DISS. ETH Nr. 15578

CIP Model-Checking

A dissertation submiited to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY

for the degree of

Doctor of Technical Sciences

presented by

Andreas Hubert Moglestue
dipl. El.-Ing ETH Zürich

born 06.09.1971

citizen of
the United Kingdom

accepted on the recommendation of

Prof. Dr. Lothar Thiele
Prof. Dr. Albert Kündig
Prof. Dr. Armin Biere

2004

Table of Contents

Zusammenfassung

Verifikation ist ein wichtiger Bestandteil im Entwicklungszyklus jedes Produktes. Um Software vollständig und wiederholbar verifizieren zu können, müssen klar definierte Methoden angewandt werden. *Model Checking* ist eine derartige Methodik. Ein *Model Checker* kann eine passend eingegebene. Systembeschreibung analysieren und Aussagen über das System machen. Einige Beispiele sind Signalsysteme bei der Bahn oder Überwachungssysteme bei der Luftfahrt, bei denen es unerlässlich ist, dass sich überschneidende Trajektorien nie gleichzeitig freigegeben werden können. Oder, im kleineren Maßstab, muss gezeigt werden, dass ein Kommunikationsprotokoll frei von Deadlocks ist. Formell definiert wird eine solche Anforderung eine *Eigenschaft* (*property*) genannt.

Es existieren bereits eine ganze Reihe von Model Checkern und einige davon werden in dieser Arbeit kurz besprochen. Allerdings weicht der Model Checker, der als Thema der vorliegenden Dissertation vorgestellt wird, von diesen ab, indem er eng verknüpft ist mit einem bereits existierenden Entwicklungsrahmen, *CIP*. CIP (Communicating Interacting Processes) bietet bereits deutliche Verbesserungen bei der Entwicklung von eingebetteter Software durch seine Methodik, Struktur, Unterhaltsfreundlichkeit und bei der Erstellung von Dokumentationen. Der Model Checker, der hier vorgestellt wird, benützt diese Strukturen vorteilhaft, indem sie als Basis für seine Verifikationstechnik dienen.

Ein Model Checker interpretiert ein System als eine Menge von *Zuständen* (*states*), welche durch *Übergänge* (*transitions*) untereinander verbunden werden. Der Model Chekker analysiert die so dargestellte Struktur und sucht *Pfade* (*paths*) (durch Übergänge miteinander verbundene Reihen von Zuständen) welche die zu beweisende Eigenschaft widerlegen. Wenn ein derartiges Gegenbeispiel nicht existiert, ist die Eigenschaft bewiesen.

Der offensichtlichste Weg, die nicht-existenz solcher Gegenbeispiele zu beweisen, ist ein systematisches Durchprobieren aller Pfade. In der Praxis ist die Anzahl der möglichen Zustände allerdings derart groß, das der benötigte Speicherplatz und Zeitbedarf ein solches Vorgehen ausschließen. Dieses Problem wird *State Explosion* genannt. In echten Systemen, die mit CIP implementiert werden, wirken die CIP-typischen Interaktionen diesem Problem entgegen. CIP verhindert gewisse Übergänge und verhindert damit die Erreichbarkeit vieler Zustände. Weitere Einsparungen werden ermöglicht durch die Wahl einer geeigneten Darstellung im Speicher.

Zusätzlich kann mit dem sogenannten *Partial Order Reduction* das State Explosion Problem wesentlich vermindert werden. Partial Order Reduction nützt Symmetrieeigenschaften des Systems und reduziert dadurch die Anzahl der Pfade, die betrachtet werden müssen, um zu einer sinnvollen Aussage zu kommen. Die Anzahl der erreichten Zustände und der Zeitbedarf nehmen dadurch ebenfalls ab. Diese Methode macht von dem Umstand Gebrauch, dass nur eine von allen möglichen Reihenfolgen betrachtet werden muß, wenn die Reihenfolge gewisser Übergänge beliebig ist.

Eine wichtige Quelle falscher Ergebnisse, die bei Model Checking auftreten können, sind Pfade, welche von den benötigten Übergängen zugelassen werden, jedoch kein tatsächlich mögliches Verhalten des zu modellierenden physikalischen Systems darstellen. Solche nicht zulässigen Pfade können verursacht werden von Übergangssequenzen, die sich unendlich wiederholen und dadurch die Ausführung anderer Übergänge verhindern, die im physikalischem System sicher zur Ausführung kommen würden. Ein solcher Pfad, welcher im Modell auftreten kann, aber im physikalischen System nicht, wird als *unfair* bezeichnet. Im Modell, fehlen die mechatronische Zusammenhänge die das Unterscheiden von zulässige und nicht-zulässige Pfade ermöglichen. Deshalb müssen vom Benutzer zusätzlich *Fairness Constraints* definiert werden, um in dieser Hinsicht das Modell zu vervollständigen. Dieses ist die einzige Ergänzung des Modells, das der Model Checker dem Benutzer abverlangt.

Der wichtigste Beitrag in diesem Projekt liegt in der Bildung eines Modells für den Model Checker aus dem CIP Modell, und die Behandlung dieses Modells innerhalb des CIP Rahmens. Dieses wird erreicht durch eine Erweiterung der CIP Funktionalität, damit er von einem Anwender, der CIP ausreichend kennt, aber nur sehr geringe Model Checking Kenntnisse besitzt, verstanden und verwendet werden kann.

Diese These führt eine Notation ein zur formellen Beschreibung von CIP Modellen und ihren Komponenten. Das CIP Tool selber wird erweitert durch die Möglichkeit, Fairness Constraints zu definieren, die es dem Benutzer erlauben, das zu modellierende System für den Model Checker besser zu beschreiben, damit keine falschen Gegenbeispiele oder Ergebnisse geliefert werden.

Ein weiterer Beitrag dieses Projektes ist das Hinzufügen eines *Execution Testers* zum CIP Tool. Dieses Werkzeug erlaubt es dem Entwickler, neu definierte oder veränderte Strukturen einfach und rasch zu testen. Dieser Schritt war eine notwendige Grundlage für den Model Checker, bildet aber ebenfalls ein selbständiges Test- und Simulationswerkzeug.

Bei der Umsetzung des CIP Model Checkers wurde auf eine enge Integration mit dem CIP Tool Wert gelegt. Mit diesem wurden Fehler korrekt identifizieren und industrielle Systeme erfolgreich traversiert. Der CIP Model Checker ist eine wertvolle Erweiterung des CIP Tools und hilft CIP Anwendern bessere und sicherere Software zu schreiben.

Abstract

Verification is a central component of the development process of any product. To be able to verify embedded software in a complete, and reproducible way, clearly defined techniques must be applied. *Model Checking* is such a technique. A model checker can analyse an appropriately presented system description and make statements about that system. Some examples are a railway signalling system or an air traffic control system, where it is vital to be able to show that conflicting paths can never be allocated simultaneously. Or on a smaller and more mundane scale, it must be shown that communications protocols are free of deadlocks. When such a requirement is formalised it is called a *property*.

Many model checkers already exist and some are briefly discussed in this thesis. However, the model checker which is developed in this project differs in being closely integrated with a pre-existing development tool, *CIP*. CIP (Communicating Interacting Processes) in itself already offers considerable advances in embedded software development through its methodology, structure and ease of maintenance and documentation. The model checker presented in this thesis takes advantage of these structures and uses them as a basis for integrating a verification technique.

A model checker basically views the system as a set of *states* which are connected by *transitions*. Looking at the structure so represented, it searches for *paths* (sequences of states connected by transitions) which disprove the property to be verified. Failure to find such a counter-example in an exhaustive search of the system proves that the property *holds*.

The most obvious way of executing such a proof is by brute force exhaustive testing of all possible paths. In practice, however, the sheer number of possible states of the system is so great that the physical memory of normal computer systems would be insufficient to hold them and the time required would be prohibitive. This problem is known as the *state explosion* problem. In real systems implemented with CIP Tool this problem is reduced to an extent by the restrictive and interactive nature of CIP. CIP disallows certain transitions and so intrinsically excludes the reacheability of many states. Further savings are made possible by the choice of a suitable coding in memory.

A method known as *partial order reduction* further strongly reduces the state explosion problem. Partial order reduction makes use of symmetries in the system to decrease the number of paths that must be followed and so the number of states found and also the time required. This method makes use of the fact that when transitions are *interleaving* (their order of execution is irrelevant), then only one of all possible orders must be looked at.

An important source of false results in Model Checking is caused by infinite paths which are legal from the point of view of all states being connected by transitions, but do not represent real behaviours of the physical system being modelled. Such illegal paths can be caused by certain sequences of transitions being infinitely repeated to the exclusion of other transitions which certainly would be executed in the real system. Such a path which can occur in the model but not in the physical system is an *unfair* path. *Fairness constraints* can be defined to tell the model checker which transition sequences are unfair.

The principle contribution of this project lies in the extraction of the Model Checking model from the CIP model and the treatment of that model within the CIP framework. This can be achieved through an extension of the CIP functionality in a way which is easily understandable for a user with very basic Model Checking knowledge but adequate understanding of CIP.

This thesis also presents a notation for formally describing CIP models and their components. The CIP Tool itself is also extended through the addition of fairness constraints which allow the user to tell the model checker more about the behaviour of the real system so that no physically impossible counter-examples or claims are produced.

Another addition to the CIP Tool made in this thesis is the *execution tester*. This greatly facilitates testing of the model by the developer. As soon as a component is defined, the user can already test how this component responds to inputs. This was a necessary step towards the creation of a model checker but also provides a useful testing and simulation tool in its own right.

The CIP model checker has been implemented in close integration with CIP Tool. It has been shown to correctly identify constructed errors and successfully traverse real industrial systems. It should prove to be a valuable enhancement to CIP Tool, helping CIP users to create better and safer software.

Acknowledgements

Many thanks are due to Hansruedi Müller, CEO of CIP System AG for his support and especially for introducing me to the internal architecture of the CIP Tool and the fine art of SmallTalk programming.

Thanks are also due to Hugo Fierz, inventor of CIP and co-founder of CIP System AG whose contagious enthusiasm accompanied me for much of the realisation phase of the CIP model checker. He introduced me to the methodology and philosophy behind CIP but also kept me up to date on real life situations and problems and enabled me to meet several CIP users.

Special thanks is due to Prof. Albert Kündig for enabling this project to come about and for his continuous support, patience and numerous suggestions, especially concerning the presentation of this thesis.

I would also like to thank Prof. Armin Biere and Prof. Lothar Thiele for their support and suggestions.

Additional support was received from Johanna Schön in proofreading parts of this thesis.

Also worthy of recognition are the computer and infrastructure support teams of TIK and all staff for creating an inspiring and pleasant working environment.

I would also like to thank Armin Deiss for his friendship and support during our time together at the ETH.

1. Introduction

1.1. Purpose of this project

How many software developers don't test run their own code? Or for that matter, how many cooks don't taste their own food? How many writers don't check their own words and how many artists don't look at their own painting? Almost all creative processes are associated with some form of verification. This verification has a dual purpose. On the one hand, it provides a feedback during the phase of creation. Is there enough salt in the food? Does a subroutine perform the required task? On the other hand it serves as a final verification before the product is passed on to the customer. This project is concerned with supporting software developers in both of these phases.

Over the last 30 years or so, software development has probably made more significant bounds than any other domain. Not only have the complexity of the problems tackled and the solutions offered increased beyond the imagining of those who pioneered the first computers, but many new methods and schools of thought have arisen as to how software should be written and how it should act and react. The verification activities are probably the part to have made the least progress. Many software companies rely largely on their customers to report bugs. In most other industries such an approach would be unacceptable. In embedded software this is also the case. A bug could cause huge costs and setbacks if it causes a component to malfunction. The safety of humans may be at risk if a railway signalling or aviation control system malfunctions. Huge sums of money and effort are wasted if a satellite cannot fulfill its specifications. Clearly, intuitive testing and feedback alone cannot serve as a basis for quality guarantees.

One approach to the quality assurance problematic is the use of formal verification techniques and in particular Model Checking. Model Checking is the exhaustive execution of a model to check that it conforms to the specification. The first succesful application of this discipline was in hardware verification. It has since come to be used for software also.

The CIP-Tool was first developed to tackle the problem from the other end. It supports the software developer by enabling him to easily use a structured and disciplined approach to building and servicing software. This approach cuts down the risk of bugs caused by sloppiness or absence of transparency while forcing developers to respect a strictly structured approach based directly on the model being implemented. Whilst this method assures a close correlation between theoretical and implemented model (ideally identical), it cannot prevent bugs from occurring in the model itself.

This project aims to integrate a model checker in the CIP-Tool to do just that, and in so doing make a contribution to quality assurance techniques in software development.

Modern quality assurance techniques lay much value on defined standards and benchmarks for testing. Test documentation is increasingly becoming part of the formal acceptance procedure of new products and this is reflected in the stringency of methods and criteria. CIP-Tool itself makes a contribution in this direction by providing a structured and referrable development platform. The CIP model checker enhances this strength considerably by adding a clearly defined Model Checking tool to that developer platform with reproducible results.

1.2. What is CIP?

1.2.1. Brief history

CIP (Communicating Interacting Processes) was created as a research project at the Swiss Federal Institute of Technology (ETH) in the 1990ties by H. Fierz [10]. Later the development was continued by a spin-off company [9]. Today CIP-Tool is available as a commercial product and is successfully used by many companies as a development tool. Products realised with CIP include a hybrid automobile, a tamping machine for railway tracks, automatic door systems and compact disk manufacturing machines.

1.2.2. What is the purpose of CIP?

The purpose of CIP is to raise the quality and serviceability of embedded software by introducing an approach to problem solving which is considerably more structured than common development methods including UML[1]. CIP provides a development environment that supports the software engineer by allowing him to concentrate on his principal job, i.e, designing the system, by relieving him of repetitive onerous tasks such as hand coding every system transition and keeping track of the system status. The finished model is stored and represented as a graphically composed model rather than code. This makes it easier to modify and service.

The developer designs the system graphically by specifying and connecting components. Once the model is created in CIP-Tool, compiler ready code can be generated from it. If any modifications are necessary, these are made in the graphic CIP-Tool environment and the code is generated again. The generation of system documentation is equally supported by CIP-Tool.

CIP-Tool also allows multiple developers or teams to work on the same project. Modules can easily be imported and exported and interfaces can be defined permitting separate teams to develop components independently.

[1] This will be discussed in section 4.8.

1.2.3. What are the constraints of CIP?

Since it was first released, CIP-Tool has been continuously streamlined and its functionality extended. So it has been able to meet and adapt to the demands of it's users. In this manner, the integration of a Model Checking facility meets the demands of many users who are faced with increasingly complex systems which make it more and more difficult to be sure that the model is still correct.

Despite the growth in the functionality of the tool, the reader should bear in mind that CIP-Tool is designed and intended for embedded systems. While isolated applications may exist in other fields, the strength of the tool lies in embedded system development.

One constraint with particular bearing on this project is the model does not contain the information permitting executions which are possible in the physical system to be distinguished from those which are not. This will be discussed in greater detail in the sections on *fairness* (sections 3.7. and 5.7.) and an extension to the model will be introduced to overcome this constraint (section 5.7.). The lack of this feature in CIP is not surprising as the tool was never designed with Model Checking in mind, and this feature does not have any other purpose.

1.3. Brief introduction to Model Checking

1.3.1. What is Model Checking?

With the continuous increase in complexity of systems, testing alone is no longer sufficient to guarantee that the behaviour of the software conforms to the specification which the testing is intended to verify. The vast number of permutations of system states and inputs make a thorough test approach impossible. Additionally, programmers have a tendency to test the situations they had in mind when writing the software, whereas it is the unforeseen and unexpected which more often causes critical situations. To some extent the latter is relativised by the common practice of independent groups performing the tests. But knowledge of the application can equally lead to the same scenarios being tested as where written for by the programmers reducing the usefulness of the test.

Model Checking is the verification of a model to check that it conforms to the specification. Model Checking is not based on testing by subjecting the system to quasi-random sequences of inputs as the human tester does. It is based on a thorough approach which is able to make general statements about the system in all situations.

In 1981 E.M.Clarke and E.A.Emerson [11] defined Model Checking as follows:

Model Checking is an automated technique that given a finite state model of a system and a logical property, systematically checks whether this property holds for that model.

1.3.2. Target properties of Model Checking

Model checking allows the user to test a whole range of logical properties. The following are typical questions we cannot be completely sure of even after extensive testing, but could verify with a model checker.

a) In a railway junction, can two trains be in a critical section simultaneously?

b) Can we be sure that a lift cabin cannot move when the door is not closed[1]?

c) When the alarm button has been pressed, can a dangerous action still be begun?

d) If a resource is requested, is it possible that it will be granted?

e) When the alarm has been pressed, will all parts of a machine enter a safe state?

f) When a resource is requested, can we be sure that it will be allocated?

(a) and (b) are examples of the *invariance* property. This demands that the state of the system never (or always) conforms to certain rules.

(c) and (d) are examples of the *reacheability* property. This demands that if the state has conformed to a rule A, then it is possible that it can at once or later conform to rule B.

(e) and (f) are examples of the *eventually* property. This demands that if the state has conformed to a rule A, then it must at once or later conform to rule B. This property is also known as *request → acknowledge*, because every *request* must eventually be *acknowledged*.

More complex properties can be defined by mixing these basic properties. This can be done by combining expressions using boolean operators. For example:

- when an action A is requested it will eventually be performed unless the alarm button is pressed

can be reformulated as:

[1] The forulation ‘not open’ is chosen intentionally as it includes also transient cases in which the door is neither open nor closed.

- when an action A is requested without the alarm button being pressed, eventually A will be performed or the alarm button will be pressed.

The CIP model checker can check several properties simultaneously.

1.3.3. Limits of Model Checking

The demands on memory space of model checkers is huge and grows exponentially with every process added. We call this the *state-explosion problem*. Although several approaches exist for reducing the impact of state-explosion, some of which will be discussed in this project, state-explosion essentially limits the size and complexity of systems to which Model Checking can be applied.

Other limits of Model Checking are that most classical Model Checking techniques are limited to finite state machines. Most real systems also include some data processing parts which may affect the behaviour of the system. This subject will also be addressed in this thesis and some workarounds presented.

1.4. What can Model Checking do for CIP?

1.4.1. Model Checking and CIP: current possibilities

Many CIP users have been demanding the possibility of formal verification. This could be achieved by using an existing model checking environment. The CIP user could easily buy a model checking software or download a free one. Why then is a dedicated CIP model checker required?

In most model checkers the model has to be described in a so called meta-code or meta-language. The code of this is similar to a programming language. The user thus has to learn the meta-language and the handling of the model checker and then transfer the algorithms to be tested to the meta-language. This is not only time consuming. but also poses an important source of errors by the possibility of the compilable code not being completely equivalent to the verifiable code.

1.4.2. Model Checking from program code

One solution to this problem is automatic extraction of verifiable code from compilable code. This is not easy due to the vast expressive power of programming languages. However some progress has recently been made. For example, G. Holzmann announced that his model checker SPIN is to integrate the option of directly extracting meta-code from C-code [32][39].

So technically, it is possible to generate C-code with CIP-Tool and then feed that code to SPIN for verification. But this is not an optimal procedure. Not only is the C-code extraction still at an early stage of its development and unable to cope with the full syntactical strength of C, but there is a more systematic proble:

In the CIP model, the system is represented in a manner reflecting the system structure and interaction mechanisms, which are also advantageous to Model Checking. This useful structure is largely lost in the C-code generation in the same way that a well structured C program does not necessarily compile into well structured machine-code. Also, the C-code model is considerably more complex than the CIP model from which it is derived (just as the machine code model is more complex still), and so an attempt to perform Model Checking on this will require considerably more resources and require the ability to respond to a larger range of inputs. Even if the growth of model complexity is linear, the growth in resource requirements will be exponential (see *state explosion*, section 5.2.3.).

1.4.3. Generating meta-code from CIP-Tool

A related and tempting alternative would be to extend CIP-Tool to generate meta-code for a model checker just as it can generate compilable C or Java code. This code would then be fed to the model checker.

This approach is considerably more practical than that of passing through C-code. In fact it is well worth looking into as a future development.

The drawbacks are, that the available model checkers were designed for problems of a general nature and are far too expressive for CIP. CIP has strict *run to completion semantics* (this will be discussed in section 2.5.5.) and strict rules governing pulse casting and propagation. To restrict an off the shelf model checker to such behaviour would require considerable auxiliary structures.

A drawback for the user would be having to run two independent applications with a possible loss of correlation between what the model checker is doing and his own designs.

1.4.4. The integrated model checker

The approach that was finally chosen is that of an integrated model checker in CIP-Tool.

Data structures in CIP are well suited to model checking because all possible interactions are conatined in the model's definition. The goal of this thesis is to be able to perform model checking operations directly on the structures as they are represented in CIP, taking full advantage of these structures.

An additional advantage of this approach is that the user interface can be made to conform to that of CIP-Tool. The properties to be tested can be made easy to understand and easy for the user to define. Ideally, the user should be able to use the tool without acquiring any specific Model Checking knowledge.

1.5. Goals

1.5.1. What the CIP model checker should do

The CIP model checker should be fully integrated within CIP-Tool so that the user can access and use it with a series of mouse clicks. It should enable the user to verify the model he is constructing, both in finished form and on the fly during the design process. The principal properties whose verification it permits are those described in section 1.3.2. The meanings of these properties and their uses should be presented to the user in a comprehensible way so that a user who has working knowledge of the CIP-Tool and CIP methods but no specific Model Checking background is able to make productive use of the checker.

The CIP model checker will compare these properties with the modelled system and either confirm they hold or otherwise produce a counterclaime violating the property.

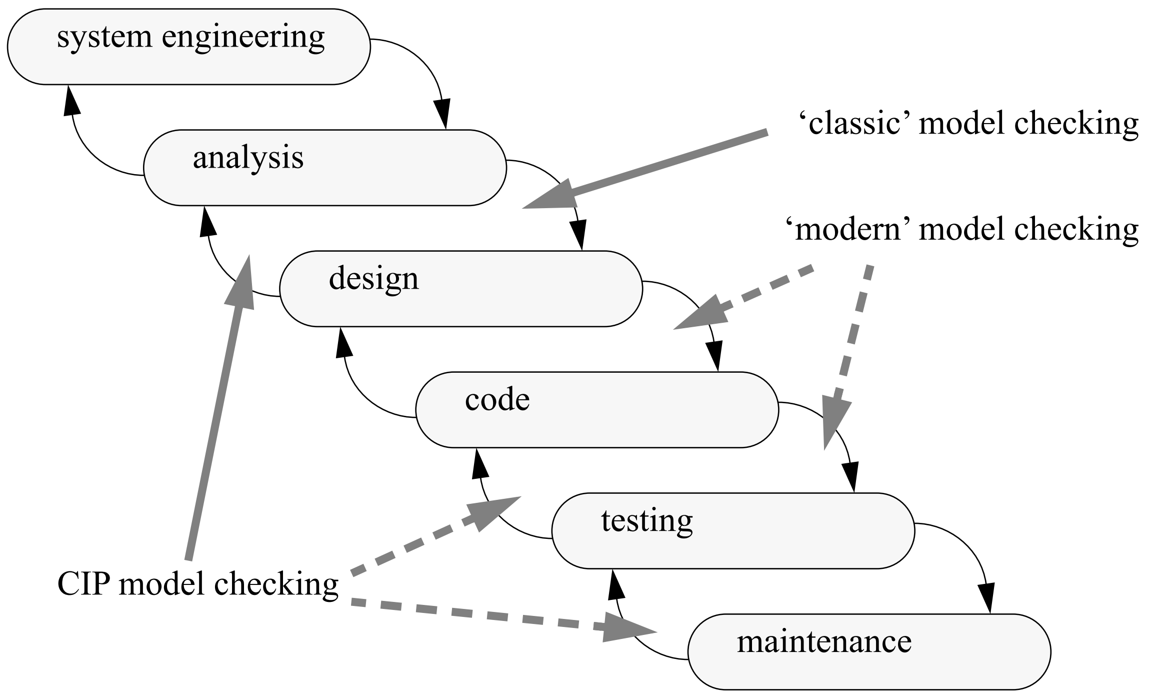
In this way, the CIP model checker can become an active and important part of the design cycle of products developed with CIP-Tool.

1.5.2. The CIP model checker and the design process

In figure 1.1. the classic “waterfall model” [17] of software development is shown. Classic model checkers are especially useful in the transition from analysis to design where they check the correctness of algorithms. Modern model checkers such as SPIN which are closer to the actual code can increasingly be used in the *design to code* and *code to test* phases. This requires of course that these model checkers can interpret and correctly translate the full expressive power of program code. At the present moment this is not necessarily the case.

The CIP model checker aims to be useful in the *analysis to design* transition. The manual part of the *design to code* phase is greatly reduced by CIP. The process is automated and risk of error greatly reduced. Hence Model Checking in this phase in the CIP environment can be considered superfluous. CIP also saves time on the *testing* and *maintenance* phases. If the model has been correctly checked the results are by implication also applicable to these phases. Of course this implication relies on the code being generated correctly by CIP Tool so that the code is fully equivalent to the design model. As CIP Tool has been in common commercial use for many years now, it is likely that all bugs in this respect have been identified, reported and corrected. A high degree of confidence can thus be placed in properties holding for the model also holding for the final product.

figure 1.1. System development “waterfall model”



1.5.3. The execution tester

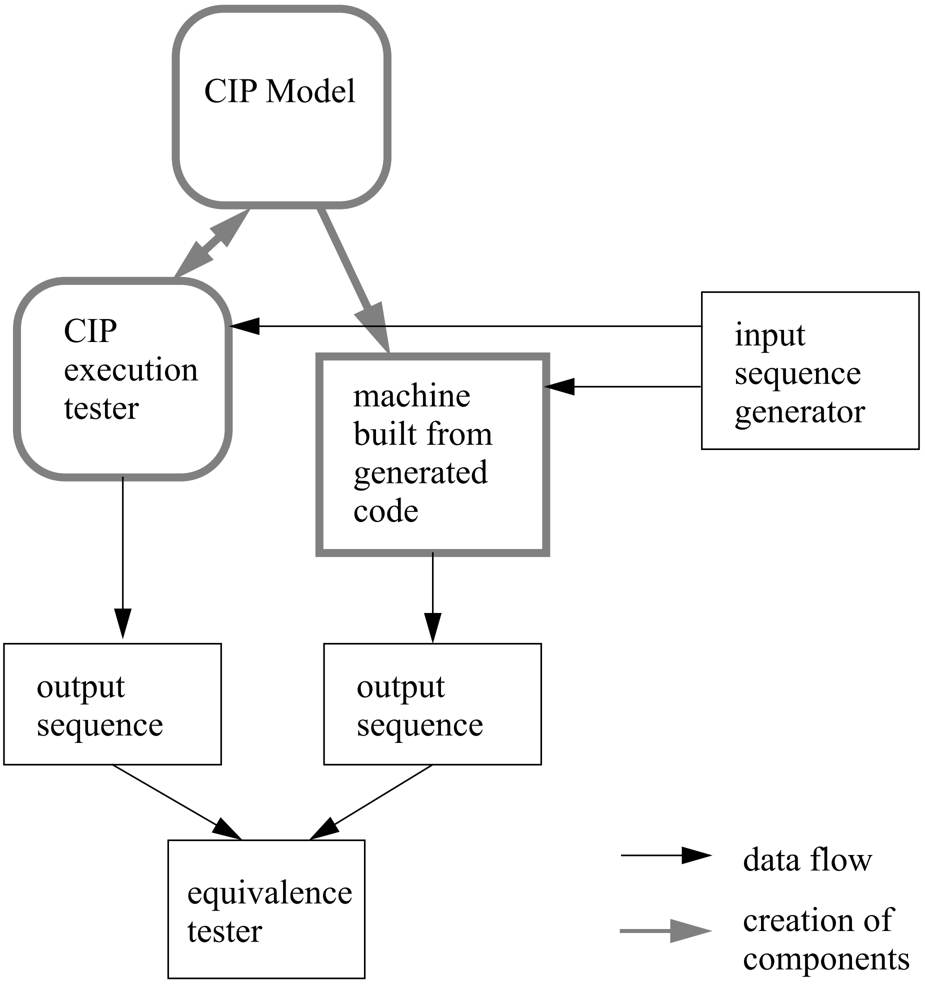
Previously, for testing purposes, C-code could be generated with CIP Tool. This could be compiled and linked to a CIP simulator permitting software testing of the C-code from the CIP environment. The model checker provides an alternative to this. The execution of models can be tested at model level thus saving the need to recompile and link for testing purposes after every change. This *execution tester* lies at the heart of the Model Checker, but is also a very useful tool in its own right. It provides an alternative to the code generator when functionaity tests are required. The CIP execution tester makes it feasible to make minor changes and then quickly retest the modified model as it stands.

Also, the C-code generator requires models to be complete to be able to generate code. Where parts of the specification are missing, the code generator cannot perform its task, even when the missing parts have no bearing on the test to be performed. The model checker is more tolerant and assumes the more general case where the specification is incomplete. However, as the complete model cannot always be extrapolated from the incomplete, no results from such a test excuses the programmer from re-running the model checker for the complete model.

One very valid question that can be asked of the execution tester, is whether it is really equivalent to the generated, compiled and linked code. All Model Checking results implicitly rely on the correctness of the execution tester. Behavioural differences can be attributed to errors in either the translation from CIP model to execution tester or any of the transitions between CIP-model and the execution of the compiled code on the target system.

To demonstrate the confidence which can be placed in this equivalence, the parallel operation of the execution tester and the compiled code would be desirable for a large number of (preferably randomly generated models) and a random event sequence applied to both machines in parallel. At present this is not possible because CIP-Tool lacks a command line interface permitting such a task to be automated. Such an extension is desirable for a future phase.

figure 1.2. Method for demonstrating confidence in equivalence of CIP execution tester and generated code



The manual testing of the model as facilitated by the execution tester is not made superfluous by Model Checking. Manual and intuitive testing remains an important feedback component in any development cycle. The value of a test case depends also on how it is handled. One structured approach to handling test cases is *Extreme Programming*, where test cases are defined before the model is designed and further test cases added based on experiences gained and problems identified during the modelling or programming phase. A command line interface would also greatly facilitate the exection of such test cases and hence the application of such structured testing approaches.

1.5.4. Goals of this project

This project aims to realise the CIP model checker as described above. In doing so optimal use will be made of the existing data structures. Where these need to be extended or modified this will be done in line with the general design philosophy of CIP-Tool.

1.6. Summary of results

1.6.1. General

The CIP model checker was successfully implemented and integrated with CIP-Tool and its function demonstrated both with constructed examples and genuine systems taken from industrial implementations.

The CIP model checker is the first approach at integrating a model checker in an embedded system development tool operating at the ‘model level of abstraction’[1]. It can be used to check real systems and make a real contribution to development and quality assurance support. Especially, it guides the user from the modelling point of view without requiring him to learn Model Checking notations or definitions. For example, the concepts of fairness and liveness are tackled from the point of view of the modeller rather than that of the verification theoretician.

1.6.2. Notation and definitions

This thesis develops a notation for the formal description of CIP Models (see chapter 3.).

It also introduces a definition of fairness specifically aimed at the type of situation which occurs in CIP models. With a couple of mouse clicks, the user can add or remove fairness attributes from transitions. These prevent the model checker from identifying false counterexamples or claims. Existing fairness definitions could also have been used, but would have required the user to redesign system components to accomodate this. The goal of the CIP model checker is to allow the user to perform model checking on existing systems without having to redesign these (see sections 3.7. and 4.4.).

[1] Tools such as Statecharts [41] offer some model checking possibilities [24], but do not offer the modelling power of CIP. See section 4.8. (page 69) for a discussion on this.

1.6.3. State-space traversal

An important step towards the realisation of this project was the creation of the execution tester. Previously, the code generated by CIP-Tool was the only means of actually traversing the system or indeed testing any particular transition of the system. CIP-Tool showed the relationship of the components but was unable to directly equate this to behaviour. Model testing during the design phase thus required code generation and compilation for every design iteration.

Besides being vital for the model checker, this behaviour tester allows the user to hand test the behaviour of the complete or incomplete model during the design phase. For the CIP user, this function can be almost as valuable as the model checker itself.

1.6.4. Coping with state-explosion

Throughout the implementation phase, care was taken to observe the extent of the state-explosion problem. The strong interaction structure of the CIP model resulted in this being less extreme than initially feared. Additionally a user-controlled reduction approach was implemented (*cluster reduction*) as was an automatic reduction mechanism (*partial order reduction*). All of these mechanisms were shown to make a noteworthy contribution to reducing the state-explosion problem. Nevertheless, very large systems tested still caused considerable strain through state-explosion.

1.7. Guide for the reader

1.7.1. General

Some readers of this thesis may have good knowledge of CIP-Tool, but less so of Model Checking. For others the situation may be vice-versa, and for others again both fields will probably be new. The reader should feel free to skip chapters on subjects with which he feels sufficiently familiar or which are of no consequence to him. For any ambiguity concerning terms defined or explained in those chapters and used in subsequent chapters, the reader is referred to the quick reference tables of section Appendix R: (page 219) or to the extensive index.

Chapters 2. to 4. discuss the background of the project and introduce many concepts and definitions to be used in later chapters.

Chapters 5. to 7. discuss the methods lying at the heart of the CIP model checker.

Results and conclusions are discussed more broadly in chapter 8.

1.7.2. Background

Chapter 2. discusses CIP. This chapter should enable the reader with no previous knowledge of CIP to gain a general overview of the functionality and method of CIP. For the more knowledgeable reader it recapitulates or can be used as a reference to the concepts used in later parts of this project.

Chapter 3. presents Model Checking basics and many of the aspects of Model Checking which are used in this project are introduced. This chapter defines many of the concepts which will be used in later chapters. It introduces *Kripke structures*, which are a tool in understanding whether properties are fulfilled. Finally a concept called *fairness* is introduced. This basically limits the possible input sequences the system model has to deal with by preventing behaviours which a classic state diagram allows but a real system doesn't (such as events occurring in a sequence which physical limitations of the system would not allow).

Chapter 4. looks into the history of Model Checking, and Model Checking in practice and discusses some of the available tools and their bearing on this project. It especially discussed the concept of fairness which is central in this project and compares it with definitions of fairness used elsewhere.

1.7.3. Realisation

The threads of chapters 2. and 3. finally join in chapter 5. This chapter discusses how the processes of the CIP machine combine to form a finite state-machine, and how this finite state-machine can be interpreted from the CIP perspective. The problem of state-explosion is discussed, as is the method of tackling this by a method called *cluster reduction*. Cluster reduction attempts to simplify models by leaving away certain components. Such components should ideally be outside the cone of influence of the property being investiagted. In strongly interconnected models it is not always practicable to determine whether this is the case and the possible effects of leaving away other components are considered. It is shown that such cluster-reduction can causes undesirable changes in the behaviour of the machine which may lead to false Model Checking results. The mechanism involved is discussed at length and mechanisms for preventing these problems are developed.

Chapter 5. also shows how the *properties* introduced briefly in section 1.3.2. and discussed at greater length in chapter 3. can be translated into the CIP environment. The concept of fairness is also discussed again, this time in perspective of its use in CIP.

The problem of state-explosion has already been mentioned and this is discussed at greater length[1] in chapter 3. Chapter 6. returns to this thematic and introduces *partial order reduction*. Partial order reduction is a method for reducing the number of states of a system without modifying its behaviour as far as the properties to be verified are concerned.

The implementation is discussed in chapter 7. and an idea is given of how the interface looks and how the tool is used in practice.

1.7.4. Conclusions

Chapter 8. presents the conclusions of the project together with a self critical appraisal and discusses the possibilities of further work.

1.7.5. Appendices

The appendices contain a lot of background information such as examples, proofs and discussions which were not sufficiently relevant or original to be placed in the main part of this thesis.

Appendix A: Sequentiality and fairness, contains notes relating to the discussion of paths and fairness of chapter 3. The correlation between time and sequentiality is investigated and illustrated with an example that was already known to the ancient Greeks: One of Zeno's paradoxes.

Appendix B: Notes on similar work, contains notes on chapter 4. going into proofs and backgrounds which would have gone beyond the limits of chapter 4.

Appendix C: Cluster reduction, discusses the cluster reduction attempts of chapter 5. and shows with examples why simpler attempts at solving the problem can lead to loss of behaviour.

Appendix D: Traversing the State-Space, introduces the *depth first search* state traversal algorithm which is used by the CIP model checker.

Appendix E: Traversal examples, provides further examples illustrating the algorithm introduced in appendix D.

Appendix F: Partial Order Reduction, provides background to chapter 6. with examples and discussions.

Appendix G: Data structures, summarises important data structures used in the realisation of this project.

[1] A so called *cone of influnce reduction* is used. This presents many problems in itself which are then addressed and solved.

Appendix R: Quick Reference, summarises the meanings of the symbols and functions used in this work and is located immediately before the index for quick access.

2. CIP in a nutshell

2.1. Purpose

In this chapter CIP will be introduced as far as it is relevant to this project. For fuller details, especially concerning the interface, the reader is referred to CIP manuals and literature [8], [9], [10] and [37].

2.2. Introduction to the CIP method

2.2.1. Why CIP?

Much of the software used in embedded systems, and indeed elsewhere, is developed and evolves in a way which can be described as organic. During the life cycle of a program, the code is continuously modified and added to, and as a result becomes increasingly unserviceable. Time pressure within the development environment favours the creation of code which is essentially functional but not developed according to any specific rigorous methodology. Different programmers with different styles modify each other's code leading to an increasingly unstructured status. This makes trouble-shooting tedious and costly. Consequently, inexcusable bugs often survive in the finished product.

CIP-Tool helps to solve these problems by moving away from the code based approach, replacing this by an object based approach. CIP provides objects of numerous types, such as communication objects or status objects, each with appropriate attributes and related to one-another in the appropriate way. The objects are represented graphically and attributes and relationships can be created or modified with simple mouse clicks. The behaviour of the system can easily be interpreted from the relationship of these objects. Different programmers can service the same objects, or can produce different objects and then connect them. Once this process is complete, the model can be converted to compiler-ready code (currently CIP- Tool supports C and Java).

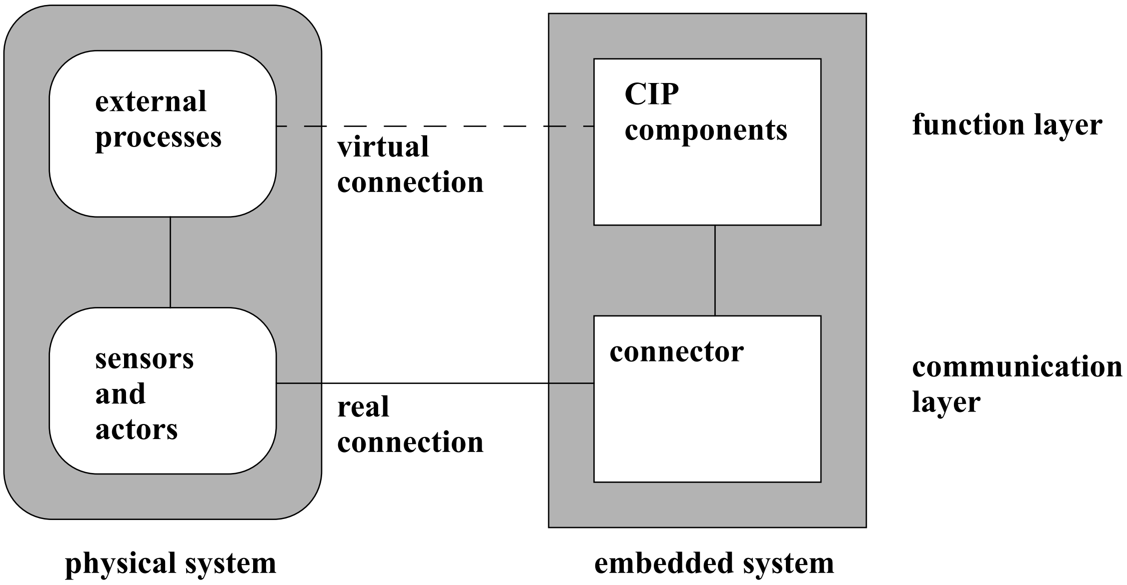
It is not just another gadget to relieve the programmer of repetitive chores. Neither is it merely a tool. It is a method which must be learnt and applied. The CIP method enforces structure on the programmer's approach to the problem and maintains a high level of serviceability and modifiability. CIP users commonly report that the method has led to considerable time and cost savings while raising the quality of the finished product.

2.2.2. Looking at an embedded system from the CIP perspective

figure 2.1. shows how embedded system software is connected to the physical system. On the function layer, the description of the physical system is a combination of the descriptions of all its components (their positions, velocities, pressures, voltages etc.). These are infinitely variable analog values. On the communication level, sensors and actors extract far simpler data from these values. A simple sensor, for example, is not concerned with the exact height of a column of liquid at all times (which is of no interest to the control software), but reduces this to atomic events (the height has fallen below or passed above a critical value). The sensor passes this information to the control software (in this case the CIP software) via the connector.

Likewise, when the control software deems it necessary to effect an action, this is passed to the corresponding actor via the connector. The actor then influences the physical state of the system by acting on one of its components. By passing through the communication layer, a virtual connection is created between the control software and the external processes on the function layer.

figure 2.1. Software architecture[1]



When programming, we can act as if this connection were real. In the software we receive *events* and transmit *actions*. These reflect events and actions taking place in the physical system.

It is central in the CIP method, that we do not ignore the mechatronic causality between our actions and events. For example, if we start a motor that closes a door, we must expect the door to be closed at some point. If necessary we must be able to act on the reception of the message that the door has closed. If the door is not closing, we do not expect such a message to occur and are not prepared to act on it (except possibly in an error handling framework). By making this distinction, we implicitly reflect that the po-

[1] From *The CIP Method: Component- and Model-Based Construction of Embedded Systems* by H. Fierz [37].

sition of the door in the physical system is more or less constant when the door is not moving and the position of the door is progressing when the door is moving. Whatever the situation, the door certainly never jumps. When moving from A to B it passes through all intermediate positions, only some of which may be of interest to the embedded software. It would, however, be too simplistic to say that the state of the door in the CIP-model is a synchronous *many to few mapping* of the physical state of the door. Such a simplification would ignore factors such as the time delay of the communication, action delay of actors, and the possibility of communication messages crossing between the CIP components and the physical system all of which may lead to discrepancies between the supposed and actual state of the CIP-components.

For further background on the links between the embedded system and the physical world, *Separate connection and functionality is the pivot in embedded system design* [28] by H. O. Trutmann is recommended reading.

2.3. Processes

2.3.1. What is a process?

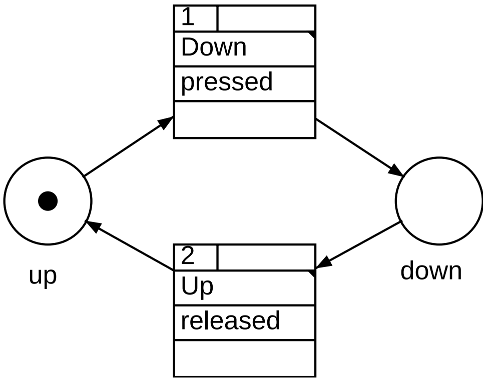
Reactive systems are commonly described by finite state machines. The decomposition of system functionality into finite state machines is also a central part of the CIP approach. When creating a finite state machine to model the system behaviour, we could describe the entire system in one large state machine. In CIP this is also possible but not encouraged. It is better practice to decompose the functionary into *processes*. Each process is concerned with one aspect of the behaviour of the system, and communicates with other processes by sending and receiving messages.

Processes are grouped into *clusters*. The importance and meaning of clusters will be discussed more fully in section 2.5. For the moment it is sufficient to state that processes within clusters work together more closely than processes in different clusters.

2.3.2. A simple process

This simple process of figure 2.2. models a button.

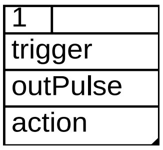
figure 2.2. A simple process modelling a button



The process has two states (*up* and *down*) and two transitions (numbered 1 and 2). Initially, the active state of the process is the state *up* (the token marker in the state circle identifies this as the initial state). When the event *Down* occurs, transition 1 is triggered changing the active state of the process to the state *down*. This transition sends the message *pressed*. When the process is in state *down*, the event *Up* can trigger transition 2 returning the process to its initial state.

Each transition is represented by two arrows and a box. The state from which the first arrow leads is the *pre-state* of the transition. The state to which the second arrow leads is the *post-state*. Between the arrows is a box as shown in figure 2.3.

figure 2.3. A transition description box



An *event* is a message received from outside the cluster, for example from a sensor of the physical system or from another cluster. An *inPulse* is a message received from another process within the cluster.

The *trigger* object shown in figure 2.3. is an event or inPulse which must be received by the process for the transition to occur. The *outPulse* object is a message sent to another process of the same cluster. The *action* object is a message sent to an actor of the physical system or to a process in another cluster.

Every transition must have a trigger attribute, whereas the *outPulse* and *action* attributes are optional.

2.3.3. Extended finite state machines

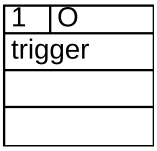
A finite state machine is a system with a finite number of states, where the active state changes in response to events[1]. The example process discussed above is a (simple) finite state machine.

Finite state machines do not always suffice to model all aspects of a system. Many systems also need to handle data or even take actions based on this data. An *extended* finite state machine is a machine which has extensions allowing such operations to be performed. Sections 2.3.4. and 2.3.5. discuss extensions to CIP enabling it to model extended finite state machines.

2.3.4. Code extensions

We cannot reduce every desired behaviour to simple state diagrams. For example, we may wish to include a control algorithm or handle other data. An *operation* is a code fragment which is an attribute of a transition. Whenever this transition is executed, the code fragment is also executed. Transitions with operations are recognisable by the letter O in the top right hand box.

figure 2.4. Transition with operation

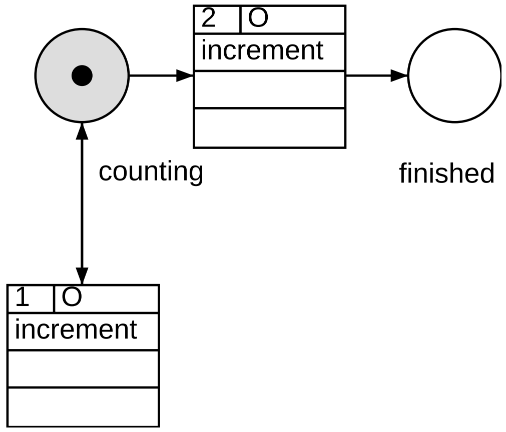


2.3.5. Conditions

Operations of the type introduced above can perform additional calculations, but cannot alone influence the overall behaviour of the system. We may wish to make transitions dependent on ‘hand programmed’ code. CIP-Tool provides a mechanism for this called a *condition*.

[1] A more comprehensive discussion of finite state machines can be found in section 3.2.2. (page 36).

figure 2.5. Process with conditional transitions



In the process of figure 2.5. we see that both represented transitions have the same pre-state and the same trigger element. In order to tell the process how to react to this trigger, we implement conditions. These are code fragments returning a boolean value determining whether or not the transition is to be executed. In this example, the condition could depend on the value of a variable n which is incremented by the operation of both transitions. When n is below a threshold value, transition 1 is executed ('n<threshhold' is the condition attribute of transition 1), otherwise transition 2 is executed (ELSE is the condition attribute of transition 2).

These conditions form a *switch* structure. The presence (or need) of such a structure is shown in the state diagram view by the grey shading of the states to which switch structures apply.

Every switch must include an ELSE condition. Thus it is guaranteed that when the process is in the pre-state of a transition and the trigger of that transition is received, then a transition will be executed.

With the exception of ELSE, which obviously is considered last of all when executing a switch structure, no order can be defined for the conditions in the structure. It is thus the programmer's responsibility to ensure all the conditions are mutually exclusive.

As the parsing of code is beyond the scope of the CIP model checker, the model checker assumes the general case that any of the alternative transitions is enabled can be executed in any given situation. Some more restrictive methods such as *gates* and *master-slave* structures introduced in section 2.6. (page 28).

2.4. Non external triggers

The user may require functions enabling a transition to be executed without an external trigger. This is done using *timers*, *chains* and *autos*. These trigger mechanisms are called *extensions*.

2.4.1. Timer

The *timer* extension allows a process to retrigger itself after a set time has elapsed.

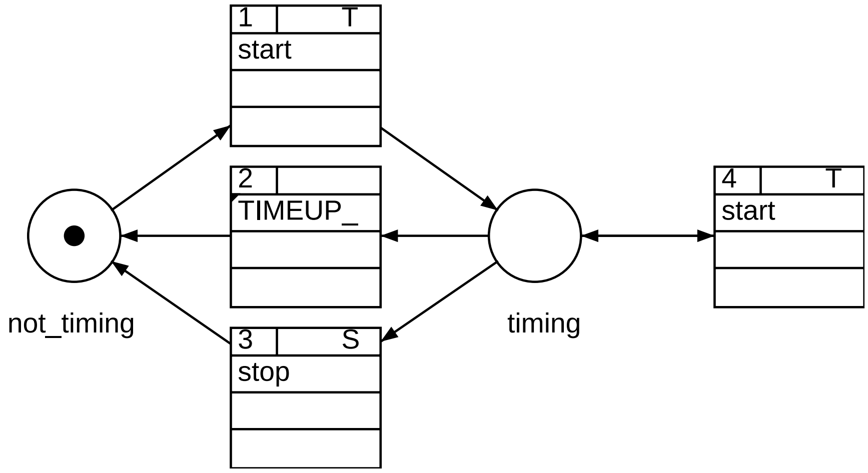
To set the timer, a transition must have the *set timer* extension as attribute. When this transition is executed, a timer is set. After the timer times out, the process is sent a *timeup* trigger.

In the example process of figure 2.6. a timer is set by transitions 1 and 4. The letter T indicates that the transition has the *set timer* attribute. Note that (as in the case of transition 4) a running trigger is restarted by such a transition.

Transition 2 is triggered by the timer when this times out. The TIMEUP_ trigger is a predefined message.

Transition 3 stops a running timer. The *stop timer* attribute is shown by the letter S. In this example it is not strictly necessary to stop the timer as a *timeup* being received in the state *not_timing* is of no consequence.

figure 2.6. Process with timer function



Cases are imagineable where incorrect behaviour is obtained in verification by a timer being able to timeout although it was never set. For this reason the CIP model checker can model timers explicitly.

2.4.2. Chains

A *chain* is a timer with a time of zero. It can be used to allow a process to activate itself. *set chain* (letter C) and *chain* (message CHAIN_) are equivalent to *set timer* and *timeup* respectively. There is no *stop chain*.

2.4.3. Auto

An *auto* is a trigger which is continuously and automatically received. It cannot be set or stopped. Autos can be used, for example, to activate processes carrying out repetitive tasks in the background. The auto message is written AUTO_.

2.5. Clusters

2.5.1. What is a cluster?

A cluster is a set of processes which are connected by a common scheduling strategy. This scheduling strategy is *run to completion*. The exact mechanism implied will be discussed in section 2.5.5.

On account of the scheduling strategy, communication between processes within clusters is synchronous whereas communication between processes of different clusters is asynchronous. Because of this, different clusters of a CIP system can be implemented on separate processors and even at physically separated locations.

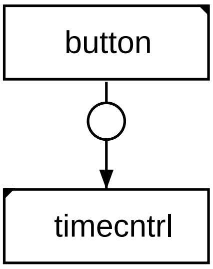
Different clusters can equally also share a single processor. The use of separate clusters in such situations is justified when the purposes of the clusters are sufficiently independent. For a further discussion of multi-cluster systems see section 2.7.

2.5.2. Pulse translation

A *pulse* is a message passed from one process to another process in the same cluster. The user must define which processes may pass pulses to one another.

Let us consider the processes of figure 2.2.and figure 2.6. We want the button process to send messages to the timer process. For this we must define a connection from process *button* to process *timecntrl*. This is done in the *pulse cast net* view of CIP Tool as shown in figure 2.7.

figure 2.7. The pulse cast net view of a cluster with processes button and timecntrl

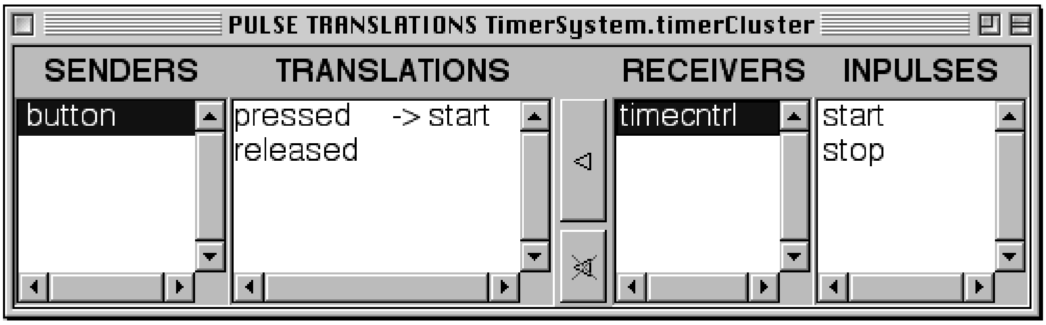


The arrow from button to timecntrl indicates that *button* can send pulses to *timecntrl*. If this relation were not defined, pulses could not be sent.

A pulse translation editor is used to define exactly which pulses may be passed by this connection and how they are to be translated.

In the pulse translation editor of figure 2.8., the outPulse *pressed* of process *button* is mapped to the inPulse *start* of process *timecntrl*.

figure 2.8. The pulse translation editor



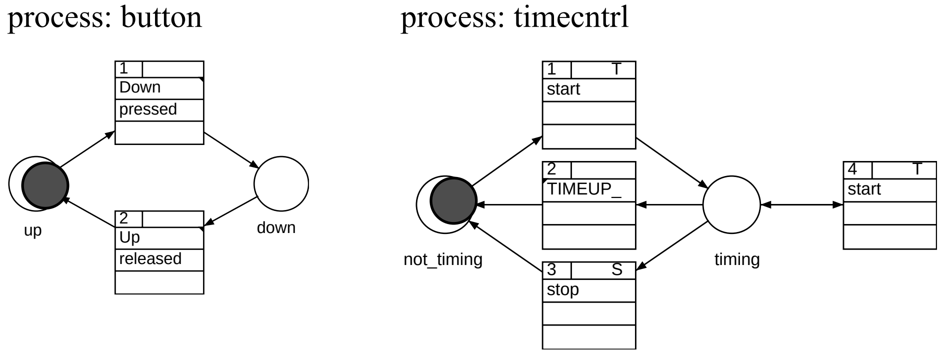
PULSE TRANSLATIONS TimerSystem.timerCluster

SENDERS	TRANSLATIONS	RECEIVERS	INPULSES
button	pressed -> start	timecntrl	start
	released		stop

The example shown in figure 2.9. shows how this pulse translation works in practice.

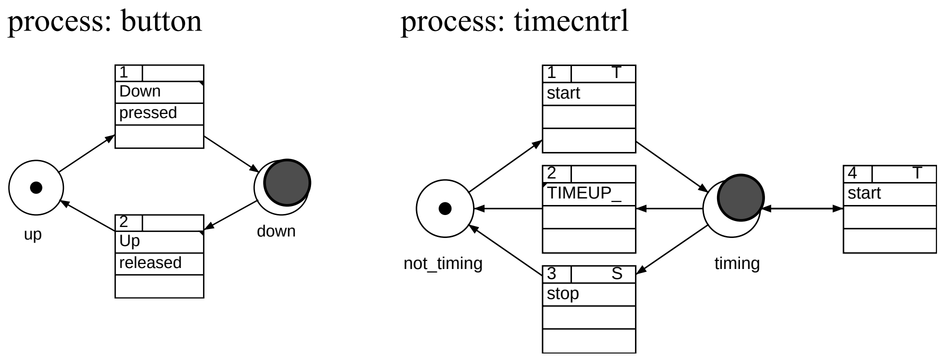
figure 2.9. Processes button and timecntrl in initial state

● marks current state of process



Process *button* receives the message Down which triggers transition 1 and changes the active state of *button* to *down*. This transition sends the outPulse *pressed* which is translated (as shown in figure 2.8.) to the inPulse *start* in process *timecntrl*. This inPulse triggers transition 1 changing the active state of *timecntrl* to *timing*.

figure 2.10. Processes of figure 2.9. after receiving message Down



process: button

process: timecntrl

2.5.3. Cast order

The user must not only define which processes may pass pulses to one another but also the order in which this passing occurs. This guarantees a deterministic and reproduceable behaviour.

The pulse cast net of figure 2.11. shows a cluster with four processes (*button*, *door*, *lamp* and *controller*). Transitions triggered in process *button* can be propagated to process *lamp* by several different ways. It could therefore even occur that process *lamp* receives more than one message as a result of a single transition in process *button*. The order in which the pulses are received influences the response of the receiving process. Therefore it is vital that a cast order is defined to lay down the order in which pulses are sent by a process. For every sending process there exists a cast order list.

figure 2.11. A pulse cast net in which transitions can be propagated by several different ways.

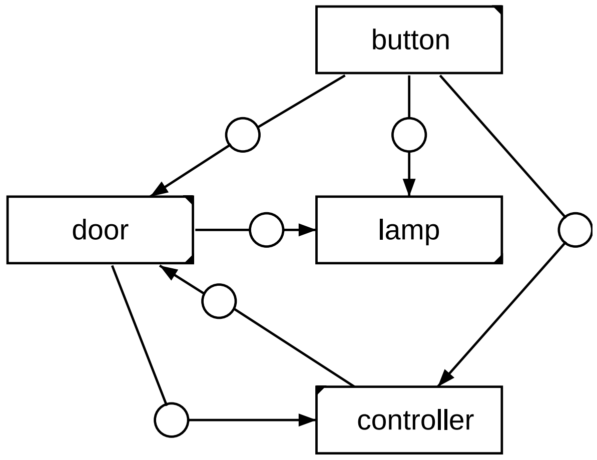
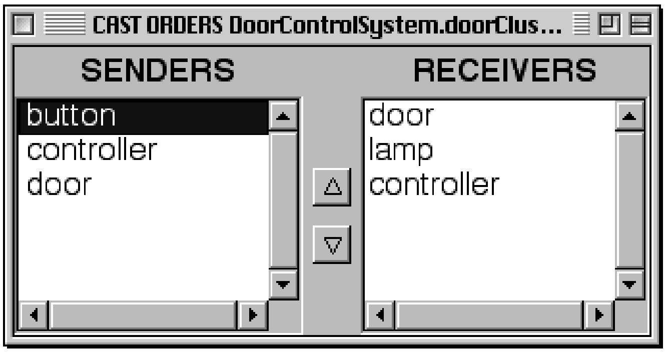


figure 2.12. Pulse cast editor defining the order in which outPulses sent by process but-ton *are cast*



CAST ORDERS DoorControlSystem.doorClus...

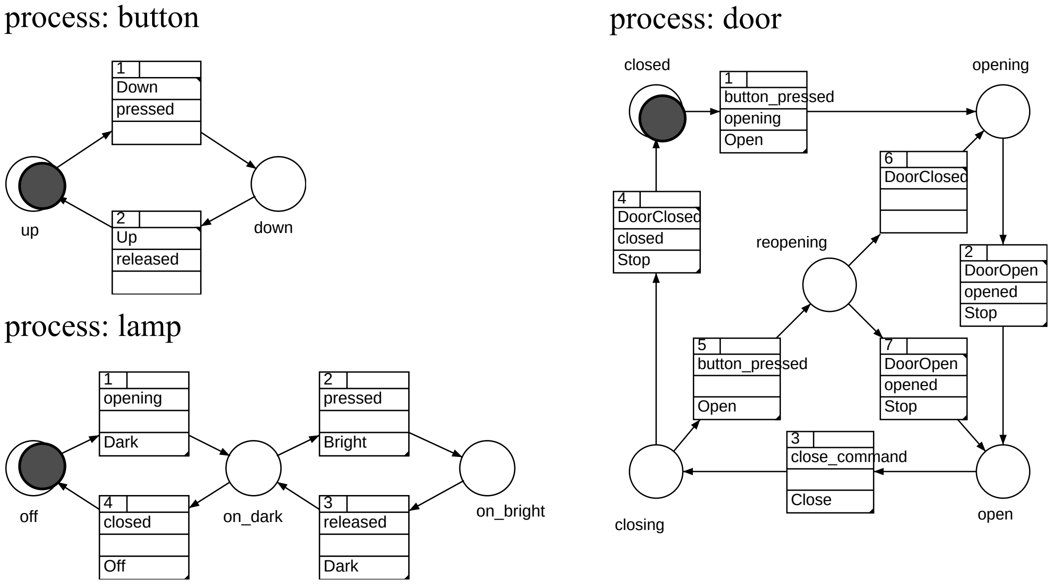
SENDERS	RECEIVERS
button	door
controller	lamp
door	controller

For the example of figure 2.12. process button first sends an outPulse to process *door*. Only when the triggered cascade is complete does process *button* send the outPulse to process *lamp*. When that trigger is also complete the pulse is sent to process *controller*.

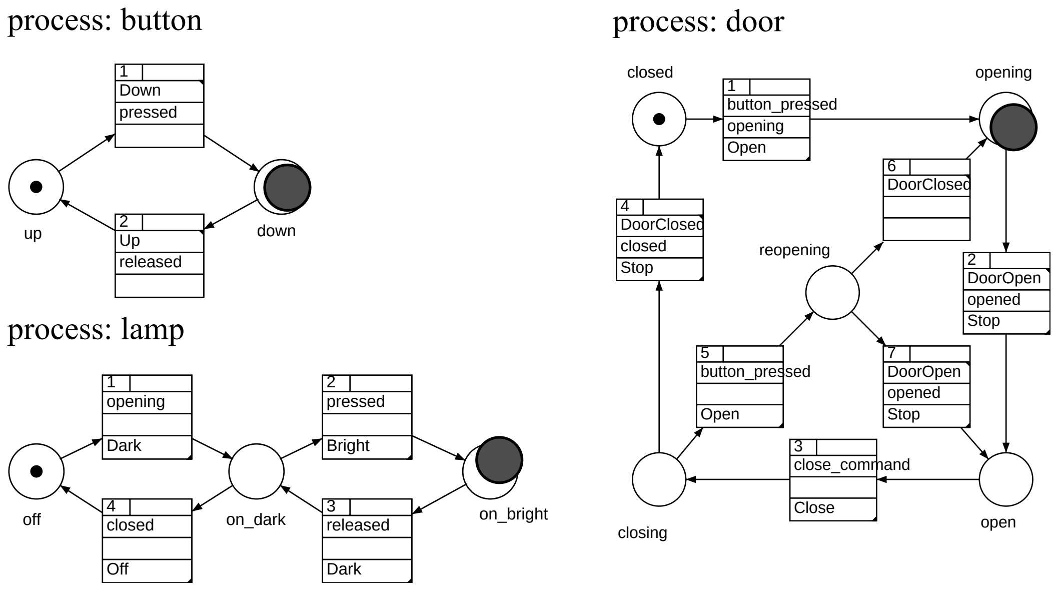
In the following example we show the importance of defining cast order when a single transition can influence a process through more than one propagation path.

figure 2.13. Some of the processes of cluster of figure 2.11.

● marks current state of process

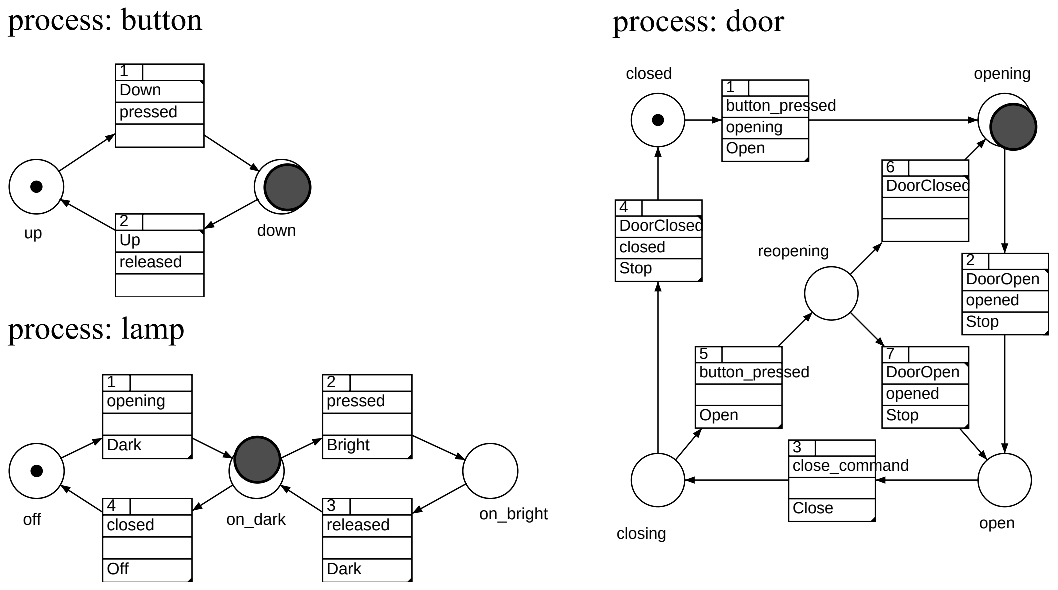


The message *Down* is received by process *button* where it triggers transition 1 and sends the outPulse *pressed*. This is first sent to process *door* where it is translated to the inPulse *button_pressed* and triggers transition 1 thus sending the outPulse *opening*. This is sent to process *lamp* where it is translated to the inPulse *pressed* and triggers transition 1. The outPulse *pressed* from process *button* is also sent to process *lamp* where it is translated to the inPulse *pressed* and triggers translation 2. Finally, the outPulse *pressed* from process *button* is sent to process *controller* (not shown) *figure 2.14. Processes of figure 2.13. after execution of cascade triggered by message* Down



If the cast order had been different, and the outPulse *pressed* first sent to process *lamp*, it would not have triggered a transition there. The outPulse *opening* of process *door* would have triggered transition 1 in process *lamp*. The final state of lamp would have been different.

figure 2.15. Processes of figure 2.13. after execution of cascade triggered by message Down *but with modified cast order (door after lamp).*

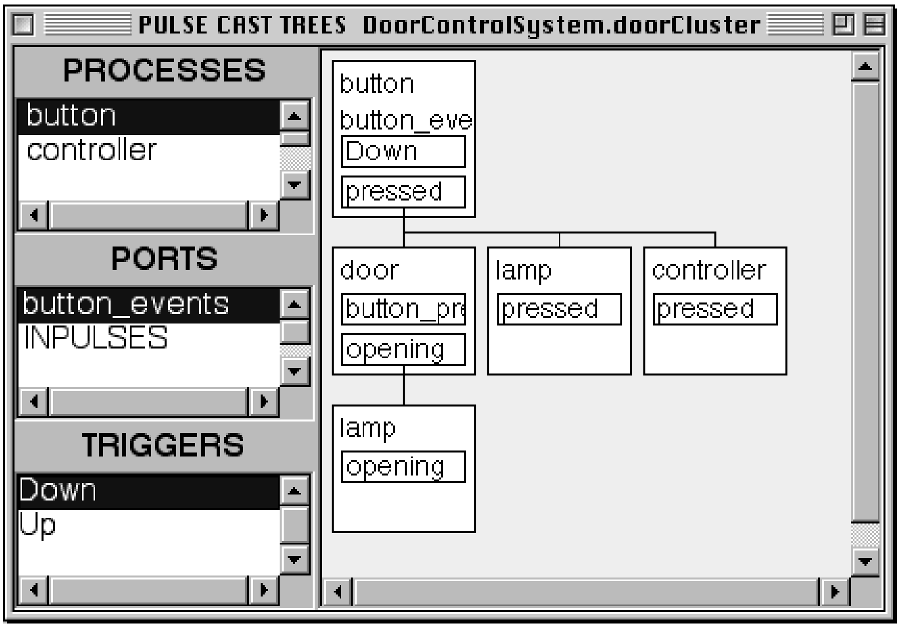


2.5.4. Interaction trees and their execution

The definitions made by the user as discussed in sections 2.5.2. and 2.5.3. define the order in which pulses are sent to processes. An *interaction tree* is a mapping of this order.

Such trees are automatically generated by CIP-Tool and can be displayed in a browser as illustrated in figure 2.16. This example shows the interaction tree discussed in the example of section 2.5.3.

figure 2.16. Interaction tree of example of section 2.5.3.



For every process it shows the trigger and the pulse sent. It is not necessary that every pulse shown in the tree is actually sent. If the process is not in the right state to execute the necessary transition, the transition is not executed and neither are any of the transitions below it in the tree.

Note that in CIP, these trees are always traversed depth-first and left to right.

In the example of figure 2.16. the processes are activated in the order: *button*, *door*, *lamp*, *lamp*, *controller*.

During the development phase, CIP-Tool automatically checks whether any change could result in a recursive interaction tree structures and does not allow such a structure to be defined (so ensuring that every interaction tree remains finite and thus guarantees termination).

2.5.5. Run to Completion Semantics

The scheduling algorithm of CIP is *run to completion semantics*. This means that within a cluster, only a single interaction tree can be in execution at any time. The scheduler accepts no external messages until the execution of the tree is complete.

Given the states of all processes in the cluster and a message, the states of all processes in the cluster after the transition are deterministic (allowing of course for the behaviour of code extensions as described in section 2.3.4.).

As the cluster transition cannot be interrupted, it can be considered to be a single instantaneous transition.

2.6. Inspections, Gates and Master-Slave structures

2.6.1. Inspections

In section 2.3.5. transitions were introduced with both the same trigger element and the same pre-state. In order to know which of the transitions is to be executed in which situation, additional code was provided. Such switches are called *conditions*.

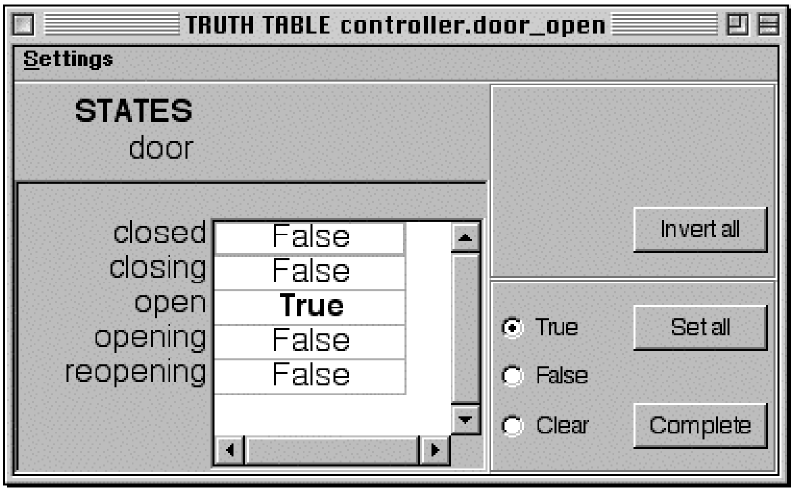
In some cases, the condition may be dependent on the state or code variable of another process. In this case, a message is sent to that process and the requested information returned. Such information sharing operations are called *inspections*. The CIP-Model provides inspection links (similar to the pulse transmission links of section 2.5.2.). The actual handling of the request, however, must be implemented in code. This code is an attribute of the *inspected* process in the model structure. It would be beyond the scope of this project for the model checker to actually parse this code. Wherever possible, the developer is encouraged to use *gates* instead.

2.6.2. Gates

Where an inspection inspects only the states of another process, a codeless and more transparent alternative is provided. This alternative is called a *gate*. Not only is a gate easier and faster to implement than ‘hand written’ code, but it can be fully described within the framework of CIP, an advantage when we come to considering this in Model Checking.

Gates inspect processes using a similar communication framework to conditions. Rather than code being attached, however, a truth table is defined.

figure 2.17. truth table inspecting process door.



TRUTH TABLE controller.door_open

Settings

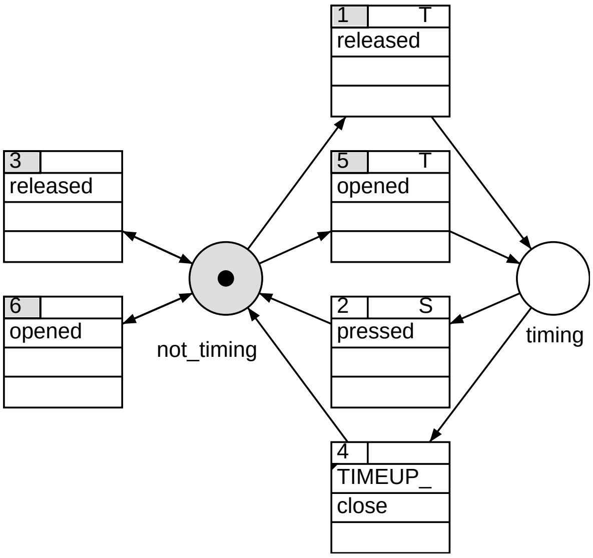
STATES
door

closed	False
closing	False
open	**True**
opening	False
reopening	False

The truth table of this example inspects a single process. Up to three processes can be inspected in a single truth table.

This gate is used in the process *controller* shown in figure 2.18. This is process *controller* of the cluster of figure 2.11. and figure 2.13.

figure 2.18. process controller *of cluster of figure 2.11. and figure 2.13.*



When the process is in state *not_timing* and the inPulse *released* is received, a switch structure is used to ascertain whether transition 1 or transition 3 should be executed. Transition 1 uses the truth table of figure 2.17. and transition 3 uses ELSE.

Therefore, when the inPulse *released* is received in state *not_timing*, transition 1 is executed iff the state of *door* is *open*. Transition 3 is executed otherwise.

Likewise, when the process is in state *not_timing* and the inPulse *opened* is received, another switch structure is used to ascertain whether transition 5 or 6 should be executed.

Note that, as with *inspections*, every gate must include an ELSE case to ensure that one of the alternative transitions is executed.

2.6.3. Master-Slave Structures

When many transitions within a structure depend on the same gate, it may be preferable to use a *master-slave* structure. In a master-slave structure, the slave process has more than one set of transitions (such a transition set is a *mode*). The *slave* switches between one mode and another depending on the state of other processes known as the *master* processes. The following example shows how the process of figure 2.18. can also be represented as two modes.

The functionality of this process remains identical to that of figure 2.18.

figure 2.19. Mode nottiming *of process* controller

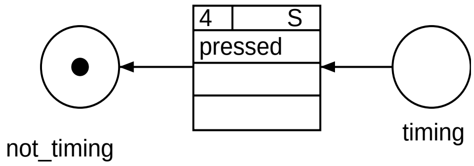
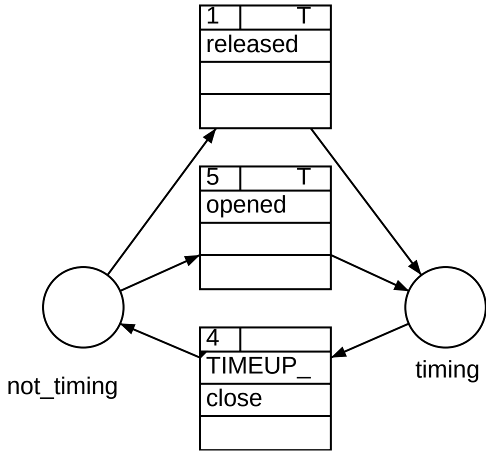


figure 2.20. Mode istiming *of process* controller



Note that all modes of a process have the same states.

These mode diagrams each resembles a process diagrams, only that the *slave* process has several such mode diagrams, each with a different transition set. The mode that applies is changed through changes in state of other processes, the *master* processes.

In the *mode control net* view of the cluster (figure 2.21.), these master slave dependencies are established.

Mode settings are defined using the mode setting editor as shown in figure 2.22.

figure 2.21. Mode control net

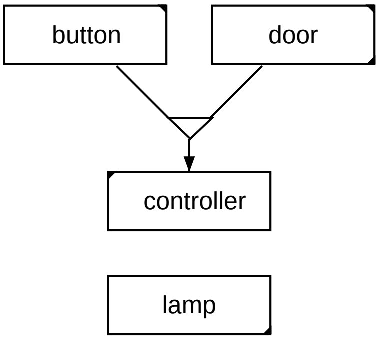
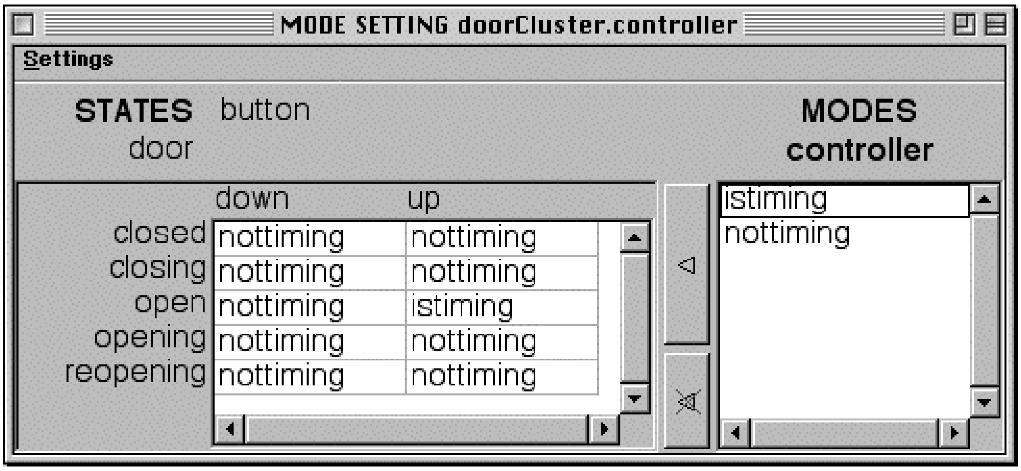


figure 2.22. Mode setting editor for process controller *from figure 2.19. and figure 2.20.*



MODE SETTING doorCluster.controller

Settings

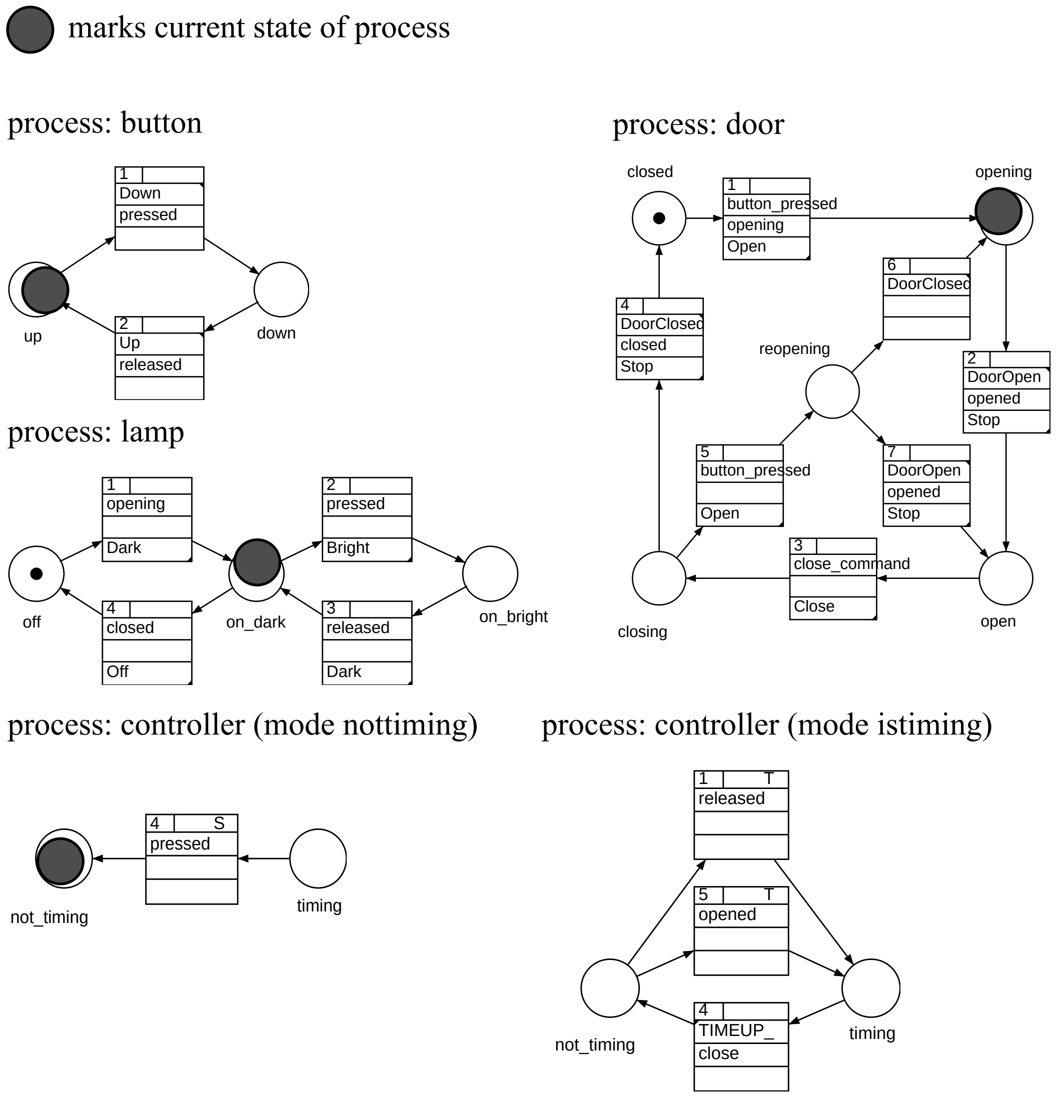
STATES button
door

MODES
controller

	down	up
closed	nottiming	nottiming
closing	nottiming	nottiming
open	nottiming	istiming
opening	nottiming	nottiming
reopening	nottiming	nottiming

istiming
nottiming

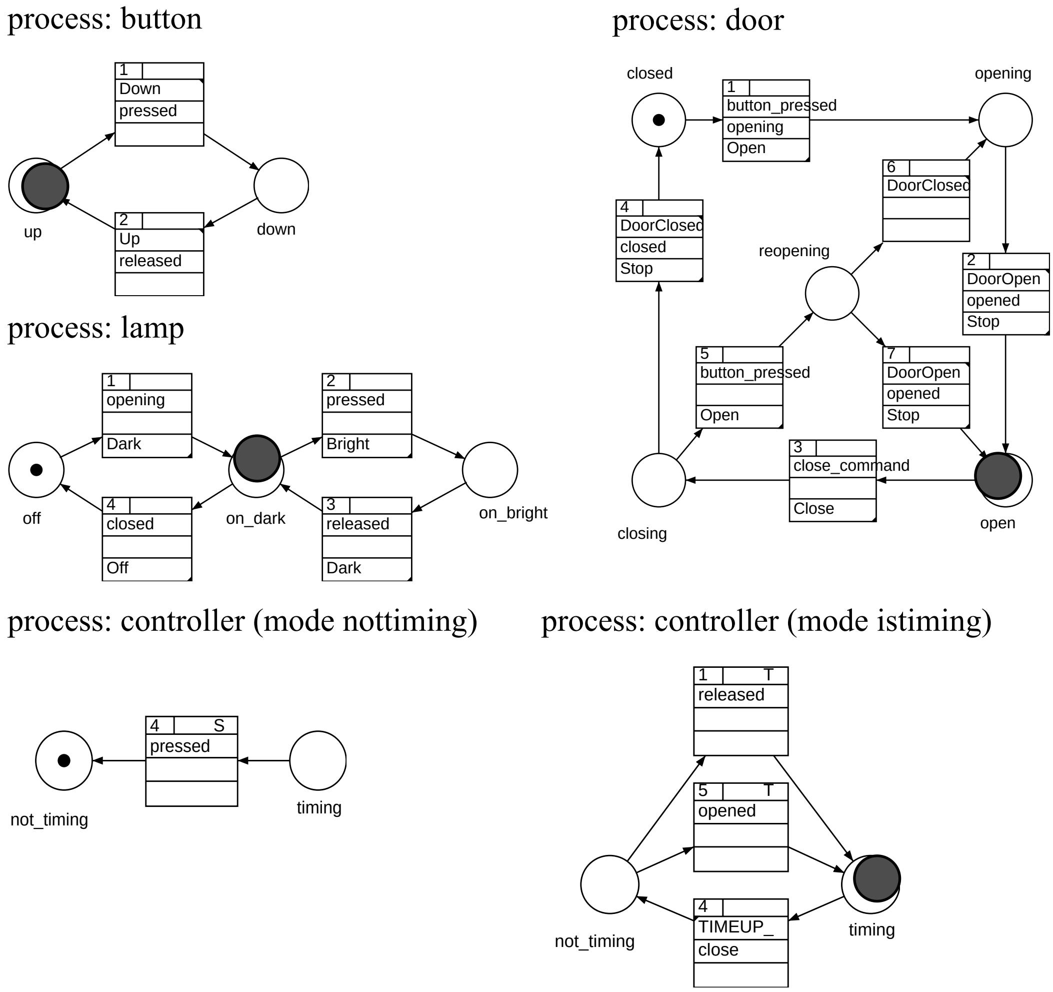
The following example illustrates the mode changing mechanism *figure 2.23. Cluster before transition*



Before the transition, the processes are in the states shown. Because the processes *button* and *door* are in the states *up* and *opening* respectively, controller is in mode *nottiming*.

The transition is triggered by the event *DoorOpen* received by process *door*. Transition 2 in process *door* is triggered. The action *Stop* is sent to the door motor and the outPulse *opened* to process *controller*, where it is translated to the inPulse *opened*. The master processes of controller now have as active states *up* and *open*, therefore the mode of controller has changed to *istiming*. Process *controller* executes transition 1 and starts the timer.

figure 2.24. Cluster of figure 2.23. after transition



process: button

process: door

process: lamp

process: controller (mode nottiming)

process: controller (mode istiming)

2.7. Multi-cluster systems

2.7.1. A single cluster

A cluster is a grouping of processes. Pulses can only be used to communicate between processes of the same cluster, and the run to completion semantics (described in section 2.5.5.) apply strictly within the cluster.

2.7.2. Multiple clusters

A CIP system can have multiple clusters, each of which has a number of processes. Run to completion semantics apply within each cluster, but different cluster are independently scheduled and can execute in parallel.

2.7.3. How can different clusters communicate?

Channels can be defined between clusters through which messages can be passed. Messages are written to inter-cluster channels in the same way that messages are sent to the physical system to cause actions. Likewise, messages received from channels are handled similarly to events received from the physical system. Data can also be passed with the messages. However, inspections are not possible and do not make sense due to the asynchronous nature of multi-cluster systems. Likewise, master-slave dependencies are not possible.

2.7.4. Why does a system need more than one cluster?

A system may need several clusters to be able to implement these on separate processors, possibly even in separate locations. It can also make sense to implement several clusters on a single processor when the function of the clusters is sufficiently separate to warrant this.

Functionality can be split between clusters when the tasks are sufficiently independent. Inspections between clusters are not possible although data exchange can be handled by attaching data to the messages. As the communication is asynchronous, such information does not necessarily reflect the state of the sending cluster at the moment it is processed by the receiving cluster. Ideally, therefore, separate clusters should not need to share real-time data.

Another advantage of having separate clusters is that each cluster can be seen as a ‘limit of development’, *i.e.* each cluster is designed and implemented by a different unit. The interfaces between clusters are defined beforehand.

2.8. CIP and model checking

The suitability of CIP for model checking lies in its formal nature and definitions. The behaviour of a system implemented in CIP can be extracted entirely from the CIP model. All information needed to check a given property is contained in the model. Whereas most model checkers require the user to extensively define the system behaviour before even simple checking tasks can be attempted, this definition is already given in a CIP model. This saves time and, more importantly, eliminates an important source of error. In CIP, the behaviour of the implemented system is identical to that of the model, pending of course errors in the code generation and compiler. Serious errors are unlikely as both CIP Tool and compilers have been extensively tested and used in practice and any such errors would have come to light and been addressed by now.

Another source of error is the model checker itself. As the checker presented in this thesis is new and only minimal experience with it is available, the correctness aspect must be taken seriously. The proofs provided in this thesis address this issue. However, there may also be errors in the implementation. In order to gain confidence on this matter, an extension to the CIP model checker with command line interface is desirable as introduced in section 1.5.3. (page 8).

3. Model Checking Concepts

3.1. Purpose

The purpose of this chapter is to define Model Checking concepts such as states, transitions, paths, Kripke structures and fairness as they are used in the context of this project.

In the case of fairness, the definition used in this thesis deviates slightly from other definitions. The definition chosen in this thesis is the easiest to apply to the CIP Model for the required purpose. A discussion of the differences to other definitions is to be found in section 4.4. (page 54). As for definitions of such structures as state machines, transitions and their attributes and dependents[1], various slightly different definitions are to be found in literature, hence the necessity to define them unambiguously here.

The properties defined in section 3.5. are defined especially for the purpose of this thesis and for their application in the CIP model checker. For more common definitions, the reader is referred to section 4.3.

For the reader interested in further background to the themes discussed here, the reading of Appendix A: Sequentiality and fairness is recommended.

3.2. Basic concepts of a model system

3.2.1. The essentials of a model system

When modelling a system we reduce the system to the properties that interest us for the purpose of the model. A system engineer might model a switch as a device with two states between which it can be switched by an external operator. For the switch designer, this approach is insufficient. He may be more interested in the materials of the switch and how they react to being moved.

When a software engineer models an embedded system, he looks at the aspects of the system that may be of interest to the software. These are essentially the events of the system to which the software should react, the actions of the system which the software can trigger and the correlation mechanism between them. This model structure is called *extended finite state machine*. For an explanation of why the term *extended* is used, see section 2.3.3.

[1] For further background see *Model Checking* by Clarke, Grumberg and Peled [18] and *Software Reliability Methods* by Peled [30].

The following sections introduce many definitions used in this thesis. The reader wishing to skip these definitions can proceed directly to section 3.5.

The formalisations introduced below are specific to this thesis.

3.2.2. Describing the finite state machine

A finite state machine is defined by the tuple:

M := (**S**, **E**, **A**, **R**, s_{init}).

M is the finite state machine.
S is the atomic set of system-states.
E is the atomic set of events.
A is the atomic set of actions and includes the null-action.
R is the transition relation: $\mathbf{R}: \subseteq \mathbf{E} \times \mathbf{S} \times \mathbf{S} \times \mathbf{A}$
$s_{init} \in \mathbf{S}$ is the initial system-state.

Note that **A** is of little relevance in this project but is included here for completeness.

3.2.3. Transition relations

The members of **R** are called transitions.

The following mappings are defined for transitions:

pre-state:

pre: $\mathbf{R} :\rightarrow \mathbf{S}$
$pre((e, s_0, s_1, a)) := s_0$

post-state:

$post$: $\mathbf{R} :\rightarrow \mathbf{S}$
$post((e, s_0, s_1, a)) := s_1$

trigger-element:

$trig$: $\mathbf{R} :\rightarrow \mathbf{E}$
$trig((e, s_0, s_1, a)) := e$

action-element:

$action$: $\mathbf{R} :\rightarrow \mathbf{A}$
$action((e, s_0, s_1, a)) := a$

3.2.4. State relations

For every state, the following mappings are defined:

successors:
the successor set of a state s is the set of post-states of transitions which have s as pre-state.

suc: $\mathbf{S} :\rightarrow 2^{\mathbf{S}}$
$suc(\mathrm{s}) := \{\mathrm{s}' \mid \exists \mathrm{r} \in \mathbf{R}: ((pre(\mathrm{r}) = \mathrm{s}) \wedge (post(\mathrm{r}) = \mathrm{s}'))\}$

predecessors:
the predecessor set of a state s is the set of pre-states of transitions which have s as post-state.

$pred$: $\mathbf{S} :\rightarrow 2^{\mathbf{S}}$
$pred(\mathrm{s}) := \{\mathrm{s}' \mid \exists \mathrm{r} \in \mathbf{R}: ((pre(\mathrm{r}) = \mathrm{s}') \wedge (post(\mathrm{r}) = \mathrm{s}))\}$

enabled transitions:
the set of transitions enabled in a state is the set of transitions of which that state is the pre-state.

$enabled$: $\mathbf{S} :\rightarrow 2^{\mathbf{R}}$
$enabled(\mathrm{s}) := \{\mathrm{r} \mid \mathrm{s} = pre(\mathrm{r})\}$

if $\mathrm{r} \in enabled(\mathrm{s})$, we say that r is **enabled** is s.

disabled transitions:
the set of transitions disabled in a state is the complement of the set of transitions enabled in that state.

$disabled$: $\mathbf{S} :\rightarrow 2^{\mathbf{R}}$
$disabled(\mathrm{s}) := \mathbf{R} \cap \neg enabled(\mathrm{s})$

if $\mathrm{r} \in disabled(\mathrm{s})$, we say that r is **disabled** is s.

dead ends:
dead ends are states without successors.

$\mathbf{D} :\subseteq \mathbf{S} \mid \forall \mathrm{s} \in \mathbf{D}: suc(\mathrm{s}) = \varnothing$

3.2.5. Enabling, disabling and independence of transitions

enabling transitions:

A transition r_0 **enables** another transition r_1 iff r_1 is disabled in the pre-state of r_0 and enabled in the post-state of r_0. The function *enables* returns all transitions enabled by a transition.

enables: $\mathbf{R} :\rightarrow 2^{\mathbf{R}}$
enables(r) := {r’ | r’ ∈ (*enabled*(*post*(r)) - *enabled*(*pre*(r)))}

disabling transitions:

A transition r_0 **disables** another transition r_1 iff r_1 is enabled in the pre-state of r_0 and disabled in the post-state of r_0. The function *disables* returns all transitions disabled by a transition.

disables: $\mathbf{R} :\rightarrow 2^{\mathbf{R}}$
disables(r) := {r’ | r’ ∈ (*disabled*(*post*(r)) - *disabled*(*pre*(r)))}

dependency of transitions:

The transitions which are *independent* of a transition r are neither enabled nor disabled by r.

independent: $\mathbf{R} :\rightarrow 2^{\mathbf{R}}$
independent(r) := {r’ | (r’ ∉ *enables*(r)) ∧ (r’ ∉ *disables*(r))}

A transition is *dependent* on r if it is not independent.

dependent: $\mathbf{R} :\rightarrow 2^{\mathbf{R}}$
dependent(r) := {r’ | (r’ ∈ *enables*(r’)) ∨ (r ∈ *disables*(r’))} = **R** - *independent*(r)

3.2.6. The state diagram

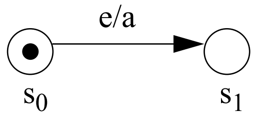
The state diagram is a diagram showing all states of the system and the transitions in relationship to those states.

A state is represented by a circle which can be labelled with the name of the state.

The initial state additionally has a marker in the circle.

A transition r is represented by an arrow pointing from *pre*(r) to *post*(r). It can be labelled with the value of *trig*(r) or the values of *trig*(r) and *action*(r) separated by the ‘/’ character.

figure 3.1. The state diagram for the simple system ({s_0, s_1}, {e}, {a}, {(e, s_0, s_1, a)}, s_0)

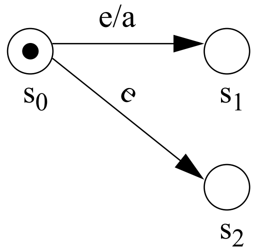


3.2.7. Determinism of transitions

A transition is deterministic iff there is no other transition in R with which it shares both pre-state and trigger.

$\textit{deterministic}$: $\mathbf{R} :\rightarrow \{\text{true, false}\}$
$\textit{deterministic}(\text{r}) := (\forall \text{r}' \in \mathbf{R}\text{: } (\textit{pre}(\text{r}) = \textit{pre}(\text{r}')) \wedge (\textit{trig}(\text{r}) = \textit{trig}(\text{r}')) \Rightarrow (\text{r} = \text{r'}))$

figure 3.2. State diagram for a simple system with a non deterministic transition



3.3. Paths

3.3.1. Path definition

A path is a sequence of states such that there exists a sequence of transitions permitting the states to be traversed in that order.

$$\mathbf{P^*} := \{\ \text{f: } \mathbb{N} \rightarrow \mathbf{S} \mid \forall \text{n} \in \mathbb{N}\text{: } \begin{array}{ll} \text{f[0]} \in \mathbf{S} \cup \{\varnothing\} & \\ \text{f[n+1]} \in \textit{suc}(\text{f[n]}) \cup \{\varnothing\} & \text{iff f[n]} \neq \varnothing \\ \text{f[n+1]} = \varnothing & \text{else} \\ \} & \end{array}$$

3.3.2. States visited, path length, first and last states

If a state s occurs in a path, we say the state is visited by the path.

$\textit{visited}$: $\mathbf{P^*} :\rightarrow 2^{\mathbf{S}}$
$\textit{visited}(\text{p}) := \{\ \text{s} \mid \exists \text{n} \in \mathbb{N}\text{: p[n]} = \text{s}\ \}$

The following functions require no additional explanation.

$$\begin{array}{ll}
\textit{length}\text{: } \mathbf{P^*} :\rightarrow \mathbb{N} \cup \{\infty\} & \\
\textit{length}(\text{p}) := 0 & \text{iff } \text{p}[0] = \varnothing \\
\textit{length}(\text{p}) := \text{n} \mid (\text{p}[\text{n-1}] \in \mathbf{S}) \wedge (\text{p}[\text{n}] = \varnothing) & \text{iff } \exists \text{n} > 0 \mid \text{p}[\text{n}] = \varnothing \\
\textit{length}(\text{p}) := \infty & \text{else}
\end{array}$$

A path of infinite length is an *infinite path*. Other paths are *finite paths*.

$\textit{first}$: $\mathbf{P^*} :\rightarrow \mathbf{S} \cup \{\varnothing\}$
$\textit{first}(\text{p}) := \text{p}[0]$

$$\begin{array}{ll}
\textit{last}\text{: } \mathbf{P^*} :\rightarrow \mathbf{S} \cup \{\varnothing\} & \\
\textit{last}(\text{p}) := \text{p}[\textit{length}(\text{p})\text{-1}] & \text{iff } \textit{length}(\text{p}) \geq 1 \text{ and } \textit{length}(\text{p}) \neq \infty \\
\textit{last}(\text{p}) := \varnothing & \text{else}
\end{array}$$

$\textit{occurrences}$: $(\mathbf{S} \cup \mathbf{R}) \times \mathbf{P^*} :\rightarrow \mathbb{N} \cup \{\infty\}$

$\forall(\text{s, p}) \in \mathbf{S} \times \mathbf{P^*}$:
occurrences(s, p) determines the number of occurrences of s in p.

$\forall(\text{r, p}) \in \mathbf{R} \times \mathbf{P^*}$:
occurrences(r, p) determines the number of occurrences in p of *post*(r) occurring immediately after *pre*(r).

3.3.3. Subpaths

A subpath of a path p is a path whose states are a sub-sequence of the sequence of p:

$$\begin{array}{lll}
\textit{subpath}\text{: } \mathbf{P^*} :\rightarrow 2^{\mathbf{P^*}} & & \\
\textit{subpath}(\text{p}) := \{ \text{ f: } \mathbb{N} \rightarrow \mathbf{S} \mid \exists \text{ (m, n)} \in \mathbb{N}^2, \forall \text{k} \in \mathbb{N}\text{:} & \text{f[k]} = \text{p[k+n]} & \text{if k} \leq \text{m} \\
 & \text{f[k]} = \varnothing & \text{if k} > \text{m} \\
 & \} &
\end{array}$$

3.3.4. Transitions and paths

It would be tempting to consider a transition as a special case of a path with length 2.

It should be noted, however, that a transition additionally has a trigger attribute and optionally an action attribute which a path does not have. Two transitions with identical pre- and post-states need not be identical. Two paths containing the same states in the same order are identical.

In this thesis, square brackets are used to enclose state sequences when these are paths and round brackets when they are transitions. This notation is ambiguous for transitions when these share pre- and post-states but have different trigger or action elements. Where this ambiguity may be relevant, the notation is expanded.

examples: paths: $[s_0, s_1, s_2, s_0]$, $[s_1, s_2]$, [].
transitions: (s_0, s_1), (s_1, s_2), (e, s_2, s_0, a).

Where s_0, s_1, s_2 are states, e is a trigger element and a an action.

The function *path* projects a transition to a path.

$$\begin{aligned}&\mathit{path}\colon \mathbf{R} :\rightarrow \mathbf{P^*}\\ &\mathit{path}(r) := \{ f\colon \mathbb{N} \rightarrow \mathbf{S} \mid\ f[0] = \mathit{pre}(r)\\ &\qquad\qquad\qquad\qquad\quad f[1] = \mathit{post}(r)\\ &\qquad\qquad\qquad\qquad\quad f[2] = \varnothing\\ &\qquad\qquad\qquad\qquad\quad \}\end{aligned}$$

The function *trans* returns all transitions of a path:

$$\begin{aligned}&\mathit{trans}\colon \mathbf{P^*} :\rightarrow 2^{\mathbf{R}}\\ &\mathit{trans}(p) := \{ r \mid \mathit{path}(r) \in \mathit{subpath}(p) \}\end{aligned}$$

3.3.5. Continuing paths

A *continuing path* is a path whose last element is a dead-end or whose length is infinite.

$$\mathbf{P\#} :\subseteq \mathbf{P^*} \mid \forall p \in \mathbf{P\#} : (\mathit{length}(p) \neq \infty) \Rightarrow (p[\mathit{length}(p)\text{-}1] \in \mathbf{D})$$

3.3.6. Full paths

A *full path* is a continuing path whose first element is s_{init}.

$$\mathbf{P} :\subseteq \mathbf{P\#} \mid \forall p \in \mathbf{P} : p[0] = s_{init}$$

3.3.7. Reacheable states

The set of reacheable states **SR** is the set of states visited by the set of full paths.

$$\mathbf{SR} := \{ s \in \mathbf{S} \mid \exists p \in \mathbf{P} : s \in \mathit{visited}(p) \}$$

3.3.8. Loops

A loop is a path of length > 1 in which the first state is the same as the last state and no other state in between is equal to the first state.

$$\mathbf{LP} :\subseteq \mathbf{P^*} \mid \forall p \in \mathbf{LP} : \quad (\mathit{length}(p) > 1) \wedge (\mathit{length}(p) \neq \infty) \wedge \\ (\forall n \in \mathbb{N} : n > 0 \Rightarrow (p[n] = p[0] \Leftrightarrow n = \mathit{length}(p)\text{-}1))$$

3.4. Kripke Structures

3.4.1. Introduction

A *Kripke structure* [18] is a formal description of a state machine. Kripke structures are named after the mathematician and philosopher Saul Kripke [22], who besides his better known work on the logics of recognition and naming and the philosophy of language, made notable contributions to formal and modal logics. Today, Kripke structures are the basis for model checking.

3.4.2. Atomic propositions

Having designed a system to conform to the principles introduced above, we wish to verify whether that system conforms to the specification it was designed to fulfil. For this, we must first decide for every state whether or not it fulfils certain conditions.

For this purpose we define **AP**, a set of atomic propositions which can be true or false in every state.

$\mathbf{AP} := 2^{\mathbf{S}}$

An examples of such an atomic proposition may be '*the door is not closed*'. The proposition returns true for all states where this is the case and false for the others.

3.4.3. Kripke structures

A Kripke structure is a structure consisting of states, transitions between the states, and labels attached to the states telling us which conditions they fulfil.

A Kripke structure is defined by the tuple:

$K := (\mathbf{S}, \mathbf{R'}, s_{init}, \mathbf{L})$

K is the Kripke structure.
S is the atomic set of system-states (as in section 3.2.2.).
R' is the transition relation: $\mathbf{R'} :\subseteq \mathbf{S} \times \mathbf{S}$.
s_{init} is the initial system-state (as in section 3.2.2.).
$\mathbf{L} : \mathbf{S} \rightarrow 2^{\mathbf{AP}}$ is a function that labels each state with the set of atomic propositions that are true in that state.

In Kripke structures, the transition relation must be *total*. This means that for every state s ∈ **S**, *suc*(s) ≠ ∅.

figure 3.3. State diagram of an example system

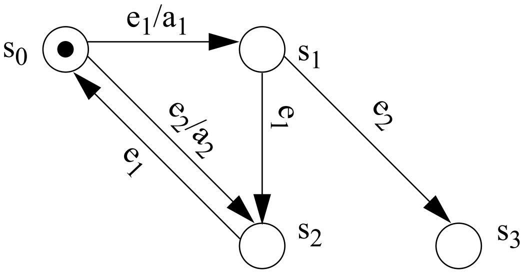
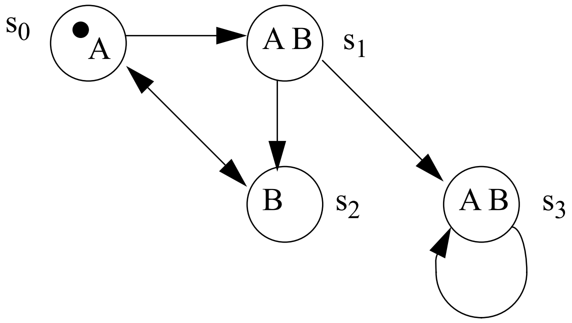


figure 3.4. Kripke structure diagram of above example



The example above shows a state diagram and the corresponding Kripke structure diagram. In this transformation, the labelling of transitions with events and actions is dropped. New information is introduced with the atomic propositions A and B.

In the Kripke structure diagram, the atomic proposition label is placed inside the circle representing the state.

In this example, A is fulfilled in states s_0, s_1 and s_3 and B is fulfilled in states s_1, s_2 and s_3.

The additional transition (s_3, s_3) is introduced to achieve the completeness of the transition relation required for Kripke structures. Dead ends do not appear in Kripke structures.

3.4.4. Propositions

Atomic propositions (section 3.4.2.) are true or false in a given state, regardless of the position of that state on a path. '*The door is not closed*' is an atomic proposition, but '*the door was once closed*' is not an atomic proposition because it may or may not be true in a given state. To be able to say whether such a proposition is true, we require knowledge of the path.

We define the set of path propositions which can be true or false at any position of any (full) path.

$\mathbf{PP} := 2^{\mathbf{P}\times\mathbb{N}}$

Because reacheable states can be mapped as positions on paths:

$\mathbf{AP} \cap 2^{\mathbf{SR}} \subseteq \mathbf{PP}$

The function *propCount* counts the number of times a proposition holds on a path:

$$propCount\text{: } \mathbf{PP} \times \mathbf{P} \rightarrow \mathbb{N} \cup \{\infty\}$$

$$propCount(\text{pr, p}) := \sum_{n=0}^{\text{length(p) - 1}} (\text{if pr(p, n) : 1 else : 0}).$$

Example: In the example of figure 3.4. on the path *propCount*(A ∧ ¬B, $[s_0, s_1, s_3]$) = 1.

When, in the notation, a path proposition is replaced its definition, a symbol may be required to denote the position on the path. This is noted as a subscript to *propCount*:

$$\forall(\text{y,z}) \in \text{symbols}^2\text{:}\quad propCount_{\text{y}}\text{: } \mathbf{PP} \times \mathbf{P} \rightarrow \mathbb{N} \cup \{\infty\}$$

$$propCount_{\text{y}}(\text{pr(z, y), p}) := \sum_{n=0}^{\text{length(p) - 1}} (\text{if pr(p, n) : 1 else : 0}).$$

Example: $propCount_{\text{n}}$(p[n] = p[n+2], $[s_0, s_2, s_0, s_1, s_3]$) = 1.

3.4.5. Property

A property is a boolean mapping which is true or false for the Kripke structure as a whole. We call **GP** the set of properties.

$\mathbf{GP} := 2^{\{\text{true, false}\}}$

3.5. Some properties

3.5.1. Note on notation

The properties defined below are defined especially for the purpose of this thesis and use in the CIP model checker. For the common definitions, the reader is referred to section 4.3.

3.5.2. Next

A *next* property requires that if a state fulfils a proposition, a, then its successors all fulfil another proposition, b.

next : $\mathbf{PP} \times \mathbf{PP} :\rightarrow \{\text{true, false}\}$
next(b, a) := $(\forall s \in \mathbf{SR}: (a(s) \Rightarrow (\forall s' \in suc(s): b(s))))$

3.5.3. Invariance

An *invariance* property requires that a proposition is fulfilled by all states on all full paths.

invariant: $\mathbf{CP} :\rightarrow \{\text{true, false}\}$
invariant(a) := $(\forall s \in \mathbf{SR}: a(s))$

Examples: in figure 3.4. the property *invariant*($A \vee B$) is fulfilled. The condition *invariant*(B) is not fulfilled (counter-example: s_0).

Practical example: A lift may never be in a state fulfilling the proposition ‘cabin moving and door open’.

3.5.4. Reacheability

A *reacheability* property requires that every reacheable state fulfilling the proposition a is the first state of at least one path which visits a state fulfilling b.

reacheable: $\mathbf{CP} \times \mathbf{CP} :\rightarrow \{\text{true, false}\}$
reacheable(b, a) := $(\forall s \in \mathbf{SR}: (a(s) \Rightarrow$
$(\exists p \in \mathbf{P}: (s = first(p) \wedge (\exists s' \in visited(p): b(s'))))))$

Examples: in figure 3.4. the property *reacheable*($A \wedge B, \neg A$) is true. The property *reacheable*($\neg B$, A) is false (counter-example: there is no path from s_3 to any state fulfilling $\neg B$).

Practical example: when button x has been pressed, action y can be performed. *Reacheability* is often more useful in it’s negated form: when the alarm button has been pressed, no dangerous actions will be performed. This is fulfilled when the following property is failed: when the alarm button is pressed, a dangerous action can be performed.

The *reacheability* property also exists with one argument. In that case the second argument is assumed to be universally true.

$\textit{reacheable}$: $\mathbf{CP} :\to \{\text{true, false}\}$
$\textit{reacheable}(b) := (\forall s \in \mathbf{SR}: (\exists p \in \mathbf{P}: (s = \textit{first}(p) \wedge (\exists s' \in \textit{visited}(p): b(s')))))$

Example: in figure 3.4. the property *reacheable*(B) is fulfilled. The property *reacheable*(¬B) is not fulfilled.

Practical example: the lift can reach the 4th floor.

Note: $\forall a \in \mathbf{AP}: \textit{invariant}(a) \Rightarrow \neg\textit{reacheable}(\neg a)$

3.5.5. Eventually

An *eventually* property requires that for every reacheable state fulfilling the proposition req, all continuing paths starting in that state visit a state fulfilling *ack*.

The property is also known as *request → acknowledge*, because every *request* must eventually be *acknowledged*.

$\textit{eventually}$: $\mathbf{CP} \times \mathbf{CP} :\to \{\text{true, false}\}$
$\textit{eventually}(\text{ack, req}) := (\forall s \in \mathbf{SR}: (\text{req}(s) \Rightarrow$
$\quad (\forall p \in \mathbf{P\#}: (s = \textit{first}(p) \Rightarrow \exists s' \in \textit{visited}(p): \text{ack}(s')))))$

Examples: in figure 3.4. the property *eventually*(A, ¬A) is fulfilled. The condition *eventually*(¬A, A) is not fulfilled.

Practical example: when the alarm button has been pressed, all dangerous activities will eventually cease.

The *eventually* property also exists with one argument. In that case the second argument is assumed to be universally true.

$\textit{eventually}$: $\mathbf{CP} :\to \{\text{true, false}\}$
$\textit{eventually}(\text{ack}) := (\forall s \in \mathbf{SR}: (\forall p \in \mathbf{P\#}: (s = \textit{first}(p) \Rightarrow \exists s' \in \textit{visited}(p): \text{ack}(s'))))$

Example: in figure 3.4. the property *eventually*(B) is fulfilled. The condition *eventually*(¬A) is not fulfilled.

Practical example: the system will always eventually return to its initial state.

3.6. Liveness requirement

By definition, all full paths fulfil the *liveness requirement*:

Liveness requirement: We require, that whatever the state of a Kripke structure, a further transition will always be triggered.

(this statement was already subsumed when requiring the transition relation to be total in section 3.4.3. on page 42)

For a full discussion and justification of the liveness requirement see appendix A.1. but to present a summary example, we study the system of figure 3.4. (page 43). It is apparent, that if the system is in state s_2, then the next system transition will change its state to s_0.

$\forall(p, n) \in \mathbf{P} \times \mathbb{N} : p[n] = s_2 \Rightarrow p[n+1] = s_0$

It could, however, be argued that in a real system this is not necessarily the case. If no further transition is triggered, then the system can remain in state s_2 forever. The liveness requirement rules out this possibility.

3.7. Fairness

3.7.1. Fairness in this project

For a more full discussion of fairness in this project, what exactly it is and why it is needed, the reader is referred to appendix A.3.

There are many different concepts of fairness in use. To read how some of these differ from that used in this thesis, the reader is referred to sections 4.4. and 4.5.

3.7.2. Fairness constraints, fair and unfair transitions

A fairness constraint is an attribute which can be associated with one or several transitions.

F is the set of all fairness constraints.

$\mathbf{F} :\subset 2^{\mathbf{R}}$

fairness is a function returning the fairness constraints attributed to a transition.

fairness: $\mathbf{R} :\rightarrow 2^{\mathbf{F}}$
fairness(r) $:= \{f \mid r \in f\}$

A transition with at least one fairness constraint is called a *fair transition*. A transition which is not fair is called an *unfair transition*.

3.7.3. Fair paths

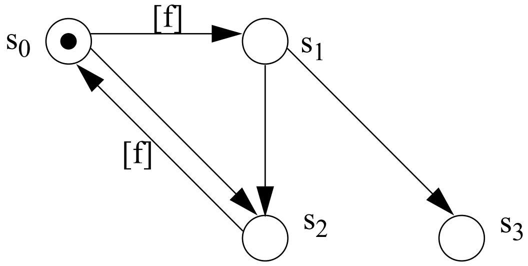
A fair path is a path for which the following statement holds:

If the path visits a state infinitely often and a transition with the fairness constraint f is enabled in that state, then the path traverses infinitely often a transition with the fairness constraint f.

Note that the above statement does not require the infinitely traversed transition to be identical to the one enabled in the infinitely traversed state.

Example:

figure 3.5. State diagram illustrating a fair path



The fairness constraint of a transition is labelled in sqaure brackets.

In the example of figure 3.5., the path p = [s_0, s_2, s_0, s_2, ...] infinitely visits the state s_0. The fair transition (s_0, s_1) is enabled in this state but never executed on this path. The path is nevertheless fair because it infinitely executes another transition (s_2, s_0) which has the same fairness transition.

fairpaths: $2^{\mathbf{P^*}} :\rightarrow 2^{\mathbf{P^*}}$
fairpaths(P_0) := $\{p \in P_0 \mid \forall s \in$ *visited*(p): (*occurrences*(s, p) $= \infty \Rightarrow$
$(\forall r \in$ *enabled*(s): $\forall f \in$ *fairness*(r): $\exists t \in$ f: *occurrences*(t, p) $= \infty))\}$

An unfair path is a path which is not a member of *fairpaths*(**P***).

Note that when **F** is empty, all paths are fair.

3.7.4. Fair loops

A fair loop is a loop which is a fair path if traversed infinitely.

FLP := **LP** $\cap$ $\{p \mid \forall s \in$ *visited*(p): $\forall r \in$ *enabled*(s): $\forall f \in$ *fairness*(r): $f \cap$ *trans*(p) $\neq \varnothing\}$

An unfair loop is a loop which is not a member of **FLP**.

Note that when **F** is empty, all loops are fair.

3.7.5. Some new definitions

Based on the definition of fair paths of section 3.7.3. the following shorthand notations are defined:

FP* := *fairpaths*(**P***)
FP# := *fairpaths*(**P#**)
FP := *fairpaths*(**P**)

3.7.6. Some basic properties of fair and unfair paths

Every finite path is fair (follows from definition).

By consequence, every unfair path is infinite.

If a state is reacheable by an unfair path it is also reacheable by a fair path (it is reacheable by a finite sub-path and any finite path is fair).

All states reacheable by an unfair full path are also reacheable by a fair full path (see proof in Appendix A, section A.4.1. on page 146)

3.7.7. Fair Kripke Structures

To be able to make use of fairness in Kripke structures, we must include fairness in the definition of Kripke structures. A fair Kripke structure is defined by the tuple:

K := (**S**, **R'**, s_{init}, **L**, **F**)

K is the Kripke structure.
S is the atomic set of system-states (as in section 3.2.2.).
R' is the transition relation: **R'** : $\subseteq$ **S** $\times$ **S** $\times$ **F**.
s_{init} is the initial system-state (as in section 3.2.2.).
L : **S** $\rightarrow 2^{\mathbf{AP}}$ is a function that labels each state with the set of atomic propositions that are true in that state.
F is the atomic set of fairness constraints.

All functions and properties of normal Kripke structures (see section 3.4. page 42) apply.

3.7.8. Applying fairness to the *eventually* property

In fair Kripke structures we restrict the definition of the *eventually* property (see section 3.5.5.) by replacing **P#** by **FP#**.

eventually: $\mathbf{CP} \times \mathbf{CP} :\rightarrow \{\text{true, false}\}$
eventually(ack, req) := $(\forall s \in \mathbf{SR}$: (req(s) $\Rightarrow$
$(\forall p \in \mathbf{FP\#}$: (s = *first*(p) $\Rightarrow \exists s' \in$ *visited*(p): ack(s')))))

eventually: $\mathbf{CP} :\rightarrow \{\text{true, false}\}$
eventually(ack) := $(\forall s \in \mathbf{SR}$: $(\forall p \in \mathbf{FP\#}$: (s = *first*(p) $\Rightarrow \exists s' \in$ *visited*(p): ack(s'))))

4. Similar Work

4.1. Purpose

In this chapter, approaches to Model Checking concerned with embedded systems are presented. The history of Model Checking is discussed briefly and common temporal logics such as CTL and LTL are introduced. The concepts of fairness used elsewhere are explained and these are compared to that adopted in this thesis. Further background on the fairness comparison can be found in appendix B. This chapter presents two examples of widely used model checkers (SPIN and SMV) with small samples of the style of their input code. Finally, *State Charts*, a software development tool which is often considered a rival of CIP is introduced and the differences between this and CIP, especially relating to Model Checking ability are discussed.

4.2. A brief history of Model Checking

The discipline of Model Checking is comparatively new. First major steps in this direction occurred in the 1970's with the development of *temporal logic* with respect to its applicability to computer programs by Burstall [27] and Kröger [16]. Temporal logic is used to describe the order of events without introducing time. During this period, automatic verification was not yet possible. Instead verification was done manually using rules and axioms making it cumbersome and inapplicable to many real problems.

Later in the 1970's Pnueli [40] advanced this method further by using it to reason about concurrent systems. By the early 1980's it was being used to analyse sequential circuits [13],[36].

Model checking algorithms paving the way towards automatic verification were introduced in 1981 by Clarke and Emerson [7],[11]. These names have since become closely associated with Model Checking. There is hardly any literature in the field that does not reference them and their contribution provides the basis for all subsequent development. Clarke and Emerson's work is especially focused on the branching time logic CTL (this will be discussed in section 4.3.3.).

At the same time progress was also being made in handling fairness [23]. Fairness has to be considered where the possibility arises of certain transitions or sets of transitions being repetitively executed to the exclusion of other enabled transitions. It is especially important to be able to handle fairness without greatly increasing the complexity of the algorithm. Some common definitions of fairness (differing from that used in this project) are introduced and discussed in sections 4.4. and 4.5.

The power of Model Checking algorithms was long confined by the limits of computing power, hence its relatively recent transition from a rather theoretical field to one that is commonly used as part of design cycles.

A large variety of model checkers are available today, ranging from commercial products to open source and shareware contributions. Two of these, SPIN and SMV are briefly discussed in sections 4.6. and 4.7.

4.3. Temporal logic

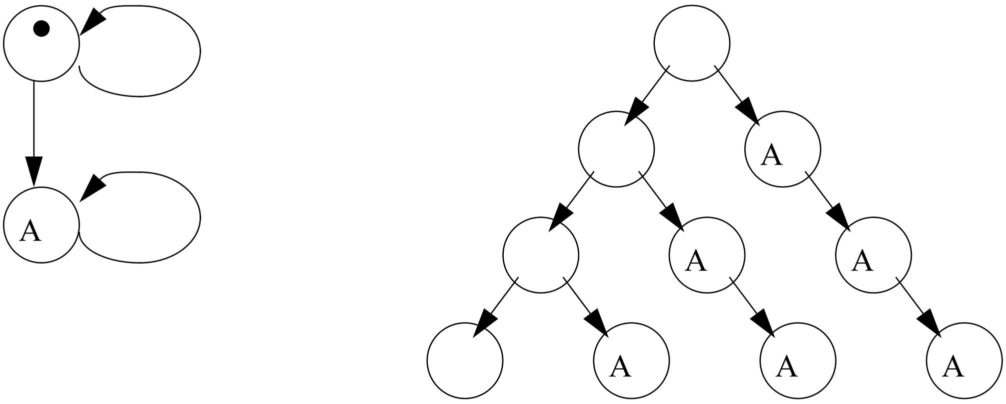
4.3.1. What is temporal logic?

Temporal logic is a logic that looks at the order in which events occur without necessarily looking at time.

4.3.2. Computation Tree Logic (CTL*)

Computation tree logic (CTL*) is a logic describing *computation trees*. A computation tree is basically a tree created by unwinding the Kripke structure starting from its initial state. A computation tree thus contains all full paths of the system.

figure 4.1. Simple Kripke structure (left) and the corresponding computation tree (right)



CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are [18]:

A : for all computational paths
E : for some computational path.

The temporal operators are [18]:

X : next time, requires that the property holds in the next state of the path.
F : in the future, requires that a property will hold in some future state of the path.
G : globally, requires that a property holds in the present and for every future state of the path.
U : until, requires that the first property holds on all future states of the path until the second property holds. The second property holds eventually on the path.
R : release, requires that the second property holds along the path up to the first state where the first property holds. The first property must not hold eventually.

A *state formula* is a formula which holds in a given state.

A *path formula* is a formula which holds along a given path of the Kripke structure.

4.3.3. CTL

CTL is a subset of CTL*. The subset is formed by the following syntax restrictions

- temporal operators applied to state formulas are path formulas.
- every path quantifier must be followed by a temporal operator.

Examples of CTL expressions:

for all these examples, (a, b) ∈

2

CTL notation	**notation used in this project**
AG(a → ***AX*** b)	*next*(b, a)
AG a	*invariant*(a)
EF a	*reacheable*(a, s_{init})
AG(a → ***EF*** b)	*reacheable*(b, a)
AG(***EF*** a)	*reacheable*(a)
AF a	*eventually*(a, s_{init})
AG(a → ***AF*** b)	*eventually*(b, a)
AG(***AF*** a)	*eventually*(a)

4.3.4. LTL

Linear Time Logic (LTL) is another subset of CTL*. It has the following restrictions:

- if p ∈ **AP**, then p is a path formula.
- if f and g are path formulae, then all boolean combinations of f and g or temporal operators applied thereto are path formulae.

It follows that the only state formulae permitted in LTL expressions are atomic propositions.

4.4. Strong and weak fairness

4.4.1. Fairness

In section 3.7. (page 47) fairness was introduced as it is understood in this project. Fairness disallows paths which repeat certain transitions infinitely and starving others which could also be executed. In that section it was mentioned that the definition of fairness used here is far from universal. Some of the alternative definitions will be introduced here and their applicability to that used in this project discussed.

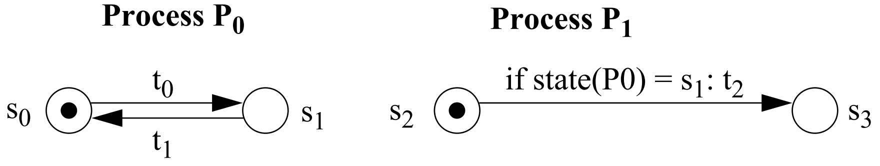
4.4.2. Definitions

To distinguish different forms of fairness, fairness can be said to be *strong* or *weak*. In *Software Reliability Methods* [30], Doron A. Peled makes the following definitions:

> **Strong transition fairness**: *Rules out a* [path] *where there is a transition that is enabled on* [the path] *infinitely many times but is executed only a finite number of times.*
>
> **Weak transition fairness**: *Rules out a* [path] *if for some state* s *on* [the path] *and forever there is a transition that is enabled, but is never executed after* s.
>
> **Strong process fairness**: *Rules out a* [path] *if transitions of process P_i are enabled on* [the path] *infinitely many times, but are executed finitely many times.*
>
> **Weak process fairness**: *Rules out a* [path] *if for some state* s *on it and forever there is at least one transition of some process P_i that is enabled, but no transition of P_i is ever executed after* s.

4.4.3. Example of strong transition fairness

figure 4.2. Example of strong transition fairness.



The transition labelled *if state(P0) =* $s_1 : t_2$ is a conditional transition[1]. The if statement is a guard. The transition t_2 can only be triggered when the formula returns *true*.

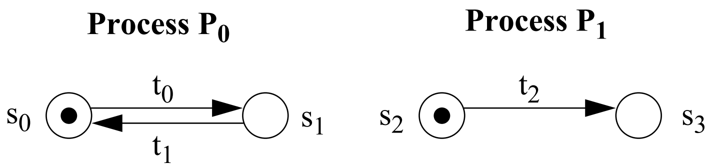
The example of figure 4.2. has two processes. s_0 and s_1 are states of process P_0. s_2 and s_3 are states of process P_1. t_0, t_1 and t_2 are transitions of these processes. t_2 is a conditional transition, it can only be enabled when a condition is fulfilled. The processes are each a finite state machine. They can be combined to a *product finite state machine* whose states are *state vectors* each containing exactly one state from every component process. State vectors and the combination of processes will be discussed more fully in section 5.2.

Let us consider the full paths of the product finite state machine formed by the example of figure 4.2. On every full path not executing t_2, t_2 is enabled infinitely many times. Strong transition fairness excludes all such paths.

In this example only one path is actually excluded by strong transition fairness. It is the full path infinitely executing t_0 and t_1 without executing any other transition.

4.4.4. Example of weak transition fairness

figure 4.3. Example of weak transition fairness.



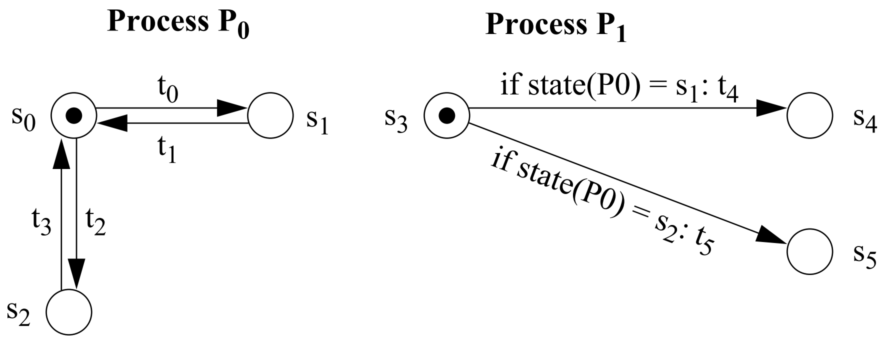
In the example of figure 4.3. on every full path not executing t_2, t_2 remains enabled forever. Weak transition fairness excludes all such paths.

Note that all paths permitted by strong transition fairness are also permitted by weak transition fairness but not vice-versa (We say that strong transition fairness is *stronger* than weak transition fairness).

[1] In CIP, a conditional transition would require an alternative for the ELSE case (see section 2.6. page 28). In these examples the general case is discussed.

4.4.5. Example of strong process fairness

figure 4.4. Example of strong process fairness.

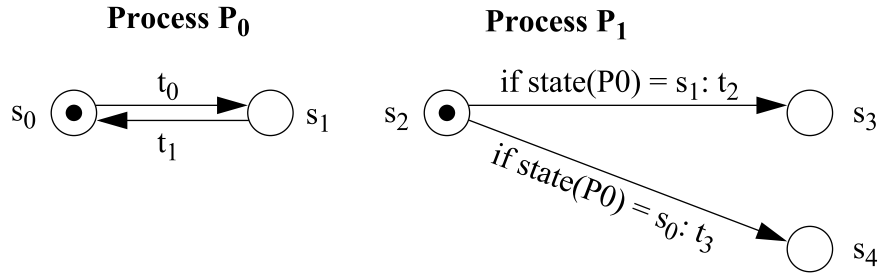


In the example of figure 4.4. on every full path not executing a transition of P_1, transitions of P_1 are enabled infinitely many times. Strong process fairness excludes all such paths.

Note that strong transition fairness is stronger than strong process fairness.

4.4.6. Example of weak process fairness

figure 4.5. Example of weak process fairness.

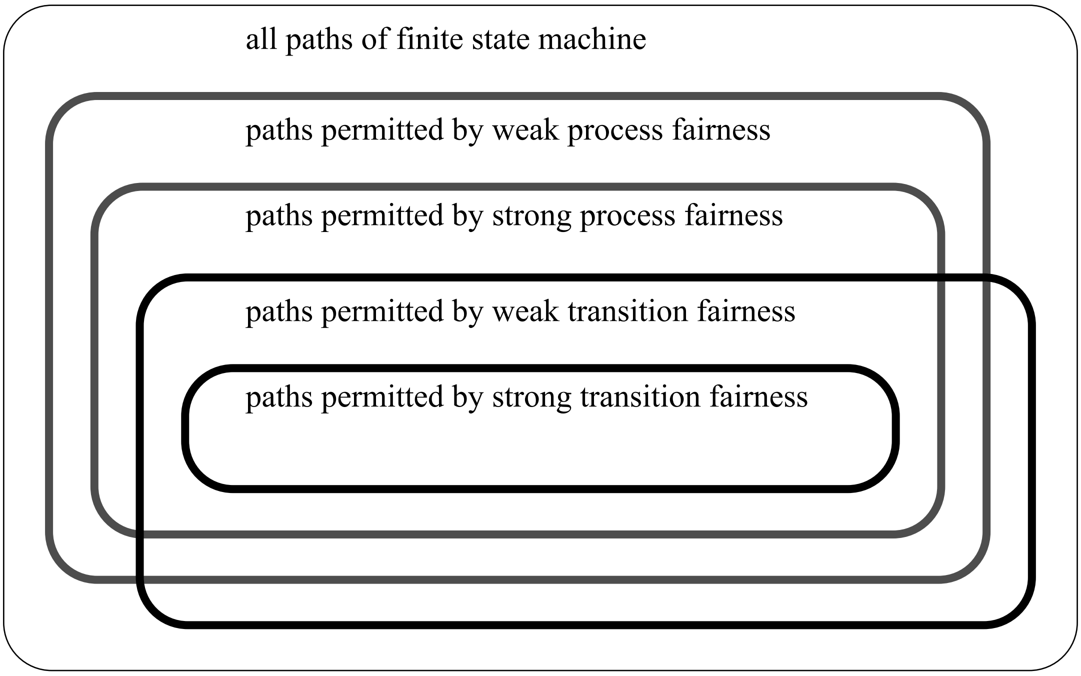


In the example of figure 4.5. on every full path not executing a transition of P_1, it holds that forever there is at least one transition of P_1 which is enabled. Weak process fairness excludes all such paths. In this example only one path is actually being excluded. It is the full path infinitely executing t_0 and t_1 without executing any other transition.

Note that strong process fairness is stronger than weak process fairness.

4.4.7. Hierarchy of fairness strength

figure 4.6. Hierarchy of fairness strength


all paths of finite state machine
paths permitted by weak process fairness
paths permitted by strong process fairness
paths permitted by weak transition fairness
paths permitted by strong transition fairness

4.4.8. The bearing of strong and weak fairness on this project

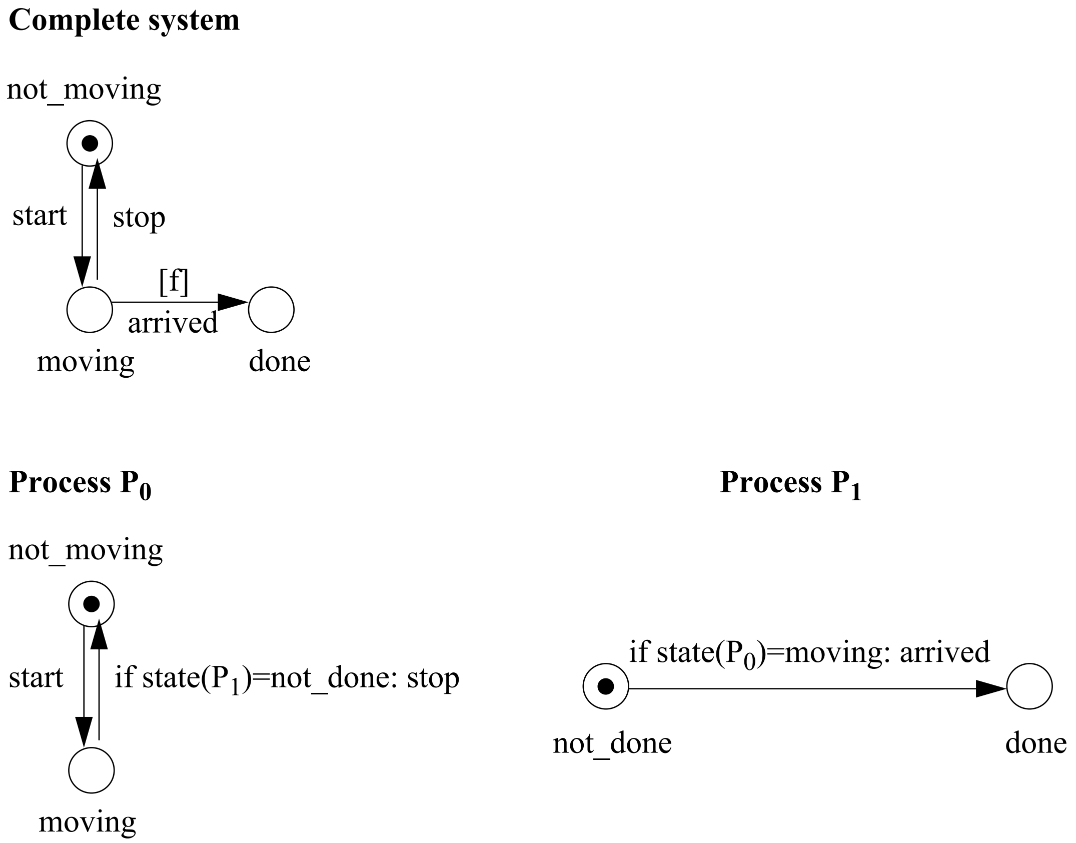
This approach to fairness is different to that adopted in this project.

In this project a fairness was desired which could be applied or withheld from individual transitions. For example, it is fair, that a door that is closing cannot continue doing so indefinitely and must at some time transit to a different state. It is equally fair that an emergency alarm button may remain in the state not pressed CIP Tool is primarily a development tool and not a Model Checking tool and the developer may wish to choose in which processes to place such transitions for reasons other than fairness.

In this project fairness is not based on processes. It is based on individual transitions. However, as the following examples show, some of the fairness requirements which can be expressed are the same.

In figure 4.7. (see also figure A.5. on page 143) a state machine is shown which is decomposed into two processes in order to express the required fairness properties. The decomposed system fulfils the same fairness requirements for both strong transition fairness and strong process fairness.

figure 4.7. Finite state machine and its decomposition into two processes.



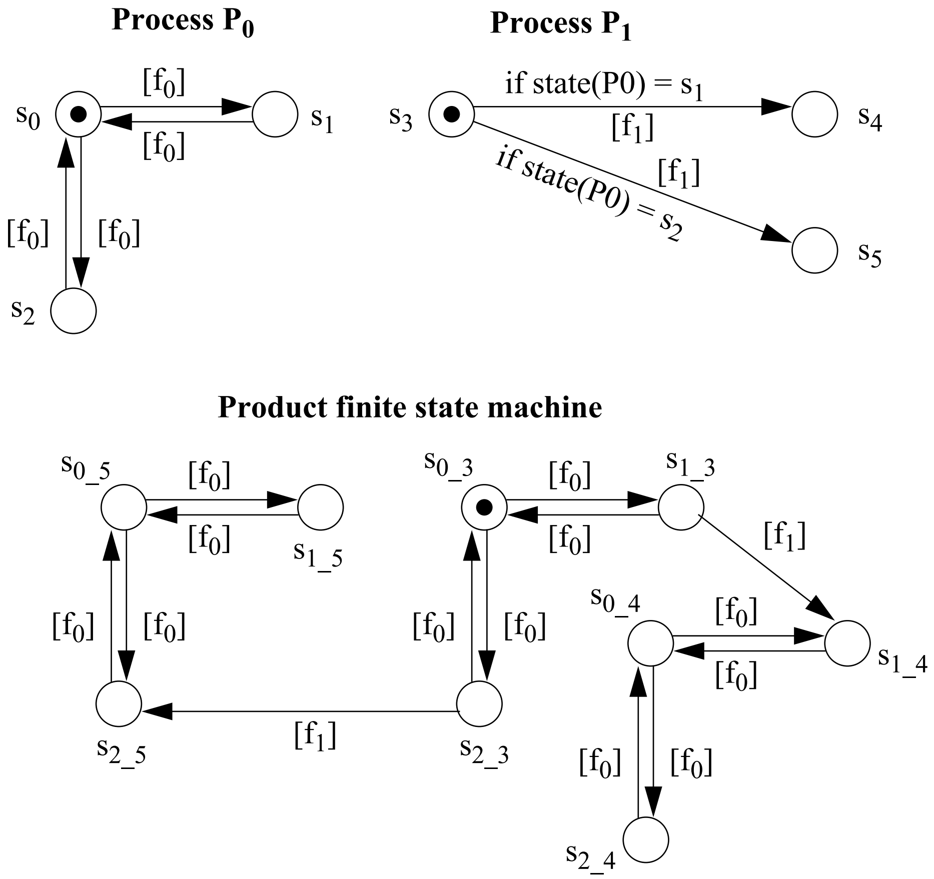
This decomposition is relatively straightforward. Studying that of figure 4.11. (page 62) however, it is clear that the decomposition of this figure is considerably more difficult and requires a large number of conditional transitions which confuse rather than enhance the user’s understanding of the system.

Decomposition according to fairness may not necessarily coincide with the functional decomposition which is the strength of the CIP model. An automatic recomposition to respect fairness is possible but would be quite cumbersome to implement.

The definition of strong and weak process and transition fairness requires that all processes and transitions are treated equally. In this project, this is not always desired. For example a transition corresponding to the pressing of an alarm button need never be executed even though it is infinitely enabled.

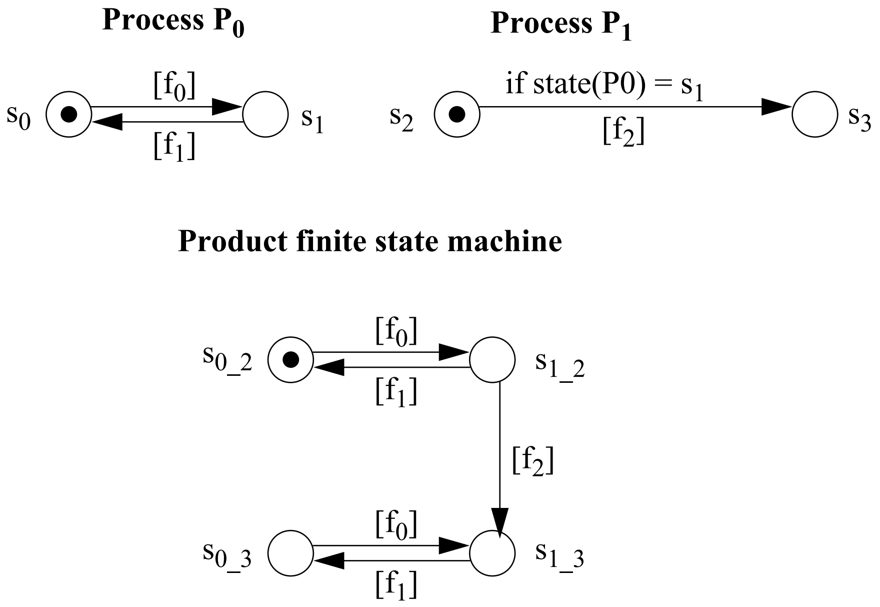
The definition of fairness used in this project (see section 3.7.) is thus more flexible. In this project, fairness rules out all paths where transitions with a fairness constraint are enabled infinitely many times but are only executed a finite number of times. This fairness can be given the same expressive power as strong process fairness if all transitions of every process have the same fairness constraint and transitions of different processes have different fairness constraints. This is illustrated in figure 4.8.

figure 4.8. Example of figure 4.4. with fairness constraints replacing strong process fairness to give same expressive power.



The expressive power of strong transition fairness can also be emulated with the fairness definition of this project. For this every process transition must be allocated a unique fairness constraint. This is illustrated in figure 4.9.

figure 4.9. Example of figure 4.2. with fairness constraints replacing strong transition fairness for same expressive power.



Note that not all fairness constraints shown are relevant. Fairness constraints f_0 of figure 4.8. and f_0 and f_1 of figure 4.9. do not lead to the rejection of any paths and so their omission has no effect on the fairness requirement being expressed.

Weak fairness cannot be modelled with the fairness concept of this project.

4.5. Fairness constraints as sets of states

4.5.1. Fairness constraints

In section 4.4. a concept of fairness was introduced which rejects paths not executing certain transitions under certain conditions. In this section a different approach is introduced. In this, a path is fair if a construct called a *fairness constraint* is true infinitely often. This fairness constraint is not the same as that of this project (introduced in section 3.7.2.).

E.M.Clarke, J.Grumberg and D.A.Peled in *Model Checking* [18] write:

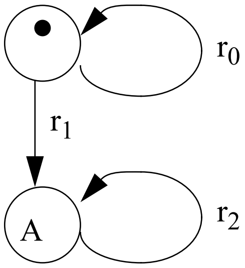
> *A* fairness constraint *[in CTL] can be an arbitrary set of states, usually described by a formula of [CTL] logic. If fairness constraints are interpreted as sets of states, then a fair path must contain an element of each fairness constraint infinitely often.*

(To avoid confusion, in this section we always write the CTL fairness constraint: *CTL fairness constraint*. The fairness constraint of this project we write *project fairness constraint*).

4.5.2. Applicability to the fairness concept of this project

A CTL fairness constraint prevents the repetition of certain sequences of states to the starvation of others. For example in the Kripke structure of figure 4.10. the infinite repetition of transition r_0 can be excluded by declaring the state labelled A as fairness constraint.

figure 4.10. Example Kripke structure

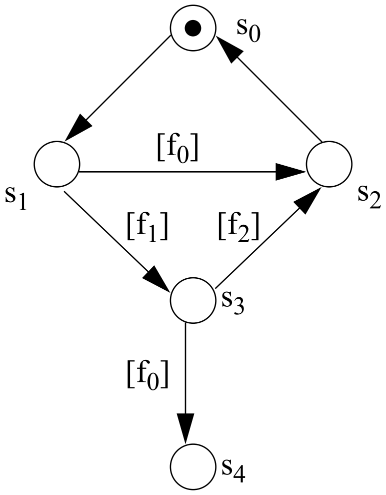


An attempt can be made to equate this definition of fairness to that used in this project (see section A.3.). The CTL fairness constraint as defined above could be the set of all post-states of transitions with the corresponding project fairness constraint.

The project fairness constraint solving the fairness problem of figure 4.10. would be to make transition r_1 fair.

For the state diagram of figure 4.11. the project fairness constraint f_0 would be represented as CTL fairness constraint by the set $\{s_2, s_4\}$, f_1 by $\{s_3\}$ and f_2 by $\{s_2\}$. With these CTL fairness constraints, the path infinitely repeating the loop (s_0, s_1, s_3, s_2, s_0, s_1, s_2, s_0) is fair (as it is with the shown project fairness constraints) because it contains states from all fairness constraints infinitely often.

figure 4.11. example finite state machine

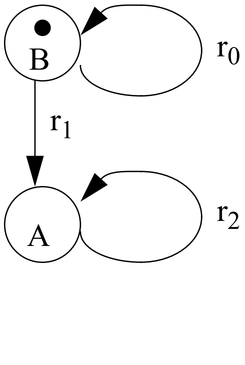


However, the path commencing (s_0, s_1, s_3, s_4) is not fair in the CTL definition although it is fair in the project definition. A workaround can be made by automatically including all dead ends in all CTL fairness constraints. In this case the above path is fair (remember that in a Kripke structure dead ends have transitions leading to themselves).

However, if s_4 were not a dead end we would not have this possibility. The definition used in this project can thus not always be transformed to match that of the definition introduced above.

An additional problem is that we can easily create fairness constraints which do not permit any paths at all if the definition is applied strictly. For example in figure 4.10. we could place the state marked A in one CTL fairness constraint and the initial state in another (illustrated in figure 4.12.). No path can visit both infinitely. This problem can be avoided in most model checking applications by assuming the user does not make unreasonable demands. In the CIP-model checker, however, it was decided to choose a transition and not process oriented approach to fairness as this is closer to the user's intuitive approach to CIP modelling. In CIP, no unreasonable fairness constraints can be defined.

figure 4.12. A finite state machine with unreasonable fairness constraints A and B



4.5.3. Fairness constraints as CTL formulas

The same source (as quoted in section 4.5.1.) [18] continues, however:

> *If fairness constraints are interpreted as CTL formulas, then a path is* fair *if each constraint is true* infinitely often *along the path.*

This definition is more general than that of section 4.5. It can be made to include the previous definition if the set of states *S* that forms the CTL fairness constraint in that definition is translated into a CTL expression.

4.5.4. CTL and project fairness constraints

Can the project approach to fairness also be expressed in this form? Does a CTL constraint exist so that a path of the Kripke structure is fair only if that constraint is true infinitely often along its length?

The definition of a fair path in this project (see section 3.7.3.) states that a path is fair only if the following statement holds:

> *If the path visits a state infinitely often and a transition with the [project] fairness constraint f is enabled in that state, then the path traverses infinitely often a transition with the [project] fairness constraint f.*

It follows that for every fair path fp and every fairness constraint f:

- no states where transitions with f are enabled is infinitely visited by fp
- or at least one transition with f is infinitely executed along fp.

Both these requirements are path requirements and so if a corresponding CTL expression exists then it must be a path formula.

However, a path formula holds for a path and not for states along the path. It is not possible to create a strict CTL state formula suitable for use as a CTL fairness constraint.

If we allow CTL* style expressions, however, making statements about a state at a given position on a given path rather than about a state or a path in general, an expression can be constructed. This is demonstrated in appendix B.1.

4.6. The model checker SMV

4.6.1. Introduction

SMV (*Symbolic Model Verifier*) is a model checking program used to check that finite state systems satisfy CTL specifications. SMV was first described by McMillan in 1993 [35]. SMV is a text based model checker, *i.e.* the input and output are text.

4.6.2. An example

The following code is a simple example of input code for SMV. It describes a basic attempt to write a mutual exclusion program for two processes and tests the proposed solution for simple properties.

figure 4.13. Sample SMV input code

```
MODULE main --two process mutual exclusion program

VAR
sp0: {noncritical, trying, critical};
sp1: {noncritical, trying, critical};
p0: process prc(sp0, sp1);
p1: process prc(sp1, sp0);

FAIRNESS !(s0 = critical)
FAIRNESS !(s1 = critical)

SPEC AG((sp0 = trying)-> AF(sp0 = critical))
SPEC AG((sp1 = trying)-> AF(sp1 = critical))
SPEC AG(!((sp0 = critical)&(sp1 = critical)))

MODULE prc(s0, s1)
ASSIGN
init(s0) := noncritical;
next(s0) :=
  case
    (s0 = noncritical) : {trying,noncritical};
    (s0 = trying) & (s1 != critical) : critical;
    (s0 = critical) : {critical, noncritical};
    1 : s0;
  esac;

FAIRNESS running;
```

The following explanation is intended to aid in the comprehension of the above code. For a full definition of the SMV syntax, the reader should refer to the appropriate documentation [29],[35].

Lines 2 to 6 declare variables. Variables `sp0` and `sp1` are atomic variables and their possible values are listed in brackets with the declaration. These variables are used as states of processes.

Variables `p0` and `p1` are processes. The keyword `process` states that *interleaved composition* is used. This means that a step of the execution of the program is a step of the execution of exactly one component. Without this keyword, *synchronous composition* is assumed, which means that a step of the execution of the program is a step of the execution of all components. `prc` is a `MODULE` (representing a process type) to be defined later. `p0` and `p1` are therefore instances of `prc`. `sp0` and `sp1` are used as parameters for `p0` and `p1`.

Lines 7 and 8 are (CTL) fairness declarations of the type presented in section 4.5. The statements `!(s0 = critical)`and `!(s1 = critical)` must be fulfilled infinitely often. Therefore neither `sp0` nor `sp1` may retain the value `critical` infinitely long.

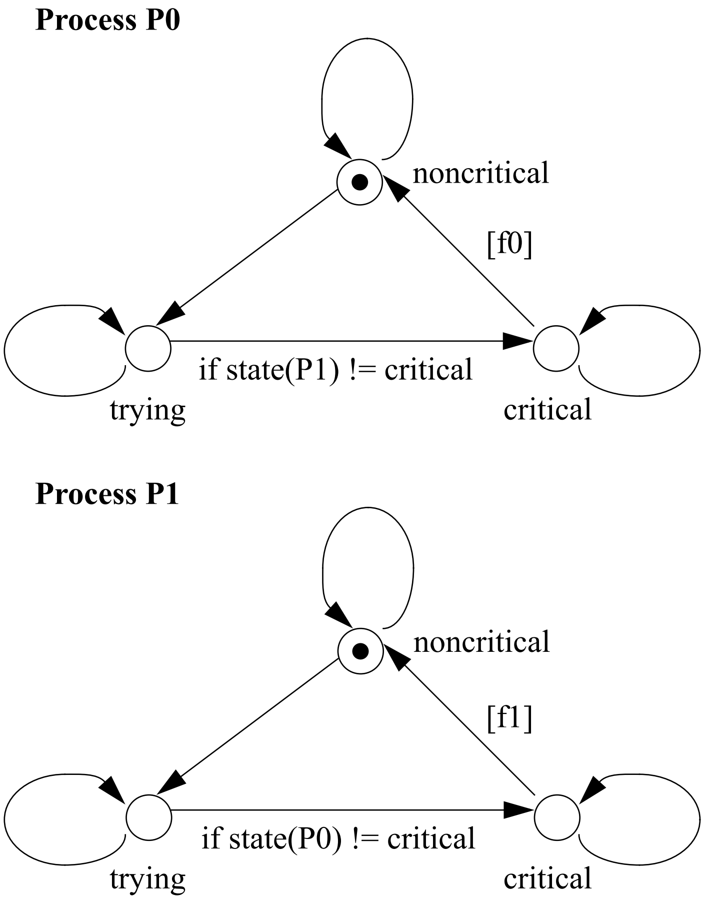
Lines 9 to 11 specify the properties to be checked. These are written in CTL notation. The first two require (for each process separately, although for reasons of symmetry only one such statement is really required) that when a process is in state `trying` then it will eventually enter state `critical`. The third statement requires that the two processes may never be in state `critical` simultaneously.

Lines 12 to 21 specify the behaviour of processes of type `prc`. The command `init(s0):= noncritical` sets the initial state of the process to `noncritical`. The following `next(s0)` declaration lays down the possible transitions of the process and associated conditions. The transitions of lines 17 and 19 are non-deterministic, that of line 18 is deterministic.

The conditions of the `case` structure (lines 16 to 21)

Line 22 is another fairness declaration. The proposition `running` requires that transitions of all processes are executed infinitely often. This is not strictly the same as strong process fairness because strong process fairness requires that transitions of a process be executed infinitely often only if transitions of the process are enabled infinitely often. However, in this example the effect is the same.

figure 4.14. The processes of the mutual exclusion program



4.7. The model checker SPIN

4.7.1. Introduction

SPIN (Simple Promela INterpreter) is an LTL based model checker. It has been developed by G. Holzmann [32] at Bell Labs since 1980 and has been freely available since 1991. The adjective *simple* is becoming increasingly stretched as the tool continues to evolve and gain increasingly sophisticated features. SPIN is widely used for software verification.

The input for SPIN is in an executable language called PROMELA (PROto MEta LAnguage). This language is based on C syntax and has a considerable expressive power.

SPIN is also available with an intuitive graphical user interface called Xspin.

4.7.2. SPIN and CIP

Holzmann has recently announced the possibility of translating C-code directly into PROMELA. SPIN thus holds in common with the CIP model checker that the user is guaranteed that the verification results apply to the final code and not just to the model.

This and the graphical user interface Xspin could lead the user to ask whether SPIN is not an alternative to the CIP model checker. Could the C-code generated by CIP Tool not be fed to SPIN for verification so eliminating the need for an integrated model checker?

The obvious similarities of the two tools should not cloud the fact that they operate on different levels of abstraction. SPIN can extract PROMELA directly from C-code. Maybe one day there will be a tool to extract PROMELA directly from machine code. If this were to happen one could be justified in asking why PROMELA should be extracted from machine code when the C-code is also available and the abstraction from the model is smaller at C-code level. Likewise one could ask why extract PROMELA from generated C-code when the original model is available and can be checked.

Maybe one day the CIP model checker will offer the possibility of generating PROMELA from the model, or other meta-codes for other model checkers to be able to check. For the moment CIP will remain a standalone system not requiring the user to learn any other tools.

4.7.3. A PROMELA example

figure 4.15. Sample PROMELA code

```
1  byte sp[2];
2  byte mutex=0;

3  proctype prc(bit i, j) {
4    sp[i] = 0;
5    do
6    ::
7      if
8      :: (sp[i] == 0) -> sp[i] = 1;
9      :: (sp[i] == 0);
10     :: (sp[i] == 1) ->
11       atomic {
12         if
13         :: (sp[j] != 2) -> sp[i] = 2; mutex++
14         :: (1);
15         fi;
16       }
17     :: (sp[i] == 2) -> sp[i] = 0; mutex--;
18     :: (sp[i] == 2);
19     fi;
20   od;
21 }

22 proctype monitor() {
23   assert (mutex !=2);
24 }

25 init {
26   atomic {
27     run prc(0,1);
28     run prc(1,0);
29     run monitor();
30   }
31 }
```

The following explanation is intended to aid in the comprehension of the above code. For a full definition of the SPIN syntax, the reader should refer to the appropriate documentation [32],[38].

Lines 1 and 2 declare global variables. `sp` is a byte array of size 2 and `mutex` is a byte which is initialised with the value 0.

Lines 3 to 21 declare a procedure type. In SPIN procedures are executed in *interleaving* mode. Only one command can be executed at any one time but scheduling is non deterministic.

Line 4 initialises `sp[i]` with value 0.

Lines 5 to 20 are enclosed in a `do` loop. This is infinitely repeated.

Lines 7 to 19 form an `if` statement. If one guard before the `->` sign returns true then the commands after it are executed. If several guards return true then it is decided non-deterministically which set of commands is executed. If no guard returns true then the execution of that process is halted until at least one guard returns true.

The `atomic` statement (lines 11 to 16) ensures that the command sequence within it cannot be interrupted. No command of any other process may be executed until this sequence is complete.

The `assert` statement (line 23) checks that the boolean evaluation within it is true. If this evaluation is failed, SPIN terminates with an error.

Lines 25 to 31 call and run the three processes.

4.7.4. LTL properties in SPIN

In addition to using the `assert` statement to check properties, LTL formulae can be tested. Some examples are:

`[]P`	always P
`<>P`	eventually P
`P U Q`	P is true until Q becomes true.

Xspin offers possibilities to edit such statements.

4.8. Model Checking and Statecharts

4.8.1. Introduction

Statecharts is a software design formalism built around UML (Unified Modelling Language) [41] developed by D. Harel (ca. 1984-87) [21] [34]. Similarly to CIP, these models can be designed graphically and compilable code can be generated from the models.

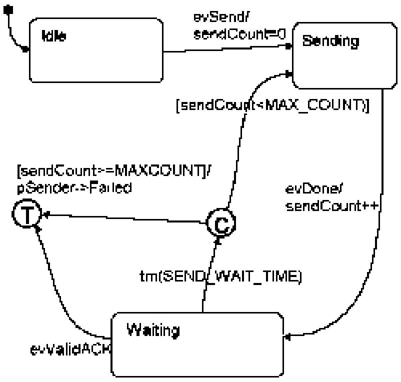
In fact many potential users who are not familiar with one or both systems assume they are basically different tools for the same job and ask whether they are not interchangeable or how easy it is to transfer models from one to the other. In this section, the similarities and differences will be discussed as will the effects of these differences on model checking.

4.8.2. Statecharts in a nutshell

A simple Statechart process is shown in figure 4.16. The process has three states. These are represented by boxes. The initial state (*Idle*) is identified by an arrow leading to it from a dot. The other arrows are transitions analogue to those of state diagrams (see section 3.2.6. on page 38). They are labelled with the trigger element followed by any action which may be taken. In contrast to CIP, where system transitions can only be triggered by events, in Statecharts they can also be triggered by boolean guards. In such a case, a transition will not execute as long as a boolean value or a function returns *false*, but executes when this returns *true*.

In figure 4.16. transitions also connect to objects marked with the letter C or T. The letter C denotes a condition. Depending on whether a condition is fulfilled or not, one or another transition is executed. The letter T denotes that the process terminates.

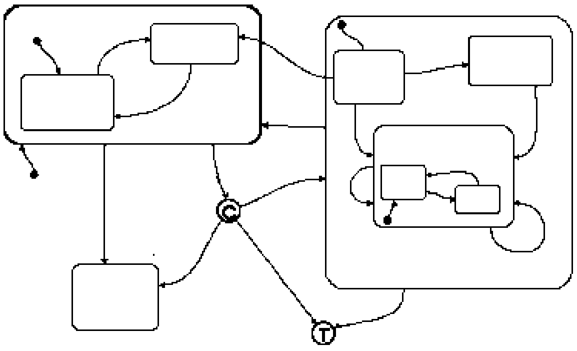
figure 4.16. A simple process in Statecharts[1]



Another property of Statecharts is that sub-states can be created. In the process of figure 4.17. three states are shown, two of which contain structures similar to the process itself. When the system enters such a state, it also enters the initial state of the sub structure. The sub-state of the state can change according to events. When the super-state changes, the sub-state also ceases to be valid. Sub-states can contain sub-states of their own and there is no formal limit to the number of nested sub-state levels.

[1] The diagrams shown in figure 4.16. and figure 4.17. are taken from *UML Statecharts* by B. P. Douglass [41].

figure 4.17. A process with sub-states[1]



4.8.3. Statecharts and CIP

It may seem at first glance, that Statecharts is similar to CIP. Like CIP, states and processes can be defined. Statecharts, additionally, appears to offer many features which CIP users do not have such as sub-states or the means to terminate a process.

CIP experts, on the other hand, will retort that this is not necessarily an advantage. Although it is clear what happens when a process terminates at the coding level, it becomes less clear at a modelling level. In figure 2.1. (page 16) the CIP modelling concept was introduced as reflecting the state of the physical model. In the physical model, processes do not terminate when they cease to be of interest. Even when we are not interested in the state of a switch, it does not cease to have a physical state, and indeed this may be of interest again in the future. Likewise, in CIP sub-state machines are modelled by separate processes rather than embedded within states. This permits them to continue to exist even when they are temporarily irrelevant.

Turning away from states and looking at transitions, *UML Statecharts* [41] by B. P. Douglass says

> *Transitions are modelled as taking approximately zero time to execute, as implied by the statement that an object spends all of its time in states. If a transition can take a significant amount of time, then the object should be decomposed into more states so that eventually, the time taken to get from a predecessor state to a subsequent state is insignificant.*

In a footnote, the author specifies:

> *Classical state machines assume zero-time transitions, but this constraint is relaxed in the UML statechart semantics definition. Nevertheless, transitions need to be "very short" and the object dwells in states virtually all of its life.*

[1] The labels have been removed from this diagram for clarity.

This is a completely different approach from CIP, where the scheduler protects us from such niceties and the user can model in sequence rather than in time mode.

To better understand the difference, the reader is referred to Appendix A.1. (page 137) where the fundamental differences between time and sequentiality are discussed and some problems which can result from not observing this difference are shown. In Appendix A.2. (page 139) the difference is illustrated with a classical example: Zeno's Paradox of Achilles and the Tortoise.

In CIP (and in classical state machines), the object does not need to *dwell in states virtually all of its life*. Such a statement merely suggests that the processing power of the hardware must be grossly overdimensioned. On a code level, the scheduler needs to assure that nothing nasty will happen while a transition is being calculated. On a model level, we can assume that the lower levels (including the scheduler) are functioning correctly. In our sequential view we do not need to consider intermediate undefined states even if in the time view the period required for calculating transitions outweighs the idle phases.

Additionally, in CIP we do not *decompose [an object] into more states so that eventually, the time taken to get from a predecessor state to a subsequent state is insignificant.* We decompose processes solely in order to separate functionality. If a transition is simple but requires a lot of time, so be it. Decomposing it will not reduce the total time required but decomposing where it is not appropriate can make a model more difficult to understand.

Figures such as 4.16. and 4.17. may at first create the impression that they represent state machines in a similar way to CIP, but on closer inspection they represent the code structures of processes (with initialisation, sub-processes and termination). Whilst relieving the user of the repetitive chore of coding processes, the user is not really on the same level of modelling abstraction as he would be with CIP.

4.8.4. Model Checking and Statecharts

There are Model Checking tools written specially for Statecharts [24]. Similarly to the CIP Model Checker, these base on deriving the Model Checking structures from the available data. As shown above, the basic philosophies of Statecharts and CIP are different so the approach taken is only of limited usefulness for this thesis.

4.9. Conclusions

4.9.1. Other model checkers and temporal logics

As presented in this chapter, other model checkers and temporal logics exist, some of which have a greater expressive power than the CIP model checker. However, it was never the goal of this thesis to duplicate these model checkers. Indeed, many were built by large teams of people over more than 10 years. Doubtlessly, techniques from these could be adopted, or better still, these could be integrated with the CIP model checker and this leaves scope for future development. Much of the expressive power of these model checkers is, however, not required in CIP, and in some cases additional structures would have to be implemented to limit these model checkers to a CIP style behaviour (such as *run to completion semantics*).

Most of these model checkers deal with a description of the input system which is structurally closely related to the compilable code. Some even permit direct conversion from meta-code to compilable code. This is different from the CIP model checker: CIP does not model code behaviour but functionality. The structure represented in a CIP diagram does not necessarily reflect the structure of the generated code. The widespread experience which customers have obtained with CIP Tool stands as a guarantee that the generated code is equivalent to the model. But the CIP model checker does not check the generated code or any abstraction thereof. It tests the model as designed by the user.

4.9.2. Other model checkers and fairness

The fairness model used in this thesis clearly separates process decomposition from fairness allowing the developer to model his system along lines of functionality in accordance with the CIP principles rather than having to compromise for the sake of fairness.

4.9.3. What can be borrowed from other model checkers?

Much of the underlying theory and formalisms used in this thesis are borrowed from other model checkers. However, the desire to make the most of the existing CIP environment make many implementation aspects of other model checkers inappropriate without substantial auxiliary structures or transformations.

4.9.4. Contribution of this thesis

CIP Tool permits the user to design his software at a level of abstraction where the behaviour can be modelled without being hampered by vestiges of code structure. A clear analysis and decomposition of the problem is enabled. CIP Tool supports the developer by providing an easy to use system which favours good practice in system design. This enhances maintenance and further development efforts and so contributes to the all-round quality of the finished product.

The CIP model checker is in line with this philosophy. It permits the developer to verify the system under development (even on the fly testing of an incomplete system or component is supported) without having to generate any meta-code or set any parameters on the model checker. By steering clear of any code or meta-code, the model checker permits the developer to verify the model on the same level of abstraction as it was created.

5. Applicability of Model Checking to CIP

5.1. Model Checking and CIP

In this chapter the applicability of Model Checking to CIP models will be discussed. The structure of CIP models makes them well suited to Model Checking without requiring significant additions or modifications. Likewise, the *run to completion semantics* of CIP makes it kinder from the Model Checking point of view than many other fields in which Model Checking is applied.

Significantly, this permits a seamless integration of the model checker within the structures of the CIP-Tool and within the CIP modelling approach. Furthermore, the CIP model checker, while offering a high level of Model Checking capability to the user, should still remain understandable by a user without specific model checking knowledge.

5.2. The system model within the CIP cluster

5.2.1. A process oriented view of the CIP cluster

In chapter 2. the basics of CIP were introduced. The CIP cluster is seen by the user as a collection of interacting processes. The CIP model so enables the developer to divide the problem into individual sub-problems. Every process handles a part of this sub-problem, each largely separated from the rest of the problem. For example, in the cluster of figure 5.1., the process *button* handles only the behaviour of the button (it ensures that button presses and releases are correctly handled). Likewise, process *door* handles the behaviour of the door by ensuring the motor is started and stopped correctly. Each process is concerned only with those messages which have a bearing on its behaviour. While developing process *button*, the developer does not need to consider what will happen when the message *Door_Open* is received. The developer thus views the cluster as a collection of individual processes. Different aspects of the behaviour are separated and the problem is split into smaller problems which are easier to solve.

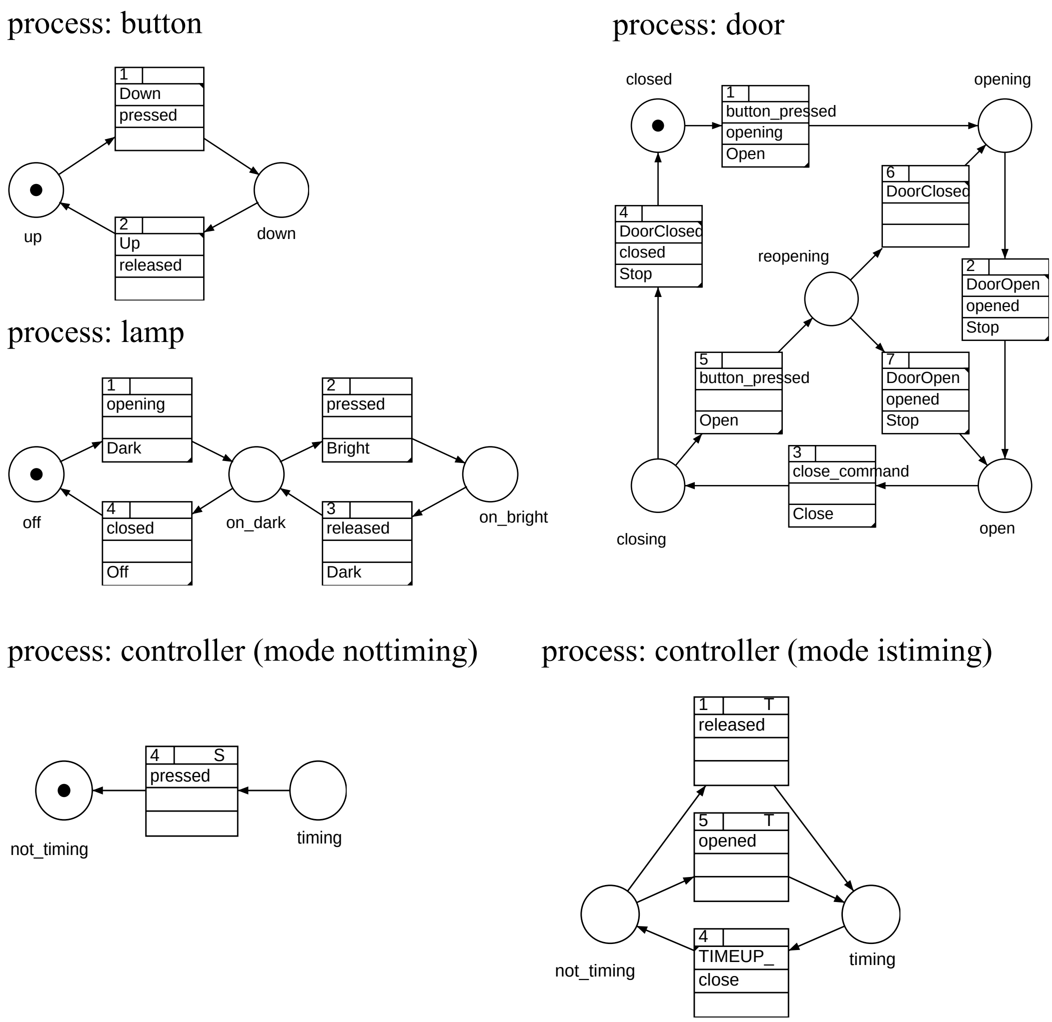
5.2.2. Describing the cluster as a unit

In Model Checking we are more often interested in the state of the cluster than in the states of individual processes. The model checker must thus reverse the division of the problem and constitute the cluster as a single unit.

The active state of the cluster is defined by a set of the active states of the processes. We call this set of states the active *state vector*.

Just as every individual process can be mapped as an extended finite state machine with process states and process transitions, so the entire cluster can be mapped with cluster states and cluster transitions. Each cluster state is a state vector containing one process state from every component process. The state machine so formed is a *product extended finite state machine*.

figure 5.1. CIP cluster from section 2.6.3.



process: button

process: door

process: lamp

process: controller (mode nottiming)

process: controller (mode istiming)

5.2.3. The state explosion problem

In the example of figure 5.1. we have 4 processes with between 2 and 5 states each. If we wish to list all possible state vectors, we would have to list 2 × 3 × 5 × 2 = 60 state vectors. The fact that the complexity of such a mapping could make the cluster far more incomprehensible than it is in the process view is not an impediment for the CIP user, as the user will not need to inspect these structures in the cluster view. The model checking program, however, will need to traverse this structure. The exact traversal required will be discussed in Appendix D, but independently of the approach implemented and no matter whether the entire state space is mapped out or another approach is tried, many if not all cluster states will have to be held in memory simultaneously.

Here lies the problem. The cluster state space can soon grow too large for memory. For example a real cluster may have 10 processes with 10 states each. The cluster state diagram would need 10^{10} states. Thus a moderately sized cluster already requires more memory than most computer systems can offer, even with relatively moderate assumptions on the memory an individual state requires. This problem is the *state explosion problem*.

5.2.4. The state explosion problem in practice

In practice, not all possible state vectors need actually be considered in the cluster state diagram. The reason for this lies in the initialisation of the processes. Rather than listing all possible state-vectors and the transitions between them, we extract our cluster state diagram by beginning with the initial state-vector (*up*, *off*, *closed*, *not_timing*) and traversing from there. To the cluster map we add all transitions of which this is a pre-state and their corresponding post-states. This operation is repeated for the newly found states until no further new states can be found.

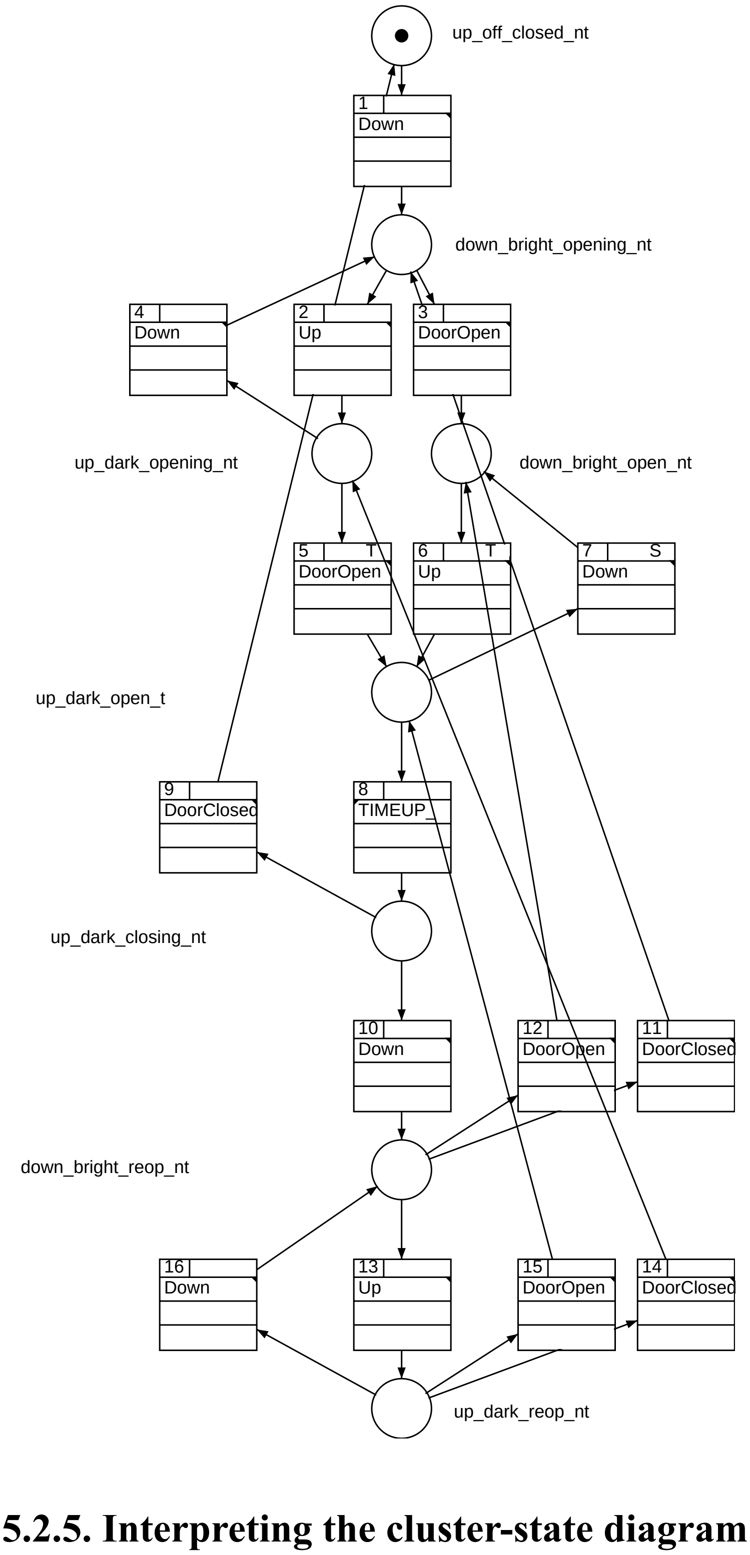
Interestingly, the state diagram obtained as shown in figure 5.2. has 8 cluster states instead of the 60 that are possible.

What are the reasons for this reduction?

The reduction in the number of state-vectors found is caused by the strong interdependence of the processes. Some transitions trigger other transitions so that they cannot be executed individually, making many state-vectors unreacheable from the initial state-vector. If a different initial state were selected, quite a different state diagram might emerge.

This strong reduction in the number of state-vectors having to be considered is a welcome help in countering the state explosion problem. Further reduction possibilities will be discussed in section 5.5. (cluster reduction) and chapter 6 (partial order reduction).

figure 5.2. Cluster of figure 5.1. condensed into one process (actions omitted)



5.2.5. Interpreting the cluster-state diagram

At first it may seem, that the structured nature of the individual processes of figure 5.1. is completely lost in figure 5.2. This is not, however, the case. The original process structures are still implicitly contained in the structure and can be read out using a simple method which will be introduced here.

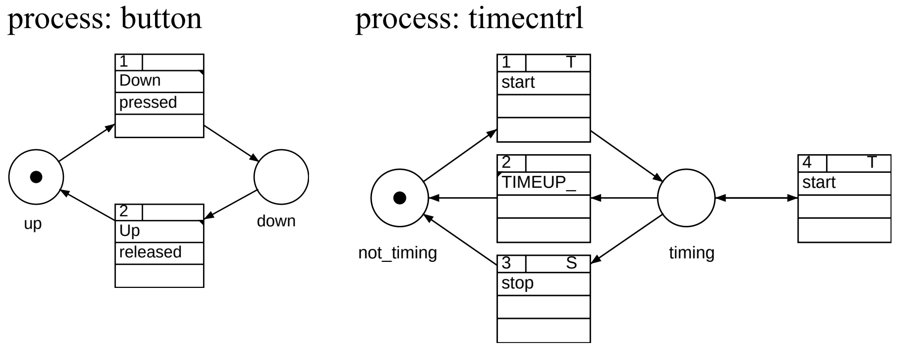
Every state in the cluster-state diagram of a cluster represents a state-vector. The state-vector contains exactly one state from every cluster process. There is a similarity here to geometric locations described by cartesian coordinates. In the latter case, each set of coordinates contains exactly one number corresponding to every dimension of the geometric space.

The cluster states can thus be represented as points in an n-dimensional space, where n is the number of processes. In contrast to the geometric space, where coordinates are unbounded and infinitely variable, cluster-states can take only discrete positions on a finite grid.

Before attempting to graphically represent the 4-dimensional grid of this example, we look into the general principle with a simpler 2-dimensional example.

The example of figure 5.3. shows the two-process cluster from the example of section 2.5.2. The cluster transitions are listed in table 5.1. This cluster-state diagram of the cluster is shown in figure 5.4. In this representation and those that follow, the transition labels are omitted as we are more interested in the structure of the cluster than in what triggers the transitions.

figure 5.3. Processes of two-process system introduced in figure 2.9.



pulse translations

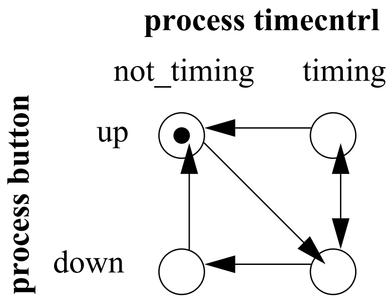
button: pressed
→ timecntrl: start

table 5.1. Cluster-transitions of cluster of figure 5.3.

pre-state		post-state		trigger
button	timecntrl	button	timecntrl	
up[1]	not_timing[1]	down	timing	Down
down	timing	up	timing	Up
down	timing	down	not_timing	_TIMEUP
up	timing	down	timing	Down
up	timing	up	not_timing	_TIMEUP
down	not_timing	up	not_timing	Up

[1] initial cluster state

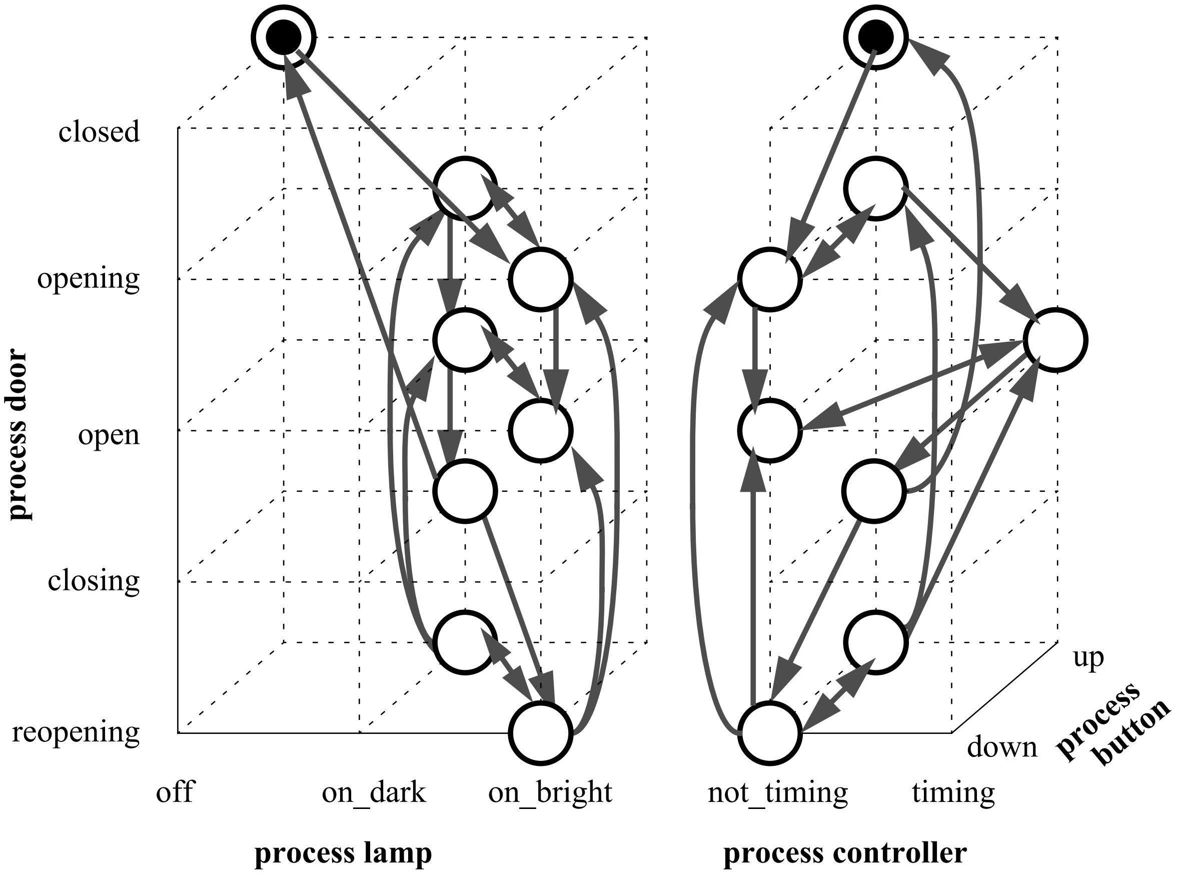
figure 5.4. Cluster-state diagram represented on two dimensional grid



Note that in this two-dimensional grid representation, transitions following the grid lines represent transitions causing state-change in only one process, whereas the diagonal transitions represent transitions causing state change in both processes.

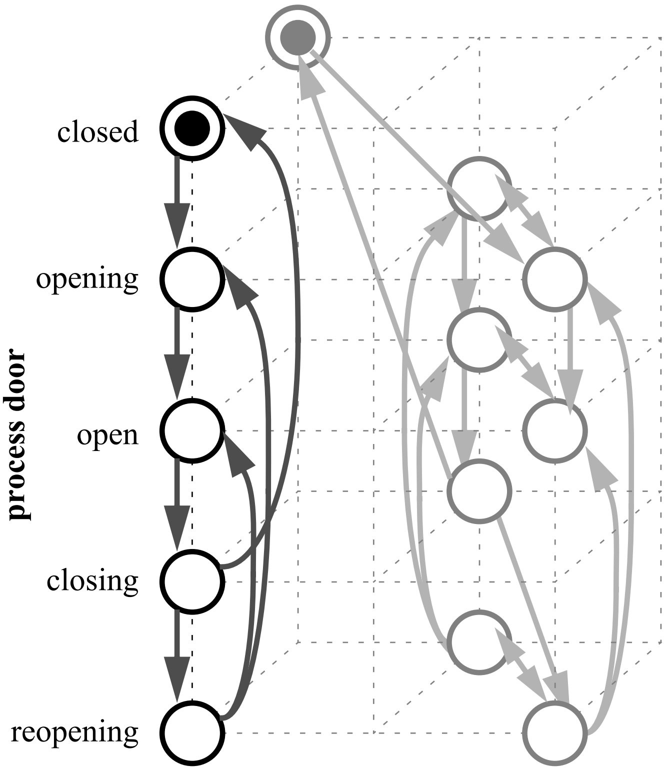
If we were to attempt an n-dimensional representation for the cluster of figure 5.2., we would have to be able to draw in 4 dimensional space. Rather than attempt this on a two-dimensional page, two different three-dimensional projections of the four-dimensional structure are illustrated in figure 5.5. These three dimensional projections are mapped from the four dimensional structure by projecting out one of the dimensions. In the structure on the left, the dimension describing process *controller* is projected out, in that on the right it is the dimension describing process *lamp*.

figure 5.5. Two there-dimensional projections of the four-dimensional cluster-state model of the cluster of figure 5.2.



By making one dimensional projections of the n-dimensional structure, we obtain the structures of the original process state diagram structures. This is illustrated in figure 5.6. with the example of process *door*.

figure 5.6. Projection of structure of figure 5.5. into 'door' dimension



This projected state diagram is structurally identical to the original process state diagram for process *door* (illustrated in figure 5.3.).

5.2.6. Conclusion

The CIP cluster is a representation of the extended finite state machine. A CIP process is a representation of a projection of that extended finite state machine.

5.3. Processes and clusters

5.3.1. Processes as extended finite state machines

Just as the whole cluster can be interpreted as a finite state machine, so too can individual processes be interpreted as finite state machines. When the user creates processes, he is designing them from this perspective.

5.3.2. Processes and outPulses

In section 3.2.2. the extended finite state machine was defined. The extended finite state machine corresponding to the CIP process could also be defined in this way, but to do so we would have to model outPulses as messages. For describing an individual process this may be sufficient, but in the context of a cluster, the two are fundamentally different. An outPulse is a pulse which is sent by one process to another during the execution of a cluster transition. *Run to completion* semantics are in force during such a transition and therefore strict rules are in force governing the sequence and the interruptability of the process transitions. A message on the other hand does not face such constraints. Any transition sequence is possible provided the laws of cause and effect are not violated. For this reason, a distinction must be made between actions and outPulses.

5.3.3. Describing the process extended finite state machine

The definition of the process extended finite state machine is extended from the definition of the extended finite state machine presented in section 3.2.2.

A process extended finite state machine is defined by the tuple:

$\pi := (\mathbf{S\pi}, (\mathbf{E} \cup \mathbf{In\pi}), \mathbf{A}, \mathbf{R\pi}, s_{init\_}\mathbf{\pi}, \mathbf{Out\pi})$.

π is the process extended finite state machine.
$\mathbf{S\pi}$ is the atomic set of process states of π.
$\mathbf{E}$ is the atomic set of events.
$\mathbf{In\pi}$ is the atomic set of process inPulses of π.
$\mathbf{A}$ is the atomic set of actions and includes the nul-action.
$\mathbf{Out\pi}$ is atomic set of process outPulses of π and includes the nul-outPulse
$\mathbf{R\pi}$ is the transition relation of π: $\mathbf{R\pi} : \subseteq (\mathbf{E} \cup \mathbf{In\pi}) \times \mathbf{S\pi} \times \mathbf{S\pi} \times \mathbf{A} \times \mathbf{Out\pi}$
$s_{init\_}\pi \in \mathbf{S\pi}$ is the initial process state.

$\prod$ is the set of all processes.

5.3.4. Process extended finite state machine operations

All operations applicable to extended finite state machines introduced in chapter 3 are also applicable to process extended finite state machines. The latter differ by additionally having $\mathbf{Out\pi}$ in the definition tuple which is not used in chapter 3. The other tuple elements correspond to those in the same position in the definition of section 3.2.2.

Additionally, the following operation applies:

outPulse-element:

outPulse: $\mathbf{R\pi} :\rightarrow \mathbf{Out\pi}$
outPulse(r) := $(u \mid r = (e, s_0, s_1, a, u))$

5.3.5. Extracting process states from cluster states

processState: $\mathbf{S} \times \prod :\rightarrow \mathbf{S\pi}$
processState(s, p) := projection of s to p.

5.4. Deterministic and non-deterministic branching

5.4.1. Branching

In sections 2.3.5. and 2.6.2. conditional branching structures were introduced. When more than one transition shares the same pre-state and trigger, additional information is required to know which of the two is to be executed. CIP-Tool offers three possibilities for providing this information. In *conditions* (section 2.3.5.), the user can embed a code fragment to determine which transition is to be executed. In *gates* (section 2.6.2.), a look-up table is defined in which the transition to be executed is mapped as a function of the states of other processes. In section 2.6.3., *master-slave* structures were introduced in which the process behaviour is also made dependent on the states of other processes.

5.4.2. Non-deterministic branching

The model checker is unable to parse the embedded code used for conditions. To make the model checker able to do this would be very difficult on account of the large expressive power of programming languages. Even if this possibility were implemented, the model checker would have to trace the values of all variables leading to a great increase in the state-explosion problem. So these switch structures must be considered non-deterministic from the model checking point of view. The model checker must assume that when a process is in the pre-state of several transitions with the same trigger, and when the execution preference is ruled by a condition, then any of these transitions can be executed when the trigger is received. In some cases this may lead to transitions being executed which would otherwise not be executed. For example when there is some dependency on the states of other processes. To reduce this, the use of gates should be preferred over that of conditions.

5.4.3. Deterministic branching

Switch structures, where the selection of the transition to be executed is dependent on gates or on master-slave structures, and where the determining process states are known, are wholly deterministic to the model checker. When a process is in the pre-state of several transitions with the same trigger, and when the execution preference is ruled by a gate or master-slave structure and all the determining process states are known, then only one of these transitions can be executed when the trigger is received.

5.5. Reduced cluster structures

5.5.1. A further measure against state-explosion

In section 5.2.3.ff (page 76) the problem of state explosion was introduced. The number of states actually found can be expected to be largely inferior to the total number of states possible. In the example discussed (figure 5.1. and figure 5.2.) the state space was reduced from 60 possible states to 8 actually occurring states. In some large systems, however, even this reduced number of states may be too large to handle.

In some situations, further savings are possible by omitting certain processes from the cluster and studying only the interaction of the remaining processes. This method of omitting processes from the cluster to reduce complexity is called *cluster reduction*. So that processes can still react to pulses being received from processes which are removed, these pulses are replaced by messages from an external source.

In this section it will be shown that cluster reduction can cause several problems for model checking. These problems are illustrated by examples in Appendix C (page 155) and various ways of working around these are discussed. These methods are rejected in favour of a simple but highly effective remedy presented in section 5.5.3.

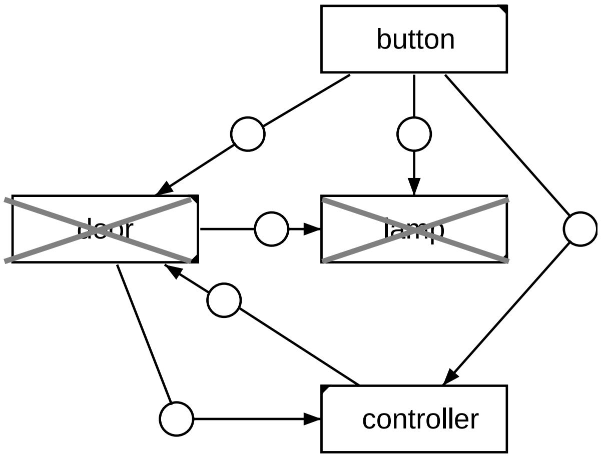
In the following an example of cluster reduction is illustrated. The reader not wishing to study this example can proceed to section 5.5.3. (page 88).

5.5.2. Example of cluster state-space reduction

Suppose we wish to investigate a condition dependent only on the processes *button* and *controller*. Is it possible to do so without considering the other processes?

In figure 5.7. the pulse cast net is shown with the processes *door* and *lamp* removed. There is still a direct pulse translation from process *button* to process *controller*. In the full pulse cast net, process *controller* also receives pulses from process *door*. So that this aspect of the behaviour of process controller is still modelled, these pulses are replaced by messages from an external source. In this example only the inPulse *opened* is affected.

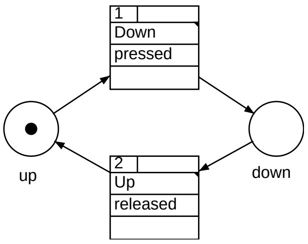
figure 5.7. pulse cast net from figure 2.11. with processes door *and* lamp *removed*



In figure 5.8. the state diagrams of the remaining processes are shown, as is the mode setting diagram for process *controller*. Note how this is projected from the mode setting table for the full cluster. The wholly deterministic structure becomes partially deterministic (it is deterministic for process *button* in state *down* and non-deterministic for process *button* in state *up*).

figure 5.8. processes of reduced cluster (as figure 5.1. but processes door *and* lamp *omitted)*

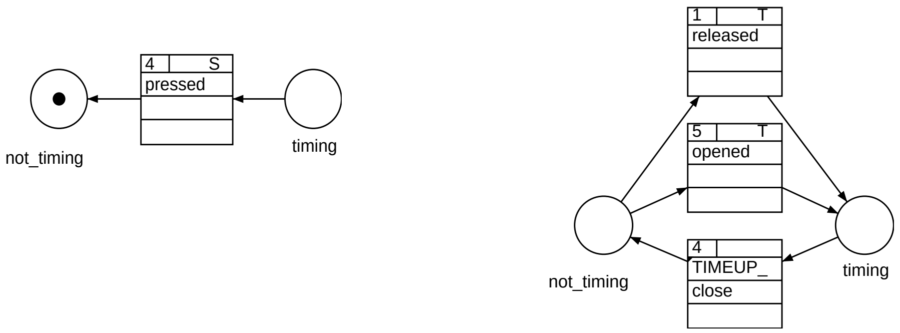
process: button



mode setting for process: controller

		button	
		up	down
door	open	istiming	nottiming
	(other)	nottiming	nottiming
	(unknown)	(either)	nottiming

process: controller (mode nottiming)   process: controller (mode istiming)



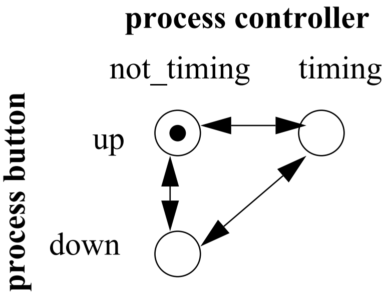
All possible cluster transitions are listed in the following table:

table 5.2. Cluster-transitions of cluster of figure 5.8.

pre-state		post-state		trigger	notes
button	controller	button	controller		
up[1]	not_timing[1]	down	not_timing	Down	
up[1]	not_timing[1]	up	timing	opened	
down	not_timing	up	not_timing	Up	non-deterministic transition
down	not_timing	up	timing	Up	
up	timing	down	not_timing	Down	
up	timing	up	not_timing	_TIMEUP	

1: initial cluster state

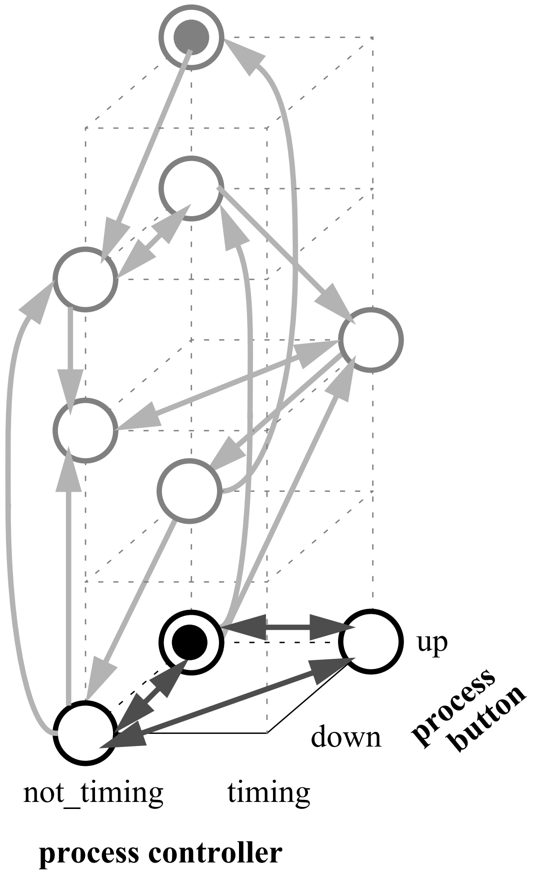
figure 5.9. Cluster-state graph of cluster of figure 5.8.



The cluster-graph obtained has only 3 states instead of the 8 states we would have obtained without omitting processes. The attempt to reduce the state-space was successful.

Does the reduced cluster-state graph still accurately represent the behaviour of these two processes? In this case it does, as the following projection from the cluster state diagram of figure 5.5. shows.

figure 5.10. Projection of state diagram of processes controller *and* button *out of full cluster state diagram*

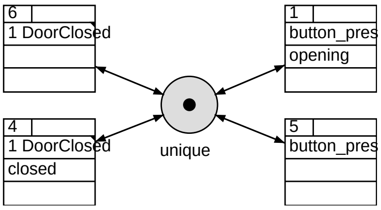


At first sight, it might seem that this approach is without dangers. However, neither a state-space reduction, nor a preservation of behaviour is guaranteed, examples showing how both additional behaviour is gained and lost, and remedies are discussed in Appendix C. In many other state machine formalisms, transitions are gained but never lost. Due to the scheduling strategy of CIP, transitions could be lost if appropriate measures are not taken.

5.5.3. The remedy

Missing processes are replaced by single state processes with non deterministic transitions which either relay outPulses or do not. For more information on this the reader is referred to C.3.9. and C.3.10. (page 172ff.).

figure 5.11. Single state process replacing process door *in cluster reduction.*



A process of this kind takes no memory space in the state vector. There is only one state to choose from and therefore no information is conveyed by the indication of this state.

5.5.4. Implications

Due to this remedy, additional transitions can appear in the cluster state diagram but transitions are not lost. What effect can this have on Model Checking?

The effect on the *invariance* property is that additional states may be reached, so wrongly failing the property. This is on the safe side as it can lead to a good design being incorrectly thought to violate the property, but every system that genuinely violates the property will being recognised as such.

The effect on the *reacheability* property is that states may wrongly be shown to be reacheable. However, reacheability is normally tested for negatively. To check that a state can be reached it is easy to manually construct a path or show that such a path exists by testing. We use a model checker on the other hand to show that a state is not reachable. In this respect the additional paths found provide a safety margin. A warning is displayed by the model checker to remind the reader that the model checker is assuming negative reacheability is being tested for.

The effect on the *eventually* property is that additional paths are found which may provide false counter examples, but existing counter examples are not lost. So when eventually is checked positively, the additional paths provide a safe margin. If eventually is tested for negatively (we desire that a *request* will not certainly be *acknowledged*), additional loops can be formed providing examples. Such properties cannot be verified under cluster reduction.

5.5.5. A *caveat* for the *eventually* property

Another problem with the *eventually* property is that fair loops can be made unfair through additional fair transitions being enabled on the loop. This can lead to valid counterexamples being rejected.

If a fair loop exists in the full cluster and is made unfair in the reduced cluster, this is because in the full cluster, the fair trigger message could not trigger a transition in that state because
i) a gate or master-slave structure prevented the execution or
ii) the first process of the pulse propagation tree was not able to accept the trigger.

The problem is solved by considering fair transitions unfair if they are dependent on a missing process through a gate or master-slave structure or if the first process of the pulse propagation tree is missing.

5.6. Segments

5.6.1. Purpose of segments

In the previous sections of this chapter, methods for creating and interpreting the cluster state diagram from the process state diagrams were discussed.

Once a way has been found of creating the cluster state diagram, we need to put it to the use for which we created it: checking properties.

The algorithm used for traversing the state space will be discussed in Appendix D. In this section, the method will be discussed which is used to recognise whether the reached state fulfils certain conditions.

We may wish to know whether a certain cluster state is reached during the state space traversal, for example the cluster state s_{crit} = [*up*, *closed*, *off*, *timing*]. This is easy to detect as we can compare every cluster state reached with s_{crit}.

Frequently, however, we are not interested in individual cluster states, but sets of cluster states that fulfil certain criteria.

For example: we wish to know whether process *door* can be in state *closed* when process *button* is in state *down*. For this we need to detect for every cluster state whether it fulfils this criterium.

A segment is a region in the cluster state space for which it interests us whether a given cluster-state is a member or not.

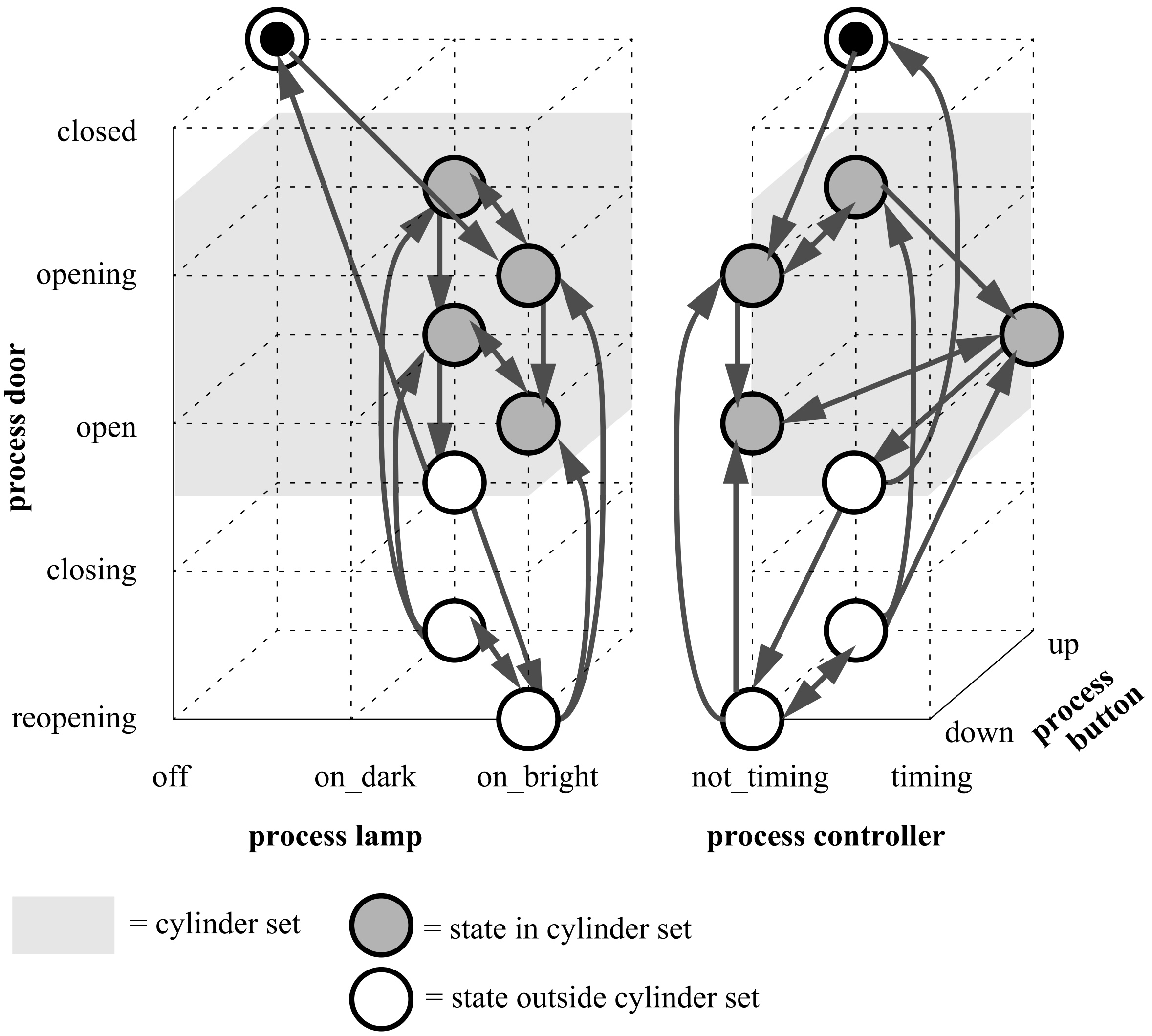
Segments can be defined by enumerating their member cluster states. Normally, however, it is preferable to have a rule to determine which states are included in the cluster and which are not.

5.6.2. Cylinder sets

If we wish to test whether a cluster state s fulfils the criterium ‘process *door* is in state *open* or *opening*’, we check the component of the state vector of s which corresponds to the process state of *door*. Mathematically speaking, we orthogonally project s onto the axis of process door and decide whether the criterium is fulfilled dependent on the point on the axis onto which it is projected. For the purpose of this thesis I define the region in the cluster state space where the criterium is fulfilled is a cylinder set.

These cylinder sets are used to map the atomic propositions introduced in section 3.4.2.

figure 5.12. the cylinder set for the criterium ‘process door *is in state* open *or* opening’ *shown on the state space of figure 5.5.*



5.6.3. Forming segments as logical intersections of cylinder sets

In the CIP model checker, segments are formed as logical combinations of cylinder sets. The user defines expressions combining cylinder sets. Logical expressions such as *and*, *or*, *exor* and *not* can be used to define the segment.

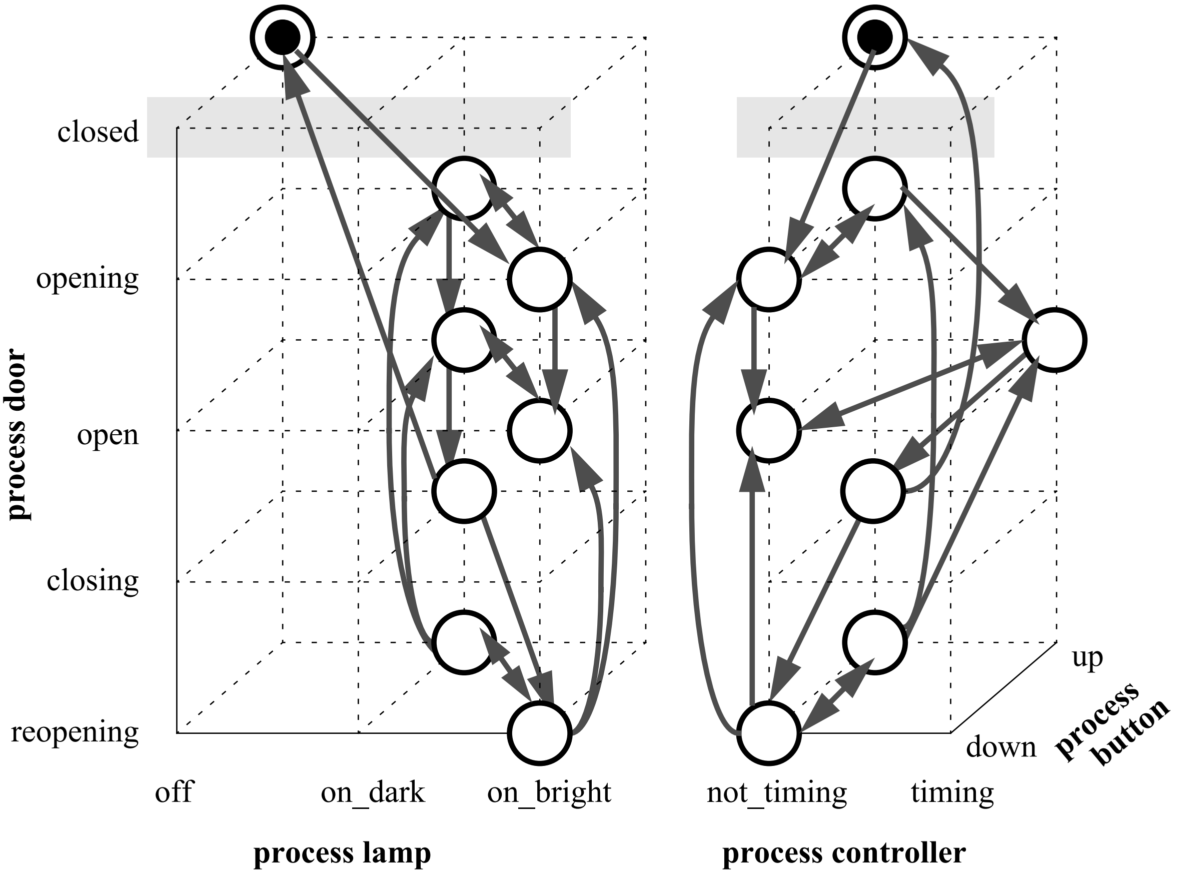
These segments are used to map the propositions introduced in section 3.4.4.

5.6.4. An example of a segment

Suppose we wish to know whether process *door* can be in state *closed* when process *button* is in state *down*. For this we need to define the cylinder sets *door_closed* (process *door* is in state *closed*) and *button_down* (process *button* is in state *down*). We need to check that no reacheable states are in the segment *door_closed* and *button_down*. This segment is illustrated in figure 5.13.

The property we might wish to check is *invariant*(*door_closed and button_down*). Inspection of figure 5.13. reveals that no reacheable states are contained in this segment. The *invariant* requirement is fulfilled.

figure 5.13. Segment for door closed and button down shown on the state space of figure 5.5.



▨ = segment for door closed and button down

5.7. Fairness

5.7.1. Fairness in CIP

In section 3.7. fairness was introduced as a way of securing that starvation of certain transitions in favour of others does not occur when this is not desired. A *fairness constraint* was defined as an attribute for a transition. Fairness was discussed more generally in sections 4.4. and 4.5. In the basic CIP-method, there is no concept of fairness as the CIP-model is concerned with all transitions that can occur, regardless of whether these are certain to occur or may occur optionally. CIP consequently has only one type of transition. To be able to reflect fairness, the CIP model needs to be extended, allowing the user to mark certain transitions as fair.

5.7.2. Example illustrating the use of fairness and a first problem

In the example of figure 5.2. and figure 5.5. the following loop can be executed infinitely often [[*down*, *opening*, *on_bright*, *not_timing*], [*up*, *opening*, *on_dark*, *not_timing*], [*down*, *opening*, *on_bright*, *not_timing*]]

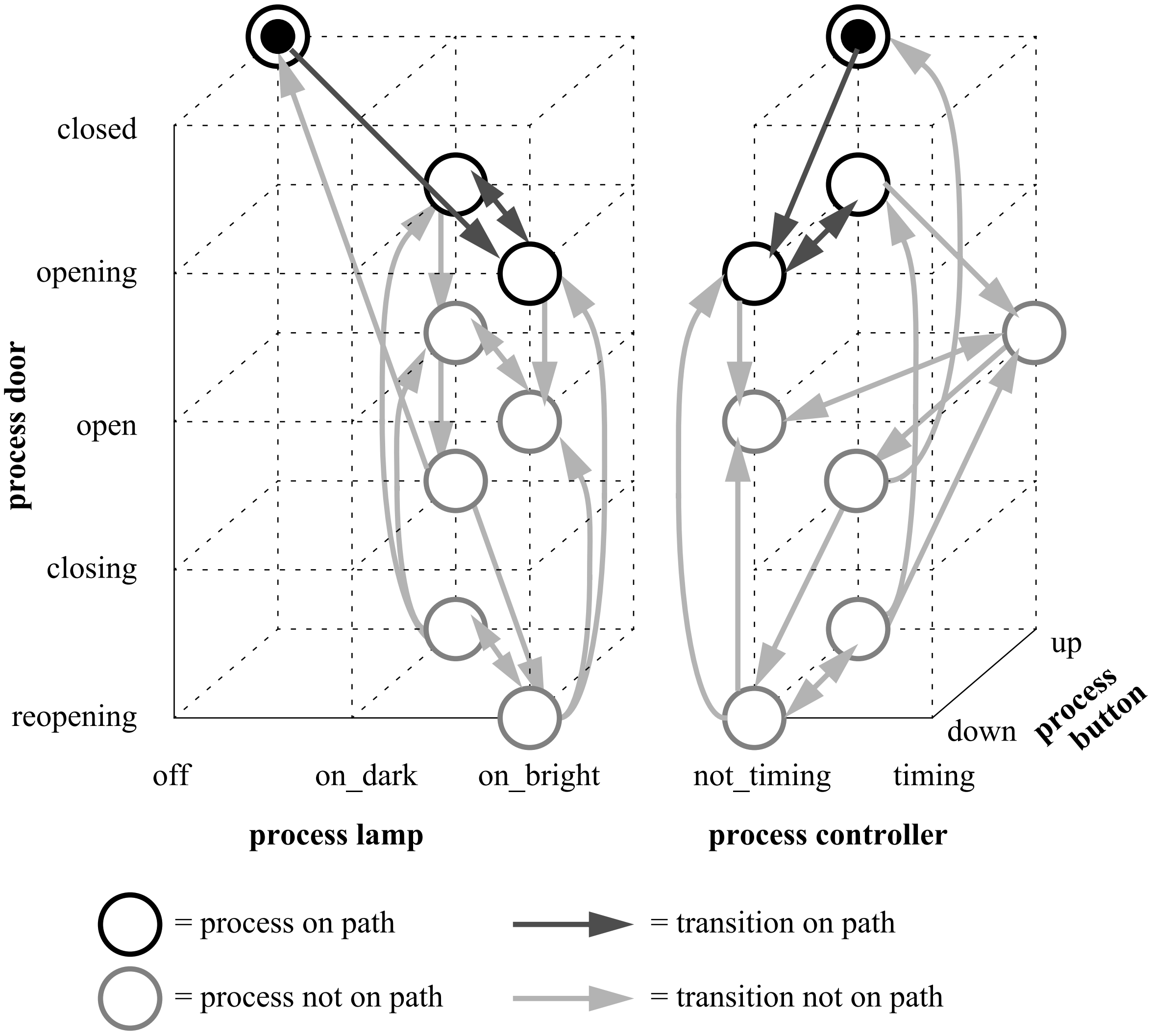
Translated into the behaviour of the real system, this means that the button is being pressed and released infinitely while the door is closing. The physical nature of the system dictates that when the door is closing it will be closed at some point in time[1]. The button cannot be pressed and released infinitely during this finite time span because it too has physical constraints.

Without fairness, the following path of the model system (infinitely repeating a loop) would be legal in the model system but illegal in the physical system.

[[*up*, *closed*, *off*, *not_timing*], [*down*, *opening*, *on_bright*, *not_timing*], [*up*, *opening*, *on_dark*, *not_timing*], [*down*, *opening*, *on_bright*, *not_timing*], ...]

[1] Unless some other effect comes into play such as the door being blocked. If the system should react to such effects then the fairness constraints can be made to allow for this.

figure 5.14. counter-example to condition eventually*(*door_open*,* Pressed*) which would be illegal in the physical system.*



The definition of path fairness (see section 3.7.3.) states that a path is fair iff the following statement holds.

If the path visits a state infinitely often and a transition with the fairness constraint f is enabled in that state, then the path traverses infinitely often a transition with the fairness constraint f.

So to ensure that all fair loops entering this loop also leave it, we have to declare as fair at least one of the transitions leaving this loop. The candidate transitions are ([*down*, *opening*, *on_bright*, *not_timing*], [*down*, *open*, *on_bright*, *not_timing*]) and ([*up*, *opening*, *on_dark*, *not_timing*], [*up*, *open*, *on_dark*, *timing*]). Declaring either of these as fair would make the path of figure 5.14. unfair, and so prevent such physically unacceptable behaviours from leading to false conclusions in Model Checking.

Should we declare one of these transitions as fair, or both? Suppose we declare only one transition t_0 as fair. Then all fair paths infinitely visiting the loop would have to execute t_0 infinitely. The other transition, t_1, could be traversed finitely or infinitely. If, however, we make both t_0 and t_1 fair (with different fairness constraints), then all infinite paths visiting the loop would have to execute both transitions infinitely. Which of these solutions best represents physical reality?

Neither option is suitable. The physical system dictates that the door must be opened at some point if it is opening. The physical effect of the door opening does not discriminate between states of the button. t_0 and t_1 must be infinitely executed between them. Whether one is executed finitely or both are executed infinitely is not part of the requirement and it would be incorrect to cement such an arbitrary decision into the model.

The solution is to make fuller use of the definition of path fairness, and declare that both these transitions must share all fairness constraints.

In the same example, a similar problem occurs with the loop [[*down*, *reopening*, *on_bright*, *not_timing*], [*up*, *reopening*, *on_dark*, *not_timing*]]. Four different transitions leave this loop, but there is no reason to prioritise any individual or set of these over the others. All four must therefore share the same fairness constraint(s).

5.7.3. Fairness for process transitions

In general, being able to attach fairness to process transitions rather than cluster transitions has the huge advantage for the CIP user as well as for the implementation, that there are usually considerably fewer process transitions than cluster transitions. The meaning of fairness in conjunction with a process transition is also much more intuitive. Managing and keeping track of the fairness constraints of processes is therefore easier and less prone to error (quite apart from the fact that generating and displaying in an editor list all cluster transitions would in itself cause considerable state explosion problems). Furthermore, most fairness conflicts occur when one process starves another of triggers. There may be few cases where applying fairness to process transitions only causes implementation conflicts for the user, but a workaround can normally be found. For this reason it was decided to implement fairness allocation on process level only.

The fairness constraint of a cluster transition is the union set of all the fairness constraints of its component process transitions.

5.7.4. Example with more than one fairness constraint

Sometimes, a transition may have more than one fairness constraint.

In figure 5.15. our door control system is illustrated with projections of processes *door* and *lamp*. The following fairness constraints were applied:

f_1: applied to transition (*up*, *down*) of process *button* because it is assumed that the button is sure to be pressed again when it has been released (it could be that there is a timer waiting a random but finite time).

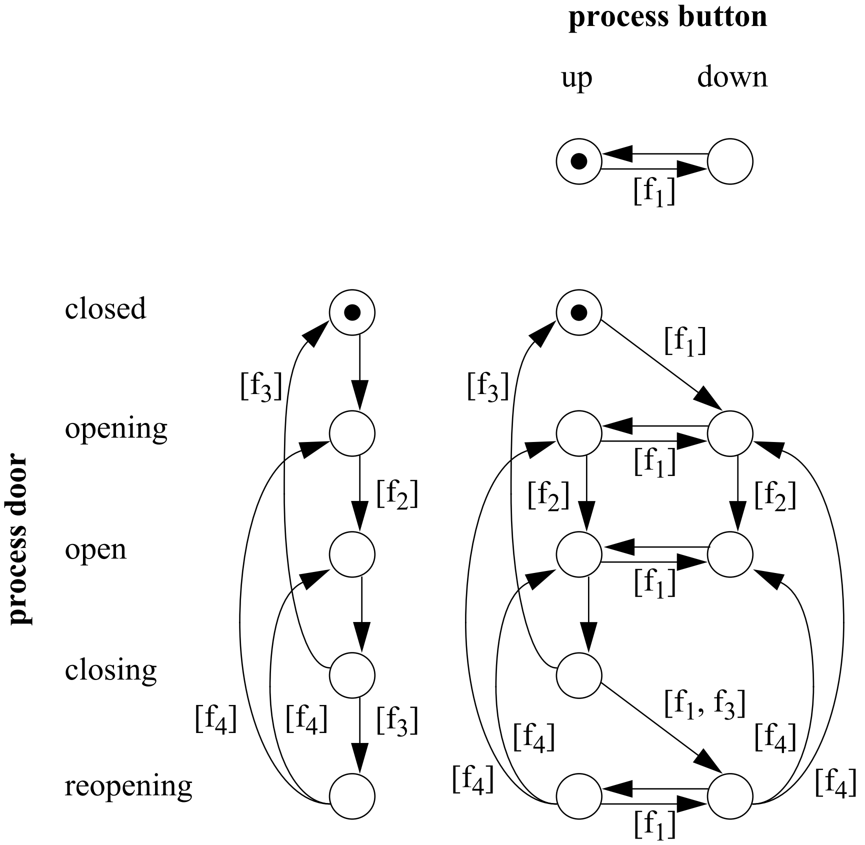
f_2: applied to transition (*opening*, *open*) of process *door* because the process may not remain in state *opening* indefinitely.

f_3: applied to transitions (*closing*, *closed*) and (*closing*, *reopening*) of process *door* because the process may not remain in state *closing* indefinitely.

f_4: applied to transitions (*reopening*, *opening*) and (*reopening*, *open*) of process *door* because the process may not remain in state *reopening* indefinitely.

In fact, fairness constraints f_2, f_3 and f_4 may also be handled as if they were the same constraint because they can never be enabled simultaneously, and can only be disabled through the execution of a transition with this fairness constraint.

figure 5.15. Door control system with fairness constraints shown in square brackets []. Only processes button *and* door *are shown. The fairness constraints are defined for the processes and applied by implication to the cluster.*



Looking at the cluster state graph, we see that the transition ([*up*, *closing*], [*down*, *reopening*]) has two fairness constraints (f_1 and f_3). It follows that all fair paths infinitely visiting [*up*, *closing*] must infinitely execute at least one transition with the constraint f_1 and infinitely execute at least one transition with the constraint f_3.

A path which infinitely executes the transition ([*up*, *closing*], [*down*, *reopening*]) can be fair. A path which infinitely executes ([*up*, *closing*], [*up*, *closed*]) is only fair if it also infinitely executes ([*up*, *closing*], [*down*, *reopening*]). It follows that all fair paths infinitely visiting [*up*, *closing*] must also infinitely visit [*down*, *reopening*]. Therefore the property *eventually*([*down*, *reopening*], [*up*, *closing*]) holds for the system. The property *eventually*([*up*, *closed*], [*up*, *closing*]) does not hold. The fair loop [[*up*, *closing*], [*down*, *reopening*], [*down*, *open*], [*up*, *open*], [*up*, *closing*]] is a counterexample (transitions with the fairness constraints f_1, f_3 and f_4 are enabled and executed).

Without the fairness constraint f_1, the property *eventually*([*down*, *reopening*], [*up*, *closing*]) does not hold.

5.7.5. General guidelines for setting fairness

- When a physical state of the system can only last finitely, then all transitions changing this state share a fairness constraint (e.g. door is closing and must either be closed or start reopening).

- When a timer is set, the timeout shares a fairness constraint with the transitions that reset or stop the timer, or take the timer to a state where the timeout does not trigger a transition.

5.8. Conclusions and summary

5.8.1. Conclusions

The CIP model is well suited for model checking purposes on account of its strict definitions of processes and their communication. Only very little additional information is needed and must be added by the user wishing to use the Model Checking capability.

In section 2.2.2. and especially in figure 2.1. the software architecture of a system implemented with CIP-Tool was illustrated. On the left side of the figure is the physical system described by a large number of analog variables. On the right side are the CIP components which follow or influence these analog values, but are themselves described by discrete values. This introduction also underlines the mechatronic causality between actions and events. We can, in our door system example, start the process of closing the door by sending the software connector of the door motor the *Close* message. When this message is sent, the CIP process *door* is in state *closing*. The connector of the door motor receives the message and then causes electrical switches to be set which then cause changes in the magnetic fields of the drive. These in turn cause kinetic change in the rotor and consequently affect the velocity of the door. The velocity of the door causes a change in its position ultimately causing the door to trigger the sensor telling the CIP cluster that the door is closed. If the door had not been set into motion, this message would not have been received.

The model checker does not know the nature of the physical system and does not aim to simulate this system with all its analog variables. It does not know that the message *DoorClosed* can only be received after *Close* has been sent. Neither does it know that the button can only be released if it is pressed. The model checker is prepared to deal with any arbitrary sequence of input messages, including those which it would be impossible to receive from the physical system such as (*DoorClosed*, *DoorClosed*, *DoorClosed*, *Down*, *Down*, *Down*, ...).

These arbitrary sequences do not, however, lead to a state explosion problem which is considerably larger than that of the physical system, because the processes themselves rigidly introduce order. The state diagram of process *button*, for example, only accepts the message *Down* when it is in state *up*. The message sequence (*Down*, *Down*, *Down*, *Up*, *Up*, *Down*) causes the same sequence of cluster transitions as the message sequence (*Down*, *Up*, *Down*).

Additional order is introduced by fairness constraints. In contrast to the structural order described above, fairness constraints cannot be detected automatically by CIP-Tool but must be defined by the user. Fairness constraints prevent the repetitive execution of certain transitions to the exclusion of events which would certainly occur in the physical system.

5.8.2. Summary

The points raised below are explained in the previous sections of this chapter. They are repeated to provide an overview of the principal points learned. They provide mechanisms which the user may wish to exploit when using the CIP model checker and tips for making best use of the tool. The numbers of the sections to be read for further information are suggested in square brackets.

- To aid the model checker, gate and master-slave structures should be preferred over conditions whenever possible [5.4.].

- To help reduce the state space, processes can be omitted. This method is *cluster reduction* [5.5.]. If the processes to be omitted are carefully chosen, the effects on the cluster behaviour can be kept minimal. By using a suitable strategy for handling missing interactions, interactions between the remaining processes are not lost [5.5.3.]. Some additional interactions may be gained however. Their presence is on the safe side as, when certain precautions [5.5.4. - 5.5.5.] are respected, these additional transitions may lead to correct conditions being failed but not to incorrect conditions being accepted [5.5.4.].

- *Segments* are used to denote groups of cluster states for which given propositions hold. In model checking these are the basis for defining the conditions which have to be verified [5.6.].

- *Fairness* constraints can be allocated to process transitions or sets of process transitions by the user [5.7.] in order to prevent the model checker identifying false counter examples by identifying loops which can otherwise be infinitely executed in the model but not on the physical system.

6. Partial Order Reduction

6.1. Purpose

This section discusses *partial order reduction*, a tool reducing state explosion problems. The basics of Partial Order Reduction are introduced and the implementation in the CIP model checker is discussed. Further information and the background are discussed in appendix F.

The concepts of partial order reduction are well known and documented [18]. Sections 6.2. to 6.4. introduce a general approach to partial order reduction as presented in handbooks on the subject. The material presented is however cut down to that which is specifically relevant for this thesis. The definitions used conserve their full relevancy but the discussion of these and examples are presented with the aims of this thesis in mind.

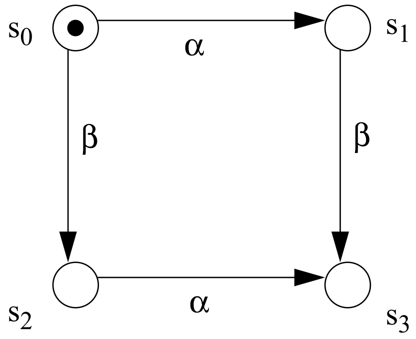
From section 6.5. onwards this chapter concentrates on the solution which was created for this thesis and deviates slightly from the earlier approach.

6.2. Introduction

Partial order reduction is the mapping of the full state graph onto a reduced state graph whilst conserving the properties of the full state graph. Such a mapping is worthwhile if the reduced state graph requires less memory or can be traversed faster.

Let us consider the following example:

figure 6.1. graph with potential for Partial Order Reduction



This cluster-state graph consists of 4 states and 4 cluster transitions. The labels on the transition arrows indicate the associated process transition. The graph has two full paths, [s_0,s_1,s_3] and [s_0,s_2,s_3]. If the intermediate states are irrelevant to the property being checked, we only need to follow one of these paths. The transitions α and β are **interleaving**. They can be executed in either order and it does not matter in which order they are traversed. The full state graph can thus be replaced by a reduced state graph containing only one of the above two paths.

This example is discussed more fully in appendix F.1.

When constructing this reduced state graph, we first had to construct the full state graph. The method thus failed to reduce memory requirements. In this chapter, methods will be introduced for constructing a reduced state graph without constructing the full state graph.

6.3. Independence and visibility of transitions

6.3.1. Enabledness, commutativity and independence

Enabledness, commutativity and independence are criteria used in recognising which transitions can be omitted from a reduced state graph and which must be retained.

For a full discussion of enabledness, the reader is referred to section 3.2.5. and appendix F.2.1. The basic concepts are presented below.

Definition: process-transitions t_1 and t_2 are **commutative** iff for all states s such that $\{t_1,t_2\} \subseteq$ *enabled*(s), $t_1(t_2(s))=t_2(t_1(s))$.

In figure 6.1. α and β are commutative.

Note that this definition implies that if t_1 and t_2 are not jointly enabled for any state, then t_1 and t_2 are commutative.

Definition: process-transitions t_1 and t_2 are **independent** iff they are commutative and not mutually disabling.

Therefore process-transitions t_1 and t_2 are **independent** iff for all states s such that $\{t_1,t_2\} \subseteq$ *enabled*(s), $t_1 \in$ *enabled*(t_2(s)) and $t_2 \in$ *enabled*(t_1(s)) and $t_1(t_2(s))=t_2(t_1(s))$.

In figure 6.1., α_0 and β are independent.

Definition: process-transitions t_1 and t_2 are **dependent** iff they are not independent.

Definition: processes p_1 and p_2 are **independent** iff all process-transitions of p_1 are independent of all process-transitions in p_2.

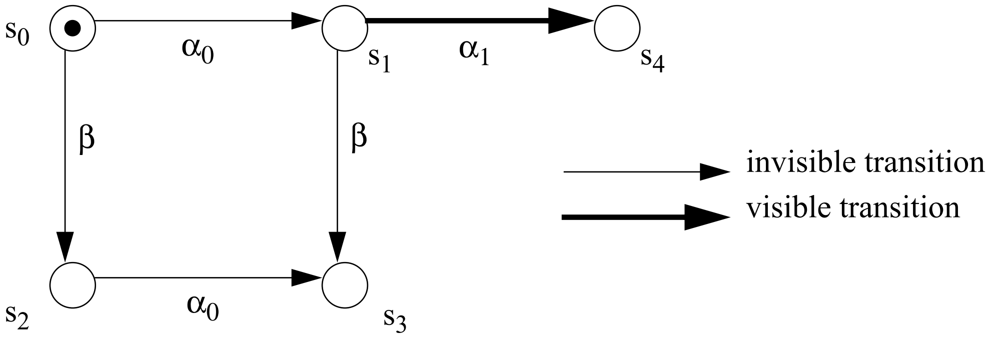
6.3.2. Visibility

Definition: a cluster-transition t is **visible** for a proposition c iff its c(*pre-state*(t)) ≠ c(*post-state*(t)).

Definition: a cluster-transition is **invisible** iff it is not visible.

In the example of figure 6.2., let us suppose we are checking a condition based on s_4.
The transitions (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_2, s_3) are invisible to this condition.
The transition (s_1, s_4) is visible.

figure 6.2. visibility of transitions to condition testing for s_4.



6.3.3. Reduced state graphs

All full paths in figure 6.2. take one of the following two forms:

-initially not s_4 and subsequently s_4

-never s_4

The individual states traversed may vary, but all paths respect one of these two patterns. But as the condition does not distinguish any states other than s_4, it cannot distinguish between the paths [s_0, s_1, s_3] and [s_0, s_2, s_3]
This full state graph can be reduced to a reduced state graph by omitting indistinguishable paths. The following figures show possible variants for a reduced state graph.

figure 6.3. possible reduction of state graph from figure 6.2.

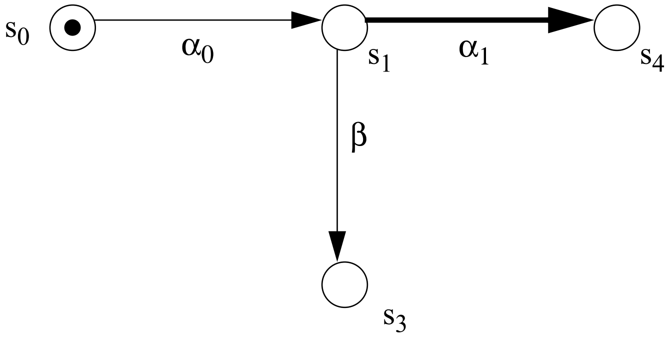
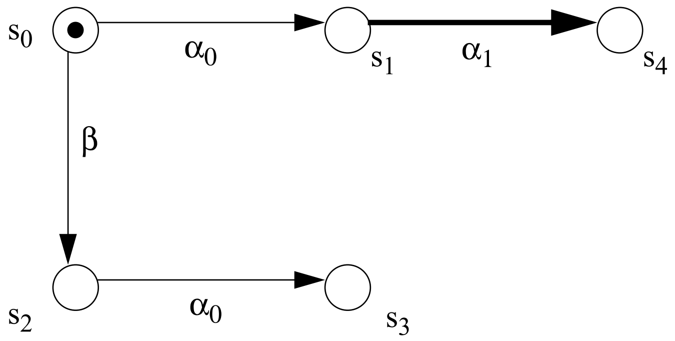


figure 6.4. another possible reduction of state graph from figure 6.2.



The reduced state graph of figure 6.3. is preferable because it uses both fewer states and fewer transitions. In a real situation however, we cannot always obtain the optimal structure. Instead we are content when we can obtain a significant reduction of the state graph.

6.4. Introduction to reduction techniques

6.4.1. The set *ample*(s)

When creating a reduced state graph, we replace the set *enabled*(s) of every reached state s by a subset which we denominate *ample*(s).

ample(s) ⊆ *enabled*(s)

Definition: a state s is **fully expanded** iff ample(s) = enabled(s).

A set of rules is required to decide which elements of enabled(s) may be omitted in *ample*(s).

In case of doubt over whether *ample*(s) should include a transition or not, the transition should be included in order not to exclude any behaviour.

6.4.2. Creating ample sets - a simple set of rules

In *Model Checking* by Clarke, Grumberg and Peled [18] on pp 147ff, a method for obtaining ample sets is introduced.

rule **C0**: *ample*(s) = ∅ iff *enabled*(s) = ∅.

rule **C1**: Along every path in the full state graph that starts in s, the following condition holds: a transition that is dependent on a transition in *ample*(s) cannot be executed without a transition in *ample*(s) occurring first.

rule **C2**: If s is not fully expanded, then every $\alpha \in$ *ample*(s) is invisible.

rule **C3**: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in *ample*(s) for some state of the cycle.

Rule C0 prevents the reduced-state graph containing dead ends where the full-state graph does not.

C1 is by far the most complex of the rules. Note that it applies to every path of the full-state graph. Fortunately, there are methods of ensuring this rule is respected without constructing the full state graph.

One important consequence of this rule is that all elements of *enabled*(s) - *ample*(s) are all independent of all elements of *ample*(s).

Proof: suppose there exists a transition $t_1 \in$ *enabled*(s) which is dependent on a transition $t_2 \in$ *ample*(s). t_1 cannot be executed without a transition in *ample*(s) being executed first. However, as t_1 is the first transition, it must also be included in *ample*(s).

An example illustrating the application of this method can be followed in appendix F.3.1.

6.4.3. Creating ample sets in practice

Rule C1 as described above still requires a full state exploration. Some of the decisions taken could not have been taken without knowledge of the full state graph. The approach described by Clarke, Grumberg and Peled [18] is to consider all pairs of transitions belonging to the same process to be dependent and all transitions in different processes which share a variable to be dependent on all transitions in both processes. Based on this assumption, we select for *ample*(s) a set of dependent transitions so that all elements of *enabled*(s) - *ample*(s) are independent of *ample*(s).

The example of appendix F.3.2. shows that less reduction is obtained than was obtained for the reduction of the same system in F.3.1. There is room for improvement.

6.5. More approaches to Partial Order Reduction

6.5.1. Purpose

In this section, an alternative set of rules is introduced that can be used to create *ample*(s).

6.5.2. Degree of process-enabledness

Definition: the **degree of process-enabledness** (dpe) of a process transition t in a cluster-state s is the maximum number of process-states the state-vector of s has in common with a state-vector where t is enabled.

dpe: $\mathbf{S} \times \mathbf{T} \rightarrow \mathbb{N}$
dpe(s,t) is the degree of process-enabledness of transition t in cluster-state s.

dpe(s,t) can be calculated using the following algorithm:

```
dpe(s, t)
    var sum := 0;
    for all p ∈ processes
        if t ∈ enabled(p component state of s)
            sum ++;
        end if
    end for
return sum
```

To interpret *dpe* in terms of the coordinate system model introduced in section 5.2.5, *dpe*(s, t) is the maximum number of dimensions a projection of the cluster can have so that t is enabled in s.

A transition is enabled when its dpe equals the number of processes.

See also the example of appendix F.3.3. (page 202)

6.5.3. Process-enabling and process-disabling

Definition: a cluster-transition t_0 **process-enables** a transition t_1 iff
dpe(*post-state*(t_0),t_1) > *dpe*(*pre-state*(t_0),t_1)

Definition: a cluster-transition t_0 **process-disables** a transition t_1 iff
dpe(*pre-state*(t_0),t_1) > *dpe*(*post-state*(t_0),t_1)

6.5.4. Rules for creating *ample*(s)

The following simple set of rules allow *ample*(s) to be determined.

rule R1: search for a non empty subset of invisible transitions of *enabled*(s) so that no process transition outside the subset is process-disabled by any member of the subset unless all members of *enabled*(s) disable that process transition.
if successful, select that subset as *ample*(s)
otherwise, *ample*(s) = *enabled*(s)

rule R2: check that no element in *ample*(s) closes a loop of the reduced state graph. If a loop is closed, check that in the loop no transition t is enabled but not included in *ample*(s) for any state of the loop, unless a path branches from the loop and t is not disabled by the first transition of that branch. Add transitions to reduced-state graph if necessary.

rule R3: dead ends are treated as transitions.

rule R4: if a fair transition is enabled in a state s, then *ample*(s) must include at least one fair transition. Additionally, preference is given to transitions in the following order: transitions whose post-states were previously visited, transitions not already on the search path, transitions of the same process as the previous transition.

See also the example of appendix F.3.5. (page 203).

Differences between this method and that of 6.4.2. are shown in the example of appendix F.3.6. (page 204).

6.5.5. Note on visibility of transitions

Visible transitions are transitions for which the pre- and post-states respond differently to propositions. The pre and post-states of such transitions are thus in different cylinder sets.

As model checking statements are based on the sequences of segments visited, any state reduction is correct if all segment sequences are conserved. To demonstrate that a state reduction mechanism is correct, it suffices to prove that any segment sequence of the full state graph also exists in the reduced state graph.

(See also appendix F.4.2.)

6.5.6. Proof of correctness of rules of 6.5.4.

In this section it is shown that if a path p of the full state graph visits segments in a given sequence, then there is a path in P’, the set of paths of the reduced state graph, so that a path of P’ visits these segments in the same sequence.

In the full state graph, all full paths start in the segment containing the initial state S_{init} and either

i. visit another segment upon leaving Sinit, or
ii. do not leave Sinit.

In case ii, the path p’ either

ii.a reaches a dead end or
ii.b loops infinitely.

By rule R3, ii.a and ii.b can be considered as one.

The path p starts in s_{init} and is either a path of P' or it branches from such a path in a state s_{branch}.

1) In the first case, p is obviously contained in P'. The latter case requires proof.

2) Let t_0 be the first transition on p after s_{branch}. t_0 is invisible because otherwise p would not remain within the initial segment. t_0 remains enabled on a path of P' until executed because if this were not the case, rule 1 would be violated at the transition of a path of P' disabling t_0. Therefore at least one path of P' executes t_0 after s_{branch} and before leaving the segment.

3) Let t_1 be the transition on p after t_0. Like t_0, t_1 is invisible. The value *dpe*(s_{branch}, t_1) is known. Along at least one path of P' the dpe of t_1 in every state after s_{branch} must be equal to or greater than this value until t_1 is executed or all enabled transitions disable t_1 (otherwise rule R1 would be violated).

4) If t_1 is not enabled in s_{branch} then it is process-enabled by t_0. If t_1 is enabled in s_{branch} then it is not process-disabled by t_0.

5) It was shown in (2) that t_0 is executed on p. Combining with (4) it is clear that t_1 becomes fully enabled on at least one path of P'. For the same reason that t_0 is executed on one such a path (2), t_1 is subsequently executed on a path of P' unless a state s_{fe} has previously been visited where all transitions from that state process-disable t_1. It must be assumed that s_{fe} occurs prior to the execution of t_0 and that t_0 process-enables t_1, otherwise t_1 would be a member of *ample*(s_{fe}) (by rule R1).

6) But *ample*(s_{fe}) must include t_0 and therefore t_0 must process-disable t_1. This contradicts (4). Therefore at least one path of P' executes t_1 after s_{branch} and before leaving the segment.

7) The arguments applied to t_1 also hold for t_2, the transition following t_1 on p. Likewise, they hold for all subsequent transitions of the loop section of p.

8) On the loop path, t_0 has become enabled by the point where the loop is closed. On all paths of P' where the loop transitions have been executed, t_0 has also become enabled. If this were not the case, the path must have previously executed a transition process-disabling t_0. If this were the case (with at least one loop transition enabled), there must be another path in P' branching off at this point where t_0 is not process disabled.

Therefore P’ must contain a path forming a loop within the initial segment.

We now proceed to case i. If there is a path p from the initial state (in segment S0) and proceeding to a segment S1 without visiting another segment, we must show that there is also a path in P’ doing the same.

> 9) p crosses from S0 to S1 by the visible transition t_V. By applying arguments (2) to (6) we show that there is a path in P’ executing t_V.

But is there a path in P’ executing t_V without passing through a segment other than S0 before doing so?

> 10) If a path in P’ passes through a segment other than S0 before executing t_V then it executes another visible transition before executing t_V. The pre-state of this visible transition is fully expanded (rule R2). On at least one of the paths in P’ branching off at this point, t_V will eventually be executed as shown in (9).

Therefore if there is a path of the full state graph visiting segments in a given sequence there is also a path of the reduced state graph visiting the same segments in the same sequence.

The equivalence of these rules to those of section 6.4.2. is discussed in appendix F.4.1. (page 206).

6.6. Implementation

6.6.1. Purpose

In this section, the implementation of the algorithm presented in section 6.5.4. (page 106) is discussed.

6.6.2. Implementation of algorithm

Rule R1 may appear highly complex at first because of the need to evaluate the dpe of every process transition in the post-state of every enabled transition for every state visited by the reduced state graph.

On closer inspection this is not necessary. A process transition t can only process-enable or process-disable process transitions which:

a) have component transitions in the same processes as t or its components or
b) process-enable or process-disable transitions in other processes because the latter are dependent on t through inspections (conditional transitions) or master-slave structures.

All transitions whose pre-state is the post-state of the executed transition are process-enabled and all transitions whose pre-state is the pre-state of the executed transition are process-disabled (**rule R1.1**).

This rule can be verified by checking for transitions sharing pre-states with the transition to be executed.

A transition process-disables any transition of which a component is process-disabled (**rule R1.2**).

This criterium can be verified by checking whether any component of the transition could trigger other components or not trigger components which are triggered.

When a process transition which is triggered by an inPulse is process-enabled, this process-disables the sending transition by potentially replacing it by a new combined transition (**rule R1.3**).

This criterium can be verified by checking in the processes with components of the executed transition whether message triggered transitions are process-enabled.

In case b, it is known for every transition which process transitions it process-enables or disables and these can be determined before the state space exploration is begun and listed (**rule R1.4**).

During exploration, for every executed transition it is verified whether any of these are process-disabled.

If several invisible enabled transitions process-disable the same set of (combined) transitions, these need not all be included in *ample*(s) (**rule R1.5**).

Sometimes a transition t is process-disabled when replaced by another transition made up of t and a transition t'. We can obtain additional partial order reduction when treating t as if it were not process-disabled provided t' is invisible and does not disable any transition other than itself. This procedure is correct because in the correctness demonstration of section 6.5.6. t and t' can be considered to be two consecutive transitions rather than t+t' (**rule R1.6**). Transitions of the type of t' can be identified before traversal begins so saving time.

Failure to find a reduced ample set leads to the entire enabled set being used as ample set (**rule R1.7**).

The implementation of rule R2 is straightforward.

To correctly implement rule R3, all dead-ends are identified before the state graph is constructed. A dead-end is a state vector of which all members are dead-ends in their process state graphs. It can thus easily be detected which transitions process-disable these dead ends (**rule R3.1**).

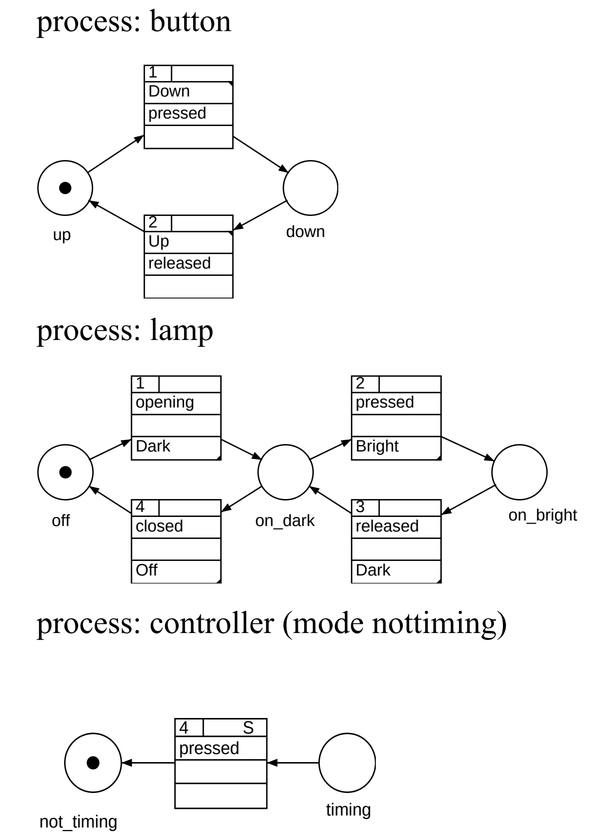
Rule R4 needs no additional commentary.

6.6.3. Example

The subject of this example is the door control example of chapter 5. Fairness constraints have been added to process *door*.

We wish to evaluate the property *eventually*(*closing*). Therefore the process transitions (*open*, *closing*), (*closing*, *closed*), (*closing*, *reopening*) and any transitions containing these as component are visible.

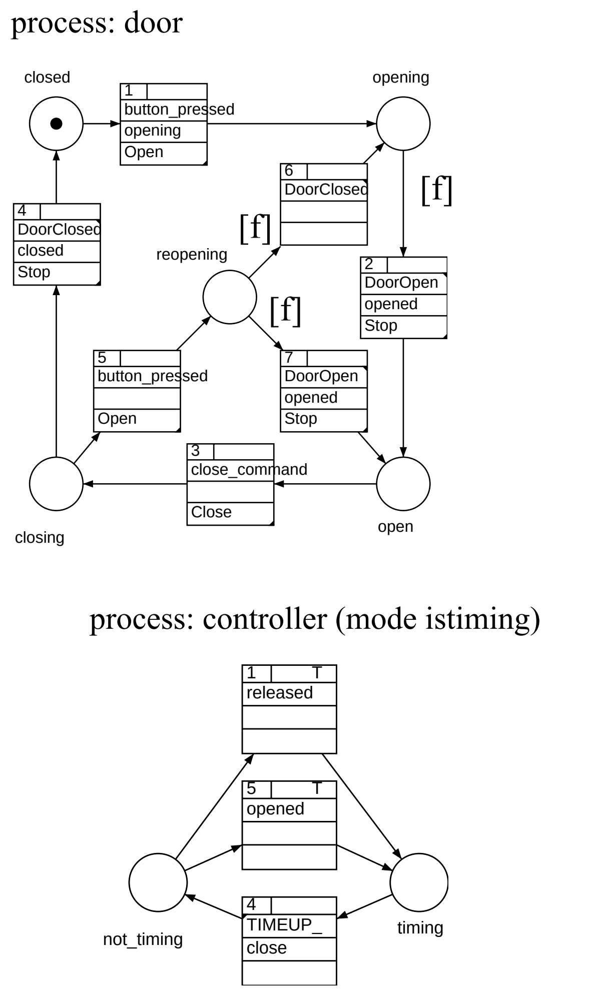
figure 6.5. CIP cluster of door control example



process: button

process: lamp

process: controller (mode nottiming)



process: door

process: controller (mode istiming)

mode setting for process: controller

		button	
		up	down
door	open	istiming	nottiming
	(other)	nottiming	nottiming

According to rule 1.4, the process transitions of process *controller* (*not_timing*, *timing*) and (*timing*, *not_timing*) (the latter triggered by *TIMEUP_*) are process-disabled by the transitions (*up*, *down*) and (*open*, *closing*) and process-enabled by (*down*, *up*) and (*opening*, *open*). Likewise, the process transition (*timing*, *not_timing*) (triggered by *pressed*) is process-disabled by the process transition (*timing*, *not_timing*) (triggered by *pressed*) and process-enabled by the process transitions (*up*, *down*) and (*open*, *closing*).

Now the construction of the reduced state graph can begin.

***ample*(s$_0$)**

The initial state vector is s_0 = [*up*, *closed*, *off*, *not_timing*].

Only one transition is enabled in s_0. Therefore *ample*(s_0) = *enabled*(s_0) = {(*up*, *down*) + (*closed*, *opening*) + (*off*, *on_dark*) + (*on_dark*, *on_bright*)} (rule R1.7).

This transition reaches s_1 = [*down*, *opening*, *on_bright*, *not_timing*].

***ample*(s$_1$)**

enabled(s_1) = {(*down*, *up*) + (*on_bright*, *on_dark*), (*opening*, *open*)}.

By rule R1.1, neither transition process-disables other transitions.

By rule R1.2, the first transition process-disables (*down*, *up*). The second transition does not process-disable other transitions by this rule.

So we only look into the suitability of the second transition for inclusion in *ample*(s_1) (rule R1.6 does not apply to the above case because (*on_bright*, *on_dark*) disables transitions).

By rule R1.3, the second transition process-disables (*timing*, *not_timing*) (triggered by *TIMEUP_*) because this could trigger (*open*, *closing*).

By rule R1.4, the second transition process-enables all process transitions of mode *istiming*. This compensates for the process-disabling of (*timing*, *not_timing*) (triggered by *TIMEUP_*). It can also process-disable transitions sending messages *released* and *opened* to controller but this can be ignored through rule R1.6. The second transition also process-disables all transitions containing (*timing*, *not_timing*) (triggered by *pressed*). But as the first transition process-disables the same, these can be ignored through rule 1.5.

Thus *ample*(s_1) = {(*opening*, *open*)}.

This transition reaches s_2 = [*down*, *open*, *on_bright*, *not_timing*].

***ample*(s$_2$)**

enabled(s_2) = {(*down*, *up*) + (*on_bright*, *on_dark*) + (*not_timing*, *timing*)}.

As there is only one element in *enabled*(s_2), rule R1.7 again applies and:

ample(s_2) = {(*down*, *up*) + (*on_bright*, *on_dark*) + (*not_timing*, *timing*)}.

This transition reaches s_3 = [*up*, *open*, *on_dark*, *timing*].

***ample*(s$_3$)**

enabled(s$_3$) = {(*up*, *down*) + (*on_dark*, *on_bright*) + (*timing*, *not_timing*), (*timing*, *not_timing*) + (*open*, *closing*)}.

The first transition disables several process-transitions through rule R1.1 (including all those including (*on_dark*, *off*)). The second is visible. Thus *ample*(s$_3$) = *enabled*(s$_3$).

Let us first follow the first transition. This transition reaches [*down*, *open*, *on_bright*, *not_timing*] which is s$_2$. A loop is closed and rule 2 is applied. No transition is included in *enabled*(s) for any state s of the loop without that transition ever being included in *ample*(s), therefore no further transitions must be added to any *ample*(s) of the loop.

But the closure of the loop also triggers another check. We are seeking to verify an *eventually* property. We therefore check whether the newly found loop is fair. It is not because the process-transition (*open*, *closing*) is fair. Therefore we proceed.

We now follow the second transition of *ample*(s$_3$).

This transition is visible and reaches s$_4$ = [*up*, *closing*, *on_dark*, *not_timing*].

This fulfils the property *eventually*(*closing*, {s$_0$, s$_1$, s$_2$, s$_3$, s$_4$}). {s$_0$, s$_1$, s$_2$, s$_3$, s$_4$} are placed into *ful*.

However, the exploration is not yet complete. The transitions enabled in s$_4$ have yet to be followed.

***ample*(s$_4$)**

enabled(s$_4$) = {(*closing*, *closed*) + (*on_dark*, *off*), (*up*, *down*) + (*closing*, *reopening*) + (*on_dark*, *on_bright*)}.

Both transitions are visible and thus *ample*(s$_4$) = *enabled*(s$_4$).

Following the first transition we reach [*up*, *closed*, *off*, *not_timing*] which is s$_0$ and is now in *ful*.

Following the second transition we reach s$_5$ = [*down*, *reopening*, *on_bright*, *not_timing*].

***ample*(s$_5$)**

enabled(s$_5$) = {(*down*, *up*) + (*on_bright*, *on_dark*), (*reopening*, *opening*), (*reopening*, *open*)}.

By rule R1.1, the second and third transition are mutually disabling so that if one is placed in *ample*(s_5), the other must too.

By rule R1.2, the first transition process-disables (*down*, *up*). The other transitions do not process-disable other transitions by this rule.

So we only look into the suitability of the second and third transitions for inclusion in *ample*(s_5) (rule 1.6 does not apply to the above case because (*on_bright*, *on_dark*) disables transitions).

By rule R1.3, the third transition process-disables (*timing*, *not_timing*) (triggered by *TIMEUP_*) because this could trigger (*open*, *closing*).

By rule R1.4, the third transition process-enables all process transitions of mode *istiming*. This compensates for the process-disabling of (*timing*, *not_timing*) (triggered by *TIMEUP_*). It can also process-disable transitions sending messages *released* and *opened* to controller but this can be ignored through rule R1.6. The third transition also process-disables all transitions containing (*timing*, *not_timing*) (triggered by *pressed*). But as the first transition process-disables the same, these can be ignored through rule 1.5.

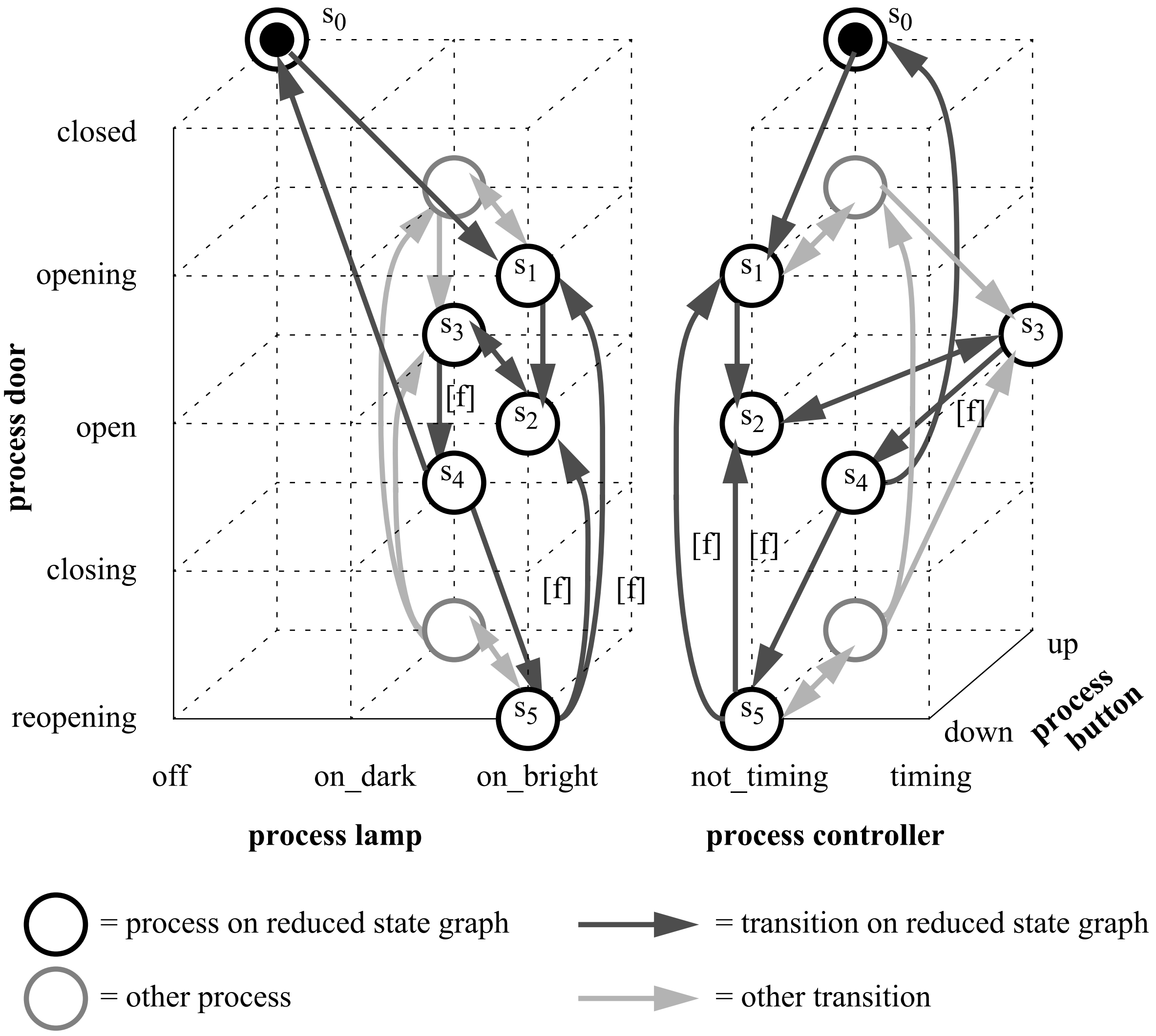
Thus *ample*(s_5) = {(*reopening*, *opening*), (*reopening*, *open*)}.

Following the first of these transitions we reach [*down*, *opening*, *on_bright*, *not_timing*] which is s_1 and is already contained in *ful*.

Following the second of these transitions we reach [*down*, *open*, *on_bright*, *not_timing*] which is s_2 and is also contained in *ful*.

All transitions of the reduced state graph have been followed and the property *eventually*(*closing*) shown to hold.

figure 6.6. Representation of reduced state graph



6.7. Discussion

6.7.1. State space reduction

The purpose of implementing partial order reduction is to reduce the state space. So to what extent is the state space reduced? Obviously this depends on the system in question.

We start by considering the best case, which is a system with the minimum possible dependency of transitions. Let:

n := number of processes
s_p := typical number of states per process
r_p := typical number of transitions per process

s_f := number of states in full state graph
r_f := number of transitions in full state graph

s_r := number of states in reduced state graph
r_r := number of transitions in reduced state graph

Best case example

For a finite state machine with a minimum possible dependency between transitions:

$s_f \approx (s_p)^n$
$r_f \approx n \cdot r_p \cdot (s_p)^{n-1}$

$s_r \approx n \cdot s_p$
$r_r \approx n \cdot r_p$

For the derivation of these formulae the reader is referred to appendix F.5.1. (page 210)

The total reduction is:

$s_r/s_f = n \cdot s_p / (s_p)^n = n / (s_p)^{n-1}$

$r_r/r_f = n \cdot r_p / (n \cdot r_p \cdot (s_p)^{n-1}) = 1 / (s_p)^{n-1}$

Assuming values of $s_p = 6$, $r_p = 9$ and $n = 12$ we obtain:

$s_r/s_f \approx 3{\cdot}10^{-8}$
$r_r/r_f \approx 3{\cdot}10^{-9}$.

The reduced state graph size would be:

$s_r = 61$ (calculated according to exact formula as shown in appendix F.5.1.)
$r_r = 108$

The savings and compactness of the reduced state graph may seem astonishing, but because the processes are fully independent, this example is highly unrealistic. If such an example should occur anyway, nothing can be learnt from checking the cluster as a whole as all properties can just as well be tested on the individual processes.

On the other hand, if we have a system where no transitions are independent and every transition process disables another, no reduction is possible at all. Fortunately, although the algorithm uses its usefulness in such a case, the strong dependency of the processes itself assures against state explosion.

The critical cases are the intermediate ones with sufficient dependency to make state space reduction cumbersome, yet insufficient to permit inherent structural state space reduction.

To create mathematical models of these requires a large number of assumptions, the basis of which may vary from system to system. It is therefore preferable to allow observations to speak for themselves.

An example [44] was tested and a state space reduction of 40% was achieved for a similar execution time. In another example from the same source (albeit implemented with conditions where inspections could have been used) no significant improvement was observed.

6.8. Conclusions

The methods of partial order reduction are applicable to CIP model checking. The frequent interactions between the processes which may at first sight seem a disadvantage for partial order reduction are turned into an advantage through the introduction of the *degree of process-enabledness*. This allows more powerful reduction in the CIP context than would have been achieved using ‘off the shelf’ algorithms.

7. Implementation

7.1. Introduction

This chapter aims to present some of the methods used in implementing the CIP model checker and to give the reader an impression of how the model checker is handled. It is not intended to be a full guide to the code or a full manual for the use or maintenance of the model checker.

7.2. Implementation methods

7.2.1. States and state vectors

Although the CIP model checker is nested within the code of the CIP Tool and many common variables are used, and the full behaviour of the system can be derived from the stored model, it is not efficient to do so at run time.

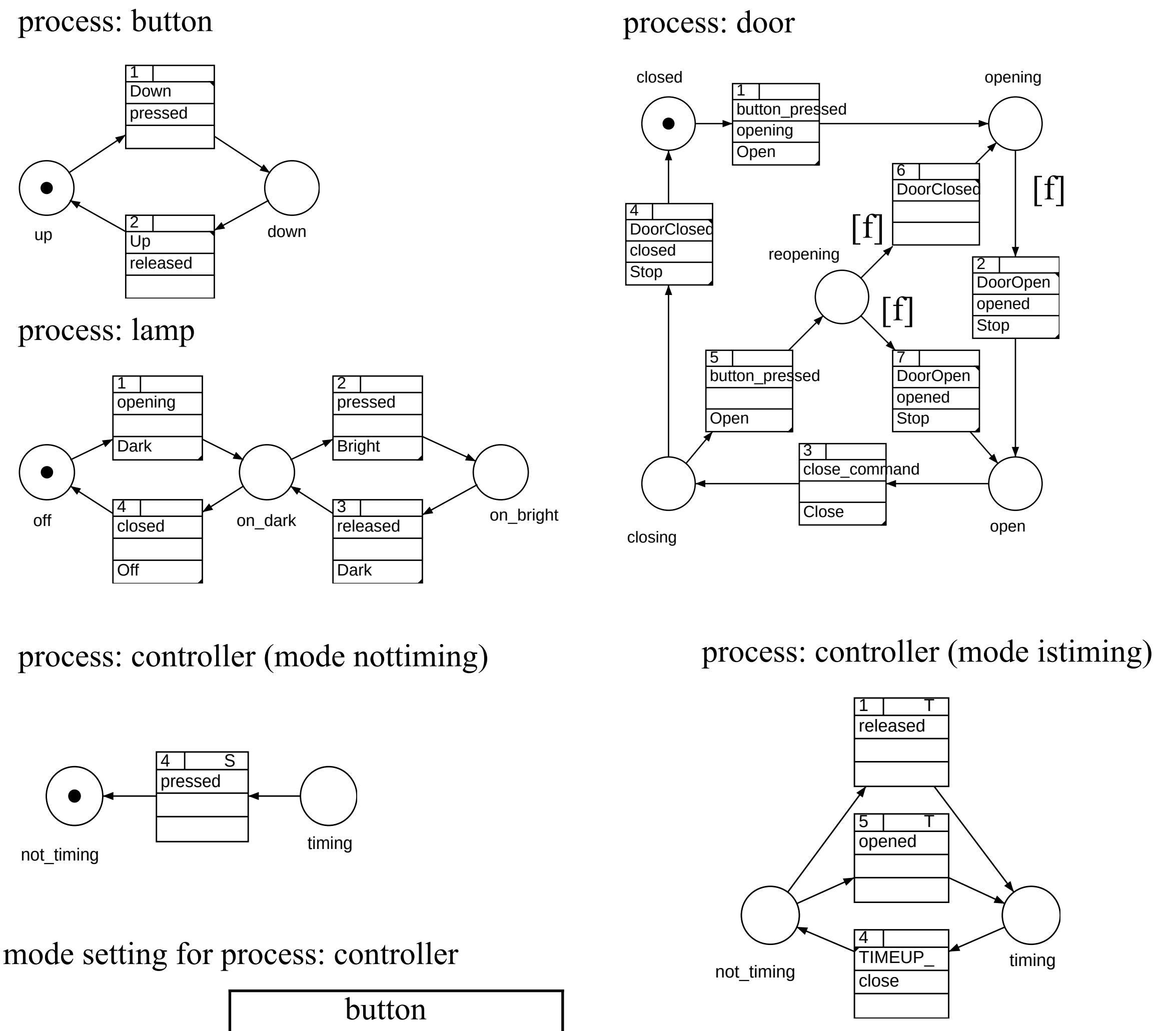
For this reason the stored model needs to be mapped in a way which renders access as fast as possible during run time.

The state vector is stored as an integer. This reduces the space required to a minimum and makes it easy to check whether the state vector is already contained in a list (especially if the list is sorted).

The bits of this integer represent the states of individual processes.

In figure 7.1. the door control example is shown which has already been amply discussed in chapter 5. and also used in chapter 6.

figure 7.1. CIP cluster of door control example



process: button

process: door

process: lamp

process: controller (mode nottiming)

process: controller (mode istiming)

mode setting for process: controller

		button	
		up	down
door	open	istiming	nottiming
	(other)	nottiming	nottiming

This example has four processes. The first process is *button* which has two states. A single bit (bit 0) is thus sufficient to describe this.

up = 0
down = 1

The next process is *lamp* which has 3 states. 2 bits (bits 1 to 2) are required to describe this. The codes advance in steps of 2 as the lowest bit is used by *button*.

off = 0
on_dark = 2
on_bright = 4

The codes for processes *door* and *controller* are allocated likewise:

closed= 0
opening = 8
open = 16
closing = 24
reopening = 32

not_timing = 0
timing = 64

The state vector integer is formed by summing the codes of the individual states. Thus [*down*, *reopening*, *on_bright*, *not_timing*] is represented by the state vector 1 + 32 + 4 + 0 = 37.

7.2.2. Process transitions

Just as state vectors need to be coded efficiently, so do transitions need to be executed efficiently.

The structure of the process is stored in an array called *transStructArray*. This array[1] is divided into blocks with one block for every trigger the process can accept. It contains integers which must be added to the state vector to determine the post-state for that process transition.

Below, the change *transStructArray* of *button* is shown. This array has two blocks of size 2 each. The first block contains transition structures triggered by *Down*, the second block contains transitions triggered by *Up*. The first address of such a block is the *triggerOffset*.

table 7.1. changeArray *for process* button

```
transStructArray
1: trCas(+1, [lamp -> 4, door -> 1, controller -> 1])
2: trNil
3: trNil
4: trCas(-1, [lamp -> 7, controller -> 3])
```

triggerOffset(*Down*) = 1
triggerOffset(*Up*) = 3

[1] Note that in SmallTalk arrays begin at 1 and not at 0 as in C.

trCas[1] indicates a transition casting pulses to other transitions. The first argument is the change in the state vector. The second argument is discussed in section 7.2.3.
trNil indicates no transition.

Down triggers no change of state if *button* is in state *down* (`transStructArray[2] = tsNil`). *Down* triggers a change of state to *down* if *button* is in state *up* (`transStructArray[1] = tsCas(+1, ...)`).

The formula for determining the transition to execute is:

transition to execute
= *transStructArray*[*triggerOffset*(trigger)
+ ((stateVector *AND processMask*) *SHIFT processShift*)]

AND and *SHIFT* are bitwise logical functions and *processMask* and *processShift* are process dependent constants. In this example they are

table 7.2. processMask *and* processShift

process	processMask	processShift	bitWidth
button	1	0	1
lamp	6	1	2
door	56	3	3
controller	64	6	1

Because all functions used are very basic, a fast execution is assured.

Example 1:

The state vector is [*down*, *reopening*, *on_bright*, *not_timing*] coded by 1 + 32 + 4 + 0 = 37. The event *Up* is sent to process *button*.

transition to execute
= *transStructArray*[*triggerOffset*(trigger)
+ ((stateVector *AND processMask*) *SHIFT processShift*)]
= *transStructArray*[3 + ((37 *AND* 1) *SHIFT* 0)] = *transStructArray*[4]
= *trCas*(-1, ...).

The new state vector is 37 - 1 = 36.

Note that this is not a cluster transition but only the first component of that transition (cluster transitions will be discussed in section 7.2.3.).

[1] For a full explanation of these different transition types see appendix G.1. (page 213)

Example 2:

The state vector is 36. The inPulse *released* is received by process *lamp*.

table 7.3. changeArray *for process* lamp

```
changeArray
1:  trTra(+2)
2:  trNil
3:  trNil
4:  trNil
5:  trTra(+2)
6:  trNil
7:  trNil
8:  trNil
9:  trTra(-2)
10: trNil
11: trTra(-2)
12: trNil
```

triggerOffset(*opening*) = 1
triggerOffset(*pressed*) = 4
triggerOffset(*released*) = 7
triggerOffset(*closed*) = 10

trTra is similar to *trCas* only that no outPulses are cast.

transition to execute
= *transStructArray*[*triggerOffset*(trigger)
+ ((stateVector *AND processMask*) *SHIFT processShift*)]
= *transStructArray*[7 + ((36 *AND* 6) *SHIFT* 1)] = *transStructArray*[9]
= *trTra*(-2).

The new state vector is 36 - 2 = 34.

7.2.3. Pulse propagation

Additionally to the information on the change in state vector, *trCas* contains information on which processes receive pulses, what the trigger element is and in which order these are sent. This is held in an array.

Example:

The state vector [*down*, *reopening*, *on_bright*, *not_timing*] = 37 receives the trigger *Up*. As described in section 7.2.2. (first example) this activates the transition structure *but-*

ton.transStructArray[4] = *tsCas*(-1, [*lamp* -> 7, *controller* -> 3]) changing the state vector to 36.

Then, trigger 7 is sent to process *lamp*. As described in section 7.2.2. (second example) this activates the transition structure *lamp.transStructArray*[9] = *tsTra*(-2) changing the state vector to 34.

Lastly, trigger 3 is sent to *controller* where it does not trigger a transition. The state vector remains 34.

The post-state of the cluster transition is 34 = 0 + 32 + 2 + 0 = [*up*, *reopening*, *on_dark*, *not_timing*].

7.2.4. Conditional transitions

There are two types of conditional transitions, those which are deterministic because all relevant information is available to the model checker and those which are non deterministic because information is missing. Consequently, the CIP model checker also has two types of structure for handling conditional transitions.

The conditional dependency of transitions is defined by tables containing all appropriate transitions. In table 7.4. the mode settings for process *controller* are shown. The process transition (*not_timing*, *timing*) is enabled and disabled accordingly.

table 7.4. mode setting table for process controller

		button	
		up	down
door	closed	nottiming	nottiming
	opening	nottiming	nottiming
	open	istiming	nottiming
	closing	nottiming	nottiming
	reopening	nottiming	nottiming

table 7.5. corresponding transition identification table

		button	
		0	1
door	0	tsNil	tsNil
	8	tsNil	tsNil
	16	tsTra(+64)	tsNil
	24	tsNil	tsNil
	32	tsNil	tsNil

Similarly to the method of obtaining the *processStateCode* of section 7.2.2. we can identify the correct field of the table through mask and shift operations.

```
modeSettingField()
{
    var field := 1;
    var usedBits := 0;
    for proc ∈ inspected processes in order do
    {
        field += (stateVector AND proc.processMask) SHIFT
          (proc.processShift - usedBits);
        usedBits += proc.bitWidth;
    }
return field;
}
```

Example:

For the state vector [*up*, *reopening*, *on_dark*, *not_timing*] = 34 we wish to ascertain whether (*not_timing*, *timing*) can be executed. The inspected processes are *button* and *door*. After the first traversal of the *for* loop,

field = 1 + (34 *AND* 1) *SHIFT* 0 = 1
usedBits = 0 + 1.

After the second traversal,

field = 1 + (34 *AND* 56) *SHIFT* (3 - 1) = 9

A glance at table 7.4. shows that the 9th field is indeed the mode setting for this situation.

A conditional deterministic transition structure of this type is called *trCoD*.

In the actual realisation, the values for *processMask* and *processShift - usedBits* are stored locally in arrays *maskArray* and *shiftArray* to reduce access and evaluation times.

Conditional transition structures can also be non-deterministic from the point of view of the model checker. This can happen when

1) code conditions decide which of several transitions is to be executed (see section 2.3.5. on page 19)

2) or when one or several of the inspected processes is missing through *cluster reduction* (see section 5.5. on page 84)

3) or when the model is not yet finished and relevant information is missing.

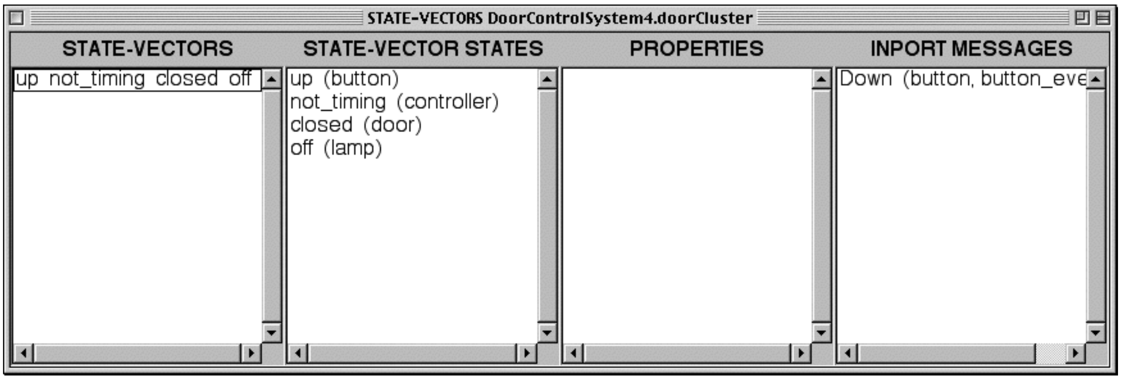
In such cases all possible transitions are evaluated and all valid post-states returned.

7.3. The CIP model checker

7.3.1. The user interface

The main user interface of the Model Checking part of CIP is shown in figure 7.2.

figure 7.2. main Model Checking window



up not_timing closed off

up (button)
not_timing (controller)
closed (door)
off (lamp)

Down (button, button_eve

This window has four list views. The list view on the left lists state vectors. Only one such state vector can be active at any one time. The active state vector can be changed by selection in the list. When the model checker is first initialised only the initial state is contained in this list and this is the active state vector. This is the situation illustrated.

The second list view shows the process states making up the active state vector. The corresponding process is shown in brackets.

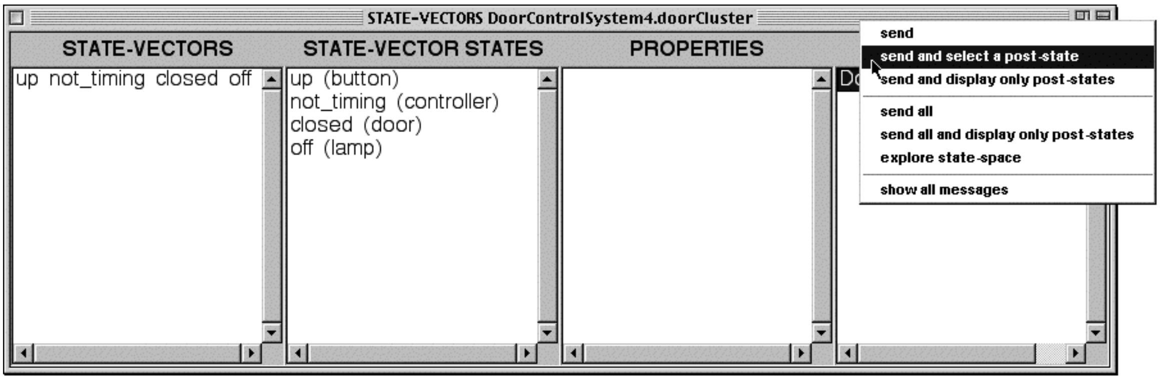
The third list view lists the properties being tested. These will be discussed in section 7.3.5.

The fourth list view lists the messages which can be sent to the cluster.

7.3.2. Simple testing

The fourth list can be used to send messages to the active state vector. The example of figure 7.3. shows the selected message being sent. The result is shown in figure 7.4.

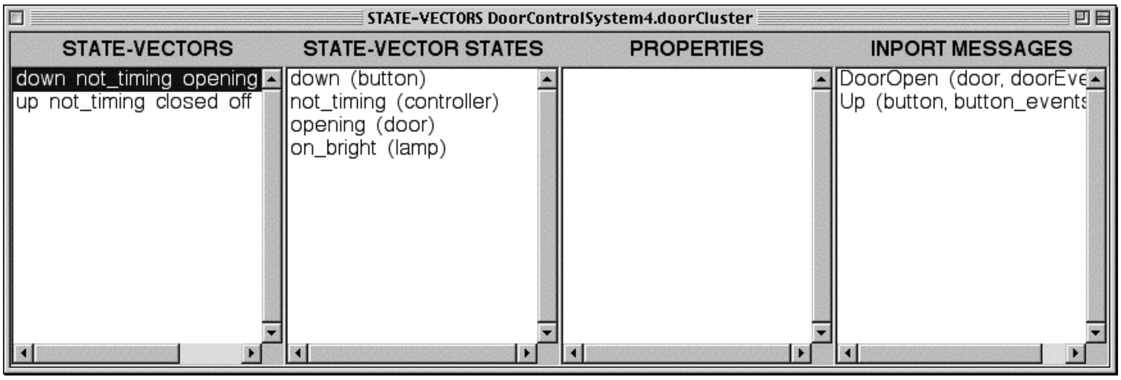
figure 7.3. Menu in message list selected



STATE-VECTORS DoorControlSystem4.doorCluster
send
send and select a post-state
send and display only post-states
send all
send all and display only post-states
explore state-space
show all messages

STATE-VECTORS	STATE-VECTOR STATES	PROPERTIES	
up not_timing closed off	up (button) not_timing (controller) closed (door) off (lamp)		Do

figure 7.4. post-state of transition is displayed.



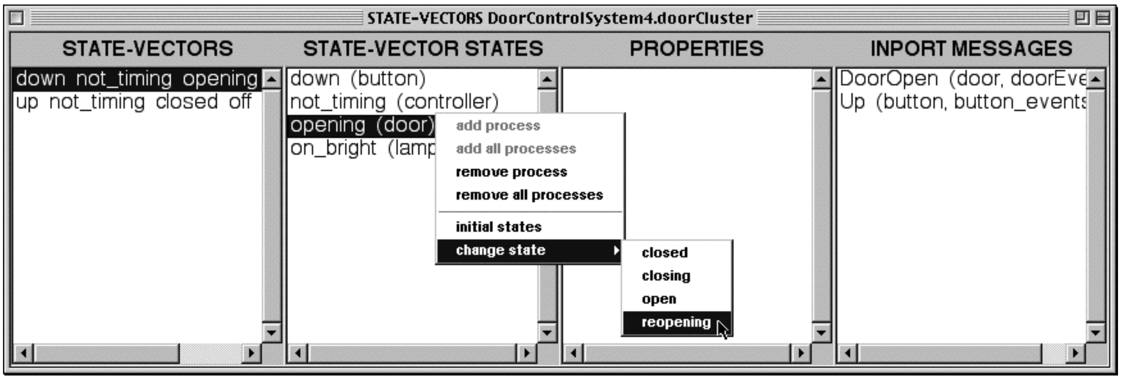
STATE-VECTORS	STATE-VECTOR STATES	PROPERTIES	INPORT MESSAGES
down not_timing opening up not_timing closed off	down (button) not_timing (controller) opening (door) on_bright (lamp)		DoorOpen (door, doorEve Up (button, button_events

Note that because the active state vector has changed, the list of process states and the list of available messages have also been adapted.

This function can be used to test the system or components of the system during development. In contrast to the simulation tool provided with CIP, the model checker does not require the model to be fully specified. Any triggerable transition with a pre- and post-state can be simulated as soon as it is created.

Process states can be changed manually using the menu of the second list. This function is useful if the test should commence at a cluster state or cluster states other than the initial state.

figure 7.5. Process state is changed manually.



STATE-VECTORS DoorControlSystem4.doorCluster

STATE-VECTORS	STATE-VECTOR STATES	PROPERTIES	INPORT MESSAGES
down not_timing opening	down (button)		DoorOpen (door, doorEve
up not_timing closed off	not_timing (controller)		Up (button, button_events
	opening (door)		
	on_bright (lamp		

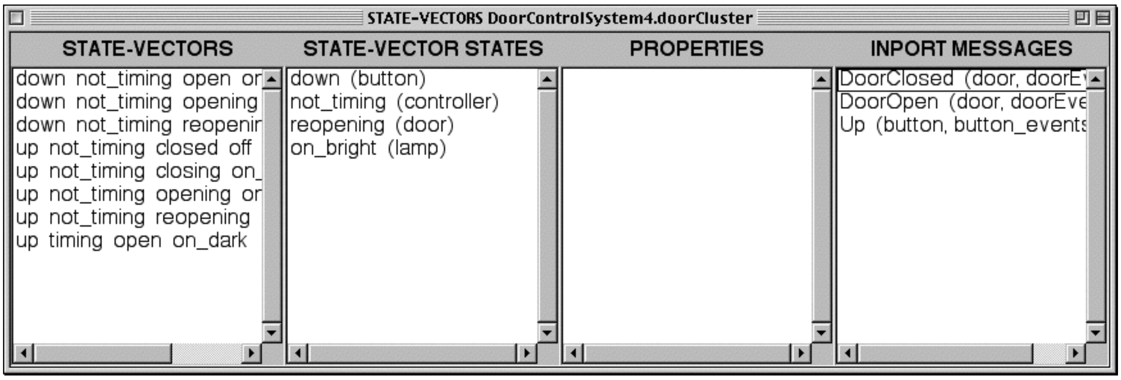
The same menu can also be used to remove (and re-add) processes if required. This function is especially useful for *cluster reduction* (see section 5.5. on page 84).

7.3.3. State space traversal

The state space traversal can be launched from the menu of the fourth list by selecting *explore state space* (see figure 7.3.).

After such a state space traversal the window may look as follows:

figure 7.6. Window after state space traversal



STATE-VECTORS DoorControlSystem4.doorCluster

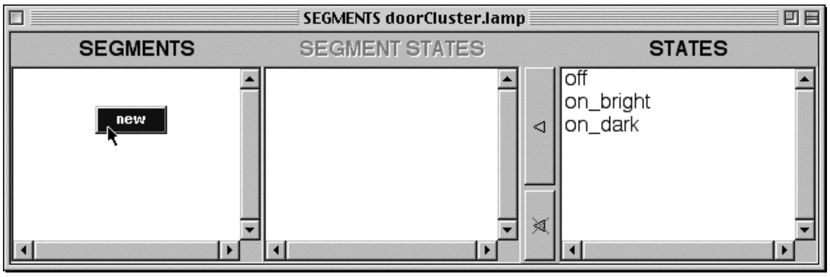
STATE-VECTORS	STATE-VECTOR STATES	PROPERTIES	INPORT MESSAGES
down not_timing open or	down (button)		DoorClosed (door, doorE
down not_timing opening	not_timing (controller)		DoorOpen (door, doorEve
down not_timing reopenir	reopening (door)		Up (button, button_events
up not_timing closed off	on_bright (lamp)		
up not_timing closing on_			
up not_timing opening or			
up not_timing reopening			
up timing open on_dark			

All state vectors found are listed in the state vector window. If this list gets very long it is automatically truncated to prevent problems with the graphic display. The cluster states remain in memory however.

7.3.4. Segments

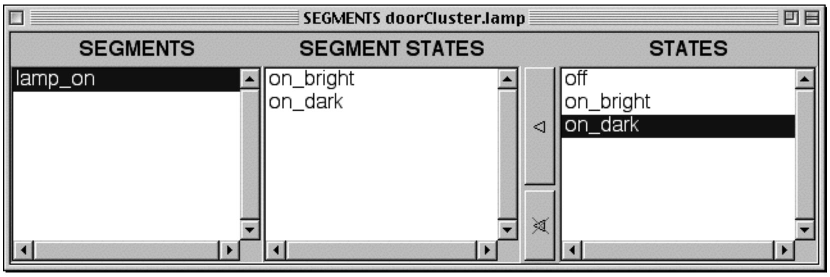
Segments are used to mark sections of the state space which interest us for Model Checking purposes (see section 5.6. on page 89). They are defined in CIP using the segment editor.

figure 7.7. The segment editor for process lamp



A segment is made up of states of the same process. Segments are created by the user in the list on the left and states added or removed as required to the corresponding list in the centre by selection or deselection from the list on the right.

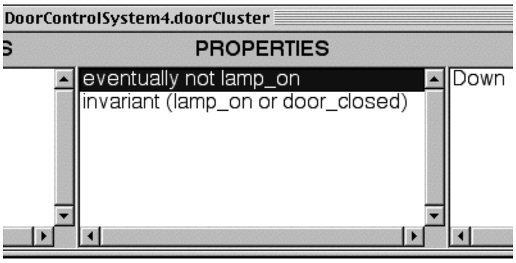
figure 7.8. Definition of the segment lamp_on



7.3.5. Properties

The properties to be tested are defined by the user in the *properties* list of the *Model Checking* window. The syntax is based on that introduced in section 3.5. (page 44).

figure 7.9. The properties list of the main Model Checking window



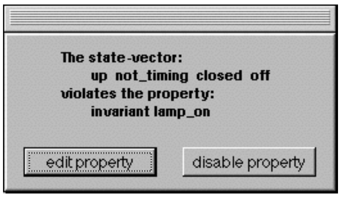
eventually not lamp_on
invariant (lamp_on or door_closed)

The properties are currently entered in text mode and parsed. This approach was chosen because it was considered to be more flexible and faster to use than combining elements from a browser. However, this can easily be changed if required.

The properties defined in figure 7.9. combine basic segments using boolean operations. Supported operations are *and*, *or*, *exor*, *exnor*, *not*, *1* (true) and *0* (false).

When a property is defined or edited, the model checker checks that no state vector of the list violates this. If so, an error message is generated of the style of that shown in figure 7.10. Of course, only properties not requiring path knowledge such as *invariant* can be verified at this point and for these states.

figure 7.10. Error message when property violated



The state-vector:
up not_timing closed off
violates the property:
invariant lamp_on

While state exploration is in progress, or while the user is simulating the system, these properties are continuously checked. Any violation prompts an error message similar to that above. Additionally a log file is saved to disk indicating the path that caused the violation.

If the user chooses to edit the property, the traversal must be begun again as already visited structures may violate the new property and transition visibilty may be affected (see section 6.3.2. on page 103).

8. Conclusions

8.1. What has been achieved?

8.1.1. The contribution of this thesis

This thesis sets out from two methods designed to support the developer of embedded system software: CIP and Model Checking. It bridges the gap between the two by giving CIP users access to Model Checking without requiring them to learn Model Checking techniques specifically. The model checker uses formulations and abstractions which are easily understood from within the CIP context.

The synthesis is achieved by extracting the extended state machine structure from the CIP model and using this for Model Checking. The user does not leave the CIP environment or context when using the model checker. Data pertaining to the model checker is defined and handled just as data for any other function of the tool would be. For example, fairness constraints are allocated to transitions and accessed within the CIP environment just as any other CIP related attribute would be allocated or defined.

Independently of this, the extraction of the extended state machine from the CIP model means that the former is closely based on the latter. So, rather than building or using a model checker which we can confront with any type of interaction, the CIP model checker is specifically oriented towards the type of problems which could occur in a CIP environment. For example, in CIP the interactions between processes of a cluster occur in a strictly defined order known as *run to completion semantics*. Most model checkers are prepared to accept interactions in any order so greatly increasing the number of possible interactions and system states. To limit this, we would have to introduce additional structures in another model checker to reduce to the CIP behaviour. But this reduction goes beyond a cutting back of the total possible interaction patterns. The partial order reduction possibilities of CIP go beyond those commonly used by using CIP-specific interaction restraints.

It is not the primary goal of this thesis to duplicate the full expressive and operational power of advanced model checkers. It is to show how Model Checking can be performed on the CIP model using what is basically the existing structure of the model. The necessary transformation stands at the centre of this thesis.

Some experience has been gained with the CIP model checker in analysing real projects. To date these investigations have concentrated on checking that these systems can be traversed (that their size is not too big and that the termination does indeed terminate). Some test properties were also verified but to date no productive use has yet been made of the model checker in such projects, since the model checking features have not yet been added to the commercial version of CIP Tool.

8.1.2. The software concept: summary

In a CIP model, structures such as processes and states already exist. This is a mixed blessing: whereas most other model checkers have the luxury of being able to define these from scratch, this was not necessary in this thesis. Besides the obvious advantage, this posed problems in that a CIP model does not behave entirely like a generic extended state machine. Especially, inter-process communication and scheduling is more restricted. Transitions cannot be triggered in an arbitrary order and through *run to completion semantics*, sequences of triggers cannot be interrupted. The possible interactions of the CIP finite state machine are more restricted than they would be in a 'classic' finite state machine. If a 'classic' model checker were to be applied to CIP, these restrictions would have to be enforced by additional constructs. In the CIP model checker where development starts from the existing CIP model, these restrictions are implemented generically.

To apply Model Checking to the CIP model, first a basic interpreter had to be created. This identifies the post-state of a cluster transition with a given pre-state and trigger. Message passing between the processes is handled as it is in a system designed with CIP Tool and transitions scheduled in the same order that such a system would schedule them. This element of the model checker is useful in its own right as it permits the user to 'simulate' the behaviour of the model without having to generate any code and can be used even when the model is largely incomplete, so enhancing the supportive power of the tool during the design phase.

This interpreter also provides the basis for the real model checker by permitting state explorations. The basic interpreter can work with incomplete models (only a single transition need be defined and this can already be executed), so all functions basing on this including the full model checker can be used on partially complete subsystems during the development process. The model checker can so be used on the fly to verify whether any sub-systems or structures the user is implementing perform the desired task as required.

The CIP model checker allows the user to manually exclude processes from the check to reduce the complexity of the problem and concentrate on the section being checked.

8.1.3. Some achievements

The state explosion problem is addressed by a range of measures. Firstly, CIP itself, through strong dependencies between processes and strict scheduling makes a significant contribution to reducing this problem.

Additionally to this, the *cluster reduction* approach permits further reduction of the state space by omitting selected processes. This omission can lead to the introduction of additional transitions in the cluster state space but also to the loss of existing transitions. The latter (which is rather CIP specific, in systems without such strict scheduling it would not occur) is tackled by adding dummy processes with a single state (therefore taking no additional space in the state vector). The former cannot be avoided entirely but can be accommodated by an appropriate interpretation of results. Additional paths may lead to false counter-examples or false examples being found. For some properties these additional paths may be on the safe side, with the property failing where it holds for the full system. In other cases such paths may lead to a property holding where it would fail in the full system. Which case is appropriate varies from property to property and special treatment was required to prevent the model checker from returning false results. In some cases certain properties may not be tested when certain processes are removed.

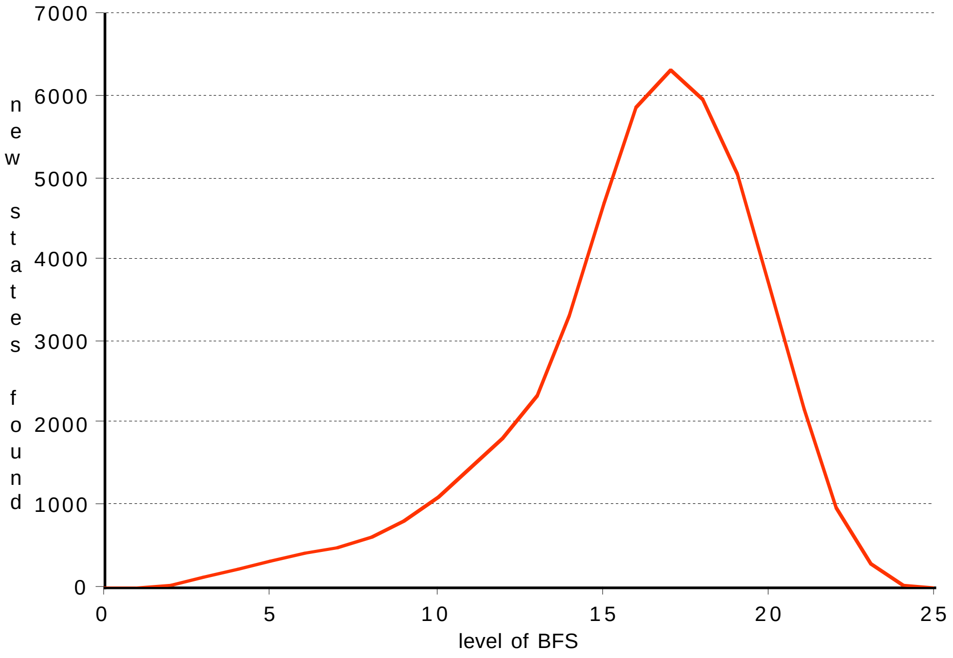
The third measure against state explosion is *partial order reduction*. This is based on reducing the number of paths of the system by omitting equivalent paths. However, in this thesis an implementation called *degree of process-enabledness* is used. This helps decide which transitions should be followed and which can be omitted.

Also implemented in this thesis is an approach to fairness which permits fairness constraints to be applied to transitions independently of their process. This supports the CIP modelling philosophy in that clusters can continue to be decomposed into processes along functional lines rather than through fairness motivations.

8.1.4. Some results

In figure 8.1. the result of a breadth first search of a sample system is shown. This 14-process system was developed and is used by a CIP customer in industry [44]. Normally the state space is searched depth first as the recognition of loops is simpler this way. But to show the state of the progress and give some idea about the structure of the system, a breadth first traversal was applied in this case. The *level of BFS* (x-axis) shows the number of transitions needed to find the number of *new states found* (y-axis). As can be expected, the number of states found per level increases in an exponential style at first as the state graph branches out. Growth drops as more and more of the states found are identical to those found previously and finally tails back to zero as the exploration is completed.

figure 8.1. Breadth first search of real system [check reference].



This traversal took about 15 minutes on a 500MHz Macintosh G4. No use of partial order reduction or any other state space reductions were activated. Many other model checkers could have traversed the same state space in a much shorter time. The drawback of the CIP model checker is that it is implemented in SmallTalk which is a high level language. Implementation in a lower level language would have provided considerably greater speed. SmallTalk was chosen because CIP Tool was already implemented in this language and this continuity provided the best possible integration and access to data structures. The advantages of higher level languages lie in the possibility to create well structured and documented programs, but the sheer mass of data requiring processing could have justified a lower level language in this case. Many model checkers achieve a high efficiency by not checking the model themselves but by generating code which can be compiled into a program to solve the one and only task which is required of it. This method combines the strengths of both approaches and is worth giving greater consideration in a further development of the project. Additionally, the CIP model checker does not make use of all Model Checking constructs such as BDDs which could greatly enhance performance, but adding these at a later point should be relatively straightforward.

8.2. A self critical appraisal

8.2.1. Implementation of the project

The SmallTalk code of the CIP Tool is well structured, documented and maintained. This facilitated the understanding of the code to an external observer such as myself who had previously not been involved with the project. Care was taken to maintain the same level of transparency and maintainability in implementing the additions and modifications necessary for the model checker so that these can continue to be used and developed as the tool continues to develop.

8.2.2. State explosion problem

From the outset of this thesis it was clear that the state explosion problem would be a major issue. As it is not clear how many states will be found during a state space traversal it can not always be predicted whether a search can be terminated or not. The full extent of the issue was, however, underestimated and the checking of several larger systems had to be aborted. It seems that the performance of SmallTalk also degrades somewhat as the memory used grows. As it is not visible how many of the functions used are implemented it is not always possible to make statements about the relative efficiencies of algorithm variants.

Had this been recognised earlier, more effort would have been put into making fuller use of model checking concepts such as BDDs. Now this must be referred to future development phases.

8.2.3. Overall appraisal

Although the system has yet to be presented to ‘guinea pigs’ among the CIP user community, in general I am convinced that the implementation is successful, that it is robust and that it addresses real verification needs.

8.3. Possibilities for further development

8.3.1. Command line interface

It is difficult to sell a new tool without there being sound evidence for its correct performance. Analytical analysis alone does not suffice to show that the execution tester really does behave identically to the generated and compiled code. Implementation errors could have occurred. In section 1.5.3. (page 8) a method for demonstrating confidence in the equivalence was described. Both systems execute in parallel and the results are compared. With a command line interface such a process could be automatised for a very large number of systems and inputs (see figure 1.2. on page 9).

8.3.2. Interfacing with SPIN

In sections 1.4.3. and 4.7.2. the possibilities for making CIP Tool work more closely with SPIN were suggested. It was mentioned that it will soon be, at least in principle, possible to feed the generated C-code to SPIN for checking. It was also pointed out that this is not generally desirable due to the different level of abstraction. In generating C-code we are moving a level of abstraction away from the model. Why run a complex verification of the abstraction when we can do a simple verification of the original? Especially bearing in mind the human errors that can be made when specifying properties for the complex abstraction?

One exception to this recommendation is when the CIP model contains C-code *operations* and *conditions* (see sections 2.3.4. and 2.3.5.) which have a vital influence on the property being tested. The reader will recall that the CIP model checker will not attempt to parse such ‘hand written’ code fragments. Any decision based on such a condition is considered non deterministic by the CIP model checker. If this is the case, this rather cumbersome passage via C-code may be the only possibility. On the other hand, re-design or deployment of other CIP structures may reduce the importance of such structures and make model checking possible in CIP.

8.3.3. Generating meta-code for other model checkers

As CIP is capable of automatically generating C-code or Java it would be but a relatively small step to also include an option for generating meta-code for model checkers. This way the full expressive power of large model checkers could be placed at the disposal of the CIP user. The underlying concepts used in this project are all borrowed from or based on those of large scale model checking and to bridge this gap by writing an interface would be a relatively simple to achieve.

Appendix A: Sequentiality and fairness

A.1. Sequence of transitions

A.1.1. Time and sequentiality

In section 3.2.6. the basic structure of state diagrams was introduced to describe systems. One interpretation of such a state diagram is that when the implementation is being observed, the active state can be mapped in function of time. When the system is initialised, the active state is the initial state. This state changes in accordance with the events the system receives. If all transitions are deterministic, then the active state of the system is a function of the sequence of events that have been received up to that point. During the course of the system's execution, exactly one path is traversed.

Looking at the system for Model Checking purposes, however, we are not interested in one path, but we want to be able to make general statements about all paths.

When looking at paths, their time schedules are of little importance. Important is the sequentiality of transitions (and thus events) which define the paths. This view of paths is known as *temporal logic*.

A.1.2. Example of an interpretation problem

In section 3.5.5. (page 46) we stated that for the example system of figure 3.3. and 3.4. (pages 43 and 43) the property *eventually*(B) holds. Does this mean that for any sequence of events we know that at all times we are sure that B will be fulfilled at some point in the future?

Suppose that at the time our observation begins, the system is in state s_0. In this state, the proposition B is not fulfilled. Let us suppose that no further events occur for the rest of time. On account of this, the proposition B is never fulfilled by the system.

Is this a valid counterexample disproving the property? This is discussed in sections A.1.3. and A.2.

A.1.3. Infinite sequences

Liveness requirement: We require, that whatever the state of a Kripke structure, a further transition will always be triggered.

This rule prevents the system from freezing in a state with enabled transitions.

In some cases this rule may cause a problem. Certain transitions may never be triggered. For example if we have a system with an emergency alarm switch, it would be incorrect to say that it is certain that this switch will ever be used. This is also true when a state is reached where no other transition is possible.

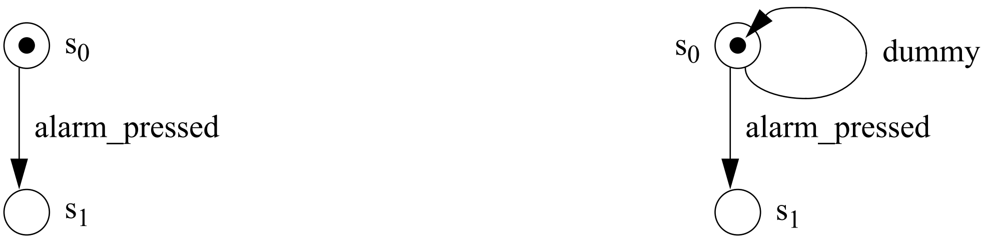
To illustrate this better, let us suppose we have a system with only one transition enabled in the initial state, and that transition is triggered by the pressing of the emergency alarm switch. According to the above statement, it is certain that this event will occur at some point in the future.

Now, we compare this to a second system where another transition triggered by another event is also enabled in the initial state. Now it is no longer certain that the emergency alarm will be pressed.

This disparity is unsatisfactory. Whether or not it is certain that the emergency alarm will be pressed is an absolute truth and should not depend on the presence or absence of alternative transitions of the system. One remedy would be to soften up the above rule and allow event sequences to terminate although further transitions are possible. This could lead to false counterexamples as described in section A.1.2.

Another remedy is to ask the user to introduce dummy transitions (where the pre- and post-states are the same and no action is taken) as alternatives to such transitions so that full paths can exist where they are not executed. The user should recognise their necessity from false counter-examples generated by the model checker. The introduction of such loops can lead to an *eventually* property being wrongly failed but not to an *eventually* property being falsely verified. Thus the addition is safe.

figure A.1. Two systems compared



In the system on the left, the property eventually*(s_1) is fulfilled because all full paths of the system each include the transition (s_0, s_1). This implies that the emergency alarm is sure to be pressed. In the system on the right, the same property is not fulfilled because there is a full path which continuously follows the loop (s_0, s_0, s_0, ...). So the emergency alarm is not sure to be pressed.*

A.1.4. Sequentiality

The rule introduced in section A.1.3. rules the correlation between time and sequentiality. However, it is well worth looking into this thematic a little deeper. The following case, which was first presented by the Greek philosopher Zeno, illustrates well the difference between the time and sequence based views of events and raises some interesting points on this subject.

A.2. Zeno’s Paradox of Achilles and the Tortoise

A.2.1. Zeno

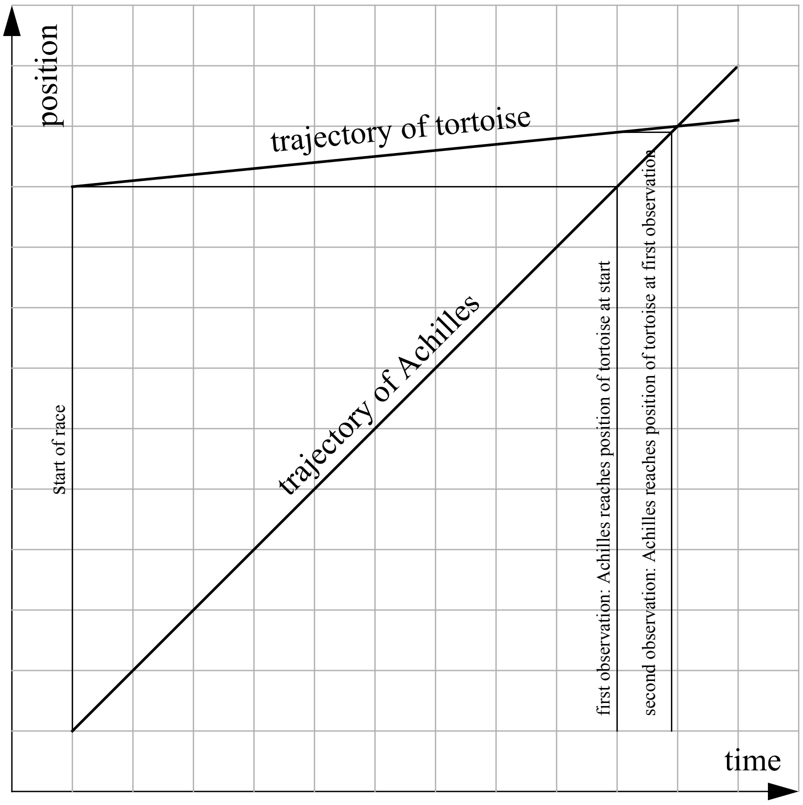
Zeno of Elea lived from circa 495 to circa 435 B.C. He presented a total of 40 paradoxes, all of which reach absurd conclusions using apparently logical proof. None of his writings survive, but some of his paradoxes were paraphrased by Plato, Aristoteles and other commentators [25]. Arguably the best known of these paradoxes is the Paradox of Achilles and the Tortoise [43].

A.2.2. The paradox

In this paradox, Zeno asks us to imagine that the athlete Achilles is to race a tortoise. Achilles is able to run ten times faster than the tortoise, but the tortoise is given a head start. Zeno provocatively states that Achilles is never able to overtake the tortoise. He argues, that when Achilles reaches the point previously occupied by the tortoise, the tortoise has also made progress and consequently still occupies a more advanced position than Achilles. By the time Achilles has reached the new position of the tortoise, the tortoise has again advanced. In this way, Achilles is unable to overtake the tortoise with any number of iterations, and consequently he is unable to overtake the tortoise at all.

Of course, studying the problem kinetically, we could plot the time-space trajectories of Achilles and the tortoise, and see that Achilles would overtake the tortoise after having travelled ten ninths of the initial distance between the contestants. That is not the problem, however. The problem is finding the fault in Zeno’s reasoning.

figure A.2. Time-space diagram showing progress of Achilles and tortoise in Zeno's paradox and the events triggering the first two observations

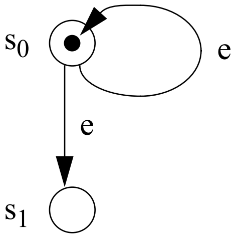


A.2.3. What does all this have to do with Model Checking?

We only make an observation when Achilles has reached the position occupied by the tortoise at the last observation. The observer is inactive at other times. We could thus say that Achilles reaching such a position is an event triggering the observer.

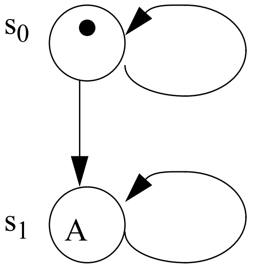
Although state diagrams were not invented until long after Zeno's time, the state diagram he is inferring in his explanation is the following.

figure A.3. State diagram of Achilles versus tortoise race



s_0 = tortoise leading.
s_1 = Achilles leading.
e = event: Achilles reaches position occupied by tortoise at last e.
conditional transitions:
(s_0, s_1) if: position(Achilles) > position(tortoise)
(s_0, s_0) ELSE

figure A.4. Kripke structure of Achilles versus tortoise race



A = proposition for “Achilles has overtaken tortoise”.

If we do not include the kinematics of the situation in the model, the triggered transition is non deterministic. *reacheable*(A, s_{init}) is fulfilled but *eventually*(A, s_{init}) is failed. This means that Achilles is capable of overtaking the tortoise but it is not certain that he will.

If we do include the kinematics in the model, both *reacheable*(A, s_{init}) and *eventually*(A, s_{init}) are false as Zeno tells us. We thus obtain a result which clearly contradicts reality.

With kinematics included in the model, when we wait sufficiently much time, Achilles is certain to win. But even if we wait infinitely many events, the tortoise will still be ahead.

This example shows how the time and event oriented approach to the problem can deliver completely different results.

A.2.4. Discussion

Is the event oriented approach sufficient to model a time oriented process, such as this race? In a real-time system, the events would occur at ever shorter intervals and ultimately occur at such short intervals that the hardware would be unable to handle them faster than they are queued in the buffer. The event buffer would then overflow, possibly leading to malfunction, or otherwise to events being lost. In either case, the result would no longer be reliable.

This statement, however, is itself based on a time oriented approach, namely that the hardware is running in a time driven and time oriented environment. In a hypothetical event driven environment where time has no significance, Achilles would never overtake the tortoise and the hardware would never cause such constraints.

But then it could also be argued that the kinematics of the race are also anchored in a time oriented view of the universe and in a universe without time but with only sequentiality, no statement could be made as to the absolute positions of the contestants. We would only know that they have advanced between events. The transition would thus be non deterministic and so would the outcome of the race.

A.2.5. Correspondence of time and sequentiality

In this example, the problem is caused by insufficient correspondence between the time oriented and sequence oriented views of the system. Infinitely many events occur in finite time, preventing a full mapping of the time oriented view onto the sequence oriented view. Just as we can reach any element of ℕ if we count long enough, although ℕ itself is infinite, so we must be able to reach any event in the infinite event sequence by finite sequential execution. This is not the case in Zeno’s example as it takes infinite events to reach the case where the transition (s_0, s_1) is executed.

Finite event rate requirement:

It is necessary that a one-to-one mapping between time space and sequence space is possible along the full relevant length of time and sequence space. For this to be possible, the number of events occurring in a finite section of time must itself be finite.

In all real systems, this property is already fulfilled. Any implementation not fulfilling it is not sustainable because such a system would suffer a hardware failure of the type described in the first paragraph of section A.2.4.

A.2.6. Conclusion

Zeno’s prediction of the outcome of the race defies kinetics because the proof is performed in event space rather than time space. The point in time where Achilles should overtake the tortoise is never reached because it requires infinite events to reach that point. The fault in the reasoning lies in Zeno choosing a model where infinite events occur in finite time.

A.2.7. Application

Where is the applicability of this paradox, seeing thst in the models likely to be modelled by CIP-Tool, such scenarios can practically be excluded? They can be excluded when the events triggering the transitions are real physical events. But what about dummy events introduced for modelling purposes such as that of figure A.1. (page 138).

The dummy transition was introduced to falsify the claim that the alarm will certainly be pressed. Because of the dummy transition, there is a fair full path which never reaches s_1. This path models the case that the alarm is not used at any point during the life of the system, which is physically possible and even likely. The dummy transition has no physical trigger, and so it will not be traversed during the real operation of the system. For the model-checker, however, it can be traversed infinitely. Because of the liveness requirement, the model assumes that either the alarm must be pressed at some point, or the dummy transition must be executed. In the latter case, the system returns to its initial state and faces the same choice again. As the model is not bound by time constraints, we can model infinite traversals of the dummy transition. We can even model infinite transitions and map this onto finite time. All events occurring beyond this finite time are no longer relevant and so this construct can be used to circumvent the liveness requirement. The finite event rate requirement is necessary to prevent this.

A.3. Fairness

A.3.1. About fairness in general

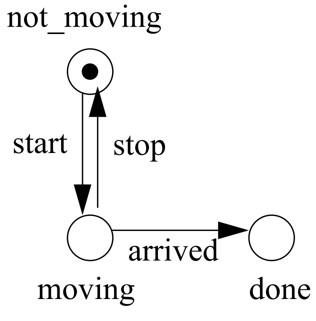
In this section the need for fairness in model checking will be introduced. Fairness will be defined and its implementation shown. Fairness as applied here is not completely identical to the general definition of fairness which has been discussed in sections 4.4. and 4.5.

A.3.2. The problem of event starvation

In section A.1. we saw that every state s in a continuing path must be followed by a further state unless s is a dead-end. In this section it will be shown that not all paths are acceptable and that further criteria are necessary for determining paths representing real behaviours of the system.

In the state diagram of figure A.5. a system is shown which moves an object towards its required position. The object is initially not moving and not in the required position. Movement is always towards the required position and this position is immobile. The movement can be started and stopped by the events *start* and *stop*. When the required position is attained, this is communicated by the event *arrived*. The movement stops and cannot be restarted.

figure A.5. An example system in which event starvation can occur



We wish to verify the property *eventually*(*done*). Considering the system being modelled, we know that this is fulfilled because the required position is immobile (in contrast to Zeno’s tortoise) and movement is always towards it.

Studying the state diagram, this is not apparent. The diagram allows the path [*not_moving*, *moving*, *not_moving*, *moving*, ...]. This path can continue infinitely without ever visiting *done*. We say that the transition (*moving*, *done*) is being starved. Although the path enters the pre-state of the transition infinitely often, the transition is never executed.

A.3.3. Preventing starvation

A first approach to preventing starvation would be to disallow paths which infinitely often visit the pre-state of a transition without executing the transition. This is insufficient as some transitions need never be executed as discussed in section A.1.3. (we would be able to infer that every system with an emergency alarm button will eventually stop because the button will eventually be used).

To solve this problem we extend the model system by the concept of *fairness*.

A.3.4. Fairness constraints, fair and unfair transitions

A fairness constraint is an attribute which can be associated with one or several transitions.

This is defined in sections 3.7.2. to 3.7.7.

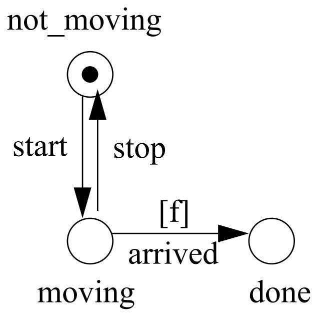
A.3.5. System behaviour

The *behaviour* of a system is the set of all its full fair paths.

A.3.6. Some examples

Returning to the example of section A.3.2. we declare that the transition (*moving*, *done*) has the fairness constraint f. The loop infinitely executing (*not_moving*, *moving*) is unfair because the transition (*moving*, *done*) is enabled in state *moving*, but no transition with the fairness constraint f transition is executed in the loop. Therefore no fair path may infinitely repeat that loop. The condition *eventually*(*done*) is fulfilled.

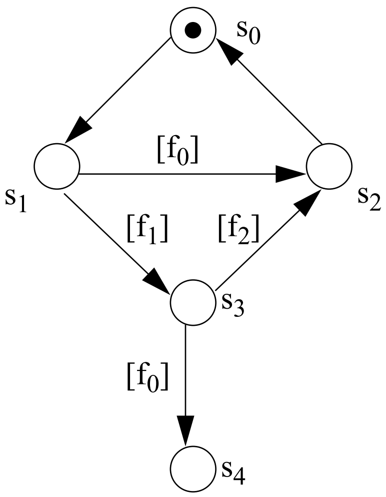
figure A.6. Example of section A.3.2. modified by adding a fairness constraint



fairness attributes of transitions are shown in square brackets.

Is the loop [s_0, s_1, s_3, s_2, s_0] of the system of figure A.7. fair? No, it is unfair. It visits the states s_1 and s_3 where transitions with the fairness constraint f_0 are enabled but no such transition is executed on the loop.

figure A.7. Example system (from figure 4.11. on page 62)



Is the loop [s_0, s_1, s_2, s_0] fair? No, it visits the state s_1 where a transition with the fairness constraint f_1 is enabled but no such transition is executed on the loop.

Is the loop [s_0, s_1, s_3, s_2, s_0, s_1, s_2, s_0] fair? Yes, because the only transition to be enabled but not executed is (s_3, s_4). This transition has a fairness attribute but this attribute is also fulfilled by the transition (s_1, s_3) which is part of the loop. This example shows how unfair loops can sometimes be combined to make fair loops.

Is the path [s_0, s_1, s_3, s_4] fair? It visits the state s_3 where a transition with the fairness constraint f_2 is enabled and no transition with that constraint is executed on the path. But the path length is not infinite so this is of no concern. All finite paths are fair.

A.3.7. Another look at Zeno's paradox

Does the introduction of fairness constraints make any difference to the paradox of section A.2.? We know that Achilles runs faster than the tortoise. Therefore it is fair that he will catch up with the tortoise. A fairness constraint must be attached to the transition (s_0, s_1) of figure A.3. Because of this fairness constraint, Achilles must overtake the tortoise in a finite number of steps.

This conclusion is almost as confusing as the original paradox. We know that Zeno is right when he says that Achilles cannot overtake the tortoise in a finite number of steps, yet we have created a case where this is not so. The reason lies once again in the finite event rate requirement of section A.2.5. If we could somehow limit the events to a finite number of events in finite time, Achilles would indeed overtake the tortoise and the model would once again be correct.

A.4. Reachability and fair paths

A.4.1. Proof

In section 3.7.6. (page 49) it was claimed that:
All states reacheable by an unfair full path are also reacheable by a fair full path.

Proof:
A state s is visited by an unfair full path u. We wish to demonstrate that there also exists a fair full path visiting s.
Because the state space is finite, there exists a finite path v' so that v' starts in the initial state and ends in s.

$\forall s \in \mathbf{SR} : \exists v' \in \mathbf{P} : (\mathit{first}(v') = s_{init}) \wedge (\mathit{last}(v') = s)$

For s to be unreacheable by fair full paths, all full paths having v' as prefix must be unfair. The following demonstration shows that this is not possible.

Let V be the set of full paths (both fair and unfair) with v' as prefix or subpath.

$V \subseteq \mathbf{P} \mid \forall v \in V : v' \in \mathit{subpath}(v)$

If s is unreacheable by fair full paths, then V contains only unfair paths.
Every path in V infinitely visits at least one state z, in which a transition r with farness constraint f is enabled, so that transitions of f are not traversed infinitely by the path.

$(V \cap FP = \emptyset) \Rightarrow \forall v \in V : \exists z \in \mathbf{SR} : (\mathit{occurrences}(z, v) = \infty \Rightarrow$
$\quad (\forall r \in \mathit{enabled}(z): \forall f \in \mathit{fairness}(r): \forall t \in f: \mathit{occurrences}(t, v) \in \mathbb{N}))$

If no transition of f is traversed infinitely by any path of V, then:
a) no transition of f is traversed by any path of V after first reaching s.
b) some transitions of f are traversed by some paths of V after first reaching s, but no path does so infinitely.

Case (a) cannot be because if a path of V enables r, then there must also be a path of V traversing r.

Case (b) implies that the fair transitions are so placed that they cannot be infinitely traversed, otherwise at least one path of V would traverse them infinitely. Therefore, for any transition t ∈ f so that *pre*(t) is reacheable from s, *pre*(t) is not reacheable from *post*(t). The state-space containing the paths of V after this transition must thus exclude *pre*(t).

$(V \cap FP = \emptyset) \Rightarrow (\forall t \in f : \mathit{reacheable}(\mathit{pre}(t), s) \Rightarrow \neg \mathit{reacheable}(\mathit{pre}(t), \mathit{post}(t)))$

All paths of V traversing r must do so exactly once. As these paths are all unfair, they must infinitely enable other transitions of f. The same argument applies for these transitions.

Therefore V must contain at least one path which infinitely traverses transitions of f, but after reaching s traverses each transition of f at most once.

This is impossible because the state space is finite.

Therefore it is impossible for V to contain only unfair paths.

Appendix B: Notes on similar work

B.1. Fairness constraints

In section 4.5.4. CTL fairness constraints were introduced and their applicability to the fairness constraints used in this project (see section 3.7.3.) was discussed (hereafter called *project fairness constraints* to avoid confusion). In the following it is demonstrated that project fairness constraints are a special case of CTL* fairness constraints.

B.1.1. A special case of a CTL* fairness constraint

The formula for a project fairness constraint is:

fairTrans(p, n, f) $\vee$ ***G***$\neg$*preFair*(p[n], f)

where:

p is the path on which these formulas are tested.
n is an integer representing the position on the path at which these formulas are tested.
f is a fairness constraint according to the definition of this project.
the formula *fairTrans*(p, n, f) is true only if constraint f applies to the nth transition of path p:

fairTrans: $\mathbf{P} \times \mathbb{N} \times \mathbf{F} \rightarrow$ {true, false}
fairTrans(p, n, f) := ($\exists$t $\in$ f : ((t $\in$ *enabled*(p[n])) $\wedge$ (p[n+1] $\in$ *post*(t))))

the formula *preFair*(s, f) is true only if a transition with constraint f is enabled in state s :

preFair: $\mathbf{S} \times \mathbf{F} \rightarrow$ {true, false}
preFair(s, f) := (*enabled*(s) $\cap$ f $\neq \emptyset$)

B.1.2. Proof

If *fairTrans*(p, n, f) $\vee$ ***G***$\neg$*preFair*(p[n], f) is true for infinite n along p, then
- either infinitely many transitions along the path have the fairness constraint f.
- or transitions with fairness constraint f are not infinitely activated (hence a point is reached on the path that no further activations occur).

Proof that this expression is equivalent to project fairness:

fairTrans(p, n, f) $\vee$ ***G***$\neg$*preFair*(p[n], f)
is true infinitely often along all fair full paths

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP}$:
$\quad \mathit{propCount}_{n}(\mathit{fairTrans}(p, n, f)$
$\quad \vee (\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f)), p) = \infty$

$\Leftrightarrow$

(splitting the *propCount* operator)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : ($
$\quad (\mathit{propCount}_{n}(\mathit{fairTrans}(p, n, f), p) = \infty)$
$\quad \vee$
$\quad (\mathit{propCount}_{n}(\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f), p) = \infty))$

$\Leftrightarrow$

(splitting the $\forall$ operator)

$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_{n}(\mathit{fairTrans}(p, n, f), p) = \infty)$
$\quad \vee$
$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_{n}(\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f), p) = \infty)$

These two statements are considered individually:

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_{n}(\mathit{fairTrans}(p, n, f), p) = \infty$

$\Leftrightarrow$

(replacing *fairTrans* by its definition)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_{n}($
$\quad (\exists t \in f : ((t \in \mathit{enabled}(p[n])) \wedge (p[n+1] \in \mathit{post}(t)))), p) = \infty$

$\Leftrightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_{n}($
$\quad \exists t \in f : ((p[n] \in \mathit{pre}(t)) \wedge (p[n+1] \in \mathit{post}(t))), p) = \infty$

$\Leftrightarrow$

(because f is a finite set at least one of its elements must occur infinitely along p)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f\text{: } \mathit{occurrences}($
$\quad t : ((p[n] \in \mathit{pre}(t)) \wedge (p[n+1] \in \mathit{post}(t))), p) = \infty$

$\Leftrightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(t, p) = \infty$

It also follows:

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(t, p) = \infty$

$\Rightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty$

Therefore:

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_n(\mathit{fairTrans}(p, n, f), p) = \infty$

$\Leftrightarrow$

$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(t, p) = \infty)$
$\wedge$
$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty)$

Now turning to the second statement:

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_n(\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f), p) = \infty$

$\Leftrightarrow$

(replacing *preFair* by its definition)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_n(\forall m \in \mathbb{N} : \neg(\mathit{enabled}(p[n+m]) \cap f \neq \emptyset), p) = \infty$

$\Leftrightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_n(\forall m \in \mathbb{N} : \mathit{enabled}(p[n+m]) \cap f = \emptyset, p) = \infty$

$\Leftrightarrow$

(infinitely many continuing sub-paths do not execute any fair transitions. We study only one of these but without losing generality because this continuing subpath itself has infinite continuing subpaths which do not execute any fair transitions)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists n \in \mathbb{N} : \forall m \in \mathbb{N} : \mathit{enabled}(p[n+m]) \cap f = \emptyset$

$\Leftrightarrow$

(the formula $\mathit{enabled}(p[k]) \cap f \neq \emptyset$ can be fulfilled at the most n times)

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_k(\mathit{enabled}(p[k]) \cap f \neq \emptyset, p) \leq n$

$\Leftrightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \forall t \in f : \mathit{occurrences}(\mathit{pre}(t), p) \leq n$

$\Rightarrow$

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \neg\exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty$

Therefore:

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \mathit{propCount}_n(\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f), p) = \infty$

$\Leftrightarrow$

$$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \forall t \in f : \mathit{occurrences}(\mathit{pre}(t), p) < \infty)$$
$$\wedge$$
$$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \neg\exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty)$$

Putting the two parts back together:

$$\begin{aligned}&\forall(f, p) \in \mathbf{F} \times \mathbf{FP} :\\ &\quad \mathit{propCount}_n(\mathit{fairTrans}(p, n, f)\\ &\quad \vee (\forall m \in \mathbb{N} : \neg\mathit{preFair}(p[n+m], f)), p) = \infty\end{aligned}$$

$\Leftrightarrow$

$$((\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(t, p) = \infty)$$
$$\wedge$$
$$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty))$$
$$\vee$$
$$((\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \forall t \in f : \mathit{occurrences}(\mathit{pre}(t), p) \leq n)$$
$$\wedge$$
$$(\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : \neg\exists t \in f : \mathit{occurrences}(\mathit{pre}(t), p) = \infty))$$

This statement is universally true. It follows that:

$$\begin{aligned}&\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : (\exists r \in f : \mathit{occurrences}(\mathit{pre}(r), p) = \infty \Rightarrow\\ &\quad \exists t \in f : \mathit{occurrences}(t, p) = \infty)\end{aligned}$$

$\Leftrightarrow$

(substituting s for *pre*(r))

$\forall(f, p) \in \mathbf{F} \times \mathbf{FP} : (\forall s \in \mathbf{S} : \mathit{occurrences}(s, p) = \infty \Rightarrow$
$\quad (f \cap \mathit{enabled}(s) \neq \emptyset \Rightarrow$
$\quad \exists t \in f : \mathit{occurrences}(t, p) = \infty))$

$\Leftrightarrow$

(moving the declaration of f)

$\forall p \in \mathbf{FP} : (\forall s \in \mathbf{S} : \mathit{occurrences}(s, p) = \infty \Rightarrow$
$\quad \forall f \in \mathbf{F} : (f \cap \mathit{enabled}(s) \neq \emptyset \Rightarrow$
$\quad \exists t \in f : \mathit{occurrences}(t, p) = \infty))$

$\Leftrightarrow$

(splitting f $\cap$ *enabled*(s))

$\forall p \in \mathbf{FP} : (\forall s \in \mathbf{S} : \mathit{occurrences}(s, p) = \infty \Rightarrow$
$\quad \forall f \in \mathbf{F} : \forall r \in \mathit{enabled}\mathbf{(s)} : (r \in f \Rightarrow$
$\quad \exists t \in f : \mathit{occurrences}(t, p) = \infty))$

$\Leftrightarrow$

($\forall f \in \mathbf{F} : \forall r \in \mathit{enabled}\mathbf{(s)} : r \in f \Rightarrow$ is equivalent to $\forall r \in \mathit{enabled}\mathbf{(s)} : \forall f \in \mathit{fairness}(r) :$)

$\forall p \in \mathbf{FP} : (\forall s \in \mathbf{S} : \mathit{occurrences}(s, p) = \infty \Rightarrow$
$\quad \forall r \in \mathit{enabled}\mathbf{(s)} : \forall f \in \mathit{fairness}(r) : \exists t \in f : \mathit{occurrences}(t, p) = \infty)$

$\Leftrightarrow$

(the statement can only hold for states s which are visited by the path)

$\forall p \in \mathbf{FP} : (\forall s \in \mathit{visited}(p) : \mathit{occurrences}(s, p) = \infty \Rightarrow$
$\quad \forall r \in \mathit{enabled}\mathbf{(s)} : \forall f \in \mathit{fairness}(r) : \exists t \in f : \mathit{occurrences}(t, p) = \infty)$

Which is the project definition of fair paths (compare section 3.7.3.)

Appendix C: Cluster reduction

C.1. Effects of cluster reduction on behaviour

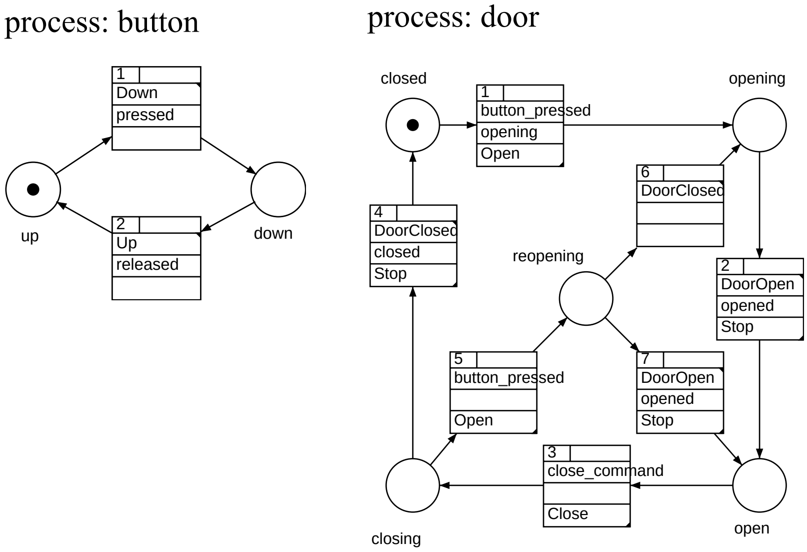
In section C.1.1. an example is discussed which does not reduce the state-space and introduces additional behaviour. In sections C.1.2. and C.1.3. examples are discussed which lead to existing behaviour being omitted. The reader who is not interested in studying these examples can jump to the discussion in section C.1.4.

C.1.1. Example of cluster reduction leading to new behaviour.

Starting off with the same system as used in the previous example, we wish to investigate a condition dependent on processes *button* and *door*. In this example the results of using the full cluster and of using a cluster reduced by removing processes *lamp* and *controller* are compared.

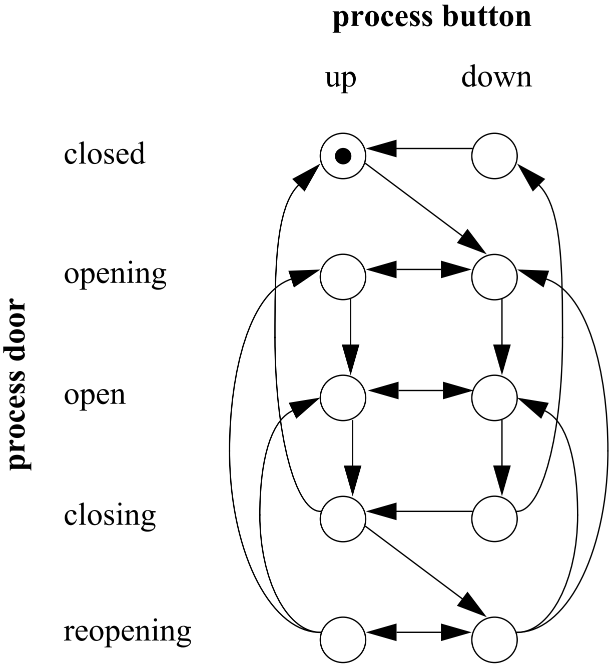
In figure C.1. the process state graphs of the reduced cluster are shown. In figure C.2. the corresponding cluster state graph is shown.

figure C.1. processes of reduced cluster (as figure 5.1. but processes lamp *and* controller *omitted)*


process: button

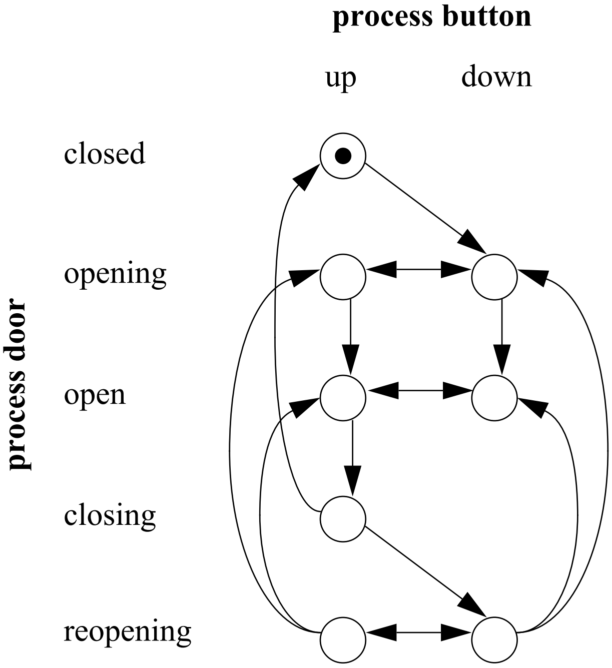
process: door

figure C.2. Cluster-state graph of cluster of figure C.1.



As comparison, the cluster state graph of the full cluster is shown in figure C.3.

figure C.3. State-graph for processes door *and* button *projected from full cluster state-graph*



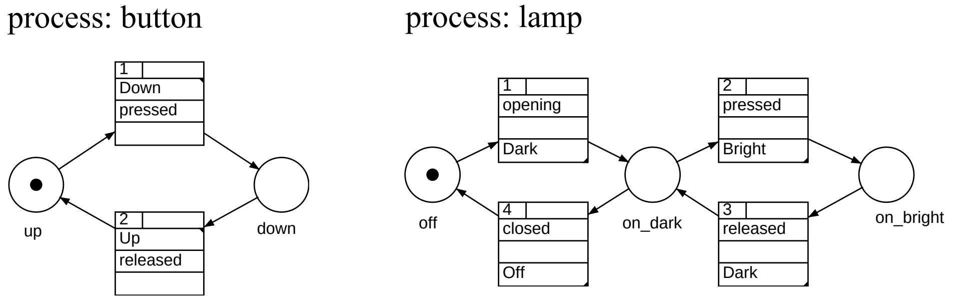
Comparison of figures C.2. and C.3. shows that two additional states and four additional transitions have been introduced by the omission of processes *lamp* and *controller*. This is because process *door* is able to receive a *timeout* in the reduced state graph whereas this is not possible in the full state graph. Whether or not these additional states and transitions are a problem depends on the condition being checked. The effects of these additional states and transitions will be discussed later (see section C.1.4).

This example shows that cluster reduction does not always reduce the complexity of the system. In this example, the number of cluster states and number of transitions has actually increased over that of the full system. Admittedly, this example was constructed for the purpose of demonstrating this, and the effect of the actual increase in the number of states and transitions can be considered to be rather exceptional. However, even if the total number of transitions decreases, the appearance of additional ones does create false behaviours.

C.1.2. Example of behaviour loss through cluster reduction.

Again, starting off with the same system as used in the previous example, we wish to investigate a condition dependent on processes *button* and *lamp*.

figure C.4. processes of reduced cluster (as figure 5.1. but processes door *and* controller *omitted)*



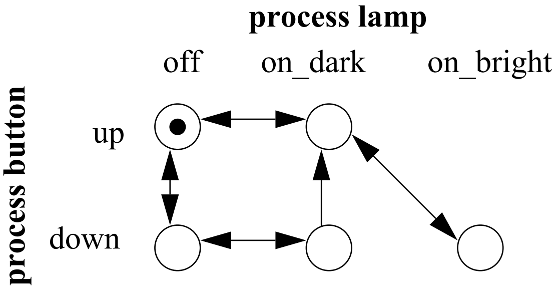
The transitions of this reduced cluster are listed in the following table:

table C.1. Cluster-transitions of cluster of figure C.4.

pre-state		post-state		trigger
button	lamp	button	lamp	
up[1]	off[1]	down	off	Down
up[1]	off[1]	up	on_dark	opening
down	off	up	off	Up
down	off	down	on_dark	opening
up	on_dark	down	on_bright	Down
up	on_dark	up	off	closed
down	on_dark	up	on_dark	Up
down	on_dark	down	off	closed
down	on_bright	up	on_dark	Up

[1]: initial cluster state

figure C.5. Cluster-state graph of reduced cluster of figure C.4.



Does the reduced cluster-state graph still accurately represent the behaviour of these two processes?

figure C.6. Projection of state diagram of processes button *and* lamp *out of full cluster state diagram.t*

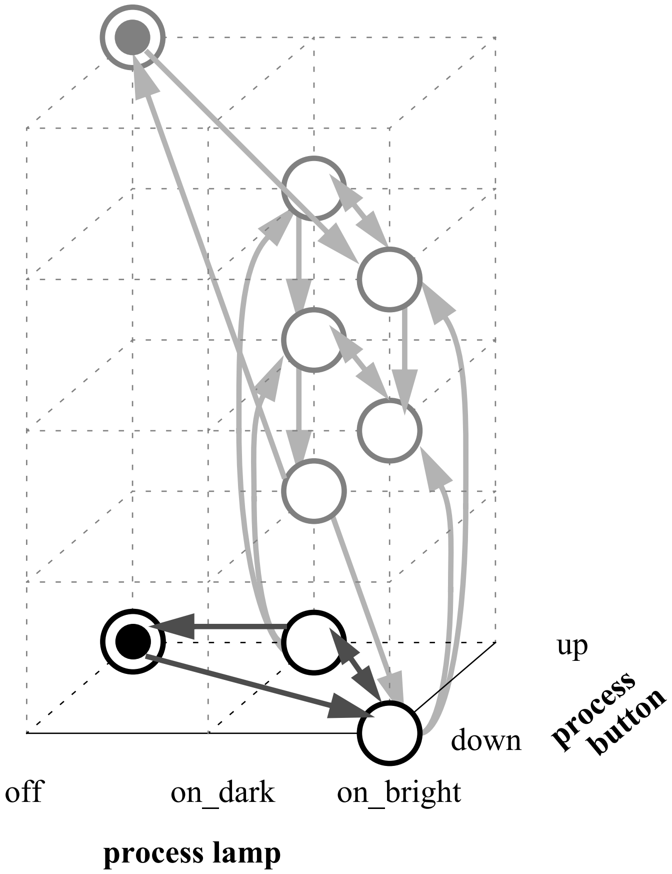
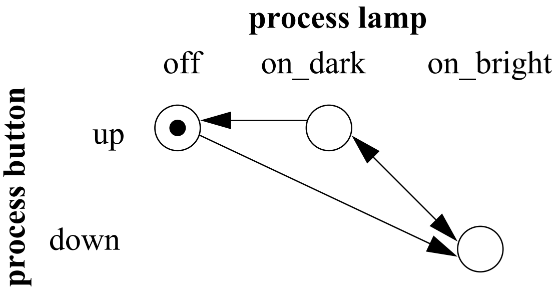
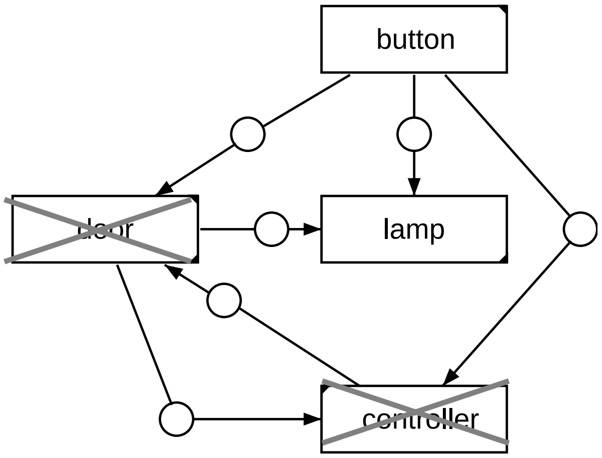


figure C.7. Projected state graph showing processes button *and* lamp*, projected from full cluster (compare figure C.5.).*



Comparing figure C.5. with figure C.7., we see some fundamental differences. The transition ([*up*, *off*], [*down*, *on_bright*]) appears in the projected state-graph but not in the reduced cluster-state graph. This is because in the full state graph, process *lamp* is activated twice in a single transition. In the same cluster transition, a pulse is first propagated from process *button* to process *lamp* via process *door*, and then a second pulse is propagated directly.

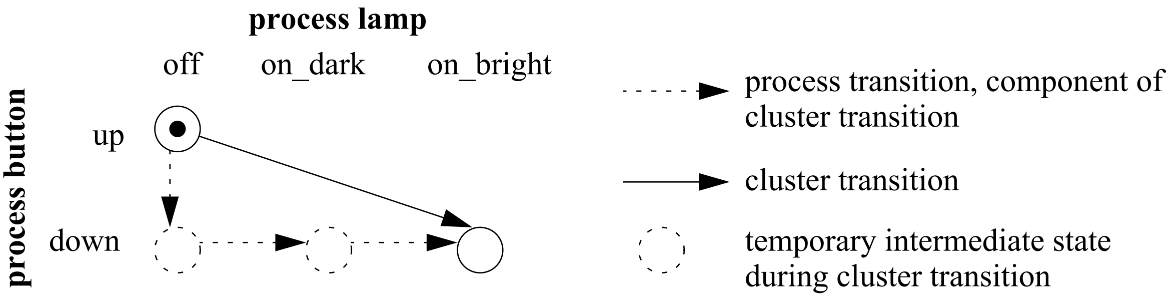
figure C.8. pulse cast net from figure 2.11. with processes door and controller removed



With process *door* removed, this double activation no longer occurs. The transition ([*up*, *off*], [*down*, *on_bright*]) can be replaced by other sequence transitions reaching the same state. This is not the same thing, however, as it is not the same behaviour which is being modelled. In figure C.9., the three process transitions which compose the cluster transition ([*up*, *off*], [*down*, *on_bright*]) are shown. The process transitions are shown as dotted lines and the cluster transition as a full line.

First, the process *button* is activated by the message *Down*. The process transition (*up*, *down*) is executed. The outPulse *pressed* is propagated via process *door* to process *lamp* where it is received as inPulse *opening*. This triggers the process transition (*off*, *on_dark*). Next, the outPulse *pressed* from process *button* is received directly by process *lamp* where it triggers the process transition (*on_dark*, *on_bright*).

*figure C.9. The cluster transition ([*up, off*], [*down, on_bright*]) of the projected full state cluster decomposed into process transitions*


process lamp
off
on_dark
on_bright
process button
up
down
process transition, component of cluster transition
cluster transition
temporary intermediate state during cluster transition

In the reduced cluster this behaviour is not possible. The process transition (*up*, *down*) can be triggered just as described above. The outPulse *pressed* is cast to the process *door* where it is lost because this process is missing in the reduced cluster. The same outPulse is then cast directly to process *lamp* but cannot trigger any transition because process *lamp* is not in a suitable pre-state. As a next step, the message *opened* (replacing the in-Pulse *opened* from process *door*) can be sent to process *lamp*. There it triggers the transition (*off*, *on_dark*). So far the behaviour of the projected full cluster has been correctly emulated. However, the third transition (*on_dark*, *on_bright*) cannot be triggered because it must be triggered by the inPulse *pressed* from process *button*. This process is unable to send the corresponding outPulse because it is in a state where the required transition is not active.

C.1.3. Further example of behaviour loss through cluster reduction

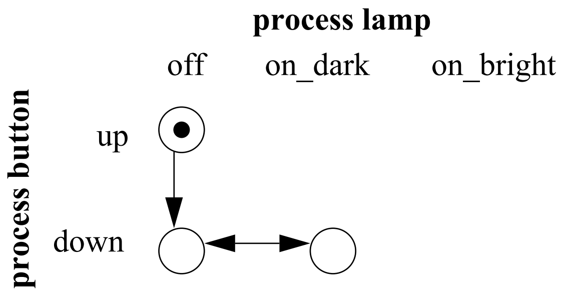
To accentuate the problem, let us make a small and seemingly insignificant modification to the cluster. If the process transition (*off*, *on_dark*) in process *lamp* were dependent on a gate (see section 2.6.2.) requiring that process *button* be in state *down*, this would have no effect on the behaviour of the full cluster because in the full cluster this is already fulfilled for the one instance where this transition occurs (transition 1 in figure 5.2., page 78). Likewise, making the process transition (*down*, *up*) in process *button* dependent on a gate requiring process *lamp* to be in state *on_bright* does not affect the behaviour of the full cluster, because it is also fulfilled in all instances where it occurs (transitions 2, 6 and 13 in the same figure). In the reduced cluster, however, the effects of these restrictions would be catastrophic.

table C.2. Table of table C.1. with the transitions which are disabled by the described changes crossed out

pre-state		post-state		trigger
button	lamp	button	lamp	
up[1]	off[1]	down	off	Down
~~up~~[1]	~~off~~[1]	~~up~~	~~on_dark~~	~~opening~~
~~down~~	~~off~~	~~up~~	~~off~~	~~Up~~
down	off	down	on_dark	opening
up	on_dark	down	on_bright	Down
up	on_dark	up	off	closed
~~down~~	~~on_dark~~	~~up~~	~~on_dark~~	~~Up~~
down	on_dark	down	off	closed
down	on_bright	up	on_dark	Up

1: initial cluster state

figure C.10. Cluster-state graph of reduced and modified cluster



Note that although six transitions remain in the transition table, only three are in the cluster state graph. The other three transitions cannot be executed because they cannot be enabled (as their pre-states are no longer reacheable).

Many of the states reacheable in the projected full cluster (figure C.7.) are not reacheable any more. Thus the apparently attractive method of reducing the cluster complexity by removing processes must be modified to circumvent this problem.

C.1.4. General discussion on the effects of cluster reduction

Cluster reduction does not always perfectly preserve behaviour. Additional cluster transitions can be created and cluster transitions can be lost. It would seem at first that the method is therefore not very useful.

Cluster reduction is best suited to removing processes having no effect on the condition being verified. Limiting the method to such cases, however, is often impractical as the results of the interconnectedness of processes is not always clear. A more general approach to handling such non-idealities is thus required.

The cases of cluster transitions being lost are due to the loss of sequentiality when a process is activated several times during a cluster transition. This phenomenon can be reliably prevented using methods which will be presented in section C.2.

In other cases, cluster transitions are conserved and additional transitions appear. What effects do the additional transitions have on model checking?

When testing for *eventually*, these transitions provide additional safety. All paths of the full cluster survive in the reduced cluster and additional ones also appear. If the *eventually* property is fulfilled for all paths of the reduced cluster, then it is also fulfilled for all paths of the full cluster. If, however, the condition is failed for the reduced cluster, there is still a possibility it might be fulfilled for the full cluster (especially if the counter example would not be possible in the full cluster), and the check could be started again with a different set of processes.

When testing for *invariance*, the same safety applies as for *eventually*.

When testing for *reacheability*, there is no safety, however, as states may be reacheable in the reduced cluster which are not reacheable in the full cluster. In the case of positive *reacheability* (the user wants the condition to be fulfilled), cluster reduction is of no use. However, positive reacheability is normally of little interest to the developer as it can be demonstrated by a simple example. Negative *reacheability* is normally of more interest (the developer wishes to make sure certain states are not reached). Here too, cluster reduction is a useful device.

C.2. Preventing loss of transitions in cluster reduction

C.2.1. The problem of loss of behaviour

Cluster reduction can lead to loss of behaviour (transitions are no longer possible and states are no longer reacheable). Loss of behaviour is possible when at some point in an interaction tree of the full cluster, a transition can be executed which is dependent on another transition which could have been executed at an earlier point during the interaction tree, but it is not known whether this transition was actually executed. Examples of this were discussed in sections C.1.2. and C.1.3.

To prevent such erroneous propagation patterns, a first approach is to design the model checker to recognise pulse propagation structures which can lead to such behaviour and give them an appropriate treatment.

C.2.2. Detecting potential loss of behaviour

When setting up the model of the (reduced) cluster for model checking, the checker identifies situations which can potentially lead to loss of transitions. It does this by traversing every potential interaction tree and analysing it for potential loss of behaviour. Behaviour can be lost when a transition is to be performed which is dependent on another transition in a part of the tree which was not executed.

algorithm C.3. Detecting potential loss of behaviour

```
A depth first traversal of the interaction tree (see section
2.5.4.) is performed (traversal in the order of activation during
execution).

For every process P of the reduced cluster reached during tra-
versal and the associated inPulse in, the following operations
are performed:

- if a process Q not in the reduced cluster occurs above P in
the tree hierarchy, then all transitions of P which can be exe-
cuted at this point in the traversal are associated with Q.

- for all transitions r of P which are triggered by in; if r is
dependent on a transition associated with a process which does
not occur above P in the tree hierarchy, then a potential loss
of behavior is detected.
```

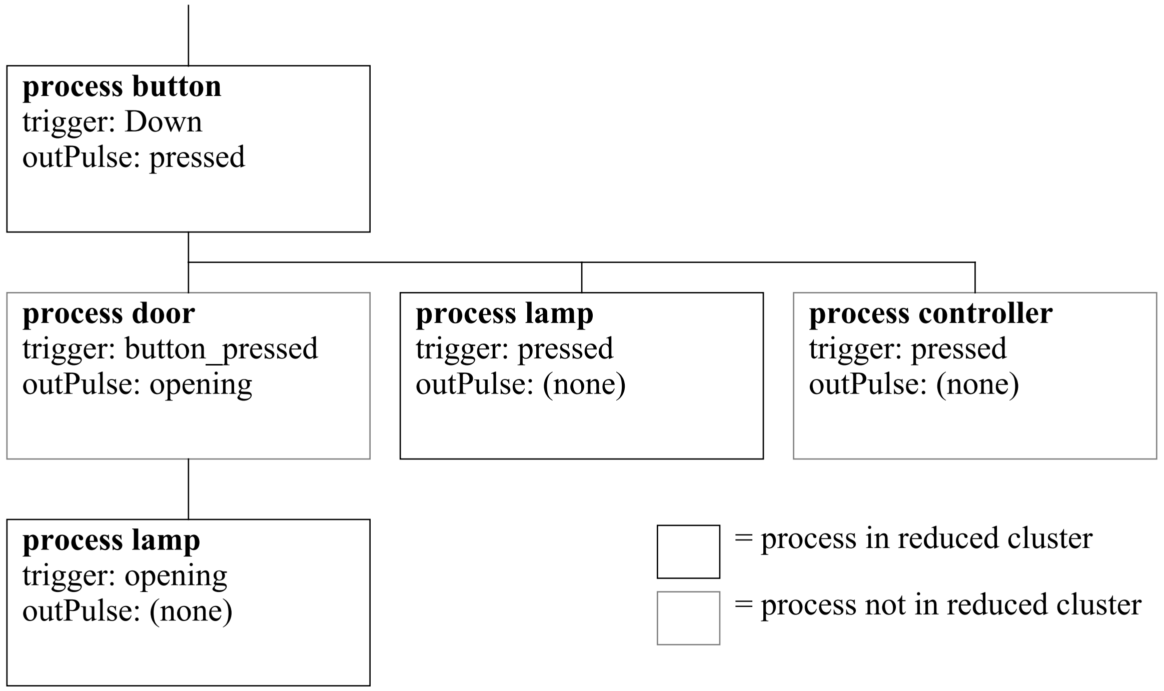
Explanation:

When calculating a traversal of the reduced cluster, the pre-state of the cluster reduction is known. Therefore the process pre-states of all processes of the reduced cluster are known. It is thus known whether an inPulse triggers a process transition or not and whether an outPulse is sent. When a process is missing, however, its pre-state is not known and so it is not known whether or not an outPulse was sent. For all processes occurring below this missing process in the interaction tree, it is not known whether a transition took place or not. These are associated with the missing process on which this transition is dependent. Any further transition of a process of the cluster that is dependent on these transitions can thus be identified.

C.3.3. Example of detecting loss of behaviour

This example shows how the method presented above detects the loss of behaviour that was illustrated in section C.1.2. The interaction tree for this example is shown in figure C.11.

figure C.11. Interaction tree (from figure 2.16.)



In this example, the reduced cluster has only processes *button* and *lamp*.

Process *button* is activated by message *Down* and can send outPulse *pressed* to process *door*. No actions are needed on process *button*, because there are no processes above it in the tree.

The outPulse is translated to inPulse *button_pressed* and sent to process *door*. No action is required on process door as it is not in the reduced cluster. The translations that can be triggered by the inPulse *button_pressed* are (*closed*, *opening*) and (*closing*, *reopening*). The former sends the outPulse *opening*, the latter sends no outPulse.

This outPulse *opening* is translated to inPulse *opening* and sent to process *lamp*. InPulse *opening* can trigger the transition (*off*, *on_dark*). The process *door* is above *lamp* in the tree hierarchy and *door* is not in the reduced cluster. So the transition (*off*, *on_dark*) is associated with the process *door*.

```
(off, on_dark) -> {door}
```

The next process to be considered in the interaction tree traversal is *lamp* again. It receives inPulse *pressed*. The inPulse *pressed* triggers the transition (*on_dark*, *on_bright*). This transition is dependent on (*off*, *on_dark*) which is associated with process *door*. Process *door* does not occur above the present instance of process *lamp* in the tree hierarchy. A potential loss of behaviour is thus detected.

C.3.4. Handling loss of behaviour by tree splitting

Loss of behaviour can occur when the execution of transitions in the correct order is not possible due to transitions being disabled when the correct order of execution cannot be observed.

We can prevent this by splitting up interaction trees. The tree is divided above every process where potential loss of behaviour is detected and the severed section is made into a separate interaction tree.

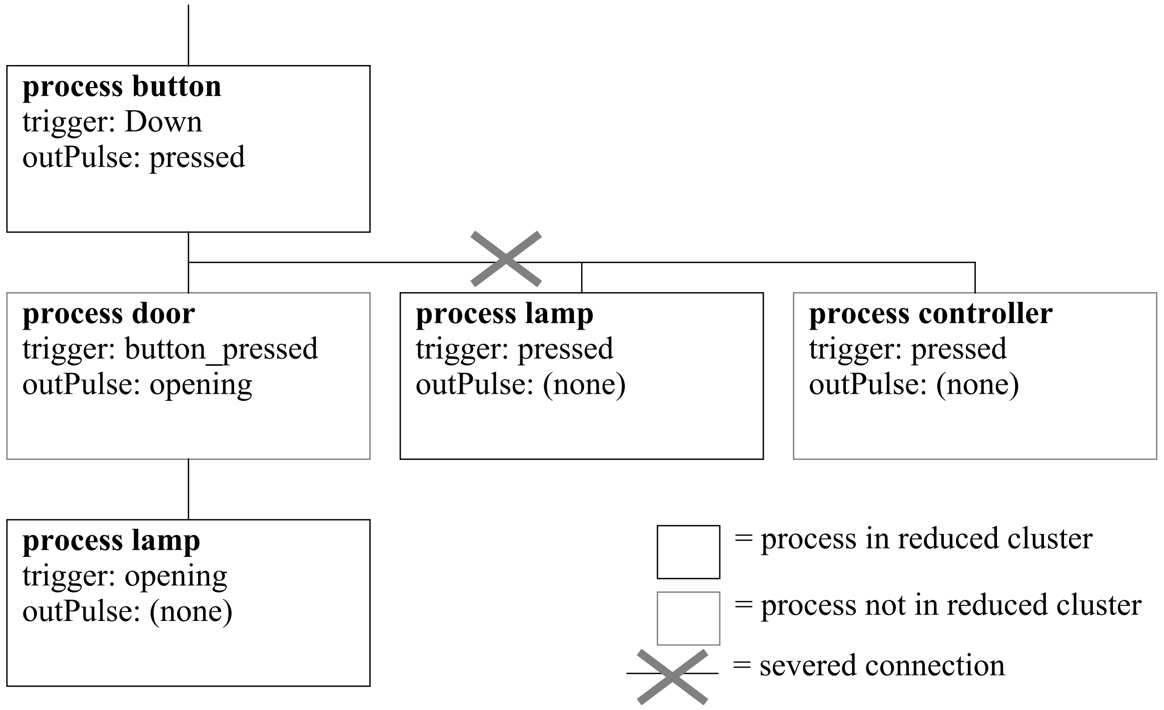
We split the tree above the process occurrence where the potential loss of behaviour is first detected. If the cast being split casts to further processes after the process occurrence where potential loss of behaviour was first detected, these further process occurrences are made part of the tree being split off. Correct behaviour could also be observed if these remained with the main tree, but splitting this way helps conserve sequentiality.

C.3.5. Example of handling loss of behaviour

We return to our now amply discussed example last visited in section C.3.3.

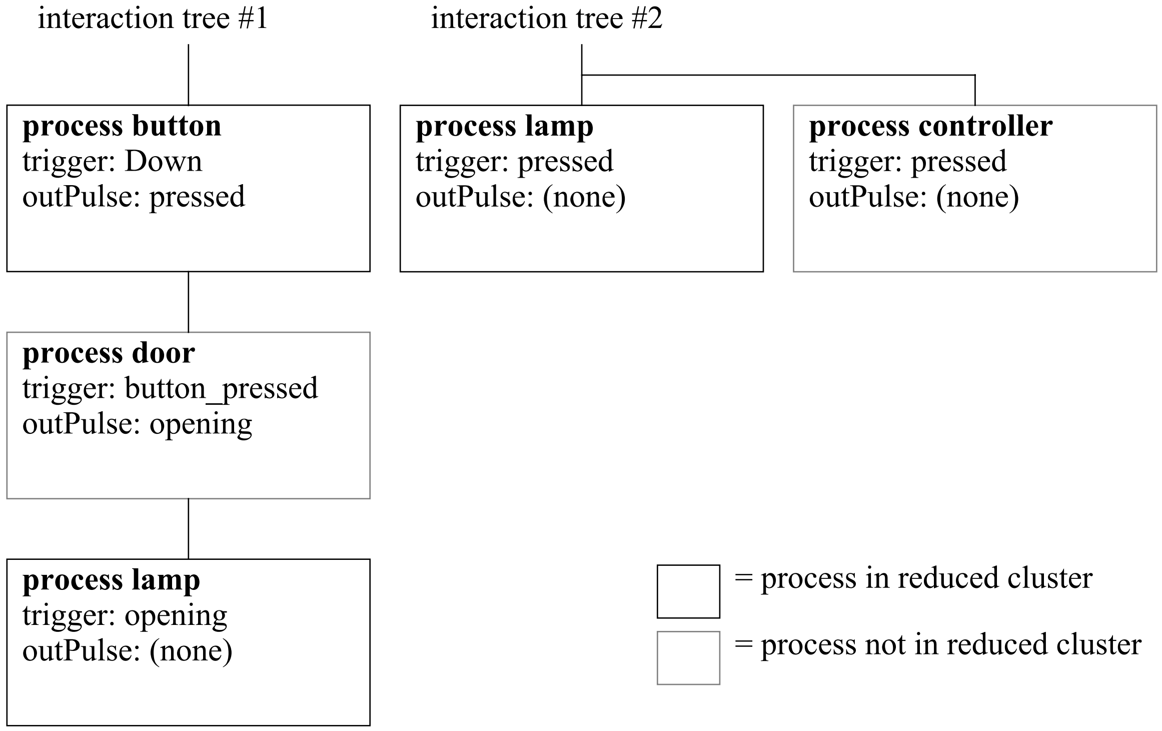
The potential loss of behaviour is detected in the second occurrence of process *lamp*. The tree is separated above this process occurrence as shown in figure C.12.

figure C.12. The interaction tree is severed above the second occurrence of process lamp



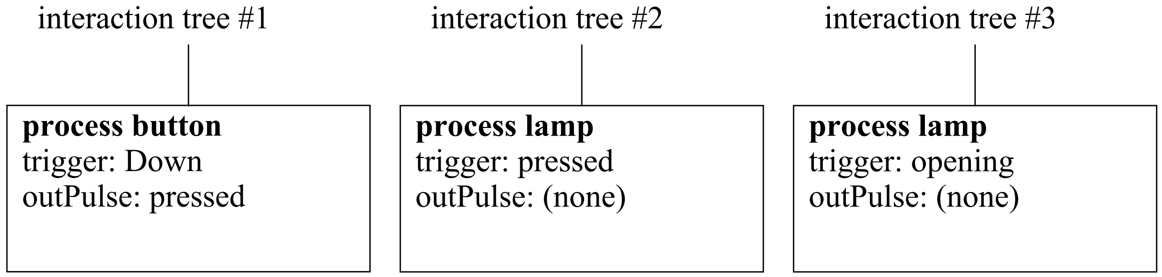
The two resulting interaction trees are shown in figure C.13.

figure C.13. The interaction tree is divided into two separate trees



Omitting the processes not included in the reduced cluster, we end up with three interaction trees formed from the original tree.

figure C.14. Processes not in the reduced cluster are removed from the interaction tree



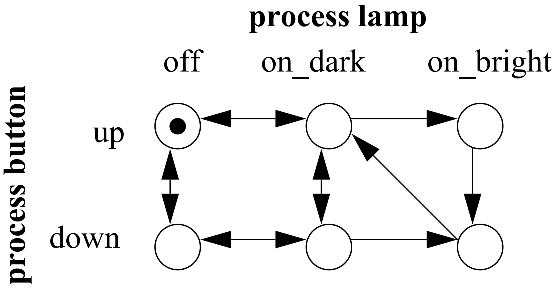
The effects of the three separated trees on the reduced cluster behaviour is the following:

table C.3. Table of transitions of this example

pre-state		post-state		trigger
button	lamp	button	lamp	
up[1]	off[1]	down	off	Down
up[1]	off[1]	up	on_dark	opening
down	off	up	off	Up
down	off	down	on_dark	opening
up	on_dark	down	on_dark	Down
up	on_dark	up	off	closed
up	on_dark	up	on_bright	pressed
down	on_dark	up	on_dark	Up
down	on_dark	down	off	closed
down	on_dark	down	on_bright	pressed
up	on_bright	down	on_bright	Down
down	on_bright	up	on_dark	Up

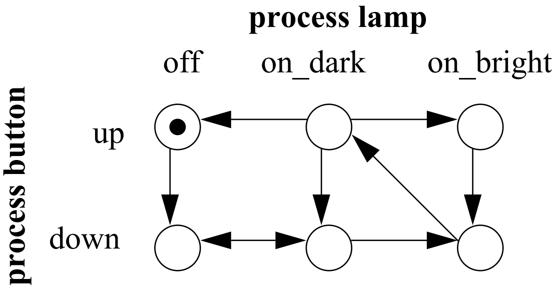
[1]: initial cluster state

figure C.15. Reduced cluster state graph



The more restrictive example of section C.1.3. is treated likewise. The resulting cluster state graph is shown in figure C.16.

figure C.16. Reduced cluster state graph for more restrictive example of section C.1.3.



Comparing figure C.15. and figure C.16. to figure C.9., we see that all states reacheable in the projection of the full cluster state graph are now again reacheable in the reduced cluster state graphs. Not all transitions from figure C.9. appear, because ([*up*, *off*], [*down*, *on_bright*]) has been decomposed into its component transitions: ([*up*, *off*], [*down*, *off*]), ([*down*, *off*], [*down*, *on_dark*]) and ([*down*, *on_dark*], [*down*, *on_bright*]). Likewise, ([*up*, *on_dark*], [*down*, *on_bright*]) has been decomposed into its component transitions: ([*up*, *on_dark*], [*up*, *on_bright*]) and ([*up*, *on_bright*], [*down*, *on_bright*]).

It is obvious that this splitting of the interaction tree has also produced a whole range of new transitions adding to the behaviour of the system but having nothing to do with the original behaviour. This adds to the state-explosion problem which cluster reduction was actually intended to alleviate. Whether or not this method is appropriate must be judged on a case to case basis.

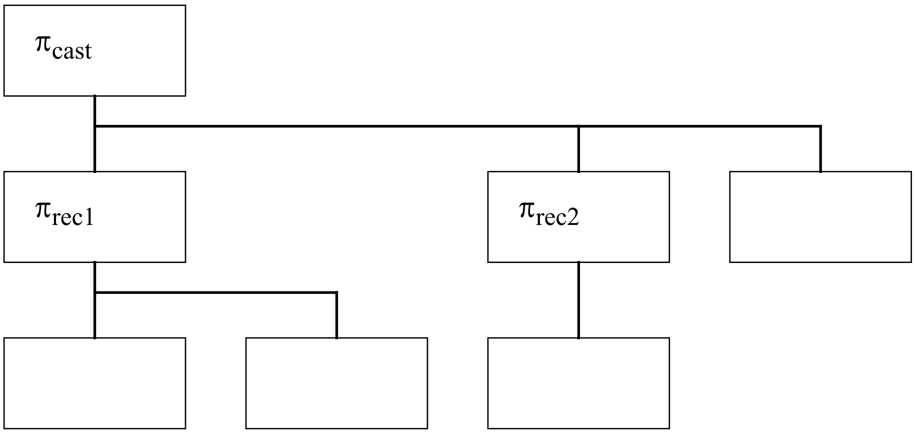
Some of the transitions of figure C.15. and figure C.16. are avoidable, however, if certain structural measures are taken to prevent transitions firing out of context.

C.3.6. Measures reducing the number of additional transitions

Until now we have assumed that process states are arbitrary when pulses are cast. This assumption is too general and the enhancement discussed here addresses the issue of making better use of available information.

When a process π_{cast} casts a pulse p to the first receiving process π_{rec1}, the state of π_{cast} must be the post-state of a transition able to cast p. If it were not then no such transition could have been executed and p would not be cast. Therefore, when π_{rec1} receives that pulse, the state of π_{cast} is not arbitrary.

figure C.17. Interaction tree structure.



The pulse cast is propagated to the processes below π_{rec1} in the interaction tree. It could be that the process π_{cast} appears below π_{rec1} in the interaction tree. If this is the case, π_{rec1} could change its state again. This is not certain however, as it depends on whether or not the interaction was propagated so far and whether the triggered transition was enabled.

When the execution of the part of the interaction tree below π_{rec1} is complete, the pulse p is cast from π_{cast} to π_{rec2}. π_{rec2} being the state after π_{rec1} in the cast order of π_{cast}.

At this point, the state of π_{cast} is either still the post-state of the transition that cast p, or it is in the post-state of another transition that was executed in the part of the interaction tree below π_{rec1}.

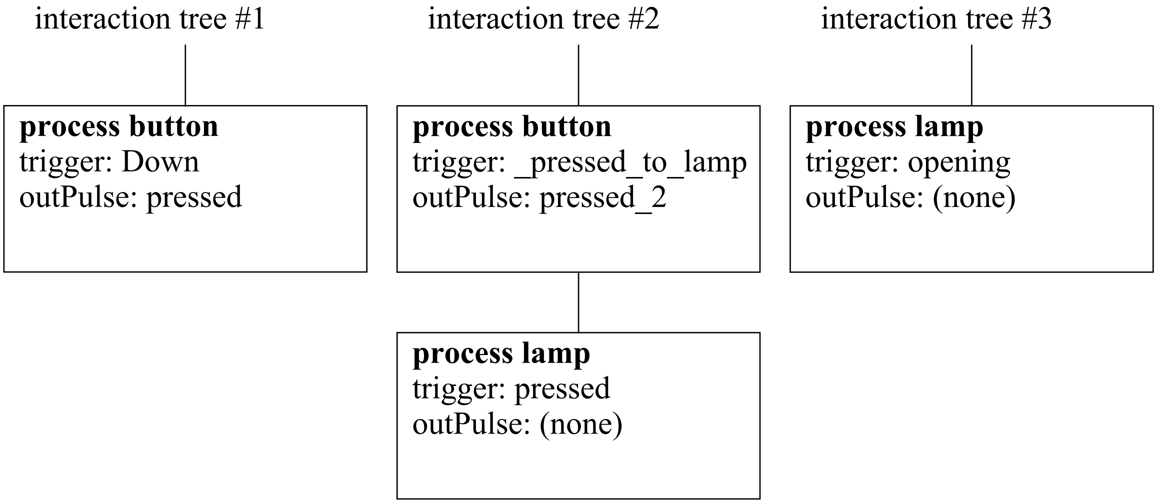
Now, supposing the tree were split between π_{rec1} and π_{rec2}. In the previous approach, the interaction tree starting with π_{rec2} is independent of that from which it was separated. Therefore it can be triggered independently of the state of π_{cast}. This approach does not make full use of our knowledge of the system. The triggering of the tree starting with π_{rec2} can be limited to cases where the state of π_{cast} is the post-state of a transition executed during the full interaction tree prior to that point.

To implement this in practice, π_{cast} is added as the root of the split off interaction tree T. Non state-changing transitions are added from all states at which T can be triggered. These transitions trigger the interaction tree T and are triggered by messages created for the purpose.

C.3.7. Example of reducing the number of additional transitions

Once again, we return to our example with the processes *button* and *door*. Interaction tree #2 (from figure C.14.) has the process *button* added at its root. The trigger is a new message introduced for the purpose.

figure C.18. process button *is added as a root to interaction tree #2 (compare figure C.14.)*



The process state graph of process *button* is extended by adding a transition 3 as shown in figure C.19. Transition 3 is a non state changing transition added at state *down*. State *down* is the post-state of the only transition. (The model checker does not actually modify the CIP-model but behaves as if this modification had taken place). No such transition is added at state *up*, and therefore the interaction tree can only be triggered when the state of *button* is *down*.

figure C.19. process state graph of process button with transition 3 added

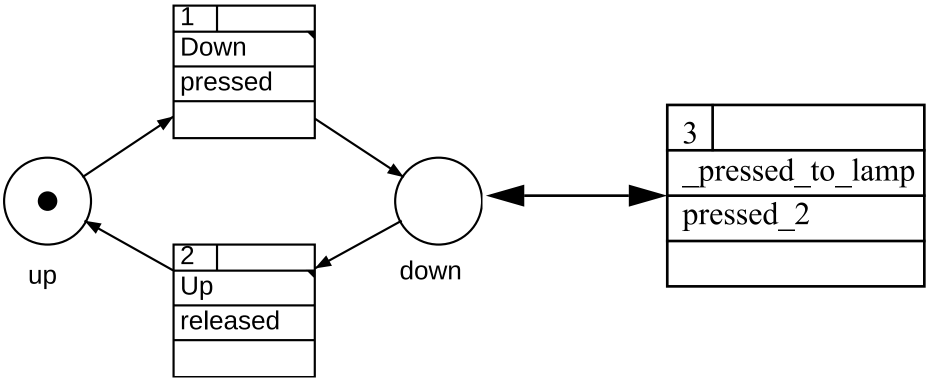


table C.4. Table of transitions replacing that of table C.3.

pre-state		post-state		trigger
button	lamp	button	lamp	
up[1]	off[1]	down	off	Down
up[1]	off[1]	up	on_dark	opening
down	off	up	off	Up
down	off	down	on_dark	opening
up	on_dark	down	on_dark	Down
up	on_dark	up	off	closed
down	on_dark	up	on_dark	Up
down	on_dark	down	off	closed
down	on_dark	down	on_bright	pressed
down	on_bright	up	on_dark	Up

1: initial cluster state

figure C.20. Reduced cluster state graph corresponding to table C.4.

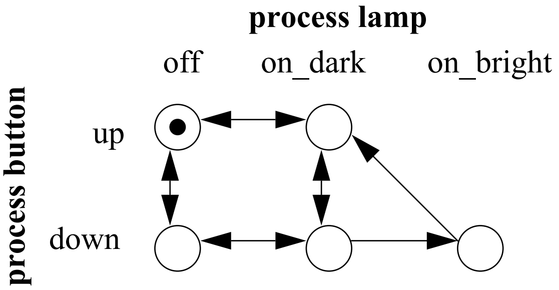
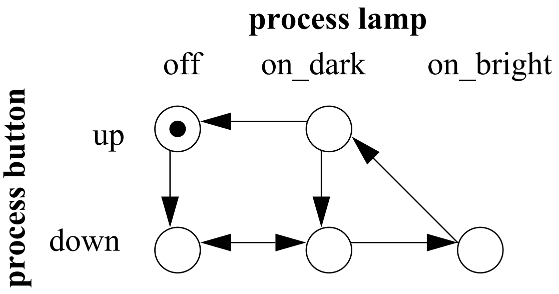


figure C.21. Corresponding cluster state graph for more restrictive example of section C.1.3.



Note that figure C.20. has two transitions and one cluster state less than figure C.15. and figure C.21. offers the same reduction over figure C.16.

C.3.8. Consequences of cluster reduction and a *caveat*

Cluster reduction is a useful tool against the state explosion problem. However, by removing some processes, some interaction mechanisms are also lost. As a result some transitions can be executed when otherwise this would not be possible. The reduced cluster state graph can contain transitions and states not occurring in the (projected) full cluster state graph. The effects of these additional transitions on the conditions being checked was discussed in section C.1.4.

In the example of section C.3.5. it was shown that although the set of states reacheable in the reduced cluster includes all states reacheable in the projected full cluster, some transitions are modified due to interaction trees being interrupted. As a result of this, the single cluster transition of the full cluster is broken up into several transitions. A transition $t_{full}(s_a, s_b)$ of the full cluster may be split up into a series of individual transitions $t_0(s_a, s_1)$, $t_1(s_1, s_2)$, $t_2(s_2, s_3)$, ... $t_n(s_n, s_b)$. As a result there may no longer be a direct transition from s_a to s_b. All paths from s_a to s_b may pass through the states s_1, s_2, s_2, ... s_n, some of which may not be reacheable in the projected full state graph.

This may seem of little consequence, but it could be, for example, that we are testing the condition *eventually*(*ack*) where $s_i \in$ *ack* and i $\in$ {1, 2, ... n}. If transition t_{full} is a transition of all paths failing this condition, and *eventually*(*ack*) is failed in the projected full system graph, then the condition could be wrongly verified in the reduced system graph.

How can we obtain security against such cases? The problem does not lie in tree splitting. Examples can be made where tree splitting is not necessary but the problem still occurs. The problem lies in the basic concept of cluster reduction.

Is it possible to make the model checker recognise that certain cluster states are only intermediate stages on cluster transitions, and not reached by the full cluster?

C.3.9. Handling part-transitions

If a process is missing from a cluster, the reduced cluster does not ‘know’ the state of this missing process. Therefore it cannot know if a pulse triggering this process in the full cluster will or will not result in the missing process sending pulses to the processes in the tree below it.

So is there no way of knowing how the full cluster will behave?

All is not lost. We may not know exactly how the missing cluster would react, but we know all possible reactions. These can be represented by an equivalent non-deterministic structure without adding to the overall complexity of the system. The function *outPulses* returns all possible outPulses which a process can send as the result of a given trigger.

$\mathit{outPulses}$: $(\mathbf{In}_\pi \cup \mathbf{E}) \times \prod :\rightarrow 2^{\mathbf{Out}\pi}$
$\mathit{outPulses}(\mathrm{in}, \pi\_0) = \{u \in \mathbf{Out}_{\pi\_0} \mid \exists r \in \mathbf{R}_{\pi\_0}: (\mathrm{in} = \mathit{trig}(r)) \wedge (u = \mathit{outPulse}(r))\}$

Note that the set returned by *outPulses* can contain the nul outPulse. This is contained in the definition of $\mathbf{Out}_\pi$ (see section 5.3.3. on page 83) and is contained in the set returned by *outPulses* if the trigger parameter of the function call triggers at least one transition without outPulse.

The equivalent generic state diagram for a missing process π_{missing} is a single state with transitions pairing all in $\in$ $\mathbf{In}_{\pi\_\mathrm{missing}}$ with all elements of *outPulses*(in, π_{missing}).

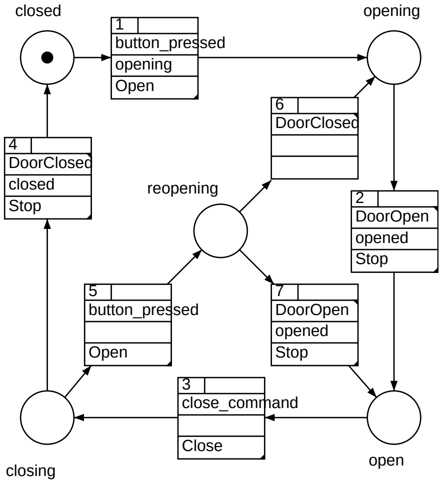
C.3.10. Example using non-deterministic replacement for process

In our example, process door is missing from the interaction tree.

Process *door* pairs triggers with outPulses as follows:

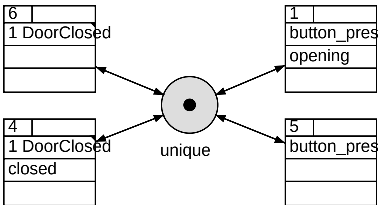
```
button_pressed -> opening
button_pressed -> none
DoorOpen -> opened
close_command -> none
DoorClosed -> closed
DoorClosed -> none
```

figure C.22. state diagram of process door



As the trigger *close_command* does not trigger a transition with an outPulse associated, it can be omitted. The trigger *DoorOpen* can also be omitted because the outPulse *opened* is sent only to process *controller* and this is also missing from the reduced cluster. The four remaining trigger, outPulse pairs form the equivalent state diagram of figure C.23.

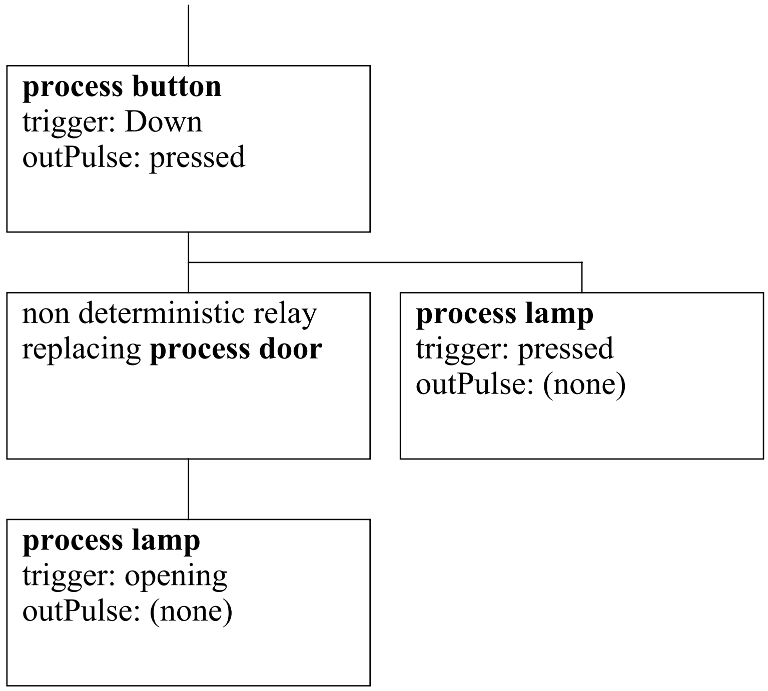
figure C.23. equivalent state diagram of process door *for reduced cluster*



The equivalent state diagram of process door has two purposes:

- to non-deterministically relay outPulse *button_pressed* from process *button* to process *lamp*.
- to non-deterministically relay the (externally sourced) trigger *closed* to process *lamp*.

figure C.24. interaction tree with process door *replaced by non-deterministic relay*



Although it would be possible to implement the CIP model checker using equivalent state diagrams of the type of figure C.23., for reasons of simplicity of implementation it was chosen to transfer this functionality to the adjacent processes. Thus trigger *closed* is sent directly from the external source to process *lamp* and the non-deterministic nature of the transmission of *button_pressed* is achieved by a non deterministic sending mechanism in process *button*.

The modified interaction tree is shown in figure C.26.

figure C.25. interaction tree of figure C.24. with non deterministic relay replaced by direct relay, but pulse cast made non deterministic so that the same behaviour is modelled

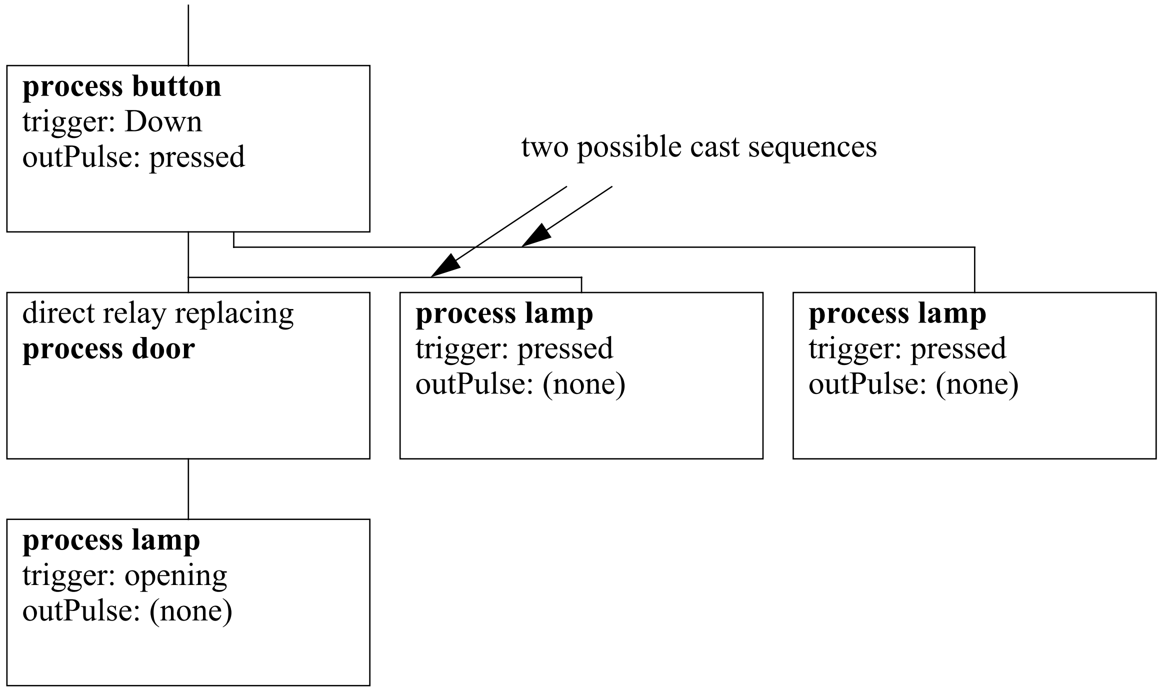


figure C.26. Modified interaction tree

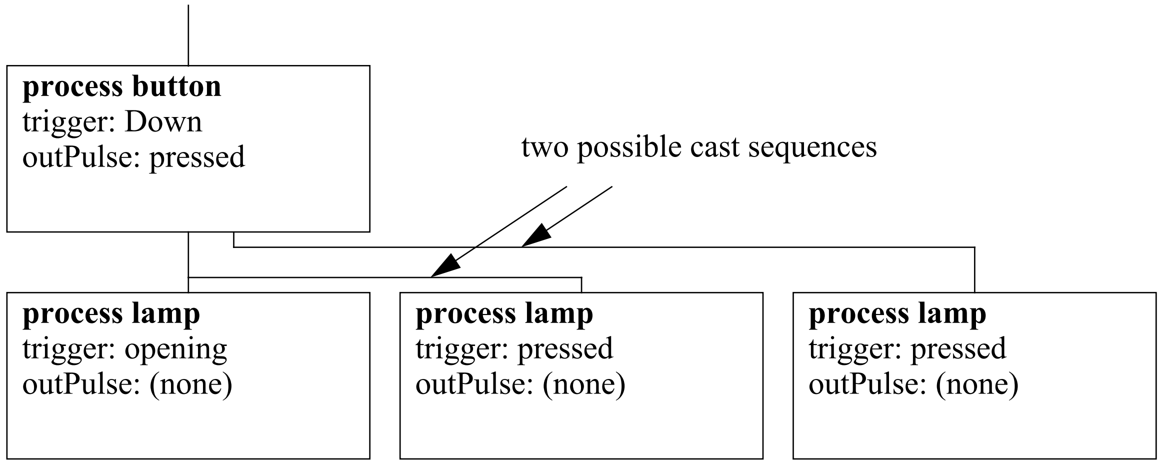


table C.5. Table of transitions

pre-state		post-state		trigger	notes
button	lamp	button	lamp		
up[1]	off[1]	down	off	Down	non-deterministic
up[1]	off[1]	down	on_bright	Down	
down	off	up	off	Up	
down	on_bright	up	on_dark	Up	
up	on_dark	down	on_bright	Up	
up	on_dark	up	off	closed	

1: initial cluster state

figure C.27. Reduced cluster state graph corresponding to table C.4.

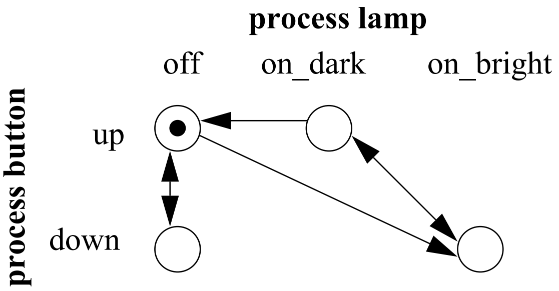
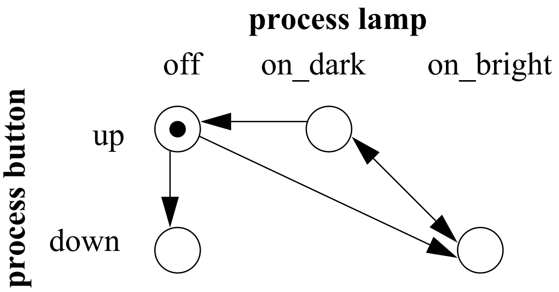


figure C.28. Corresponding cluster state graph for more restrictive example of section C.1.3.



The resulting state diagrams are closer to those of the projected cluster state diagram than any other approach discussed until now.

This is the method used for handling cluster reduction in the CIP model checker.

Appendix D: Traversing the State-Space

D.1. Introduction to state-space traversal

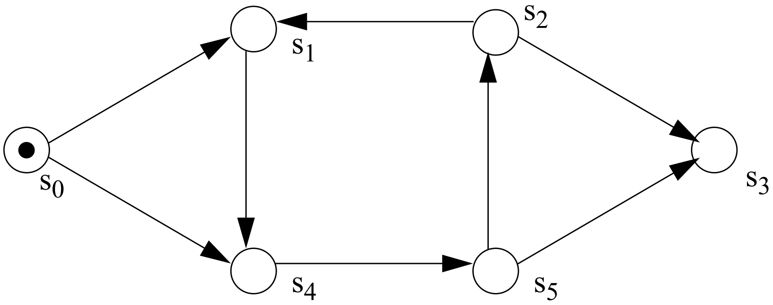
D.1.1. Purpose

Whether a property holds on a given model depends on the structure of that model. Some properties were introduced along with model checking basics in chapter 3. In chapter 5. system modelling concepts were applied to the CIP-Model. In this chapter methods will be introduced for looking at the system structure to ascertain whether conditions are fulfilled by that structure. Some methods presented are applicable for state space traversals in general. The focus of the chapter, however, is to show how they are specifically applied to CIP.

D.1.2. A simple example

The state graph of a simple finite state machine is shown in figure D.1.

figure D.1. A simple state-graph.



Suppose we wish to traverse this system to verify the following properties:
(1) *eventually*($\{s_3\}$, S)
(2) *eventually*($\{s_5\}$, $\{s_1\}$)
where S is the set of all states in the graph.

The meaning of *eventually* was defined in section 3.5.5. The meanings of these specific formulas are recapitulated briefly as follows:

Property (1) requires that for every state $s_n \in$ S, all continuing paths starting in s_n subsequently visit s_3.

(For the definition of *continuing paths* see section 3.3.5.)

Property (2) requires that all full paths visiting s_1 subsequently visit s_5.

Visual inspection of the state-graph reveals that the first property is failed by the full path (s_0, s_4, s_5, s_2, s_1, s_4, ...). No obvious path can be found failing the second property, but failure to produce a counter-example in a non-exhaustive search is no proof.

A method will be presented which inspects the state-space and proves or disproves properties of this type.

D.2. Proving the *eventually* property

The property *eventually*(*ack*, *req*) is failed iff there exists a path p starting in a state of *req* and not visiting any state in *ack* which takes one of the following forms:

(i) p reaches a dead-end.
(ii) p closes a fair loop.

proof: If *eventually*(*ack*, *req*) is fulfilled, then every full path visiting *req* subsequently visits *ack*. Therefore that path has a continuing sub-path which starts in *req* and visits *ack*.

If a path p starts in *req* and reaches a dead-end without reaching *ack*, p does not reach *ack*. The condition is not fulfilled.

If a path p starts in *req* and closes a fair loop before reaching *ack*, then there also exists a path p′ starting in *req* and repeating the fair loop infinitely. p′ is fair and infinite and does not reach *ack*. The condition is not fulfilled.

We have so far shown that the presence of a path p not visiting *ack* so that p runs from *req* to a dead end or closing a fair loop is sufficient to fail the condition *eventually*(*ack*, *req*), but to complete the proof, we must also demonstrate it is necessary.

Suppose there is an extended finite state machine in which *eventually*(*ack*, *req*) does not hold, but in which no path exists which starts in *req* and reaches a dead end or closes a fair loop without visiting *ack*.

Because the condition is failed, the system must contain a continuing fair path cfp so that cfp starts in *req* and does not reach *ack*.

If p is infinite, p has to visit at least one state infinitely (the state space being finite). Let $S_i(p)$ be the set of states visited infinitely by p. For every $s \in S_i(p)$, s has at least one successor in $S_i(p)$ (because p is infinite). Therefore there exists at least one loop of which all states are in $S_i(p)$. We call L(p) the set of loops visiting only states of $S_i(p)$. A path can be traced from every state of $S_i(P)$ to any other state of $S_i(p)$ visiting only states of $S_i(p)$ (because if this were not the case then the path p could not visit all states of $S_i(p)$ infinitely). As p is fair, a fair transition f enabled in at least one state of $S_i(p)$ must be executed infinitely. Therefore at least one post-state of f must be included in $S_i(p)$. At least one fair loop can thus be traced within $S_i(p)$. Let Q(p) be the set of fair loops traceable within $S_i(p)$. If p exists and is infinite, an infinite path p′(f) exists for every f ∈ Q(p) so that *visited*(p′(f)) ⊆ *visited*(p). The existence of p′(f) is contrary to our initial assumption and therefore p cannot be infinite.

So p must be finite and may not close a fair loop. The last state of p must have successors because it cannot be a dead-end. We call P′(p) the set of fair paths whose first part is identical to p and which do not visit *ack* except in their final state. The only paths of P′(p) which do not visit ack are infinite.

Therefore, the condition *eventually*(*ack*, *req*) is failed iff there exists a path p visiting a reacheable state of *req* and not subsequently visiting any state in *ack* so that p takes one of the following forms:

(i) p reaches a dead-end.
(ii) p closes a fair loop.

This property is at the heart of our verification algorithm.

D.2.1. Depth-first-search algorithm (DFS)

The depth-first-search method traces a single path through the state-space. States where the condition is known to be fulfilled are collected in a set we call *ful*.

We first introduce the simpler algorithm which does not take fairness into account. The fuller algorithm is discussed in section D.2.3. (page 188).

The search algorithm is as follows:

```
DFS (ack, req) {
    ful = empty set
    p = empty path
    sucp = empty list of sets
    while (req∩¬ful not empty) {
        p(0) = element from req∩¬ful
        sucp(0) = suc(p(0))
        if (sucp(0) empty AND p(0) ∉ ack) return p
        n = 0
        while (n>=0) {
            if (p has loop) return p
            if (sucp(n) empty OR p(n) ∈ ack)
                ful = ful + p(n)
            else {
                p(n+1) = element from sucp(n)
                sucp(n) = sucp(n) - p(n+1)
                n += 1
                sucp(n) = suc(p(n))
                if (sucp(n) empty) return p }
            if (p(n) ∈ ful) {
                if (n>0) sucp(n-1) = sucp(n-1) - p(n)
                p(n) = nil
                n -= 1 }
        }
    }
return nil }
```

The algorithm returns a counter example if found and nil otherwise.

explanation:

1) Every state in *ful* is either in *ack* or it’s successors are all in *ful*.

> **proof**: only in one place in the algorithm are states added to *ful*. This can be done to p(n) if *sucp*(n) is empty or p(n) is in *ack*.
>
> In the latter case, the above statement is obviously true.
>
> In the former case it is true because *sucp*(n) is initially *suc*(p(n)). No states are added to *sucp*(n) and states are only removed if they are in *ful* or immediately follow p(n) in the path . p(n) can only be placed in *ful* when it is at the end of the search path. p(n) therefore has no successor in the path and all states of *suc*(p(n)) are either in *sucp*(n) or in *ful*. If *sucp*(n) is empty, then all successors of p(n) are in *ful*.

2) If a dead-end is reacheable from a state s by a path not visiting *ack*, then s cannot be placed in *ful*.

> **proof**: if a dead-end is reacheable from s by a path not visiting *ack*, then s cannot be in *ack* and also either s is a dead end or s has at least one successor s_1 so that a dead-end is reacheable from s_1 by a path not visiting *ack*. s cannot be placed into *ful* when it has a successor not in *ful*. Likewise, this successor s_1 cannot be placed into *ful* and so on until the dead-end is considered. So for s to be placed into *ful*, the dead-end must be placed into *ful* first. This is not possible, because in order to be placed into *ful*, the dead-end must first become p(n). At this stage the algorithm sees that *suc*(p(n)) is empty and terminates. p(n) can therefore not be placed into *ful*. Neither can s.

As a result of this, we know that if a state is in *ful*, then no path from that state can reach a dead end without visiting *ack*.

3) If a path from a state s closes a loop without visiting *ack*, then s cannot be placed in *ful*.

> **proof**: if a path p from s closes a loop without visiting *ack*, then p has a subpath p´ so that p´ does not close a loop or visit *ack* but there exists a state s´ in *suc*(*last*(p´)) so that s´ is a state of p´. *last*(p´) cannot be placed in ful while s´ is not in *ful*. s´ cannot be placed in ful while *last*(p´) is not in *ful*. And whilst these states are not in *ful*, s cannot be in *ful*.

consequence: If a state is in *ful*, then there does not exist any path from s reaching a dead-end or closing a loop without visiting *ack*.

consequence: All full paths starting in *ful* reach *ack*.

If the algorithm does not find any counter-example, it will terminate when all states of *req* are also in *ful*.

If all states of *req* are in *ful*. All full paths starting in *req* reach *ack*.

example: An example of this algorithm is discussed step by step in section E.1.

The algorithm needs to be expanded however, to take fairness into account. Suppose the transition (s_2,s_3) is fair, then the loop [s_1,s_4,s_5,s_2,s_1] is unfair. Then this counter-example of figure E.5. is not valid.

D.2.2. Expanding the DFS algorithm to accommodate fairness.

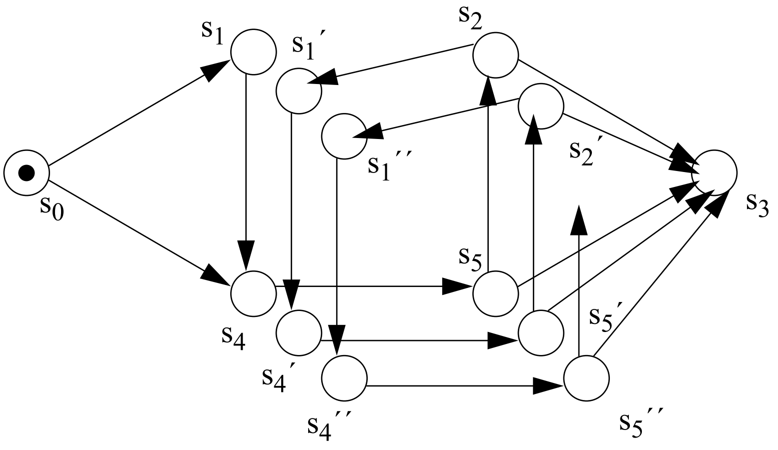
The DFS algorithm as introduced in section D.2.1. must be extended to accommodate fairness.

We must find a way of distinguishing between a fair loop (which can be repeated infinitely often sequentially) and an unfair loop (which cannot).

One way to do this is by *unravelling* the unfair loops.

Unravelling a loop is a transformation of the system which removes a loop by introducing new states where a fair path would otherwise revisit states. As these loops are unfair, a finite number of states is sufficient for unravelling and the finite nature of the system is preserved.

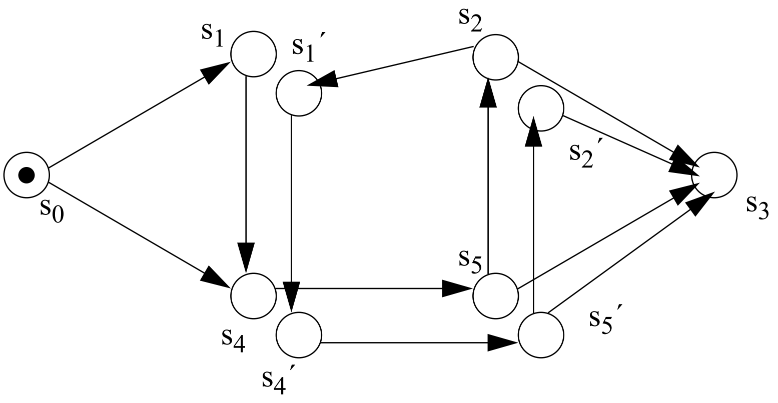
figure D.2. Unravelling of loop (s_1,s_4,s_5,s_2) of example system of figure D.1. (assuming that the transformation (s_5,s_3) is fair.)



Applying the DFS algorithm of section D.2.1.to an unravelled state graph would verify the system correctly. In practice, however, this is not possible as the states cannot be held in a finite memory. Simplification is required.

As we learn nothing new from paths traversing the loop twice in sequence or more, we can reduce the unravelled state-space by removing these.

figure D.3. Reduction of system of figure D.2. by removing paths traversing loops more than once in sequence.)



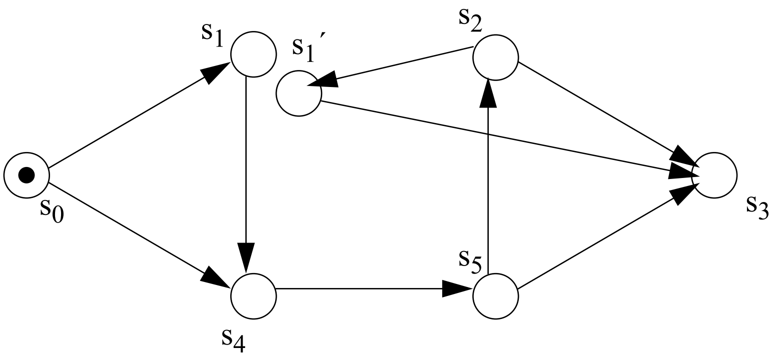
The system no longer displays all the behaviours of the original system. However, if there exists a fair path traversing a loop n-times in sequence, there also exists a fair-path traversing the loop once in sequence. A new loop is closed when the path reaches a state that is already in it's path history, and multiple traversals of the same loop do not contribute to the path history.

A further simplification is possible in DFS by removing states that occur twice due to unravelling, and instead constructing new transitions in the search path to by-pass these.

These new transitions have the same fairness constraints as the paths they replace. The paths found are now no longer identical to those of the original system, but the path histories remain the same, which is sufficient for detecting loops.

Thus in the example below, the transition (s_1',s_3) replaces the path (s_1',s_4',s_5',s_3). The new cluster-transition is treated as having as component the same process transition that causes the fairness of (s_1,s_3).

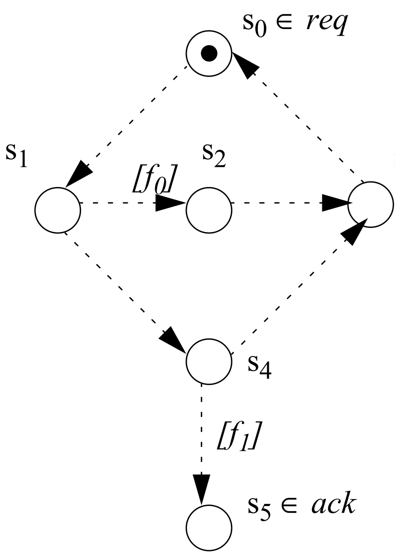
figure D.4. Modified state-graph for DFS with search path (s_0,s_4,s_5,s_2,s_1).)



The paths found are now no longer identical to those of the original system, but the path histories remain the same, which is sufficient for detecting loops.

Unravelling alone is not sufficient to detect all fair loops. Let us consider the following example:

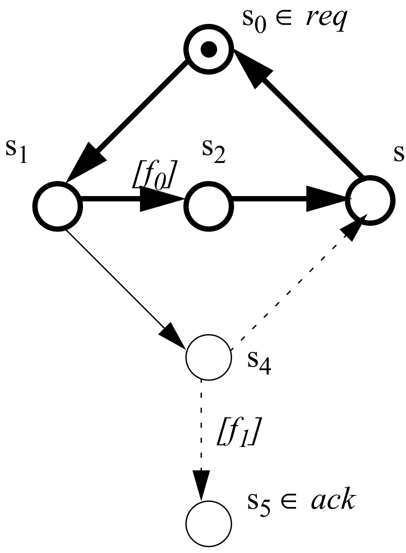
figure D.5. Example system, with fair transitions f_o and f_1



The search path can traverse the system in many different manners.

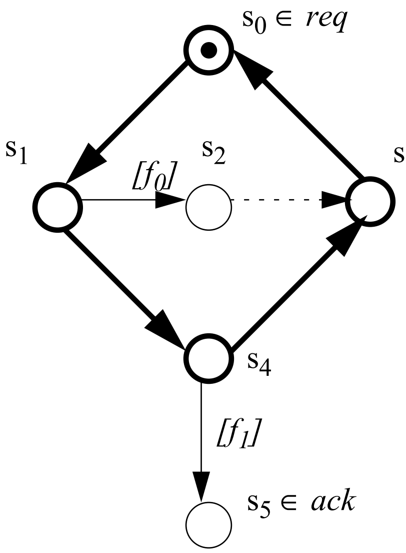
The first example of traversal sees the search path follow (s_0, s_1, s_2, s_3, s_0). A fair loop is closed without *ack* being visited. This is a counter-example and the condition is disproved.

figure D.6. Counter-example disproving condition.,



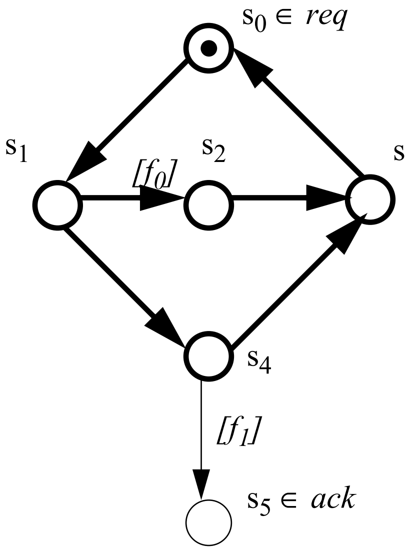
The second example of traversal sees the search path follow (s_0, s_1, s_4, s_3, s_0). A loop is closed. The loop is unfair because the transitions f_o and f_1 are enabled and disabled but not executed by the loop.

figure D.7. Example of figure D.5., with unfair loop (s_0, s_1, s_4, s_3, s_0)



We unravel the loop along (s_0, s_1) and continue exploration from s_1. At s_3, a loop is again closed. The path now closes the loop (s_0, s_1, s_4, s_3, s_0, s_1, s_2, s_3). This loop is again unfair because it enables and disables f_1 without executing it.

figure D.8. Closing of loop (s_0, s_1, s_4, s_3, s_0, s_1, s_2, s_3)

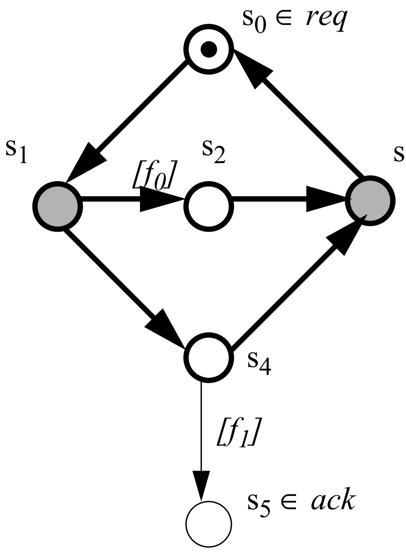


The unfair loop, [s_0, s_1, s_4, s_3, s_0, s_1, s_2, s_3] contains a sub-loop [s_3, s_0, s_1, s_2, s_3] which is fair.

This example shows that a system with multiple interconnected unfair loops can have fair loops. To detect these, it is necessary to look at the whole system of interconnected unfair loops which can be very large and indeed take up much of the total state space.

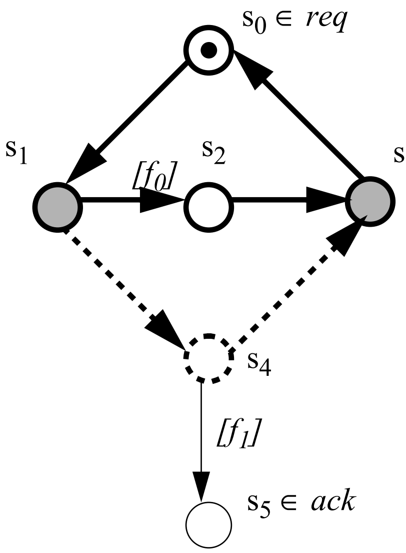
The loop system consists of several interconnected loops which we divide into path sections connecting nodes.

figure D.9. The loop system has two nodes (s_1 and s_3), connected by three path sections.



If a subset of these path sections form a system of loops not containing a fair loop, the section (s_1, s_4, s_3) cannot be part of any of the interconnected loops because f_1 is disabled by (s_4, s_3) but not executed by any loop of the search path. The section is thus removed.

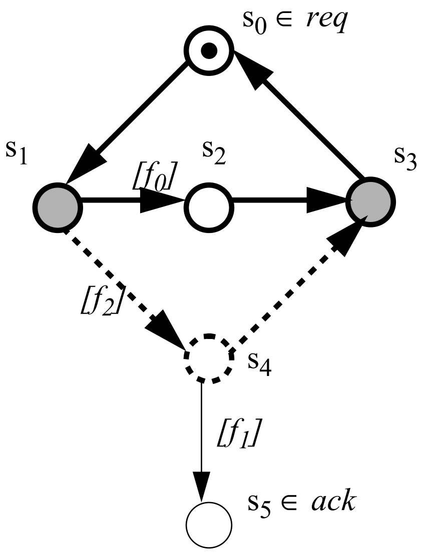
figure D.10. The path section (s_1, s_4, s_3) is removed from the system of loops.



The remaining path sections form a loop which is fair. This is a counter-example and the search can be ended.

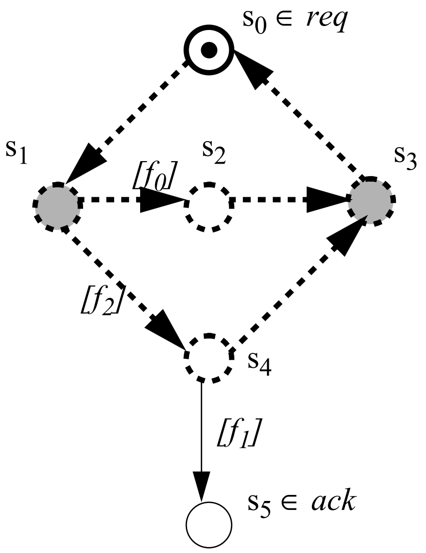
Modifying the example slightly, let us suppose (s1, s4) is also a fair transition:

figure D.11. The path section (s_1, s_4, s_3) is removed from the system of loops and (s_1, s_4) is fair.



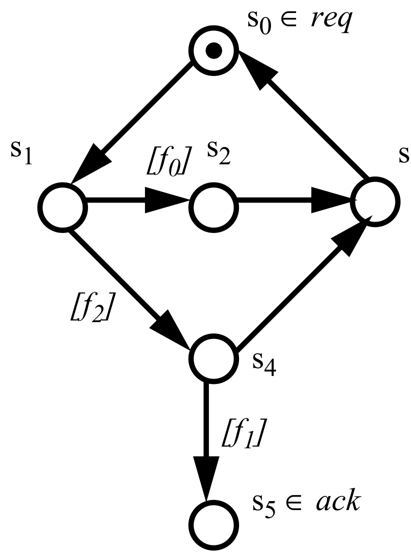
The remaining loop (s_0, s_1, s_2, s_3) is unfair because (s_1, s_2) disables f_2 and f_2 is not executed at any point of the loop. The path sections visiting s_1 are thus removed from the loop system.

figure D.12. The path section (s_0, s_1, s_2, s_3, s_0) is removed from the system of loops.



Only the initial state s_0 remains in the loop system. No fair loops are formed, therefore the initial loop system did not contain any fair loops. The search must continue. All removed sections are restored to the search path.

figure D.13. All removed sections are restored to the search path, which reaches s_5.



The search path reaches s_5 which is a member of *ack*. The *eventually* condition is proved.

The search path reaches *ack*, and all states of the system can be placed into *ful*, verifying the condition.

D.2.3. Depth-first-search algorithm with fairness.

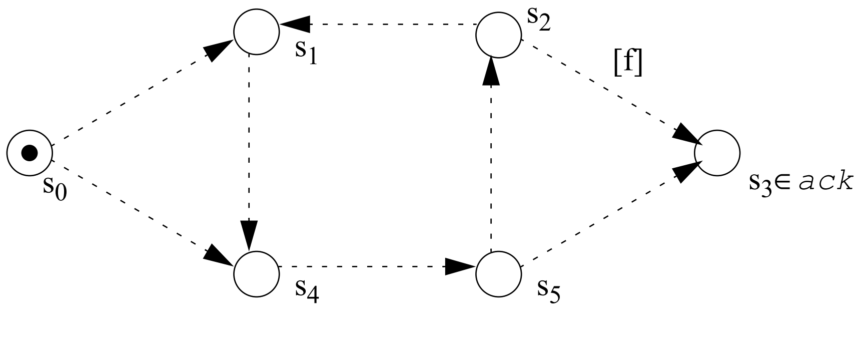
This algorithm is essentially the same as that of section D.2.1. but has following additions as explained in section D.2.2.:
- on closing a loop, we do not automatically return the path as counterexample, but first verify the fairness of the loop. If a fair subloop is found, we return this, otherwise the search continues.
- due to the introduction of new transitions, p is no longer necessarily a path allowed by the behaviour of the system. However, it represents a path of the system with some duplicate states removed. To reconstitute a correct path we introduce the function *path*(p). *path* is applied to all returned paths.
- preference is given to fair successors.

```
fairDFS (ack, req) {
    ful = empty set
    loopNodes = empty set
    p = empty path
    sucp = empty list of sets
    while (req∩¬ful not empty) {
        p(0) = element from req∩¬ful
        sucp(0) = suc(p(0))
        if (sucp(0) empty AND p(0) ∉ ack) return path(p)
        n = 0
        while (n>=0) {
            if (p(n) closes loop) {
                if p has fair subloop **return path(p)**
                else {
                    loopNodes = loopNodes + p(n)
                    if there exists k so that p(k) on loop and
                    sucp(k) not empty {
                        p(n = n+1) = p(k)
                        loopNodes = loopNodes + p(n) }
                    }
                }
            if (sucp(n) empty OR p(n) ∈ ack)
                ful = ful + p(n)
            else {
                p(n+1) = element from sucp(n)
                sucp(n) = sucp(n) - p(n+1)
                n += 1
                sucp(n) = suc(p(n))
                if (sucp(n) empty) **return path(p)** }
            if (p(n) ∈ ful) {
                p(n) = nil
                n -= 1 }
        }
    }
return nil }
```

This algorithm is equivalent to that of section D.2.1. for systems without fair transitions because in such a case no loop can be unfair so all loops are valid as counter examples.

To illustrate this algorithm, we return to the example of figure D.1.and the condition *eventually*($\{s_3\}$, S)
The transition (s_2, s_3) is marked as fair.

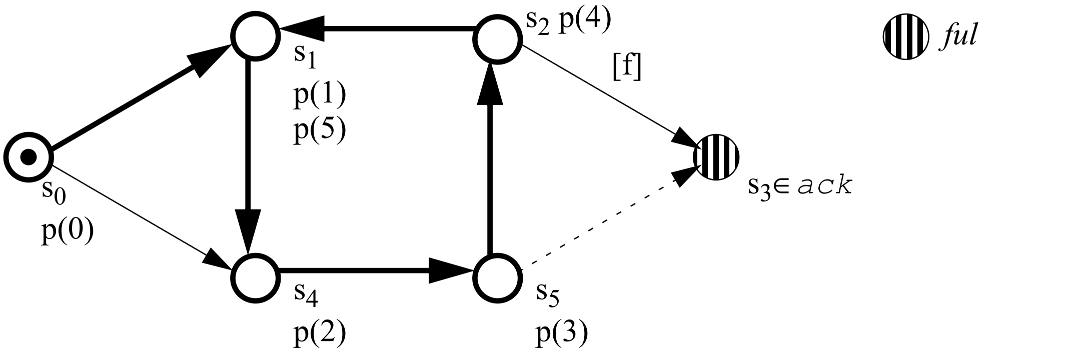
figure D.14. State-graph from figure D.1. with transition (s_2, s_3) marked fair.



Our initial search path is (s_0,s_1,s_4,s_5,s_3) as shown in figure E.3. s_3 is placed in ful as shown in figure E.4. and the path closes a loop at s_1 as shown in figure E.5.

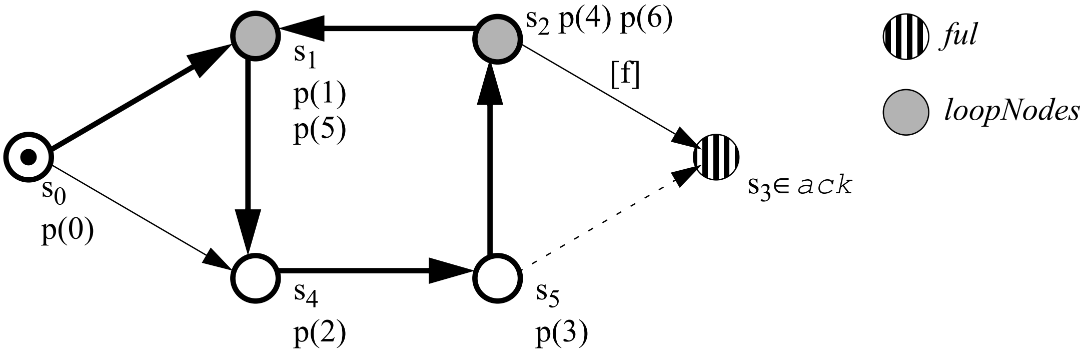
However, in contrast to the example of figure E.5., this loop is not fair and has no fair subloops.

figure D.15. The search path closes a loop.



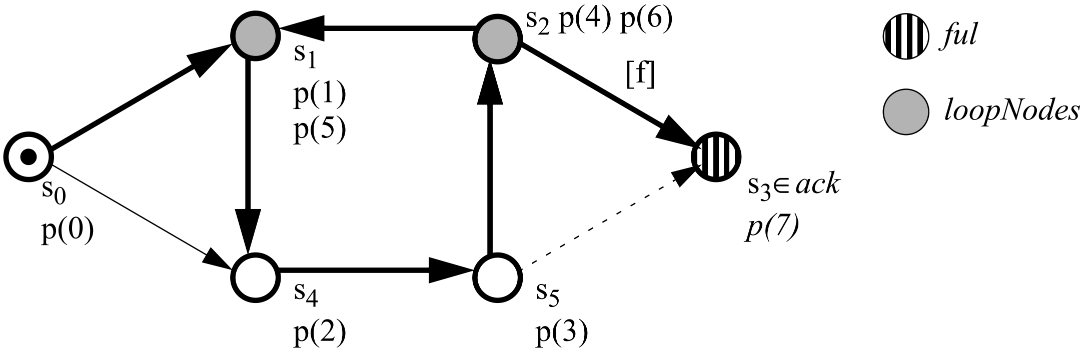
s_1 is placed in *loopNodes*, the loop has a state with not-empty *sucp*, and this state (s_2) is added to the end of the path as p(6) and also to *loopNodes*.

figure D.16. Search path after closing an unfair loop.



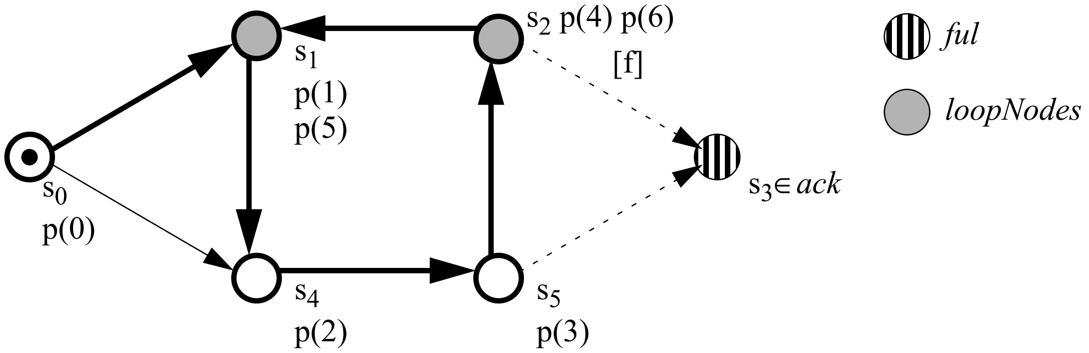
From s_2, the search path reaches s_3 again.

figure D.17. The search path reaches ack.



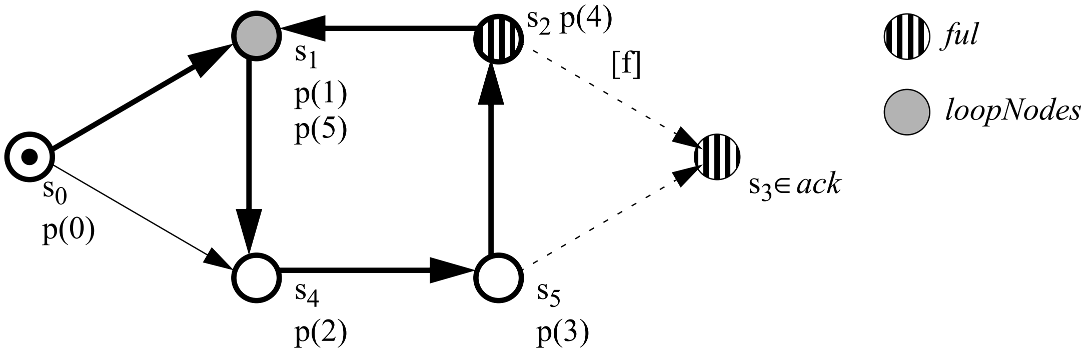
s_3 is in *ack*, it is removed from the search path.

figure D.18. The search path again closes a loop.



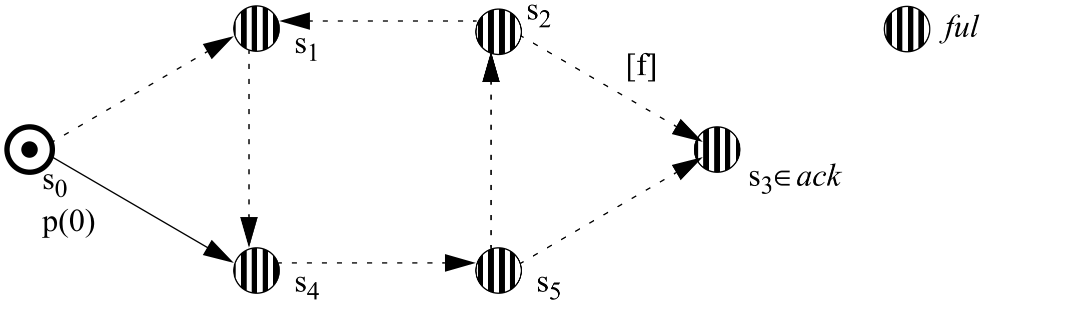
The last state of the search path (s_2) once again closes an unfair loop. This time, however, no state of the loop has a not-empty *sucp*. p(6) is removed from the search path and is placed into *ful*.

figure D.19. p(4) is placed into ful.



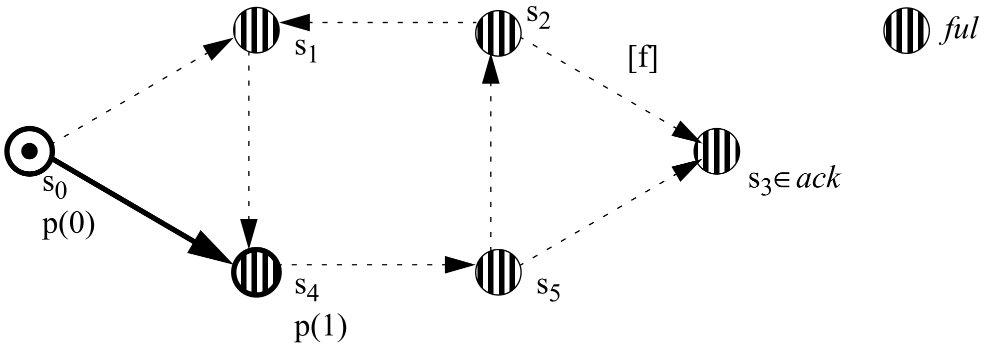
Likewise, p(5), p(4), p(3), p(2) and p(1) are removed from the search path and placed into *ful*.

figure D.20. The search path is reduced to (s_0).



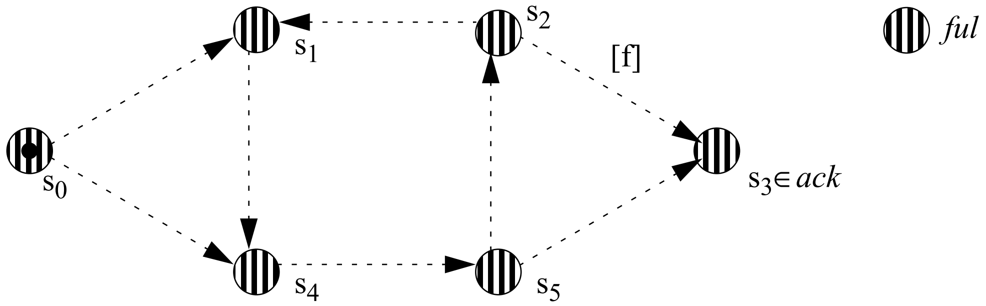
sucp of s_0 is not empty, so s_4 is added to the path.

figure D.21. The search path grows to (s_0, s_4).



s_4 is already in *ful*, however, and is removed again. *sucp* of s_0 is now empty and s_0 is also removed from the path leaving the path empty.

figure D.22. The search path grows to (s_0, s_4).



All elements of *req* are now in *ful*, therefore the condition is fulfilled and the demonstration is complete.

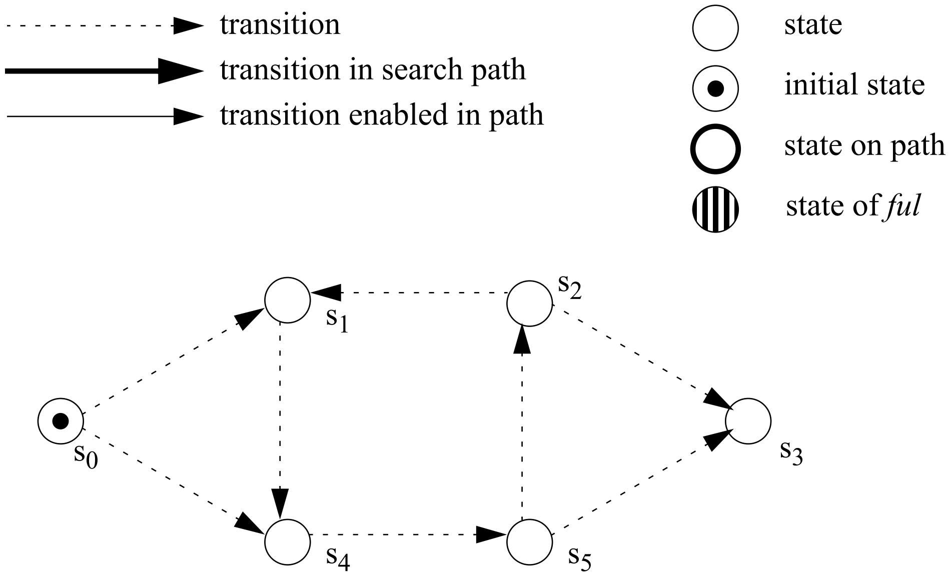
Appendix E: Traversal examples

E.1. DFS algorithm

E.1.1. Without fairness, property fails

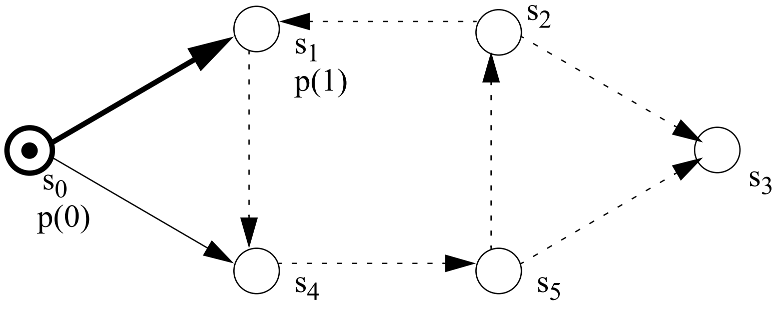
We consider the example from figure D.1. and check the property *eventually*($\{s_3\}$, S)

figure E.1. state-graph from figure D.1. before start of search.



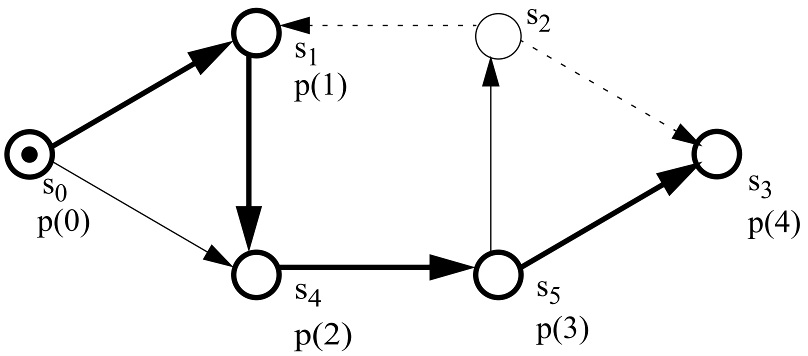
req = S.
We start by setting $p(0)=s_0$.
sucp$(p(0))=\{s_1,s_4\}$. We choose $p(1)=s_1$.

figure E.2. search path of length 1 starting at s_0.



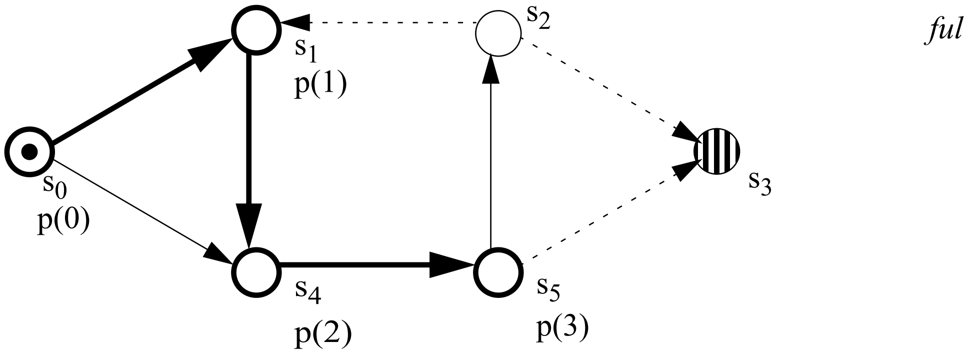
sucp$(p(1))=\{s_4\}$. So $p(2)=s_4$.
sucp$(p(2))=\{s_5\}$. So $p(3)=s_5$.
sucp$(p(3))=\{s_2,s_3\}$. We choose $p(4)=s_3$.

figure E.3. search path of length 4 starting at s_0.



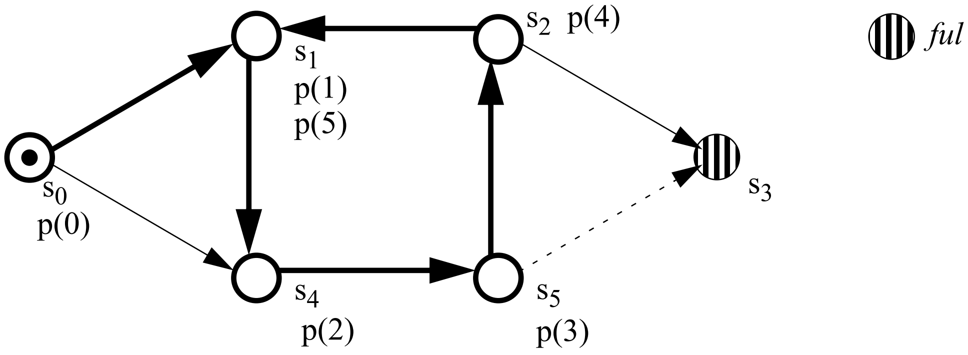
$s_3 \in$ ack. s_3 is placed in *ful* and removed from the search path.

figure E.4. ack *state is placed in* ful.



sucp(p(3))={s_2}. So p(4)=s_2.
sucp(p(4))={s_1,s_3}. We choose p(5)=s_1.
The path now completes a loop:

figure E.5. search path completes a loop.



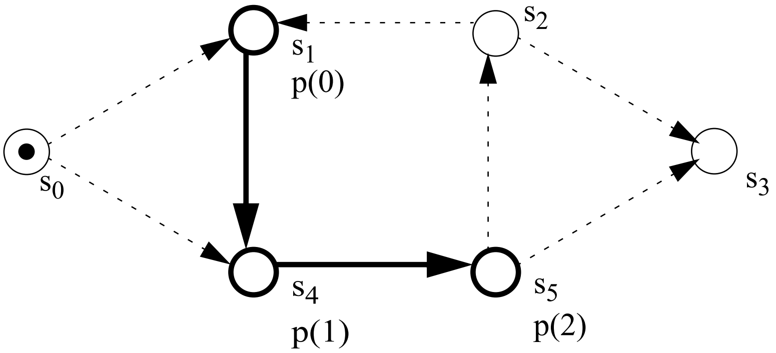
This is a counter-example and shows the condition is not fulfilled. The search need not be continued.

E.1.2. Without fairness, property holds

We now look at the second condition: *eventually*($\{s_5\}$, $\{s_1\}$)

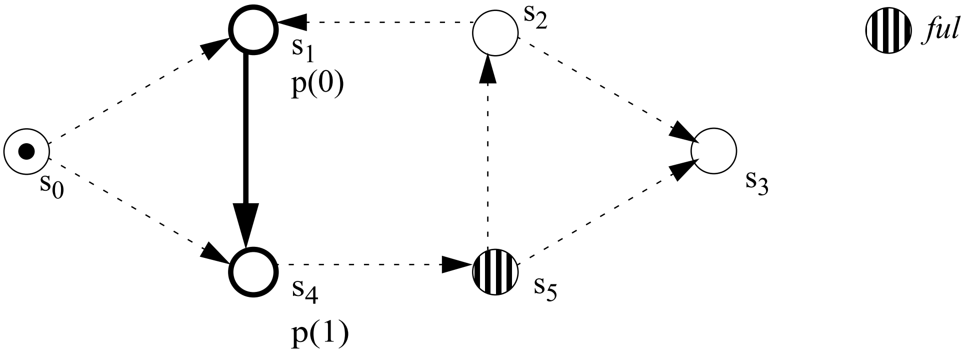
req = $\{s_1\}$
We start by setting p(0) = s_1.
sucp(p(0))=$\{s_4\}$. So p(1)=s_4.
sucp(p(1))=$\{s_5\}$. So p(2)=s_5.

figure E.6. path of length 2 starting at s_1



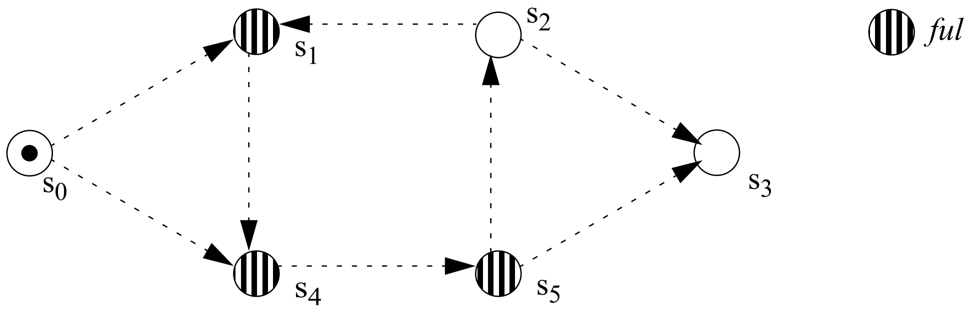
$s_5 \in$ ack. s_3 is placed in *ful* and removed from the search path.

figure E.7. ack *state is placed in* ful.



sucp(p(1))=$\{\}$. So p(1) is placed in *ful* and removed from the search path.
sucp(p(0))=$\{\}$. So p(0) is placed in *ful* and removed from the search path.
All *req* states are now in *ful* and the condition is fulfilled.

figure E.8. all req *states are in* ful.



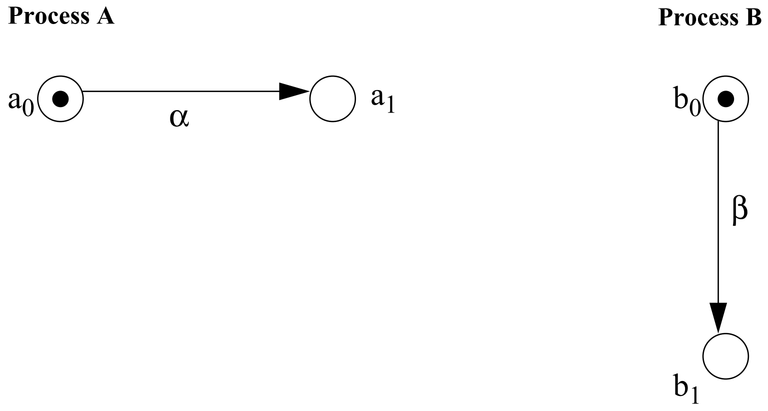
Appendix F: Partial Order Reduction

F.1. Introduction

Partial order reduction is the mapping of the full state graph onto a reduced state graph whilst conserving the properties of the full state graph. Such a mapping is worthwhile if the reduced state graph requires less memory or can be traversed faster.

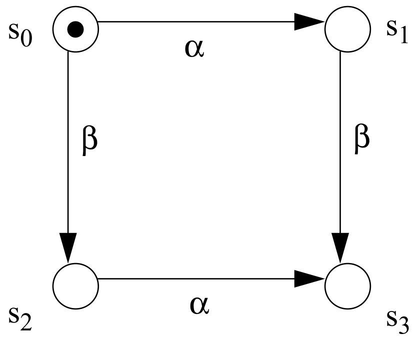
Let us consider the following example:

figure F.1. process-state graphs.



The system consists of two processes with each two states and one transition. These are combined into a cluster-state graph as follows:

figure 6.2. cluster-state graph



This cluster-state graph consists of 4 states, $s_0...s_3$, and 4 cluster transitions, (s_0,s_1), (s_0,s_2), (s_1,s_3), (s_2,s_3). It has two full paths, $[s_0,s_1,s_3]$ and $[s_0,s_2,s_3]$. If the intermediate states are irrelevant to the property being checked, we only need to follow one of these paths. The transitions α and β are **interleaving**. They can be executed in either order and it does not matter in which order they are traversed. The full state graph can thus be replaced by a reduced state graph containing only one of the above two paths.

In order to construct this reduced state graph, we first had to construct the full state graph. The method thus failed to reduce memory requirements. In this section and in chapter 6, methods are introduced for constructing a reduced state graph without constructing the full state graph.

F.2. Independence and visibility of transitions

F.2.1. Enabledness

Reminder: a process-transition t is **enabled** in a cluster-state s iff the pre-state of t. is an element of the state-vector of s.

In figure 6.2., α is enabled in s_0 and s_2. β is enabled in s_0 and s_1.

Reminder: a transition t is **disabled** in a cluster-state s iff t is not enabled in s.

In figure 6.2., all transitions are disabled in s_3.

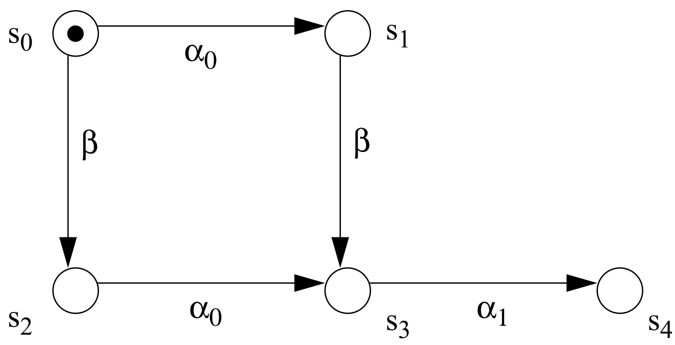
Reminder: the set **enabled**(s) contains all transitions which are enabled in cluster-state s.

In figure 6.2., *enabled*(s_0) = {α,β}, *enabled*(s_1) = {β}, *enabled*(s_2) = {α}, *enabled*(s_3) = {}.

Reminder: a cluster-transition t_1 **enables** a transition t_2 iff $t_2 \notin$ *enabled*(*pre-state*(t_1)) and $t_2 \in$ *enabled*(*post-state*(t_1)).

In the figure F.3. the cluster-transitions (s_1,s_3) and (s_2,s_3) enable the transition α_1

figure F.3. example of transition being enabled.

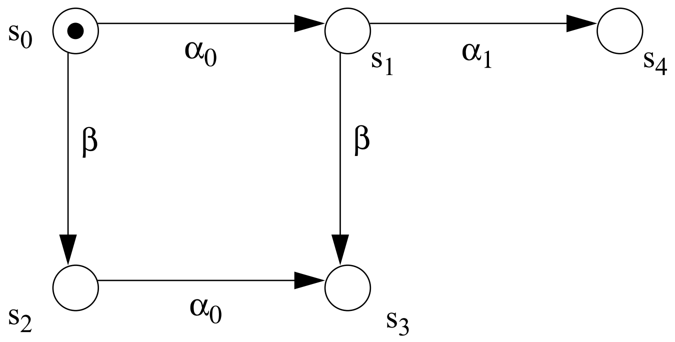


Note that this definition allows only cluster-transitions to enable. It does not always make sense to extend this label to process-transitions. In the above example, we cannot say whether or not α_0 enables α_1 because one occurence of α_0 enables α_1 whereas the other does not. Thus we restrict the ability to enable to cluster-transitions.

Reminder: a cluster-transition t_1 **disables** a process-transition t_2 iff $t_2 \in$ *enabled*(*pre-state*(t_1)) and $t_2 \notin$ *enabled*(*post-state*(t_1)).

In the example of figure F.4. the cluster-transitions (s_1,s_3) disables the transition α_1

figure F.4. example of transition being disabled.



F.3. Examples

F.3.1. Partial Order Reduction of a simple system

Reminder of rules of section 6.4.2:

rule C0: ample(s) = ∅ iff enabled(s) = ∅.

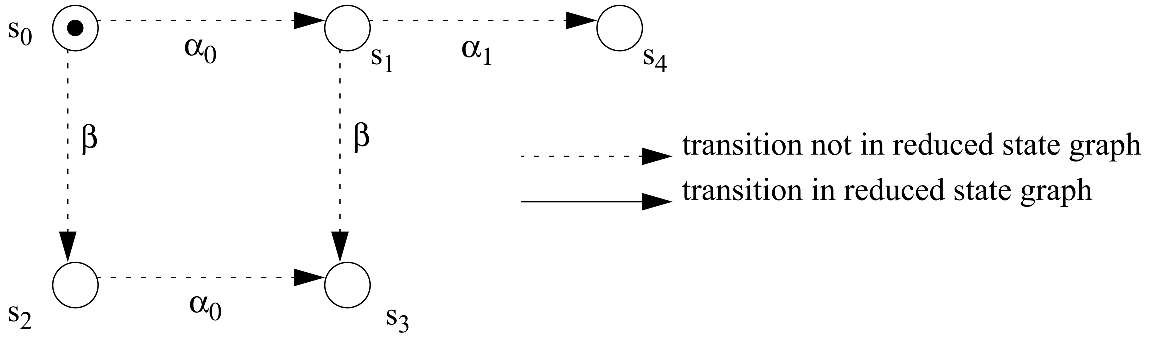
rule C1: Along every path in the full state graph that starts in s, the following condition holds: a transition that is dependent on a transition in ample(s) cannot be executed without a transition in ample(s) occurring first.

rule C2: If s is not fully expanded, then every $\alpha \in$ ample(s) is invisible.

rule C3: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in ample(s) for some state of the cycle.

Let us consider the system of figure F.5.

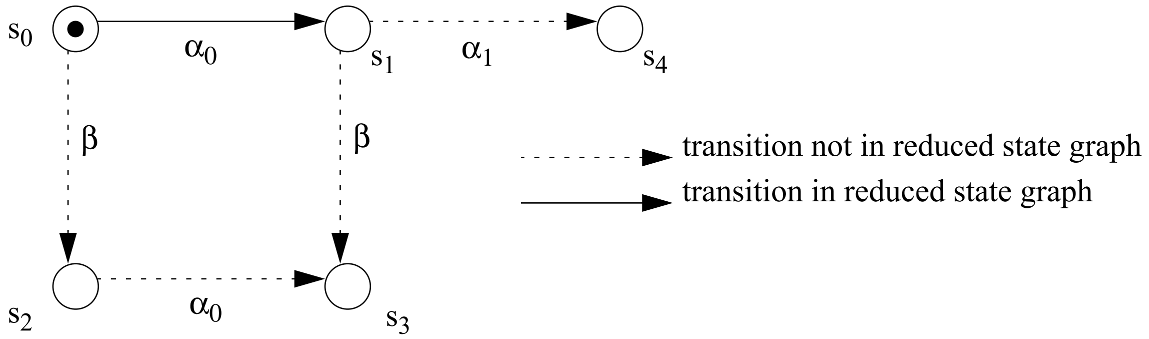
figure F.5. example system



s_0
α_0
s_1
α_1
s_4
β
β
transition not in reduced state graph
transition in reduced state graph
s_2
α_0
s_3

We start by determining *ample*(s_0).
enabled(s_0) is not empty so *ample*(s_0) must be $\{\alpha_0\}$, $\{\beta\}$ or $\{\alpha_0, \beta\}$ (rule **C0**).
α_1 is enabled by α_0 and disabled by β. Therefore it is dependent both. If *ample*(s_0) = $\{\beta\}$, then, α_1 cannot be executed without β being executed first (rule **C1**). This is not the case, so $\alpha_0 \in$ *ample*(s_0).
Therefore, *ample*(s_0) is $\{\alpha_0\}$ or $\{\alpha_0,\beta\}$.
As we are interested in reducing the state space, we choose the smaller of these and *ample*(s_0)=$\{\alpha_0\}$.

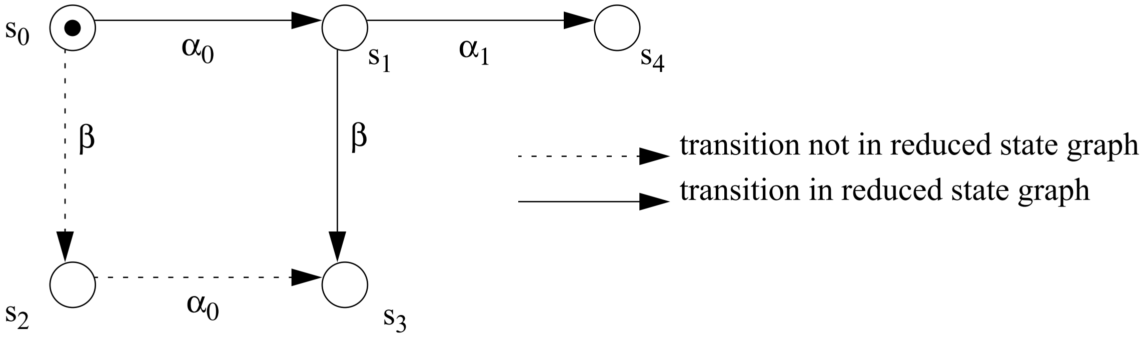
figure F.6. ample*(s_0)=$\{\alpha_0\}$*



s_0
α_0
s_1
α_1
s_4
β
β
transition not in reduced state graph
transition in reduced state graph
s_2
α_0
s_3

Now we must determine *ample*(s_1)
α_1 and β are dependent, therefore either both or neither must be included in *ample*(s_1).
Including neither would leave *ample*(s1) empty, therefore *ample*(s_0) = {α_1,β}

figure F.7. ample*(s_1)={α_1,β}*


s0
α0
s1
α1
s4
β
β
s2
α0
s3
transition not in reduced state graph
transition in reduced state graph

We have thus obtained the same reduction as in figure 6.3.

Note that the reduction of figure 6.4. which is also correct cannot be obtained by this method.

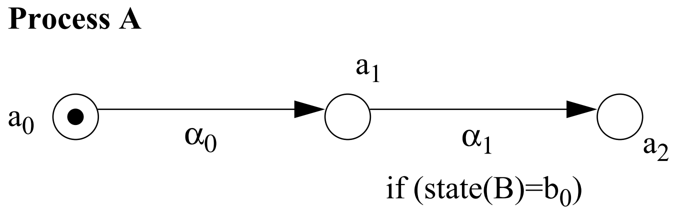
F.3.2. The same example solved using method of section 6.4.3.

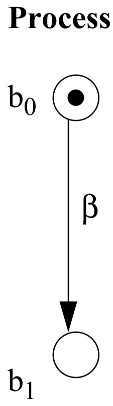
The decomposition of our example system into component processes as shown in figure F.8.shows that

- α_0 and α_1 are dependent because they are in the same process.
- α_1 and β share a variable (through inspection), therefore β is dependent on all transitions in A.

All transitions are dependent and therefore no reduction is possible.

figure F.8. component processes of system of figure F.5..





Clearly, this approach is insufficient for CIP-systems, where the dependency of processes makes significant state-space reduction difficult. In the above example it would lead to no reduction at all.

F.3.3. Degree of process enabledness

The degree of process enabledness is introduced in section 6.5.2.

In the example of figure F.4. (decomposed in figure F.8.), the only cluster state where α_1 is enabled is s_1 whose state vector is (a_1,b_0). This has both process-states in common with s_1, only one process state in common with s_0, s_3 and s_4 and no process-states in common with s_2. Thus:

$$dpe(s_1, \alpha_1) = 2,$$
$$dpe(s_0, \alpha_1) = dpe(s_3, \alpha_1) = dpe(s_4, \alpha_1) = 1,$$
$$dpe(s_2, \alpha_1) = 0$$

F.3.4. Example of process enabling and disabling

The concepts of process enabling and disabling are introduced in section 6.5.3.

In the example discussed above, (s_0,s_2), (s_1,s_3) and (s_1,s_4) process-disable (s_1,s_4). (s_0,s_1) and (s_2,s_3) process-enable (s_1,s_4).

F.3.5. Example of F.3.1 using rules of 6.5.4

Reminder of rules of section 6.4.2

rule R1: search for a non empty subset of invisible transitions of *enabled*(s) so that no process transition outside the subset is process-disabled by any member of the subset unless all members of *enabled*(s) disable that process transition.
if successful, select that subset as *ample*(s).
otherwise, *ample*(s) = *enabled*(s)

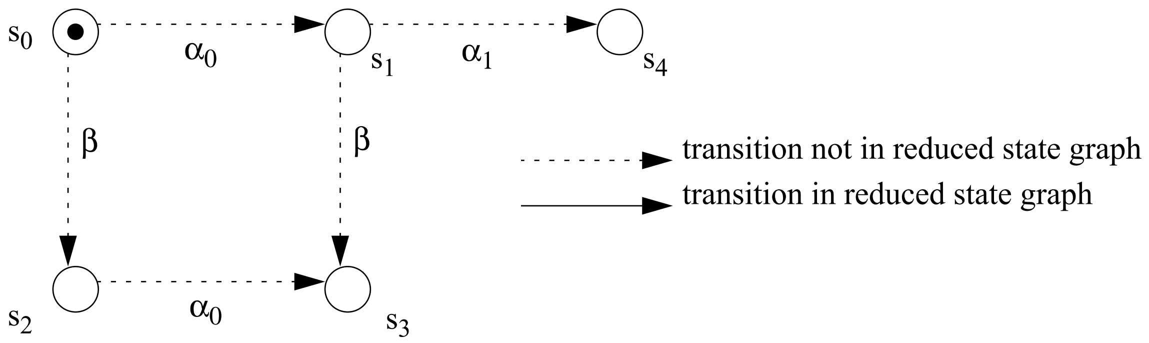
rule R2: check that no element in *ample*(s) closes a loop of the reduced state graph. If a loop is closed, check that in the loop no transition t is enabled but not included in *ample*(s) for any state of the loop, unless a path branches from the loop and t is not disabled by the first transition of that branch. Add transitions to reduced-state graph if necessary.

rule R3: dead ends are treated as transitions.

rule R4: If a fair transition is enabled in a state s, then *ample*(s) must include at least one fair transition.

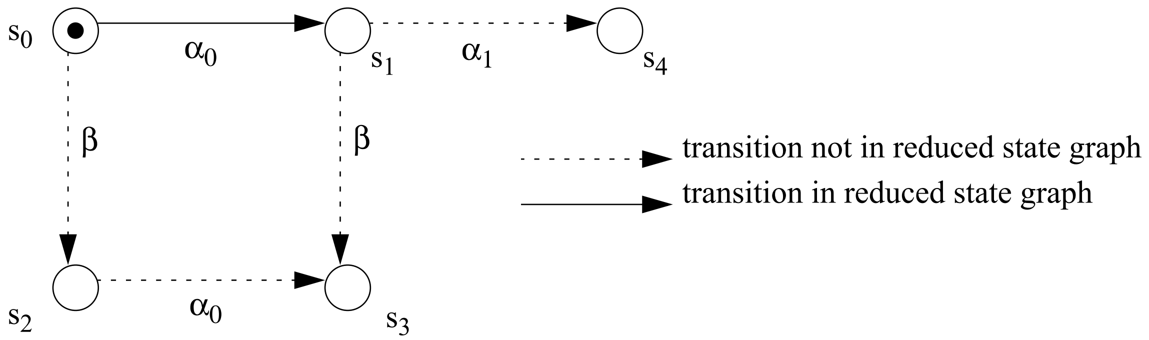
To illustrate these rules, we reconsider the example of figure F.5.

figure F.9. system from figure F.5.


s_0
α_0
s_1
α_1
s_4
β
β
transition not in reduced state graph
transition in reduced state graph
s_2
α_0
s_3

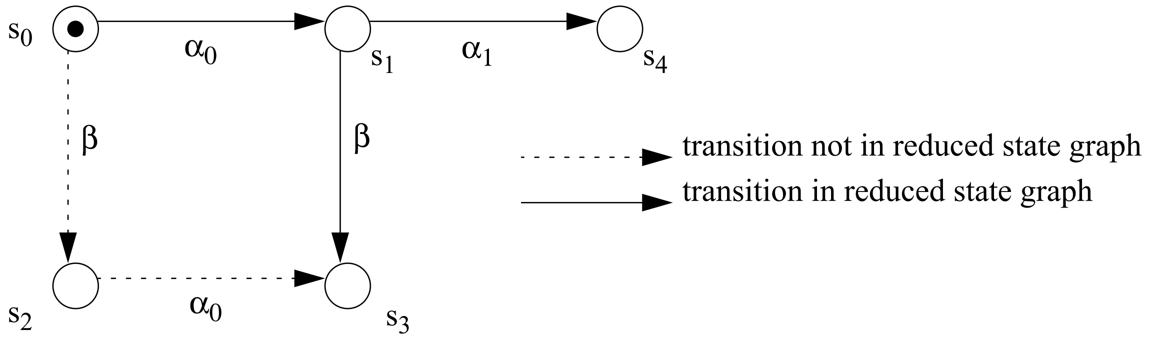
(s_0,s_2) process-disables α_1 whereas (s_0,s_1) does not affect the *dpe* of any state.
Therefore *ample*(s_0) = {α_0}.

figure F.10. ample$(s_0) = \{\alpha_0\}$


s0
α0
s1
α1
s4
β
β
transition not in reduced state graph
transition in reduced state graph
s2
α0
s3

(s_1,s_4) process-disables β and (s_1,s_3) process-disables α_1.
Therefore *ample*$(s_1) = \{\alpha_1, \beta\}$.

figure F.11. ample$(s_1) = \{\alpha_1, \beta\}$

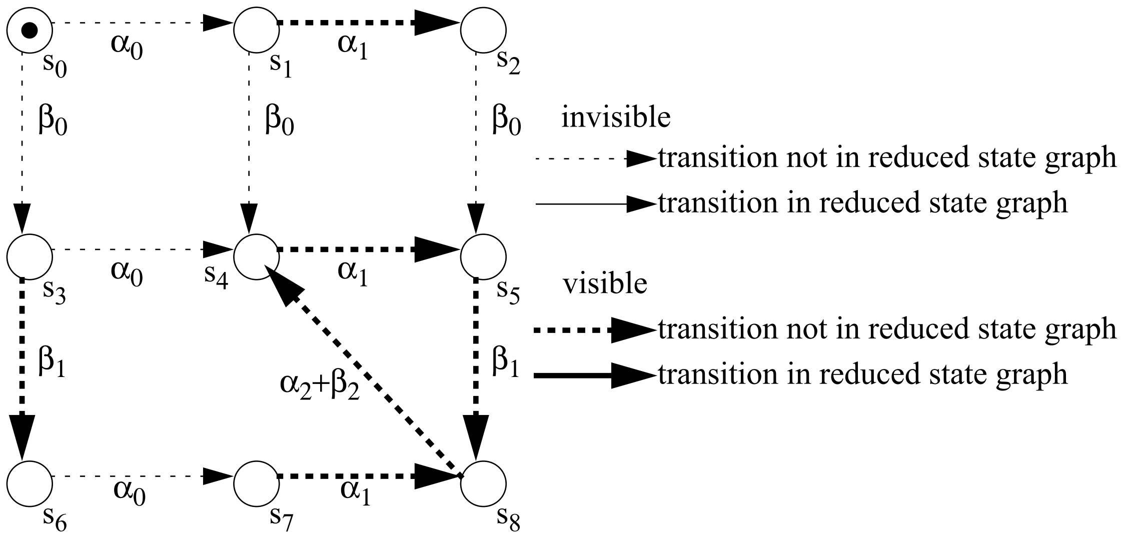

s0
α0
s1
α1
s4
β
β
transition not in reduced state graph
transition in reduced state graph
s2
α0
s3

The reduced state-graph has now been completed. In this example it is identical to that obtained in figure F.7. but this need not always be the case.

F.3.6. Differences between the methods

Let us consider the following slightly more complex system:

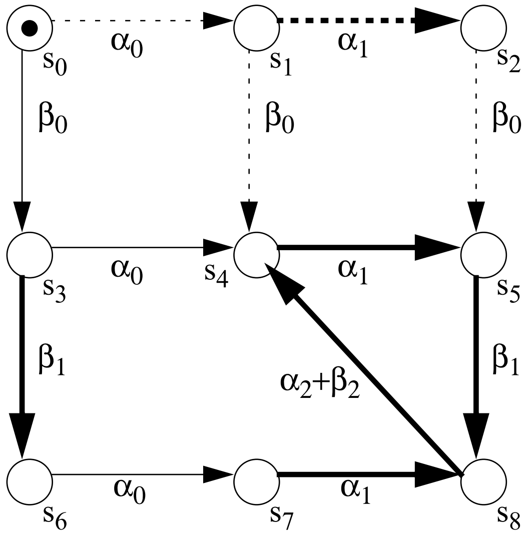
figure F.12. example system



First we construct the reduced state graph using the method of section F.3.5.

ample(s_0)={β_1} because α_0 process-disables β_1.
ample(s_3)={α_0, β_1} because α_0 process-disables β_1 and β_1 is visible.
ample(s_4)={α_1} because α_1 is the only element of *enabled*(s_4).
ample(s_5)={β_1} because β_1 is the only element of *enabled*(s_5).
ample(s_8)={$\alpha_2+\beta_2$} because $\alpha_2+\beta_2$ is the only element of *enabled*(s_8).
This transition closes a loop. In this loop, no transition is enabled that is not included in *ample*(s) for any state of the loop. Therefore no further action is required.
ample(s_6)={α_0} because α_0 is the only element of enabled(s_6).
ample(s_7)={α_1} because α_1 is the only element of enabled(s_7).

figure F.13. reduced state graph created using method of section F.3.5.

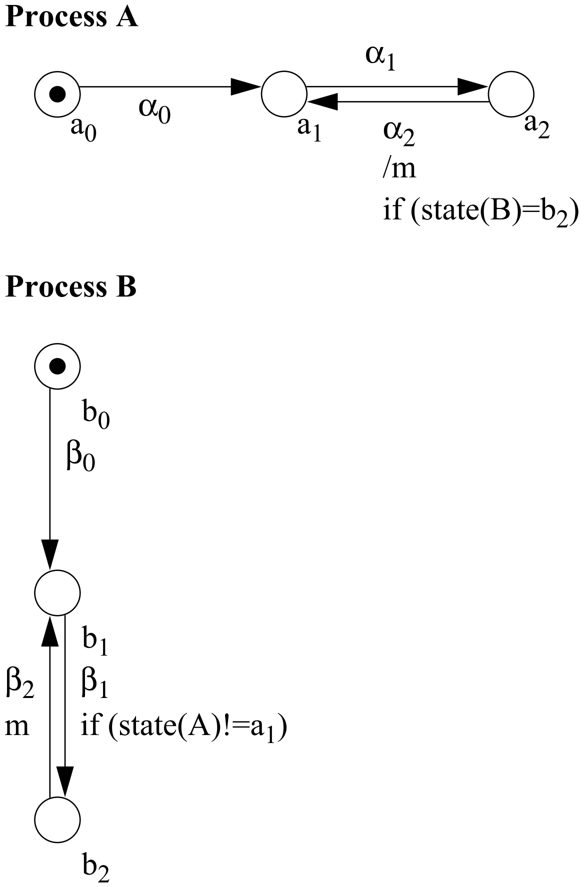


This reduction reduces graph complexity from 9 states and 13 transitions to 7 states and 9 transitions.

Now as a comparison, we apply the method of section F.3.1.

$\alpha_{0,}$ α_1 and α_2 are dependent because they are in the same process.
$\beta_{0,}$ β_1 and β_2 are dependent because they are in the same process.
β_1 is dependent on α_0, α_1 and α_2 because of inspections.
α_2+β_2 is dependent on $\alpha_{0,}$ α_1, β_0, and β_1.
By consequence all transitions are dependent. No reduction is possible.

figure F.14. component processes of system of figure F.13.



F.4. Discussion

F.4.1. Comparison of rules

The rules of section F.3.1. do not provide identical results to those of section F.3.5. In this section the differences will be discussed.

Rules **C0**, **C2** and **C3** from section F.3.1. are respected completely by the algorithm of section F.3.5.

rule **C0**: *ample*(s) = ∅ iff *enabled*(s) = ∅.

by rule R1, *ample*(s) cannot be empty unless *ample*(s) = *enabled*(s).

rule **C2**: If s is not fully expanded, then every α ∊ *ample*(s) is invisible.

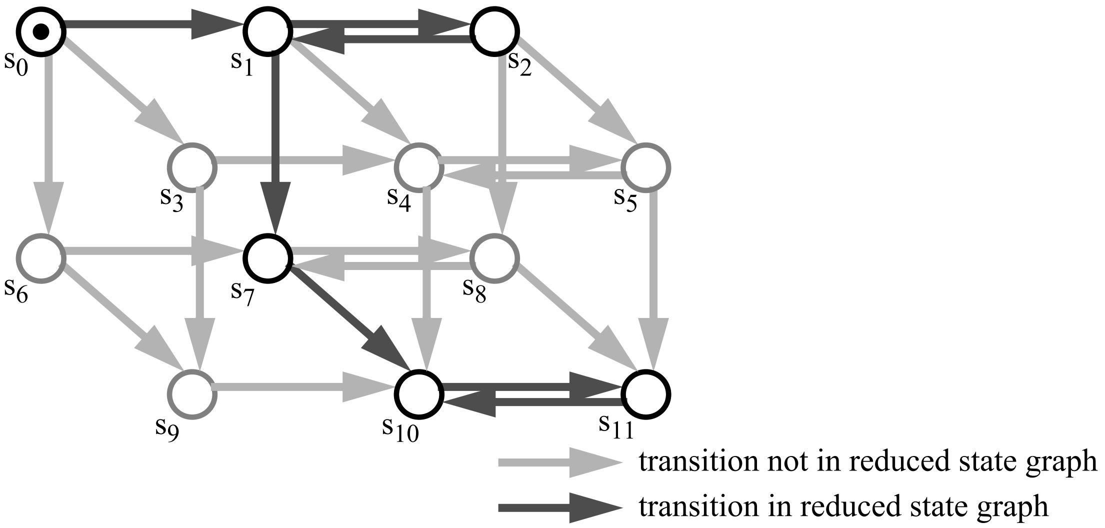
by rule R1, *ample*(s) cannot contain visible transitions unless *ample*(s) = *enabled*(s).

rule **C3**: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in *ample*(s) for some state of the cycle.

this is largely covered by rule R2. This rule is to prevent the reduced state graph from reaching a loop which it fails to leave (Model Checking by Clarke, Grumberg and Peled page 150 [18]). Rule 2 also prevents such a loop from forming, and also prevents such a loop from being recognised as fair when in fact it is not. Rule R2 offers a more powerful reduction than rule C3.

The reduced state graph of figure F.15.shows a reduced state graph which conforms to the rules of section F.3.5. but not those of F.3.1. This is because [s_1, s_2, s_1] forms a loop but there are transitions which are enabled on this loop but never included in the ample set. (Note that this is not the optimal reduced state graph. It is possible to construct one with two transitions less)

figure F.15. Example reduced state graph conform to rules of section F.3.5.


s_0
s_1
s_2
s_3
s_4
s_5
s_6
s_7
s_8
s_9
s_{10}
s_{11}
transition not in reduced state graph
transition in reduced state graph

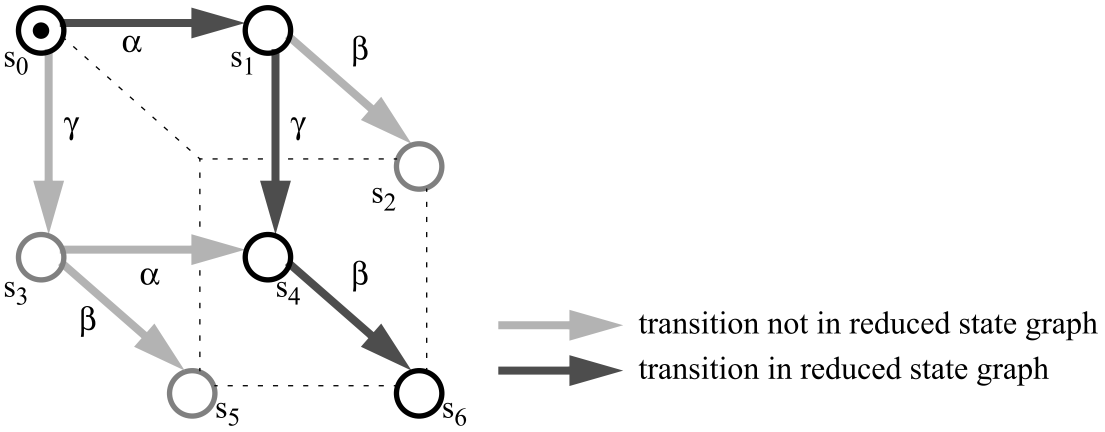
Finally we come to the trickiest of these rules:

rule **C1**: Along every path in the full state graph that starts in s, the following condition holds: a transition that is dependent on a transition in *ample*(s) cannot be executed without a transition in *ample*(s) occurring first.

This rule can be violated by the rules of section F.3.5. This is illustrated in the following example. The reader not interested in the example can proceed to section F.4.2.

The state graph of figure F.16. is considered. α, β and γ are transitions of three separate but dependent processes. The 3-dimensional state space is shown.

figure F.16. Example reduced state graph conform to rules of section F.3.5.


α
s_0
s_1
β
γ
γ
s_2
s_3
α
s_4
β
β
s_5
s_6
transition not in reduced state graph
transition in reduced state graph

By the rules of section F.3.5.

ample(s_0) = {α} because α does not process-disable β or γ.
ample(s_1) = {γ} because γ does not process-disable α or β.
ample(s_4) = {β} by default.

By the rules of section F.3.1. this reduction is incorrect:

α, β and γ are all dependent. Therefore the path [s_0, s_3, s_5] violates rule 3. Transition β and transition α are dependent and $\alpha \in$ *ample*(s_0), yet no element of *ample*(s_0) is executed on [s_0, s_3, s_5].

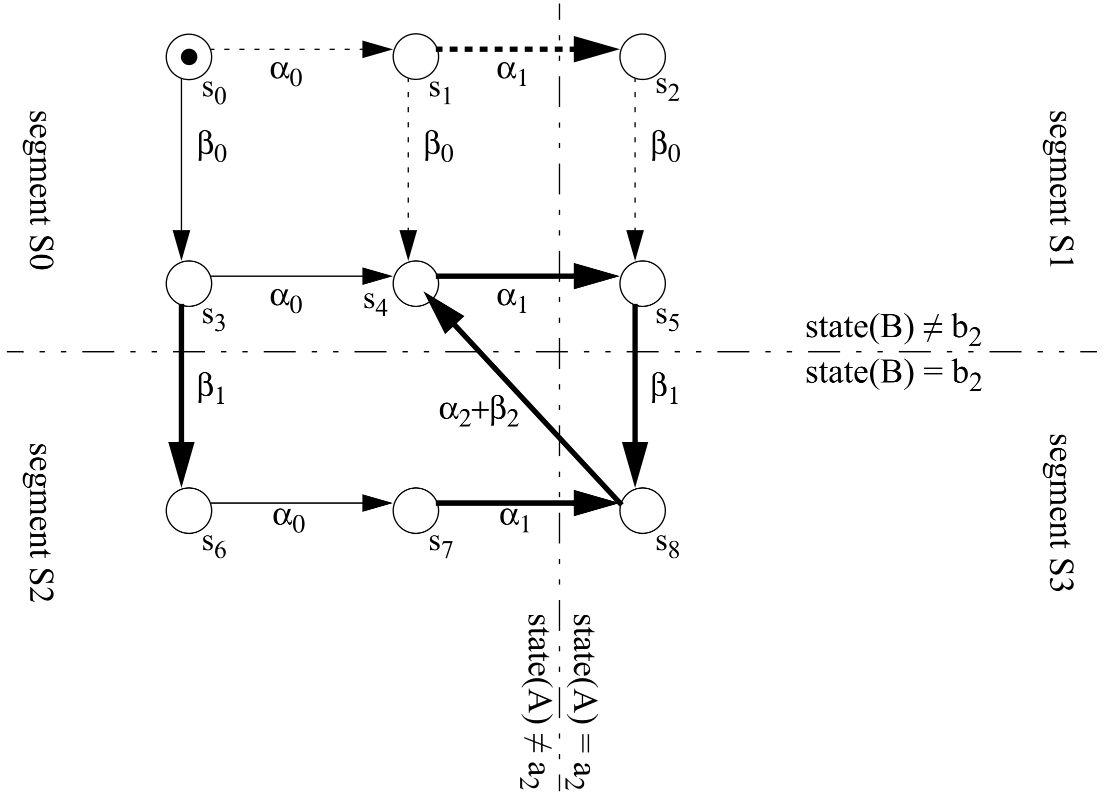
Clearly, the two methods are not equivalent. We must therefore prove the correctness of the rules of section F.3.5.

F.4.2. Discussion of visibility of transitions

Visible transitions are transitions for which the pre and post-states respond differently to propositions. The pre and post-states of such transitions are thus in different cylinder sets.

In figure F.17. the state graph of figure F.13. is repeated but with cylinder set divisions added.

figure F.17. state diagram with cylinder set divisions added



We can thus distinguish four segments marked S0 ... S4.

Note that transitions crossing from one segment into another are visible. The others are invisible.

In the full state graph, the sequences of segments visited by paths are:

a. S0, S1, S3, S0, S1, S3, S0, S1, S3...
b S0, S2, S3, S0, S1, S3, S0, S1, S3...

In the reduced state graph, both sequences are represented.

Properties such as *reacheable*(S3, S0) and *eventually*(S1, S2) can be checked in either graph and the results are the same.

As model checking statements are based on the sequences of segments visited, any state reduction is correct if all segment sequences are conserved. To demonstrate that a state reduction mechanism is correct, it suffices to prove that any segment sequence of the full state graph also exists in the reduced state graph.

F.5. Notes on discussion

These notes are intended as a background to the discussion of section 6.7.

F.5.1. State space reduction

In section 6.7.1. the complexity of full and reduced state graphs are compared. For a state machine with minimum dependency between processes, the following are stated:

$s_f \approx (s_p)^n$
$r_f \approx n \cdot r_p \cdot (s_p)^{n-1}$

If two processes are combined which have s_0, s_1, r_0 and r_1 states and transitions respectively, for the combined process:

$s_f = s_0 \cdot s_1$
$r_f = s_0 \cdot r_1 + s_1 \cdot r_0$

We assume for simplification that every process has exactly the same number of states and transitions.

The first formula $s_f \approx (s_p)^n$ is obvious because the number of states of the full state graph grows exponentially with the number of processes.

The formula $r_f \approx n \cdot r_p \cdot (s_p)^{n-1}$ is derived as follows:

For $n = 1$, $r_f = r_p$

For $n = 2$, $r_f = r_p \cdot s_p + r_p \cdot s_p = 2 \cdot r_p \cdot s_p$

For $n = 3$, $r_f = (2 \cdot r_p \cdot s_p) \cdot s_p + r_p \cdot (s_p)^2 = 3 \cdot r_p \cdot (s_p)^2$

So the formula holds for $n = 1, 2$ or 3.

For any $n>1$, $r_f = ((n-1) \cdot r_p \cdot (s_p)^{n-2}) \cdot s_p + r_p \cdot (s_p)^{n-1} = n \cdot r_p \cdot (s_p)^{n-1}$

In the same section, the following are stated:

$s_r \approx n \cdot s_p$
$r_r \approx n \cdot r_p$

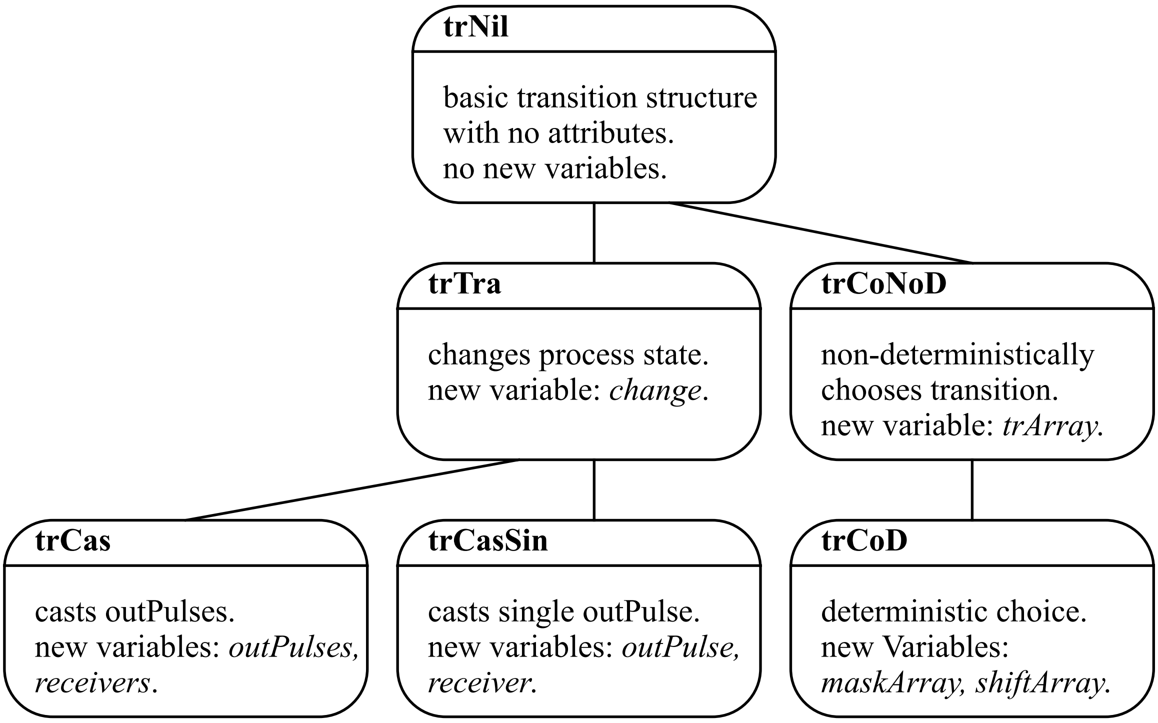
By the reduction rules and their implementation (section 6.6.2. page 109), transitions are preferred which close loops or otherwise do not already occur in the search path. This favours a search path containing every transition exactly once. The post-state of the last transition is either a dead-end or an earlier state of the path. The number of transitions of the cluster is thus sum of the numbers of transitions of the processes, hence $r_r = n \cdot r_p$ if all processes have exactly r_p transitions. The number of states is not directly derived from the number of transitions as some states could indeed occur several times in the search path. But each process state must be a component of at least one cluster state. So the reduced state graph has a state for every process state minus the first which is already contained in the previous cluster state. Thus $s_r = n \cdot s_p - n + 1$. This is approximated to $s_r \approx n \cdot s_p$.

Appendix G: Data structures

G.1. Transitions

As shown in section 7.2.2. (page 121), there are many types of transitions. For each, a separate class exists. The hierarchy of these classes is shown in figure G.1.

figure G.1. Class hierarchy of transitions


trNil
basic transition structure with no attributes. no new variables.
trTra
changes process state. new variable: *change*.
trCoNoD
non-deterministically chooses transition. new variable: *trArray.*
trCas
casts outPulses. new variables: *outPulses, receivers*.
trCasSin
casts single outPulse. new variables: *outPulse, receiver.*
trCoD
deterministic choice. new Variables: *maskArray, shiftArray.*

Explanation of variables:

change: an integer which is added to the state vector of the pre-state to provide the value of the post-state.

maskArray: array of integers with mask values. See section 7.2.4. (page 124).

outPulse: integer which is sent to the receiving process to denote *triggerOffset* of the trigger being sent.

outPulses: array of integers *outPulse* when several triggers cast.

receiver: process receiving *outPulse*.

receivers: array of processes *receiver* when several triggers cast.

shiftArray: array of integers with shift values. See section 7.2.4. (page 124).

trArray: array of transitions of types *trNil* and subclasses.

Bibliography and references

[1] A Case Study in Model Checking Software Systems, J. M. Wing, M. Vazari-Farahani, Science of Computer Programming 28, pp273-299, Elsevier Science B.V, 1997.

[2] Abstract Interpretation of Reactive Systems, D. Dams, R. Gerth, O. Grumberg, ACM Transactions on Programming Languages and Systems, Vol 19, No 2, pp253-291, March 1997.

[3] An automata-theoretic approach to automatic program verification, M. Y. Vardi, P. Wolper, Proc. First IEEE Symp. on Logic in Computer Science, pp. 322-331, 1986.

[4] An Improvement in Formal Verification, G. Holzmann, D. Peled, Proc. FORTE 1994 Conference, Bern 1994.

[5] Analysis of Real-Time Systems Using Symbolic Techniques, S Campos, E. Clarke, M. Minea, Formal Methods for Real-Time Computing, C. Heitmeyer, D. Mandrioli (editors), pp217-235, John Wiley and Sons 1996.

[6] Automatic Analysis of Architectural Style, D. Jackson, School of Computer Science, Carbegie Mellon University, downloadable from website: http://www.cs.cmu.edu/~dnj.

[7] Branching time temporal logic and the Design of Correct Concurrent Programs, E. A. Emerson, PhD thesis, Harvard University 1981.

[8] CIP Grundkurs: Konstruktion und Implementation von CIP-Modellen, H. Fierz, CIP System AG 1999.

[9] CIP Tool, (documentation, tutorial and a trial version can be downloaded from the CIP Tool website: http://www.ciptool.ch).

[10] CIP Tool - Model-based Construction of Embedded Real-Time Components, H. Fierz, downloadable from CIP Tool website [9].

[11] Design and synthesis of synchronisation skeletons using branching time temporal logic, E. M. Clarke, E. A. Emerson, in Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, MNCS 131, Springer 1981.

[12] Eingebettete Systeme als Architektur Mechanistischer Modelle: Architekturorientierte Modellkonstruktion mit CIP Tool, H. Fierz, downloadable from CIP Tool website [9].

[13] Hardware specification with temporal logic: An example, G. von Bochmann, IEEE Transactions on Computers C-31.

[14] Improving Efficiency of Symbolic Model Checking for State-Based System Requirements, W. Chan, R. J. Anderson, P. Beame, D. Notkin, Software Engineering Notes, ACM Press, Volume 23, No 2, pp102-112, March 1998.

[15] Introduction to Modal Logic, G. E. Hughes, M. J. Creswell, Methuen 1977.

[16] LAR: A logic of algorithmic reasoning, F. Kröger, Acta Informatica 1975.

[17] Managing the Development of Large Software Systems, W. W. Royce, Proceedings of IEEE WESCON, August 1970.

[18] Model Checking, E. M. Clarke (Jr), O. Grumberg, D. A. Peled, MIT Press 1999.

[19] Model Checking and Abstraction, E. M. Clarke, O. Grumberg, D. E. Long, Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, pp343-354, 1992.

[20] Model Checking Large Software Specifications, R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, J. D. Reese, Software Engineering Notes, ACM Press, Volume 21, No 6, pp156-166, November 1996.

[21] Modelling Reactive Systems with Statecharts: The STATEMATE Approach, D. Harel and M. Politi, McGraw-Hill, 1998.

[22] Naming and Necessity, Saul Kripke, Analytic Philosophy 1972.

[23] On the temporal analysis of fairness, D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, in Proceedings of the 7th Symposium on Principles of Programming Languages, pp. 163-173, ACM 1980.

[24] Optimizing Symbolic Model Checking for Statecharts, W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, W. E. Warner, IEEE Transactions on Software Engineering, February 2001, Vol. 27, No. 2, pp. 170-190.

[25] “Parmenides” in Plato: Complete Works, J. M. Cooper (ed), Hackett Publishing Co 1997.

[26] Process and Reality, A. N. Whitehead, The MacMillan Co New York 1929.

[27] Program proving as hand simulation with a little induction, R. M. Burstall, in IFIP Congress 74, pp 308-312, North Holland 1974.

[28] Separate connection and functionality is the pivot in embedded system design, O. Trutmann, Diss., Technische Wissenschaften ETH Zürich, Nr. 13891, 2000.

[29] SMV (model checker and documentation downloadable from Carnegie Mellon university: www-2.cs.cmu.edu/~modelcheck/smv.html). Many related resources can be found by searching for "SMV model checker" using search engine.

[30] Software Reliability Methods, D. A. Peled, Springer-Verlag 2001.

[31] Specification and verification of concurrent systems in CESAR, J. P. Quielle, J. Sifakis, in Proceedings of the 5th international Symposium on Programming, pp. 337-350, 1981.

[32] SPIN (model checker and documentation downloadable from SPIN website: www.spinroot.com/spin/whatisspin.html). Many related resources can be found by searching for "SPIN model checker" using search engine.

[33] SPIN Beginners' Tutorial, T. C. Ruys, SPIN Workshop Grenoble 2002, can be downloaded from SPIN website [32].

[34] STATEMATE: A working Environment for the Development of Complex Reactive Systems, D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, IEEE Transactions on Software Engineering, Vol. 16, No. 14, April 1990.

[35] Symbolic Model Checking: An Approach to the State Explosion Problem, K. L. McMillan, Kluwer Academic 1993.

[36] Temporal specifications of self-timed systems, Y. Malachi, S. S. Owicki, in VLSI Systems and Computations, Computer Science Press 1981.

[37] The CIP Method: Component- and Model-Based Construction of Embedded Systems, H. Fierz, ESEC 1999, downloadable from CIP Tool website [9].

[38] The Model Checker SPIN, G. Holzmann, IEEE Trans. on Software Engineering, Vol. 23, No. 5, pp. 279-295, 1997.

[39] The SPIN Model Checker, Primer and Reference Manual, Addison Wesley 2003

[40] The temporal logic of programs, A. Pnueli, in 18th IEEE Symposium on Foundation of Computer Science, pp 46-57, IEEE Computer Society Press 1977.

[41] UML Statecharts, B. P. Douglass, http://www.embedded.com/1999/9901/9901feat1.htm. Many related resources can be found by searching for “UML” or “Statecharts” using search engine.

[42] Visual Verification of Safety and Liveness, A. Valmari, M. Setälä, Proc. FME (Formal Methods Europe) ‘96, LNCS 1051, pp228-247, 1996.

[43] Zeno’s Paradoxes, in the Stanford Encyclopedia of Philosophy, E. N. Zalta (ed), Stanford 2002 and online: www.plato.stanford.edu.

[44] CIP system designed by Zühlke Engineering.

Appendix R: Quick Reference

R.1. Symbols

Note that only symbols for sets are written **bold**.

symbol	short description	see section	page
Π	set of processes	5.3.3.	83
A	set of actions	3.2.2.	36
AP	set of atomic propositions	3.4.2.	42
CTL	Computation Tree Logic	4.3.3.	53
CTL*	Computation Tree Logic (general form)	4.3.2.	52
D	set of dead-end states: $\mathbf{D} :\subseteq \mathbf{S} \mid \forall s \in \mathbf{D}: \mathit{suc}(s) = \emptyset$	3.2.4.	37
E	set of events	3.2.2.	36
F	set of fairness constraints	3.7.2.	47
false	boolean symbol		
FLP	set of fair loops	3.7.4.	48
FP	set of fair full paths	3.7.5.	49
FP*	set of fair paths	3.7.5.	49
FP#	set of fair continuing paths	3.7.5.	49
GP	set of properties: $\mathbf{GP} := \{gp \mid gp: \rightarrow \{\text{true}, \text{false}\}\}$	3.4.5.	44
$\mathbf{In}_\pi$	set of inPulses of process π	5.3.3.	83
K	Kripke structure	3.4.3.	42
L	Labelling function: $\mathbf{L} : \mathbf{S} \rightarrow 2^{\mathbf{AP}}$	3.4.3.	42
LP	set of loops	3.3.8.	41
LTL	Linear Time Logic	4.3.4.	53
M	system	3.2.2.	36
$\mathbb{N}$	set of natural numbers (includes zero)		
$\mathbf{Out}_\pi$	set of outPulses of process π	5.3.3.	83
P	set of full paths	3.3.6.	41
P*	set of paths	3.3.1.	39
P#	set of continuing paths	3.3.5.	41
PP	set of path propositions	3.4.4.	43
R	transition relation: $\mathbf{R}: \subseteq \mathbf{E} \times \mathbf{S} \times \mathbf{S} \times \mathbf{A}$	3.2.2.	36
R’	(in Kripke structure) transition relation: $\mathbf{R'}: \subseteq \mathbf{S} \times \mathbf{S}$	3.4.3.	42
R’	(in fair Kripke structure) transition relation: $\mathbf{R'}: \subseteq \mathbf{S} \times \mathbf{S} \times \mathbf{F}$	3.7.8.	50
$\mathbf{R}_\pi$	transition relation: $\mathbf{R}_\pi: \subseteq (\mathbf{E} \cup \mathbf{In}_\pi) \times \mathbf{S}_\pi \times \mathbf{S}_\pi \times \mathbf{A} \times \mathbf{Out}_\pi$	5.3.3.	83
S	set of states	3.2.2.	36
$\mathbf{S}_\pi$	set of process states of process π	5.3.3.	83

s_{init}	initial state	3.2.2.	36
s_{init_π}	initial state of process π	5.3.3.	83
SR	set of reacheable states	3.3.7.	41
true	boolean symbol		

R.2. Functions and relations

Arguments shown in square brackets [] are optional.

name	**mapping**	**short description**	**see section**	**page**
action	$\mathbf{R} \rightarrow \mathbf{A}$	action of transition	3.2.3.	36
ample	$\mathbf{S} \rightarrow 2^{\mathbf{R}}$	subset of *enabled*(s) used in partial order reduction	6.4.1.	104
dependent	$\mathbf{R} \rightarrow 2^{\mathbf{R}}$	dependent transitions	3.2.5.	37
deterministic	$\mathbf{R} \rightarrow$ boolean	is transition deterministic?	3.2.7.	39
disabled	$\mathbf{S} \rightarrow 2^{\mathbf{R}}$	transitions disabled in state	3.2.4.	37
disables	$\mathbf{R} \rightarrow 2^{\mathbf{R}}$	transitions disabled by transition	3.2.5.	37
dpe	$\mathbf{S} \times \mathbf{T} \rightarrow \mathbb{N}$	degree of process-enabledness	6.5.2.	106
enabled	$\mathbf{S} \rightarrow 2^{\mathbf{R}}$	transitions enabled in state	3.2.4.	37
enables	$\mathbf{R} \rightarrow 2^{\mathbf{R}}$	transitions enabled by transition	3.2.5.	37
eventually	$\mathbf{CP}$ [$\times$ $\mathbf{CP}$] $\rightarrow$ boolean	a state fulfilling proposition will eventually be reached [from all states fulfilling second proposition] ?	3.5.5.	46
		(with fairness)	(3.7.8.	50)
fairness	$\mathbf{R} \rightarrow 2^{\mathbf{F}}$	fairness constraints	3.7.2.	47
fairpaths	$2^{\mathbf{P^*}} \rightarrow 2^{\mathbf{P^*}}$	extracts paths which are fair	3.7.3.	48
fairTrans	$\mathbf{P} \times \mathbb{N} \times \mathbf{F} \rightarrow$ boolean	fair transition at position?	B.1.1.	149
first	$\mathbf{P^*} \rightarrow \mathbf{S} + \mathrm{nil}$	first state of path	3.3.2.	39
independent	$\mathbf{R} \rightarrow 2^{\mathbf{R}}$	transitions independent of transition	3.2.5.	37
invariant	$\mathbf{CP} \rightarrow$ boolean	proposition is invariant?	3.5.3.	45
last	$\mathbf{P^*} \rightarrow \mathbf{S} + \mathrm{nil}$	last state of path	3.3.2.	39
length	$\mathbf{P^*} \rightarrow \mathbb{N}$	length of path	3.3.2.	39
next	$\mathbf{CP} \times \mathbf{CP} \rightarrow$ boolean	proposition holds in next state	3.5.2.	45
occurrences	$(\mathbf{S} \cup \mathbf{R}) \times \mathbf{P^*} \rightarrow \mathbb{N}$	occurrences of object on path	3.3.2.	39
outPulse	$\mathbf{R}_\pi \rightarrow \mathbf{Out}_\pi$	outPulse of transition	5.3.4.	83

outPulses	$(\mathbf{In}_\pi \cup \mathbf{E}) \times \prod \rightarrow 2^{\mathbf{Out}\pi}$	outPulses triggered by inPulse or event for process	C.3.9.	172
post	$\mathbf{R} \rightarrow \mathbf{S}$	post-state of transition	3.2.3.	36
pre	$\mathbf{R} \rightarrow \mathbf{S}$	pre-state of transition	3.2.3.	36
pred	$\mathbf{S} \rightarrow 2^{\mathbf{S}}$	set of predecessor states	3.2.4.	37
preFair	$\mathbf{S} \times \mathbf{F} \rightarrow$ boolean	is pre-state of fair transition?	B.1.1.	149
processState	$\mathbf{S} \times \prod :\rightarrow \mathbf{S}_\pi$	process state for process and cluster state	5.3.5.	83
propCount	$\mathbf{CP} \times \mathbf{P} \rightarrow \mathbb{N}$	occurrences of proposition on path	3.4.4.	43
reacheable	$\mathbf{CP}$ [$\times$ $\mathbf{CP}$] $\rightarrow$ boolean	a state fulfilling proposition is reacheable [from all states fulfilling second proposition] ?	3.5.4.	45
subpath	$\mathbf{P}^* \rightarrow 2^{\mathbf{P}^*}$	subpaths	3.3.3.	40
suc	$\mathbf{S} \rightarrow 2^{\mathbf{S}}$	set of successor states	3.2.4.	37
sucp	-	(see DFS algorithm)	D.2.1.	179
trans	$\mathbf{P}^* :\rightarrow 2^{\mathbf{R}}$	all transitions on path	3.3.4.	40
trig	$\mathbf{R} \rightarrow \mathbf{E}$	trigger of transition	3.2.3.	36
visited	$\mathbf{P}^* \rightarrow 2^{\mathbf{S}}$	states visited	3.3.2.	39

Index

Curriculum Vitae

Personal information

Name:	Moglestue
Forenames:	Andreas Hubert
Address:	Bucheggstrasse 103 / 13 8057 Zürich Switzerland
Date of birth:	06/09/1971
Place of birth:	Reading, United Kingdom
Nationality	British

Education

until 1988:	Primary and secondary schools in Reading (UK)
1988-1991:	*Deutsch Französisches Gymnasium* in Freiburg im Breisgau (Germany)
1991-1996:	Study of Electrical Engineering at Swiss Federal Institute of Technology (ETH) Zürich
First semester project:	Programming of a DSP for controlling a high-speed drive (*Professur für Leistungselektronik und Messtechnik*, 1994-5)
Second semester project:	Design and construction of an energy saving switch for remote operation via ethernet (*Institut für Elektrische Maschinen*, 1995)
Diploma project:	Optimisation of switching patterns for a 1.5kHz voltage source inverter (*Professur für Leistungselektronik und Messtechnik*, 1995-6)
Doctorate thesis:	Doctorate at ETH Zürich (CIP Model Checking, *Institut für Technische Informatik und Kommunikationsnetze*, 1999-2004)

Professional experience

1996-1999:	ABB Semiconductors AG, Quality and Reliability department.
1999-2003:	Employee of Swiss Federal Institute of Technology (ETH), *Institut für Technische Informatik und Kommunikationsnetze*