is a deadlocking node, and $\sigma'' \in \mathit{fmaxtr}(L_\sigma \;|[\Sigma]|\; L^\dagger)$. This shows that the claim 1 above holds in one direction.

Suppose then that $\sigma'' \in \mathit{fmaxtr}(L_\sigma \;|[\Sigma]|\; L^\dagger)$. This means that there is a transition sequence ts of $L_\sigma \;|[\Sigma]|\; L^\dagger$ corresponding to σ'', leading to a node (t, s) which is either divergent or deadlocking. Projecting ts to the component alts $L^\dagger$ gives a transition sequence ts_1 of $L^\dagger$ corresponding to σ, leading to s. Similarly, projecting ts to L_σ gives a transition sequence ts_2 of L_σ, leading to node t. From the structure of L_σ, we can see that t must be m.

If $(t, s) = (m, s)$ is divergent in $L_\sigma \;|[\Sigma]|\; L^\dagger$, then s must be divergent in $L^\dagger$, since m in L_σ clearly is not divergent. In this case $\sigma \in \mathit{divtr}(L^\dagger) \subseteq \mathit{fmaxtr}(L^\dagger)$. Suppose then that (m, s) is deadlocking in $L_\sigma \;|[\Sigma]|\; L^\dagger$. This means that there cannot be any transition labelled with any $l \in \Gamma \cup \Gamma' \cup \{\tau\}$ that would be originating from s in $L^\dagger$, since otherwise there would be a transition originating from (m, s) in $L_\sigma \;|[\Sigma]|\; L^\dagger$. But for every $l \in \Sigma_\tau$, if there is some s' such that $(s, l, s') \in \Delta^\dagger$, then $l \in \Gamma \cup \Gamma' \cup \{\tau\}$. Therefore, s is a deadlocking node of $L^\dagger$ and $\sigma \in \mathit{fmaxtr}(L^\dagger)$. Consequently, claim 1 holds in the other direction, as well. $\square$

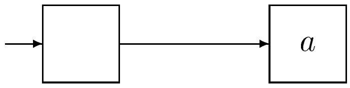


Figure 4.5: Alts L_{div}

Lemma 4.4.5 Let $\stackrel{x}{\approx}$ be an equivalence between altss, such that

- $\stackrel{x}{\approx}$ is compositional with respect to $||[\emptyset]||$, and
- for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$.

Then for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $\mathit{divtr}(L) = \mathit{divtr}(L')$

Proof: Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be (finite) altss such that $L \stackrel{x}{\approx} L'$. We need to show that $\mathit{divtr}(L) = \mathit{divtr}(L')$.

Take any finite visible behaviour $\sigma \in VB_*$. If $AP(\sigma) \not\subseteq A \cup A'$, then clearly $\sigma \notin \mathit{divtr}(L)$ and $\sigma \notin \mathit{divtr}(L')$. Suppose then that $AP(\sigma) \subseteq A \cup A'$. We shall construct a finite alts L_{div}, such that for both $L^\dagger = L$ and $L^\dagger = L'$:

$$\sigma \in \mathit{divtr}(L^\dagger) \text{ iff } \sigma \in \mathit{fmaxtr}(L_{div} \,||[\emptyset]||\, L^\dagger)$$

The main claim follows from this as in the proof of Lemma 4.4.2.

Take now some $a \in AP$ such that $a \notin A \cup A'$. Such an a exists as in the proof of Lemma 4.4.2. Define $L_{div} = (\{0,1\}, \{0\}, \emptyset, \{(0, \tau, 1)\}, \{a\}, V_{div})$, where $V_{div}(0) = \emptyset$ and $V_{div}(1) = \{a\}$, i.e. L_{div} is the alts described by the graph in Figure 4.5.

Suppose that $\sigma \in \mathit{divtr}(L^\dagger)$. This means that there is an infinite transition sequence ts_1 corresponding to σ in $L^\dagger$. Since $L^\dagger$ and L_{div} are not synchronised by any transition labels in $L_{div} \,||[\emptyset]\, L^\dagger$, there is a transition sequence ts in $L_{div} \,||[\emptyset]\, L^\dagger$ consisting of the sequence ts_1 of transitions of $L^\dagger$ and no transitions of L_{div}, i.e. a sequence where L_{div} does not ever proceed from its initial state. Since in the initial state of L_{div} no propositions are true, the behaviour corresponding to ts in $L_{div} \,||[\emptyset]||\, L^\dagger$ is σ. As ts_1 is infinite, ts is infinite, and by definition $\sigma \in \mathit{divtr}(L_{div} \,||[\emptyset]||\, L^\dagger) \subseteq \mathit{fmaxtr}(L_{div} \,||[\emptyset]||\, L^\dagger)$.

Suppose then that $\sigma \in \mathit{fmaxtr}(L_{div} \,||[\emptyset]||\, L^\dagger)$. This means that there is a transition sequence ts corresponding to σ in $L_{div} \,||[\emptyset]||\, L^\dagger$ that is either infinite or leads to a deadlocking node (s,t) of $L_{div} \,||[\emptyset]||\, L^\dagger$. Notice that since ts corresponds to σ, it cannot contain any transitions of L_{div}; if the only transition of L_{div} were included in ts, then we would have $a \in AP(\sigma)$, contrary to the assumption $AP(\sigma) \subseteq A \cup A'$. Therefore, ts is a transition sequence of $L^\dagger$. Furthermore, if ts were finite and leading to a deadlocking node (s,t) of $L_{div} \,||[\emptyset]||\, L^\dagger$, then s would be the initial node of L_{div}. But then (s,t) would not be deadlocking, since there is a transition of L_{div} labelled with τ out from this node. Consequently, ts must be infinite. But then by definition $\sigma \in \mathit{divtr}(L^\dagger)$, as required. $\square$

Lemma 4.4.6 Let $\stackrel{x}{\approx}$ be an equivalence between altss, such that for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$ and $\mathit{divtr}(L) = \mathit{divtr}(L')$. Then for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $nddltr(L) = nddltr(L')$

Proof: By Def. 4.1.9, for $L^\dagger = L$ and $L^\dagger = L'$, $nddltr(L^\dagger) = \mathit{fmaxtr}(L^\dagger) \setminus \mathit{divtr}(L^\dagger)$. $\square$

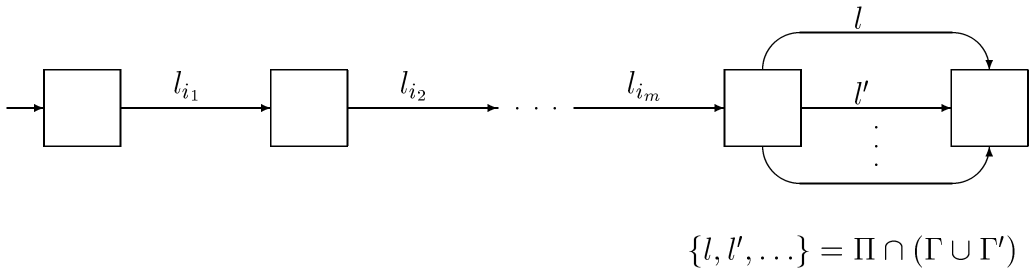


Figure 4.6: Alts $L_{\sigma\Pi}$

Lemma 4.4.7 Let $\stackrel{x}{\approx}$ be an equivalence between altss, such that

- $\stackrel{x}{\approx}$ is compositional with respect to $|[\Sigma]|$, and
- for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $\mathit{nddltr}(L) = \mathit{nddltr}(L')$.

Then for all (finite) altss L and L', $L \stackrel{x}{\approx} L'$ implies $\mathit{ndfail}(L) = \mathit{ndfail}(L')$

Proof: Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be (finite) altss such that $L \stackrel{x}{\approx} L'$. We need to show that $\mathit{ndfail}(L) = \mathit{ndfail}(L')$.

Take any finite visible behaviour $\sigma \in VB_*$, and a set of non-empty transition labels $\Pi \subseteq \Sigma$. We shall construct an alts $L_{\sigma\Pi}$, which is finite if both L and L' are, such that for both $L^\dagger = L$ and $L^\dagger = L'$:

$$(\sigma, \Pi) \in \mathit{ndfail}(L^\dagger) \text{ iff } \sigma \in \mathit{nddltr}(L^\dagger \,|[\Sigma]|\, L_{\sigma\Pi})$$

The main claim follows from this as in the proof of Lemma 4.4.2.

Write now $\sigma = A_0 l_1 A_1 \ldots l_n A_n$, and let $i_1, \ldots, i_m$ denote the subsequence of $1, \ldots, n$ such that $l_{i_1}, \ldots, l_{i_m}$ is the subsequence of $l_1, \ldots, l_n$ containing all the non-empty transition labels in it. Let us define $L_{\sigma\Pi} = (\{0, \ldots, m+1\}, \{0\}, \Sigma(\sigma) \cup (\Pi \cap (\Gamma \cup \Gamma')), \Delta_{\sigma\Pi}, \emptyset, V_{\sigma\Pi})$, where

$$\begin{array}{rcl} \Delta_{\sigma\Pi} & = & \{(j-1, l_{i_j}, j) \mid 1 \le j \le m\} \cup (\{m\} \times (\Pi \cap (\Gamma \cup \Gamma')) \times \{m+1\}) \\ V_{\sigma\Pi} & : & S \to P(A); i \mapsto \emptyset \end{array}$$

This is described by the graph in Figure 4.6.

If L and L' are finite, then $L_{\sigma\Pi}$ is finite, as well. Write $L^\dagger = (S^\dagger, I^\dagger, \Gamma^\dagger, \Delta^\dagger, A^\dagger, V^\dagger)$ and $L_{tot} = (S_{tot}, I_{tot}, \Gamma_{tot}, \Delta_{tot}, A_{tot}, V_{tot}) = L^\dagger \,|[\Sigma]|\, L_{\sigma\Pi}$. We would like to show that

$$(\sigma, \Pi) \in \mathit{ndfail}(L^\dagger) \text{ iff } \sigma \in \mathit{nddltr}(L_{tot})$$

Let us see first that $\sigma \in \mathit{divtr}(L^\dagger)$ iff $\sigma \in \mathit{divtr}(L_{tot})$. Since $L_{\sigma\Pi}$ does not have any infinite transition sequences, any infinite transition sequence ts of L_{tot} corresponding to σ can be projected to an infinite transition sequence ts_1 of $L^\dagger$. What is more, since all the transitions of $L_{\sigma\Pi}$ are synchronised with transitions of $L^\dagger$ on L_{tot}, and the set of atomic propositions of $L_{\sigma\Pi}$ is empty, the visible behaviour corresponding to ts_1 in $L^\dagger$ is σ, as well. Consequently, if $\sigma \in \mathit{divtr}(L_{tot})$ then $\sigma \in \mathit{divtr}(L^\dagger)$. Conversely, any infinite transition sequence ts_1 of $L^\dagger$ corresponding to σ can be combined with the transition sequence leading from 0 to m in $L_{\sigma\Pi}$ to yield an infinite transition sequence ts of L_{tot} corresponding to σ. Consequently, if $\sigma \in \mathit{divtr}(L^\dagger)$, then $\sigma \in \mathit{divtr}(L_{tot})$.

Suppose that $(\sigma, \Pi) \in \mathit{ndfail}(L^\dagger)$. Then $\sigma \notin \mathit{divtr}(L^\dagger)$, and by last paragraph $\sigma \notin \mathit{divtr}(L_{tot})$. Consequently, to show the claim in one direction, it is enough to show that $\sigma \in \mathit{dltr}(L_{tot})$, as then $\sigma \in \mathit{nddltr}(L_{tot})$, as required,

Since $(\sigma, \Pi) \in \mathit{ndfail}(L^\dagger) \subseteq \mathit{sfail}(L^\dagger)$, there is a transition sequence ts_1 in $L^\dagger$ corresponding to σ, leading to a node s such that for all $l \in \Pi \cup \{\tau\}$ and for all $s' \in S^\dagger$, $(s, l, s') \notin \Delta^\dagger$. Since the sequence of non-empty transition labels in σ is $l_{i_1}, \ldots, l_{i_m}$, we can combine ts_1 with the transition sequence ts_2 leading from 0 to m in $L_{\sigma\Pi}$ to yield a transition sequence ts of L_{tot}, corresponding to σ and leading to the node (s, m) of L_{tot}.

We would like to establish now that (s, m) is deadlocking in L_{tot}, i.e. that for all $l \in \Sigma \cup \{\tau\}$ and for all $(s', m') \in S_{tot}$, $((s, m), l, (s', m')) \notin \Delta_{tot}$.

Take first any $l \in \Pi$. We know that for all $s' \in S^\dagger$, $(s, l, s') \notin \Delta^\dagger$. Since all transitions of L_{tot} not labelled with τ are a result of synchronising a transition of $L^\dagger$ with a transition of $L_{\sigma\Pi}$, this implies that for all $(s', m') \in S_{tot}$, $((s, m), l, (s', m')) \notin \Delta_{tot}$. Take then any $l \in \Sigma \setminus \Pi$. This time, by definition of $\Delta_{\sigma\Pi}$, for all $m' \in S_{\sigma\Pi}$, $(m, l, m') \notin \Delta_{\sigma\Pi}$. As above, this implies that $((s, m), l, (s', m')) \notin \Delta_{tot}$, for all $(s', m') \in S_{tot}$. Finally, take $l = \tau$. As for all $s' \in S^\dagger$, $(s, \tau, s') \notin \Delta^\dagger$, and for all $m' \in S_{\sigma\Pi}$, $(m, \tau, m') \notin \Delta_{\sigma\Pi}$, we know that for all $(s', m') \in S_{tot}$, $((s, m), \tau, (s', m')) \notin \Delta_{tot}$. Consequently, we know that (s, m) is deadlocking and $\sigma \in \mathit{nddltr}(L_{tot})$. This completes the proof in one direction.

Suppose then $\sigma \in \mathit{nddltr}(L_{tot})$. Then $\sigma \notin \mathit{divtr}(L_{tot})$, and by the observations above $\sigma \notin \mathit{divtr}(L^\dagger)$. Consequently, to show the claim in the other direction, it is enough to show that $(\sigma, \Pi) \in \mathit{sfail}(L^\dagger)$, as then $(\sigma, \Pi) \in \mathit{ndfail}(L^\dagger)$, as required,

Since $\sigma \in \mathit{nddltr}(L_{tot}) \subseteq \mathit{dltr}(L_{tot})$, there is a transition sequence ts of L_{tot} corresponding to σ and leading to a deadlocking node (s, t) of L_{tot}. Projecting ts to the component alts $L^\dagger$, we get a transition sequence ts_1 of $L^\dagger$ leading to node s, and also corresponding to σ. We would like to establish now that for all $l \in \Pi \cup \{\tau\}$ and for all $s' \in S^\dagger$, $(s, l, s') \notin S^\dagger$, as then, by definition, we would have $(\sigma, \Pi) \in \mathit{sfail}(L^\dagger)$, as required.

Notice first that since $l_{i_1}, \ldots, l_{i_m}$ is the sequence of non-empty transition labels in σ, and since $L^\dagger$ and $L_{\sigma\Pi}$ are synchronised with respect to all non-empty transition labels, the only possible t in the node (s, t) the transition sequence ts leads to, is $t = m$.

Take first any $l \in \Pi \setminus (\Gamma \cup \Gamma')$. Then clearly $(s, l, s') \notin \Delta^\dagger$ for all $s' \in S^\dagger$. Take then any $l \in \Pi \cap (\Gamma \cup \Gamma')$. Since $(m, l, m + 1) \in \Delta_{\sigma\Pi}$, but there is no $(s', m') \in S_{tot}$ such that $((s, m), l, (s', m')) \in \Delta_{tot}$, there cannot be any $s' \in S^\dagger$ such that $(s, l, s') \in \Delta^\dagger$ holds. Finally, take $l = \tau$. Since there is no $(s', m') \in S_{tot}$ such that $((s, m), \tau, (s', m')) \in \Delta_{tot}$, there cannot be any $s' \in S^\dagger$ such that $(s, \tau, s') \in \Delta^\dagger$ holds. But then for all $l \in \Pi \cup \{\tau\}$ and all $s' \in S^\dagger$, $(s, l, s') \notin S^\dagger$, i.e. $(\sigma, \Pi) \in \mathit{sfail}(L^\dagger)$. This concludes the proof in the other direction. $\square$

Theorem 4.4.8 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$ and **hide** Σ **in** ,
- for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{maxtr}{\approx} L'$, and
- for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\mathrm{ndfd}}{\approx} L'$.

Proof: Since for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $maxtr(L) = maxtr(L')$, we know by Lemma 4.1.10 that for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $inftr(L) = inftr(L')$, $fmaxtr(L) = fmaxtr(L')$ and $tr(L) = tr(L')$.

Since for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $fmaxtr(L) = fmaxtr(L')$, we know by Lemma 4.4.4 that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $fmaxtr(L) = fmaxtr(L')$. By Lemma 4.4.5, this means that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $divtr(L) = divtr(L')$, and by Lemma 4.4.6 we see then that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $nddltr(L) = nddltr(L')$. Applying Lemma 4.4.7, this implies that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $ndfail(L) = ndfail(L')$.

This far we know that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $AP(L) = AP(L')$, $divtr(L) = divtr(L')$ and $ndfail(L) = ndfail(L')$. To see that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\mathrm{ndfd}}{\approx} L'$, we still need to show that for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $inftr(L) = inftr(L')$.

The method for showing this depends on whether we knew initially that $L \overset{x}{\approx} L'$ implies $L \overset{maxtr}{\approx} L'$ for all closed L and L' or only for all finite closed L and L'.

In the former case, we know now that for all closed L and L', $L \overset{x}{\approx} L'$ implies $inftr(L) = inftr(L')$. By Lemma 4.4.3, this means that for all L and L', $L \overset{x}{\approx} L'$ implies $inftr(L) = inftr(L')$, as required.

In the latter case, we know now that for all closed finite altss L and L', $L \overset{x}{\approx} L'$ implies $tr(L) = tr(L')$. By Lemma 4.4.2 this means that for all finite altss L and L', $L \overset{x}{\approx} L'$ implies $tr(L) = tr(L')$, which implies by Lemma 4.1.17 that for all finite altss L and L', $inftr(L) = inftr(L')$, as required. □

Corollary 4.4.9 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$ and **hide** Σ **in** ,
- for all closed finite altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{ltl'}{\approx} L'$, and
- for all finite altss L and L', $L \overset{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all finite altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\mathrm{ndfd}}{\approx} L'$.

Proof: Immediate from 4.1.18 and 4.4.8. □

The result shows that in the class of finite altss $\overset{\mathrm{ndfd}}{\approx}$ is precisely the equivalence characterised by its containing $\overset{ltl'}{\approx}$ and being compositional with respect to all the syntactic operators presented before, under the assumption that an equivalence should relate only systems having the same values for the syntactic parameters $\Sigma(L)$ and $AP(L)$.

An interesting question the previous proposition does not answer to, is whether $\overset{\mathrm{ndfd}}{\approx}$ is the equivalence characterised by its containing $\overset{ltl'}{\approx}$ and being compositional with respect

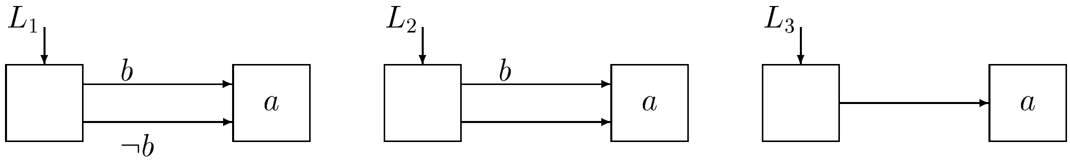


Figure 4.7: Indistinguishable altss 1

to the operators $\|_v$ and **encaps** B **in** . In other words, the question is whether $\overset{\text{ndfd}}{\approx}$ is optimal in a scenario where all the interaction between components of a system takes place by shared variables. As the following examples show, the answer to this question is negative; there are altss L and L' such that for all syntactic contexts $C[\cdot]$ and all LTL'-formulae ϕ, $C[L] \models \phi$ iff $C[L'] \models \phi$, but $L \overset{\text{ndfd}}{\not\approx} L'$. It appears to be possible to develop a semantic equivalence based on the intuitions behind $\overset{\text{ndfd}}{\approx}$ which would be precisely the equivalence characterised by containing $\overset{ltl'}{\approx}$ and being compositional with respect to $\|_v$ and **encaps** B **in** . However, this line of research was not pursued further here.

Example 4.4.10 Let L_1, L_2 and L_3 be altss for which $AP(L_1) = AP(L_2) = AP(L_3) = \{a\}$, and the transition graphs are described in Figure 4.7. Notice that

$$\begin{aligned}
\mathit{divtr}(L_1) &= \mathit{divtr}(L_2) = \mathit{divtr}(L_3) = \emptyset \\
\mathit{inftr}(L_1) &= \mathit{inftr}(L_2) = \mathit{inftr}(L_3) = \emptyset \\
\mathit{ndfail}(L_1) &= \{(\emptyset, \Pi) \mid \Pi \subseteq \Sigma \text{ and } b \notin \Pi \text{ and } \neg b \notin \Pi\} \cup \\
&\quad\ \{(\emptyset\, b\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\} \cup \\
&\quad\ \{(\emptyset\, \neg b\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\} \\
\mathit{ndfail}(L_2) &= \{(\emptyset\, \tau\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\} \cup \\
&\quad\ \{(\emptyset\, b\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\} \\
\mathit{ndfail}(L_3) &= \{(\emptyset\, \tau\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\}
\end{aligned}$$

However, L_1, L_2 and L_3 cannot be distinguished from each other by their external effects or by the way they depend from the environment in any context; in any situation a transition from the first node of the alts to the second can be taken, no matter what the value of the proposition b is. □

Example 4.4.11 Let L_1 and L_2 be altss for which $AP(L_1) = AP(L_2) = \{a\}$ and the transition graphs are described in Figure 4.8. Notice that

$$\begin{aligned}
\mathit{divtr}(L_1) &= \mathit{divtr}(L_2) = \emptyset \\
\mathit{inftr}(L_1) &= \mathit{inftr}(L_2) = \emptyset
\end{aligned}$$

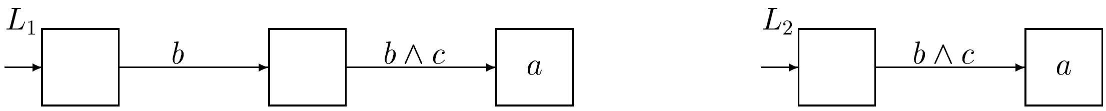


Figure 4.8: Indistinguishable altss 2

$$\begin{array}{rcl}
\mathit{ndfail}(L_1) & = & \{(\emptyset, \Pi) \mid \Pi \subseteq \Sigma \text{ and } b \notin \Pi\} \cup \\
& & \{(\emptyset\, b\, \emptyset, \Pi) \mid \Pi \subseteq \Sigma \text{ and } b \wedge c \notin \Pi\} \cup \\
& & \{(\emptyset\, b\, \emptyset\, b \wedge c\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\} \\
\mathit{ndfail}(L_2) & = & \{(\emptyset, \Pi) \mid \Pi \subseteq \Sigma \text{ and } b \wedge c \notin \Pi\} \cup \\
& & \{(\emptyset\, b \wedge c\, \{a\}, \Pi) \mid \Pi \subseteq \Sigma\}
\end{array}$$

However, in any environment in which the execution of L_1 may proceed from the initial node to the final one, so can the execution of L_2 and vice versa. □

4.5 Algorithms

This far we have concentrated on characterising an equivalence that can be applied in verification using compositional reductions. Let us look now at the algorithmic aspects of the issue: the complexity of determining whether two finite altss are NDFD-equivalent, and the complexity of minimising a finite alts with respect to the NDFD-equivalence. We show that both of these problems are PSPACE-complete.

Let us first see that both determining equivalence and minimising are PSPACE-hard. The proof of the PSPACE-hardness of determining NDFD-equivalence is similar to the proof presented by Kanellakis and Smolka in [KS90] in a related failure-based equivalence. Brooks and Rounds have also reported a similar result [BR83].

Theorem 4.5.1 Determining whether $L \overset{\text{ndfd}}{\approx} L'$ for given finite altss L and L' is a PSPACE-hard problem.

Proof: The proof works by reducing the following PSPACE-hard problem to the problem of determining NDFD-equivalence:

> Assume a fixed finite alphabet $\mathcal{A}$ such that $|\mathcal{A}| \geq 2$, and write $\mathcal{A}^*$ for the set of all finite $\mathcal{A}$-strings.
>
> Given a nondeterministic finite automaton (ndfa) L on $\mathcal{A}$-strings, determine whether $\mathcal{L}(L)$, the language recognised by L, is universal, i.e. whether $\mathcal{L}(L) = \mathcal{A}^*$.

This problem is known to be PSPACE-complete, and therefore PSPACE-hard [SM73] (see also [AHU74, Theorem 10.14] and [GJ78, Section 7.4]). It should be pointed out that this PSPACE-hardness, as stated e.g. in [GJ78], holds for automata without any kind of invisible transitions.

Using the terminology of the current work, we can view an ndfa as a restricted kind of alts with an acceptance condition, and rephrase the PSPACE -hard problem above as:

> Assume a fixed finite set of labels $\Pi \subseteq \Sigma$ such that $|\Pi| \geq 2$, and write
>
> $$\Pi^* \quad = \quad \{\sigma \in VB_* \mid AP(\sigma) = \emptyset \text{ and } \Sigma(\sigma) \subseteq \Pi\}$$
>
> Given a finite alts $L = (S, I, \Gamma, \Delta, A, V)$ and a set $F \subseteq S$, such that $A = \emptyset$, $\Gamma = \Pi$ and $\Delta \subseteq S \times \Pi \times S$, determine whether $acctr(L) = \Pi^*$ where
>
> $$acctr(L) \quad = \quad \{\sigma \in VB_* \mid \exists s \in I, s' \in F : s \stackrel{\sigma}{\Longrightarrow} s' \text{ in } L\}$$

We can reduce this problem to determining NDFD-equivalence as follows. Define first an alts $L^* = (S^*, I^*, \Gamma^*, \Delta^*, A^*, V^*)$ by

$$\begin{array}{rcl} S^* & = & \{0\} \\ I^* & = & S^* \\ \Gamma^* & = & \Pi \\ \Delta^* & = & S^* \times (\Pi \cup \{\tau\}) \times S^* \\ A^* & = & \emptyset \\ V^* & : & S^* \to P(A^*); s \mapsto \emptyset \end{array}$$

Notice that $tr(L^*) = divtr(L^*) = \Pi^*$ and $ndfail(L^*) = \emptyset$.

For any given alts L and set F as above, define an alts $L^\dagger$ by $L^\dagger = (S, I, \Gamma, \Delta^\dagger, A, V)$ where $\Delta^\dagger = \Delta \cup \{(s, \tau, s) \mid s \in F\}$.

We claim now that $acctr(L) = \Pi^*$ iff $L^\dagger \stackrel{\text{ndfd}}{\approx} L^*$. Assume first that $acctr(L) = \Pi^*$. As L does not have any τ-transitions, we have $divtr(L^\dagger) = acctr(L) = \Pi^*$. As $divtr(L^\dagger) \subseteq tr(L^\dagger) \subseteq \Pi^*$, we have also $tr(L^\dagger) = \Pi^*$. But $tr(L^\dagger) = divtr(L^\dagger)$ means that $ndfail(L^\dagger) = \emptyset$, i.e. $tr(L^\dagger) = divtr(L^\dagger) = \Pi^* = tr(L^*) = divtr(L^*)$ and $ndfail(L^\dagger) = \emptyset = ndfail(L^*)$, which imply $L^\dagger \stackrel{\text{ndfd}}{\approx} L^*$. Assume then that $L^\dagger \stackrel{\text{ndfd}}{\approx} L^*$. This implies $acctr(L) = divtr(L^\dagger) = divtr(L^*) = \Pi^*$, and the claim holds.

Finally, notice that the reduction is clearly polynomial. $\Box$

The most natural formulation for the minimisation problem is as a search problem:

> Given an alts L, find an alts L' such that
>
> - $L \stackrel{\text{ndfd}}{\approx} L'$ and
> - there is no alts L'' such that $L \stackrel{\text{ndfd}}{\approx} L''$ and L'' has fewer states than L' or L'' has as many states but fewer transitions than L'

However, as complexity classes are traditionally defined for decision problems, we reformulate the minimisation problem as the task of recognising a minimal NDFD-equivalent alts:

Given altss L and L', determine whether

- $L \overset{\text{ndfd}}{\approx} L'$ and
- there is no alts L'' such that $L \overset{\text{ndfd}}{\approx} L''$ and L'' has fewer states than L' or L'' has as many states but fewer transitions than L'

Notice that this decision problem is at least as hard as the search problem; if we are able to find a minimal NDFD-equivalent alts for a given alts L, then we can solve the decision problem by just comparing the sizes of this minimal alts and L'.

Theorem 4.5.2 Determining whether

- $L \overset{\text{ndfd}}{\approx} L'$ and
- there is no alts L'' such that $L \overset{\text{ndfd}}{\approx} L''$ and L'' has fewer states than L' or L'' has as many states but fewer transitions than L'

for given altss L and L' is a PSPACE-hard problem.

Proof: Like in the proof of 4.5.1, we can reduce the problem of universality of an ndfa to this problem of recognising a minimal NDFD-equivalent alts. For a given finite alts L and set F, construct $L^\dagger$ as in the proof of 4.5.1. Notice then that for the L^* described in the proof of 4.5.1 there clearly exists no L'' such that $L'' \overset{\text{ndfd}}{\approx} L^*$ and L'' has fewer states or transitions than L^*. Therefore, the question of whether $acctr(L) = \Pi^*$ can be answered by determining whether $L^\dagger \overset{\text{ndfd}}{\approx} L^*$ and there is no L'' such that $L'' \overset{\text{ndfd}}{\approx} L^\dagger$. This reduction is clearly polynomial. $\square$

Let us show then that determining NDFD-equivalence is in PSPACE. This is done by describing a nondeterministic polynomial space algorithm recognising non-equivalence, and applying Savitch's theorem saying PSPACE = NPSPACE [Sav70] (see also e.g. [BDG88, Cor. 2.1]). As a first step towards the algorithm, let us consider how to recognise $tr(L) \neq tr(L')$ for altss L and L'. Nondeterministically this can be done by guessing a behaviour $\rho \in beh(L)$ such that $vis(\rho) \notin tr(L')$.

Lemma 4.5.3 The algorithm *check trace* (L, L') in Figure 4.9 accepts iff $tr(L) \not\subseteq tr(L')$. Furthermore, the algorithm can be implemented in $O(|L'| + \log |L|)$ space.

Proof: Notice first that if $tr(L) \not\subseteq tr(L')$ then there exists a behaviour $\rho \in beh(L)$ such that $vis(\rho) \notin tr(L')$. To show the first claim, it is enough to show that if there exists any such ρ, then there exists one for which $|\rho| \leq |S| \cdot 2^{|S'|}$.

For this take any $\rho = A_0 l_1 A_1 \ldots l_n A_n$ such that $\rho \in beh(L)$ and $vis(\rho) \notin tr(L')$. Since $\rho \in beh(L)$, we have a sequence $s_0, s_1, \ldots, s_n \in S$ of states of L such that $s_0 \in I$ and for every $0 \leq i \leq n$, $V(s_i) = A_i$ and $(s_{i-1}, l_i, s_i) \in \Delta$. Define also a sequence $S'_0, S'_1, \ldots, S'_n \subseteq S'$ by: $S'_i = \{s' \in S' \mid \exists s \in I' : s \overset{\text{vis}(\rho[0\ldots i])}{\Longrightarrow} s'\}$. As $vis(\rho) \notin tr(L')$, we have $S'_n = \emptyset$.

check trace (L, L')

input: finite altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$

accept: iff $tr(L) \not\subseteq tr(L')$

begin
  if $I = \emptyset$, reject
  guess $n \in \{0, \ldots, |S| \cdot 2^{|S'|}\}$
  guess $s_0 \in I$
  $A_0 := V(s_0)$
  $S'_0 := \{s' \in S' \mid \exists t' \in I' : t' \stackrel{\epsilon A_0}{\Longrightarrow} s' \text{ in } L'\}$
  for $i := 1$ to n do
    guess $s_i \in S$, $l_i \in \Sigma(L) \cup \{\tau\}$
    if $(s_{i-1}, l_i, s_i) \notin \Delta$, reject
    $A_i := V(s_i)$
    $S'_i := \{s' \in S' \mid \exists t' \in S'_{i-1} : t' \stackrel{A_{i-1} l_i A_i}{\Longrightarrow} s' \text{ in } L'\}$
  end for
  if $S'_n \neq \emptyset$, reject
  accept
  /* Let $\rho = A_0 l_1 \ldots l_n A_n$. Now $vis(\rho) \in tr(L)$ but $vis(\rho) \notin tr(L')$ */
end

Figure 4.9: Checking trace non-containment

check divergent state (L, s_0)

input: finite alts $L = (S, I, \Gamma, \Delta, A, V)$ and state $s_0 \in S$

accept: iff s_0 divergent

begin
    for $i := 1$ to $|S|$ do
        guess $s_i \in S$ such that $(s_{i-1}, \tau, s_i) \in \Delta$ and $V(s_{i-1}) = V(s_i)$
        if no such s_i exists, reject
    end for
    accept
end

Figure 4.10: Checking state divergence

If there exist $0 \leq j < k \leq n$ such that $s_j = s_k$ and $S'_j = S'_k$, then it is easy to see that for the behaviour $\rho' = A_0 l_1 A_1 \ldots l_j A_j l_{k+1} A_{k+1} \ldots l_n A_n$ we have $\rho' \in tr(L)$ and $vis(\rho') \notin tr(L')$. This process of removing the segment from point j to k from σ can be repeated until there are no $0 \leq j < k \leq n$ such that $s_j = s_k$ and $S'_j = S'_k$. For the resulting behaviour ρ', we know that $\rho' \in beh(L)$ and $vis(\rho') \notin tr(L')$. Furthermore, as there are only $|S| \cdot 2^{|S'|}$ distinct choices for s_i and S'_i, we have $|\rho'| \leq |S| \cdot 2^{|S'|}$. Consequently, the first claim holds.

For the second claim, notice first that n and i can clearly be stored in $O(\log(|S| \cdot 2^{|S'|}))$ space, i.e. in $O(|S'| + \log |S|)$ space. Secondly, each s_i and l_i can be represented in $O(\log |L|)$ space, since there are only $|S|$ possible choices for s_i and $|\Delta|$ possible choices for l_i. Thirdly, the sets A_i do not need to be explicitly represented at all, since A_i is always determined by s_i and V. Fourthly, as each $S'_i \subseteq S'$, each S'_i can be stored in $O(|S'|)$ space. Fifthly, it is easy to see that each S'_i can be computed in $O(|L'|)$ space. Finally, notice that after s_i and S'_i are computed, no reference is made to s_{i-1}, l_i or S'_{i-1}, and the space used for storing these may be reused for storing s_{i+1}, l_{i+1} and S_{i+1}. Consequently, the algorithm can be implemented in $O(|L'| + \log |L|)$ space. $\square$

To check for trace non-equivalence of two altss L and L', we can nondeterministically choose to execute either *check trace* (L, L') or *check trace* (L', L). If either of them accepts, $tr(L) \neq tr(L')$.

A similar approach will be used for checking divergence trace non-containment. For this let us first see that recognising whether a state of an alts L is divergent or not can be done in nondeterministic $O(\log(|L|))$ space, or in deterministic $O(\log^2(|L|))$ space.

Lemma 4.5.4 The algorithm *check divergent state* (L, s_0) in Figure 4.10 accepts iff state s_0 in L is divergent, and can be implemented in $O(\log |L|)$ space. Furthermore, there exists a deterministic algorithm which accepts iff state s_0 in L is divergent and works in $O(\log^2 |L|)$ space.

check divergence trace (L, L')

input: finite altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$

accept: iff $divtr(L) \not\subseteq divtr(L')$

begin
 ...
 as in 'check trace' in Figure 4.9
 ...
 end for
 for every $s'_n \in S'_n$ do
  if s'_n is divergent in L', reject
  /* check this deterministically */
 end for
 if s_n is divergent in L then accept else reject
 /* check this nondeterministically */
 /* Now: $A_0 l_1 \dots l_n A_n \in divtr(L)$ but $A_0 l_1 \dots l_n A_n \notin divtr(L')$ */
end

Figure 4.11: Checking divergence trace non-containment

Proof: If s_0 is divergent, the algorithm in Figure 4.10 clearly has an accepting computation. On the other hand, if the algorithm has an accepting computation, then there is a sequence of states $s_0, s_1 \dots s_n$ with $n = |S|$, $V(s_i) = V(s_0)$ for all $0 \leq i \leq n$, and $(s_{i-1}, \tau, s_i) \in \Delta$ for all $1 \leq i \leq n$. As $n = |S|$, there must be some $1 \leq j < k \leq n$ such that $s_j = s_k$. But this means that we can construct an infinite sequence of invisible transitions starting from s_0, i.e. s_0 is divergent.

The algorithm can be implemented in $O(\log |L|)$ space, since i and each s_i can be stored in $O(\log |L|)$ space, and since the space used for storing s_{i-1} can be reused for storing s_{i+1}. The claim concerning the deterministic algorithm follows directly from Savitch's theorem [Sav70] (see also e.g. [BDG88, Cor. 2.1]). □

We can now apply this method of checking divergent states to obtain an algorithm for divergence trace non-containment from the algorithm for trace non-containment.

Lemma 4.5.5 The algorithm *check divergence trace* (L, L') in Figure 4.11 accepts iff $divtr(L) \not\subseteq divtr(L')$, and can be implemented in $O(|L'| + \log |L|)$ space.

Proof: As in the proof of 4.5.3, remembering that by 4.5.4 checking divergence of a state can be done nondeterministically for L in $O(\log |L|)$ space, and deterministically for L' in $O(\log^2 |L'|)$ space, i.e. in $O(|L'|)$ space. □

check failure (L, L')

input: finite altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$

accept: iff $\mathit{sfail}(L) \not\subseteq \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma))$

begin
  ...
  as in 'check trace' in Figure 4.9
  ...
  end for
  $\overline{\Pi} := \{l \in \Sigma_\tau \mid \exists s \in S : (s_n, l, s) \in \Delta\}$
  if $\tau \in \overline{\Pi}$, reject
  for every $s'_n \in S'_n$ do
    if s'_n is divergent in L', reject
    /* check this deterministically */
  end for
  for every $s'_n \in S'_n$ do
    $\overline{\Pi}' := \{l \in \Sigma_\tau \mid \exists s' \in S' : (s'_n, l, s') \in \Delta'\}$
    if $\overline{\Pi}' \subseteq \overline{\Pi}$, reject
    /* I.e. if $\Sigma \setminus \overline{\Pi} \subseteq \Sigma \setminus \overline{\Pi}'$, reject */
  end for
  accept
  /* Now: $(A_0 l_1 \ldots l_n A_n, \Sigma \setminus \overline{\Pi}) \in \mathit{sfail}(L)$ */
  /* but $(A_0 l_1 \ldots l_n A_n, \Sigma \setminus \overline{\Pi}) \notin \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma))$ */
end

Figure 4.12: Checking failure non-containment

check NDFD inequivalence (L, L')

input: finite altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$

accept: iff $L \overset{\text{ndfd}}{\not\approx} L'$

begin

choose nondeterministically to execute one of the following:

check trace (L, L')
check divergence trace (L, L')
check failure (L, L')

check trace (L', L)
check divergence trace (L', L)
check failure (L', L)

end

Figure 4.13: Checking NDFD inequivalence

In a similar way, we can modify the algorithm for trace non-containment to yield an algorithm for failure non-containment. This time the algorithm does not compare sets of nondivergent failures of L and L' directly, but the set of stable failures of L' to the sets of nondivergent failures and divergences of L'. However, we shall see shortly that it can still be used to check for NDFD-inequivalence.

Lemma 4.5.6 The algorithm *check failure* (L, L') in Figure 4.12 accepts iff $\mathit{sfail}(L) \not\subseteq \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma))$, and can be implemented in $O(|L'| + \log |L|)$ space.

Proof: The correctness of the algorithm is easy to show as in the proof of 4.5.3. To establish the space bound, notice that the set $\overline{\Pi}$ does not need to be represented explicitly, as it is fully determined by s_n and Δ. Checking whether $\tau \in \overline{\Pi}$ and whether $\overline{\Pi}' \subseteq \overline{\Pi}$ can be done deterministically by looping through all the transitions of L starting from s_n, which can be done in $O(\log |L|)$ space. □

We can now pull the pieces together to construct a nondeterministic algorithm recognising NDFD-inequivalence.

Lemma 4.5.7 The nondeterministic algorithm *check NDFD inequivalence* (L, L') in Figure 4.13 accepts iff $L \overset{\text{ndfd}}{\not\approx} L'$, and works in $O(n)$ space where $n = \max\{|L|, |L'|\}$.

Proof: If $L \overset{\text{ndfd}}{\approx} L'$, it is clear that $\mathit{tr}(L) = \mathit{tr}(L')$ and $\mathit{divtr}(L) = \mathit{divtr}(L')$. Moreover,

$divtr(L) = divtr(L')$ and $ndfail(L) = ndfail(L')$ imply

$$sfail(L) \subseteq ndfail(L) \cup (divtr(L) \times P(\Sigma)) = ndfail(L') \cup (divtr(L') \times P(\Sigma))$$

and vice versa. Consequently, if $L \overset{\text{ndfd}}{\approx} L'$ the algorithm rejects.

On the other hand, if the algorithm rejects, we have

$$\begin{aligned} tr(L) &= tr(L') \\ divtr(L) &= divtr(L') \\ sfail(L) &\subseteq ndfail(L') \cup (divtr(L') \times P(\Sigma)) \\ sfail(L') &\subseteq ndfail(L) \cup (divtr(L) \times P(\Sigma)) \end{aligned}$$

These imply $ndfail(L) = sfail(L) \setminus (divtr(L) \times P(\Sigma)) \subseteq ndfail(L')$ and vice versa, i.e. $ndfail(L') = ndfail(L)$. Consequently, if the algorithm rejects, we have $L \overset{\text{ndfd}}{\approx} L'$.

The space bound follows from 4.5.3, 4.5.5 and 4.5.6 and the fact that $O(n + \log n)$ is $O(n)$. □

By applying some elementary complexity theorem, we can now obtain deterministic algorithms for checking NDFD-equivalence.

Theorem 4.5.8 Determining whether $L \overset{\text{ndfd}}{\approx} L'$ for given finite altss L and L' can be done deterministically in $O(n^2)$ space and $O(2^{O(n^2)})$ time, where $n = \max\{|L|, |L'|\}$.

Proof: By 4.5.7 the nondeterministic algorithm *check NDFD inequivalence* (L, L') accepts iff $L \overset{\text{ndfd}}{\not\approx} L'$. Determinising this algorithm by Savitch's construction [Sav70] (see also e.g. [BDG88, Cor. 2.1]) yields a deterministic algorithm that accepts iff $L \overset{\text{ndfd}}{\not\approx} L'$ and works in $O(n^2)$ space, hence in $O(2^{O(n^2)})$ time. By reversing the accepting and rejecting states of this algorithm we have an algorithm that accepts iff $L \overset{\text{ndfd}}{\approx} L'$. □

Corollary 4.5.9 Determining whether $L \overset{\text{ndfd}}{\approx} L'$ for given finite altss L and L' is a PSPACE-complete problem.

Proof: Immediate from 4.5.1 and 4.5.8. □

Using a different determinisation construction, we can obtain an algorithm that is faster but uses more space.

Theorem 4.5.10 Determining whether $L \overset{\text{ndfd}}{\approx} L'$ for given finite altss L and L' can be done deterministically in $O(2^{O(n)})$ space and time, where $n = \max\{|L|, |L'|\}$.

Proof: Similar to the proof of 4.5.8, but using the construction of [BDG88, Theorem 2.8 (g)]. □

Let us show then that the problem of determining a minimal NDFD-equivalent alts is in PSPACE, as well. Notice that although the algorithm outlined in the following theorem is formulated for the decision problem, it can be trivially modified to solve the search problem of finding a minimal NDFD-equivalent alts for a given alts L.

Theorem 4.5.11 Determining whether

- $L \overset{\text{ndfd}}{\approx} L'$ and
- there is no alts L'' such that $L \overset{\text{ndfd}}{\approx} L''$ and L'' has fewer states than L' or L'' has as many states but fewer transitions than L'

for given altss L and L' can be done in polynomial space.

Proof: This problem can be solved deterministically by looping through all altss $L^\dagger = (S^\dagger, I^\dagger, \Gamma^\dagger, \Delta^\dagger, A^\dagger, V^\dagger)$ with $\Sigma(L^\dagger) = \Sigma(L)$, $AP(L^\dagger) = AP(L)$ and $|S^\dagger| \leq |S|$ in size order, starting from the smallest, terminating once an $L^\dagger$ with $L^\dagger \overset{\text{ndfd}}{\approx} L$ has been found, and comparing the sizes of this $L^\dagger$ and L'. Each $L^\dagger$ can be written down in space polynomial in $|L|$, and $L^\dagger \overset{\text{ndfd}}{\approx} L$ can be tested in polynomial space by 4.5.8. Finally, $L' \overset{\text{ndfd}}{\approx} L$ can also be tested in polynomial space by 4.5.8. Therefore, the algorithm works in polynomial space. □

Corollary 4.5.12 Determining whether

- $L \overset{\text{ndfd}}{\approx} L'$ and
- there is no alts L'' such that $L \overset{\text{ndfd}}{\approx} L''$ and L'' has fewer states than L' or L'' has as many states but fewer transitions than L'

for given altss L and L' is a PSPACE-complete problem.

Proof: Immediate from 4.5.2 and 4.5.11. □

Notice that in discussing minimisation of altss, we have not claimed that for an arbitrary alts L there would be a unique minimal NDFD-equivalent alts L'. In fact, this is not the case; there are systems that are NDFD-equivalent and minimal, but not isomorphic. For an example of such systems and for further analysis on minimal transition systems in general, see [Elo94].

Above we have presented the algorithms in a fairly abstract way. A more implementation-oriented view to these questions has been taken earlier in [VT91, KV91, KV92]. Let us now briefly outline the argument presented there.

From the discussion above it is easy to see that the problem of minimising an ndfa while preserving the language accepted by it is PSPACE-hard. However, when faced with this problem one can try to apply the heuristic of trying to produce the minimal deterministic finite automaton accepting the same language. This can be done by first

- determinising the automaton by the standard subset construction (see e.g. [HU79, pp. 22-23]), and then
- minimising the deterministic automaton by the functional coarsest partitioning method (see e.g. [AHU74, pp. 157-162]).

An analogous approach can be taken to the problem of producing a reduced alts L' such that $L \overset{\text{ndfd}}{\approx} L'$ for a given alts L. Some examples of this approach for related equivalences can be found in [BC89] and [CH90]. An algorithm based on this intuition for reducing ltss with respect to the CFFD-equivalence (see Definition 6.1.1) is described in detail in [VT91]. Modifications enabling this algorithm to be used for NDFD-equivalence, and with altss, are briefly outlined in [KV92].

The algorithm has a singly exponential worst-case complexity. However, practical results from a similar approach to the reduction problem of closely related equivalences, such as CFFD-equivalence in [VKCL93], or failure equivalence in [CPS90], provide some empirical evidence suggesting that in practice this exponential worst-case complexity is rarely a problem.

The same approach of determinisation and minimisation can also be used for checking the equivalence $L \overset{\text{ndfd}}{\approx} L'$ of two given altss L and L'. To do this, we apply the reduction procedure both to L and L' and check whether the results in both cases are the same. Without going into the details of this, the result produced by the reduction algorithm can be seen as a kind of a unique canonical representation of an alts, in the same way as the minimal dfa recognising a language can be seen as a canonical representation of the language.

4.6 Ramifications of the theory

In this section we shall look at some extensions of the main results of previous sections. First, in Subsection 4.6.1, we consider some additional composition operators. In addition to the parallel composition and hiding operators discussed above, most process algebras, e.g. CCS [Mil89], CSP [Hoa85] or LOTOS [BB89] have two further ways of composing processes: some form of sequential composition and a branching structure operator called *choice*. We show how these ways of composition can be incorporated into the current framework, and prove that the NDFD-equivalence is compositional with respect to all the new operators, except for one form of choice.

In Subsection 4.6.2 we discuss a way of refining the NDFD-equivalence so that it becomes compositional also with respect to this choice operator. The resulting finer equivalence is called the *(non-initial) non-divergent failures divergences equivalence* or the *NINDFD-equivalence*. We show that in the class of nomal labelled transition systems with unique initial states and no state labels, the NINDFD-equivalence coincides with the original definition of the NDFD-equivalence in [KV92, VT95].

Finally, in Subsection 4.6.3 we discuss extending the logic LTL' with an infinitary disjunction and conjunction, and show how this allows us to extend the full abstractness result of Corollary 4.4.9 from the class of finite altss to the class of all altss.

4.6.1 Other composition operators

4.6.1.1 Sequential composition

In this subsection we show how some forms of sequential composition can be incorporated into the current framework. This is done by introducing two new operators:

- *prefixing* $(B \xrightarrow{l}); L$, which takes an alts and adds a new transition before the initial states of L, and

- *sequential composition* $L_1 \gg L_2$, which takes two altss L_1 and L_2 and produces an alts behaving like L_1 until its successful completion, and like L_2 after that. Technically, this is done by introducing a special non-empty transition label δ reflecting the successful completion of the execution of an alts. Then in $L_1 \gg L_2$ all transitions in L_1 labelled with δ are replaced by a transition to an initial state of L_2.

These operators are closely based on corresponding ones in the process algebra LOTOS [BB89].

To make the definitions of this section easier, we use a technique of renaming the states of an alts, formulated in the following definition.

Definition 4.6.1 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts and f an injective function with domain S. Then $f(L)$ is the alts $f(L) = (S', I', \Gamma', \Delta', A', V')$ where

$$\begin{array}{rcl}
S' & = & \{f(s) \mid s \in S\} \\
I' & = & \{f(s) \mid s \in I\} \\
\Gamma' & = & \Gamma \\
\Delta' & = & \{(s_1', l, s_2') \in S' \times \Sigma_\tau \times S' \mid (f^{-1}(s_1), l, f^{-1}(s_2)) \in \Delta\} \\
A' & = & A \\
V' & : & S' \to P(A'); s \mapsto V(f^{-1}(s))
\end{array}$$

where f^{-1} is the inverse of f. □

An execution of $(B \xrightarrow{l}); L_1$ starts from a situation described by B, takes an initial step labelled by l, and then continues like some execution of L_1. An execution of $L_1 \gg L_2$ consists of a successful complete execution of L_1, followed by an execution of L_2.

Definition 4.6.2 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts, $l \in \Sigma_\tau$ a transition label, and $B \subseteq AP$ a set of atomic propositions.

Then $(B \xrightarrow{l}); L_1$ is the alts $(B \xrightarrow{l}); L' = (S', I', \Gamma', \Delta', A', V')$ where

$$\begin{array}{rcl}
S' & = & S \cup \{0\} \text{ where } 0 \notin S \\
I' & = & \{0\} \\
\Gamma' & = & \begin{cases} \Gamma & \text{if } l = \tau \\ \Gamma \cup \{l\} & \text{otherwise} \end{cases} \\
\Delta' & = & \Delta \cup \{(0, l, s) \mid s \in I\} \\
A' & = & A \cup B \\
V' & : & S' \to P(A); s \mapsto \begin{cases} V(s) & \text{if } s \in S \\ B & \text{if } s = 0 \end{cases}
\end{array}$$

□

Definition 4.6.3 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $S_1 \cap S_2 = \emptyset$. Then $L_1 \gg L_2$ is the alts $L_1 \gg L_2 = (S, I, \Gamma, \Delta, A, V)$, where

$$
\begin{array}{rcl}
S & = & S_1 \cup S_2 \\
I & = & I_1 \\
\Gamma & = & (\Gamma_1 \setminus \{\delta\}) \cup \Gamma_2 \\
\Delta & = & \{(s, l, s') \mid (s, l, s') \in \Delta_1 \text{ and } l \neq \delta\} \cup \\
 & & \{(s, \tau, s') \mid s \in S_1, s' \in I_2 \text{ and } \exists s'' \in S_1 : (s, \delta, s'') \in \Delta_1\} \cup \\
 & & \Delta_2 \\
A & = & A_1 \cup A_2 \\
V & : & S \to P(A); s \mapsto \begin{cases} V_1(s) & \text{if } s \in S_1 \\ V_2(s) & \text{if } s \in S_2 \end{cases}
\end{array}
$$

If L_1 and L_2 are altss such that $S_1 \cap S_2 \neq \emptyset$, $L_1 \gg L_2$ is defined as $f_1(L_1) \gg f_2(L_2)$ where f_1 and f_2 are the renaming functions $f_i : S_i \to S_i \times \{i\}; s \mapsto (s, i)$. $\square$

The NDFD-equivalence is compositional with respect to both of these operators

Lemma 4.6.4 Let L, L' be altss such that $L \overset{\text{ndfd}}{\approx} L'$, and let $B \subseteq AP$ and $l \in \Sigma_\tau$. Then both $(B \xrightarrow{l}); L$ and $(B \xrightarrow{l}); L'$ are defined, and $(B \xrightarrow{l}); L \overset{\text{ndfd}}{\approx} (B \xrightarrow{l}); L'$.

Proof: Notice that

$$
\begin{array}{rcl}
\Sigma((B \xrightarrow{l}); L) & = & \begin{cases} \Sigma(L) & \text{if } l = \tau \\ \Sigma(L) \cup \{l\} & \text{otherwise} \end{cases} \\
AP((B \xrightarrow{l}); L) & = & AP(L) \cup B \\
divtr((B \xrightarrow{l}); L) & = & \{\sigma \in VB_* \mid \exists \sigma' \in divtr(L) \text{ such that } \sigma = vis(Bl\sigma'_{init} \cdot \sigma')\} \\
inftr((B \xrightarrow{l}); L) & = & \{\sigma \in VB_\omega \mid \exists \sigma' \in inftr(L) \text{ such that } \sigma = vis(Bl\sigma'_{init} \cdot \sigma')\} \\
ndfail((B \xrightarrow{l}); L) & = & \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \ \exists(\sigma', \Pi) \in ndfail(L) \\
 & & \qquad \text{such that } \sigma = vis(Bl\sigma'_{init} \cdot \sigma')\} \cup \\
 & & \{(\varepsilon_B, \Pi) \in VB_* \times P(\Sigma) \mid l \notin \Pi \cup \{\tau\}\}
\end{array}
$$

and that similar equations hold for $(B \xrightarrow{l}); L'$. Since $\Sigma(L) = \Sigma(L')$, $AP(L) = AP(L')$, $divtr(L) = divtr(L')$, $inftr(L) = inftr(L')$ and $ndfail(L) = ndfail(L')$ by assumption, this implies that $(B \xrightarrow{l}); L \overset{\text{ndfd}}{\approx} (B \xrightarrow{l}); L'$ holds. $\square$

Lemma 4.6.5 Let L_1, L'_1, L_2, L'_2 be altss such that $L_1 \overset{\text{ndfd}}{\approx} L'_1$ and $L_2 \overset{\text{ndfd}}{\approx} L'_2$. Then both $L_1 \gg L_2$ and $L'_1 \gg L'_2$ are defined, and $L_1 \gg L_2 \overset{\text{ndfd}}{\approx} L'_1 \gg L'_2$.

Proof: Let us define for any alts L the set of successful complete traces:

$$
\begin{array}{rcl}
sctr(L) & = & \{\sigma \in VB_* \mid \delta \notin \Sigma(\sigma) \text{ and} \\
 & & \qquad \exists \pi \in tr(L) \text{ such that} \\
 & & \qquad |\pi| > 0 \text{ and} \\
 & & \qquad \pi_{|\pi|} = (\delta, A) \text{ for some } A \text{ and } \sigma = \pi[0 \ldots (|\pi| - 1)]\}
\end{array}
$$

Since $L_1 \overset{\text{ndfd}}{\approx} L_1'$, by Lemma 4.3.1 $tr(L_1) = tr(L_1')$ and by definition $sctr(L_1) = sctr(L_1')$.
It is trivial that both $L_1 \gg L_2$ and $L_1' \gg L_2'$ are defined. Notice then that

$$
\begin{aligned}
\Sigma(L_1 \gg L_2) &= (\Sigma(L_1) \setminus \{\delta\}) \cup \Sigma(L_2) \\
AP(L_1 \gg L_2) &= AP(L_1) \cup AP(L_2) \\
divtr(L_1 \gg L_2) &= \{\sigma \in divtr(L_1) \mid \delta \notin \Sigma(\sigma)\} \cup \\
&\quad\ \{\sigma \in VB_* \mid \exists \pi \in sctr(L_1), \pi' \in divtr(L_2) \text{ such that} \\
&\qquad\qquad \sigma = vis(\pi \cdot (\pi_{fin} \tau \pi'_{init}) \cdot \pi')\} \\
inftr(L_1 \gg L_2) &= \{\sigma \in inftr(L_1) \mid \delta \notin \Sigma(\sigma)\} \cup \\
&\quad\ \{\sigma \in VB_\omega \mid \exists \pi \in sctr(L_1), \pi' \in inftr(L_2) \text{ such that} \\
&\qquad\qquad \sigma = vis(\pi \cdot (\pi_{fin} \tau \pi'_{init}) \cdot \pi')\} \\
ndfail(L_1 \gg L_2) &= \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \delta \notin \Sigma(\sigma) \text{ and} \\
&\qquad\qquad (\sigma, \Pi \cup \{\delta\}) \in ndfail(L_1)\} \cup \\
&\quad\ \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \exists \pi \in sctr(L_1), (\pi', \Pi) \in ndfail(L_2) \\
&\qquad\qquad \text{such that} \\
&\qquad\qquad \sigma = vis(\pi \cdot (\pi_{fin} \tau \pi'_{init}) \cdot \pi')\}
\end{aligned}
$$

and that similar equations hold for $L_1' \gg L_2'$. By assumption $\Sigma(L_i) = \Sigma(L_i')$, $AP(L_i) = AP(L_i')$, $divtr(L_i) = divtr(L_i')$, $inftr(L_i) = inftr(L_i')$ and $ndfail(L_i) = ndfail(L_i')$ for both $i = 1$ and $i = 2$. From the above we also know that $sctr(L_1) = sctr(L_1')$. Consequently, $L_1 \gg L_2 \overset{\text{ndfd}}{\approx} L_1' \gg L_2'$. □

4.6.1.2 Choice

In this subsection, we introduce two versions of a nondeterministic choice operator, [:] and []. The first of these simply forms a disjoint union of two altss L_1 and L_2, keeping their initial states separate. The second also forms a disjoint union of two altss L_1 and L_2, but joins each pair of an initial state of L_1 and an initial state of L_2 having the same labels into a single initial state. One way of describing the difference between [:] and [] is that in L_1 [:] L_2 the decision between behaving like L_1 or like L_2 is made already in the initial state of an execution, whereas in L_1 [] L_2 the decision between L_1 and L_2 is made only when the first transition is taken. The more common one of these two versions of choice is [], which corresponds to the choice-operators of e.g. CCS and LOTOS. The other choice operator [:] would not be meaningful in the framework of these process algebras, where transition systems always have unique initial states.

Definition 4.6.6 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $S_1 \cap S_2 = \emptyset$. Then L_1 [:] L_2 is the alts L_1 [:] $L_2 = (S, I, \Gamma, \Delta, A, V)$ where

$$
\begin{aligned}
L &= S_1 \cup S_2 \\
I &= I_1 \cup I_2 \\
\Gamma &= \Gamma_1 \cup \Gamma_2 \\
\Delta &= \Delta_1 \cup \Delta_2 \\
A &= A_1 \cup A_2
\end{aligned}
$$

$$V \quad : \quad S \to P(A); s \mapsto \begin{cases} V_1(s) & \text{if } s \in S_1 \\ V_2(s) & \text{if } s \in S_2 \end{cases}$$

If L_1 and L_2 are altss such that $S_1 \cap S_2 \neq \emptyset$, $L_1[:]L_2$ is defined as $f_1(L_1)[:]f_2(L_2)$ where f_1 and f_2 are the renaming functions $f_i : S_i \to S_i \times \{i\}; s \mapsto (s, i)$. □

Definition 4.6.7 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $S_1 \cap S_2 = \emptyset$, and define $I_X = \{(s_1, s_2) \in I_1 \times I_2 \mid V_1(s_1) = V_2(s_2)\}$. Then $L_1 \,[]\, L_2$ is the alts $L_1 \,[]\, L_2 = (S, I, \Delta, A, V)$, where

$$\begin{array}{rcl} S & = & S_1 \cup S_2 \cup I_X \\ I & = & I_X \\ \Gamma & = & \Gamma_1 \cup \Gamma_2 \\ \Delta & = & \Delta_1 \cup \Delta_2 \cup \\ & & \{(s, l, s') \mid s = (s_1, s_2) \in I_X \text{ and } (s_1, l, s') \in \Delta_1\} \cup \\ & & \{(s, l, s') \mid s = (s_1, s_2) \in I_X \text{ and } (s_2, l, s') \in \Delta_2\} \\ A & = & A_1 \cup A_2 \\ V & : & S \to P(A); (s, i) \mapsto \begin{cases} V_1(s) & \text{if } s \in S_1 \\ V_2(s) & \text{if } s \in S_2 \\ V_1(s_1) & \text{if } s = (s_1, s_2) \in I_X \end{cases} \end{array}$$

If L_1 and L_2 are altss such that $S_1 \cap S_2 \neq \emptyset$, $L_1 \,[]\, L_2$ is defined as $f_1(L_1) \,[]\, f_2(L_2)$ where f_1 and f_2 are the renaming functions $f_i : S_i \to S_i \times \{i\}; s \mapsto (s, i)$. □

To clarify the relation between [:] and [] operators, notice that for all altss L_1 and L_2,

$$(B \xrightarrow{l}); (L_1 \,[:]\, L_2) = (B \xrightarrow{l}); L_1 \,[]\, (B \xrightarrow{l}); L_2$$

for all $B \subseteq AP$ and $l \in \Sigma_\tau$. Let us see then that $\overset{\text{ndfd}}{\approx}$ is compositional with respect to [:].

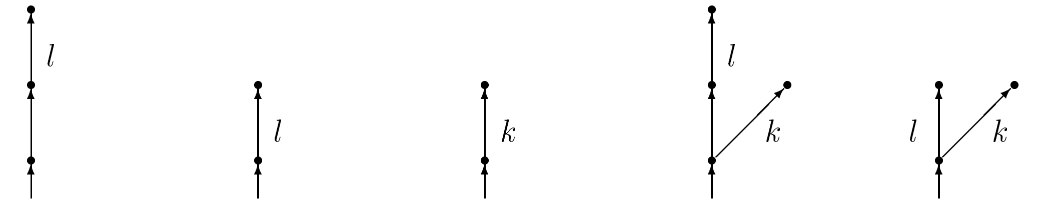
Lemma 4.6.8 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \overset{\text{ndfd}}{\approx} L_1'$ and $L_2 \overset{\text{ndfd}}{\approx} L_2'$. Then

- both $L_1 \,[:]\, L_2$ and $L_1' \,[:]\, L_2'$ are defined, and
- $L_1 \,[:]\, L_2 \overset{\text{ndfd}}{\approx} L_1' \,[:]\, L_2'$.

Proof: It is easy to see that

$$\begin{array}{rcl} \Sigma(L_1 \,[:]\, L_2) & = & \Sigma(L_1) \cup \Sigma(L_2) \\ AP(L_1 \,[:]\, L_2) & = & AP(L_1) \cup AP(L_2) \\ \mathit{divtr}(L_1 \,[:]\, L_2) & = & \mathit{divtr}(L_1) \cup \mathit{divtr}(L_2) \\ \mathit{inftr}(L_1 \,[:]\, L_2) & = & \mathit{inftr}(L_1) \cup \mathit{inftr}(L_2) \\ \mathit{ndfail}(L_1 \,[:]\, L_2) & = & (\mathit{ndfail}(L_1) \cup \mathit{ndfail}(L_2)) \setminus \mathit{divtr}(L_1 \,[:]\, L_2) \times P(\Sigma) \end{array}$$

and that similar equations hold for $L_1' \,[:]\, L_2'$. Since by assumption $L_1 \overset{\text{ndfd}}{\approx} L_1'$ and $L_1 \overset{\text{ndfd}}{\approx} L_2'$, $L_1 \,[:]\, L_2 \overset{\text{ndfd}}{\approx} L_1' \,[:]\, L_2'$. □



L_1 L_2 L_3 L_1 [] L_3 L_2 [] L_3

Figure 4.14: Noncompositionality of NDFD-equivalence w.r.t. choice []

However, as the following example shows, it turns out that $\overset{\text{ndfd}}{\approx}$ is *not* compositional with respect to the second choice-operator []. In next section we shall show how to modify NDFD-equivalence to solve this problem.

Example 4.6.9 Let L_1, L_2 and L_3 be altss for which $AP(L_1) = AP(L_2) = AP(L_3) = \emptyset$ and the transition graphs are described in Figure 4.14. Notice that

$$\begin{array}{rrl} divtr(L_1) = & divtr(L_2) = & \emptyset \\ inftr(L_1) = & inftr(L_2) = & \emptyset \\ ndfail(L_1) = & ndfail(L_2) = & \{(\emptyset, \Pi) \mid \Pi \subseteq \Sigma \text{ and } l \notin \Pi\} \cup \{(\emptyset l \emptyset, \Pi) \mid \Pi \subseteq \Sigma\} \end{array}$$

i.e. $L_1 \overset{\text{ndfd}}{\approx} L_2$. However,

$$\begin{array}{rcl} (\emptyset, \{k\}) & \in & ndfail(L_1 \,[]\, L_3) \\ (\emptyset, \{k\}) & \notin & ndfail(L_2 \,[]\, L_3) \end{array}$$

i.e. $L_1 \,[]\, L_3 \overset{\text{ndfd}}{\not\approx} L_2 \,[]\, L_3$. □

4.6.2 NINDFD-equivalence

Intuitively, the reason for the non-compositionality of the NDFD-equivalence with respect to the choice-operator [] lies in the fact that the NDFD-equivalence does not tell us whether a non-divergent failure (ε_A, Π) arises because of a stable initial state or because of a non-empty sequence of invisible transitions from an initial state to a stable state. To be able to distinguish between these two cases, we need to add some new information to our semantic model.

Definition 4.6.10 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts, $\sigma \in VB_*$ a finite visible behaviour, and $\Pi \subseteq \Sigma$ a set of non-empty transition labels. The ordered pair $(\sigma, \Pi) \in VB_* \times P(\Sigma)$ is *a non-initial stable failure of* L iff there is an $s \in I$, an $s' \in S$ and a $\rho \in B_*$ such that

- $|\rho| > 0$,
- $vis(\rho) = \sigma$,
- $s \xrightarrow{\rho} s'$ and
- there is no $s'' \in S$ and $l \in \Pi \cup \{\tau\}$ such that $(s', l, s'') \in \Delta$.

The set of all non-initial stable failures of L is denoted by $nisfail(L)$.

Furthermore, (σ, Π) is *a non-initial non-divergent failure of L* iff

- (σ, Π) is a non-initial stable failure of L, $(\sigma, \Pi) \in nisfail(L)$, and
- σ is not a divergence trace of L, $\sigma \notin divtr(L)$.

The set of all non-initial non-divergent failures of L is denoted by $nindfail(L)$. □

Definition 4.6.11 Let L_1 and L_2 be altss. We say that *L_1 and L_2 are (non-initial) non-divergent failures divergences equivalent* or *NINDFD-equivalent* and write $L_1 \overset{\text{nindfd}}{\approx} L_2$ iff

- $L_1 \overset{\text{ndfd}}{\approx} L_2$, and
- $nindfail(L_1) = nindfail(L_2)$. □

For the relation of this definition of the NINDFD-equivalence with the definition of NDFD-equivalence in [KV92, VT95], please see discussion at the end of the current section.

It should be easy to see that for all L and all (σ, Π) such that $|\sigma| > 0$,

$$(\sigma, \Pi) \in nindfail(L) \quad \text{iff} \quad (\sigma, \Pi) \in ndfail(L)$$

Consequently, the only way in which $\overset{\text{nindfd}}{\approx}$ is finer than $\overset{\text{ndfd}}{\approx}$, is with respect to nondivergent failures (σ, Π) with $|\sigma| = 0$.

It is clear that NINDFD-equivalence preserves all LTL' and $tLTL'$-properties.

Proposition 4.6.12 Let L_1 and L_2 be altss such that $L_1 \overset{\text{nindfd}}{\approx} L_2$. Then $L_1 \overset{maxtr}{\approx} L_2$, $L_1 \overset{ltl'}{\approx} L_2$ and $L_1 \overset{tltl'}{\approx} L_2$.

Proof: Immediate from 4.2.8 and 4.2.9. □

Let us check that $\overset{\text{nindfd}}{\approx}$ is compositional with respect to all the previously introduced operators.

Lemma 4.6.13 Let $L_1 = (S_1, I_1, \Gamma_1, \Delta_1, A_1, V_1)$ and $L_2 = (S_2, I_2, \Gamma_2, \Delta_2, A_2, V_2)$ be altss such that $A_1 \cap A_2 = \emptyset$, and let $\Pi \subseteq \Sigma$. Then for all behaviours $\sigma \in VB_\omega$ and transition sets $\Pi_f \subseteq \Sigma$,

$$(\sigma, \Pi_f) \in nindfail(L_1 \,|[\Pi]|\, L_2)$$

iff $\sigma \notin divtr(L_1 \,|[\Pi]|\, L_2)$ and there are $\pi, \pi' \in VB_\infty$ and $\Pi_{f1}, \Pi_{f2} \subseteq \Sigma$ such that

- $(\pi, \Pi_{f1}) \in ndfail(L_1)$,
- $(\pi', \Pi_{f2}) \in ndfail(L_2)$,

- either $(\pi, \Pi_{f1}) \in \mathit{nindfail}(L_1)$ or $(\pi', \Pi_{f2}) \in \mathit{nindfail}(L_2)$,
- $\Pi_f = ((\Pi_{f1} \cup \Pi_{f2}) \cap \Pi) \cup ((\Pi_{f1} \cap \Pi_{f2}) \setminus \Pi)$, and
- $\mathit{par}_s(\sigma, \pi, \pi', \Pi)$ holds.

Proof: Direct from 4.3.8, and the observation that any non-empty sequence of transitions of $L_1 \;|[\Pi]|\; L_2$ can be projected to a sequence of transitions of L_1 and a sequence of transitions of L_2, at least one of which must be non-empty. □

Proposition 4.6.14 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$, and let $\Pi \subseteq \Sigma$. Then

- $L_1 \;|[\Pi]|\; L_2$ is defined iff $L_1' \;|[\Pi]|\; L_2'$ is defined, and
- if both are defined, then $L_1 \;|[\Pi]|\; L_2 \overset{\text{nindfd}}{\approx} L_1' \;|[\Pi]|\; L_2'$.

Proof: Since $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$, by definition $L_1 \overset{\text{ndfd}}{\approx} L_1'$ and $L_2 \overset{\text{ndfd}}{\approx} L_2'$. From Proposition 4.3.9 we know then that $L_1 \;|[\Pi]|\; L_2$ is defined iff $L_1' \;|[\Pi]|\; L_2'$ is defined, and if both are defined, then $L_1 \;|[\Pi]|\; L_2 \overset{\text{ndfd}}{\approx} L_1' \;|[\Pi]|\; L_2'$. Suppose then that both are defined. By assumption, $\mathit{ndfail}(L_1) = \mathit{ndfail}(L_1')$, $\mathit{nindfail}(L_1) = \mathit{nindfail}(L_1')$, $\mathit{ndfail}(L_2) = \mathit{ndfail}(L_2')$ and $\mathit{nindfail}(L_2) = \mathit{nindfail}(L_2')$. Since $L_1 \;|[\Pi]|\; L_2 \overset{\text{ndfd}}{\approx} L_1' \;|[\Pi]|\; L_2'$, we also know that $\mathit{divtr}(L_1 \;|[\Pi]|\; L_2) = \mathit{divtr}(L_1' \;|[\Pi]|\; L_2')$. By Lemma 4.6.13, this implies

$$\mathit{nindfail}(L_1 \;|[\Pi]|\; L_2) = \mathit{nindfail}(L_1' \;|[\Pi]|\; L_2')$$

Consequently, $L_1 \;|[\Pi]|\; L_2 \overset{\text{nindfd}}{\approx} L_1' \;|[\Pi]|\; L_2'$. □

Proposition 4.6.15 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$. Then

- $L_1 \parallel_s L_2$ is defined iff $L_1' \parallel_s L_2'$ is defined, and
- if both are defined, then $L_1 \parallel_s L_2 \overset{\text{nindfd}}{\approx} L_1' \parallel_s L_2'$.

Proof: Immediate from 4.6.14, as in the proof of 4.3.10. □

Lemma 4.6.16 Let L_1 and L_2 be condition-labelled altss s.t. $AP(L_1) \cap AP(L_2) = \emptyset$. Then for all behaviours $\sigma \in VB_\omega$ and transition sets $\Pi \subseteq \Sigma$,

$$(\sigma, \Pi) \in \mathit{nindfail}(L_1 \parallel_v L_2)$$

iff $\sigma \notin \mathit{divtr}(L_1 \parallel_v L_2)$ and there are $\pi, \pi' \in VB_\infty$ such that

- $(\pi, \Pi_1) \in \mathit{ndfail}(L_1)$,
- $(\pi', \Pi_2) \in \mathit{ndfail}(L_2)$,
- either $(\pi, \Pi_1) \in \mathit{nindfail}(L_1)$ or $(\pi', \Pi_2) \in \mathit{nindfail}(L_2)$, and
- $\mathit{par}_v(\sigma, \pi, \pi', AP(L_1), AP(L_2))$ holds,

where

- $\Pi_1 = \{l \in \Sigma_{\mathit{cond}}(AP \setminus AP(L_1)) \mid \mathit{eval}(AP(L_2), \pi'_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$, and

- $\Pi_2 = \{l \in \Sigma_{cond}(AP \setminus AP(L_2)) \mid \mathit{eval}(AP(L_1), \pi_{\mathit{fin}})(l) \in \Pi \cup \{\tau\}\}$.

Proof: As in 4.6.13, but based on 4.3.18. □

Proposition 4.6.17 Let L_1, L_1', L_2 and L_2' be condition-labelled altss such that $L_1 \stackrel{\text{nindfd}}{\approx} L_1'$ and $L_2 \stackrel{\text{nindfd}}{\approx} L_2'$. Then

- $L_1 \parallel_v L_2$ is defined iff $L_1' \parallel_v L_2'$ is defined, and
- if both are defined, then $L_1 \parallel_v L_2 \stackrel{\text{nindfd}}{\approx} L_1' \parallel_v L_2'$.

Proof: Like the proof of Proposition 4.6.14, but instead of relying on Lemma 4.6.13, it relies on Lemma 4.6.16. □

Proposition 4.6.18 Let L_1 and L_1' be altss such that $L_1 \stackrel{\text{nindfd}}{\approx} L_1'$, and let $\Pi \subseteq \Sigma$. Then

- both **hide** Π **in** L_1 and **hide** Π **in** L_1' are defined, and
- **hide** Π **in** $L_1 \stackrel{\text{nindfd}}{\approx}$ **hide** Π **in** L_1'.

Similarly, if $f : \Sigma \to \Sigma$ is a renaming function, then

- both $L_1[f]$ and $L_1'[f]$ are defined and
- $L_1[f] \stackrel{\text{nindfd}}{\approx} L_1'[f]$,

If $B \subseteq AP$, then

- both **encaps** B **in** L_1 and **encaps** B **in** L_1' are defined, and
- **encaps** B **in** $L_1 \stackrel{\text{nindfd}}{\approx}$ **encaps** B **in** L_1'.

Furthermore, let $B \subseteq AP$ and $l \in \Sigma_\tau$. Then

- both $(B \xrightarrow{l}); L_1$ and $(B \xrightarrow{l}); L_1'$ are defined, and
- $(B \xrightarrow{l}); L_1 \stackrel{\text{nindfd}}{\approx} (B \xrightarrow{l}); L_1'$.

Proof: The claim for hiding follows from 4.3.20 and the equation

$$\begin{array}{rcl}\mathit{nindfail}(\textbf{hide}\ \Pi\ \textbf{in}\ L_1) & = & \{(\sigma, \Pi_f) \in VB_* \times P(\Sigma) \mid \sigma \notin \mathit{divtr}(\textbf{hide}\ \Pi\ \textbf{in}\ L_1) \text{ and} \\ & & \qquad \exists \sigma' \in VB_* \text{ such that} \\ & & \qquad g(\sigma') = \sigma \text{ and} \\ & & \qquad (\sigma', \Pi_f \cup \Pi) \in \mathit{nindfail}(L_1)\}\end{array}$$

where g is as in the proof of 4.3.20. The claims for renaming and encapsulation are proved in the same way, based on 4.3.21 and 4.3.22.

Let us then show the claim for prefixing. Since $L_1 \stackrel{\text{nindfd}}{\approx} L_1'$, $L_1 \stackrel{\text{ndfd}}{\approx} L_1'$, and by 4.6.4, $(B \xrightarrow{l}); L_1 \stackrel{\text{ndfd}}{\approx} (B \xrightarrow{l}); L_1'$. Seeing that $\mathit{nindfail}((B \xrightarrow{l}); L_1) = \mathit{nindfail}((B \xrightarrow{l}); L_1')$ can be done as in 4.6.4, on the basis of the equality

$$\begin{array}{rcl}\mathit{nindfail}((B \xrightarrow{l}); L_1) & = & \{(\sigma, \Pi) \in VB_* \times P(\Sigma) \mid \exists (\sigma', \Pi) \in \mathit{ndfail}(L_1) \text{ such that} \\ & & \qquad \sigma = \mathit{vis}(Bl\sigma'_{init} \cdot \sigma')\}\end{array}$$

□

Proposition 4.6.19 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$. Then

- both $L_1 \gg L_2$ and $L_1' \gg L_2'$ are defined, and
- $L_1 \gg L_2 \overset{\text{nindfd}}{\approx} L_1' \gg L_2'$.

Furthermore,

- both $L_1 \,[:]\, L_2$ and $L_1' \,[:]\, L_2'$ are defined, and
- $L_1 \,[:]\, L_2 \overset{\text{nindfd}}{\approx} L_1' \,[:]\, L_2'$.

Proof: Let us first show the claim for $\gg$. Since $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$, $L_1 \overset{\text{ndfd}}{\approx} L_1'$ and $L_2 \overset{\text{ndfd}}{\approx} L_2'$, and by 4.6.5 $L_1 \gg L_2$ is defined iff $L_1' \gg L_2'$ is defined, and if both are defined $L_1 \gg L_2 \overset{\text{ndfd}}{\approx} L_1' \gg L_2'$. Seeing that

$$nindfail(L_1 \gg L_2) = nindfail(L_1' \gg L_2')$$

can be done as in the proof of 4.6.5, on the basis of the equation

$$\begin{array}{rcll} ndfail(L_1 \gg L_2) & = & \{(\sigma,\Pi) \in VB_* \times P(\Sigma) \mid & \delta \notin \Sigma(\sigma) \text{ and} \\ & & & (\sigma, \Pi \cup \{\delta\}) \in nindfail(L_1)\} \cup \\ & & \{(\sigma,\Pi) \in VB_* \times P(\Sigma) \mid & \exists \pi \in sctr(L_1), (\pi',\Pi) \in ndfail(L_2) \\ & & & \text{such that } \sigma = \pi \cdot (\pi_{fin}\tau\pi'_{init}) \cdot \pi'\} \end{array}$$

The claim for [:] can be shown as above, on the basis of 4.6.8 and the equation

$$nindfail(L_1 \,[:]\, L_2) \;=\; (nindfail(L_1) \cup nindfail(L_2)) \setminus divtr(L_1 \,[:]\, L_2) \times P(\Sigma)$$

□

Let us then see that, unlike $\overset{\text{ndfd}}{\approx}$, $\overset{\text{nindfd}}{\approx}$ is indeed compositional with respect to the other choice operator [].

Proposition 4.6.20 Let L_1, L_1', L_2, L_2' be altss such that $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_2 \overset{\text{nindfd}}{\approx} L_2'$. Then

- both $L_1 \,[]\, L_2$ and $L_1' \,[]\, L_2'$ are defined, and
- $L_1 \,[]\, L_2 \overset{\text{nindfd}}{\approx} L_1' \,[]\, L_2'$.

Proof: It is trivial to see that both $L_1 \,[]\, L_2$ and $L_1' \,[]\, L_2'$ are defined. Define then for any alts L the set of initial truth sets of L by

$$it(L) = \{A \in P(AP) \mid \exists \sigma \in tr(L) \text{ such that } A = \sigma_{init}\}$$

Notice that since $L_1 \overset{\text{ndfd}}{\approx} L_1'$, by 4.3.1 $tr(L_1) = tr(L_1')$, which implies that $it(L_1) = it(L_1')$. Similarly, $it(L_2) = it(L_2')$.

Notice then that

$$\begin{array}{rcl} \Sigma(L_1 \,[]\, L_2) & = & \Sigma(L_1) \cup \Sigma(L_2) \\ AP(L_1 \,[]\, L_2) & = & AP(L_1) \cup AP(L_2) \end{array}$$

$$
\begin{aligned}
\mathit{divtr}(L_1 \,[]\, L_2) &= \{\sigma \in \mathit{divtr}(L_1) \mid \exists A \in it(L_2) \text{ such that } \sigma_{init} = A\} \cup \\
&\quad \{\sigma \in \mathit{divtr}(L_2) \mid \exists A \in it(L_1) \text{ such that } \sigma_{init} = A\} \\
\mathit{inftr}(L_1 \,[]\, L_2) &= \{\sigma \in \mathit{inftr}(L_1) \mid \exists A \in it(L_2) \text{ such that } \sigma_{init} = A\} \cup \\
&\quad \{\sigma \in \mathit{inftr}(L_2) \mid \exists A \in it(L_1) \text{ such that } \sigma_{init} = A\} \\
\mathit{ndfail}(L_1 \,[]\, L_2) &= (\{(\sigma, \Pi) \in \mathit{nindfail}(L_1) \mid \exists A \in it(L_2) \text{ such that } \sigma_{init} = A\} \cup \\
&\quad \{(\sigma, \Pi) \in \mathit{nindfail}(L_2) \mid \exists A \in it(L_1) \text{ such that } \sigma_{init} = A\} \cup \\
&\quad (\mathit{ndfail}(L_1) \cap \mathit{ndfail}(L_2))) \\
&\quad \setminus \mathit{divtr}(L_1 \,[]\, L_2) \times P(\Sigma) \\
\mathit{nindfail}(L_1 \,[]\, L_2) &= (\{(\sigma, \Pi) \in \mathit{nindfail}(L_1) \mid \exists A \in it(L_2) \text{ such that } \sigma_{init} = A\} \cup \\
&\quad \{(\sigma, \Pi) \in \mathit{nindfail}(L_2) \mid \exists A \in it(L_1) \text{ such that } \sigma_{init} = A\}) \\
&\quad \setminus \mathit{divtr}(L_1 \,[]\, L_2) \times P(\Sigma)
\end{aligned}
$$

and that similar equations hold for $L_1' \,[]\, L_2'$. Since by assumption $L_1 \overset{\text{nindfd}}{\approx} L_1'$ and $L_1 \overset{\text{nindfd}}{\approx} L_2'$, $L_1 \,[]\, L_2 \overset{\text{nindfd}}{\approx} L_1' \,[]\, L_2'$. □

We can now summarise the compositionality results with respect to all the operators introduced in this work.

Theorem 4.6.21 The NDFD-equivalence $\overset{\text{ndfd}}{\approx}$ is compositional with respect to the operations

- $\ldots |[\Pi]| \ldots$
- $\ldots \|_s \ldots$
- $\ldots \|_v \ldots$
- **hide** Π **in** $\ldots$
- $\ldots [f]$
- **encaps** B **in** $\ldots$
- $(B \xrightarrow{l}); \ldots$
- $\ldots \gg \ldots$ and
- $\ldots [:] \ldots$

for all $\Pi \subseteq \Sigma$, $f : \Sigma \to \Sigma$, $l \in \Sigma$ and $B \subseteq AP$.

The NINDFD-equivalence $\overset{\text{nindfd}}{\approx}$ is compositional with respect to all the above operations and $\ldots [] \ldots$

Proof: The first statement summarises 4.3.23, 4.6.4, 4.6.5 and 4.6.8, and the second statement 4.6.14, 4.6.15, 4.6.17, 4.6.18, 4.6.19 and 4.6.20. □

We can naturally now ask the same question about full abstractness with respect to $\overset{\text{nindfd}}{\approx}$ as we did in the case of $\overset{\text{ndfd}}{\approx}$. It turns out that $\overset{\text{nindfd}}{\approx}$ is precisely the weakest equivalence that contains $\overset{ltl'}{\approx}$ and is compositional with respect to all the operators introduced in this work, including [].

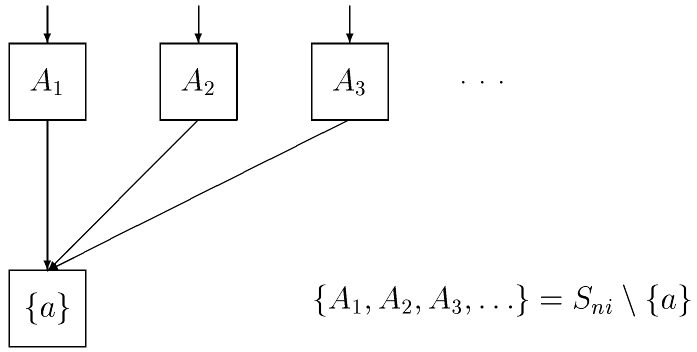


Figure 4.15: Alts L_{ni}

Lemma 4.6.22 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $[]$, and
- for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\text{ndfd}}{\approx} L'$.

Then for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\text{nindfd}}{\approx} L'$.

Proof: Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be (finite) altss such that $L \overset{x}{\approx} L'$. We need to show that $L \overset{\text{nindfd}}{\approx} L'$. Since by assumption we already know that $L \overset{\text{ndfd}}{\approx} L'$, it is enough to show that $\mathit{nindfail}(L) = \mathit{nindfail}(L')$.

Take any finite visible behaviour $\sigma \in VB_*$ and any set of non-empty transition labels $\Pi \subseteq \Sigma$. If $AP(\sigma) \not\subseteq A \cup A'$, then clearly $(\sigma, \Pi) \notin \mathit{nindfail}(L)$ and $(\sigma, \Pi) \notin \mathit{nindfail}(L')$. Consequently, for all such (σ, Π) we know that $(\sigma, \Pi) \in \mathit{nindfail}(L)$ iff $(\sigma, \pi) \in \mathit{nindfail}(L')$.

Suppose then that $AP(\sigma) \subseteq A \cup A'$. We shall construct an alts L_{ni}, which is finite if both L and L' are, such that for both $L^\dagger = L$ and $L^\dagger = L'$:

$$(\sigma, \Pi) \in \mathit{nindfail}(L^\dagger) \text{ iff } (\sigma, \Pi) \in \mathit{ndfail}(L_{ni} \, [] \, L^\dagger)$$

The result follows from this as in the proof of 4.4.2.

The alts L_{ni} is defined by $L_{ni} = (S_{ni}, I_{ni}, \Gamma_{ni}, \Delta_{ni}, A_{ni}, V_{ni})$, where

$$\begin{array}{rcl}
S_{ni} & = & \{B \in P(A) \mid \exists s \in I \text{ such that } V(s) = B\} \cup \\
 & & \{B \in P(A') \mid \exists s \in I' \text{ such that } V'(s) = B\} \cup \\
 & & \{a\}, \qquad \text{where } a \notin A \cup A' \\
I_{ni} & = & S_{ni} \setminus \{a\} \\
\Gamma_{ni} & = & \emptyset \\
\Delta_{ni} & = & \{(A, \tau, \{a\}) \mid A \in S \setminus \{a\}\} \\
A_{ni} & = & A \cup A' \cup \{a\} \\
V_{ni} & : & S_{ni} \to P(A_{ni}); s \mapsto s
\end{array}$$

This is described by the graph in Figure 4.15. If $(\sigma, \Pi) \in \mathit{nindfail}(L^\dagger)$, it is easy to see that

$$(\sigma, \Pi) \in \mathit{nindfail}(L_{ni} \, [] \, L^\dagger) \subseteq \mathit{ndfail}(L_{ni} \, [] \, L^\dagger)$$

i.e. the claim holds in one direction.

Suppose then that $(\sigma, \Pi) \in \mathit{ndfail}(L_{ni} \mathbin{[]} L^\dagger)$. Since all the initial nodes of $L_{ni} \mathbin{[]} L^\dagger$ are unstable, this implies $(\sigma, \Pi) \in \mathit{nindfail}(L_{ni} \mathbin{[]} L^\dagger)$, i.e. either $(\sigma, \Pi) \in \mathit{nindfail}(L_{ni})$ or $(\sigma, \Pi) \in \mathit{nindfail}(L^\dagger)$. The former is not possible, since it is easy to see that for all $(\sigma', \Pi') \in \mathit{nindfail}(L_{ni})$, $a \in AP(\sigma')$, but by assumption $a \notin A' \cup A'$, $AP(\sigma) \subseteq A \cup A'$, and therefore $a \notin AP(\sigma)$. Consequently, $(\sigma, \Pi) \in \mathit{nindfail}(L^\dagger)$, and the claim holds in the other direction, as well. □

Theorem 4.6.23 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$, **hide** Σ **in** , and $[]$,
- for all closed (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{maxtr}{\approx} L'$, and
- for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all (finite) altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\text{nindfd}}{\approx} L'$.

Proof: Immediate from 4.4.8 and 4.6.22. □

Corollary 4.6.24 Let $\overset{x}{\approx}$ be an equivalence between altss, such that

- $\overset{x}{\approx}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$, **hide** Σ **in** and $[]$,
- for all closed finite altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{ltl'}{\approx} L'$, and
- for all finite altss L and L', $L \overset{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all finite altss L and L', $L \overset{x}{\approx} L'$ implies $L \overset{\text{nindfd}}{\approx} L'$.

Proof: Immediate from 4.1.18 and 4.6.23. □

Finally, let us see that for altss with unique initial states, NINDFD-equivalence has a simpler formulation. The following proposition shows that for such altss, we only need to consider initial stability in addition to the NDFD-equivalence to arrive at an equivalence that is the same as the NINDFD-equivalence. What this means is that for such altss the definition of NDFD-equivalence of [KV92, VT95] coincides with the definition of the NINDFD-equivalence above.

Definition 4.6.25 Let $L = (S, I, \Gamma, \Delta, A, V)$ be an alts. We say that L is *initially stable* iff there are no $s \in I$, $s' \in S$ such that $(s, \tau, s') \in \Delta$. □

Proposition 4.6.26 Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be altss such that $|I| = |I'| = 1$. Then $L \overset{\text{nindfd}}{\approx} L'$ iff

- $L \overset{\text{ndfd}}{\approx} L'$, and
- L is initially stable iff L' is initially stable

Proof: If $L \overset{\text{ndfd}}{\not\approx} L'$, then both sides of the equality fail, and the claim holds. Suppose then that $L \overset{\text{ndfd}}{\approx} L'$. If both L and L' are initially stable,

$$\begin{array}{rcl} \mathit{nindfail}(L) & = & \{(\sigma, \Pi) \in \mathit{ndfail}(L) \mid |\sigma| > 0\} \\ & = & \{(\sigma, \Pi) \in \mathit{ndfail}(L') \mid |\sigma| > 0\} \quad = \quad \mathit{nindfail}(L') \end{array}$$

i.e. $L \overset{\text{nindfd}}{\approx} L'$.

If neither L nor L' is initially stable, then

$$nindfail(L) = ndfail(L) = ndfail(L') = nindfail(L')$$

i.e. again $L \overset{\text{nindfd}}{\approx} L'$. Consequently, the implication from right to left holds.

Suppose then that L is initially stable but L' is not. As L is initially stable,

- for all $\sigma \in tr(L)$ such that $|\sigma| > 0$, $\sigma_1 = (l, B)$ for some $l \neq \tau$ and B,
- for all $\sigma \in divtr(L)$, $|\sigma| > 0$, and
- for all $(\sigma, \Pi) \in nindfail(L)$, $|\sigma| > 0$.

However, as L' is not initially stable, either

- there is a $\sigma \in tr(L')$ such that $|\sigma| > 0$ and $\sigma_1 = (\tau, B)$ for some B, or
- there is a $\sigma \in divtr(L')$ such that $|\sigma| = 0$, or
- there is a $(\sigma, \Pi) \in nindfail(L')$ such that $|\sigma| = 0$.

As either of the first two would imply $L \overset{\text{ndfd}}{\not\approx} L'$, contrary to the assumption, the third case has to hold. But then $nindfail(L) \neq nindfail(L')$, and $L \overset{\text{nindfd}}{\not\approx} L'$. For an initially stable L' and initially unstable L', we can show in a symmetric fashion that $L \overset{\text{nindfd}}{\not\approx} L'$. Consequently, the claim holds in the other direction, as well. $\Box$

In addition to the composition operators discussed above, the process algebra LOTOS also contains a primitive *interrupt* operator [>. In [VT95] it was shown that for altss with a unique initial node and no atomic propositions, the equivalence corresponding to NINDFD is compositional with respect to [>, as well. In the current work the interrupt operator was not considered, as it is not as common in process algebras in general as the other operators discussed here, and as the intuition behind the operator does not appear to be as naturally amenable to the current framework as with the other operators.

4.6.3 Full abstractness for infinite systems

Previously it was shown in Theorem 4.1.12 that for all altss $\overset{maxtr}{\approx}$ implies $\overset{ltl'}{\approx}$, and in Theorem 4.1.18 that for finite closed altss the converse holds as well. The question to be addressed now is whether the converse holds for infinite closed altss.

A crucial step where finiteness of altss is needed in the proof of 4.1.18 is the one from preservation of finite traces to preservation of infinite traces in Lemma 4.1.17. This lemma is not generally valid for infinite altss. The following two examples locate the position of $\overset{ltl'}{\approx}$ with respect to $\overset{maxtr}{\approx}$ and the equivalence defined by having just the same traces and finite maximal traces. They show that

- there are altss L and L' such that $tr(L) = tr(L')$ and $fmaxtr(L) = fmaxtr(L')$, but $L \overset{ltl'}{\not\approx} L'$, and
- there are altss L and L' such that $tr(L) = tr(L')$, $fmaxtr(L) = fmaxtr(L')$ and $inftr(L) \neq inftr(L')$, but $L \overset{ltl'}{\approx} L'$.

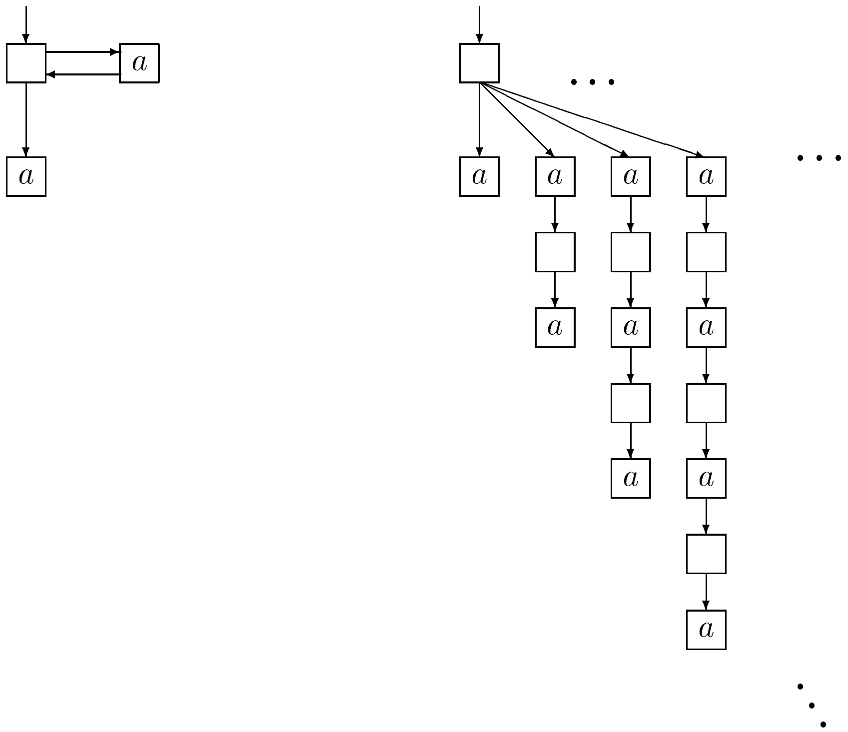


Figure 4.16: Altss L_{fin} (left) and L_{inf} (right)

These two results show that in the class of all closed altss, the equivalence $\stackrel{ltl'}{\approx}$ is strictly less discriminating than $\stackrel{maxtr}{\approx}$, i.e. $\stackrel{ltl'}{\approx}$ does not imply $\stackrel{maxtr}{\approx}$, but strictly more discriminating than the equivalence defined by having the same traces and finite maximal traces.

Example 4.6.27 Two closed altss L_{fin} and L_{inf} are described in Figure 4.16. For them $\mathit{tr}(L_{\mathit{fin}}) = \mathit{tr}(L_{\mathit{inf}})$ and $\mathit{fmaxtr}(L_{\mathit{fin}}) = \mathit{fmaxtr}(L_{\mathit{inf}})$. However, $L_{\mathit{fin}} \stackrel{ltl'}{\not\approx} L_{\mathit{inf}}$, since $L_{\mathit{fin}} \not\models \Diamond\Box a$, but $L_{\mathit{inf}} \models \Diamond\Box a$. $\Box$

Example 4.6.28 In this example we construct two closed altss L and L' such that $\mathit{tr}(L) = \mathit{tr}(L')$, $\mathit{fmaxtr}(L) = \mathit{fmaxtr}(L')$ and $L \stackrel{ltl'}{\approx} L'$, but $\mathit{inftr}(L) \neq \mathit{inftr}(L')$. The construction is based on the so-called *finite model property* of linear temporal logic. This

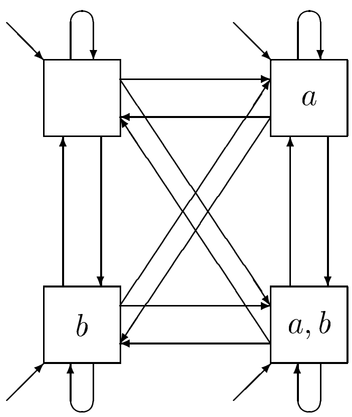
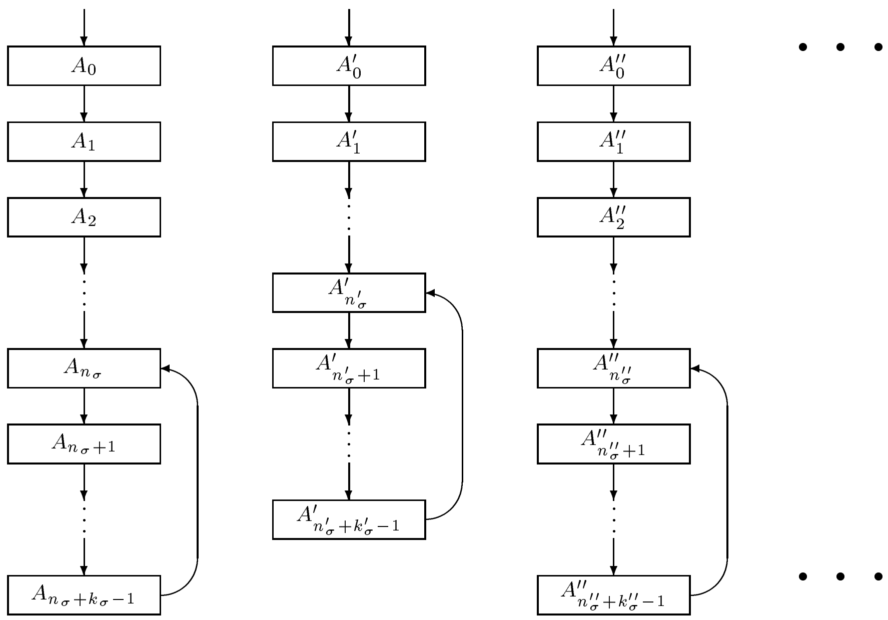


Figure 4.17: Alts L



$R = \{\sigma, \sigma', \sigma'', \ldots\}$
$\sigma = A_0 \tau A_1 \tau A_2 \ldots$
$\sigma' = A'_0 \tau A'_1 \tau A'_2 \ldots$
$\sigma'' = A''_0 \tau A''_1 \tau A''_2 \ldots$
etc.

Figure 4.18: Alts L'

says that if a formula is satisfiable, i.e. if it has any models at all, it is also satisfiable in a finite model. Let us first make this statement more accurate and express it in the terminology of this work.

Definition Let $\rho \in B_\omega$ be an infinite behaviour. We say that ρ is an *ω-regular behaviour* iff there are $n \geq 0$ and $k > 0$ such that for all $i \geq n$, $\rho_i = \rho_{i+k}$. Given an ω-regular behaviour ρ, we use n_ρ and k_ρ to denote some n and k fulfilling the requirements above.

Intuitively, an ω-regular behaviour consist of a finite initial part (up to point n in the definition) and a finite cycle which is repeated infinitely many times (from point n to point $n+k$ in the definition). Now the finite model property can be formulated as follows.

Fact ([SC85, Theorem 4.7], see also [Eme90, Sti92]) Let ϕ be an LTL'-formula. If there exists a behaviour $\rho \in B_\infty$ such that $\rho \models \phi$, then there exists an ω-regular behaviour $\rho' \in B_\omega$ such that $\rho' \models \phi$. Furthermore, we can choose the ρ' so that $AP(\rho') \subseteq AP(\rho)$ and $\Sigma(\rho') \subseteq \Sigma(\rho)$.

Let us define

$$\begin{array}{rcl} R & = & \{\rho \in B_\omega \mid \rho \text{ is } \omega\text{-regular and } AP(\rho) \subseteq \{a,b\} \text{ and } \Sigma(\rho) = \emptyset\} \\ G & = & \{\rho \in B_\infty \mid AP(\rho) \subseteq \{a,b\} \text{ and } \Sigma(\rho) = \emptyset\} \end{array}$$

Notice that the fact above implies that for any LTL'-formula ϕ, if there is a $\rho \in G$ such that $\rho \models \phi$, then there is a $\rho' \in R$ such that $\rho' \models \phi$. Since $R \subseteq G$ by definition, the implication clearly holds in the other direction, as well.

Let us now define an alts $L = (S, I, \Gamma, \Delta, A, V)$ by

$$\begin{array}{rcl} S & = & P(\{a,b\}) \\ I & = & S \\ \Gamma & = & \emptyset \\ \Delta & = & S \times \{\tau\} \times S \\ A & = & \{a,b\} \\ V & : & S \to P(A); s \mapsto s \end{array}$$

and an alts $L' = (S', I', \Gamma', \Delta', V', A')$ by

$$\begin{array}{rcl} S' & = & R \times \mathbb{N} \\ I' & = & R \times \{0\} \\ \Gamma' & = & \emptyset \\ \Delta' & = & \{((\sigma, i-1), \tau, (\sigma, i)) \mid \sigma \in R \text{ and } 0 < i < n_\sigma + k_\sigma\} \cup \\ & & \{((\sigma, n_\sigma + k_\sigma - 1), \tau, (\sigma, n_\sigma)) \mid \sigma \in R\} \\ A' & = & \{a,b\} \\ V' & : & S \to P(A'); (\sigma, i) \mapsto A_i \quad \text{where } \sigma_i = (\tau, A_i) \end{array}$$

The alts L is depicted in Figure 4.17 and the alts L' in Figure 4.18. Notice that $maxbeh(L) = G$ and $maxbeh(L') = R$. Take any LTL'-formula ϕ. Now

$L \not\models \phi$
iff there is a $\rho \in maxbeh(L)$ such that $\rho \models \neg\phi$
iff there is a $\rho \in G$ such that $\rho \models \neg\phi$
iff there is a $\rho' \in R$ such that $\rho' \models \neg\phi$
iff there is a $\rho' \in maxbeh(L')$ such that $\rho' \models \neg\phi$
iff $L' \not\models \phi$.
Consequently, $L \overset{ltl'}{\approx} L'$.

Notice then that

- $dltr(L) = dltr(L') = \emptyset$, and
- $divtr(L) = divtr(L') = tr(L) = tr(L') =$
  $\{\sigma \in VB_* \mid AP(\sigma) \subseteq \{a, b\}$ and $\Sigma(\sigma) = \emptyset\}$

This means that $fmaxtr(L) = fmaxtr(L')$ and $tr(L) = tr(L')$.

Define finally an infinite behaviour $\sigma \in VB_\omega$ by

$$\sigma \; = \; \alpha \cdot \beta \cdot \alpha \cdot \beta^2 \cdot \alpha \cdot \beta^3 \cdot \cdots$$

where $\alpha = \emptyset\tau\{a\}\tau\emptyset$ and $\beta = \emptyset\tau\{b\}\tau\emptyset$. It is easy to see that $\sigma \in inftr(L)$. However, there is no ω-regular ρ such that $vis(\rho) = \sigma$. Consequently, $\sigma \notin inftr(L')$, i.e. $inftr(L) \neq inftr(L')$. □

From this example we see that sets of LTL'-formulae cannot fully represent all infinite traces of all altss. This feature is by no means peculiar to LTL'. In fact, the finite model property, which is at the heart of the previous example, holds for most propositional temporal and modal logics. Therefore, it seems likely that similar counterexamples could be constructed for any such logics.

There is also another way of looking at this discrepancy arising in the class of arbitrary altss, between equivalences based on some finitary methods of descriptions, and equivalences based on sets on infinite traces. As there are only countably many possible descriptions (formulae of LTL' in our case) but uncountably many possible infinite traces, it is impossible to uniquely describe each set of infinite traces by a set of descriptions. In other words, there are always systems fulfilling precisely the same descriptions but having different sets of infinite traces.

If we want to have an exact correspondence between an equivalence based on infinite traces and an equivalence based on logical descriptions, there are at least two ways to proceed. First, we can restrict the set of allowed transition systems, e.g. by requiring that the transition relations have to be recursive or that the systems are described by finite expressions in some formal language, such as CCS or LOTOS. In this case there will be only countably many possible altss, so the discrepancy will not arise. Secondly, we can extend the language of logic by allowing infinitary descriptions. In this case there will be uncountably many possible descriptions, so the discrepancy vanishes again. This approach is taken by e.g. Hennessy and Milner [HM85, Mil89] to establish the well-known correspondence between strong bisimulation equivalence and the Hennessy-Milner modal logic. We shall also follow the same track in this work.

Let us first define an extension of LTL' allowing infinitary conjunction and disjunction.

Definition 4.6.29 The alphabet of the logic $\omega LTL'$ is that of LTL' with the binary disjunction $\vee$ replaced by a family of indexed disjunction operators $\bigvee_{n\in N'}$ for each $N' \subseteq \mathbb{N}$. The set of well-formed formulae (wffs) of $\omega LTL'$ is defined as the minimal set closed under the rules 1, 2 and 4 of definition 3.2.1 and the following rule:

3' if $N' \subseteq \mathbb{N}$ and for all $n \in N'$, ϕ_n is a wff, then $(\bigvee_{n\in N'} \phi_n)$ is a wff.

Furthermore, let us define the syntactic abbreviation $\bigwedge_{n\in N'}$ by:

$$(\bigwedge_{n\in N'} \phi_n) \quad = \quad (\neg(\bigvee_{n\in N'} (\neg\phi_n)))$$

□

Definition 4.6.30 If ϕ is a $\omega LTL'$-formula and $\sigma \in B_\omega$ a behaviour, $\sigma \models \phi$ is defined by the rules 1, 2 and 4 of Definition 3.2.2 and the following rule:

3' $\sigma \models \bigvee_{n\in N'} \phi_n$ iff there is an $n \in N'$ such that $\sigma \models \phi_n$. □

Definition 4.6.31 Let L and L' be altss. We say that L and L' are *$\omega LTL'$-equivalent* and write $L \overset{\omega ltl'}{\approx} L'$ iff for all $\omega LTL'$-formulae ϕ, $L \models \phi$ iff $L' \models \phi$. □

We can easily extend the earlier results to show that $\overset{maxtr}{\approx}$ preserves not only the validity of all LTL'-formulae but also that of all $\omega LTL'$-formulae.

Lemma 4.6.32 Let ϕ be an $\omega LTL'$-formula. Then for all behaviours $\rho \in B_\infty$, $\rho \models \phi$ iff $vis(\rho) \models \phi$.

Proof: As the proof of 4.1.6. □

Proposition 4.6.33 Let L and L' be altss such that $L \overset{maxtr}{\approx} L'$. Then $L \overset{\omega ltl'}{\approx} L'$.

Proof: As the proof of 4.1.12. □

It is easy to see that $\overset{\text{ndfd}}{\approx}$ implies $L \overset{\omega ltl'}{\approx}$.

Corollary 4.6.34 Let L and L' be altss such that $L \overset{\text{ndfd}}{\approx} L'$. Then $L \overset{\omega ltl'}{\approx} L'$.

Proof: Immediate from 4.2.8 and 4.6.33. □

We can now use the earlier techniques to show directly that for all closed altss, $\overset{\omega ltl'}{\approx}$ preserves all maximal traces.

Lemma 4.6.35 Let L and L' be closed altss such that $L \overset{ltl'}{\approx} L'$. Then $fmaxtr(L) = fmaxtr(L')$.

Proof: Exactly like the proof of Lemma 4.1.16, except now the formulae $ch(B_i)$ are not necessarily finite. □

Lemma 4.6.36 Let L and L' be closed altss such that $L \overset{\omega ltl'}{\approx} L'$. Then $\mathit{inftr}(L) = \mathit{inftr}(L')$.

Proof: The proof follows that of Lemma 4.1.15. Suppose that $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ are closed altss such that $L \overset{ltl'}{\approx} L'$. Define $A'' = A \cup A'$. Take any infinite visible behaviour $\sigma = B_0 l_1 B_1 \ldots \in VB_\omega$. We have to show that $\sigma \in \mathit{inftr}(L)$ iff $\sigma \in \mathit{inftr}(L')$.

If σ contains a non-empty transition label $l_i \neq \tau$, then $\sigma \notin \mathit{inftr}(L)$ and $\sigma \notin \mathit{inftr}(L')$, since both L and L' are closed. Therefore, we can assume in the following that $l_i = \tau$ for all $i \geq 1$. As in 4.1.15, we construct an $\omega LTL'$-formula $ch(\sigma)$ such that for both $L^\dagger = L$ and $L^\dagger = L'$, the following holds:

$$L^\dagger \not\models \neg ch(\sigma) \quad \text{iff} \quad \sigma \in \mathit{inftr}(L^\dagger)$$

The required result follows from this as in the proof of 4.1.15.

For every finite visible behaviour $\sigma' \in VB_*$, we define $ch(\sigma')$ as in the proof of 4.1.15. From there we also remember the fact that for all behaviours $\rho = B'_0 l'_1 B'_1 \ldots \in B_\infty$ for which each $l'_i = \tau$ and $B'_i \subseteq A''$, $\rho \models ch(\sigma')$ iff $\sigma' \preceq vis(\rho)$. Define then the characteristic formula of σ by

$$ch(\sigma) \;=\; \bigwedge_{n \in \mathbb{N}} ch(\sigma[0 \ldots n])$$

Now for all infinite behaviours $\rho = B'_0 l'_1 B'_1 \ldots \in B_\omega$ for which each $l'_i = \tau$ and $B'_i \subseteq A''$, $\rho \models ch(\sigma)$ iff for all $n \in \mathbb{N}$, $\rho \models ch(\sigma[0 \ldots n])$, i.e. iff for all $n \in \mathbb{N}$, $\sigma[0 \ldots n] \preceq vis(\rho)$, i.e. iff $\sigma = vis(\rho)$. This means that for both $L^\dagger = L$ and $L^\dagger = L'$,

$L^\dagger \not\models \neg ch(\sigma)$
iff there is a $\rho \in \mathit{maxbeh}(L^\dagger)$ such that $\rho \models ch(\sigma)$ (by Definition 3.2.3)
iff there is a $\rho \in \mathit{maxbeh}(L^\dagger)$ such that $\sigma = vis(\rho)$ (by the observations above)
iff $\sigma \in \mathit{inftr}(L^\dagger)$ (by Lemma 4.1.10).
This concludes the proof. □

Theorem 4.6.37 Let L and L' be closed altss such that $L \overset{\omega ltl'}{\approx} L'$. Then $L \overset{maxtr}{\approx} L'$.

Proof: Immediate from 4.6.35 and 4.6.36. □

Now we have an exact correspondence between $\overset{\omega ltl'}{\approx}$ and $\overset{maxtr}{\approx}$ for all closed altss, not just finite ones.

Corollary 4.6.38 Let L and L' be closed altss. Then $L \overset{\omega ltl'}{\approx} L'$ iff $L \overset{maxtr}{\approx} L'$.

Proof: Immediate from 4.6.33 and 4.6.37. □

This allows us to state the full abstractness result for all altss, not just finite ones as before.

Corollary 4.6.39 Let $\stackrel{x}{\approx}$ be an equivalence between altss, such that

- $\stackrel{x}{\approx}$ is compositional with respect to $||[\Sigma]||$, $||[\emptyset]||$ and **hide** Σ **in** ,
- for all closed altss L and L', $L \stackrel{x}{\approx} L'$ implies $L \stackrel{\omega ltl'}{\approx} L'$, and
- for all altss L and L', $L \stackrel{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all altss L and L', $L \stackrel{x}{\approx} L'$ implies $L \stackrel{\mathrm{ndfd}}{\approx} L'$.

Proof: Immediate from 4.6.37 and 4.4.8. □

One comment with respect to this result might be in order. In [KV92] a full abstractness result similar to the one above was reported for $\stackrel{ltl'}{\approx}$, not $\stackrel{\omega ltl'}{\approx}$, and all labelled transition systems, not just finite ones. In the framework of that paper, where systems are modelled using standard ltss without node labelling, the result is correct. However, it hinges on the ability to synchronise with all visible transitions of a system. In the present framework this cannot be done, as a transition may well be visible, but still have an empty transition label τ that cannot be synchronised with.

We naturally also have a similar full abstractness result for $\stackrel{\mathrm{nindfd}}{\approx}$.

Corollary 4.6.40 Let $\stackrel{x}{\approx}$ be an equivalence between altss, such that

- $\stackrel{x}{\approx}$ is compositional with respect to $||[\Sigma]||$, $||[\emptyset]||$, **hide** Σ **in** and $[]$,
- for all closed altss L and L', $L \stackrel{x}{\approx} L'$ implies $L \stackrel{\omega ltl'}{\approx} L'$, and
- for all altss L and L', $L \stackrel{x}{\approx} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all altss L and L', $L \stackrel{x}{\approx} L'$ implies $L \stackrel{\mathrm{nindfd}}{\approx} L'$.

Proof: Immediate from 4.6.37 and 4.6.23. □

4.6.4 Practical considerations

Although it was shown above that NDFD-equivalence is optimal for compositional reductions in verifying LTL'-formulae, there are some aspects in which we can do better. When trying to verify that $L \models \phi$ holds, it is not necessary to preserve the validity of *all* formulae as the NDFD-equivalence does. Here we do not want to enter the general discussion on equivalences that preserve only some particular LTL'-properties, instead of all of them, as this seems an entire research area in its own right. However, a very simple way to reduce the amount of information needed for deciding whether $L \models \phi$ can be obtained by looking purely at the alphabet of ϕ, i.e. the set of atomic propositions occurring in it, and, in the case of a $tLTL'$-formula, the set of transition labels occurring in it. This method is outlined in the following proposition.

Lemma 4.6.41 Let ϕ by an LTL'-formula and let $\rho = A_0 l_1 A_1 \ldots$ and $\rho' = A'_0 l'_1 A'_1 \ldots$ be behaviours such that

- $|\rho| = |\rho'|$ and
- for all $0 \leq i \leq |\rho|$, $A_i \cap AP(\phi) = A'_i \cap AP(\phi)$

Then $\rho \models \phi$ iff $\rho' \models \phi$.

Furthermore, let ϕ be a $tLTL'$-formula and let $\rho = A_0 l_1 A_1 \ldots$ and $\rho' = A'_0 l'_1 A'_1 \ldots$ be behaviours such that

- $|\rho| = |\rho'|$,
- for all $0 \leq i \leq |\rho|$, $A_i \cap AP(\phi) = A'_i \cap AP(\phi)$, and
- for all $0 \leq i \leq |\rho|$, if $l_i \in \Sigma(\phi)$ or $l'_i \in \Sigma(\phi)$, then $l_i = l'_i$.

Then $\rho \models \phi$ iff $\rho' \models \phi$.

Proof: Transition labels and atomic propositions not occurring in ϕ cannot affect whether ϕ is valid over a behaviour or not. □

Proposition 4.6.42 Let ϕ be an LTL' or $tLTL'$-formula, L an alts, and $B \subseteq AP$ a set of atomic propositions such that $AP(\phi) \cap B = \emptyset$. Then

$$L \models \phi \qquad \text{iff} \qquad \textbf{encaps } B \textbf{ in } L \models \phi$$

Let ϕ and L be as above, and let $\Pi \subseteq \Sigma$ be a set of labels such that $\Sigma(\phi) \cap \Pi = \emptyset$. Then

$$L \models \phi \qquad \text{iff} \qquad \textbf{hide } \Pi \textbf{ in } L \models \phi$$

Proof: It is easy to see that for every $\rho = A_0 l_1 A_1 \ldots \in B_\infty$, $\rho \in \mathit{maxbeh}(\textbf{encaps } B \textbf{ in } L)$ iff there is some $\rho' = A'_0 l'_1 A'_1 \ldots \in \mathit{maxbeh}(L)$ such that

- $|\rho| = |\rho'|$,
- for all $0 \leq i \leq |\rho|$, $A_i = A'_i \setminus B$, and
- for all $1 \leq i \leq |\rho|$, $l_i = l'_i$.

Similarly, for every $\rho = A_0 l_1 A_1 \ldots \in B_\infty$, $\rho \in \mathit{maxbeh}(\textbf{hide } \Pi \textbf{ in } L)$ iff there is some $\rho' = A'_0 l'_1 A'_1 \ldots \in \mathit{maxbeh}(L)$ such that

- $|\rho| = |\rho'|$,
- for all $0 \leq i \leq |\rho|$, $A_i = A'_i$, and
- for all $1 \leq i \leq |\rho|$, either $l_i = l'_i$, or $l'_i \in \Pi$ and $l_i = \tau$.

Therefore, the claim follows immediately from 4.6.41. □

Consequently, when trying to find out whether $L \models \phi$ holds, it is unnecessary to record in the model atomic propositions and transition labels not occurring in ϕ. What is more, the following propositions imply that this information may be removed already from the component altss of a system, as long as it is not used for interaction between components.

Proposition 4.6.43 Let L, L' be altss, $\Pi \subseteq \Sigma$ and $B \subseteq AP$. Then

$$\textbf{encaps } B \textbf{ in } (L \,|[\Pi]|\, L') \overset{\text{ndfd}}{\approx} (\textbf{encaps } B \textbf{ in } L) \,|[\Pi]|\, (\textbf{encaps } B \textbf{ in } L')$$

Let L, L' be altss and $\Pi, \Pi' \subseteq \Sigma$ such that $\Pi \cap \Pi' = \emptyset$. Then

$$\textbf{hide } \Pi' \textbf{ in } L \,|[\Pi]|\, L' \overset{\text{ndfd}}{\approx} (\textbf{hide } \Pi' \textbf{ in } L) \,|[\Pi]|\, (\textbf{hide } \Pi' \textbf{ in } L')$$

Let L, L' be altss and $B \subseteq AP$, such that

$$\begin{array}{rcll} B \cap \{a \in AP \mid a \text{ occurs in some } l \in \Sigma(L)\} & = & \emptyset & \text{and} \\ B \cap \{a \in AP \mid a \text{ occurs in some } l \in \Sigma(L')\} & = & \emptyset & \end{array}$$

i.e. no proposition in B occurs in the transition labels of L or L'. Then

$$\textbf{encaps } B \textbf{ in } (L \parallel_v L') \overset{\text{ndfd}}{\approx} (\textbf{encaps } B \textbf{ in } L) \parallel_v (\textbf{encaps } B \textbf{ in } L')$$

Proof: The altss on both sides of each equation above are isomorphic. □

Proposition 4.6.44 Let L, L' be altss, For all $B, B' \subseteq AP$, $\Pi, \Pi' \subseteq \Sigma$ and $f : \Sigma \to \Sigma$:

$$\begin{array}{rcl} \textbf{encaps } B \textbf{ in } (\textbf{hide } \Pi \textbf{ in } L) & \overset{\text{ndfd}}{\approx} & \textbf{hide } \Pi \textbf{ in } (\textbf{encaps } B \textbf{ in } L) \\ \textbf{encaps } B \textbf{ in } (\textbf{encaps } B' \textbf{ in } L) & \overset{\text{ndfd}}{\approx} & \textbf{encaps } B \cup B' \textbf{ in } L \\ \textbf{hide } \Pi \textbf{ in } (\textbf{hide } \Pi' \textbf{ in } L) & \overset{\text{ndfd}}{\approx} & \textbf{hide } \Pi \cup \Pi' \textbf{ in } L \\ \textbf{encaps } B \textbf{ in } L[f] & \overset{\text{ndfd}}{\approx} & (\textbf{encaps } B \textbf{ in } L)[f] \\ \textbf{hide } \Pi \textbf{ in } L[f] & \overset{\text{ndfd}}{\approx} & (\textbf{hide } \{l \in \Sigma \mid f(l) \in \Pi\} \textbf{ in } L)[f] \end{array}$$

Proof: The altss on both sides of each equation above are isomorphic. □

A particular case of ignoring superfluous information occurs when the formula to be verified refers only to the state information of some of the processes. If the interaction taking place is synchronisation, we may now remove all the state information of all the other processes of the system.

Chapter 5

Preorders

5.1 General

The approach of compositional verification using equivalences was inspired by the idea of replacing a model L by a smaller model L' so that for all formulae ϕ, $L \models \phi$ iff $L' \models \phi$. Using preorders for the same task arises from similar intuitions, only this time the relation is no longer symmetric. This time the aim is to replace L by a smaller L' such that for all formulae ϕ the following holds: if $L' \models \phi$, then $L \models \phi$.

It is clear that if we can produce an L' as above and verify that $L' \models \phi$, then $L \models \phi$ as well. What cannot be done this time is using the inference the other way; even if $L' \not\models \phi$, it is perfectly possible that $L \models \phi$.

To use this method to show that a system lacks a required property, it can be applied symmetrically in the other direction. Suppose that we replace L by a smaller L' such that for all formulae ϕ the following holds: if $L \models \phi$, then $L' \models \phi$. If we then find out that $L' \not\models \phi$, we know that $L \not\models \phi$, i.e. the original model lacks the required property. Naturally, if we now have $L' \models \phi$, we do not know anything about whether $L \models \phi$ holds or not.

Developing the theory in the case of preorders goes almost exactly as in the case of equivalences. Therefore, we shall just list the major steps of the theory here.

Definition 5.1.1 Let X be a set and $R \subseteq X \times X$ a relation on X. We say that R is a *preorder* iff it is

- reflexive, i.e. $R(x, x)$ holds for all $x \in X$, and
- transitive, i.e. $R(x, y)$ and $R(y, z)$ imply $R(x, z)$ for all $x, y, z \in X$. □

Definition 5.1.2 Let L and L' be altss. We say that L is lower than L' in the *LTL'-preorder* and write $L \overset{ltl'}{\prec} L'$ iff for all LTL'-formulae ϕ, if $L' \models \phi$ then $L \models \phi$.

Similarly, we say that L is lower than L' in the *$tLTL'$-preorder* and write $L \overset{tltl'}{\prec} L'$ iff for all $tLTL'$-formulae ϕ, if $L' \models \phi$ then $L \models \phi$. □

Definition 5.1.3 Let L and L' be altss. We say that L is lower than L' in the *maximal trace preorder* and write $L \overset{maxtr}{\prec} L'$ iff $maxtr(L) \subseteq maxtr(L')$ □

Theorem 5.1.4 Let L and L' be altss such that $L \overset{maxtr}{\prec} L'$. Then $L \overset{ltl'}{\prec} L'$ and $L \overset{tltl'}{\prec} L'$.

Proof: Like the proof of 4.1.12, but in one direction only. □

The reverse also holds for closed altss.

Theorem 5.1.5 Let L and L' be finite closed altss such that $L \overset{ltl'}{\prec} L'$. Then $L \overset{maxtr}{\prec} L'$.

Proof: Like the proof of 4.1.18, but in one direction only. □

Corollary 5.1.6 Let L and L' be finite closed altss. Then $L \overset{ltl'}{\prec} L'$ iff $L \overset{maxtr}{\prec} L'$.

Proof: Immediate from 5.1.4 and 5.1.5. □

The development of compositional preorders follows that of equivalences.

Definition 5.1.7 Let $\overset{x}{\prec}$ be a preorder relation on the class of altss, and f an n-place syntactic operator constructing altss from n component altss. We say that $\overset{x}{\prec}$ *is compositional with respect to* f iff for all altss $L_1, \ldots, L_n$, $L'_1, \ldots, L'_n$ such that for all $1 \leq i \leq n$, $L_i \overset{x}{\prec} L'_i$:

- $f(L_1, \ldots, L_n)$ is defined iff $f(L'_1, \ldots, L'_n)$ is defined
- if both are defined, $f(L_1, \ldots, L_n) \overset{x}{\prec} f(L'_1, \ldots, L'_n)$. □

Notice above that we require $f(L_1, \ldots, L_n)$ to be defined iff $f(L'_1, \ldots, L'_n)$ is defined. This may seem unnecessarily strong; we might require the implication to hold in one direction only. However, if we wanted to take this approach, it is not clear in which direction the implication should be required to hold.

Suppose we required only that if $f(L_1, \ldots, L_n)$ is defined, then $f(L'_1, \ldots, L'_n)$ is defined. Then we could have a situation in which $f(L_1, \ldots, L_n)$ is not defined, but $f(L'_1, \ldots, L'_n)$ is. Remembering the aim of the approach, we might now verify that $f(L'_1, \ldots, L'_n) \models \phi$ for some required property ϕ. What we would like to conclude from $f(L'_1, \ldots, L'_n) \models \phi$ is that $f(L_1, \ldots, L_n) \models \phi$ holds. However, in this case the conclusion would not be correct, since $f(L_1, \ldots, L_n)$ does not even exist.

Suppose then that we required the implication to hold only in the other direction, i.e. that if $f(L'_1, \ldots, L'_n)$ is defined, then $f(L_1, \ldots, L_n)$ is defined. Now we would always be justified in inferring $f(L_1, \ldots, L_n) \models \phi$ from $f(L'_1, \ldots, L'_n) \models \phi$. However, we could have a situation in which $f(L_1, \ldots, L_n)$ is defined but $f(L'_1, \ldots, L'_n)$ is not. But this would be contrary to the intuition according to which each L'_i is an approximation or abstraction of L_i; if L_i can be used in some place to create a system, certainly replacing L_i by L'_i should also give rise to a valid system.

As there seem to be arguments against requiring the implication to hold in one direction only, no matter which direction we choose, we have chosen to require that $f(L_1, \ldots, L_n)$ and $f(L'_1, \ldots, L'_n)$ are either both defined or both undefined.

We can now define the preorder corresponding to NDFD-equivalence.

Definition 5.1.8 Let L_1 and L_2 be altss. We say that L_1 is lower than L_2 in *the non-divergent failures divergences preorder* or *NDFD-preorder* and write $L_1 \overset{\text{ndfd}}{\prec} L_2$ iff

- $\Sigma(L_1) = \Sigma(L_2)$,
- $AP(L_1) = AP(L_2)$,
- $\mathit{inftr}(L_1) \subseteq \mathit{inftr}(L_2)$,
- $\mathit{divtr}(L_1) \subseteq \mathit{divtr}(L_2)$, and
- $\mathit{ndfail}(L_1) \subseteq \mathit{ndfail}(L_2) \cup (\mathit{divtr}(L_2) \times P(\Sigma))$. □

Notice above that some nondivergent failures of the lower alts L_1 may not correspond to nondivergent failures of the greater alts L_2 but to divergences of it, instead.

Let us see first that the NDFD-equivalence and the NDFD-preorder are definable in terms of each other.

Proposition 5.1.9 Let L and L' be altss. Then

$$L \overset{\text{ndfd}}{\approx} L' \quad \text{iff} \quad L \overset{\text{ndfd}}{\prec} L' \text{ and } L' \overset{\text{ndfd}}{\prec} L$$

Proof: Immediate from definitions. □

Proposition 5.1.10 Let L and L' be altss. Then

$$\begin{array}{lll} L \overset{\text{ndfd}}{\prec} L' & \text{iff} & L\,[:]\,L' \overset{\text{ndfd}}{\approx} L', \\ & & \Sigma(L) = \Sigma(L') \quad \text{and} \\ & & AP(L) = AP(L') \end{array}$$

Proof: Suppose that $L \overset{\text{ndfd}}{\prec} L'$. Then by definition

$$\begin{array}{rcl} \Sigma(L) & = & \Sigma(L') \\ AP(L) & = & AP(L') \\ \mathit{divtr}(L) & \subseteq & \mathit{divtr}(L') \\ \mathit{inftr}(L) & \subseteq & \mathit{inftr}(L') \\ \mathit{ndfail}(L) & \subseteq & \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma)) \end{array}$$

Consequently,

$$\begin{array}{rcl} \Sigma(L\,[:]\,L') & = & \Sigma(L) \cup \Sigma(L') \\ & = & \Sigma(L') \\ AP(L\,[:]\,L') & = & AP(L) \cup AP(L') \\ & = & AP(L') \\ \mathit{divtr}(L\,[:]\,L') & = & \mathit{divtr}(L) \cup \mathit{divtr}(L') \\ & = & \mathit{divtr}(L') \\ \mathit{inftr}(L\,[:]\,L') & = & \mathit{inftr}(L) \cup \mathit{inftr}(L') \\ & = & \mathit{inftr}(L') \end{array}$$

$$\begin{array}{rcl}
ndfail(L\,[:]\,L') & = & (ndfail(L) \cup ndfail(L')) \setminus divtr(L\,[:]\,L') \times P(\Sigma) \\
& = & (ndfail(L) \cup ndfail(L')) \setminus divtr(L') \times P(\Sigma) \\
& = & ndfail(L') \setminus divtr(L') \times P(\Sigma) \\
& = & ndfail(L')
\end{array}$$

i.e. $L\,[:]\,L' \stackrel{\mathrm{ndfd}}{\approx} L'$.

Suppose then that $L\,[:]\,L' \stackrel{\mathrm{ndfd}}{\approx} L'$, $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$. This means that

$$\begin{array}{rcl}
divtr(L\,[:]\,L') & = & divtr(L') \\
inftr(L\,[:]\,L') & = & inftr(L') \\
ndfail(L\,[:]\,L') & = & ndfail(L')
\end{array}$$

Notice then that as above

$$\begin{array}{rcl}
divtr(L\,[:]\,L') & = & divtr(L) \cup divtr(L') \\
inftr(L\,[:]\,L') & = & inftr(L) \cup inftr(L') \\
ndfail(L\,[:]\,L') & = & (ndfail(L) \cup ndfail(L')) \setminus divtr(L\,[:]\,L') \times P(\Sigma) \\
& = & (ndfail(L) \setminus divtr(L\,[:]\,L') \times P(\Sigma)) \;\cup \\
& & (ndfail(L') \setminus divtr(L\,[:]\,L') \times P(\Sigma)) \\
& = & (ndfail(L) \setminus divtr(L') \times P(\Sigma)) \;\cup \\
& & (ndfail(L') \setminus divtr(L') \times P(\Sigma)) \\
& = & (ndfail(L) \setminus divtr(L') \times P(\Sigma)) \cup ndfail(L')
\end{array}$$

Consequently,

$$\begin{array}{rcl}
divtr(L) & \subseteq & divtr(L') \\
inftr(L) & \subseteq & inftr(L') \\
ndfail(L) & \subseteq & ndfail(L') \cup divtr(L') \times P(\Sigma)
\end{array}$$

Since by assumption $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$, we know that $L \stackrel{\mathrm{ndfd}}{\prec} L'$. □

For a similar characterisation of the relation between a related failure preorder and equivalence, see [Bro83, Prop. 1.3.9].

We have a series of results concerning NDFD-preorder, similar to those concerning NDFD-equivalence.

Lemma 5.1.11 Let L_1 and L_2 be altss such that $L_1 \stackrel{\mathrm{ndfd}}{\prec} L_2$. Then $L_1 \stackrel{maxtr}{\prec} L_2$.

Proof: As in 4.2.8. □

Proposition 5.1.12 Let L_1 and L_2 be altss such that $L_1 \stackrel{\mathrm{ndfd}}{\prec} L_2$. Then $L_1 \stackrel{ltl'}{\prec} L_2$ and $L_1 \stackrel{tltl'}{\prec} L_2$.

Proof: Immediate from 5.1.11 and 5.1.4. □

Theorem 5.1.13 The NDFD-preorder $\overset{\text{ndfd}}{\prec}$ is compositional with respect to the operations

- $\ldots|[\Pi]|\ldots$
- $\ldots\|_s\ldots$
- $\ldots\|_v\ldots$
- **hide** Π **in** $\ldots$
- $\ldots[f]$
- **encaps** B **in** $\ldots$
- $(B \xrightarrow{l}); \ldots$
- $\ldots \gg \ldots$ and
- $\ldots[:]\ldots$

for all $\Pi \subseteq \Sigma$, $f : \Sigma \to \Sigma$, $l \in \Sigma$ and $B \subseteq AP$.

Proof: As the proof of 4.6.21, but in one direction only. □

Theorem 5.1.14 Let $\overset{x}{\prec}$ be a preorder between altss, such that

- $\overset{x}{\prec}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$ and **hide** Σ **in**
- for all closed (finite) altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{maxtr}{\prec} L'$, and
- for all (finite) altss L and L', $L \overset{x}{\prec} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all (finite) altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{\text{ndfd}}{\prec} L'$.

Proof: As the proof of 4.4.8, but in one direction only. □

Corollary 5.1.15 Let $\overset{x}{\prec}$ be an equivalence between altss, such that

- $\overset{x}{\prec}$ is compositional with respect to $|[\Sigma]|$, $|[\emptyset]|$ and **hide** Σ **in**
- for all closed finite altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{ltl'}{\prec} L'$, and
- for all finite altss L and L', $L \overset{x}{\prec} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all finite altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{\text{ndfd}}{\prec} L'$.

Proof: Immediate from 5.1.14 and 5.1.5. □

Determining whether $L \overset{\text{ndfd}}{\prec} L'$ for given finite altss can be done using the same techniques as for the NDFD-equivalence in Section 4.5. Unsurprisingly, this problem is also PSPACE-complete.

Theorem 5.1.16 Determining whether $L \overset{\text{ndfd}}{\prec} L'$ for given finite altss L and L' is a PSPACE-hard problem.

Proof: By 4.5.1, determining whether $L \overset{\text{ndfd}}{\approx} L'$ is a PSPACE-hard problem. By 5.1.9, checking for $L \overset{\text{ndfd}}{\approx} L'$ can be reduced to checking $L \overset{\text{ndfd}}{\prec} L'$ and $L' \overset{\text{ndfd}}{\prec} L$. This reduction is clearly polynomial. □

check NDFD non-preorder (L, L')

input: finite altss $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$

accept: iff $L \overset{\text{ndfd}}{\not\prec} L'$

begin

choose nondeterministically to execute one of the following:

check trace (L, L') /* see Fig. 4.9 */
check divergence trace (L, L') /* see Fig. 4.11 */
check failure (L, L') /* see Fig. 4.12 */

end

Figure 5.1: Checking NDFD non-preorder

Lemma 5.1.17 The nondeterministic algorithm *check NDFD non-preorder* (L, L') in Figure 5.1 accepts iff $L \overset{\text{ndfd}}{\not\prec} L'$, and works in $O(|L'| + \log |L|)$ space.

Proof: By 4.5.3, 4.5.5 and 4.5.6, the algorithm *check NDFD non-preorder* (L, L') rejects iff $\mathit{tr}(L) \subseteq \mathit{tr}(L')$, $\mathit{divtr}(L) \subseteq \mathit{divtr}(L')$ and $\mathit{sfail}(L) \subseteq \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma))$. Therefore, if the algorithm rejects, we have $L \overset{\text{ndfd}}{\prec} L'$, since $\mathit{ndfail}(L) \subseteq \mathit{sfail}(L)$. Conversely, if $L \overset{\text{ndfd}}{\prec} L'$, we have $\mathit{tr}(L) \subseteq \mathit{tr}(L')$, $\mathit{divtr}(L) \subseteq \mathit{divtr}(L')$ and

$$\mathit{sfail}(L) \subseteq \mathit{ndfail}(L) \cup (\mathit{divtr}(L) \times P(\Sigma)) \subseteq \mathit{ndfail}(L') \cup (\mathit{divtr}(L') \times P(\Sigma))$$

and the algorithm rejects. The space bound follows directly from 4.5.3, 4.5.5 and 4.5.6. □

Theorem 5.1.18 Determining whether $L \overset{\text{ndfd}}{\prec} L'$ for given finite altss L and L' can be done deterministically in in $O(|L'|^2 + \log^2 |L|)$ space and $O(2^{O(|L'|^2 + \log^2 |L|)})$ time.

Proof: By applying Savitch's theorem [Sav70] (see also e.g. [BDG88, Cor. 2.1]) to the algorithm of Lemma 5.1.17 as in the proof of 4.5.8. Notice also that $O((|L'| + \log |L|)^2)$ is $O(|L'|^2 + \log^2 |L|)$. □

Corollary 5.1.19 Determining whether $L \overset{\text{ndfd}}{\prec} L'$ for given finite altss L and L' is a PSPACE-complete problem.

Proof: Immediate from 5.1.16 and 5.1.18. □

As with the equivalence checking algorithms, we can obtain an algorithm that is faster but uses more space using a different determinisation construction. Notice that the resource bound of the following theorem, while still exponential in the size of L', is only linear in the size of L. Intuitively, this means that when approximating a module L by an abstraction L', we can check $L \stackrel{\text{ndfd}}{\prec} L'$ effectively if the approximation L' is concise enough.

Theorem 5.1.20 Determining whether $L \stackrel{\text{ndfd}}{\prec} L'$ for given finite altss L and L' can be done deterministically in $O(|L| \cdot 2^{|L'|})$ space and time.

Proof: Similar to the proof of 5.1.18, but using the construction of [BDG88, Theorem 2.8 (g)]. □

We can also extend these results in the same fashion as in Section 4.6 for the NDFD-equivalence.

Definition 5.1.21 Let L_1 and L_2 be altss. We say that L_1 is lower than L_2 in the *(non-initial) non-divergent failures divergences preorder* or *NINDFD-preorder* and write $L_1 \stackrel{\text{nindfd}}{\prec} L_2$ iff

- $L_1 \stackrel{\text{ndfd}}{\prec} L_2$, and
- $nindfail(L_1) \subseteq nindfail(L_2) \cup (divtr(L_2) \times P(\Sigma))$. □

Theorem 5.1.22 The NINDFD-preorder $\stackrel{\text{nindfd}}{\prec}$ is compositional with respect to all the operations listed in 5.1.13 and ... [] ...

Proof: As in 4.6.21, but in one direction only. □

Theorem 5.1.23 Let $\stackrel{x}{\prec}$ be a preorder between altss, such that

- $\stackrel{x}{\prec}$ is compositional with respect to $||[\Sigma]||$, $||[\emptyset]||$, **hide** Σ **in** , and []
- for all closed (finite) altss L and L', $L \stackrel{x}{\prec} L'$ implies $L \stackrel{maxtr}{\prec} L'$, and
- for all (finite) altss L and L', $L \stackrel{x}{\prec} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all (finite) altss L and L', $L \stackrel{x}{\prec} L'$ implies $L \stackrel{\text{nindfd}}{\prec} L'$.

Proof: As in 4.6.23, but in one direction only. □

Corollary 5.1.24 Let $\stackrel{x}{\prec}$ be a preorder between altss, such that

- $\stackrel{x}{\prec}$ is compositional with respect to $||[\Sigma]||$, $||[\emptyset]||$, **hide** Σ **in** and []
- for all closed finite altss L and L', $L \stackrel{x}{\prec} L'$ implies $L \stackrel{ltl'}{\prec} L'$, and
- for all finite altss L and L', $L \stackrel{x}{\prec} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all finite altss L and L', $L \stackrel{x}{\prec} L'$ implies $L \stackrel{\text{nindfd}}{\prec} L'$.

Proof: Immediate from 5.1.5 and 5.1.23. □

The results for the logic $\omega LTL'$ and NDFD-equivalence also generalise to the NDFD-preorder.

Definition 5.1.25 Let L and L' be altss. We say that L is lower than L' in the $\omega LTL'$-*preorder* and write $L \stackrel{\omega ltl'}{\prec} L'$ iff for all $\omega LTL'$-formulae ϕ, if $L' \models \phi$, then $L \models \phi$. □

Theorem 5.1.26 Let L and L' be closed altss such that $L \overset{\omega ltl'}{\prec} L'$. Then $L \overset{maxtr}{\prec} L'$.

Proof: Like the proof of 4.6.37, but in one direction only. □

Corollary 5.1.27 Let $\overset{x}{\prec}$ be a preorder between altss, such that

- $\overset{x}{\prec}$ is compositional with respect to $||[\Sigma]||$, $||[\emptyset]||$ and **hide** Σ **in**
- for all closed altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{\omega ltl'}{\prec} L'$, and
- for all altss L and L', $L \overset{x}{\prec} L'$ implies $\Sigma(L) = \Sigma(L')$ and $AP(L) = AP(L')$.

Then for all altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{\text{ndfd}}{\prec} L'$.

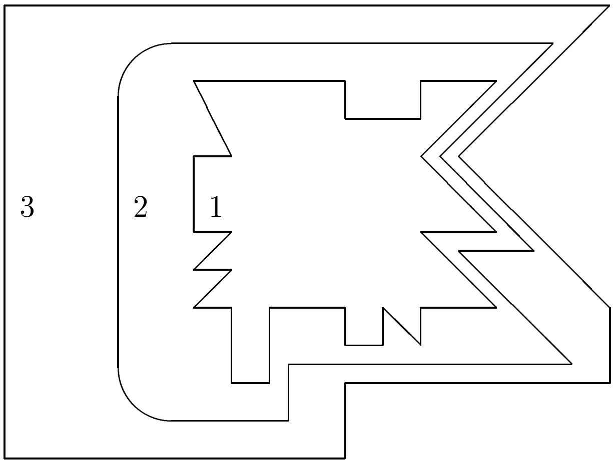
Furthermore, if $\overset{x}{\prec}$ is also compositional with respect to $[]$, then for all altss L and L', $L \overset{x}{\prec} L'$ implies $L \overset{\text{nindfd}}{\prec} L'$.

Proof: Immediate from 5.1.14, 5.1.23 and 5.1.26. □

When the NDFD-equivalence is used for verification with compositional reductions, our strategy is straightforward: Try to replace each module by one that is equivalent to it but as small in size as possible. However, in the case of the NDFD-preorder the situation is not quite as clear. What we would naturally like to do is to replace each module by another that is greater in the preorder and as small in size as possible. However, not all the altss that are greater in the preorder are actually usable; we would still like to verify that the resulting system fulfils the required property, and this cannot necessarily be done with all the possible replacements. Intuitively, the greater in the NDFD-preorder an alts is, the more general it is, and the fewer particular properties it validates. This means that we are faced with a balancing act between how much greater in the NDFD-preorder a replacement is, against how much smaller in size it is when compared to the original module.

The task of finding a replacement can be seen as an abstraction step, where we try to replace a component of a system by an approximation that reflects the relevant characteristics of the component while ignoring the irrelevant ones. To reiterate, the NDFD-equivalence preserves all characteristics of a component that may be relevant to the preservation of some LTL'-property in some environment. However, when working with the NDFD-preorder, the verifier can choose to ignore some of those characteristics that do not contribute to the fulfilment of the required property in the actual environment. This means that the amount of information preserved in the replacement is smaller and may be expressed more concisely than in equivalence-based replacements. This process of approximating away irrelevant characteristics of the system while preserving those characteristics that are important for fulfilling the specification is illustrated in Figure 5.2.

Generally, this abstraction step requires an intuitive understanding of how the system is supposed to work and what the crucial aspects of the system for a required property are. This can be seen both as a positive and a negative feature. On the one hand it allows the verifier to use her intuition in finding suitable small approximations, making it possible to verify systems beyond the reach of the equivalence-based approach. On the other hand, applying the technique also requires this intuitive step, which seems unlikely to be feasibly automatisable. Consequently, the approach does not lead to a fully automatisable verification method.



1 = the behaviours produced by the system
2 = the behaviours of the abstraction
3 = the behaviours fulfilling the specification

Figure 5.2: Approximating the behaviours of a system

To aid the process of finding good replacements, various heuristics can be developed. These can naturally never guarantee success in the verification process, but can help. Some heuristics that were applied in the case studies of this work were making a system less deterministic, adding divergences to a system and deleting information about possible behaviours of subsystems in situations that should never arise in the context of the complete system.

5.2 Modular validity

In this section we look at one particular heuristic to finding useful approximations for components of a system, and show that it naturally corresponds to Manna's and Pnueli's concept of *modular validity* or validity in all environments. This means that we can view modular validity as a particular extremity in compositional verification with the NDFD-preorder.

The heuristic is based on singling out one component of a system and replacing all the rest with an approximation which is as coarse as possible, behaving completely arbitrarily. To be more precise, we view a system as being of the form $L \,||[\Pi]||\, L_{env}$ or $L \,||_v\, L_{env}$ and replace L_{env} with an NDFD-maximal approximation.

Definition 5.2.1 Let L be an alts. The *NDFD-maximal approximation* for L is defined as the alts $maxappr(L) = (S, I, \Gamma, \Delta, A, V)$ where

$$\begin{array}{rcl} S & = & P(AP(L)) \\ I & = & S \\ \Gamma & = & \Sigma(L) \\ \Delta & = & S \times (\Sigma(L) \cup \{\tau\}) \times S \\ A & = & AP(L) \\ V & : & S \to P(A); s \mapsto s \end{array}$$

□

Notice that $L \overset{\text{ndfd}}{\prec} maxappr(L)$ and $L' \overset{\text{ndfd}}{\prec} maxappr(L)$ for every L' such that $L \overset{\text{ndfd}}{\prec} L'$, which justifies calling $maxappr(L)$ maximal. Notice also that $maxappr(L)$ depends only on the set of atomic propositions of L and the set of transition labels of L, and that it allows all possible behaviours with truth sets from $AP(L)$ and transition labels from $\Sigma(L) \cup \{\tau\}$.

The idea of verifying that ϕ holds of a system when the environment of a component L behaves completely arbitrarily is very close to the idea of modular validity introduced in [MP91]. In this scenario one verifies that ϕ holds of all the systems consisting of L and an arbitrary environment. In the current framework this concept can be formulated as follows.

Definition 5.2.2 Let L be an alts, ϕ an LTL'-formula and $\Pi \subseteq \Sigma$. We say that ϕ is *modularly valid over L with respect to synchronisation by* Π and write $L \models_m^{\Pi} \phi$ iff $L|[\Pi]|L' \models \phi$ for all altss L' such that $L|[\Pi]|L'$ is well-defined and $AP(\phi) \subseteq AP(L|[\Pi]|L')$. □

Definition 5.2.3 Let L be a condition-labelled alts and ϕ an LTL'-formula. We say that ϕ is *modularly valid over L with respect to interaction by variables* and write $L \models_m^{v} \phi$ iff $L \parallel_v L' \models \phi$ for all altss L' such that $L \parallel_v L'$ is well-defined and $AP(\phi) \subseteq AP(L \parallel_v L')$. □

The motivation behind the concept of modular validity is to avoid the state explosion problem by allowing compositional reasoning over systems. For example, in verifying $L_1 \parallel_v L_2 \models \phi$, the aim is to find formulae ϕ_1 and ϕ_2 such that $L_1 \models_m^v \phi_1$, $L_2 \models_m^v \phi_2$ and $\models \phi_1 \wedge \phi_2 \Rightarrow \phi$. By the following easy propositions this guarantees $L_1 \parallel_v L_2 \models \phi$.

Proposition 5.2.4 Let L_1 and L_2 be altss, $\Pi \subseteq \Sigma$ and ϕ_1 and ϕ_2 LTL'-formulae such that $AP(\phi_1) \subseteq AP(L_1 \,|[\Pi]|\, L_2)$ and $AP(\phi_2) \subseteq AP(L_1 \,|[\Pi]|\, L_2)$.

If $L_1 \models_m^{\Pi} \phi_1$ and $L_2 \models_m^{\Pi} \phi_2$ then $L_1 \,|[\Pi]|\, L_2 \models_m^{\Pi} \phi_1 \wedge \phi_2$.

Proof: Direct from the definition of $\models_m^{\Pi}$. □

Proposition 5.2.5 Let L_1 and L_2 be condition-labelled altss and ϕ_1 and ϕ_2 LTL'-formulae such that $AP(\phi_1) \subseteq AP(L_1 \parallel_v L_2)$ and $AP(\phi_2) \subseteq AP(L_1 \parallel_v L_2)$.

If $L_1 \models^v_m \phi_1$ and $L_2 \models^v_m \phi_2$ then $L_1 \parallel_v L_2 \models^v_m \phi_1 \wedge \phi_2$.

Proof: Direct from the definition of $\models^v_m$. □

Proposition 5.2.6 Let L be an alts and ϕ an LTL'-formula. If $L \models^\Pi_m \phi$ for any $\Pi \subseteq \Sigma$, or $L \models^v_m \phi$, then $L \models \phi$.

Proof: Direct from the definitions of $\models^\Pi_m$ and $\models^v_m$. □

In a more general perspective, both compositional verification using modular validity and that using NDFD-preorder and approximants can be seen as particular instances of the general approach of abstracting from a module the characteristics relevant for the required property and combining these abstractions to verify the required property in the level of the whole system. For example, suppose we are verifying that $L_1 \parallel_v L_2 \models \phi$. In modular validity the properties of L_i relevant for ϕ are expressed by the formula ϕ_i, and combining these abstractions in the level of the whole system takes place with the logical $\wedge$-operator. In using NDFD-preorder, the relevant properties of L_i are expressed by an approximation L'_i and these are combined with the $\parallel_v$-operator.

The following two theorems spell out the precise relation between modular validity and NDFD-maximal approximations. Theee proof of the first one is really rather simple. However, we have written it down in some detail, since the same priciples are used in the proof of Theorem 5.2.8, which is not quite so straightforward.

Theorem 5.2.7 Let L_1 and L_2 be altss, $\Pi \subseteq \Sigma$ and ϕ an LTL'-formula such that $AP(\phi) \subseteq AP(L_1 \,|[\Pi]|\, L_2)$ and $\Pi \subseteq \Sigma(L_2)$. Then

$$L_1 \models^\Pi_m \phi \;\text{ iff }\; L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2) \models \phi$$

Proof: If $L_1 \models^\Pi_m \phi$ then $L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2) \models \phi$ by the definition of $\models^\Pi_m$.

Suppose then that not $L_1 \models^\Pi_m \phi$, i.e. that there is an alts $L^\dagger$ such that $AP(\phi) \subseteq AP(L_1) \cup AP(L^\dagger)$ and $L_1 \,|[\Pi]|\, L^\dagger \not\models \phi$, i.e. $\rho \not\models \phi$ for some $\rho \in \mathit{maxbeh}(L_1 \,|[\Pi]|\, L^\dagger)$.

In the rest of the proof we shall define a behaviour $\rho' \in \mathit{beh}(L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2))$ such that $\rho \models \phi$ iff $\rho' \models \phi$. As every state of $L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2)$ is divergent, we have

$$\mathit{beh}(L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2)) = \mathit{maxbeh}(L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2))$$

As $\rho \not\models \phi$ by assumption, we have then $\rho' \in \mathit{maxbeh}(L_1|[\Pi]|\mathit{maxappr}(L_2))$ such that $\rho' \not\models \phi$. This implies $L_1 \,|[\Pi]|\, \mathit{maxappr}(L_2) \not\models \phi$, proving the other direction of the theorem.

Let us write $\rho = A_0 l_1 A_2 \ldots l_n A_n \ldots$ By projecting the transition sequence underlying ρ in $L_1 \,|[\Pi]|\, L^\dagger$ to transition sequences in the components L_1 and $L^\dagger$, we can project ρ to behaviours $\pi \in \mathit{beh}(L_1)$ and $\sigma \in \mathit{beh}(L^\dagger)$. To be more exact, there are $\pi = B_0 k_1 B_1 \ldots \in \mathit{beh}(L_1)$ and $\sigma = C_0 h_1 C_1 \ldots \in \mathit{beh}(L^\dagger)$ and projection functions $p_1 : \{i \mid i \leq |\rho|\} \to \{i \mid i \leq |\pi|\}$ and $p_2 : \{i \mid i \leq |\rho|\} \to \{i \mid i \leq |\sigma|\}$ such that

- $p_1(0) = p_2(0) = 0$
- for all $1 \leq i \leq |\rho|$, if $l_i \in \Pi$ then $p_1(i) = p_1(i-1)+1$ and $p_2(i) = p_2(i-1)+1$, otherwise either $p_1(i) = p_1(i-1)+1$ and $p_2(i) = p_2(i-1)$, or $p_1(i) = p_1(i-1)$ and $p_2(i) = p_2(i-1)+1$.
- for all $0 \leq i \leq |\rho|$, $A_i = B_{p_1(i)} \cup C_{p_2(i)}$
- for all $1 \leq i \leq |\rho|$, if $p_1(i) = p_1(i-1)+1$ then $l_i = k_{p_1(i)}$, and if $p_2(i) = p_2(i-1)+1$ then $l_i = h_{p_2(i)}$.

Let us define $\rho' = A'_0 l'_1 A'_1 \ldots$ by $|\rho'| = |\rho|$ and

- for all $0 \leq i \leq |\rho|$, $A'_i = A_i \cap (AP(L_1) \cup AP(L_2))$
- for all $1 \leq i \leq |\rho|$, if $p_2(i) = p_2(i-1)$ or $l_i \in \Sigma(L_2)$ then $l'_i = l_i$, otherwise $l'_i = \tau$

By assumption $AP(\phi) \subseteq AP(L_1) \cup AP(L_2)$, implying $A_i \cap AP(\phi) = A'_i \cap AP(\phi)$ for all $0 \leq i \leq |\rho|$. By Lemma 4.6.41 this implies $\rho \models \phi$ iff $\rho' \models \phi$.

To show $\rho' \in beh(L_1 \,|[\Pi]|\, maxappr(L_2))$, define a $\sigma' = C'_0 h'_1 C'_1 \ldots \in beh(maxappr(L_2))$ such that $|\sigma'| = |\sigma|$,

- for all $0 \leq i \leq |\rho'|$, $A'_i = B_{p_1(i)} \cup C'_{p_2(i)}$, and
- for all $1 \leq i \leq |\rho'|$, if $p_2(i) = p_2(i-1)+1$ then $l'_i = h'_{p_2(i)}$.

This means that we can combine the transition sequence underlying π in L_1 with the transition sequence underlying σ' in $maxappr(L_2)$ in the way described by the projection functions p_1 and p_2 to show $\rho' \in beh(L_1 \,|[\Pi]|\, maxappr(L_2))$. The behaviour σ' is defined by $|\sigma'| = |\sigma|$ and

- for all $0 \leq i \leq |\sigma'|$, $C'_i = C_i \cap AP(L_2)$, and
- for all $1 \leq i \leq |\sigma'|$, if $h_i \in \Sigma(L_2)$ then $h'_i = h_i$, otherwise $h_i = \tau$.

Since each $C'_i \subseteq AP(L_2)$ and $h'_i \in \Sigma(L_2) \cup \{\tau\}$, we have $\sigma' \in beh(maxappr(L_2))$. It is easy to see that σ' fulfils the required properties. $\square$

Theorem 5.2.8 Let L_1 and L_2 be condition-labelled altss and ϕ an LTL'-formula such that $AP(\phi) \subseteq AP(L_1 \,\|_v\, L_2)$. Then

$$L_1 \models^v_m \phi \;\text{ iff }\; L_1 \,\|_v\, maxappr(L_2) \models \phi$$

Proof: If $L_1 \models^v_m \phi$ then $L_1 \,\|_v\, maxappr(L_2) \models \phi$ by definition of $\models^v_m$.

Suppose then that not $L_1 \models^v_m \phi$ i.e. that there is an alts $L^\dagger$ such that $AP(\phi) \subseteq AP(L_1) \cup AP(L^\dagger)$ and a behaviour $\rho \in maxbeh(L_1 \,\|_v\, L^\dagger)$ such that $\rho \not\models \phi$. As in the proof of 5.2.7, we use ρ to define a behaviour $\rho' \in beh(L_1 \,\|_v\, maxappr(L_2))$ such that $\rho \models \phi$ iff $\rho' \models \phi$. The other direction of the theorem follows from this as in the proof of 5.2.7.

Let us write $\rho = A_0 l_1 A_2 \ldots l_n A_n \ldots$ By projecting the transition sequence underlying ρ in $L_1 \,\|_v\, L^\dagger$ to transition sequences in the components L_1 and $L^\dagger$, we can project ρ to behaviours $\pi \in beh(L_1)$ and $\sigma \in beh(L^\dagger)$. To be more exact, there are $\pi = B_0 k_1 B_1 \ldots \in beh(L_1)$ and $\sigma = C_0 h_1 C_1 \ldots \in beh(L^\dagger)$ and projection functions $p_1 : \{i \mid i \leq |\rho|\} \to \{i \mid i \leq |\pi|\}$ and $p_2 : \{i \mid i \leq |\rho|\} \to \{i \mid i \leq |\sigma|\}$ such that

- $p_1(0) = p_2(0) = 0$
- for all $1 \leq i \leq |\rho|$, either $p_1(i) = p_1(i-1)+1$ and $p_2(i) = p_2(i-1)$, or $p_1(i) = p_1(i-1)$ and $p_2(i) = p_2(i-1)+1$.
- for all $0 \leq i \leq |\rho|$, $A_i = B_{p_1(i)} \cup C_{p_2(i)}$
- for all $1 \leq i \leq |\rho|$, if $p_1(i) = p_1(i-1)+1$ then $eval(AP(L^\dagger), C_{p_2(i-1)})(k_{p_1(i)}) = l_i \neq \perp$, and if $p_2(i) = p_2(i-1)+1$ then $eval(AP(L_1), B_{p_1(i-1)})(h_{p_2(i)}) = l_i \neq \perp$.

As all the transition labels l_i, k_i, h_i are condition expressions (A^+, A^-) with A^+, A^- sets of atomic propositions, we use the notation $l_i^+, l_i^-, k_i^+, k_i^-, h_i^+, h_i^-$ to denote the components of these condition expressions. We also use the notation $\lfloor m \rfloor = \max\{n \in \mathbb{N} \mid n \leq m\}$.

Defining ρ' in the basis of ρ is more complicated than in the proof of 5.2.7. The essential problem is that there is no reason why $L^\dagger$ and $maxappr(L_2)$ would have the same set of atomic propositions, which means that the transition conditions of L_1 are evaluated in a different context in $L^\dagger$ and $maxappr(L_2)$. Because of this, ρ' cannot emulate each step of ρ by a corresponding step, but needs two steps. If a step ρ_i corresponds to a step of L_1, i.e. if $p_1(i) = p_1(i-1)+1$, then ρ' emulates this by first executing a step of $maxappr(L_2)$ to a situation where the step ρ_i of L_1 is possible and then executing a step of L_1 corresponding to ρ_i. If a step ρ_i corresponds to a step of $L^\dagger$, i.e. if $p_2(i) = p_2(i-1)+1$, then ρ' emulates this by first executing an idle step of $maxappr(L_2)$ and then executing a step of $maxappr(L_2)$ corresponding to the step ρ_i of $L^\dagger$.

Writing $\rho' = A'_0 l'_1 A'_1 \ldots$ the relation between ρ and ρ' guaranteeing $\rho \models \phi$ iff $\rho' \models \phi$ is:

a) $|\rho'| = 2 \cdot |\rho|$,

b) for all $0 \leq i \leq |\rho'|$,

$$A'_i \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi))) = A_{\lfloor i/2 \rfloor} \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi)))$$

These imply $\rho \models \phi$ iff $\rho' \models \phi$ on the basis of 4.6.41 and 4.1.6.

Simultaneously with ρ' we define two projection functions p'_1 and p'_2 and a behaviour $\sigma' = C'_0 h'_1 C'_1 \ldots$ such that

c) $p'_1(0) = p'_2(0) = 0$

d) for all $1 \leq i \leq |\rho'|$, either $p'_1(i) = p'_1(i-1)+1$ and $p'_2(i) = p'_2(i-1)$, or $p'_1(i) = p'_1(i-1)$ and $p'_2(i) = p'_2(i-1)+1$.

e) for all $0 \leq i \leq |\rho'|$, $A'_i = B_{p'_1(i)} \cup C'_{p'_2(i)}$

f) for all $1 \leq i \leq |\rho'|$, if $p'_1(i) = p'_1(i-1)+1$ then
$eval(AP(L_2), C'_{p'_2(i-1)})(k_{p'_1(i)}) = l'_i \neq \perp$,
and if $p'_2(i) = p'_2(i-1)+1$ then $eval(AP(L_1), B_{p'_1(i-1)})(h'_{p'_2(i)}) = l'_i \neq \perp$.

If we manage to show that $\sigma' \in beh(maxappr(L_2))$, the projection funtions p'_1 and p'_2 witness $\rho' \in beh(L_1 \parallel_v maxappr(L_2))$ on the basis of the properties c-f.

Define ρ', p'_1 and p'_2 as follows:

- $|\rho'| = 2 \cdot |\rho|$,
- $A'_0 = A_0 \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi)))$,

- $p'_1(0) = p'_2(0) = 0$,
- for all $1 \leq i \leq |\rho|$, if $p_1(i) = p_1(i-1)+1$ and $p_2(i) = p_2(i-1)$ then

$$\begin{array}{lcl}
p'_1(2i-1) & = & p'_1(2i-2) \\
p'_2(2i-1) & = & p'_2(2i-2)+1 \\
p'_1(2i) & = & p'_1(2i-1)+1 \\
p'_2(2i) & = & p'_2(2i-1) \\
l'_{2i-1} & = & \tau \\
A'_{2i-1} & = & (A_{i-1} \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi)))) \cup \\
 & & (k^+_{p_1(i)} \cap (AP(L_2) \setminus AP(\phi)) \\
l'_{2i} & = & \mathit{eval}(AP(L_2), A'_{2i-1} \cap AP(L_2))(k_{p_1(i)}) \\
A'_{2i} & = & (A_i \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi)))) \cup (k^+_{p_1(i)} \cap (AP(L_2) \setminus AP(\phi))
\end{array}$$

and if $p_1(i) = p_1(i-1)$ and $p_2(i) = p_2(i-1)+1$ then

$$\begin{array}{lcl}
p'_1(2i-1) & = & p'_1(2i-2) \\
p'_2(2i-1) & = & p'_2(2i-2)+1 \\
p'_1(2i) & = & p'_1(2i-1) \\
p'_2(2i) & = & p'_2(2i-1)+1 \\
l'_{2i-1} & = & \tau \\
A'_{2i-1} & = & A_{i-1} \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi))) \\
l'_{2i} & = & \tau \\
A'_{2i} & = & A_i \cap (AP(L_1) \cup (AP(L_2) \cap AP(\phi)))
\end{array}$$

Define $\sigma' = C'_0 h'_1 C'_1 \ldots$ as follows:

- if $\max\{j \in \mathbb{N} \mid \exists i \in \mathbb{N} : p'_2(i) = j\}$ exists, then $|\sigma'|$ is this value, otherwise $|\sigma'| = \omega$,
- for all $0 \leq i \leq |\rho'|$, $C'_{p'_2(i)} = A'_i \cap AP(L_2)$, and
- for all $1 \leq i \leq |\sigma'|$, $h'_i = \tau$.

The σ' is well-defined, since if $p'_2(i) = p'_2(i-1)$ then $A'_i \cap AP(L_2) = A'_{i-1} \cap AP(L_2)$. As each $C'_i \subseteq AP(L_2)$ and each $h'_i = \tau$, $\sigma' \in \mathit{beh}(\mathit{maxappr}(L_2))$, as required.

Let us see then that ρ' is well-defined, in the sense that if $p_1(i) = p_1(i-1)+1$ then

$$l'_{2i} = \mathit{eval}(AP(L_2), A'_{2i-1} \cap AP(L_2))(k_{p_1(i)}) \neq \perp$$

Notice first that since $AP(\phi) \subseteq AP(L_1) \cup AP(L_2)$, $AP(\phi) \subseteq AP(L_1) \cup AP(L^\dagger)$ and $AP(L_1) \cap AP(L_2) = AP(L_1) \cap AP(L_2) = \emptyset$, we have

- $AP(\phi) \cap AP(L_2) \subseteq AP(L^\dagger)$

Furthermore, since $A_{i-1} \cap AP(L^\dagger) = C_{p_2(i-1)}$ and $\mathit{eval}(AP(L^\dagger), C_{p_2(i-1)})(k_{p_1(i)}) \neq \perp$, we have

- $k^-_{p_1(i)} \cap A_{i-1} \cap AP(L^\dagger) = \emptyset$ and
- $k^+_{p_1(i)} \cap (AP(L^\dagger) \setminus (AP(L^\dagger) \cap A_{i-1})) = \emptyset$, i.e. $k^+_{p_1(i)} \cap AP(L^\dagger) \subseteq A_{i-1}$.

To show $\mathit{eval}(AP(L_2), A'_{2i-1} \cap AP(L_2))(k_{p_1(i)}) \neq \perp$, we need to show

- $k^{-}_{p_1(i)} \cap A'_{2i-1} \cap AP(L_2) = \emptyset$, and
- $k^{+}_{p_1(i)} \cap (AP(L_2) \setminus (A'_{2i-1} \cap AP(L_2))) = \emptyset$, i.e. $k^{+}_{p_1(i)} \cap AP(L_2) \subseteq A'_{2i-1}$.

Notice first that by definition of a condition-labelled alts, $k^{-}_{p_1(i)} \cap AP(L_1) = \emptyset$, and by definition of a condition label, $k^{-}_{p_1(i)} \cap k^{+}_{p_1(i)} = \emptyset$. Secondly, as $k^{-}_{p_1(i)} \cap A_{i-1} \cap AP(L^\dagger) = \emptyset$ and $AP(\phi) \cap AP(L_2) \subseteq AP(L^\dagger)$, we have $k^{-}_{p_1(i)} \cap A_{i-1} \cap AP(L_2) \cap AP(\phi) = \emptyset$. Therefore,

$$
\begin{aligned}
k^{-}_{p_1(i)} \cap A'_{2i-1} \cap AP(L_2) \quad = \quad & k^{-}_{p_1(i)} \cap A_{i-1} \cap AP(L_1) \quad \cup \\
& k^{-}_{p_1(i)} \cap A_{i-1} \cap AP(L_2) \cap AP(\phi) \quad \cup \\
& k^{-}_{p_1(i)} \cap k^{+}_{p_1(i)} \cap (AP(L_2) \setminus AP(\phi)) \\
= \quad & \emptyset \cup \emptyset \cup \emptyset \quad = \quad \emptyset
\end{aligned}
$$

Notice then that as $AP(L_2) \cap AP(\phi) \subseteq AP(L^\dagger)$ and $k^{+}_{p_1(i)} \cap AP(L^\dagger) \subseteq A_{i-1}$, we have

$$
\begin{aligned}
k^{+}_{p_1(i)} \cap AP(L_2) \cap AP(\phi) \quad = \quad & k^{+}_{p_1(i)} \cap AP(L^\dagger) \cap AP(L_2) \cap AP(\phi) \\
\subseteq \quad & A_{i-1} \cap AP(L_2) \cap AP(\phi)
\end{aligned}
$$

This clearly implies

$$
\begin{aligned}
k^{+}_{p_2(i)} \cap AP(L_2) \quad = \quad & k^{+}_{p_2(i)} \cap AP(L_2) \cap AP(\phi) \quad \cup \quad k^{+}_{p_2(i)} \cap (AP(L_2) \setminus AP(\phi)) \\
\subseteq \quad & A_{i-1} \cap AP(L_2) \cap AP(\phi) \quad \cup \quad k^{+}_{p_2(i)} \cap (AP(L_2) \setminus AP(\phi)) \\
\subseteq \quad & A'_{2i-1}
\end{aligned}
$$

Consequently, $\mathit{eval}(AP(L_2), A'_{2i-1} \cap AP(L_2))(k_{p_1(i)}) \neq \bot$ and ρ' is well-defined.

It is easy to see from the definitions that ρ', p'_1, p'_2 and σ' satisfy conditions a-f above, implying $\rho' \in \mathit{beh}(L_1 \parallel_v \mathit{maxappr}(L_2))$ and $\rho' \models \phi$ iff $\rho \models \phi$. □

Theorem 5.2.7 gives us an algorithm deciding whether $L \models^{\Pi}_{m} \phi$ holds for given L, ϕ and Π. This can be decided simply by picking some L' such that $AP(\phi) \subseteq AP(L) \cup AP(L')$ and $\Pi \subseteq \Sigma(L')$ and checking whether $L\, |[\Pi]|\, \mathit{maxappr}(L') \models \phi$ holds, which in turn can be done by the normal model-checking algorithm for linear temporal logic. The smaller $\mathit{maxappr}(L')$ is, the less work this model-checking will require. As the size of $\mathit{maxappr}(L')$ depends on $AP(L')$ and $\Sigma(L')$, it is therefore best to choose L' so that $AP(L') = AP(\phi) \setminus AP(L)$ and $\Sigma(L') = \Pi$. Since $\mathit{maxappr}(L')$ in fact only depends on $AP(L')$ and $\Sigma(L')$, we introduce the following notation. The term *chaos* has been adopted from Hoare [Hoa85, p. 126]. Although technically very different, the current concept is based on the same intuition as Hoare's, i.e. *chaos* is in a sense maximally nondeterministic and unpredictable.

Definition 5.2.9 Let $B \subseteq AP$ and $\Pi \subseteq \Sigma$. The alts $chaos(B, \Pi)$ is defined as $chaos(B, \Pi) = (S, I, \Gamma, \Delta, A, V)$ where

$$\begin{array}{rcl} S & = & P(B) \\ I & = & S \\ \Gamma & = & \Pi \\ \Delta & = & S \times (\Pi \cup \{\tau\}) \times S \\ A & = & B \\ V & : & S \to P(A); s \mapsto s \end{array}$$

□

Corollary 5.2.10 Let L be an alts, $\Pi \subseteq \Sigma$ and ϕ an LTL'-formula.
Then $L \models_m^\Pi \phi$ iff $L \,|[\Pi]|\, chaos(AP(\phi) \setminus AP(L), \Pi) \models \phi$

Proof: Immediate from 5.2.7. □

An analogous argument holds for $\models_m^v$, except that this time there is no need to care about $\Sigma(L')$ at all.

Corollary 5.2.11 Let L be an alts and ϕ an LTL'-formula.
Then $L \models_m^v \phi$ iff $L \,\|_v\, chaos(AP(\phi) \setminus AP(L), \emptyset) \models \phi$

Proof: Immediate from 5.2.8. □

Proposition 5.2.12 Determining whether $L \models_m^\Pi \phi$ for given L, ϕ and Π can be done in deterministic $O(|L| \cdot |\Pi| \cdot 2^{O(|\phi|)})$ time.

Proof: The size of $chaos(AP(\phi) \setminus AP(L), \Pi)$ is $O(|\Pi| \cdot 2^{|AP(\phi) \setminus AP(L)|})$, hence $O(|\Pi| \cdot 2^{|\phi|})$. Therefore the size of $L \,|[\Pi]|\, chaos(AP(\phi) \setminus AP(L), \Pi)$ is $O(|L| \cdot |\Pi| \cdot 2^{|\phi|})$.

By 3.4.2 this means that we can decide whether $L \,|[\Pi]|\, chaos(AP(\phi) \setminus AP(L), \Pi) \models \phi$ in deterministic $O(|L| \cdot |\Pi| \cdot 2^{|\phi|} \cdot 2^{O(|\phi|)})$ time, i.e. in $O(|L| \cdot |\Pi| \cdot 2^{O(|\phi|)})$ time. By 5.2.10 this answers the question whether $L \models_m^\Pi \phi$ holds or not. □

Proposition 5.2.13 Determining whether $L \models_m^v \phi$ for given L and ϕ can be done in deterministic $O(|L| \cdot 2^{O(|\phi|)})$ time.

Proof: As the proof of 5.2.12, based on 3.4.2 and 5.2.11. □

The algorithm outlined by 5.2.13 and 5.2.11 is essentially the same as originally presented in [Kai92], although the technical presentation is different here. This algorithm is more efficient than the one proposed by Manna and Pnueli in [MP91], which takes $O(|L| \cdot 2^{O(|\phi|+|A_{ext}|)})$ time, where A_{ext} is the set of atomic propositions occurring in the condition expressions labelling the transitions of the alts L. Effectively, their algorithm considers L in the context

$$L \,\|_v\, chaos(A_{ext} \cup (AP(\phi) \setminus AP(L)), \emptyset)$$

as opposed to the context

$$L \parallel_v chaos(AP(\phi) \setminus AP(L), \emptyset)$$

used here. This allows handling the full linear-time temporal logic, with a nexttime-operator, which the current algorithm cannot do.

A drawback in using the approach based on modular validity in the current framework is that it can only be used to verify safety properties. As an arbitrary environment can at any moment diverge and prevent any other event from happening, there is no way of guaranteeing that a liveness property of the type 'eventually ϕ' would hold of all executions in all environments. A solution to this is adopting a richer process model that can incorporate fairness requirements. A verification algorithm for modular validity in this framework is described in [Kai92]. However, the approach will not be discussed further here, as models with fairness requirements fall beyond the scope of the current work. Notice also that another aspect reflecting this restriction to safety properties is the fact that in the extremal case of NDFD-preorder corresponding to modular validity in the sense of Theorems 5.2.7 and 5.2.8, NDFD-preorder coincides with trace preorder.

Chapter 6

Verification examples

Let us illustrate now applications of the theory by using the NDFD-equivalence and preorder to verify properties of the example systems introduced earlier.

6.1 Tools

To examine practical aspects of verification using compositional reductions, we took advantage of a toolset called ARA [VKCL93], built around an equivalence very closely related to $\overset{\text{ndfd}}{\approx}$, called the CFFD-equivalence. In the framework of the current work this equivalence can be formalised as follows.

Definition 6.1.1 Let L_1 and L_2 be altss. We say that L_1 *and* L_2 *are chaos-free failures divergences equivalent* or *CFFD-equivalent* and write $L_1 \overset{\text{cffd}}{\approx} L_2$ iff

- $\Sigma(L_1) = \Sigma(L_2)$,
- $AP(L_1) = AP(L_2)$,
- $\mathit{inftr}(L_1) = \mathit{inftr}(L_2)$,
- $\mathit{divtr}(L_1) = \mathit{divtr}(L_2)$, and
- $\mathit{sfail}(L_1) = \mathit{sfail}(L_2)$. □

The ARA tool set supports, among others, the following operations:

- reducing a labelled transition system with respect to $\overset{\text{cffd}}{\approx}$
- comparing two labelled transition systems with respect to the $\overset{\text{cffd}}{\approx}$
- presenting small labelled transition systems in graphical form

To enable ARA to work with the NDFD-equivalence, instead of CFFD, i.e. to use non-divergent failures instead of stable failures, we introduced an intermediate processing step to the system. Intuitively this step removes from a system those stable failures (σ, Π) that are 'masked' by a corresponding divergence trace σ.

The ARA toolset was designed to operate on standard labelled transition systems, not augmented ones. This means that the tool could be used directly only for altss L for which

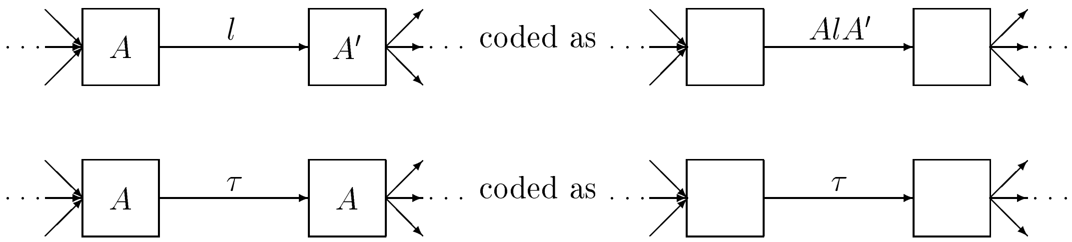


Figure 6.1: Coding state labels as transition labels

$AP(L) = \emptyset$, i.e. for altss without state labels. To overcome this restriction, we devised a way of coding altss as ltss. This coding is based on presenting the state labels of an alts as parts of the transition labels of the corresponding lts, as depicted in the Figure 6.1. However, we shall not discuss the details of the coding here.

The ARA toolset does not directly support the use of preorders in the verification, either. However, by the interreducibility of $\overset{\text{ndfd}}{\approx}$ and $\overset{\text{ndfd}}{\prec}$ (Proposition 5.1.10), we could check whether

$$L \overset{\text{ndfd}}{\prec} L'$$

holds by checking whether

$$L\,[:]\,L' \overset{\text{ndfd}}{\approx} L'$$

holds.

Summarising, these modifications and codings meant that we could use the ARA toolset

- to reduce a given alts L with respect to $\overset{\text{ndfd}}{\approx}$
- to check whether for two given altss L and L', $L \overset{\text{ndfd}}{\approx} L'$ holds, and
- to check whether for two given altss L and L', $L \overset{\text{ndfd}}{\prec} L'$ holds.

In verifying the example systems, no automated temporal logic model checker was used, since no suitable implementation of the model checking algorithm was available. However, with the one exception of the liveness properties of the SW-protocol, this was not felt to be a great problem, since the final reduced altss were small enough to make it easy to check manually the validity of the required properties. The method used for verifying the liveness properties of the SW-protocol is discussed fully later.

6.2 Mutual exclusion

6.2.1 Using NDFD-equivalence

Our aim is to verify the properties of the mutual exclusion system described in Subsection 2.5.1. The properties were formulated in LTL' in Example 3.3.1. In the following we verify the 'divided' forms of the formulae expressing the required properties for a particular fixed C_i or a particular fixed pair C_i, C_j. Let L denote the alts modelling the complete system

$$L = Arb_n \parallel_v C_1 \parallel_v C_2 \parallel_v \dots \parallel_v C_n$$

Safety Let us look first at the safety property establishing mutual exclusion between C_i and C_j:

$$\Box\neg(c_i \wedge c_j)$$

As was stated in Example 3.4.3, the size of the alts L modelling the complete system grows infeasibly large for any but the very small values of n. Here we construct a smaller alts NDFD-equivalent to L.

When verifying the formula, the only atomic propositions that need to be kept track of in the level of the complete system are c_i and c_j. Consequently, by 4.6.42, all the other atomic propositions may be encapsulated away. However, as they are used for interaction between processes, they cannot be removed right away but only after they are no longer needed for interaction. This means that verifying that $\Box\neg(c_i \wedge c_j)$ holds of L can be done by verifying that it holds of

$$L' \equiv_{df} \textbf{encaps } X \textbf{ in } (Arb_n \parallel_v C_1 \parallel_v \dots \parallel_v C_i \parallel_v \dots \parallel_v C_j \parallel_v \dots \parallel_v C_n)$$

where $X = \{g_1, \dots, g_n, r_1, \dots, r_n, c_1, \dots, c_{i-1}, c_{i+1}, \dots, c_{j-1}, c_{j+1}, \dots, c_n\}$

The construction of an L'' such that $L'' \overset{\text{ndfd}}{\approx} L'$, is done in steps by defining

- $A_n \equiv_{df} Arb_n$
- $A_{n-1} \equiv_{df} \textbf{encaps } g_n, r_n, c_n \textbf{ in } A_n \parallel_v C_n$

  $\vdots$
- $A_j \equiv_{df} \textbf{encaps } g_{j+1}, r_{j+1}, c_{j+1} \textbf{ in } A_{j+1} \parallel_v C_{j+1}$
- $A_{j-1} \equiv_{df} A_j$
- $A_{j-2} \equiv_{df} \textbf{encaps } g_{j-1}, r_{j-1}, c_{j-1} \textbf{ in } A_{j-1} \parallel_v C_{j-1}$

  $\vdots$
- $A_i \equiv_{df} \textbf{encaps } g_{i+1}, r_{i+1}, c_{i+1} \textbf{ in } A_{i+1} \parallel_v C_{i+1}$
- $A_{i-1} \equiv_{df} A_i$
- $A_{i-2} \equiv_{df} \textbf{encaps } g_{i-1}, r_{i-1}, c_{i-1} \textbf{ in } A_{i-1} \parallel_v C_{i-1}$

  $\vdots$
- $A_0 \equiv_{df} \textbf{encaps } g_1, r_1, c_1 \textbf{ in } A_1 \parallel_v C_1$

Defining

$$L'' \equiv_{df} \textbf{encaps } X \textbf{ in } (A_0 \parallel_v C_i \parallel_v C_j)$$

we can see from Proposition 4.6.43 that $L' \overset{\text{ndfd}}{\approx} L''$.

Now we may construct an A_0' such that $A_0' \overset{\text{ndfd}}{\approx} A_0$ in the following way. Construct A_{n-1} on the basis of Arb_n and C_n, and apply the NDFD-preserving reduction algorithm to produce an A'_{n-1} such that $A'_{n-1} \overset{\text{ndfd}}{\approx} A_{n-1}$. Substitute A'_{n-1} for A_{n-1} when constructing

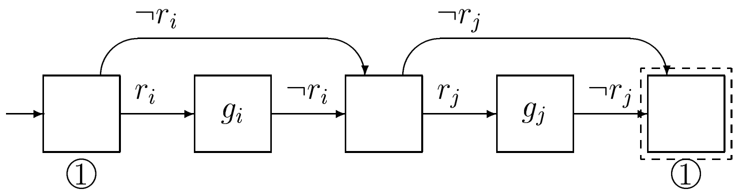


Figure 6.2: Mutual exclusion: Reduced alts A'_0

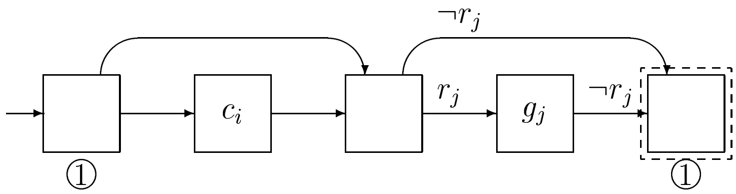


Figure 6.3: Mutual exclusion: Reduced alts B'

A_{n-2} and reduce the resulting alts again giving A'_{n-2}. This process can be repeated until A'_0 is reached.

An interesting observation is that the A'_{n-1} considered above is, in fact, isomorphic to Arb_{n-1}. In other words, when we hide away from the system one customer and the interaction between that customer and the arbiter for n customers, we get a process that is equivalent to an arbiter dealing with $n-1$ customers.

Notice that Arb_n has $2n$ states and C_i has 4 states. As A_n is isomorphic to Arb_n, it has also $2n$ states, and consequently A_{n-1} has $8n$ states. As A'_{n-1} is isomorphic to Arb_{n-1}, it has $2(n-1)$ states. Repeating the above argument it can be seen that when A'_0 is constructed, the size of the intermediate graphs never grows beyond $8n$ nodes. This means that the process can be easily carried through for very large values of n, too.

The graph A'_0 is shown in Figure 6.2. Given the discussion above it should come as no surprise that A'_0 is similar to Arb_2 (Figure 2.3) with the exception that all 1:s are replaced by i:s and all 2:s by j:s.

On the basis of A'_0 and C_i we can construct the graph corresponding to

$$B \equiv_{df} \textbf{encaps } g_i, r_i \textbf{ in } (A'_0 \parallel_v C_i)$$

The structure of the graph is similar to that presented in Figure 2.5. Now the reduction algorithm may be run on the alts B. Denote the result by B'. This is depicted in Figure 6.3.

In the similar way we can construct

$$C \equiv_{df} \textbf{encaps } r_j, g_j \textbf{ in } (B' \parallel_v C_j)$$

and reduce C, yielding C'. This graph is depicted in Figure 6.4. From C' it is easy to see that $C' \models \Box\neg(c_i \wedge c_j)$. What is more, by the construction of C', $C' \overset{\text{ndfd}}{\approx} L$, which implies $L \models \Box\neg(c_i \wedge c_j)$.

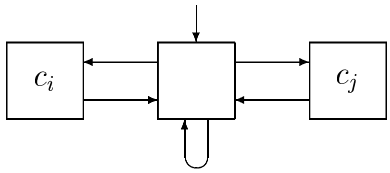


Figure 6.4: Mutual exclusion: Reduced alts C'

Figure 6.5: Mutual exclusion: Reduced altss D' (left) and E' (right)

Liveness Next we are interested in verifying the liveness property $\Box(r_i \Rightarrow \Diamond c_i)$. The verification proceeds as before up to the construction of A'_0. The next step is to construct

$$D \equiv_{df} \textbf{encaps } r_j, g_j, c_j \textbf{ in } (A'_0 \parallel_v C_j)$$

and reduce D, yielding the D' depicted in Figure 6.5. After this we construct

$$E \equiv_{df} \textbf{encaps } g_i \textbf{ in } (D' \parallel_v C_i)$$

and reduce E yielding E' (also in Figure 6.5). From this it is easy to see that $E' \models \Box(r_i \Rightarrow \Diamond c_i)$, and consequently that $L \models \Box(r_i \Rightarrow \Diamond c_i)$.

One-bounded overtaking Verification of the one-bounded overtaking property

$$\Box(r_i \Rightarrow (\neg g_j)\,\mathcal{U}_w(g_j\,\mathcal{U}_w((\neg g_j)\,\mathcal{U}_w g_i)))$$

takes place in a similar way. This time we construct first

$$F \equiv_{df} \textbf{encaps } r_j, c_j \textbf{ in } (A'_0 \parallel_v C_j)$$

reduce F yielding F', construct then

$$G \equiv_{df} \textbf{encaps } c_i \textbf{ in } (F' \parallel_v C_i)$$

and reduce G yielding G'. From this graph (Figure 6.6) it can be seen that

$$G' \models \Box(r_i \Rightarrow (\neg g_j)\,\mathcal{U}_w(g_j\,\mathcal{U}_w((\neg g_j)\,\mathcal{U}_w g_i)))$$

and, consequently, that

$$L \models \Box(r_i \Rightarrow (\neg g_j)\,\mathcal{U}_w(g_j\,\mathcal{U}_w((\neg g_j)\,\mathcal{U}_w g_i)))$$

The table in Figure 6.7 lists the number of reachable states in the largest alts used in the verification, first using the straightforward approach, and then using NDFD-preserving reductions. The advantages of the method are clear in this case.

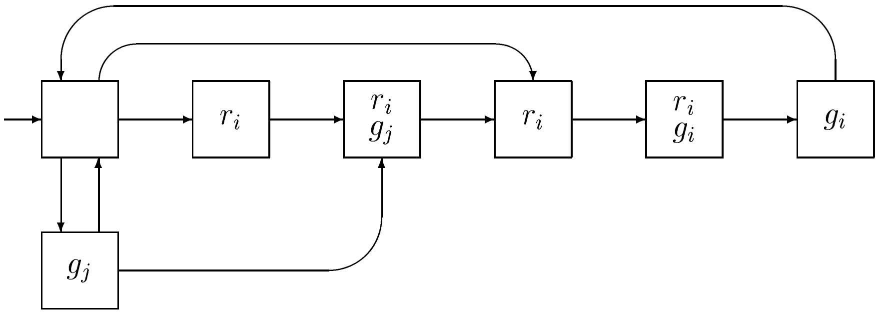


Figure 6.6: Mutual exclusion: Reduced alts G'

n	brute force	using NDFD
2	36	16
5	2430	40
10	1180980	80
100	$> 2^{166}$	800
500	$> 2^{800}$	4000

Figure 6.7: Mutual exclusion: Number of states in largest alts

6.2.2 Using modular validity

To illustrate the technique of modular validity discussed in Section 5.2, let us present an alternative way to verify the safety property expressing mutual exclusion in the system introduced in Section 2.5.1 and analysed above in Subsection 6.2.1.

Let L denote the alts modelling the complete system

$$L = Arb_n \parallel_v C_1 \parallel_v C_2 \parallel_v \ldots \parallel_v C_n$$

Verifying that $L \models \Box\neg(c_i \wedge c_j)$ can be done by verifying that

$$\begin{array}{rcl} C_i & \models^v_m & \phi_i \\ C_j & \models^v_m & \phi_j \\ Arb_n & \models^v_m & \phi_A \\ & \models & \phi_i \wedge \phi_j \wedge \phi_A \Rightarrow \Box\neg(c_i \wedge c_j) \end{array}$$

where

$$\begin{array}{lcl} \phi_i & \equiv_{df} & \neg c_i \ \wedge \ \Box(\neg c_i \Rightarrow (\neg c_i)\mathcal{U}_w(c_i \wedge g_i)) \ \wedge \ \Box(c_i \Rightarrow r_i) \\ \phi_j & \equiv_{df} & \neg c_j \ \wedge \ \Box(\neg c_j \Rightarrow (\neg c_j)\mathcal{U}_w(c_j \wedge g_j)) \ \wedge \ \Box(c_j \Rightarrow r_j) \\ \phi_A & \equiv_{df} & \Box\neg(g_i \wedge g_j) \ \wedge \ \Box(g_i \Rightarrow g_i\mathcal{U}_w\neg r_i) \ \wedge \ \Box(g_j \Rightarrow g_j\mathcal{U}_w\neg r_j) \end{array}$$

Intuitively, the lemma ϕ_i states that the customer C_i is not initially in the critical section, will never move to the critical section without a permission being granted, and always requests access to the resource while in the critical section. The formula ϕ_j states the same property for customer C_j. Finally, formula ϕ_A states that the arbiter will never grant access to the critical section for both C_i and C_j at the same time, and that if the arbiter has granted access to one customer, it will continue granting access to the same customer, until the customer no longer requests access.

Verifying the statements concerning $\models^v_m$ can be done as outlined in 5.2.11. For example, to verify $C_i \models^v_m \phi_i$ we can verify $C_i \parallel_v chaos(AP(\phi_i) \setminus AP(C_i), \emptyset) \models \phi$, i.e. that

$$C_i \parallel_v chaos(\{g_i\}, \emptyset) \models \phi$$

The altss C_i, $chaos(\{g_i\}, \emptyset)$ and $C_i \parallel_v chaos(\{g_i\}, \emptyset)$ are shown in Figure 6.8. It is easy to see from the picture that ϕ_i is valid over the alts. Verifying $C_j \models^v_m \phi_j$ and $Arb_n \models^v_m \phi_A$ can be done similarily.

Finally, verifying $\models \phi_i \wedge \phi_j \wedge \phi_A \Rightarrow \Box\neg(c_i \wedge c_j)$ could by done by an automated satisfiability checker. However, as no such tool was available, the validity of the formula was verified manually.

6.3 Sliding window protocol

Let us now verify the required properties of the SW-protocol example, introduced in Subsection 2.5.2. The required safety and liveness properties were formulated in Example 3.3.2.

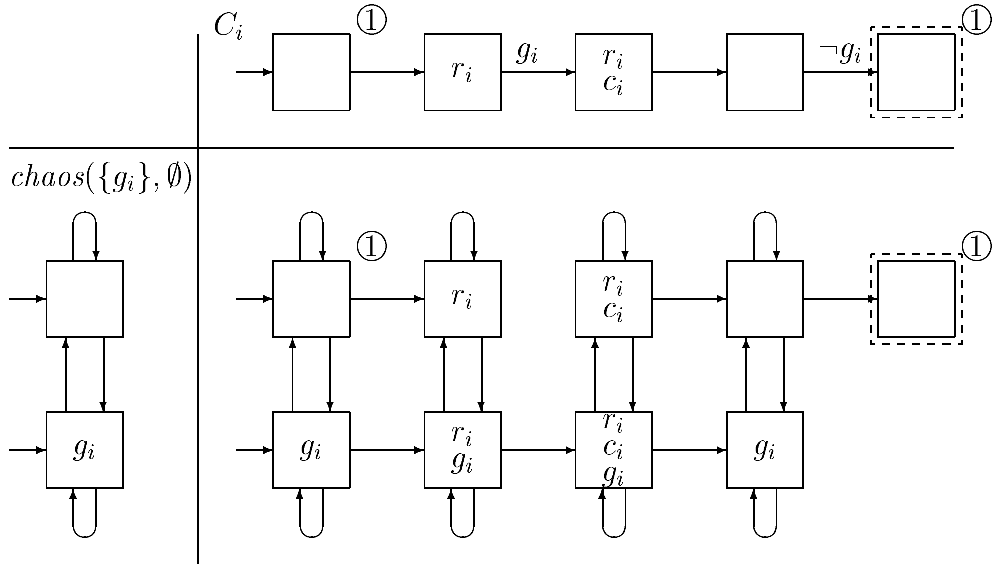


Figure 6.8: Mutual exclusion: Alts $C_i \parallel_v chaos(\{g_i\}, \emptyset)$

For verifying both safety and liveness properties of the SW-protocol, we use the general approach of constructing an abstraction $\overline{S}$ on the basis of the sender alts S. The particular abstractions used for safety and liveness properties are different, the abstraction for safety properties preserving less information about the behaviour of the system than the abstraction for liveness properties. However, both of the abstractions have some similar characteristics. Due to reasons of representation, we shall look at the verification of the liveness property first, and at that of the safety properties later, although in reality safety properties were verified first and liveness properties afterwards.

6.3.1 Liveness

6.3.1.1 Preliminary transformations

As discussed in Example 3.3.2, to verify that the protocol satisfies its liveness requirements, we have to verify that

$$D_{live} \parallel_s P \models \phi_{live}$$

The most straightforward way to proceed would be to work directly on the basis of the definitions, and construct first the alts P and then compose this with D_{live}. However, the fixed data source D_{live} produces only a fraction of all the potential data sequences, which means, in turn, that a large part of the state space of P will not be reachable when P is composed with D_{live}. If we now construct the whole of P before composing it with D_{live}, we are, in fact, doing a lot of unnecessary work. Because of this, it is advantageous to use the strategy of 'pushing the data source inward', i.e. composing the sender alts S with the data source D_{live} *before* building SC and all the other altss up to P. In more formal terms this can be done by defining:

$$
\begin{aligned}
DS &\equiv_{df} D_{live} \parallel_s S \\
DSC_0 &\equiv_{df} DS \\
DSC_{i+1} &\equiv_{df} \textbf{hide } sc* \textbf{ in } (DSC_i[sc*/cr*] \parallel_s C) \\
DSC &\equiv_{df} DSC_{n_c-1} \\
DSCA_0 &\equiv_{df} DSC \\
DSCA_{i+1} &\equiv_{df} \textbf{hide } as* \textbf{ in } (DSCA_i[as*/ra*] \parallel_s A) \\
DSCA &\equiv_{df} DSCA_{n_a-1} \\
DSC_FA &\equiv_{df} DSCA[sc*/cr*] \parallel_s C_F \\
DSC_FA_F &\equiv_{df} DSC_FA[as*/ra*] \parallel_s A_F \\
DP &\equiv_{df} DSC_FA_F \parallel_s R
\end{aligned}
$$

From this definition of DP and the definition of P (see page 35), it is fairly easy to see on the basis of the limited associativity of $\parallel_s$ (Proposition 2.4.12) that DP is isomorphic to $D_{live} \parallel_s P$. This implies that $D_{live} \parallel_s P \overset{\text{ndfd}}{\approx} DP$, and $D_{live} \parallel_s P \models \phi_{live}$ iff $DP \models \phi_{live}$.

As another preliminary step, let us ease the verification task by using the techniques discussed in Section 4.6.4 to ignore information that is recorded in the model of the system, but that is not actually needed for verifying $DP \models \phi_{live}$. Since $AP(\phi_{live}) = \emptyset$ and $\Sigma(\phi_{live}) \cap \{ds*, tim\} = \emptyset$, by Proposition 4.6.42 we know that

$$DP \models \phi_{live}$$

iff

$$\textbf{hide } ds*, tim \textbf{ in } (\textbf{encaps } AP \textbf{ in } DP) \models \phi_{live}$$

It is advantageous to remove the unnecessary information from the model in the level of the component altss as soon as it is not used for interaction between components. To this purpose, let us define

$$
\begin{aligned}
S' &\equiv_{df} \textbf{encaps } AP \textbf{ in } S \\
DS' &\equiv_{df} \textbf{hide } ds* \textbf{ in } (D_{live} \parallel_s S') \\
DSC'_0 &\equiv_{df} DS' \\
DSC'_{i+1} &\equiv_{df} \textbf{hide } sc* \textbf{ in } (DSC'_i[sc*/cr*] \parallel_s C) \\
DSC' &\equiv_{df} DSC'_{n_c-1} \\
DSCA'_0 &\equiv_{df} DSC' \\
DSCA'_{i+1} &\equiv_{df} \textbf{hide } as* \textbf{ in } (DSCA'_i[as*/ra*] \parallel_s A) \\
DSCA' &\equiv_{df} DSCA'_{n_a-1} \\
DSC_FA' &\equiv_{df} DSCA'[sc*/cr*] \parallel_s C_F \\
DSC_FA'_F &\equiv_{df} DSC_FA'[as*/ra*] \parallel_s A_F \\
R' &\equiv_{df} \textbf{encaps } AP \textbf{ in } R \\
DP' &\equiv_{df} \textbf{hide } tim \textbf{ in } (DSC_FA'_F \parallel_s R')
\end{aligned}
$$

On the basis of the definitions of DP' and DP, propositions 4.6.43 and 4.6.44 imply that

$$\textbf{hide } ds*, tim \textbf{ in } (\textbf{encaps } AP \textbf{ in } DP) \overset{\text{ndfd}}{\approx} DP'$$

On the basis of the above, this means that it suffices to verify that

$$DP' \models \phi_{live}$$

6.3.1.2 Abstraction for liveness properties

Before describing the actual abstraction $\overline{S}$ for liveness properties, let us discuss first some characteristics of it that are used in the verification task. Given the abstraction $\overline{S}$, let us define

$$\begin{array}{rcl}
\overline{S}' & \equiv_{df} & \textbf{encaps } AP \textbf{ in } \overline{S} \\
\overline{DS}' & \equiv_{df} & \textbf{hide } ds* \textbf{ in } (D_{live} \parallel_s \overline{S}') \\
\overline{DSC}'_F & \equiv_{df} & \overline{DS}'[sc*/cr*] \parallel_s C_F \\
\overline{DSC_F A}'_F & \equiv_{df} & \overline{DSC}'_F[as*/ra*] \parallel_s A_F \\
\overline{DP}' & \equiv_{df} & \textbf{hide } tim \textbf{ in } (\overline{DSC_F A}'_F \parallel_s R')
\end{array}$$

The crucial characteristics of the abstraction are

A: $DS' \overset{\text{ndfd}}{\prec} \overline{DS}'$

B: $\textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C) \overset{\text{ndfd}}{\prec} \overline{DS}'$

C: $\textbf{hide } as* \textbf{ in } (\overline{DS}'[as*/ra*] \parallel_s A) \overset{\text{ndfd}}{\prec} \overline{DS}'$

D: $\overline{DP}' \models \phi_{live}$

Let us see why these properties of $\overline{S}$ are so useful. Notice first that since $DS' \overset{\text{ndfd}}{\prec} \overline{DS}'$ by Claim A, and $DSC'_0 = DS'$ by definition, we have

$$DSC'_0 \overset{\text{ndfd}}{\prec} \overline{DS}'$$

Suppose then that $DSC'_i \overset{\text{ndfd}}{\prec} \overline{DS}'$ holds. By definition

$$DSC'_{i+1} = \textbf{hide } sc* \textbf{ in } (DSC'_i[sc*/cr*] \parallel_s C)$$

By compositionality of $\overset{\text{ndfd}}{\prec}$ and $DSC'_i \overset{\text{ndfd}}{\prec} \overline{DS}'$, this means that

$$DSC'_{i+1} \overset{\text{ndfd}}{\prec} \textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C)$$

But by Claim B,

$$\textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C) \overset{\text{ndfd}}{\prec} \overline{DS}'$$

which imply

$$DSC'_{i+1} \overset{\text{ndfd}}{\prec} \overline{DS}'$$

Since we saw above that $DSC'_0 \overset{\text{ndfd}}{\prec} \overline{DS}'$, this means that

$$DSC'_i \quad \overset{\text{ndfd}}{\prec} \quad \overline{DS}'$$

for all $i \in \mathbb{N}$, i.e. that for any choice of n_c

$$DSC' \quad \overset{\text{ndfd}}{\prec} \quad \overline{DS}'$$

A fact that needs to be stressed is that this holds no matter what the value of n_c is, i.e. for arbitrary message channel lengths.

In the same way we can see on the basis of Claim C that

$$DSCA' \quad \overset{\text{ndfd}}{\prec} \quad \overline{DS}'$$

no matter what the value of n_a is, i.e. for arbitrary acknowledgement channel lengths.

By the compositionality of $\overset{\text{ndfd}}{\prec}$ and the definitions of DSC_FA' and $\overline{DSC}'_F$, this implies that

$$DSC_FA' \quad \overset{\text{ndfd}}{\prec} \quad \overline{DSC}'_F$$

In a similar way we can see that

$$DSC_FA'_F \quad \overset{\text{ndfd}}{\prec} \quad \overline{DSC_FA}'_F$$

By the compositionality of $\overset{\text{ndfd}}{\prec}$ and the definitions of DP' and $\overline{DP}'$, this implies that

$$DP' \quad \overset{\text{ndfd}}{\prec} \quad \overline{DP}'$$

Since $\overset{\text{ndfd}}{\prec}$ implies $\overset{ltl'}{\prec}$, we know then that

$$DP' \quad \overset{ltl'}{\prec} \quad \overline{DP}'$$

But as by Claim D

$$\overline{DP}' \quad \models \quad \phi_{live}$$

this implies

$$DP' \quad \models \quad \phi_{live}$$

which is precisely what we are trying to establish in the first place.

Summarising, if we manage to check that Claims A-D above hold, we know that the protocol fulfils the liveness requirement, no matter what the channel capacity n_c and the acknowledgement channel capacity n_a are. We believe that this is quite interesting, since in case studies it has often been pointed out that introducing even short channels between components of a system tends to increase the size of the state space so that straightforward model-checking becomes very hard [RRSV87, Ern91]. However, here this problem does not emerge at all.

SENDER ABSTRACTION

Notation

  $x \oplus y$, $x \ominus y$ as before

  $rnd(x, \ldots, y)$ returns a random element from $\{x, \ldots, y\}$

Variables

  $d[(-tw) \ldots (tw-1)]$: table of data items

  i, j, k : int /* $i \in \{-tw, \ldots, tw\}$, $j \in \{-tw, \ldots, tw\}$, $k \in \{0, \ldots, w\}$ */

  ch : bool /* Tells whether the sender is in a chaotic state. */

  /* x, a: temporary variables, as in the sender */

Initially

  $i = 0, j = 0, k = 0, ch = \bot$

loop infinitely
  choose nondeterministically
    ch $\longrightarrow$ receive(*, *);
  []
    ch $\longrightarrow$ send(*, *);
  []
    ch $\longrightarrow$ idle;
  []
    $\neg ch \wedge i < tw$ $\longrightarrow$ receive(data source, $d[i]$);
      $i := i + 1$;
  []
    $\neg ch \wedge j < i$ $\longrightarrow$ send(channel, $(j \oplus k, d[j])$);
      $j := j + 1$;
      if $j < 0$ then $j := rnd(j, \ldots, 0)$;
  []
    $\neg ch \wedge j = i \wedge i > 0$ $\longrightarrow$ receive(timeout);
      $j := 0$;
  []
    $\neg ch$ $\longrightarrow$ receive(ack channel, x);
      $a := (x \ominus k) + 1$;
      if $((a - \min(j, 0)) \leq tw)$ then
        $d[(-tw) \ldots (tw - 1 - a)] := d[(-tw + a) \ldots (tw - 1)]$;
        $i := \max(-tw, i - a)$;
        $j := j - a$;
        if $j < 0$ then $j := rnd(j, \ldots, 0)$
        $k := x \oplus 1$ /* i.e. $= k \oplus a$ */
      else if $a \neq w + 1$ then /* if not a copy of the last ack */
        $ch := \top$ /* then go to a chaotic state */
      end if;
  end choose;
end loop;

Figure 6.9: SW-protocol: Abstraction for liveness properties

Let us now describe the abstraction $\overline{S}$ for liveness properties. As shall be seen later, the abstraction for safety properties can be obtained as a simplification of this. The structure of the abstraction is described in pseudo-code in Figure 6.9.

The abstraction was crafted by considering intuitively which characteristics of the protocol were actually relevant for the verification of the required liveness properties. The final formulation was arrived at by the process of trial and error, by considering various supposed abstractions based on similar intuitions, and by trying to understand why these either did or did not fulfil the required properties.

The abstraction is based on the sender process of the protocol, so probably the easiest way of understanding it is contrasting the abstraction with the sender itself (Figure 2.7). The two basic intuitions behind the abstraction are the following.

- The abstraction mimics the behaviour of the sender with a message channel and an acknowledgement channel of unspecified length. What this means is that the abstraction works basically as the sender, but the sent messages do not necessarily appear immediately, but can stay in the channel for some while before emerging from it and becoming visible to the outside world. Similarly, acknowledgement messages do not necessarily cause effect immediately, but can stay in the acknowledgement channel for some while before the sender reacts to them.
- The behaviour of the sender process is well-defined for all possible acknowledgement messages at any moment in the execution of the system. However, in the context of the complete system it is often the case that in a given situation only a subset of all possible acknowledgement messages can actually occur. In the abstraction we do not record information about reactions to acknowledgement messages that are not going to occur in a given situation. Instead, it is specified that after the reception of an unexpected acknowledgement message, the abstraction becomes 'chaotic', capable of performing any possible action and diverging.

The latter of these intuitions is reflected in the abstraction by a new truth-valued variable ch. In normal operation, ch is false. If the sender abstraction receives an acknowledgement message that should not occur in the current situation at the context of the complete system, the variable becomes true. After this has happened, the abstraction can perform any possible action, i.e. it can send or receive any message or idle. In the context of the complete system, the variable ch should, of course, never become true. However, in the sender abstraction it allows us to forget about the potential courses of action after such events.

The former of these intuitions is reflected by several aspects of the abstraction:

- The data table d has been extended to negative indices from $-tw$ to -1. Intuitively, the data items in these elements of the table correspond to messages which
    - the sender may have sent earlier but which have not emerged from the channel yet, and
    - to which an acknowledgement may have been or is going to be received by it, i.e. to which an acknowledgement has been delivered to the acknowledgement channel.

As in the sender, i points to the first unused element of d and j to the element of d to be sent next. However, their range has been extended to reflect the new elements of d.

- When a message containing the data item $d[j]$ has been sent, j is increased by 1, as in the sender. However, if the resulting j points to an element in $d[(-tw)\ldots(-1)]$, i.e. corresponds to a message that may have been sent earlier and that has been or is going to be acknowledged, j may be arbitrarily updated to any value between j and 0. This skipping over of messages corresponds to the intuition that the sender *may have* sent these messages; it is possible that the acknowledgement reached the sender before the corresponding message was sent, and so the message was never actually sent.
- When an acknowledgement is delivered to the acknowledgement channel, it is deemed to be a valid one, if it acknowledges a message that is
    - at most tw sequence numbers ahead of the first message in d that may emerge from the channel (i.e. the data item in $d[j]$), and
    - at the current transmission window, i.e. at most tw sequence numbers ahead of the message in $d[0]$

  In this case the data table d is shifted to the left so that the first unacknowledged data item is in $d[0]$, and i, j and k are adjusted accordingly. As above, if j points to an element in $d[(-tw)\ldots(-1)]$ it can be arbitrarily updated to any value between j and 0.
- If an acknowledgement is not valid according to the rules above, it is ignored if it is a copy of the previous succesful acknowledgement. Otherwise, it is treated as a completely unexpected acknowledgement, and the approximation moves to a chaotic state.

6.3.1.3 Verifying liveness properties

To verify the liveness properties of the SW-protocol for given parameters w, tw and rw (and for all n_c and n_a), we constructed the approximation $\overline{S}$ as explained above and verified Claims A-D, i.e. that

$$\begin{array}{rcl} DS' & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\ \textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\ \textbf{hide } as* \textbf{ in } (\overline{DS}'[as*/ra*] \parallel_s A) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\ \overline{DP}' & \models & \phi_{live} \end{array}$$

where $\overline{DS}'$ and $\overline{DP}'$ are as before. To verify Claims A-C, we used the modified ARA toolset, discussed in Section 6.1.

To make the task easier, we first run the reduction algorithm on $\overline{DS}'$, producing an alts $\overline{DS}''$ such that

$$\overline{DS}' \overset{\text{ndfd}}{\approx} \overline{DS}''$$

Parameters			Reachable nodes					
w	tw	rw	$\overline{DS}'$	$\overline{DS}''$	DS'	$\overline{DS}'' \parallel_s C$	$\overline{DS}'' \parallel_s A$	$\overline{DP}'$
1	1	1	22	29	12	56	95	61
2	1	1	32	43	18	82	181	85
	2	1	74	106	36	211	441	607
3	2	2	98	141	48	280	723	915
	3	1	178	269	80	560	1373	2375
4	2	2	122	176	60	349	1075	1131
	3	2	222	336	100	699	2045	3814
	4	1	352	561	150	1214	3047	6613
5	2	2	146	211	72	418	1497	1347
	3	3	266	403	120	838	2851	5179
	4	2	422	673	180	1456	4753	11209
	5	1	614	1033	252	2308	7287	15223
6	2	2	170	246	84	487	1989	1563
	3	3	310	470	140	977	3791	6018
	4	3	492	785	210	1698	6323	16637
7	2	2	194	281	96	556	2551	1779
	3	3	354	537	160	1116	4865	6857
	4	4	562	897	240	1940	8117	21379

Figure 6.10: SW-protocol: Sizes of abstraction altss for liveness properties

and then used this alts $\overline{DS}''$ in the place of $\overline{DS}'$ in verifying the claims. The table in Figure 6.10 lists sizes of some of the altss involved for some selected parameter values w, tw and rw. A point that may seem counterintuitive is why $\overline{DS}''$ was used; in all the examined cases it is actually slightly larger in size than $\overline{DS}'$. However, we found out in practice that the later steps in the verification, like verifying $DS' \overset{\text{ndfd}}{\prec} \overline{DS}'$, could be performed more easily and requiring less resources by using $\overline{DS}''$, and therefore used it instead of $\overline{DS}'$. This is an example of the fact that the primary aim of the whole approach is making verification easier, and that reducing the sizes of altss is of interest as a means, not as an end to itself. As the particular structure of $\overline{DS}''$, a certain kind of determinism in its transition relation, made the verification easier using $\overline{DS}''$, there was no reason to slavishly use $\overline{DS}'$ just because it is smaller in size.

What we still need to check is that

$$\overline{DP}' \models \phi_{live}$$

i.e. that

$$\overline{DP}' \models \phi_{fair} \Rightarrow (\Box\Diamond \mathit{rt0} \vee \Diamond \mathit{rt1})$$

i.e. that

$$\overline{DP}' \models \bigwedge_{0\leq i\leq w} (\Box\Diamond(\bigvee_{0\leq j\leq 2} \textcircled{scij}) \Rightarrow \Box\Diamond(\bigvee_{0\leq j\leq 2} \textcircled{crij})) \wedge \bigwedge_{0\leq i\leq w} (\Box\Diamond \textcircled{rai} \Rightarrow \Box\Diamond \textcircled{asi})$$

$$\Rightarrow$$

$$(\Box\Diamond \textcircled{rt0} \vee \Diamond \textcircled{rt1})$$

In all the other examples of this work, the actual verification of the required temporal logic properties could be done just by informally examining the final reduced altss, thanks to the conciseness of both the graphs and the formulae. However, here we have altss of several thousand states and a formula that looks quite intimidating. Therefore, some form of automatised model-checking was necessary. As no linear temporal logic model-checker was readily available, an ad hoc solution was devised, allowing us to use the available tools to model-check the liveness property. The results of this verification confirmed that the protocol fulfils its liveness requirements. As the details of the solution used for model-checking are not relevant to the main work discussed here, they are postponed to Appendix B.

6.3.2 Safety

6.3.2.1 Preliminary transformations

The verification of the safety properties follows the same pattern as that of the liveness properties. We have to verify that

$$D_{safe} \parallel_s P \models \phi_{safe}$$

Using the strategy of composing the data source with the sender before constructing the whole system, this is the same as verifying that

$$DP \models \phi_{safe}$$

where DP is defined as in page 141 except for

$$DS \equiv_{df} D_{safe} \parallel_s S$$

Again, let us first use the techniques of Section 4.6.4 to ignore information not needed for verifying $DP \models \phi_{safe}$. Since $AP(\phi_{safe}) = \emptyset$ and

$$\Sigma(\phi_{safe}) \cap \{sc*, cr*, ra*, as*, ds*, tim\} = \emptyset$$

by Proposition 4.6.42 we know that

$$DP \models \phi_{safe}$$

iff

$$\textbf{hide } sc*, cr*, ra*, as*, ds*, tim \textbf{ in } (\textbf{encaps } AP \textbf{ in } DP) \models \phi_{safe}$$

Notice that here we are able to ignore more transition labels than in the case of the liveness properties. Intuitively, the labels $sc*$, $cr*$, $ra*$ and $as*$ are only required for distinguishing fair executions from unfair ones. As the safety properties are not dependent on fairness assumptions, we can hide these labels.

To remove unnecessary information as soon as it is not used for interaction, let us define this time

$$\begin{array}{rcl}
S' & \equiv_{df} & \textbf{encaps } AP \textbf{ in } S \\
DS' & \equiv_{df} & \textbf{hide } ds* \textbf{ in } (D_{safe} \parallel_s S') \\
DSC'_0 & \equiv_{df} & DS' \\
DSC'_{i+1} & \equiv_{df} & \textbf{hide } sc* \textbf{ in } (DSC'_i[sc*/cr*] \parallel_s C) \\
DSC' & \equiv_{df} & DSC'_{n_c-1} \\
DSCA'_0 & \equiv_{df} & DSC' \\
DSCA'_{i+1} & \equiv_{df} & \textbf{hide } as* \textbf{ in } (DSCA'_i[as*/ra*] \parallel_s A) \\
DSCA' & \equiv_{df} & DSCA'_{n_a-1} \\
DSC_FA' & \equiv_{df} & \textbf{hide } sc* \textbf{ in } (DSCA'[sc*/cr*] \parallel_s C_F) \\
DSC_FA'_F & \equiv_{df} & \textbf{hide } as* \textbf{ in } (DSC_FA'[as*/ra*] \parallel_s A_F) \\
R' & \equiv_{df} & \textbf{encaps } AP \textbf{ in } R \\
DP' & \equiv_{df} & \textbf{hide } cr*, ra*, tim \textbf{ in } (DSC_FA'_F \parallel_s R')
\end{array}$$

On the basis of the definitions of DP' and DP, propositions 4.6.43 and 4.6.44 imply that

$$\textbf{hide } sc*, cr*, ra*, as*, ds*, tim \textbf{ in } (\textbf{encaps } AP \textbf{ in } DP) \overset{\text{ndfd}}{\approx} DP'$$

which means that it suffices to verify

$$DP' \models \phi_{safe}$$

6.3.2.2 Abstraction for safety properties

The abstraction $\overline{S}$ used for verifying safety properties is based on that for the liveness properties and enjoys similar characteristics. Defining

$$\begin{array}{rcl}
\overline{S}' & \equiv_{df} & \textbf{encaps } AP \textbf{ in } \overline{S} \\
\overline{DS}' & \equiv_{df} & \textbf{hide } ds* \textbf{ in } (D_{safe} \parallel_s \overline{S}') \\
\overline{DP}' & \equiv_{df} & \textbf{hide } cr*, ra*, tim \textbf{ in } (\overline{DS}' \parallel_s R')
\end{array}$$

the abstraction has the following characteristics similar to the liveness case:

E: $DS' \stackrel{\text{ndfd}}{\prec} \overline{DS}'$

F: **hide** $sc*$ **in** $(\overline{DS}'[sc*/cr*] \parallel_s C) \stackrel{\text{ndfd}}{\prec} \overline{DS}'$

G: **hide** $as*$ **in** $(\overline{DS}'[as*/ra*] \parallel_s A) \stackrel{\text{ndfd}}{\prec} \overline{DS}'$

However, this time the abstraction enjoys also some other properties that can be used to facilitate the verification task. These are:

H: **hide** $sc*$ **in** $(\overline{DS}'[sc*/cr*] \parallel_s C_F) \stackrel{\text{ndfd}}{\prec} \overline{DS}'$

I: **hide** $as*$ **in** $(\overline{DS}'[as*/ra*] \parallel_s A_F) \stackrel{\text{ndfd}}{\prec} \overline{DS}'$

As in the case of liveness properties, Claims E-G guarantee that

$$DSCA' \quad \stackrel{\text{ndfd}}{\prec} \quad \overline{DS}'$$

However, this and Claim H imply first that

$$DSC_FA' \quad \stackrel{\text{ndfd}}{\prec} \quad \overline{DS}'$$

and then, together with Claim I, that

$$DSC_FA'_F \quad \stackrel{\text{ndfd}}{\prec} \quad \overline{DS}'$$

On the basis of the definition of DP', the above implies

$$DP' \quad \stackrel{\text{ndfd}}{\prec} \quad \overline{DP}'$$

Consequently, if we can verify Claims E-I, and

J: $\overline{DP}' \models \phi_{safe}$

we have verified

$$DP' \quad \models \quad \phi_{safe}$$

i.e. that the system fulfils its safety requirements.

Notice also that the additional characteristics H and I mean that the analysis could have been carried through in exactly the same way as here, even if we had represented the unreliability of a channel by building it from n_c faulty buffers C_F, instead of $n_c - 1$ reliable buffers C and one faulty one C_F, and similarly for the acknowledgement channel.

The abstraction for safety properties is described in pseudo-code in Figure 6.11. Starting from the abstraction for liveness properties, it has been obtained by the following simplifications:

- The abstraction may diverge in any state.
- The abstraction no longer carries information about the order in which messages in the transmission window are sent; it only specifies that if a messages is sent, it belongs to the transmission window.
- In connection with the previous simplification, the abstraction no longer carries information about timeouts causing retransmission of messages starting from the beginning of the transmission window.

SENDER ABSTRACTION FOR SAFETY PROPERTIES

Notation

$x \oplus y$, $x \ominus y$ as before

$rnd(x, \ldots, y)$ returns a random element from $\{x, \ldots, y\}$

Variables

$d[(-tw) \ldots (tw-1)]$: table of data items

i, j, k : int /* $i \in \{-tw, \ldots, tw\}$, $j \in \{-tw, \ldots, 0\}$, $k \in \{0, \ldots, w\}$ */

ch : bool /* Tells whether the sender is in a chaotic state. */

/* x, a: temporary variables, as in the sender */

Initially

$i = 0, j = 0, k = 0, ch = \bot$

```
loop infinitely
    choose nondeterministically
              ⊤              ⟶ idle;
    []
              ⊤              ⟶ receive( timeout );
    []
              ch             ⟶ receive( *, * );
    []
              ch             ⟶ send( *, * );
    []
              ¬ch ∧ i < tw   ⟶ receive( data source, d[i] );
                                i := i + 1;
    []
              ¬ch ∧ j < i    ⟶ x := rnd(j, ..., min(i, j + tw) − 1);
                                send( channel, (x ⊕ k, d[x]) );
    []
              ¬ch ∧ j < 0    ⟶ j := j + 1;
    []
              ¬ch            ⟶ receive( ack channel, x );
                                a := (x ⊖ k) + 1;
                                if (a − j ≤ tw) then
                                    d[(−tw) ... (tw − 1 − a)] := d[(−tw + a) ... (tw − 1)];
                                    i := max(−tw, i − a);
                                    j := j − a;
                                    k := x ⊕ 1 /* i.e. = k ⊕ a */
                                else if a ≠ w + 1 then /* if not a copy of the last ack */
                                    ch := ⊤    /* then go to a chaotic state */
                                end if;
    end choose;
end loop;
```

Figure 6.11: SW-protocol: Sender abstraction for safety properties

Corresponding to the first and the third simplification, the abstraction can take the idle action and receive a timeout signal in any state without effect. Reflecting the second simplification, the variable j no longer specifies the element in d to be sent next. Instead, it specifies the first element in d that is in the current transmission window. Each time a message is sent, the abstraction picks randomly an element in the transmission window and sends it.

The first data item to which the sender cannot have seen an acknowledgement is in $d[0]$. Conversely, for any data item in $d[(-tw)\ldots(-1)]$, either the sender has seen or is going to see an acknowledgement to it. Reflecting this, if the current transmission window starts in d from below 0, i.e. if $j < 0$, the abstraction may move the transmission window upwards, i.e. increment j. Intuitively, this corresponds to the event of an acknowledgement message that has earlier been delivered to the acknowledgement buffer, being received by the actual sender.

6.3.2.3 Verifying safety properties

To verify the safety properties of the SW-protocol for given parameters w, tw and rw (and for all n_c and n_a), we constructed the approximation $\overline{S}$ and verified Claims E-J:

$$\begin{array}{rcl}
DS' & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\
\textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\
\textbf{hide } as* \textbf{ in } (\overline{DS}'[as*/ra*] \parallel_s A) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\
\textbf{hide } sc* \textbf{ in } (\overline{DS}'[sc*/cr*] \parallel_s C_F) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\
\textbf{hide } as* \textbf{ in } (\overline{DS}'[as*/ra*] \parallel_s A_F) & \overset{\text{ndfd}}{\prec} & \overline{DS}' \\
\overline{DP}' & \models & \phi_{safe}
\end{array}$$

where $\overline{DS}'$ and $\overline{DP}'$ are as before. Claims E-I were verified using the ARA tools. As with the liveness properties, we first run the reduction algorithm on $\overline{DS}'$, producing an alts $\overline{DS}''$ such that $\overline{DS}'' \overset{\text{ndfd}}{\approx} \overline{DS}'$ and then used $\overline{DS}''$ in the place of $\overline{DS}'$ in the rest of the analysis. The table in Figure 6.12 lists sizes of some of the altss involved for some selected parameter values w, tw and rw. When these are contrasted with the sizes of altss in the straightforward model-checking approach (table in Figure 3.4), it is clear how great an advantage using the NDFD-preorder gives in this case.

To verify that

$$\overline{DP}' \models \phi_{safe}$$

holds, i.e. that

$$\begin{array}{rcl}
\overline{DP}' & \models & ((rt1) \Rightarrow (rt1)(\neg (rt0) \wedge \neg (rt1))) \quad \wedge \\
& & ((rt2) \Rightarrow (rt2)(\neg (rt0) \wedge \neg (rt1))) \quad \wedge \\
& & ((rt2) \Rightarrow (rt1))
\end{array}$$

holds, we first constructed $\overline{DP}'$ on the basis of $\overline{DS}''$. After this, the reduction algorithm was run on $\overline{DP}'$ producing an alts $\overline{DP}''$ such that $\overline{DP}' \overset{\text{ndfd}}{\approx} \overline{DP}''$. In all the examined cases, $\overline{DP}''$ turned out to be the alts described in Figure 6.13. From $\overline{DP}''$ it is fairly easy to see informally that the required properties hold.

Parameters			Reachable nodes					
w	tw	rw	$\overline{DS}'$	$\overline{DS}''$	DS'	$\overline{DS}'' \parallel_s C$	$\overline{DS}'' \parallel_s A$	$\overline{DP}'$
1	1	1	26	14	16	38	46	77
2	1	1	38	20	24	56	85	113
	2	1	110	42	60	172	176	516
3	2	2	146	54	80	226	279	513
	3	1	322	127	160	646	649	1801
4	2	2	182	66	100	280	406	633
	3	2	402	155	200	796	945	2736
	4	1	752	310	350	1880	1883	4878
5	2	2	218	78	120	334	557	753
	3	3	482	183	204	946	1297	2170
	4	2	902	365	420	2226	2579	7703
	5	1	1514	723	672	5008	5097	11268
6	2	2	254	90	140	388	732	873
	3	3	562	211	280	1096	1705	2506
	4	3	1052	420	490	2752	3385	10709
	5	2	1766	747	784	5314	6013	18248
7	2	2	290	102	160	442	931	993
	3	3	642	239	320	1246	2169	2842
	4	4	1202	475	560	2918	4301	7313
	5	3	2018	843	896	6016	7625	26809
Notice that $\lvert\overline{DS}'' \parallel_s C\rvert = \lvert\overline{DS}'' \parallel_s C_F\rvert$ and $\lvert\overline{DS}'' \parallel_s A\rvert = \lvert\overline{DS}'' \parallel_s A_F\rvert$								

Figure 6.12: SW-protocol: Sizes of abstraction altss for safety properties

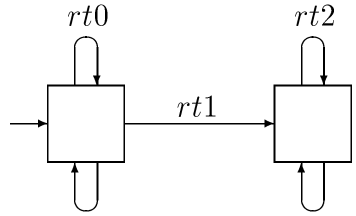


Figure 6.13: SW-protocol: Alts $\overline{DP}''$

6.4 Sliding window protocol: variation

In Subsection 2.5.3.1 and Figure 2.13 we introduced a variant of the SW-sender process differing from the original only in the logic of deciding when to treat an acknowledgement as valid and when not to. Modelling this alternative sender by an alts S_a and building an alts DP_a corresponding to the complete system on the basis of S_a can be done exactly as with the original system. Let us look now briefly at verifying the required properties of the variant.

The verification of the safety properties of the variant is very easy, as it turns out that $DS'_a \overset{\text{ndfd}}{\prec} \overline{DS}'$, where

$$\begin{aligned} DS'_a &\equiv_{df} \textbf{hide } ds* \textbf{ in } (D_{safe} \parallel_s S'_a) \\ S'_a &\equiv_{df} \textbf{encaps } AP \textbf{ in } S_a \end{aligned}$$

and $\overline{DS}'$ is as before. This means that we can use the same abstraction $\overline{S}$ as before, and the only new point to be verified is $DS'_a \overset{\text{ndfd}}{\prec} \overline{DS}'$.

Unfortunately, the same approach does not work in the case of liveness properties. Trying to use the same abstraction $\overline{S}$ as with liveness properties in the original case fails, since $DS'_a \overset{\text{ndfd}}{\not\prec} \overline{DS}'$. What we can do, though, is to modify $\overline{S}$ slightly so as to yield an abstraction $\overline{S}_a$ that treats acknowledgements as valid or invalid according to the same logic as S_a. Now $DS'_a \overset{\text{ndfd}}{\prec} \overline{DS}'_a$. However, when we now build $\overline{DP}'_a$ on the basis of $\overline{DS}'_a$, it quite surprisingly turns out that $\overline{DP}'_a \not\models \phi_{live}$. This in itself is not necessarily anything alarming; as was pointed out earlier, we cannot infer anything about whether $DP'_a \models \phi_{live}$ or not on the basis of $DP'_a \overset{\text{ndfd}}{\prec} \overline{DP}'_a$ and $\overline{DP}'_a \not\models \phi_{live}$.

However, after careful analysis of the reasons behind $\overline{DP}'_a$ failing ϕ_{live}, it emerged that there actually *are* values of the parameters w, tw and rw such that $DP'_a \not\models \phi_{live}$, i.e. the problem was not choosing a bad abstraction but the fact that the system indeed does not fulfil the liveness requirement.

To see an example of this failure, let us look at a case where parameter values are $w = 2$, $tw = 2$, $rw = 1$ and $n_c = n_a = 1$, and the system is connected to a data source that would like to send an infinite sequence of 0:s. Figure 6.14 contains now an example of an infinite execution sequence of the system that is fair but not live.

Notice first that the sequence in the example is certainly fair with respect to the channel, as the channel never loses a message. It is also fair with respect to the acknowledgement channel, since the only message that is send to acknowledgement channel infinitely often (in steps 16 and 20) is $ra1$, and the corresponding message $as1$ is delivered to the sender infinitely often (in step 17). Looking at the original liveness requirements LIVE-1, LIVE-2 and LIVE-3, notice also that the sequence fulfils the liveness property LIVE-1, since the system transmits as many data items to the target as have been read from the source. It also fulfils LIVE-2 trivially. However, it does not fulfil LIVE-3 requiring the system to accept data from the source if there is any data available, as after the first two data items, no more are accepted along this sequence.

Intuitively, the core of this counterexample is that although the acknowledgement message $as1$ is delivered to the sender infinitely often, it always arrives at a time when the sender regards it as invalid and does not pay attention to it.

Step	Action	Explanation
1	$ds0$	sender receives data item 0 from source
2	$ds0$	sender receives data item 0 from source
3	$sc00$	sender sends first data item to channel
4	$cr00$	channel forwards it to receiver
5	$rt0$	receiver forwards it to target
6	$ra0$	receiver sends an ack to first data item to ack channel
7	τ	ack channel loses it
8	$sc10$	sender sends second data item to channel
9	$cr10$	channel forwards it to receiver
10	$rt0$	receiver forwards it to target
11	$ra1$	receiver sends an ack to second data item to ack channel
12	τ	ack channel loses it
13	tim	timeout: sender starts sending data items again
14	$sc00$	sender sends first data item to channel
15	$cr00$	channel forwards it to receiver
16	$ra1$	receiver sends an ack to **second** data item to ack channel
17	$as1$	ack channel forwards it to sender, **sender ignores it as invalid**
18	$sc10$	sender sends second data item to channel
19	$cr10$	channel forwards it to receiver
20	$ra1$	receiver sends an ack to second data item to ack channel
21	τ	ack channel loses it
22 –	repeating from 13 –	

Parameters: $w = 2$, $tw = 2$, $rw = 1$, $n_c = n_a = 1$.

Figure 6.14: SW-protocol: Failure of liveness

Chapter 7

Related Work

In this chapter we review some other work that has been done on compositional verification and discuss its relation to the work presented here.

7.1 Equivalences and logics

The process algebra literature contains a bewildering number of different notions of equivalence or preorder. For a survey of various equivalences and their relations to each other, see [vGl90, vGl93]. A basic way of classifying equivalences is whether they are simulation or bisimulation-based, such as observation equivalence [Mil89], or whether they are trace or failure-based, such as language equivalence or failures equivalence [BHR84, Hoa85].

The equivalences discussed in the current work belong to the category of failure-based ones. Such equivalences have been discussed earlier in at least [DH84, BHR84, Hoa85, BR85, BKO87, Hen88]. The equivalences presented in these papers mainly differ from each other in their treatment of divergences. However, a common factor is that none of them necessarily preserves all divergences of a system. The first failure-based equivalence that had this property and was compositional with respect to the standard operators was the CFFD or *chaos-free failures divergences* equivalence, introduced by Valmari and Tienari in [VT91]. The NDFD-equivalence discussed in the current work evolved from this work, and originally appeared in [KV92] and [VT95], formulated for standard labelled transition systems (i.e. without state labels). Independent of the thread of research to which the current work belongs, Darondeau and Gamatie have also arrived at a semantic model that is essentially the same as NDFD-equivalence, starting from an observational model based on infinitary tests [DG90]. For further discussion on the CFFD and NDFD-equivalences and their relation to other failure-based equivalences, see [VT91, VT95].

Much of the theoretical part of the current work concentrated on establishing the exact correspondence between NDFD-equivalence and the minimal LTL'-preserving congruence. Similar correspondence results are known for several other equivalences and logics. For a survey of some of these, see [Sti92, Section 5.3].

Perhaps the best known example in this area is the correspondence between the strong bisimulation or strong observation equivalence and the Hennessy-Milner logic, a modal logic with indexed modalities $<a>\ \phi$ and $[a]\phi$, *after some/every a-transition, ϕ holds* [HM85, Mil89]. One direction of this correspondence states that if two transition systems are strong bisimulation equivalent, then they satisfy same formulae of Hennessy-Milner logic. The other direction states that if two transition systems are image-finite and satisfy same formulae of Hennessy-Milner logic, then they are strong bisimulation equivalent. The requirement of image-finiteness in the second half can be disposed with, if the logic is extended with an infinitary conjunction operator, as was done here is Section 4.6.3. The correspondence also carries over to the modal mu-calculus, a language extending the Hennessy-Milner logic with fixpoint operators [Sti92]. The original correspondence result is formulated for normal labelled transition systems, i.e. systems with transition labels but without state labels. However, as pointed out in [Sti92], the result can be naturally generalised to a semantic model corresponding to altss. In the model with state labels, the strong bisimulation equivalence can be defined as follows.

Definition 7.1.1 Let $L = (S, I, \Gamma, \Delta, A, V)$ and $L' = (S', I', \Gamma', \Delta', A', V')$ be altss such that $|I| = |I'| = 1$. We say that L and L' are *strong bisimulation equivalent* iff $\Gamma = \Gamma'$, $A = A'$ and there is a relation $R \subseteq S \times S'$ such that for all $s \in S$ and $s' \in S'$, if sRs' then:

- $s \in I$ iff $s' \in I'$,
- $V(s) = V'(s')$,
- if $(s, l, t) \in \Delta$, then there is some $t' \in S'$ such that $(s', l, t') \in \Delta'$ and tRt', and
- if $(s', l, t') \in \Delta'$, then there is some $t \in S$ such that $(s, l, t) \in \Delta$ and tRt'. □

A correspondence analogous to the above exists between the weak bisimulation equivalence, also known as observation equivalence, and weak versions of the Hennessy-Milner logic and the modal mu-calculus [Sti92].

Closer to the framework of the current work, several correspondences between simulation-based equivalences and branching time temporal logics are also known. Browne, Clarke and Grumberg [BCG88] show that two finite Kripke structures, i.e. transition systems with state labels but no transition labels, satisfy the same formulae of the full branching time temporal logic CTL^* precisely when they are strong bisimulation equivalent. The logic CTL^* extends standard linear time temporal logic with path quantifiers or path switching operators $\exists\phi$ and $\forall\phi$, *for some/every path ϕ holds.* The same paper also contains a weak version of the correspondence result, showing that two finite Kripke structures satisfy the same formulae of the logic CTL^* without nexttime-operator iff the structures are equivalent *with respect to stuttering.* The concept of being equivalent with respect to stuttering is defined as a limit of an infinite sequence of simulation-based equivalence relations. The results of [BCG88] have been generalised by Josko [Jos90] to a semantic model consisting of Kripke structures with additional constraints on the possible behaviours of their environments.

The relation between bisimulation-based equivalences and the nexttime-less version of CTL^* was examined further by De Nicola and Vaandrager [DV90b, DV90c]. They first consider a scenario where the logic is interpreted in a nonstandard fashion. Intuitively, the difference with the normal interpretation is that while usually the logic is interpreted over maximal paths, here it is interpreted over all paths. In the current work this would correspond to changing the definition 3.2.3

$L \models \phi$ iff for all $\sigma \in \mathit{maxbeh}(L)$: $\sigma \models \phi$

to

$L \models \phi$ iff for all $\sigma \in \mathit{beh}(L)$: $\sigma \models \phi$

Using this nonstandard interpretation, it is shown that two finite Kripke structures satisfy the same formulae of CTL^* without nexttime iff the structures are *divergence blind stuttering* equivalent iff they are *branching bisimulation* [vGW89, vGW91] equivalent. Furthermore, using the standard interpretation, it is shown that two finite Kripke structures satisfy the same formulae of CTL^* without nexttime iff the structures are *divergence sensitive stuttering* equivalent iff they are *divergence sensitive branching bisimulation* equivalent. The results of De Nicola and Vaandrager are formulated for transition systems without state labels, and involve an additional implicit decoding of state labels from transition labels to allow interpreting state-based logics over such systems.

In the area of linear-time temporal logics, several correspondence results are also known. In [Sti92] Stirling shows that if two systems are *history* equivalent, then they validate the same formulae of the full linear-time temporal logic (with nexttime), and that the converse holds if the systems are finite, or the logic has an infinitary conjunction operator. In the terminology of the current work, two systems are history equivalent iff they have the same maximal behaviours. For the nexttime-less version of the language, i.e. LTL', the analogous correspondence result is formulated in theorems 4.1.12, 4.1.18 and 4.6.37. The first half of this, i.e. that nexttime-less linear temporal logic is immune to finite stuttering was originally pointed out by Lamport in [Lam83]. However, the other direction was not discussed in his work. Furthermore, neither [Sti92] nor [Lam83] consider compositionality issues in this context.

7.2 Compositional verification techniques

7.2.1 Equivalence-based compositional reduction

The technique of compositional reduction, i.e. reducing components of a system with respect to an equivalence before building the complete system from them, is very natural in the context of any compositional equivalence. The earliest explicit reference to this technique that we are aware of is in the work [SLU89] of Sabnani & al., who use compositional reduction with respect to observation equivalence to verify safety properties of communication protocols. Compositional reductions using observation equivalence also form the basis of the tool *AUTO* of Madelaine and Vergamini [MV89]. In [VT91] Valmari and Tienari discuss compositional reduction using CFFD-equivalence, and in [Val93] Valmari examines general preconditions of compositional reductions.

Closer to the theme of the current work, equivalence-based compositional reduction for verifying temporal logic properties has also been discussed in the literature. In [DGG93] Dams & al. use reduction with respect to simulation equivalence to verify properties expressed in the universal fragment of CTL^*, i.e. the class of CTL^* formulae without existential path quantification.

Clarke & al. [CLM89] describe a slightly more complicated scheme for verifying temporal logic properties, based on compositional reduction. For compositional reasoning, they first formulate a general *interface rule*, which works for any compositional equivalence $\approx$ and any logic that the equivalence preserves, given certain natural technical assumptions. The rule works intuitively as follows. Assume that we want to verify $L_1 \parallel L_2 \models \phi$ where ϕ only makes reference to visible characteristics of L_2. This is done by hiding all the features of L_1 that are not relevant for the interaction between L_1 and L_2, and reducing the resulting L_1' with respect to the equivalence $\approx$. Writing A_1 for the reduced component, we can now verify $L_1 \parallel L_2 \models \phi$ by verifying $A_1 \parallel L_2 \models \phi$. As a particular instance of this general scheme, the paper examines verification of nexttime-less CTL^* properties using the stuttering equivalence of [BCG88].

Both [CLM89] and [DGG93] concentrate on branching time logics and simulation-based equivalences. Outside the current work, we are not aware of equivalence-based compositional reductions having been used in the context of linear time logics and failure-based equivalences.

All the works listed above use the straightforward approach of applying a reduction algorithm w.r.t. some equivalence to the components of a system. However, in systems where components interact closely, they can strongly restrict each other's possible behaviour, and considered individually the components may contain potential for behaviour that is never possible in the context of the complete system. Reducing such components individually with respect to an equivalence requires then a large amount of unnecessary work, as also those parts of a component that will not be relevant in the context of the complete system need to be taken into account. Because of this problem, Graf and Steffen [GS90] adopted a preorder instead of an equivalence as the basis of their compositional reduction method. The preorder is based on a simulation relation with *undefinedness predicates* $p \uparrow a$, intuitively expressing that the result of executing an a-transition from state p is undefined. In this way we can abstract away from a component information about its behaviour after events that should not take place in the context of the complete system. Notice that a similar idea was used in in Section 6.3 of the current work, in constructing abstractions for the sliding window sender process. As in the current work, the move from equivalences to preorders in [GS90] means that unlike the methods listed above, this method is no more fully automatisable, but the verifier must supply so-called *interface specifications*, which limit the sets of sequences of events that need to be considered to some supersets of those that are actually possible in the context of the complete system.

7.2.2 Other compositional reduction

In addition to the compositional reduction methods discussed in previous paragraphs, there are also several approaches based on similar principles but not formulated in terms of equivalences or preorders.

In [CGL94], Clarke, Grumberg and Long present a method of compositional reduction using abstraction mappings. Using the notation of the current work, an abstraction mapping for a labelled transition system $L = (S, I, \Gamma, \Delta)$ is a function h from the set

S of states of L to some smaller set of states S'. Such a mapping induces naturally a transition system $L' = (S', I', \Gamma, \Delta')$ by $I' = \{h(s) \mid s \in I\}$ and $(s'_1, l, s'_2) \in \Delta'$ iff there are $s_1 \in h^{-1}(s'_1)$ and $s_2 \in h^{-1}(s'_2)$ such that $(s_1, l, s_2) \in \Delta$. The approach of [CGL94] is based on applying user-given abstraction mappings to reduce components of a system before constructing the state space of the complete system from the reduced components. It is shown that this approach produces correct results in verifying properties expressed in the universal fragment of CTL^*. An issue that is not addressed in [CGL94] is that an abstraction mapping does not necessarily preserve all the deadlocks of a system. In the framework of communicating Moore machines that is used in the paper this does not matter, as by definition a system composed of these cannot deadlock. However, it means that applying the theory to an asynchronous model similar to that used in the current work is problematic.

Further to their work already discussed above, Dams & al. [DGG93] describe an algorithm constructing reduced models for verifying that a given single formula of the universal fragment of CTL holds over a system. Notice the difference between the aim here and in the previous approaches. Here the reductions used depend on the actual property to be verified, whereas the equivalences and mappings discussed above preserve a whole class of properties. A similar approach for any CTL formula is discussed by Shiple & al. in [SCSB92].

For a very general view on transformations of systems and properties preserved by such transformations, see [BBLS92].

Finally, notice that the technique of data independence introduced by Wolper [Wol86] and used here in the verification of the sliding window protocol is also essentially a reduction technique, allowing us to ignore those data values that are not essential for the verification task. For another application of this technique to protocol verification, see [Sab88].

7.2.3 Direct compositional verification

All the work on compositional verification discussed here this far can be categorised as being based on compositional reduction. The idea has been to construct a reduced version of the global state space of a system and verify the required properties on the basis of it. However, there is also another main category, direct compositional verification. Here the aim is to verify properties of a system directly on the basis of the individual components, without ever constructing a global state space of the system. Let us review very briefly some of the main developments in this research theme.

Much of the work on direct compositional verification of temporal logic properties [BKP84, Pnu85, Sti85, Bar86, NDGO86] has concentrated on devising sound and complete proof rules by which properties of systems could be derived from properties of their components. To have a flavour of this approach, consider the method of compositional verification of Manna and Pnueli [MP91] discussed above in Section 5.2. To verify $L_1 \parallel_v L_2 \models \phi$ by this method, we can use a reasoning rule allowing us to infer $L_1 \parallel_v L_2 \models \phi$ from $L_1 \models^v_m \phi_1$, $L_2 \models^v_m \phi_2$ and $\models \phi_1 \wedge \phi_2 \Leftrightarrow \phi$. Notice that in this approach the proof rule essentially specifies what counts as valid reasoning, not what steps of reasoning should be taken to verify a particular system, which means that these verification methods are not amenable to full automatisation. However, they can naturally be used to divide the verification task to subtasks (such as verifying $L_1 \models^v_m \phi_1$ above) that can be succesfully automated.

Based on this idea, a full specification methodology for verifying temporal logic properties of Petri nets has been developed by Josko & al. [Jos87, DDGJ89, Jos89]. Their approach follows the so-called *assumption-guarantee* paradigm [Pnu85], where the specification of a module L is of the form 'assuming that the environment of L fulfils a property ϕ, then L together with its environment should fulfil ψ'. Showing that a module fulfils its assumption-guarantee specification can be done with a model-checking algorithm described in [Jos89]. The specifications of the modules of a system can then be manually combined by a set of composition rules [DDGJ89] to show that the complete system fulfils its requirements. A similar approach using assume-guarantee style reasoning for communicating Moore-machines is described by Grumberg and Long in [GL91].

An approach for verifying temporal logic properties of Petri nets, based on direct compositional model checking, is presented by Best and Esparza in [BE91]. The essential difference between this work and the ones discussed above is that here verification takes place fully automatically. However, the drawback is that the method can handle only a relatively weak logic.

An entire research area in the field of compositional verification is using partial orders to alleviate the state explosion problem [GW94, KP92, Val92, Val90b, WG93]. The idea behind the approach is that instead of considering all possible interleavings of actions of individual components that are executed concurrently, as is usually done when constructing the global state space of a system, one tries to verify the required properties by examining only some representative interleavings. For surveys of the area, see [WG93, Val94].

7.3 Sliding window protocol

Finally, let us review some other verification case studies discussing the sliding window protocols of HDLC protocol, and their relation to the current work.

In Stenning's original paper on SW-protocol [Ste76], safety properties of the protocol are verified manually using program assertions. In the formal part of the verification it is assumed that sequence numbers for messages are unbounded, ever increasing. Stenning then argues informally that sequence numbers modulo n suffice, supposing that n is related to the window sizes by a certain equation. Liveness properties of the protocol are not analysed in this work.

Like Stenning's work, most other verification studies on SW-protocol were done manually or concentrated on safety properties, often both. Knuth [Knu81] examines more general principles behind the particular protocol proposed by Stenning, and verifies manually safety properties of a class of protocols built on the same principles. Higashi & al. [H&al84] apply an algebraic specification method to verify safety properties of the data transfer part of the HDLC protocol.

Shankar and Lam [SL83] consider a version of the full HDLC protocol and verify manually safety properties of the protocol using the technique of image projections. The liveness of the protocol is shown by reasoning about its real-time behaviour. Instead of a property like 'every data item will eventually be delivered to the target', the liveness property verified is: 'if within a time duration t a data item is not delivered to the target, then at least n retransmissions of the data block have reoccurred, all of which have failed.' In Shankar's and Lam's more recent work [SL92], several versions of the sliding window protocol are derived from their specifications by a manual stepwise refinement technique. Both safety and liveness properties are considered.

Hailpern [Hai82] verifies both safety and liveness properties of the SW-protocol, assuming unbounded sequence numbers, using temporal logic to formulate the requirements and informal reasoning to establish the validity of these requirements. Jonsson [Jon87] also verifies both safety and liveness properties of the protocol using temporal logic and a manual compositional verification technique. Another manual proof of both safety and liveness properties of the protocol appears in [vdS95].

The earliest case study to apply an automated verification technique to the SW-protocol was by Brand and Joyner [BJ82]. They analyse safety properties of the HDLC protocol and verify them for the most part automatically, using symbolic execution.

Another automated approach, closer to the current work, was taken by Richier & al. [RRSV87]. They use temporal logic as the specification language and Estelle/R, a process-algebra-based language, for describing the protocol. The verification method is brute-force model-checking: the total state space of the system is first constructed and then the required properties checked from it automatically. The analysis covers only safety properties of the protocol. Due to the state-explosion problem, verification is only possible for fairly restricted values of parameters.

In all these case studies, we can see two trends: either looking at an abstract, high-level description of the protocol, in a sense verifying the validity of the general principle of a sliding window, or looking at a particular, rather detailed implementation. A good example of the former kind of analysis is Jonsson's work [Jon87]. Here the protocol is described by stating when different events of the protocol entities are possible, and giving a fairness constraint specifying loosely when events have to take place. For example, the order in which sender transmits messages is not specified and the mechanism allowing messages to be retransmitted is not discussed. Of the other works listed above, [Knu81] and [SL92] take similar abstract views. In the latter, more detailed type of analysis the protocol is usually described in a directly implementable level. The approaches taken in [Ste76, H&al84, SL83, Hai82, RRSV87] and the current work belong to this group.

The advantage of the more abstract approach is that the verification task becomes more manageable and proofs of correctness often more elegant. On the other hand, the distance from the verified protocol to an actually implementable one grows, and it becomes easier to introduce errors in this stage. For example, the difference that was noticed in the current work between the live and the faulty versions of the protocol does not surface at all in the more abstract level. The advantages and disadvantages of the more implementation-like description are directly opposite. In the verification work much more effort is needed to take care of all the details, but implementing the verified protocol becomes more straightforward.

Chapter 8

Conclusion

In this work, we have examined notions of equivalence and preorder between transition systems with the aim of applying these to ease the verification of properties of concurrent systems. The precise framework that has been considered above is based on using nexttime-less linear-time temporal logic LTL' for the description of properties, and using transition systems with labels on both states and transitions for the description of systems and their components.

The main contribution of the work is locating NDFD-equivalence as the unique weakest equivalence that preserves all nexttime-less linear-time temporal logic properties, and is compositional with respect to the rather standard set of operators used in this work. We also feel that the technical points of using a system model with labels on both states and transitions, and dealing with both synchronisation and variable-based interaction in the same model may be of some general interest.

The full-abstractness result for the NDFD-equivalence means that it is both a necessary and a sufficient condition for replacing a module of a system with another one without affecting the LTL'-properties of the complete system. But does this necessarily mean that the NDFD-equivalence is the best criterion in practice? After all, in Section 4.5 we saw that determining NDFD-equivalence of two systems is a PSPACE-complete problem, and therefore rather hard in practice in the worst case.

This issue is really an instance of a more general divide between simulation-based and failure-based equivalences. From the point of view of compositional reductions, the two classes of equivalences generally differ in two important aspects. On the one hand, simulation-based equivalences are usually more discerning than failure-based ones, since they record the branching structure of a system. On the other hand, the algorithmic complexity of checking the equivalence of two systems or minimising a system with respect to an equivalence is usually lower for bisimulation-based ones. For example, checking whether two systems are observation equivalent can be done in polynomial time, whereas this problem for trace equivalence or NDFD-equivalence is PSPACE-hard.

This difference in algorithmic complexity has sometimes been understood as showing that simulation-based equivalences are better suited for compositional reduction than failure-based ones. However, this argument is fallacious for two reasons. First, the suitability of an equivalence for this task depends not only on the efficiency of the reduction algorithm but also on the amount of reduction possible. Here simulation-based equivalences are superior in the first count, but inferior in the second, since being more discerning implies having less potential for reduction. In a sense there is a tradeoff between how much resources it takes to reduce a component and how small the resulting reduced component will be. Secondly, the issue in compositional reduction is *reduction*, not minimisation. The fact that the minimisation problem for failure-based equivalences is PSPACE-hard, does not imply that no efficient reduction algorithm can exist. In particular, if any other equivalence preserves nexttime-less linear-time temporal logic properties and is a congruence, this equivalence is always a refinement of the NDFD-equivalence. If there is a fast minimisation algorithm for this other equivalence, the algorithm can therefore be readily used as a reduction algorithm for the NDFD-equivalence. On top of this, the weaker NDFD-equivalence may have some other reduction strategies that are not available for the more restrictive equivalence. Consequently, using the NDFD-equivalence it is always possible to achieve at least the same amount of reduction as using any stronger equivalence. However, as it does not have the same restrictions as the stronger equivalences, it is also possible to achieve better results. For a thorough analysis of these issues, see [Val95].

In Chapter 5 we examined the preorder corresponding to the NDFD-equivalence. Although the theoretical results for the NDFD-preorder are very similar to those for the equivalence, there is an important difference between the styles of verification using the preorder and the equivalence. In short, verification using the equivalence and compositional reductions can be done entirely automatically, whereas using the preorder requires educated guesses on the part of the verifier. The advantage this gives is that the verifier can use her intuitions about the system to ease the verification task, which in turn means that the range of applicability of the approach is wider. It should be noticed here that the validity of the result of the verification is not dependent on whether the guesses made by the verifier happen to be correct or not: checking that the required preorder relations between modules of a system and their replacements indeed hold, and that the resulting system fulfils its requirements can be done automatically, and if these checks are succesful, the original system is guaranteed to fulfil the requirements.

The power of the NDFD-preorder-based verification was illustrated by the main case study of the current work, the sliding window protocol. As far as we know, the current work is the first to apply a semi-automated verification technique to the liveness properties of the SW-protocol beyond the simple subcase of the alternating bit protocol. The analysis here also handled a wider range of parameter values than in the previous studies taking an automated approach and, in particular, dealt with channels of arbitrary length. Moreover, the analysis allowed us to find a previously unrecognised liveness problem in the alternative version of the protocol.

In our opinion this last point illustrates two more general issues. First, SW-protocol is a widely used protocol that in itself is both safe and live. However, a seemingly minor change to the system causes it to fail the liveness requirement in a fairly nontrivial way. What is more, this change is natural and seems unimportant enough that, for example, an earlier case study [RRSV87] verified the faulty variant of the system (although only safety properties of it), and we originally started working on the basis of the same variant, as well. In our opinion, this is a good example of the benefits of the rigorousness induced by a formal verification method. In an informal approach to verifying the liveness properties, the fault would have probably been quite easy to overlook.

Secondly, the example serves as a reminder of the fact that liveness properties cannot be reliably verified unless divergences and fairness conditions are properly kept track of. For example, if channel events are abstracted away and the services provided by the protocols compared with respect to observation equivalence [Mil89], which ignores divergences, both the original, correct version and the modified, faulty version of the protocol would be considered equivalent and the difference overlooked. In doing this we would in essence be confusing two different kinds of divergences: those which correspond to unfair executions and can therefore be ignored, and those which correspond to fair executions and should not be ignored.

Bibliography

[Arn94] Arnold, A.: *Finite Transition Systems*, Prentice-Hall, 1994

[AHU74] Aho, A. V. & Hopcroft, J. E. & Ullman, J. D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974

[AS89] Alpern, B. & Schneider, F. B.: Verifying temporal properties without temporal logic, in *ACM Transactions on Programming Languages*, vol. 11, 1989, pp. 147-167

[BDG88] Balcázar, J. L. & Diáz, J. & Gabarró, J.: *Structural Complexity I*, EATCS Monographs on Theoretical Computer Science, vol. 11, Springer-Verlag, 1988

[Bar86] Barringer, H.: Using temporal logic in the compositional specification of concurrent systems, in Galton, A. (ed.): *Temporal Logics and Their Applications*, Academic Press, 1987, pp. 59-90

[BKP84] Barringer, H. & Kuiper, R. & Pnueli, A.: Now you may compose temporal logic specifications, in *Conference Record of the Sixteenth Annual ACM Symposium on Theory of Computing*, 1984, pp. 51-63

[BBLS92] Bensalem, S. & Bouajjani, A. & Loiseaux, C. & Sifakis, J.: Property preserving simulations, in *Computer Aided Verification, Fourth International Workshop, CAV'92, Proceedings*, LNCS, vol. 663, Springer-Verlag, 1993, pp. 260-273

[BKO87] Bergstra, J. A. & Klop, J. W. & Olderog, E.-R.: Failures without chaos: a new process semantics for fair abstraction, in *Formal Description of Programming Concepts III, Proceedings*, North-Holland, 1987, pp. 77-103

[BE91] Best, E. & Esparza, J.: Model checking of persistent Petri nets, in *Computer Science Logic, 5th Workshop, CSL'91, Proceedings*, LNCS, vol. 626, Springer-Verlag, 1992, pp. 35-52

[BB89] Bolognesi, T. & Brinksma, E.: Introduction to the ISO specification language LOTOS, in *The Formal Description Technique LOTOS*, North-Holland, 1989, pp. 23-73

[BC89] Bolognesi, T. & Caneve, M.: Equivalence verification: theory, algorithms and a tool, in *The Formal Description Technique LOTOS*, North-Holland, 1989, pp. 303-326

[BS92] Bradfield, J. & Stirling, C.: Local model checking for infinite state spaces, in *Theoretical Computer Science*, vol. 96, 1992, pp. 157-174

[BJ82] Brand, D. & Joyner, W. H.: Verification of HDLC, in *IEEE Transactions on Communications*, vol. COM-30, no. 5, 1982, pp. 1136-1142

[Bro83] Brookes, S. D.: On the relationship of CCS and CSP, in *Automata, Languages and Programming, 10th Colloquium*, LNCS, vol. 154, Springer-Verlag, 1983, pp. 83-96

[BR83] Brookes, S. D. & Rounds, W. C.: Behavioural equivalence relationships induced by programming logics, in *Automata, Languages and Programming, 10th Colloquium*, LNCS, vol. 154, Springer-Verlag, 1983, pp. 97-108

[BR85] Brookes, S. D. & Roscoe, A. W.: An improved failures model for communicating sequential processes, in *Seminar on Concurrency, held at Carnegie-Mellon University, 1984, Proceedings*, LNCS, vol. 197, Springer-Verlag, 1985, pp. 281-305

[BHR84] Brookes, S. D. & Hoare, C. A. R. & Roscoe, A. D.: A theory of communicating sequential processes, in *Journal of ACM*, vol. 31, no. 3, 1984, pp. 560-599

[BCG88] Browne, M. C. & Clarke, E. M. & Grumberg, O.: Characterizing Kripke structures in temporal logic, in *Theoretical Computer Science*, vol. 59, 1988, pp. 115-131

[Bur84] Burgess, J. P.: Basic tense logic, in Gabbay, D. & Guenthner, F. (eds.): *Handbook of Philosophical Logic*, vol. II, Reidel, 1984, pp. 89-113

[CES86] Clarke, E. M. & Emerson, E. A. & Sistla, A. P.: Automatic verification of finite state concurrent systems using temporal logic, in *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, 1986, pp. 244-263

[CGL94] Clarke, E. M. & Grumberg, O. & Long, D. E.: Model checking and abstraction, in *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, 1994, pp. 1512-1542

[CLM89] Clarke, E. M. & Long, D. E. & McMillan, K. L.: Compositional model checking, in *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, 1989, pp. 353-362

[CH90] Cleaveland, R. & Hennessy, M.: Testing equivalence as a bisimulation equivalence, in *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, vol. 407, Springer-Verlag, 1990, pp. 11-23

[CPS90] Cleaveland, R. & Parrow, J. & Steffen, B.: The Concurrency Workbench, in *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, vol. 407, Springer-Verlag, 1990, pp. 24-37

[Dam92] Dam, M.: Fixed points of Büchi-automata, in *Proceedings of 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS, vol. 652, Springer-Verlag, 1992, pp. 39-50

[DDGJ89] Damm, W. & Döhmen, G. & Gerstner, V. & Josko, B.: Modular verification of Petri nets, the temporal logic approach, in *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop 1989, Proceedings*, LNCS, vol. 430, Springer-Verlag, 1990, pp. 180-207

[DGG93] Dams, D. & Grumberg, O. & Gerth, R.: Generation of reduced models for checking fragments of CTL, in *Computer Aided Verification, 5th International Conference, CAV'93*, LNCS, vol. 697, Springer-Verlag, 1993, pp. 479-490

[DH84] De Nicola, R. & Hennessy, M.: Testing equivalences for processes, in *Theoretical Computer Science*, vol. 34, 1984, pp. 83-133

[DV90a] De Nicola, R. & Vaandrager, F.: Action versus state based logics for transition systems, in *Semantics of Systems of Concurrent Processes, Proceedings*, LNCS, vol. 469, Springer-Verlag, 1990, pp. 407- 419

[DV90b] De Nicola, R. & Vaandrager, F.: Three logics for branching bisimulation, in *Proceedings of the 5th IEEE Symposium on Logic in Computer Science, LICS'90*, 1990, pp. 118-129 (an extended abstract of [DV90c])

[DV90c] De Nicola, R. & Vaandrager, F.: *Three Logics for Branching Bisimulation*, Report CS-R9012, Centrum voor Wiskunde en Informatica, Amsterdam, 1990

[DG90] Darondeau, P. & Gamatie, B.: Infinitary behaviours and infinitary observations, in *Fundamenta Informaticae*, vol. XIII, 1990, pp. 353-386

[Elo94] Eloranta, J.: *Minimal Transition Systems with respect to Divergence Preserving Behavioural Equivalences*, Ph.D. Thesis, Report A-1994-1, Department of Computer Science, University of Helsinki, Finland, 1994

[Eme90] Emerson, E. A.: Temporal and modal logic, in van Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, Elsevier, 1990, pp. 995-1072

[EL87] Emerson, E. A. & Lei, C. L.: Modalities for model checking: branching time strikes back, in *Science of Computer Programming*, vol. 8, 1987, pp. 275-306

[Ern91] Ernberg, P.: *CCS as a Method of Specification and Verification: Analysis of a Case Study*, Swedish Institute of Computer Science, Technical report T 91:05, 1991

[GPSS80] Gabbay, D. & Pnueli, A. & Shelah, S. & Stavi, J.: On the temporal analysis of fairness, in *Conference Record of the 7th ACM Symposium on Principles of Programming Languages*, 1980, pp. 163-173

[GJ78] Garey, M. R. & Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1978

[vGl90] van Glabbeek, R. J.: The linear time – branching time spectrum, in *CONCUR'90, International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 458, Springer-Verlag, 1990, pp. 278-297

[vGl93] van Glabbeek, R. J.: The linear time – branching time spectrum II, the semantics of sequential systems with silent moves, in *CONCUR'93, 4th International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 715, Springer-Verlag, 1993, pp. 66-81

[vGW89] van Glabbeek, R. J. & Weijland, W. P.: Branching time and abstraction in bisimulation semantics (extended abstract), in *Proceedings of the 11th IFIP International Conference on Information Processing '89*, IFIP, North-Holland, 1989, pp. 613-618

[vGW91] van Glabbeek, R. J. & Weijland, W. P.: *Branching Time and Abstraction in Bisimulation Semantics*, Report CS-R9120, Centrum voor Wiskunde en Informatica, Amsterdam, 1991

[GW94] Godefroid, P. & Wolper, P.: A partial approach to model checking, in *Information and Computation*, vol. 110, 1994, pp. 305-326

[GS90] Graf, S. & Steffen, B.: Compositional minimization of finite-state systems, in *Computer Aided Verification, 2nd International Conference, CAV'90, Proceedings*, LNCS, vol. 531, Springer-Verlag, 1991, pp. 186-196

[GL91] Grumberg, O. & Long, D. E.: Model checking and modular verification, in *CONCUR'91, 2nd International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 527, Springer-Verlag, 1991, pp. 250-265

[Hai82] Hailpern, B. T.: *Verifying Concurrent Processes Using Temporal Logic*, LNCS, vol. 129, Springer-Verlag, 1982

[HM85] Hennessy, M. C. & Milner, A. J. R. G.: Algebraic laws for nondeterminism and concurrency, in *Journal of ACM*, vol. 32, no. 1, 1985, pp. 137-161

[Hen88] Hennessy, M.: *Algebraic Theory of Processes*, MIT Press, 1988

[H&al84] Higashino, T. & Mori, M. & Sugiyama, Y. & Taniguchi, K. & Kasami, T.: An algebraic specification of HDLC procedures and its verification, in *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, 1984, pp. 825-836

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice-Hall, 1985

[HU79] Hopcroft, J. E. & Ullman, J. D.: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979

[ISO79a] International Standards Organisation: *Data Communications – HDLC Procedures – Frame Structure*, Ref. No. ISO 3309, International Standards Organisation, Geneva, 1979

[ISO79b] International Standards Organisation: *Data Communications – HDLC Procedures – Elements of Procedures*, Ref. No. ISO 4335, International Standards Organisation, Geneva, 1979

[ISO80] International Standards Organisation: *Data Communications – HDLC Unbalanced Class of Procedures*, Ref. No. ISO 6159, International Standards Organisation, Geneva, 1980

[ISO89] International Standards Organisation: *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, Ref. No. ISO 8807, International Standards Organisation, Geneva, 1989

[Jon87] Jonsson, B.: *Compositional Verification of Distributed Systems*, DoCS 87/09, Ph.D. Thesis, Department of Computer Systems, Uppsala University, Uppsala, Sweden, 142 pp.

[Jos87] Josko, B.: Modelchecking of CTL formulae under liveness assumptions, in *Automata, Languages and Programming, 14th International Colloquium*, LNCS, vol. 267, Springer-Verlag, 1987, pp. 280-289

[Jos89] Josko, B.: Verifying the correctness of AADL-modules using model checking, in *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop 1989, Proceedings*, LNCS, vol. 430, Springer-Verlag, 1990, pp. 386-400

[Jos90] Josko, B.: A context dependent equivalence relation between Kripke structures, in *Computer Aided Verification, 2nd International Conference, CAV'90, Proceedings*, LNCS, vol. 531, Springer-Verlag, 1991, pp. 204-213

[KV91] Kaivola, R. & Valmari, A.: Using truth-preserving reductions to improve the clarity of Kripke-models, in *CONCUR'91, 2nd International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 527, Springer-Verlag, 1991, pp. 361-375

[KV92] Kaivola, R. & Valmari, A.: The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic, *CONCUR'92, Third International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 630, Springer-Verlag, 1992, pp. 207-221

[Kai92] Kaivola, R.: Compositional model checking for linear-time temporal logic, in *Computer Aided Verification, Fourth International Workshop, CAV'92, Proceedings*, LNCS, vol. 663, Springer-Verlag, 1993, pp. 248-259

[Kai93] Kaivola, R.: *Compositional Linear Temporal Logic Model-Checking for Concurrent Systems*, Lic. Phil. thesis, Report C-1993-1, University of Helsinki, Department of Computer Science, 1993, 100 p.

[KS90] Kanellakis, P. & Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence, in *Information and Computation*, vol. 86, 1990, pp.43-68

[KP92] Katz, S. & Peled, D.: Verification of distributed programs using representative interleaving sequences, in *Distributed Computing*, vol. 6, 1992, pp. 107-120

[KMMP93] Kesten, Y. & Manna, Z. & McGuire, H. & Pnueli, A.: A decision algorithm for full propositional temporal logic, in *Computer Aided Verification, 5th International Conference, CAV'93*, LNCS, vol. 697, Springer-Verlag, 1993, pp. 97-109

[Knu81] Knuth, D. E.: Verification of link-level protocols, in *BIT*, vol. 21, 1981, pp. 31-36

[Lam83] Lamport, L.: What good is temporal logic?, in *Proceedings of the IFIP 9th World Computer Congress*, 1983, pp. 657-668

[Lam85] Lamport, L.: An axiomatic semantics of concurrent programming languages, in Apt, K. R. (ed.): *Logics and Models of Concurrent Systems*, NATO ASI Series, vol. F13, Springer-Verlag, 1985, pp. 77-122

[LL90] Lamport, L. & Lynch, N.: Distributed computing: Models and methods, in van Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, Elsevier, 1990, pp. 1157-1199

[LP85] Lichtenstein, O. & Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification, in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985, pp. 97-107

[LPZ85] Lichtenstein, O. & Pnueli, A. & Zuck, L.: The glory of the past, in *Logics of Programs, Proceedings*, LNCS, vol. 193, Springer-Verlag, 1985, pp. 196-218

[LT89] Lynch, N. A. & Tuttle, M. R.: An introduction to input/output automata, in *CWI Quarterly*, vol. 3, no. 2, 1989, pp. 219-246

[MV89] Madelaine, A. & Vergamini, D.: AUTO: a verification tool for distributed systems using reduction of finite automata networks, in *Formal Description Techniques II, FORTE'89, Proceedings*, North-Holland, 1990, pp. 61-66

[MP81] Manna, Z. & Pnueli, A.: Verification of concurrent programs: the temporal framework, in Boyer, R. S. & Moore, J. S. (eds.): *Correctness Problem in Computer Science*, Academic Press, 1981, pp. 215-273

[MP90] Manna, Z. & Pnueli, A.: A hierarchy of temporal properties, in *Proceedings of the 9th ACM Symposium on the Principles of Distributed Computing*, 1990, pp. 377-408

[MP91] Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, vol. 1, Specification*, Springer-Verlag, 1991

[Mil89] Milner, A. J. R. G.: *Communication and Concurrency*, Prentice Hall, 1989

[NDGO86] Nguyen, V. & Demers, A. & Gries, D. & Owicki, S.: A model and temporal proof system for networks of processes, in *Distributed Computing*, vol. 1, 1986, pp. 7-25

[Pnu77] Pnueli, A.: The temporal logic of programs, in *IEEE 18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46-57

[Pnu85] Pnueli, A.: In transition from global to modular temporal reasoning about programs, in Apt, K. R. (ed.): *Logics and Models of Concurrent Systems*, NATO ASI Series, vol. F13, Springer-Verlag, 1985, pp. 123-146

[Pnu86] Pnueli, A.: Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends, in *Current Trends in Concurrency*, LNCS, vol. 224, Springer-Verlag, 1986, pp. 510-584

[RU71] Rescher, N. & Urquhart, A.: *Temporal Logic*, Library of Exact Philosophy, vol. 3, Springer-Verlag, 1971

[RRSV87] Richier, J. L. & Rodriguez, C. & Sifakis, J. & Voiron, J.: Verification in Xesar of the sliding window protocol, in *Protocol Specification, Testing and Verification VII*, North-Holland, 1987, pp. 235-248

[Sab88] Sabnani, K.: An algorithmic technique for protocol verification, in *IEEE Transactions on Communications*, vol. 36, no. 8, 1988, pp. 924-931

[SLU89] Sabnani, K. K. & Lapone, A. M. & Uyar, M. Ü.: An algorithmic procedure for checking safety properties of protocols, in *IEEE Transactions on Communications*, vol. 37, no. 9, 1989, pp. 940-948

[Sav70] Savitch, W. J.: Relationships between nondeterministic and deterministic tape complexities, in *Journal of Computer and System Sciences*, vol. 4, 1970, pp. 177-192

[SL83] Shankar, A. U. & Lam, S. S.: An HDLC protocol specification and its verification using image protocols, in *ACM Transactions on Computer Systems*, vol. 1, no. 4, 1983, pp. 331-368

[SL92] Shankar, A. U. & Lam, S. S.: A stepwise refinement heuristic for protocol construction, in *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 3, 1992, pp. 417-461

[SCSB92] Shiple, T. R. & Chiodo, M. & Sangiovanni-Vincentelli, A. L. & Brayton, R. K.: Automatic reduction in CTL compositional model checking, in *Computer Aided Verification, Fourth International Workshop, CAV'92, Proceedings*, LNCS, vol. 663, Springer-Verlag, 1993, pp. 234-247

[SC85] Sistla, A. P. & Clarke, E. M.: The complexity of propositional linear temporal logics, in *Journal of the ACM*, vol. 32, no. 3, 1985, pp. 733-749

[vdS95] van de Snepscheut, J. L. A.: The sliding-window protocol revisited, in *Formal Aspects of Computing*, vol. 7, 1995, pp. 3-17

[Ste76] Stenning, N. V.: A data transfer protocol, in *Computer Networks*, vol. 11, 1976, pp. 99-110

[Sti85] Stirling, C.: A complete compositional modal proof system for a subset of CCS, in *Automata, Languages and Programming, 12th Colloquium*, LNCS, vol. 194, Springer-Verlag, 1985, pp. 475-486

[Sti92] Stirling, C.: Modal and temporal logics, in Abramsky, S. & Gabbay, D. & Maibaum, T. (eds.): *Handbook of Logic in Computer Science*, Oxford University Press, Oxford, UK, 1992, pp. 477-563

[SW91] Stirling, C. & Walker, D.: Local model checking in the modal mu-calculus, in *Theoretical Computer Science*, vol. 89, 1991, pp. 161-177

[SM73] Stockmeyer, L. & Meyer, A.R.: Word problems requiring exponential time, in *Proceedings of the Fifth Annual ACM Symposium on the Theory of Computing*, 1973, pp. 1-9

[Tho90] Thomas, W.: Automata on infinite objects, in van Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, Elsevier, 1990, pp. 133-191

[Val90b] Valmari, A.: Stubborn sets for reduced state space generation, in *Advances in Petri Nets 1990*, LNCS, vol. 483, Springer-Verlag, 1991, pp. 491-515

[Val92] Valmari, A.: A stubborn attack on state explosion, in *Formal Methods in System Design*, vol. 1, 1992, pp. 297-322

[Val93] Valmari, A.: Compositional state space generation, in *Advances in Petri Nets 1993*, LNCS, vol. 674, Springer-Verlag, 1993, pp. 427-457

[Val94] Valmari, A.: State of the art report: stubborn sets, in *Petri Net Newsletter*, 46, 1994, pp. 6-14

[Val95] Valmari, A.: Failure-based equivalences are faster than many believe, in *Structures in Concurrency Theory: Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), 1995,* Workshops in Computing Series, Springer-Verlag, 1995, pp. 326-340

[VKCL93] Valmari, A. & Kemppainen, J. & Clegg, M. & Levanto, M.: Putting advanced reachability analysis techniques together: the "ARA" tool, in *FME'93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe*, LNCS, vol. 670, Springer-Verlag, 1993, pp. 597-616

[VT91] Valmari, A. & Tienari, M.: An improved failures equivalence for finite-state systems with a reduction algorithm, in *Protocol Specification, Testing and Verification XI*, North-Holland, 1991, pp. 3-18

[VT95] Valmari, A. & Tienari, M.: Compositional failure-based semantic models for Basic LOTOS, in *Formal Aspects of Computing*, vol. 7, 1995, pp. 440-468

[Wol83] Wolper, P.: Temporal logic can be more expressive, in *Information and Control*, vol. 56, 1983, pp. 72-99

[Wol86] Wolper, P.: Expressing interesting properties of programs in propositional temporal logic, in *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 184-193

[WG93] Wolper, P. & Godefroid, P.: Partial-order methods for temporal verification, in *CONCUR'93, 4th International Conference on Concurrency Theory, Proceedings*, LNCS, vol. 715, Springer-Verlag, 1993, pp. 233-246

Appendix A

Data independence and SW-protocol

A.1 Data independence

Data independence is a syntactic property of a system that makes it possible to verify the requirements of the system just for some particular kinds of execution sequences, and still be able to conclude that the system works correctly with all possible execution sequences. For a full discussion of the topic, please see [Wol86, Sab88]. Here we shall just outline the argument for the case of the SW-protocol.

Intuitively, data-independence means that a system does not pay attention to what the values of some data items processed are, but treats them as unanalysed entities. In particular, such a system does not make decisions between alternative courses of execution based on the values of such data items. To be slightly more formal, a system is said to be data-independent with respect to a set of data-variables $\{d_1, \ldots, d_n\}$, if any operation such a variable is subjected to is one of the following three:

- receiving its value from an external source
- transmitting its value to an external target
- copying its value from another data variable d_i

In particular, no constant values are assigned to data variables, no comparisons are done on their values and no decisions on what course of action the system is going to follow are based on the values of these variables.

Suppose that e is an execution sequence of a system that is data-independent with respect to $\{d_1, \ldots, d_n\}$, and that v is a data item that is at some point received from an external source to some data variable. If we change v to v' we still get an execution sequence e' of the system. The only difference between e and e' is that at some points the value of a data variable d_i that was v in e will be v' in e', and similarly that at some points a value transmitted to an external target will be v' in e' instead of v in e. Alternatively, we can take e, change globally all occurrences of v to v' in $\{d_1, \ldots, d_n\}$ and the related receptions and transmissions in it, and still get an execution sequence e' of the system. Now the only difference between e and e' is that if the value of a data variable d_i at some point is v in e, it is v' in e', and similarly with the receptions and transmissions.

The analysis here is slightly different from that of Wolper and Sabnani [Wol86, Sab88].

In their work it is shown that for safety properties it is enough to consider data sequences of the type $0^* \cdot 1 \cdot 0^* \cdot 2 \cdot 0^\omega$, whereas here we consider sequences of the type $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$. We believe that the current formulation allows for more efficient verification. For liveness properties, Wolper and Sabnani consider only sequences of the type 0^ω. Here we consider sequences of the type $0^\omega \cup 0^* \cdot 1$. The difference is that the formulation here covers also liveness for finite sequences of data, whereas that of Wolper and Sabnani considers only infinite sequences.

A.2 Safety

It is easy to see that the SW-protocol is data-independent with respect to the data-table variables d in the sender and t in the receiver process, and the data items read from the source and transmitted to the target.

Let 0, 1 and 2 be three distinct data items, and let x^*, x^ω, $\cup$, $\cdot$ and ϵ have their standard meaning in building ω-regular expressions. Furthermore, let $x^\infty = x^* \cup x^\omega$.

Let us apply the principle of data-independence to make the safety requirement SAFE easier to verify.

SAFE The data delivered to the target should be correct. More exactly, take any moment in an execution sequence of the protocol, and let s_r denote the sequence of data items that the sender has received from the source, and s_t the sequence of data items transmitted to the target before the given moment. Then s_t should be a prefix of s_r.

We shall show that the protocol fulfils the requirement SAFE iff it fulfils the requirement SAFE' below:

SAFE' Take any moment in an execution sequence of the protocol and let s_r denote the sequence of data items that the sender has received from the source, and s_t the sequence of data items transmitted to the target before the given moment. If s_r belongs to the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, then s_t belongs to the same language.

If the protocol satisfies SAFE, it clearly satisfies SAFE', since the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$ is closed under taking prefixes.

Suppose then that the protocol does not satisfy SAFE, and take an execution sequence e leading to a situation violating SAFE. We shall show that there is an execution e' leading to a situation violating SAFE'. We can take advantage of the data-independence and replace individual data items received from the source along e so that they are all distinct and e still violates SAFE. This means that without loss of generality we may assume that all the data items received from the source along e are distinct. Write $a_1 a_2 \ldots$ for the sequence of data items received from the data source along e. If this sequence is finite, let n_a denote its length. Similarly, write $b_1 b_2 \ldots$ for the sequence of data items transmitted to the data target along e, and, if it is finite, let n_b denote its length.

Since e violates SAFE, $b_1 b_2 \ldots$ is not a prefix of $a_1 a_2 \ldots$. This means that there is some $1 \le n \le n_b$ such that either $a_n \ne b_n$ or a_n does not exist, i.e. $n_a < n$. Let us take the smallest such n, in which case we know that $a_i = b_i$ for all $1 \le i < n$. Now there are two possibilities: either there is an $1 \leq n' < n$ such that $a_{n'} = b_n$, or then there is not. Let us handle these two cases separately.

Suppose first that there is some $1 \leq n' < n$ such that $a_{n'} = b_n$. Make the following substitutions to e and denote the result by e'

- For all $1 \leq i < n'$, replace globally all occurrences of a_i in data variables and related receptions and transmissions by the data item 0
- Replace all occurrences of $a_{n'}$ by 1:s in the same way.
- Replace all occurrences of all other values in data variables and related receptions and transmissions by 2:s.

Due to data-independence, e' is an execution sequence of the protocol. Let $a'_1 a'_2 \ldots$ denote the sequence of data items received from the source and $b'_1 b'_2 \ldots$ those transmitted to the target along e'. Notice that on the basis of the definition of e', $a'_1 = \ldots = a'_{n'-1} = 0$, $a'_{n'} = 1$ and $a_{n'+1} = \ldots = 2$, i.e. $a'_1 a'_2 \ldots$ belongs to the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$. However, since $b_{n'} = a_{n'} = b_n$, we have $b'_{n'} = b'_n = a'_{n'} = 1$. But as $n' \neq n$, the sequence $b'_1 \ldots b'_{n'} \ldots b'_n \ldots = 0 \ldots 1 \ldots 1 \ldots$ cannot belong to $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$. In other words, e' violates SAFE'.

Suppose then that there is no $1 \leq n' < n$ such that $a_{n'} = b_n$. This time make the following substitutions to e and denote the result by e'

- For all $1 \leq i < n$, replace globally all occurrences of a_i in data variables and related receptions and transmissions by the data item 0
- Replace similarly all occurrences of a_n by 1, if a_n exists, i.e. if $n \leq n_a$
- Replace all occurrences of all other values in data variables and related receptions and transmissions by 2:s.

Again, due to data-independence, e' is an execution sequence of the protocol. Let $a'_1 a'_2 \ldots$ denote the sequence of data items received from the source and $b'_1 b'_2 \ldots$ those transmitted to the target along e'. On the basis of the definition of e', $a'_1 = \ldots = a'_{n-1} = 0$, $a'_n = 1$ and $a_{n+1} = \ldots = 2$, i.e. $a'_1 a'_2 \ldots$ belongs to the language $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$.

As by assumption b_n is different from all of $a_1, \ldots, a_n$, by the definition of e', b'_n must be 2. However, for all $1 \leq i < n$, we have $b'_i = a'_i = 0$, since $b_i = a_i$. But now the sequence $b'_1 \ldots b'_{n-1} b'_n \ldots = 0 \ldots 02 \ldots$ cannot belong to $0^\infty \cdot (1 \cdot 2^\infty \cup \epsilon)$, meaning that e' violates SAFE' this time as well.

A.3 Liveness

Let us suppose that the protocol meets the safety requirements SAFE and SAFE', and use the principle of data-independence to simplify the liveness requirements. Let us show first that, assuming that the protocol fulfils the safety property SAFE, if fulfils the property LIVE-1 iff it fulfils LIVE-1', where the properties LIVE-1 and LIVE-1' are:

LIVE-1 Take any execution sequence of the protocol such that n data items are read from the data source along it, and the execution is complete under the assumption that no more data items are available for reading from the source. Then n data items should be delivered to the target along the execution sequence.

LIVE-1' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it belongs to the language $0^* \cdot 1$, and the execution is complete under the assumption that no more data items are available for reading from the source. Then the data item 1 should be delivered to the target along the execution sequence.

Suppose that the protocol fulfils LIVE-1. Take an execution sequence e of the protocol such that the sequence of data items read from the data source along it belongs to the language $0^* \cdot 1$, i.e. the sequence is $0^n \cdot 1$ for some $n \in \mathbb{N}$, and e is complete under the assumption that no more data items are available for reading from the source. By the property LIVE-1, $n + 1$ data items are delivered to the target along e. By the property SAFE, the only possible sequence of data items of length $n + 1$ that can be delivered to the target is $0^n \cdot 1$. Therefore the data item 1 is delivered to the target along e, and the property LIVE-1' holds.

Suppose then that the protocol fails LIVE-1. Then there is an execution sequence e of the protocol such that n data items $a_1 \dots a_n$ are read from the data source along e, the execution e is complete under the assumption that no more data items are available for reading from the source, but fewer than n data items are delivered to the target along e. As with safety-properties, due to data-independence we can assume without loss of generality that all the data items $a_1 \dots a_n$ are distinct. Let us make the following substitutions to e and denote the result by e':

- Replace globally all occurrences of a_n in data variables and related receptions and transmissions by the data item 1
- Replace all occurrences of all other values in data variables and related receptions and transmissions by 0:s.

Due to data-independence, e' is an execution sequence of the protocol. Let $a'_1 \dots a'_n$ denote the sequence of data items received from the source along e'. On the basis of the definition of e', $a'_1 = \dots = a'_{n-1} = 0$ and $a'_n = 1$, and the sequence of data items read from the data source along e' is $0^{n-1} \cdot 1$, which belongs to the language $0^* \cdot 1$. Since e is complete under the assumption that no more data items are available for reading from the source, so is e'. Since fewer than n data items are delivered to the target along e, the same holds for e'. By the property SAFE, this means that the sequence of data items delivered to the target along e' must be a proper prefix of $0^{n-1} \cdot 1$. Consequently, the data item 1 is not delivered to the target along e', and the property LIVE-1' fails to hold.

Let us show then that, assuming that the protocol fulfils the safety property SAFE, if fulfils the property LIVE-2 iff it fulfils LIVE-2', where the properties LIVE-2 and LIVE-2' are:

LIVE-2 Take any execution sequence of the protocol such that infinitely many data items are read from the data source along it. Then infinitely many data items should be delivered to the target along the execution sequence.

LIVE-2' Take any execution sequence of the protocol such that the sequence of data items read from the data source along it is 0^ω. Then the data item 0 should be delivered infinitely often to the target along the execution sequence.

Suppose that the protocol fulfils LIVE-2. Take an execution sequence of the protocol such that the sequence of data items read from the data source along it is 0^ω. By the property LIVE-2, infinitely many data items are delivered to the target along the execution sequence. By the property SAFE, the only possible infinite sequence of data items that can be delivered to the target is 0^ω. Therefore, the data item 0 is delivered to the target infinitely often along the execution sequence and the property LIVE-2' holds.

Suppose then that the protocol fails LIVE-2. Then there is an execution sequence e of the protocol such that infinitely many data items $a_1 a_2 \ldots$ are read from the data source along e (implying that e is infinite), but only finitely many data items are delivered to the target along e. Let us make the following substitutions to e and denote the result by e':

- Replace globally all occurrences of all values in data variables and related receptions and transmissions by 0:s.

Due to data-independence, e' is an execution sequence of the protocol. Let $a'_1 a'_2 \ldots$ denote the sequence of data items received from the source along e'. Since $a'_1 = a'_2 = \ldots = 0$, the sequence of data items read from the source along e' is 0^ω. Since only finitely many data items are delivered to the target along e, the same holds for e'. Therefore, the data item 0 cannot be delivered to the target infinitely often along e', and the property LIVE-2' fails.

Finally, let us show that the protocol fulfils the property LIVE-3 iff it fulfils LIVE-3', where the properties LIVE-3 and LIVE-3' are:

LIVE-3 The sender should accept data from the data source. More exactly, for any moment in a complete execution sequence of the protocol, there should be a future moment when the sender reads a data item from the data source, unless no more data is available for reading from the data source.

LIVE-3' Take any moment in a complete execution sequence of the protocol such that the sequence of data items read from the data source along the execution belongs to the language 0^*. Then there should be a future moment when the sender reads a data item from the data source, unless no more data is available for reading from the data source.

Since the property LIVE-3' is a restricted case of the property LIVE-3, it is clear that if the protocol fulfils LIVE-3, it also fulfils LIVE-3'.

Suppose then that the protocol fails LIVE-3. Then there is a complete execution sequence e of the protocol and a moment in e such that no more data items are read from the data source after that moment, despite data being available. Let $a_1 \ldots a_n$ be the sequence of data items read from the data source along e. Let us make the following substitutions to e and denote the result by e':

- Replace globally all occurrences of all values in data variables and related receptions and transmissions by 0:s.

Due to data-independence, e' is a complete execution sequence of the protocol. Let $a'_1 \ldots a'_n$ denote the sequence of data items received from the source along e'. Since

$a'_1 = \ldots = a'_n = 0$, the sequence of data items read from the source along e' belongs to the language 0^*. Since no more data is read from the data source along e after the given moment despite data being available, the same holds for e'. Consequently, e' violates the property LIVE-3'.

Appendix B

Model-checking liveness properties

As no linear temporal logic model-checker was available to us during this work, a particular, admittedly ad hoc solution was devised that allowed us to use the ARA toolset for model-checking the liveness property of the SW-protocol. Without going too deeply into the technical details, the intuition behind this solution is as follows.

To reduce the size of the altss to be dealt with, the NDFD reduction algorithm was run on the alts $\overline{DP}'$, yielding a smaller alts $\overline{DP}''$ such that $\overline{DP}' \overset{\text{ndfd}}{\approx} \overline{DP}''$. This was used as the basis of the further analysis.

We start by transforming the fairness assumption ϕ_{fair} into a more convenient form. Let us define the following syntactic abbreviations

$$\begin{aligned}
\mathit{sci}{*} \quad &\equiv_{df} \quad \bigvee_{0 \leq j \leq 2} \mathit{scij} \\
\mathit{cri}{*} \quad &\equiv_{df} \quad \bigvee_{0 \leq j \leq 2} \mathit{crij}
\end{aligned}$$

We can now transform ϕ_{fair} into a disjunctive normal form by straightforward propositional reasoning.

$$\begin{aligned}
\phi_{fair} \quad \equiv_{df} \quad & \bigwedge_{0 \leq i \leq w} (\Box\Diamond \mathit{sci}{*} \Rightarrow \Box\Diamond \mathit{cri}{*}) \ \wedge \ \bigwedge_{0 \leq i \leq w} (\Box\Diamond \mathit{rai} \Rightarrow \Box\Diamond \mathit{asi}) \\
\Leftrightarrow \quad & \bigwedge_{0 \leq i \leq w} (\Diamond\Box\neg \mathit{sci}{*} \vee \Box\Diamond \mathit{cri}{*}) \ \wedge \ \bigwedge_{0 \leq i \leq w} (\Diamond\Box\neg \mathit{rai} \vee \Box\Diamond \mathit{asi}) \\
\Leftrightarrow \quad & \left(\bigvee_{I \subseteq \{0,\ldots,w\}} (\bigwedge_{i \in I} \Diamond\Box\neg \mathit{sci}{*}) \ \wedge \ (\bigwedge_{i \in \{0,\ldots,w\} \setminus I} \Box\Diamond \mathit{cri}{*}) \right) \quad \wedge \\
& \left(\bigvee_{J \subseteq \{0,\ldots,w\}} (\bigwedge_{i \in J} \Diamond\Box\neg \mathit{rai}) \ \wedge \ (\bigwedge_{i \in \{0,\ldots,w\} \setminus J} \Box\Diamond \mathit{asi}) \right) \\
\Leftrightarrow \quad & \bigvee_{I,J \subseteq \{0,\ldots,w\}} \Bigg((\bigwedge_{i \in I} \Diamond\Box\neg \mathit{sci}{*}) \ \wedge \ (\bigwedge_{i \in \{0,\ldots,w\} \setminus I} \Box\Diamond \mathit{cri}{*}) \wedge \\
& \qquad (\bigwedge_{i \in J} \Diamond\Box\neg \mathit{rai}) \ \wedge \ (\bigwedge_{i \in \{0,\ldots,w\} \setminus J} \Box\Diamond \mathit{asi}) \Bigg)
\end{aligned}$$

Intuitively, this simply says that a path σ is fair iff there are sets $I, J \subseteq \{0, \ldots, w\}$ such that

1 for all $i \in I$, $sci*$ occurs only finitely many times in σ

2 for all $i \in \{0, \ldots, w\}, i \notin I$, $cri*$ occurs infinitely many times in σ

3 for all $i \in J$, rai occurs only finitely many times in σ

4 for all $i \in \{0, \ldots, w\}, i \notin J$, asi occurs infinitely many times in σ

Notice then that $\overline{DP}''$ fails the liveness requirement, i.e.

$$\overline{DP}'' \;\not\models\; \phi_{live}$$

iff there is a maximal path σ of $\overline{DP}''$ such that

$$\begin{array}{lcll} \sigma & \models & \neg\phi_{live} & \text{i.e.} \\ \sigma & \models & \neg(\phi_{fair} \Rightarrow (\Box\Diamond\, rt0 \,\vee\, \Diamond\, rt1)) & \text{i.e.} \\ \sigma & \models & \phi_{fair} \wedge \Diamond\Box\neg\, rt0 \;\wedge\; \Box\neg\, rt1 & \end{array}$$

A path σ fulfils the two last requirements iff

5 $rt0$ occurs only finitely many times in σ, and

6 $rt1$ does not occur in σ at all.

Remembering the observations above, this means that $\overline{DP}''$ fails the liveness requirement iff it has a maximal path σ and sets $I, J \subseteq \{0, \ldots, w\}$ such that the conditions 1–6 are fulfilled.

Given a pair of sets $I, J \subseteq \{0, \ldots, w\}$, figuring out whether $\overline{DP}''$ has a maximal path σ fulfilling 1–6 for these I and J could be done by a fairly straightforward graph algorithm. However, what we did was to construct a tester alts $T_{I,J}$ with a special 'success' transition $succ$, such that $\overline{DP}''$ has a maximal path σ fulfilling 1–6 iff

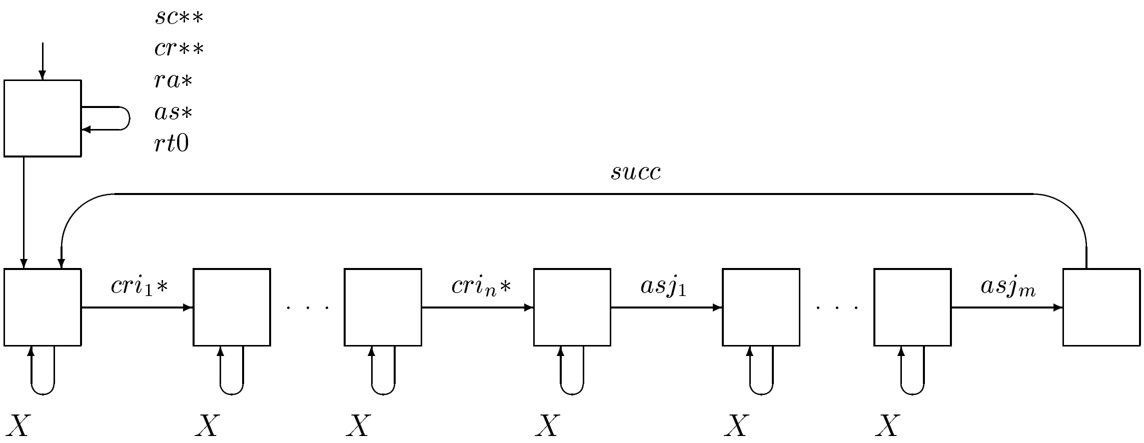
$$\textbf{hide } \Sigma' \textbf{ in } (\overline{DP}'' \,|[\Sigma']|\, T_{I,J}) \quad \text{where } \Sigma' = \Sigma \setminus \{succ\}$$

has a maximal path σ' where $succ$ occurs infinitely often. This last condition can be checked by a simple depth-first search. The tester alts $T_{I,J}$ is presented in figure B.1. Notice that it has never more than $2w + 4$ nodes.

Summarising this, to verify that $\overline{DP}'' \models \phi_{live}$, we checked for every possible $I, J \subseteq \{0, \ldots, w\}$ that

$$\textbf{hide } \Sigma' \textbf{ in } (\overline{DP}'' \,|[\Sigma']|\, T_{I,J})$$

does not have a maximal path σ' where $succ$ would occur infinitely often. Notice that for a given w, there are 2^{2w+2} different pairs I, J to be checked. The table in figure B.2 lists the sizes of some altss used in this task.



X denotes the set of labels $\{sci* \mid i \in \bar{I}\} \cup \{raj \mid j \in \bar{J}\} \cup \{cr{*}{*}\} \cup \{as*\}$

$$\bar{I} = \{i_1, \ldots, i_n\} = \{0, \ldots, w\} \setminus I$$
$$\bar{J} = \{j_1, \ldots, j_m\} = \{0, \ldots, w\} \setminus J$$

Figure B.1: SW-protocol: Tester alts $T_{I,J}$

Parameters			Reachable nodes		
w	tw	rw	$\overline{DP}'$	$\overline{DP}''$	max $\overline{DP}'' \,\lVert[\Sigma']\rVert\, T_{I,J}$
1	1	1	61	28	53
2	1	1	85	41	75
	2	1	607	214	704
3	2	2	915	458	996
	3	1	2375	1182	2817
4	2	2	1131	572	1177
	3	2	3814	1882	4418
	4	1	6613	3232	8532
5	2	2	1347	1009	2276
	3	3	5179	2420	5638
	4	2	11209	5486	13927
	5	1	15223	7400	21203
6	2	2	1563	800	1659
	3	3	6018	2823	6320
	4	3	16637	7954	20034
7	2	2	1779	914	1900
	3	3	6857	3226	7052
	4	4	21379	9770	24417†
† : Not all testers examined					

Figure B.2: SW-protocol: Sizes of altss for verifying $\overline{DP}' \models \phi_{live}$