DEONTIC LOGICS FOR SPECIFICATION OF FAULT-TOLERANCE

DEONTIC ACTION LOGICS

FOR

SPECIFICATION AND ANALYSIS OF FAULT-TOLERANCE

By

PABLO F. CASTRO, Lic.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree

Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2009)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE:	Deontic Action Logics for Specification and Analysis of Fault-Tolerance
AUTHOR:	Pablo F. Castro, Lic. (Univ.Nac.Rio Cuarto, Argentina)
SUPERVISOR:	Dr. T. S. E. Maibaum
NUMBER OF PAGES:	xii, 246

Abstract

In this thesis we develop a mathematical framework to express and reason about properties of fault-tolerant computing systems. The main idea behind this mathematical framework is to use axiomatic theories to specify systems. The standard logical operators allow us to describe the basic behavior of the system, while we use deontic predicates on actions to express prescriptions about the system's behavior. Deontic logics have proved to be useful for reasoning about legal and moral systems, where the situation is more or less similar to fault-tolerance: there exists a set of rules that states what the normal behaviours or scenarios are. Violations arise when these rules are not followed and, as a consequence, some actions must be performed to return to a normal or desirable state. We develop our own deontic logic, keeping in mind that we want to use it for specifying fault-tolerant systems. We investigate the properties of this logic, commenting on those that are relevant to the use of the logic in practice. We provide two different deductive systems; one of them is a standard (Hilbert style) deductive system, while the other one is a tableaux system, which can be applied automatically to prove properties of specifications.

In any specification language, it is important to have at hand mechanisms which enable designers to modularize the system description; we investigate how to apply these mechanisms to the logics proposed in this thesis, and, in particular, we focus on how the modularization of specifications affects the local prescriptions of a module (or component). We study the problems that arise from the interaction between components. We show that, in some cases, we can guarantee that the locality of violations in a particular component is preserved. Some examples are provided throughout this thesis to illustrate how the ideas described below can be applied in practice .

Acknowledgements

I would like to thank Prof. Tom Maibaum for being my supervisor during these four years and for his guidance throughout this time. I also would like to thank Gabriel Baum, Javier Blanco and Nazareno Aguirre for their help before coming to Canada. All this work would not be possible without the support and understanding of Valeria and my parents.

Contents

List of Figures

Chapter 1

Introduction

The aim of this thesis is to study the theory and application of different mathematical formalisms to the specification of fault-tolerant systems. In particular, we focus on using logical formalisms arising from the study of moral and ethical norms (called *deontic logics*). These logics have been widely used by philosophers and lawyers to investigate the reasoning that is used when statements with prescriptions or norms are involved. The analogy with fault-tolerance is more or less straightforward: faults in software may produce incorrect behaviour (i.e., violations to some requirements), and therefore, corrective or recovery actions are needed. This is similar to the situation in legal and normative systems where persons or entities may infringe some laws, and therefore, some actions must be undertaken in consequence (e.g., this person must pay a fine). However, to use these mathematical frameworks for reasoning about computer systems, we must ensure that they allow us to express the basic properties of fault-tolerant software. Moreover, the techniques used in formal languages of computer science must be supported for these logics to make it possible to apply them in practice. In this thesis we develop a mathematical, *deontic* framework, analyzing the characteristics that are needed to specify fault-tolerant systems and, in consequence, adding these features to our framework. Towards this goal, we introduce some typical examples that illustrate the usefulness of the formalisms discussed below.

Logics, and mathematical frameworks in general, have been shown to be useful for the design, specification and for the verification of systems. In particular, mathematical reasoning is essential in the development of critical systems where faults or errors may cause financial loss, or worse, unexpected behaviour can result in the loss of human life (e.g., airplane software). Different logical languages and mathematical theories have been used in the last few decades for developing software and systems in general, which are free of faults or errors (e.g., Hoare logic [Hoa69]). However, when

systems become more complex, the task of proving the absence of faults becomes harder and more expensive. As a result, *fault-tolerance* techniques are a good alternative; they allow software to continue working (perhaps in a degraded state) in the presence of errors or faults. Roughly speaking, fault-tolerant systems are those which have the possibility of overcoming, more or less successfully, unexpected behaviour during their execution.

There exist several techniques to implement fault-tolerance (e.g., code replication, voting algorithms and exception mechanisms, see [LA90]), but these are mainly for the implementation phase and not for the design phase. In the few last years, several researchers from the formal methods community have proposed using fault-tolerance techniques together with formal methods to provide more reliable software. Related with this, recently, some researchers ([KQM91, CJ96, MM06, Mai93, LS04, FM91a]) have pointed out that *deontic logic*, a variation of logic advocated for the study of norms, is useful for reasoning about fault-tolerant systems. An interesting feature of this logic is that the notions of permission and obligation are naturally embedded in the formalism. Through this thesis, we take both ideas and develop different deontic formalisms which we assert can be used to specify fault-tolerant software. We also add the dimension of time in our logics; temporal logics have been shown to be very useful for the verification of systems, particularly for reactive systems and automatic verification of specifications (see [MP92] and [Eme90]).

Furthermore, the decomposition of systems into modules or components is fundamental to create products of good quality. In recent years, software systems have become so complex and large that trying to design them without good techniques of modularization is an almost impossible task. If we pursue the idea of specifying a system using formal methods, modularization techniques are necessary for several reasons: first, the formal specification must reflect the structure of the system which encodes several design decisions, and second, proving properties (and reasoning in general) about small specifications is simpler than reasoning about large specifications. However, how to produce components of specifications and how to compose several modules to obtain the final specification is not a trivial issue. Here we follow the main ideas introduced by Goguen and Burstall in [BG77], where techniques coming from *category theory* are used to put components together; we explain these ideas later on when the basic definitions of category theory are introduced. An important topic in this thesis (see chapter 7) is understanding how the deontic constructions used to specify norms fit into modularization techniques, and how the structure of the violations (i.e., those violations or errors arising during a possible execution of the system and the logical relationships between them) occurring in the different modules can be composed to approximate the structure of the violations occurring in the final specification. In the following sections we introduce in more detail fault-tolerant systems and, after this, we argue why deontic formalisms can help in specifying and

verifying fault-tolerant systems.

1.1 Fault-Tolerant Systems

The first definition of *fault-tolerant* system can be traced back to [Avi67], where it is stated: "*we say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults.*" Since then, computing systems have changed dramatically and so have the faults that programs must tolerate. In particular, the increasing complexity of computing systems implies that *design faults* in systems are more common (in [TP00] it is argued that all the software faults are because of design faults). Moreover, today most computing systems interact by means of the internet or large networks, and therefore they are exposed to faults from other software, faults during communication with other systems or faults arising from the interaction with users. Furthermore, systems which interact with the environment are exposed to unexpected environmental behaviour. Following the terminology in the fault-tolerance literature, we can distinguish between *error* and *fault.* A *fault* is a hardware defect or a software mistake (i.e., a *bug*). An *error* is an undesired state of a system which is the consequence of a fault. As explained in [TP00], developers have four ways to deal with faults:

- Fault prevention,
- Fault removal,
- Faul tolerance,
- Input sequence workarounds (i.e., the users have to deal with the errors).

Fault prevention is the most common way to minimize faults; well-known techniques, tools and methods from software engineering are used to produce software of better quality. In particular, formal methods provide an appealing approach to produce software without faults; theories and languages arising from mathematics are used to prove mathematically that software is free of faults.

Fault removal is usually achieved by means of testing [TP00] (which can be done in several different ways) to discover faults during the development process and therefore eliminate them from the design or the implementation. Using formal methods, properties of the specification can be checked to know if the specification is sound with respect to the requirements. In a later step, *model checking* can be used to know if a

possible implmentation satisfies the specification (i.e., it is a correct implementation of the specification).

However, as already remarked upon by Dijkstra [Dij72], testing can show the existence of faults, but not their absence. On the other hand, formal methods are hard to apply to large and complex systems. Moreover, although a system is free of faults, it is exposed to faults from the operating system, from hardware or from any unexpected behaviour of the environment. This implies that in critical systems the capability to continue working in an acceptable way in spite of the presence of faults is needed, and therefore fault-tolerance techniques become necessary. There are many fault-tolerance techniques that can be used to improve the capability of a system to deal with faults. It is common to divide them between *fault-tolerance hardware techniques* and *fault-tolerance software techniques* depending on whether they are used at the hardware level or the software level. We review them briefly.

Hardware fault-tolerance is achieved, commonly, by means of hardware redundancy (i.e., additional hardware is used). Hardware redundancy techniques can be classified into *static*, *dynamic* or *hybrid* redundancy. Static techniques do not detect errors or change the configuration of the hardware dynamically. A standard example of these techniques is when several modules (e.g., processors) are used instead of only one and some mechanism is used to decide which is the correct output of the hardware. Usually, a voting strategy is used; for example, if two of three modules return the same value, then this value will be the final result. On the other hand, dynamic techniques react to errors. For example, in the technique called *duplication with comparison*, two modules perform the same action; if the output of the two disagree then some corrective actions are executed to take the system to an acceptable state. This approach can be extended (a so called *dual-dual configuration*) using many pairs, one of them being the principal one and when an error is found, this piece of hardware is put in a maintenance state, and therefore another pair is used. Hybrid approaches utilize both static and dynamic techniques. Hardware fault-tolerance has been investigated deeply and applied in practice (in aeronautics, nuclear applications, aerospace systems and health science applied systems). A good reference is [SS98].

Software fault-tolerance techniques are used to build software capable of tolerating faults. Following [TP00], they can be divided in two classes:

- *Single version fault-tolerance:* in this case individual pieces of software are designed to support faults.

- *Multiversion fault-tolerance:* in this case several versions of a piece of software are used to prevent system failures.

There are several techniques that use a single-version of some software. As explained in [TP00], decomposing software into several components or pieces which are independent to some degree is important to avoid the propagation of errors from one part of the system to other parts, or possibly all of the system. Here, it is important to use techniques that restrict the propagation of errors from one component to others when designing the architecture of the software. For example, in *system closure* [Den76], no action is allowed unless it is explicitly permitted (and here we can see the shadow of deontic notions which we will use later on). The main goal here is to reduce the possibility that the errors propagate through the system; when some error occurs, some valid actions can be disabled to avoid error propagation. Another approach uses *atomic actions* [AL81]; atomic actions are those in which some components interact, but during the execution of these actions there is no flow of information between these components and the rest of the system. Atomic actions allow us to isolate the errors to the participating components, and, moreover, if an error is detected, a *rollback* to a consistent state before the execution of the action is possible. (This is usually done with *transactions* in databases.) Another often used technique for fault-tolerance in software is the use of *exception mechanisms* in programming languages, where interruption mechanisms are used to stop the normal execution of code when an error is detected, and then an exception handler is executed to try to return to a consistent state. Many programming languages provide mechanisms for exception handling. Well-known examples are: *Java*, *C++* and *Eiffel*. Other techniques used are *checkpoints and recovery* mechanisms. These can be static or dynamic. In static checkpoints, the execution of the software is checked at some points and, in case of an error, the system is returned to a state free of errors.

In *multiversion software fault-tolerance* techniques, several versions of the same component (or module) are used to improve the fault-tolerant features of software. Some examples of these techniques are the following. *N-version programming* [Avi95] is a technique where several versions of some software are produced to satisfy the same requirements, and then the output of the system is decided taking into account all the outputs (using some decision technique like, for example, voting). In this case, it is required that there is total independence in the development of each version, i.e., different programming languages, design languages and indeed different teams of developers must be used to produce each version. This, of course, implies that the cost of the software increases seriously. There exist several variations of this approach, basically changing the way in which the output is selected, and also combining this technique with single-version methods. See [TP00] for a more exhaustive list of methods that have been used in practice.

One important aspect of fault-tolerance is *error detection*, i.e., the mechanisms used to detect if an error has occurred during the execution of the system. There exist several techniques for this: *replication checks* (several versions of a component

can be used to check if their outputs coincide), *timing checks* (when we have timing constraints, for example in real time systems, we can use clocks or time deadlines to check if a specific task has been done), *reversal checks* (use the output to calculate the corresponding input and then compare it with the actual input), *coding checks* (codes are used to analyze if some piece of information is valid, for example, when downloading a file from the internet, a checksum is usually also provided and this code can be used to detect if the file obtained is the original one) and *structural checks* (properties of data types are used to detect errors).

1.2 Formal Methods and Fault-Tolerance

In the last few decades, significant effort has been made to use formal methods to specify and verify fault-tolerant systems. In this section we review briefly some of these approaches.

Program Verification and Fault-Tolerance: Many of these works have applied extensions of Hoare logic to verify fault-tolerant programs. For example, in [Cri85], an approach to the design and verification of programs that are tolerant to faults is proposed. Faults are formalized as operations which are performed at random time intervals. The approach is presented using an example of a stable storage device; the main idea is to extend Floyd/Hoare logic with rules which enable reasoning about crashes and failures in storage disks. The logical rules suggested in this work are dependent on the example used (for example a *decay* operation which produces faults on the disks is axiomatized and used in the logical calculus). In [SS83], a formal methodology to design computing systems is presented. The approach is based on the concept of *fail-stop processors* (processors that halt in the presence of an internal failure). An axiomatic verification technique is described to verify programs running on these kinds of processors. In [Aro92], fault-tolerant programs are characterized by means of predicates. Invariants are used to describe those states which are free of errors. The author formally defines fault-tolerance using the concepts of *closure* and *convergence*. *Closure* is the property of a system of staying in a certain set of states, indeed during the occurrence of faults. On the other hand, *convergence* is the property that, if faults stop occurring, then the program reaches a state where the invariant describing the correct states is true. Faults are defined as unexpected actions of programs written in a concurrent programming language.

Another line of research is proposed in [LMJ93], where a formal framework is provided to reason about concurrent programs (defined in a language similar to UNITY [CM88]). Several predicates are introduced to formalize fault-tolerance

features of algorithms. Some of them are: *p degrades to q* (the predicate *p* stops holding at some instant where a weaker predicate *q* starts holding), *p upgrades to q* (at some point where *p* is true, there is a sequence of computations that makes a stronger predicate *q* true). Many other similar predicates are proposed, and these predicates are used to classify the states of a program into *correct* (there is no fault), *safe* (states where, if no more faults occur, the program will reach a correct state) and *recoverable* (states where, though there are errors, we can execute a recovery action to reach a safe or correct state). This classification of states and the logic introduced are used to reason about two examples: an algorithm to calculate the invariant of a Markov chain and an algorithm to find a solution of a set of linear equations.

Program Transformation and Fault-Tolerance: On the other hand, some authors have pointed out that many aspects of fault-tolerance can be captured using program transformations. For example, in [Gär98], a fault is defined as an "*unwanted nevertheless possible state transition*". In this work the notion of *failure model* is defined as a program transformation which captures the faults that a given program might exhibit. If a program is still correct after a program transformation, then the program is called fault-tolerant for that *failure-model.* The paper uses a UNITY style programming setting, and the UNITY programming logic is used to prove that some programs are tolerant to faults. [LJ92] extends the refinement calculus introduced in [Bac87]. A fail-stop scheme is assumed. The faults that may come from the hardware or the environment are considered; the programs are assumed to be fault-free or correct. Code transformation techniques are used to modify a program to tolerate hardware faults. Other approaches using program transformations are described in [PJ94, AK98].

Self-Stabilizing Programs: *Self-stabilizing* programs (programs which will eventually reach and stay in a predetermined set of states regardless of its initial state) have been an active area of research from the seminal work of Dijkstra [Dij74]. Extensions of the weakest precondition calculus can be used to prove self-stabilization properties of specific programs, as shown in [FvG99]. Several other works have proposed a formal framework to reason about self-stabilization, some of them are: [KP93, LS93, PS05], mainly extending the logic of UNITY.

Theorem Provers: The (semi-)automatic theorem prover PVS [ORS92] has been used to verify particular cases of critical systems such as airplane systems and distributed protocols. PVS uses higher-order logic and several other formalisms, like temporal logic or Hoare logic, can be embedded in this logic to reason about specific domains like imperative programs and reactive systems. Other theorem provers have been used to prove fault-tolerant properties in specific scenarios; examples can be found in [Mor02, Zha08, MG00, LR93, QS98]. On the other hand, model checking [CES86] has been used to verify and to validate specific

systems; for example, in [SECH98] the requirements of an embedded spacecraft controller were validated using SPIN [Hol97], and in [GLL+00] several properties of a railway control system were proven using SPIN. A more general approach is taken in [YTK01], where programs written in the programming language introduced in [Aro92] are translated to SMV [McM00], and then properties of a given program are verified using the SMV tool.

Process Algebra based Approaches: In several works, notions coming from process algebra [Mil79] are used to specify and verify fault-tolerant concurrent programs. For instance, in [Dix83, dBCG92] exceptions and interrupts are formalized in the language CSP [Hoa85]. [Pel91] uses redundancy in CSP to model fault-tolerance. [AP94] extends the π-calculus [MPW92] with failures, and gives some examples of fault-tolerant systems. [RH97] defines a process algebra and introduces a model of failures and "locations" (processes run in locations). Locations can be killed and new processes can be spawned in remote locations. In [Jan95], Janowski presents a CCS [Mil79] based approach to deal with fault-tolerance; in this thesis a new type of transition (called faulty-transition) is introduced in the labelled transition system. This allows the author to formalize the notion of fault, then a notion of bisimulation is introduced to define fault-tolerance. However, in these works no extension of Hennessy-Milner logic [HM80] with a corresponding deduction system is provided to describe and prove properties about the fault-tolerant processes defined with these languages.

Specification Languages and Fault-Tolerance: Several formal languages and frameworks have been used to formalize and to prove properties of specific examples of fault-tolerant systems. For instance, in [LM94] the byzantine generals problem is formalized with TLA$^+$ [Lam94]. In [Abr06], the Event-B language [AH07] is used to specify a train system. Another example using Event-B is given in [YB09], where a broadcast protocol is specified and verified. In [KJ08], a file system is specified with the Alloy language [Jac06], and verified using the Alloy analyzer. Duration calculus [CHR91] has been designed for reasoning about real time systems; several examples related with fault-tolerance and real time systems are described in [CH03] (e.g., a gas burner). In contrast to the other frameworks, the duration calculus uses a continuous flow of time.

Open Systems: Systems where the environment is taken into account as an active player (this is the case of fault-tolerant systems) and the control of the environment's behaviour is imperfect or non-existent are called *open systems* [Bar87]. This is in contrast to *closed systems*, where the entire behaviour of the system can be deduced from the behaviour of the components or modules which are part of it. In open systems, the interaction with an environment which cannot be controlled implies that the behaviour of it affects the behaviour of the system. One of the most common techniques to deal formally with open systems

are the rely-guarantee techniques [DM00, Jon83, CJ07, CC96, AL95], where the specification of the system relies on the assumption that the environment behaves in some way and, therefore, it guarantees some behaviour. However, as we say above, in fault-tolerance this assumption on the environment must be as minimal as possible.

Summarizing, the approaches focused on programming languages are useful when we analyze the fault-tolerant characteristics of a given program, but they are designed to reason at a low level of abstraction where the amount of detail involved may be cumbersome for reasoning about specifications. The extensions of process algebra provide interesting frameworks where fault-tolerant concurrent processes can be specified, but they lack well-studied logical frameworks. On the other hand, existing specification languages have been shown to be useful for specifying and verifying specific case studies; however, the difference between correct, expected or ideal behaviour and incorrect, unexpected or abnormal behaviour is just stated using *ad-hoc* mechanisms.

Deontic logics offer a natural way of distinguishing between normal and abnormal behaviour by means of deontic predicates. Systems can be specified by means of logical axioms and properties of systems can be proven using the logical laws of these formalisms. The general properties of deontic predicates can be used in different settings, avoiding the use of *ad-hoc* logical mechanisms. In addition, the benefit of deontic predicates in open systems is that we can impose norms on the behaviour of the environment and in cases where these norms are not followed, we can act in consequence. For these reasons, some authors have proposed deontic formalisms to reason about fault-tolerance. We review these approaches in the next section.

1.3 Deontic Formalisms and Fault-Tolerance

Deontic formalisms have been used in computer science, for different purposes (see [WM93]): database specification, reactive system specification, artificial intelligence and legal reasoning. As we said before, norms and normative reasoning arise naturally in fault-tolerance and it seems attractive to include deontic predicates into existing formal languages to have the possibility of distinguishing between normal and abnormal behaviour. However, the application of deontic logic to fault-tolerance is a recent topic of research. The following works use deontic logic for reasoning about problems related with fault-tolerance.

In [CJ96] an extension of *standard deontic logic* (see the next chapter) is proposed to reason about constraints in databases and to distinguish between *hard* (necessary)

and *soft* (deontic) constraints. The *soft* constraints admit violations and then the notion of recovery (from violation of static or state violation) is characterized. But the notion of transition constraint is not considered in this work, i.e., only norms regarding states are investigated.

In [LS04] *deontic interpreted logic* is used to formalize the bit-transmission protocol. The approach classifies agent states into *green* and *red*, and using this a deontic machinery is developed. However, as explained by the authors, the investigation of how to divide transitions into *red* and *green* is left as further research. Also, a question raised by the authors is *how these methods will scale up to deal with realistic examples with many agents and many kind of faults.*

In [Coe94], a dyadic deontic logic is proposed to formalize fault-tolerant programs and an example of an application is described. However, the semantics and the calculus of the described logic are not investigated by the author.

In [KM85] and [Kho88], Khosla and Maibaum propose a deontic logic to specify systems, although fault-tolerance is not dealt with in this work. The authors state clearly that this logic can be used for the prescription and description of systems and this can be used for characterizing abnormal executions. Khosla and Maibaum argue that the difference between prescription and description of systems is important when specifying systems. The description of a system action is usually given by establishing its precondition and postcondition. This is usually interpreted as a contract between the action being specified and the rest of the system. If the rest of the system ensures the precondition, then the action ensures the postcondition. However, there are some missing details, such as: in which scenarios is this action allowed to be executed? The precondition only tells us under what conditions the good behaviour of the action will be the expected. However, it is plausible that in some scenarios which satisfy the precondition, the action should still not be executed. Let us present a simple example to illustrate this fact. Consider the case of a bank account. The specification of the withdraw action could be stated informally in a pre/postcondition style as follows: *if the balance of the account is greater than an amount X, then after withdrawing X dollars, the balance of the account is the original balance minus X.* However, there could be cases where the account is not available to be used (for example a block on the account of the customer is imposed for some legal reason). Of course, several conditions can be added to take into account these cases, but then, in the specification, it is not possible to distinguish what is a normal scenario and what is an abnormal one.

Differentiating between normal and abnormal scenarios is important; dramatic actions can be taken in abnormal scenarios that are not taken in normal cases. For example, in the instance given above, the machine may make a phone call to the

police. Summarizing, the classic approach to systems specifications with pre and postconditions does not distinguish between normal situations and abnormal ones. This distinction is needed in fault-tolerance where abnormal scenarios must be taken into account to prevent undesired consequences. On the other hand, differentiating between normal and abnormal executions allows us to prove the properties that are true in normal situations [FM91a], and those that are true in consequence of an unexpected behaviour. Following Khosla and Maibaum [KM85], this approach can be called *total* specification since more cases are taken into account when defining an action.

In [KMQ93], a deontic logic is introduced and used to describe a library system. By means of this example, the authors show how this logic can be used to specify temporal constraints and error recovery. However, the logic is sketched and only a partial axiomatization is presented.

Many other authors (e.g., [WM93]) have stated that deontic systems are useful for reasoning about fault-tolerance because of the analogy between faults in computer systems and the situation in legal systems, but the analogy is not taken further.

1.4 Aims of the thesis

The specific goal of this thesis is to propose a logical framework with deontic predicates in which concepts related to fault-tolerance can be formalized and, therefore, fault-tolerant properties of systems (if any) can be proven. With this goal in mind, we investigate the meta properties of the logic proposed, and we show that it has some desirable properties: soundness, completeness, compactness and decidability. These properties make the logic appealing for practical use. The deontic predicates we use have novel properties with respect to those used in related work; we explain these characteristics in chapter 3. We study the properties of this new characterization of deontic predicates and their applicability to fault-tolerance.

The logical machinery introduced in the following chapters is intended to be used at a design level, when the main characteristics and the architecture of the system are delineated. We follow the ideas arising from modal action logics [Ken91, KQM91] and dynamic logic [HKT00], where systems are specified by means of logical theories and the actions of the system are described using modal operators in a pre/postcondition style. Moreover, we take some ideas of [FM92] where concepts arising in category theory are used to modularize temporal theories in such a way that the interconnections between the different logical theories reflect the architecture of the system. However, as we state above, since we consider the environment as an active player

in the system, and we take the view that the environment and some components of the system may have unexpected behaviour, the difference between description and prescription becomes important, and then we claim that the use of deontic predicates is important to state this difference at a design level. We ground our claims with practical examples.

During the description of a system, we specify what its structure is, how the different states of the system are defined (its variables and data structures) and what the effects of the actions are (i.e., we describe what the effects of the actions are when they are executed in a particular context). The prescription of the system, on the other hand, is the specification of the behaviour that the system should have, which may perhaps not be the case for several reasons: a malicious environment, a fault in the code or in the hardware, a design fault, etc.

Summarizing, deontic constructs allow us to incorporate in the specification non-normal behaviour. It is important to stress once more that this is different from the approach taken in many formal methods where non-normal behaviour yields undefined states. For example, in standard Hoare logic [Hoa69], if a precondition is not satisfied, then the effects of the action are not defined (and this is similar in several related approaches).

Today, most critical systems are concurrent, i.e., they are made up of several processes running in parallel. Manna and Pnueli have shown that temporal logics are useful formalisms to reason about concurrent and reactive systems [MP92], where perhaps a transitional approach (i.e., analyzing the correctness of software taking into account only the states before and after the execution of it) is not possible, in particular in those systems where there is no final state. For this reason, we include temporal logics in the formalisms described in this thesis; we also investigate concurrency at the level of actions, providing parallel composition of actions (a feature which is not always available in modal action logics).

Finally, when considering deontic predicates in specification, the notion of violation arises naturally. It is obvious that different violations may occur during the execution of a system; furthermore, since we decompose specifications into several components, it is worth asking what the relationship is between the different violations and how violations produced in one component affect the other components. Related to this, there is an important question: how can we modularize the deontic constructs in such a way that permissions or obligations imposed in one component do not affect (unless this is desired by the designers) the other components. In other words, when we decompose a system into different modules and relationships between them, we must take into account that the obligations or permissions established in some component may affect other parts of the system, and, in consequence,

some mechanisms for avoiding undesired over-specification are needed. We investigate these questions and we propose some theoretical devices to deal with them.

1.5 Structure of the thesis

In the next chapter we review briefly the notions needed to tackle the rest of the thesis. Firstly, we discuss the basic definitions of propositional, modal and temporal logics; then we take a look at deontic logics and we review some criticisms found in the literature about the different deontic formalisms that have been proposed by researchers in the field. As explained above, we use some constructions coming from category theory to compose a specification from several parts, and so in section 2.3 we introduce the basic notions of category theory, although this topic is vast and we mainly point to the literature for the interested reader. We also introduce *Institutions* and π-*Institutions*, two abstract views of logical systems which allow a designer to have a more general vision of specifications and to use different logical systems to reason about them.

In chapter 3 we introduce our own version of deontic logic, which has some novel features (as we explain in that chapter). Furthermore, since we intend to use this logic to specify systems by means of logical theories and to prove properties over these specifications, we describe an axiomatic system, which we prove is complete and sound with respect to the given semantics. We also investigate some other important properties of this logic, e.g., decidability.

In chapter 4 we present some case studies to show how the logical system presented in chapter 3 can be used in practice. We provide several specifications and we investigate their properties. In these examples we use the deontic predicates to introduce some prescriptions in the specification and from these prescriptions we observe that, frequently, violations arise naturally. However, as we discuss in the chapter, to describe a system in only one specification is complicated, and therefore, some formal machinery to decompose specifications into smaller ones must be provided. We discuss this again in chapter 7.

Proving properties by means of Hilbert style axiomatic systems is a powerful way of verifying systems. However, for practical application, automatic tools are needed. In chapter 5, we define a tableaux deductive system for this logic. Tableaux systems have been demonstrated as being useful in automatic theorem proving [Fit90]. A nice property of this formal system is that, in the case that a formula is not valid (i.e., it is not a theorem), then a counterexample can be obtained (i.e., a model which does not satisfy the formula). This is particularly important for analyzing specifications,

as we can obtain counterexamples which show what was wrong and, therefore, we can decide what we can do about it. We extend this system to cover temporal operators and we prove that this extended system is sound and complete with respect to the semantics. Moreover, in chapter 6 we prove that the two systems given (the Hilbert style and the tableaux systems) are logically equivalent, i.e., the theorems that we can prove using the two systems coincide.

Finally, in chapter 7 we present some techniques which allow us to use the logical systems introduced in preceding chapters in a modular way. Using these techniques, we can produce a system specification from smaller ones and the relationships between them. This induces some theoretical questions, for example, how the model of the entire system and the models of the smaller specifications are related and what kinds of properties we need to assume to preserve the properties of the components when we put them together. These questions are addressed in this chapter. It is worth remarking that we introduce a variation of bisimulation, which allows us to capture a notion of locality (or encapsulation) in the semantics. Furthermore, we show that the decomposition of a specification into smaller modules imposes a similar decomposition in the logical structure of the violations that may arise during the execution of the system. This enables modular reasoning over violations. To show the applicability of these techniques in practice, we revisit the example shown in chapter 4, but now from a modularity perspective, which allows us to show the possible benefits of this approach.

1.6 Notation

We follow standard notation in logic; usually we use the following symbols: $\rightarrow$ (material implication), $\wedge$ (conjunction), $\vee$ (disjunction), $\Box$ (modal necessity), $\Diamond$ (modal possibility). We use the following set operators: $\cup$ (union), $\cap$ (intersection), $\emptyset$ (the empty set), $-$ (set complement). To indicate the end of (the proof of) a theorem, we use a filled square ($\blacksquare$) and we use a blank square ($\square$) to indicate the end of a definition.

Chapter 2

Basic Concepts

In this chapter we introduce the basic concepts needed to tackle the rest of the thesis. Since the important work of Floyd [Flo67] and Hoare [Hoa69], computer scientists have used mathematical theories to develop software which behaves correctly (i.e., it behaves as is expected). In particular, a rigorous development and analysis of programs is necessary in critical applications. Examples of these applications are *airplane systems*, *health related systems* and *software in nuclear generators*. The work of Floyd and Hoare provided a promising basis for the mathematical development of programs. However, today computing systems are very complex; they are made of several subsytems and, often, they interact with an environment, and therefore additional techniques are needed to deal with these systems. From the seminal paper of Parnas [Par72], the notion of module or component is considered to be the basis of good practice for software design; modularization allows a designer to decompose a large system into several subsystems related to each other. Reasoning about the subsystems is much easier than reasoning about the entire system. Thus, any formal methodology which is aimed at being applied in practice must provide the possibility of specifying software in a modular way.

Another important step towards the mathematical development and analysis of software was given by the ADT (abstract data type) community [LZ75], in particular, by the algebraic school [EM85] where equational logic, and therefore algebraic models, were used to specify abstract data types. Abstract data types (formal descriptions of encapsulated data types) were recognized to be a key component of software systems, which are frequently reused for different applications. The techniques to support the modular specification of ADTs have been studied deeply, in particular by Goguen and Burstall in [GB92], where the concept of Institutions (abstract logical systems) is presented. However, software systems are in practice more complex than abstract

data types and algebraic methods are not powerful enough to express all the aspects of software. This is particularly true in reactive and concurrent systems, where it may not be required that the software terminates and the interaction with the environment is a key factor. Moreover, the notion of atomic computation is fundamental for analyses of some properties of these kinds of systems. As pointed out by Manna and Pnueli [MP92], temporal logics have been shown to be more adequate for the analysis of these kinds of systems. However, the first attempts at using temporal logics in specifications lacked the notion of module, which made it hard to apply this logical machinery to large systems. Subsequently, several approaches to solve this problem have been proposed. In particular, in [FM92] where the basic concepts of Goguen and Burstall were used in combination with temporal logics. The main idea is to specify components of software using temporal theories and after this use categorical constructions (see next section) to compose the components. This line of work follows the philosophy introduced in [MT84], where axiomatic theories are the fundamental unit of construction used during the process of software development, and translations between theories are used to relate components and to guide the development process. We adhere to this philosophy throughout this thesis, i.e., we specify the components of systems using axiomatic theories (expressed in some underlying logical system) and the relationships between them are established by means of the notion of translation between theories. (This notion is dependent on the logical system used to express the components.)

It is important to stress one relevant aspect of temporal logics. In most of them, automatic techniques of verification can be used for the analysis of specifications. In particular, most temporal logics used in computer science have as semantics transition systems. Thus finite state programs can be mapped to semantic models, and therefore, it is possible to check possible implementations against their specifications to see if both agree. This methodology is called *model checking* [EC80, CES83, CES86] and has been widely used in the formal methods community to verify programs. However, a problem with this technique is that the representation of systems as transition system suffers from state explosion when the systems become larger and more complex [BCM+90]. In this case, also, decomposing specifications is fundamental. This is an active topic of research and some progress has been made using *assume-guarantee* mechanisms [Lam83, CLM89, ASS94].

Summarizing, we have different approaches. On the one hand, we can use axioms and axiomatic systems to specify and prove properties of systems; in this case the proofs could be done by hand (which is a hard task), or this task can be done with the help of (semi-)automatic provers. These software tools guide the developer during the task of proving, but they are not fully automatic, except in simple cases. Another approach is to take advantage of the decidability of temporal logics and use techniques like model checking, where the process is fully automatic, though large systems are

hard to verify because of the state explosion problem. We think that both have benefits, and therefore in the following we describe different deduction systems which allow us to combine the benefits of both approaches. It is important to remark that for decomposing a specification into smaller components, we use the ideas of Fiadeiro and Maibaum introduced above, where components are specified by temporal theories and syntactical translations between these theories reflect the architecture of the system. Here category theory plays a central role, as it provides basic notions to combine components. All these concepts are reviewed briefly in this chapter.

The chapter is structured as follows. We start with a brief introduction to *modal action logics* and *dynamic logic*, a variation of modal logics, which have been shown to be useful to formalize concepts arising from computer science. These logics have the notion of action embedded in their constructions and their semantics is given by transition systems. We also point out the basic ideas behind temporal logics, which will allow us to talk about the future of a component in specifications. Finally, we introduce the ideas of category theory and their application to software engineering that we will use later on in this thesis.

2.1 Propositional Logic, Modal Action Logics and Temporal Logics

We start this section presenting *Hilbert-style* and *tableaux-style* deductive systems for propositional logic. We use these two well-known and simple examples to illustrate the way in which we define the different logical systems introduced throughout this thesis. In addition, both systems are used as a basis for the more complex systems presented later in this chapter. The language or vocabulary of propositional logic is given by an enumerable set of propositional letters P. We denote its elements by lowercase letters $p, q, s, \ldots$. The set of formulae are defined recursively as follows:

- if $\varphi \in P$, then φ is a formula,
- if φ and ψ are formulae, then $\neg\varphi$ and $\varphi \rightarrow \psi$ are formulae.

The rest of the boolean connectives can be defined from these operators, see [Mon76] or [Eme72]. As usual, the semantics of propositional logic is defined by interpretations of the propositional variables which assign truth values to them. We present the axiomatic system introduced in [Mon76] which can be proven sound and complete. We have the following axioms.

- $\varphi \rightarrow (\psi \rightarrow \varphi)$
- $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$
- $(\neg\varphi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \varphi)$

and the deduction rule *modus ponens*: from φ and $\varphi \rightarrow \psi$ we deduce ψ. We can give a formal definition of proof. Given a set of formulae Γ, we say that $\Gamma \vdash \varphi$ (or φ is provable from Γ) when there is a finite sequence $\varphi_0, \ldots, \varphi_n$ such that $\varphi_n = \varphi$ and each φ_i is either an axiom, belongs to Γ, or follows from formulae φ_k and φ_j with $j, k < i$, by modus ponens. We can define a similar semantical relation $\Gamma \models \varphi$, see [Mon76] (or any classical textbook of logic) for the definition. In any logic, the relationships between $\vdash$ and $\models$ are important so as to know that the deductive system is coherent with respect to the desired semantics. $\vdash$ is a consequence relation as is $\models$, and it can be defined in different ways; we talk about this below. This style of presentation of a logical system is followed in the next section, i.e., we present the language of the logic, its semantics, and its deductive apparatus. In what follows, we use the standard properties of propositional logic and the reader can consult the good introductions given in [Mon76, Eme72, Men79]

On the other hand, *tableaux systems* offer another flavour of deductive systems. The main ideas behind tableaux systems come from the seminal work of Gentzen [Gen69]. Here we follow the style of presentation given in [Smu68]. The method is simple and elegant; given a formula, we analyze its parts to try to find a contradiction. In presenting the tableaux system, it is better to consider *disjunction*, *negation* and *conjunction* as primitive operators instead of *implication*, i.e., we have the following class of formulae (which can be defined as above): $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\neg\psi$. We say that $\{p, \neg p\}$ (where p is a propositional symbol) is a closed set of formulae. Closed sets of formulae are intended to capture the notion of inconsistency. Before continuing with the presentation of the method, it is important to introduce a classification of formulae (see figure 2.1). This classification allows us to differentiate formulae based on their structure. Using this classification, we can introduce the rules given

A	A_1	A_2		B	B_1	B_2
$\varphi \wedge \psi$	φ	ψ		$\varphi \vee \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$		$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$
$\neg\neg\varphi$	φ					

Figure 2.1: Classification for formulae A and B.

in figure 2.2. The entire idea of the tableaux method is as follows: we start with the negation of the formula that we want to prove; then we use the rules of figure 2.2 to produce a tree which reflects the underlying structure of the formula. Every branch

$$A : \frac{A}{\begin{array}{l} A_1 \\ A_2 \end{array}} \qquad B : \frac{B}{B_1 \mid B_2}$$

Figure 2.2: Classic rules for formulae of type A and B

(informally, a path from the root of the tree to some leaf node) is a possible model. If all the branches are closed (i.e., contain a closed set), then there is no model for the negation of the formula, and therefore the original formula is valid. The reader can consult [Smu68] and [Fit90] for details about the tableaux method and examples of application. An important point to note is that this method, for many logics, is automatable, and in case of a formula that cannot be proven, a counterexample is obtained. This is very useful in software specification since counterexamples allow us to deduce what was wrong with a given program and what we need to do to fix it. We say that $\vdash \varphi$ when we succeed in proving φ with the tableaux rules, and we say that $\Gamma \vdash \varphi$ if there is a finite set $\{\varphi_0, \ldots, \varphi_n\} \subseteq \Gamma$ such that $\vdash \varphi_0 \wedge \cdots \wedge \varphi_n \rightarrow \varphi$. This notion of deduction can be proven to coincide with that given for propositional logic.

2.1.1 Modal Logic

Modal logics arose from the study of modalities (e.g., *can*, *could*, *necessary*, etc.) in reasoning. Usually, two modalities are considered: $\Box\varphi$ (φ *is necessarily true*) and $\Diamond\varphi$ (φ *is possibly true*). It is worth remarking that there exists a duality between them which resembles the duality between the $\forall$ and $\exists$ in first order logic:

$$\Box\varphi \Leftrightarrow \neg\Diamond\neg\varphi.$$

The semantics of a modal logic is given by Kripke structures, i.e., by structures $\langle W, R, I \rangle$ where W is a set of worlds or states, $R \subseteq W \times W$ is a relationship which tells us which worlds are accessible from which other worlds, and I is a function which indicates which propositions are true in each world. The axioms of modal logics depend on the type of relation R that we consider. This relationship could be reflexive, symmetric, transitive, serial, etc. The most common axiomatic system is the **K** system (**K** for Kripke) in which no restriction on relation R is assumed. The **K** system contains the following axioms:

- The axioms of propositional logic

- $\Box(\varphi \rightarrow \psi) \rightarrow \Box\varphi \rightarrow \Box\psi$
- $\Diamond\varphi \leftrightarrow \neg\Box\neg\varphi$

The rules of deduction are modus ponens and modal generalization: from $\vdash \varphi$ we get $\vdash \Box\varphi$. This axiomatic system is sound and complete for Kripke structures with no restrictions. The literature on modal logic is vast; the reader can consult the good introductions given in [Che99] and [HC96]; a book more oriented to computer science is [BRV01]. Tableaux systems for modal logics have been investigated deeply, standard references are [Fit72, Fit83, Gor95]. Most of these systems use labelled formulae (i.e., formulae that have some kind of label attached). Intuitively, the labels point out some kind of relationship between the semantics and the syntax. Labelled deductive systems are dealt with in detail in [Gab96]. We use this technique in chapter 5. We are interested in modal systems which can be used to express statements about computing systems (in particular *modal action logics* and *dynamic logics*).

2.1.2 Dynamic Logics and Modal Action Logics

In [Pra76] Pratt proposed to extend the modalities of modal logic to formalize the notion of program correctness in a way that is similar to the approach in Hoare logic. The logic obtained is called *dynamic logic* (see [HKT00]). Dynamic logic introduces the concept of action as a modality; for example, instead of having $\Box\varphi$, we have $[a]\varphi$ where a is an action. Actions can be combined in different ways to generate more complex ones. Usually, the action combinators considered in dynamic logics are *composition*, *iteration*, *choice* and *converse*. On the other hand, in the early eighties the FOREST project [KQM91, RFM91, Mai87] started developing *modal action logics* (or MALs for brevity); these logics are closely related to dynamic logic but they consider other operators in addition to the standard ones. (This work was actually motivated by the work reported in [Gol82] rather than the dynamic logic approach.) Following Broersen [Bro03], we consider dynamic logic as a specific modal action logic which only uses a specific set of action combinators. Several works extend these operators with less usual ones, e.g., *intersection* (parallel execution) and *complement*. However, these operators bring some technical problems. For example, under the relational interpretation of actions (given in many modal logics), the complement of an action returns a relation which relates states which are not related by the original program, i.e., for specifying computing systems another kind of complement is desirable. Intersection of actions is also not less problematic, since the relational interpretation (i.e., intersection of relations) together with composition and iteration exhibit properties that are not intuitively correct. Furthermore, iteration plus complement make any logic undecidable (see [Bro03]). Note that both operators are useful at design time

to specify software systems: intersection allows us to specify the notion of parallel execution, which is important to have when working with concurrent systems, and complement allows us to express the notion of alternative action, which is important to express frame axioms, e.g., when we need to say that a determined behaviour is exclusive of some action.

As an example, we present the definition of standard dynamic logic (or DL for short); see [HKT00] for details. The language of DL is given by a pair $\langle \Phi_0, \Delta_0 \rangle$ where Φ_0 is an enumerable set of propositions and Δ_0 is an enumerable set of primitive actions (or names of actions). We use the Greek letters $\alpha, \beta, \ldots$ as variables over actions. The set of (composed) actions is defined as follows:

- if $\alpha \in \Delta_0$, then α is an action.
- if α and β are actions, then $\alpha;\beta$, $\alpha \sqcup \beta$ and α^* are actions.
- if φ is a formula (see below), then $\varphi?$ is an action.

Note that the definition is mutually recursive with the definition of formulae given below. Intuitively, ; is the sequential composition of actions, $\sqcup$ is the nondeterministic choice, * is iteration and ? is a test ($\varphi?$ means proceed if φ is true, fail otherwise). The set of formulae is given by the following recursive definition.

- if $\varphi \in \Phi_0$, then φ is a formula.
- if φ and ψ are formulae, then $\neg\varphi$ and $\varphi \rightarrow \psi$ are formulae.
- if φ is a formula and α is an action, then $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are formulae.

Intuitively, $[\alpha]\varphi$ is true when after executing action α it is necessarily the case that φ holds; and $\langle\alpha\rangle\varphi$ is true when there is some way of executing action α such that we reach a state where φ is true. The semantics of DL is given by a Kripke structure $\langle W, I \rangle$ where W is a set of states or worlds, and I is a meaning function mapping each proposition to a subset of W (the subsets of states which satisfy the proposition), and each primitive action to a binary relation between states, that is:

- $I(p) \subseteq W$, for every $p \in \Phi_0$.
- $I(a) \subseteq W \times W$, for every $a \in \Delta_0$.

The function I can be extended to formulae and actions as follows:

- $I(\alpha;\beta) = I(\alpha) \fatsemi I(\beta)$.
- $I(\alpha \sqcup \beta) = I(\alpha) \cup I(\beta)$.
- $I(\alpha^*) = I(\alpha)^*$.
- $I(\varphi?) = \{\langle u, u\rangle \mid u \in I(\varphi)\}$.
- $I(\langle\alpha\rangle\varphi) = I(\alpha) \fatsemi I(\varphi)$.
- $I([\alpha]\varphi) = I(\neg\langle\alpha\rangle\neg\varphi)$.
- $I(\neg\varphi) = W - I(\varphi)$.
- $I(\varphi \to \psi) = W - (I(\varphi) \cup I(\psi))$.

(Here $\fatsemi$ denotes the forward relational composition, see [HKT00].) The remaining standard formulae can be defined from the ones presented above. Using these definitions we say that $w, M \models \varphi$, for a Kripke structure M and a state w of M, iff $w \in I(\varphi)$. In [HKT00] a complete and sound axiomatic system is given for DL, though this system is not compact, which is a consequence of having iteration in the logic (which cannot be defined in first order logic, see [Gol82]).

Note that the semantics of actions is given by relations, and the semantics of the action combinators is given by relational operators; we call this approach a *relational semantics*. This kind of semantics is the one mainly used in modal action logics [Pra78, Par78, Dan84, BV01, GP90, dR98, HKT00, Bro03]. In [KQM91] another possibility is proposed. Actions are interpreted as a set of events, and events are the labels on the transitions of the semantic model, though the properties that relate the semantics and the syntax are not investigated there. We call this kind of semantics an *algebraic semantics*, since the events in the semantic model have some kind of algebraic structure. This is similar to the approach taken by Hennessy and Milner in [HM80], where a multimodal logic is presented to reason about processes defined in a process algebra style [Mil89], and actions are interpreted as labels of transitions. A similar approach is also taken in [FM97] to give the semantics of the language *CommUnity*, where the interpretations of designs or programs are labelled transition systems, and the labels of transitions are events that can be "observed" by several actions.

We pursue the algebraic approach in this thesis, and we show later on that nice properties can be obtained from this semantical choice. For example, we give a complete and sound system for a modal action logic with boolean combinators; furthermore, we prove the compactness of this logic, which is an improvement on the

similar logics proposed in the literature [GP90] and [Bro03]. We come back to this issue in chapter 3.

Following the terminology introduced in [Bro03], we use MAL(...) for naming a modal action logic which has the action combinators appearing inside the brackets; e.g., dynamic logic is named MAL($;, \sqcup, *, ?$). The operators which can be found in the literature are: $;$ (*composition*), $\sqcap$ (*intersection*), $\sqcup$ (*union*), $*$ (*iteration*), $?$ (*test*), $-$ (*complement*), $^{-1}$ (*converse*), 1 (*skip*) and 0 (*fail*). The following are the results known about modal action logics; almost all of them use a relational approach to the semantics:

- MAL($\emptyset$) - this is multimodal logic, i.e., modal logic extended with multiple modalities, one for each action. Sound and complete systems can be found in the literature; the logic is decidable. This logic is also called Hennessy-Milner logic in process algebra, though in this case the semantical structures used are labelled transition systems. (Note that multimodal logics use a relational semantics, i.e., each modality is interpreted as a relation.) The two systems are equivalent, but it is interesting to note that the Hennessy-Milner logic was born in the process algebra community, and therefore it is not strange that for this logic an algebraic interpretation of modalities was used. As remarked in [Bro03], the logic MAL($\sqcup, ;, 1, 0, ?$) is definable from MAL($\emptyset$), i.e., no expressivity is gained when these operators over actions are added.

- MAL($;, \sqcup, *, ?$) - this is dynamic logic; sound and complete systems exist for this logic. See [HKT00]. This logic is decidable, but it is not compact.

- MAL($;, \sqcup, *, ^{-1}$) - this is dynamic logic with converse; sound and complete systems are known for this logic [Par78], [GM96]. This logic is also decidable (obviously it is not compact since it extends dynamic logic). Dynamic logic with converse is more expressive than standard dynamic logic.

- MAL($;, \sqcup, ^{-1}$) - this is a fragment of the logic shown above, using the axiomatization given in [Par78]. We can obtain a sound and complete system for this logic.

- MAL($;, \sqcup, *, \sqcap$) - called dynamic logic with intersection (or IPDL). This is perhaps the variant of dynamic logic least studied in the literature. The logic was proven to be decidable [Dan84], though this logic does not satisfy the *finite model property*. The axiomatization for IDPL was an open problem for several years, in part because the relational semantics of intersection is not modally definable. In [BV01], a sound and complete axiomatization for this logic is given. However, it is important to note that the interpretation of parallel execution as intersection of relations is sometimes not in correspondence with the behaviour

of concurrent programs. This is mainly because we have composition and iteration in the logic and the notion of an atomic step is lost in the semantics. Indeed, if we have an action which does not finish, in the relational semantics, this is interpreted as an empty relationship. This engenders the same problem that Hoare logic has to deal with for programs which run forever. Moreover, in this case the notion of concurrency loses sense in the relational approach (the intersection of any action with an empty action is empty). Here it seems much more interesting to interpret actions as sequences of labels or events. We will come back to this issue later on.

- MAL$(;, \sqcup, \sqcap, -)$ - called *boolean modal logic* or BML [GP90]; a sound and complete axiomatic system is given in this paper. This logic is not compact as shown in [Bro03]. Here it is important to note that the complement used in these logics is an *absolute complement*, i.e., if we have an action a which is interpreted as a relation R_a and $U = W \times W$ is the universal relation, then we have that the complement of a is $U - R_a$ (where $-$ is the set difference). It is not hard to see why this semantics of complement it is not desirable from a computer science point of view. In this context the complement is used to formalize the notion that an alternative action is executed; with the absolute complement we can relate states that are not related by the original action, obtaining in this way a behaviour that is chaotic with respect to the system being specified. A better approach is to consider a relative complement. In spite of this being a more adequate approach in computer science, modal logics with relative complement have not been investigated deeply. In [Bro03] a boolean modal logic with relative complement is proposed, but its possible axiomatization is not investigated further.

- MAL$(;, \sqcup, \sqcap, -, ^{-1}, ?)$ - this system is investigated in [dR98]; the system is proven undecidable, but a sound and complete axiomatization is described.

As we said earlier, we propose in chapter 3 a boolean modal logic which, differing from [GP90], uses a relative complement, and, a difference with respect to what is proposed (but not axiomatized) in [Bro03], it uses an algebraic approach, i.e., actions are interpreted as a set of transition labels. This choice is useful to obtain a simple formalization of the boolean operators and the relative complement.

2.1.3 Temporal Logics

In [Pnu77], Pnueli proposed to use temporal logics to specify and reason about programs. Since then, temporal logics have been used extensively by computer scientists

to underpin the production of reliable software. Temporal constructs are essential when we need to reason about temporal properties of programs. Some standard temporal properties are: *a property φ is always true in the future*, *eventually a property ψ will be true*, *a property φ is true until the property ψ becomes true*, etc. In particular these kinds of statements are useful in reactive systems or systems that do not necessarily have to terminate (e.g., an operating system), where the machinery of Hoare logic is hard to use (following Manna and Pnueli [MP92] this is one of the main reasons to use temporal logic in reactive and concurrent systems). There are different temporal logics, each one reflects a different conception of time. We can classify temporal logics in different ways.

We can distinguish between *linear time logics* and *branching time logics*. Linear temporal logics [Pnu77] are widely used in computer science to specify and verify computing systems. These logics assume that the flow of time is linear and discrete, i.e., any given instant of time has only one successor. On the other hand, *branching time logics* suppose a non-deterministic discrete setting where a given instant has multiple successors. This approach is particularly used when non-determinism is present in specifications. In the following we briefly introduce both systems.

Linear Temporal Logics

Linear temporal logics (or LTLs) were proposed by Pnueli in his seminal paper [Pnu77]. This logic extends propositional logic with a simple set of temporal modalities: $\mathsf{N}\varphi$ (*in the next instant φ is true*), $\mathsf{F}\varphi$ (*eventually φ is true*) and $\mathsf{G}\varphi$ (*always in the future φ is true*) and $\varphi \,\mathcal{U}\, \psi$ (*ψ is true at some moment in the future, and until ψ becomes true, φ is true*). Several other operators can be defined using these, in particular a weak version of $\varphi \,\mathcal{U}\, \psi$ (i.e., *ψ may not be true in the future*) can be defined (see [MP92]).

The semantics of LTL is given by Kripke structures and *traces* over it (i.e., paths in the Kripke structures). Paths are usually maximal (though some works consider all the possible paths), and the relation of satisfaction is defined with respect to an instant, a path and a Kripke structure, i.e., $i, \pi, M \models \varphi$ says φ is true at moment i of path π in the structure M. The semantics for these operators (with the intuitive reading) can be found in the standard textbooks [MP92] and [Eme90].

Sound and complete proof systems are known for LTL, for example in [MP83] the following Hilbert-style system for LTL is presented.

LTL1. $\quad \neg\mathsf{F}\varphi \leftrightarrow \mathsf{G}\neg\varphi.$

LTL2. $\mathsf{G}(\varphi \rightarrow \psi) \rightarrow (\mathsf{G}\varphi \rightarrow \mathsf{G}\psi)$.
LTL3. $\mathsf{G}\varphi \rightarrow \varphi$.
LTL4. $\mathsf{N}\neg\varphi \rightarrow \neg\mathsf{N}\varphi$.
LTL5. $\mathsf{N}(\varphi \rightarrow \psi) \rightarrow (\mathsf{N}\varphi) \rightarrow (\mathsf{N}\psi)$.
LTL6. $\mathsf{G}\varphi \rightarrow \mathsf{N}\varphi$.
LTL7. $\mathsf{G}\varphi \rightarrow \mathsf{NG}\varphi$.
LTL8. $\mathsf{G}(\varphi \rightarrow \mathsf{N}\varphi) \rightarrow (\varphi \rightarrow \mathsf{G}\varphi)$.
LTL9. $\varphi \,\mathcal{U}\, \psi \leftrightarrow (\psi \vee (\varphi \wedge \mathsf{N}(\varphi \,\mathcal{U}\, \psi)))$.
LTL10. $\varphi \,\mathcal{U}\, \psi \rightarrow \mathsf{F}\psi$.

with the rules:

- If φ is an instance of a propositional tautology, then $\vdash \varphi$.
- If $\vdash \varphi \rightarrow \psi$ and $\vdash \varphi$, then $\vdash \psi$.
- If $\vdash \varphi$, then $\vdash \mathsf{G}\varphi$.

LTL1 defines a duality between F and G. **LTL2** is the **K** axiom of modal logic; note that since the last rule given above is the generalization rule, we have a normal modal system. **LTL3** says that the present is included in the future. **LTL4** says that, if something is false in the next instant, then it is false that this property is false in the next instant (note that this formula is valid since we are in a linear flow of time). **LTL5** is the **K** axiom for the next operator. **LTL6** tells us that the future includes the next instant. **LTL8** can be thought of as an induction property, and the remaining axioms only express the definition of the operators.

LTL has been widely used for model checking [Hol97]. In [SC85] it was proven that the LTL model checking problem is PSPACE complete. In a context where non-determinism is possible, branching time logics offer the possibility of quantifying over the branching produced by the non-determinism. As stated in [HR04]: *Branching time appears to make the non-deterministic nature of future more explicit.*

Branching Time Logics

Branching time logics consider, for a given instant of time, multiple successors, and therefore operators to quantifying over different possible futures become possible. In [EC80, BAMP81], the quantifiers A and E were introduced. One of the most used branching temporal logics in computer science is CTL (*computational tree logic*) introduced in [EC80]. (They also introduced a restricted version of CTL, called UB,

which does not have the until operator.) Now we have two kinds of modal operators: on the one hand, we have operators quantifying over paths, and on the other hand we have temporal operators. CTL only allows us to combine them in certain ways. More specifically, we can only use the pairs: $\mathsf{A}\,\mathcal{U}$, AG, AF, AN, $\mathsf{E}\,\mathcal{U}$, EG, EF and EN. The intuitive interpretation of the operators is as follows:

- $\mathsf{A}(\varphi\,\mathcal{U}\,\psi)$, *in all the possible futures,* φ *is true until* ψ *becomes true.*
- $\mathsf{AG}\varphi$, *in all the possible futures,* φ *is always true.*
- $\mathsf{AF}\varphi$, *in all the possible futures,* φ *is eventually true.*
- $\mathsf{AN}\varphi$, *in all the possible next instants,* φ *is true.*
- $\mathsf{E}(\varphi\,\mathcal{U}\,\psi)$, *in some possible future,* φ *is true until* ψ *becomes true.*
- $\mathsf{EG}\varphi$, *in some possible future,* φ *is always true.*
- $\mathsf{EF}\varphi$, *in some possible future,* φ *is eventually true.*
- $\mathsf{EN}\varphi$, *in some possible next instant,* φ *is true.*

Sound and complete axiomatizations are given for CTL in [Eme90]. In [SC85] it was proven that the complexity of model checking a CTL formula φ is $O(K*|\varphi|)$ where K is the size of the model and $|\varphi|$ is the length of φ. LTL and CTL are incomparable in the sense that there are sentences which are expressible in LTL and are not expressible in CTL and vice versa [Eme90].

Though CTL has interesting expressivity, in [EH86] an extension of CTL called CTL* is presented. CTL* provides more flexibility for combining temporal operators with branching quantifiers. LTL and CTL are strictly contained in CTL* (i.e., all the properties that can be expressed in LTL and CTL can be expressed in CTL*). This logic is decidable with the same complexity as LTL. The known axiomatizations of CTL* are far from being as simple as those for CTL and LTL (see [Rey01]). There have been a lot of discussion about which of these logics is better for specifying programs (see [Sch04] for an introduction to this discussion). However, in a context where we have nondeterminism, branching time logic is a natural choice. On the other hand, CTL admits simpler axiomatizations than CTL*, which can be a benefit when proving properties by hand.

In [MP90], (following the terminology introduced in [Lam77] for concurrent programs in general) temporal properties are divided into categories. There are two which in practice are very common: *safety properties* (they state that some property

is true for every instant and any state of the system, i.e., this property is a system invariant), and *liveness properties* (they state that some property will be satisfied in the future).

There exist several other variants of temporal logic (some of them have also been used in computer science), e.g., there are logics with a continuous flow of time, with intervals, with convergent flows of time, etc. These logics have varying applications, e.g., temporal logics with a continuous flow of time are usually used for reasoning about real time systems.

2.2 Deontic Logic

Deontic logic is a branch of modal logic which focuses on the study of the reasoning arising in ethical and moral contexts, which usually involve norms and prescriptions (see [Aqv84] and [Che99]). Two modalities which can be found in most of the deontic logics are *permission* and *obligation*. Of course, related to them are the concepts of *prohibition* and *violation*. Mally was the first to try to capture the reasoning underlying norms and prescriptions, in particular Mally introduced obligation as a predicate (together with other related operators) and provided an axiomatic system for his logic. However, in Mally's logic the concept of obligation is superfluous, in the sense that, if we take $\mathsf{O}(\varphi)$ as saying it ought to be the case that φ, then we obtain $\varphi \rightarrow \mathsf{O}(\varphi)$! Since then, several deontic logics have been proposed in the literature. Perhaps the most studied is the so-called *Standard Deontic Logic* (or SDL) [Che99]. A particular case of normal modal logics, SDL has the modality $\mathsf{O}(\varphi)$ (φ is obligatory); and the following axioms are proposed in this logic to capture the notion of obligation [McN06]:

SDL0. all the tautologies of the language.
SDL1. $\mathsf{O}(\varphi \rightarrow \psi) \rightarrow (\mathsf{O}(\varphi) \rightarrow \mathsf{O}(\psi))$.
SDL2. $\mathsf{O}(\varphi) \rightarrow \neg\mathsf{O}(\neg\varphi)$.

For the rules we have:

- If $\vdash \varphi \rightarrow \psi$ and $\vdash \varphi$, then $\vdash \psi$,
- If $\vdash \varphi$, then $\vdash \mathsf{O}(\varphi)$.

Equivalent axiomatizations of SDL can be found in [Che99] (this system is called $\mathbf{OK}^+$ in [Aqv84]). The second deduction rule means that we have a normal modal system.

The semantics of SDL is given with Kripke structures, and the interpretation of the obligation operator is the same as the modal necessity (although this axiomatization imposes a different structure on the Kripke models; note that the axioms imply that the Kripke structures are *serial*, i.e., every state has a successor). The intuition of the semantics is as follows. If a state w is related with another state v, this means that the obligations occurring in w are true in v (in some way we can think of v as an "ideal" world for w). Several consequences of this axiomatic system have been criticized for being contrary to the intuitive properties of obligation. For example, a consequence of these axioms is the property $\mathsf{O}(\mathsf{T})$ (which can be read as saying that there are always obligations; at least we have that all the tautologies are obliged). Some people have argued that there could be scenarios where nothing is obliged, and these kinds of scenarios are not possible in SDL. Another problematic issue is the definition of permission which is introduced in the logic as a dual of obligation, that is: $\mathsf{P}(\varphi) \leftrightarrow \neg\mathsf{O}(\neg\varphi)$. Note that this definition together with axiom **SDL2** imply that we have the following theorem: $\mathsf{O}(\varphi) \rightarrow \mathsf{P}(\varphi)$, i.e., obligation implies permission. If we see permission as modal possibility (which is the case in SDL), then we have what is sometimes called Kant's law: obligation implies possibility. It is not hard to find examples where this property is not desirable. It is not our intention to defend or argue against this logical system; readers can take their own position. Further discussion about these topics can be found in [Che99]. Below, we present some predicates that have been problematic not only in SDL, but also in different deontic systems that have been proposed in the literature. These predicates are called *paradoxes* (but do not confuse them with logical paradoxes, e.g., Russell's paradox of set theory) since they are properties which are contrary to our intuition.

2.2.1 Paradoxes in Deontic Logic

Many paradoxes of deontic logic were discussed through the years, some of the most well-known are:

- $\mathsf{O}(\varphi) \rightarrow \mathsf{O}(\varphi \vee \psi)$ (Ross's Paradox). An intuitive reading is: *if you are obliged to send a letter, then you are obliged to send it or burn it.*
- $\mathsf{P}(\varphi) \vee \mathsf{P}(\psi) \leftrightarrow \mathsf{P}(\varphi \vee \psi)$ (No free choice permission). This is similar to the sentence explained above.
- $\varphi \rightarrow \psi \vdash \mathsf{O}(\varphi) \rightarrow \mathsf{O}(\psi)$ (Good Samaritan paradox). As the name of this paradox implies the following reading seems problematic: *if Jones helps Smith, who has been injured, implies Smith has been injured, so if it is obliged that Jones helps Smith who has been injured, then it is obliged that Smith has been injured.* That is, we can obtain an obligation from a hypothetical situation.

A complete list can be consulted in [MWD94] and [Aqv84]. However, we are interested in the paradoxes called *contrary-to-duty* paradoxes where a secondary obligation appears after violating a primary obligation (like the *good Samaritan paradox* described above). These paradoxes have been the most controversial in deontic logic, and they are the most difficult to solve. It is not hard to see that reasoning of this style arises in fault-tolerance, where after the violation of an obligation, some recovery action must be performed to restore a state free of errors. Some of the most well known contrary-to-duty paradoxes are the following: Chisholm's paradox, which can be informally stated as: *If you are obliged to go to a party, then it is obliged that, if you go, you have to say that you are going; but if you do not go you are obliged to not say that you are going; you do not go to the party.* In SDL, for example, we can obtain $\bot$ from these statements. This seems unreasonable since this is a plausible scenario in the real world. Chisholm's paradox can be formalized as follows:

$$(\mathsf{O}(p) \wedge \mathsf{O}(p \rightarrow t) \wedge \neg p \rightarrow \mathsf{O}(\neg t) \wedge p) \rightarrow \bot$$

Another contrary-to-duty paradox is the *Gorbachov-Reagan paradox* [Bel87], which is as follows: *You must not tell the secret to Reagan and you must not tell the secret to Gorbachev. If you tell the secret to Gorbachev, you must tell the secret to Reagan. If you tell the secret to Reagan, you must tell the secret to Gorbachev. You tell the secret to Reagan and Gorbachev.* If we formalize this situation in SDL, we obtain a contradiction since we get two contradictory obligations which yield false in SDL. However, this scenario is plausible (if we change the names Reagan and Gorbachev!). There exist many other contrary-to-duty paradoxes the reader can consult [MWD94]. We remark that these kinds of paradoxes are very common in a fault-tolerance context, and therefore any logic which is intended to be used to specify and reason about fault-tolerant programs needs to provide some effective way to deal with contrary-to-duty reasoning. We show how our logic deals with contrary-to-duty statements in section 3.6.

2.2.2 Deontic Action Logics

The logics that we have introduced in the section above can be classified as *ought-to-be* deontic logics, since the norms are applied to predicates. On the other hand, *ought-to-do* deontic logics are those where we impose norms on actions, which (as argued by several authors [Mey88, Bro03]) are more suitable to use for specifying computing systems, mainly because these logics have a notion of state change caused by the execution of actions. One of the most well-known ought-to-do systems is Dynamic Deontic Logic introduced in [Mey88]. In this work, deontic constructions are reduced to dynamic logic constructions using a violation constant which indicates that a violation has been produced. Meyer proposes to use the following combinators:

; (composition), $\sqcup$ (non-deterministic choice), $\sqcap$ (parallel execution), and $-$ (alternative action), and an algebra of actions is proposed for these action combinators. Using modalities, Meyer defines:

$$\mathsf{F}(\alpha) \leftrightarrow [\alpha]\mathbf{v}.$$

That is, an action is forbidden if and only if every execution of this action yields a violation. From this Meyer defines the rest of the deontic predicates: $\mathsf{O}(\alpha) \leftrightarrow \mathsf{F}(\overline{\alpha})$ (obligation) and $\mathsf{P}(\alpha) \leftrightarrow \neg\mathsf{F}(\alpha)$ (permission). Broersen [Bro03] called this approach *goal oriented norms* since, from evaluating the truth value of a deontic predicate, only the resulting state of an action is important and not what happens during its execution. Several criticisms have arisen to this approach. For example in [vdM96], the following formula is exhibited as a paradox of dynamic deontic logic: $\langle\alpha\rangle\mathsf{P}(\beta) \rightarrow \mathsf{P}(\alpha;\beta)$, which can be read as *if after shooting the president it is allowed to remain silent, then it is allowed to shoot the president and remain silent!* which is undoubtedly undesireable. In [Ang08] these ideas are used to establish a more serious paradox: $\mathsf{F}(\alpha) \rightarrow [\alpha]\bot$, i.e., forbidden actions cannot be executed. In spite of these facts, Meyer's approach is interesting since in deontic dynamic logic a clear division between predicates and actions is established and, as Meyer argues, some paradoxes vanish in this approach, mainly since here we have a notion of time or state change. Moreover, some problematic statements, like nested deontic constraints, are no longer expressible.

Several variations and extensions to the Meyer work have been presented. For example, Broersen [Bro03] describes other possible formulations of dynamic deontic logic where different violation constants are used for defining obligation and permission, rejecting in this way the interdefinibility of these operators as given by Meyer. In particular, Broersen uses a version of relative complement in his logic, arguing that this approach is more appropriate for computer science.

A different approach to goal norms is the so called *process norms* in [Bro03]. Here the norms do not only take into account what happens in the resulting state of an action, but also what happens during its execution. Basically, in this approach every step during an execution of an action must be allowed for this action to be allowed (and similarly for the other deontic constructions). This avoids paradoxes like the *shooting president* one introduced above. However, in the case of atomic actions (actions which are not obtained by means of combinations of other action), goal norms and process norms coincide [Bro03]. A related approach is presented in [vdM96], where instead of considering a relational interpretation of actions, actions are interpreted as sequences of states (note that this is similar to the approach that we called algebraic semantics; however, in this case labels of transition are not used). Also, in this approach there is a strong relationship between modalities and deontic predicates: if an action cannot be executed in a given state, then this action is allowed in this state.

Another approach to *ought-to-do* deontic logic is presented in [KM85], where a modal action logic with deontic operators is described. The operators over actions considered in this approach are: ; (sequence), $\sqcap$ (parallel execution) and $\sqcup$ (non-deterministic choice). The logic uses a constant n (normativeness) to define permission (which plays a similar role to the violation constant of Meyer), with the difference that permission is defined as $\mathrm{n} \rightarrow \mathsf{P}(A, \alpha) \leftrightarrow [\alpha]\mathrm{n}$. n is intended to indicate which states are free of errors or are "normal" states. This conditional definition says that, in normative states, allowed actions are those which yield normal states. This approach seems more acceptable for our purposes since in error states the characterization of permission by means of modalities is not present.

A modal action system (called MAL) with a similar flavour is investigated by the FOREST project in [Ken91, KQM91, Mai87]. Here complement and intersection are considered, and some examples of applications are developed. Although the state normativeness and permission are interpreted as semantically different concepts, a similar relation to those imposed by Meyer between permission and state properties is introduced in the semantics. In this work a partial proof system is presented, although its properties are not investigated in detail. An interesting feature introduced in that work is that actions are interpreted as a set of "events"; intuitively the set of events that this action produces during its execution (this is different from the relational approach, and also to the process approach where an action is interpreted as a sequence of states). We follow this approach throughout this thesis, although we define an algebra over the events and we take advantage of the underlying structure of this algebra. Considering events in the semantics is particularly useful for the semantics of concurrency, where the parallel execution of different actions may generate different events, and furthermore we can distinguish between local events (events produced by the systems) and environment events (events produced by the environment). We use these ideas to propose a way of modularizing the logic, see chapter 7. Our approach is related to the ideas introduced in [RFM91], where category theory is used to structure a first-order version of MAL [KQM91].

Finally, let us introduce the approach described in [FM92] where the notions of permission and obligation are not reduced to modalities and violation constants. In this work deontic predicates are used to define normative trajectories; these are sequences of actions where only permitted actions are executed, and obligations are eventually fulfilled. This work presents interesting ideas to combine deontic specifications with temporal logic, although the relationships between the deontic predicates and the operators over actions are not investigated by the authors.

From the point of view of fault-tolerance, *goal norms* are not appropriate since we want to distinguish between *recovery actions* and *permitted actions*. In the Meyer and Broersen work, permitted actions yield an error-free state; however, in computing

systems this is only true when a recovery action is executed; permitted actions might carry forward violations until a recovery action is executed. Using similar arguments, Sergot rejects goal norms [SC06]. In chapter 3 we present a deontic logic where the prohibition or permission to execute transitions are not directly related to modalities or violation constants, which we argue gives us more flexibility at the time of specifying fault-tolerant systems. We explain the differences between our logic and the logics introduced here in chapters 3 and 4.

2.3 Logic in General

Sometimes, a more abstract view of logical systems is useful, in particular when we need to reason about different logical systems and their relationship. Goguen and Burstall in [GB92] propose an abstract definition of logical system and they call it *Institutions*. We introduce Institutions below. Before that, we need to review the basic definitions of *category theory* since Institutions and the related approaches are defined in a categorical way.

In the words of Saunders Mac Lane [Mac98] (one of the founders of category theory): *Category theory starts with the observation that many properties of mathematical systems can be unified and simplified by a presentation with diagrams of arrows.* Then, basically category theory is the study of the properties of diagrams of arrows. A category is defined as follows [MM92].

Definition 1. *A category consists of a collection of objects (denoted by letters $A, B, C, ...$) and a collection of morphisms (or maps or arrows) (denoted by letters $f, g, h, \ldots$) and four operations:* $\mathbf{dom}(f)$ *returns an object for every arrow f (its domain),* $\mathbf{ran}(f)$ *returns an object for every arrow f (its range). (We write $f : A \rightarrow B$ if A is the domain of f and B is its range.) We have an operation $\mathbf{1}_{\_}$ which, given an object A, returns an arrow $\mathbf{1}_A : A \rightarrow A$ called its identity and we have a binary operation $\circ$, which given two arrows $f : A \rightarrow B$ and $g : B \rightarrow C$ returns an arrow $g \circ f : A \rightarrow C$ (its composition). These operations satisfy the following laws:*

Unit law: *For all arrows $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition with the identity arrow $\mathbf{1}_B$ gives: $\mathbf{1}_B \circ f = f$ and $g \circ \mathbf{1}_B = g$.*

Associativity: *For all arrows: $f : A \rightarrow B$, $g : B \rightarrow C$ and $h : C \rightarrow D$, we have the equality:*

$$f \circ (g \circ h) = (f \circ g) \circ h.$$

□

The dual category of a category $\mathbf{C}$ is denoted $\mathbf{C}^{op}$ and obtained by reversing its arrows. Many of the mathematical structures that are widely used in computer science are indeed categories; here we give a (partial) list:

- **Set**, the category of *small* sets; it contains as objects all the small sets (i.e., in the Gödel-Bernays set theory [Ber91] these are sets which are not classes) and total functions between them.
- **Grp**, the category of groups; its objects are all the *small* groups and its arrows are all the group homomorphisms.
- **Graph**, the category of graphs; its objects are graphs, and its arrows are graph homomorphisms.
- **CPO**, the category of complete partial orders with continuous functions as arrows.
- Any deductive system where objects are formulae and the arrows proofs; in this case note that the proof system must be reflexive (from a formula we can prove this formula) and transitive.

As explained above, in category theory it is usual to reason about diagrams (collections of arrows and objects). To introduce formally the notion of diagram, we need to introduce functors. A functor is, roughly speaking, a mapping between categories which preserves structure. The formal definition of functor is as follows [MM92].

Definition 2. *Given two categories C and D, a functor is an operation F which assigns to each object C of $\mathbf{C}$ an object $F(C)$ of $\mathbf{D}$, and to each morphism f of $\mathbf{C}$, a morphism $F(f)$ of $\boldsymbol{D}$, in such a way that F respects the domain and codomain, as well as the identity and the composition. that is:*

- $\mathbf{dom}(F(f)) = F(\mathbf{dom}(f))$.
- $\mathbf{ran}(F(f)) = F(\mathbf{ran}(f))$.
- $\mathbf{1}_{F(C)} = F(\mathbf{1}_C)$.
- $F(g \circ f) = F(g) \circ F(f)$.

□

We use for functors the same notation as for arrows, i.e., we write $F : \mathbf{C} \to \mathbf{D}$ when F is a functor from category $\mathbf{C}$ to category $\mathbf{D}$.

Now, we can define a *diagram* of a category $\mathbf{C}$ as a functor $D : \mathbf{I} \to \mathbf{C}$, where $\mathbf{I}$ is sometimes called the "indexing category" [MM92]. (Intuitively, D says how the "figure" $\mathbf{I}$ is projected into $\mathbf{C}$.) Sometimes it is required to compare two functors. In these cases, we can use the notion of *natural transformation* [Mac98].

Definition 3. *Let F and G be two functors from a category $\mathbf{C}$ to a category $\mathbf{D}$. A natural transformation α from F to G (denoted by $\alpha : F \stackrel{\cdot}{\to} G$) is an operation associating with each object C of $\mathbf{C}$ a morphism $\alpha_C : F(C) \to G(C)$, in such a way that, for any morphism $f : C' \to C$ in $\mathbf{C}$ we have:*

$$G(f) \circ \alpha_{C'} = \alpha_C \circ F(f).$$

□

Given a natural transformation α, the condition of naturality means that the diagram of figure 2.3 commutes. In category theory the concept of *universal construction*

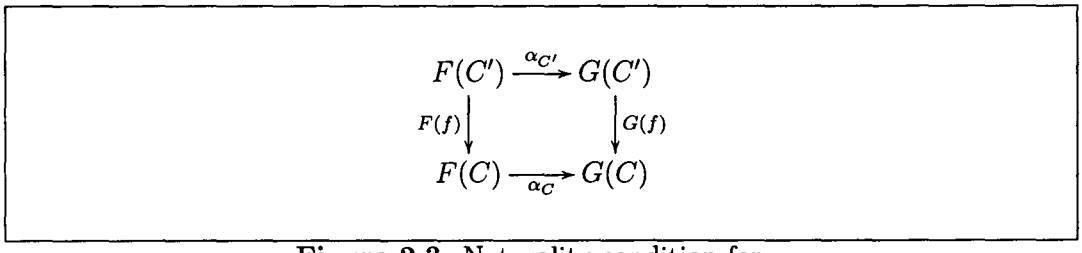


Figure 2.3: Naturality condition for α

is important. A universal construction is a construction (made up of objects and morphisms) which is characterized up to isomorphism by means of its relationship with the other related constructions in the category. One of the nice properties of universal constructions is that it is not necessary to "look at" the structure of the objects or morphisms in the entity to be characterized; we only take into account how this entity is related with the rest of the entities in the category. These kinds of constructions are useful in computer science where systems are often composed of different modules or components and it is too complicated to look inside every one of them. The concepts of *initial objects* and *terminal objects* are, perhaps, the simplest universal constructions. An *initial object* in a category is an object from which we have a unique arrow to any other object in the category; it is not hard to prove that initial objects are unique up to isomorphism. *Terminal objects* are the dual concept to initial objects; a terminal object is one for which we have a unique arrow from any object to this one. For example, the initial object in **Set** is the empty set, and the terminal objects are the singleton sets (which are isomorphic to each other in **Set**).

In particular, we are interested in the notion of *colimit*, which we use to combine specifications (see chapter 7). (Colimits are widely used in computer science literature to combine theories, designs, graphs, etc. See [Fia05] for examples.) Given a diagram $D : \mathbf{I} \rightarrow \mathbf{C}$, a *cocone* is an object C of $\mathbf{C}$ together with a family of $\mathbf{C}$-morphisms: $\{f_i : D(i) \rightarrow C \mid \text{for every object } i \text{ of I}\}$ (denoted by $f : D \rightarrow C$), such that for every morphism $u : i \rightarrow j$ of I we have $f_j \circ D(u) = f_i$, i.e., the diagram of figure 2.4 commutes. A *colimit* of D is a cocone $f : D \rightarrow C$ such that for every other cocone $f' : D \rightarrow C'$, we have a unique arrow $t : C \rightarrow C'$ in $\mathbf{C}$, such that, for every object i of $\mathbf{I}$, we have: $t \circ f_i = f'_i$.

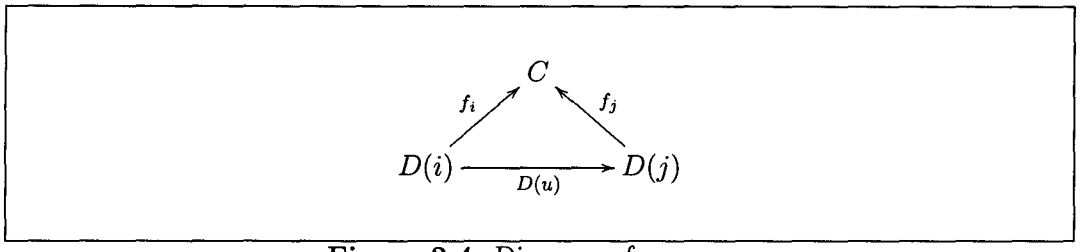


Figure 2.4: Diagram of a cocone

Many universal constructions can be obtained as colimits of special kinds of diagrams [Mac98]. For example, initial objects are colimits of the empty diagram. Another useful universal construction is a pushout. Given two morphisms of $\mathbf{C}$ $f : C \rightarrow C_1$ and $g : C \rightarrow C_2$, a pushout of $\langle f, g \rangle$ is given by two morphisms $u : C_1 \rightarrow P$ and $v : C_2 \rightarrow P$ which satisfies $u \circ f = v \circ g$ (i.e., the left diagram of figure 2.5 commutes), and for any other pair of morphisms $u' : C_1 \rightarrow P'$ and $v' : C_2 \rightarrow P'$ such that $u' \circ f = v' \circ g$, we have a unique arrow $t : P \rightarrow P'$ with $t \circ u = u'$ and $t \circ v = v'$; this is illustrated by the right diagram of figure 2.5. Pushouts of a category $\mathbf{C}$ are colimits of a diagram $D : \mathbf{I} \rightarrow \mathbf{C}$ where $\mathbf{I}$ is a category which looks exactly just like $\bullet \leftarrow \bullet \rightarrow \bullet$. In particular we are interested in colimits of finite diagrams, i.e.,

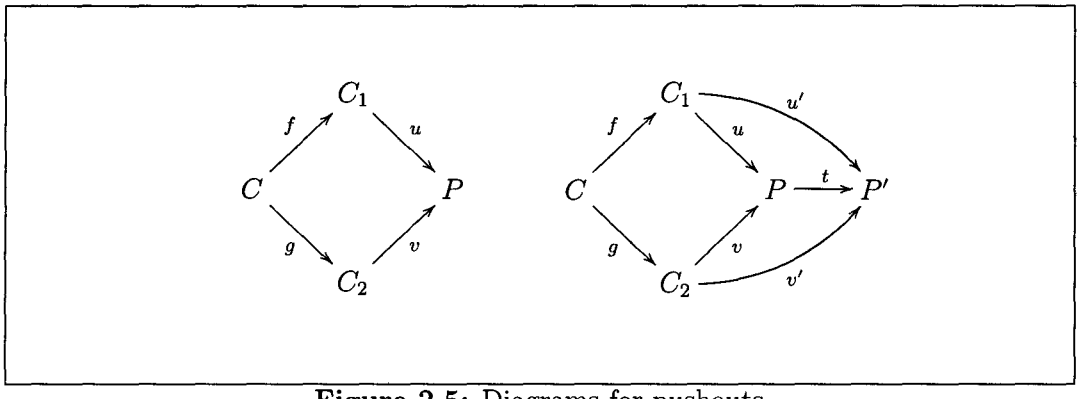


Figure 2.5: Diagrams for pushouts

diagrams $D : \mathbf{I} \to \mathbf{C}$ where $\mathbf{I}$ is finite. If a category has pushouts and initial objects, then it has any finite colimit [AHS09]. Categories which have any finite colimit are called finitely cocomplete, and, if they have any colimit in general, they are called cocomplete.

Here we only reviewed the concepts needed for the following chapters. For a detailed introduction to category theory with examples oriented to computer science and software engineering, the reader can consult [Fia05, BW95].

Institutions are a categorical view of logical systems consisting of an abstract formulation of grammar and a general notion of satisfaction. The formal definition of an Institution is as follows [GB92].

Definition 4. *An Institution is given by:*

- *A category of signatures* $\mathbf{Sig}$.
- *A functor defining the grammar* $\mathcal{G} : \mathbf{Sig} \to \mathbf{Set}$.
- *A functor:* $Sem : \mathbf{Sig} \to \mathbf{Cat}^{op}$.
- *A function* $\models$, *which, given a signature* Σ, *returns a relation* $\models_\Sigma \subseteq |Sem(\Sigma)| \times \mathcal{G}$, *such that, for each arrow* $\sigma : \Sigma \to \Sigma'$, *the following condition holds (**satisfaction condition**):*
  - $M' \models_{\Sigma'} \mathcal{G}(\sigma)(\varphi)$ *iff* $Sem(\sigma)(M') \models_\Sigma \varphi$.

□

Intuitively, the category $\mathbf{Sig}$ defines the possible languages of our logical system, and how they are related to each other. The functor $\mathcal{G}$ defines how the formulae are built from the basic components of the language (usually by a recursive definition) and, given a translation between languages, this functor extends this translation to a translation between formulae. The functor Sem defines the structures that conform to the semantics or interpretation of a language and the relation $\models$ captures the notion of satisfaction (where $\mathbf{Cat}^{op}$ is the opposite category of the category of all small categories and functors between them). The satisfaction condition requires that the notion of truth must be invariant with respect to language change.

Several logical system are Institutions [GB92]; examples are: *propositional logic*, *first-order logic*, *equational logic* and *order-sorted logic*. The notion of morphism between Institutions can be defined, and therefore we can translate results of one

logic to other logics (once we have a suitable morphism). For example, we can prove that equational logic is sound with respect to first-order logic with equality [GB92], and therefore the theorems of equational logic can be translated to first-order logic with equality, i.e., instead of using a unique logical system we can combine them.

Institutions take a semantical view of logic. Another approach is introduced in [FS87] where π-Institutions are defined. π-Institutions capture the idea of logical consequence or entailment systems (i.e., they consider a syntactic formalization of logical systems). Formally, a π-Institution is defined as follows:

Definition 5. *A π-Institution consists of:*

- *A category* **Sign**.
- *A functor* $Gram : \mathbf{Sign} \rightarrow \mathbf{Set}$.
- *For every object* Σ *of* **Sign**, *a relation* $\vdash\colon 2^{Gram(\Sigma)} \times Gram(\Sigma)$ *is defined satisfying the following properties:*
  - *For every* $p \in Gram(\Sigma)$, $p \vdash_\Sigma p$.
  - *For every* $p \in Gram(\Sigma)$ *and* $\Phi, \Phi' \subseteq Gram(\Sigma)$, *if* $\Phi \subseteq \Phi'$ *and* $\Phi \vdash_\Sigma p$, *then* $\Phi' \vdash_\Sigma p$.
  - *For every* $p \in Gram(\Sigma)$ *and* $\Phi, \Phi' \subseteq Gram(\Sigma)$, *if* $\Phi \vdash_\Sigma p$ *and* $\Phi' \vdash_\Sigma p'$ *for every* $p' \in \Phi$, *then* $\Phi' \vdash_\Sigma p$.
  - *For every* $\sigma : \Sigma \rightarrow \Sigma'$, $p \in Gram(\Sigma)$ *and* $\Phi \subseteq Gram(\Sigma)$, $\Phi \vdash_\sigma p$ *implies* $Gram(\sigma)(\Phi) \vdash_\Sigma Gram(\sigma)(p)$.

□

Here the notion of consequence is presented by means of its principal properties. Following Tarski [Tar56], a logical consequence must satisfy reflexivity, monotonicity and transitivity (or cut). However, these properties can be relaxed (for example, non-monotonic logics are widely used in computer science). Interesting features of Institutions and π-Institution arise when we consider theories, i.e., pairs $\langle \Sigma, T \rangle$ where Σ is an object of **Sign** and $T \subseteq \mathcal{G}(\Sigma)$, where usually the set T must satisfy closure properties with respect to $\vdash_\Sigma$, i.e., if $\Phi \vdash_\Sigma p$, then $p \in \Phi$. Theories can be used to specify software or components of systems [MT84], and then using the categorical constructions we can put together different specifications. The principal idea comes from [BG77] where large theories can be built from smaller theories using colimits. The colimits of a diagram of theories and morphisms between them is, in some sense, a theory in which every theory is included (by means of the morphisms of the cocone),

and is minimal in the sense that it is the initial object which satisfies this requirement. We can consider the category **Th** of theories of a given logic. Obviously, we have a forgetful functor (see [Mac98]) U : **Th** $\rightarrow$ **Sig**. The important fact is that this functor *reflects* colimits (see [AHS09]), and this implies that, if the category of signatures is finitely cocomplete, then the category of theories is finitely cocomplete. Putting theories together is fundamental when specifying systems; it is not viable to specify large systems in one complex and large specification. Instead, dividing it in several parts and establishing how the different parts are related is a more practical option, which coincides with the usual view in software engineering. Fiadeiro and Maibaum used this idea, but using temporal theories which are more adequate to specify concurrent and reactive system [FM92].

It is important to note that, in π-Institutions we have that morphisms preserve theorems, i.e., $\Gamma \vdash_{\Sigma} p \Rightarrow \text{Gram}(\sigma)(\Gamma) \vdash_{\sigma(\Sigma)} \text{Gram}(\sigma)(p)$, which implies that, if we have a diagram of theories and we obtain its colimit, then the colimit object preserves all the theorems of the smaller theories in the diagram. This undoubtedly is an interesting property.

2.4 Summary

In this chapter we have introduced the basic concepts that we use throughout this thesis. We have introduced the standard definitions of *propositional logic*, *modal logic*, *temporal logic*, *deontic logic* and *general logics*. These concepts will be widely used in the following chapters. We adopt the view that logical systems provide a powerful theoretical framework to develop reliable software. Furthermore, category theory, a mathematical analysis of properties of diagrams of arrows, allows us to reason on an abstract level about systems. Moreover, using standard constructions of category theory, we can compose and put specifications together. It is interesting to remark that sometimes we abstract from the details of the logical systems and we only use the main properties that define them. This coincides with the view taken by Institutions and π-Institutions.

Chapter 3

A Deontic Action Logic

In this chapter we present a logic which will be a cornerstone for the rest of this thesis. This logic is somewhat different from other well-known deontic logics, and these differences imply some theoretical and practical benefits when specifying fault-tolerant software. The main difference between the logic presented in this chapter and the dynamic deontic logics presented in chapter 2 is that the formalism described here uses an algebraic interpretation of actions, which allows us to use the structure of the generated algebra to label the transitions on the models. Meanwhile, most other logics, as explained in chapter 2, use a relational interpretation of actions, following the style of Dynamic Logic [HKT00]; the algebraic approach allows us to to take advantage of the properties of the underlying algebra to prove meta logical properties over the logic (e.g., completeness).

A preliminary version of the logic was presented in [CM07a, CM07b, CM09]. Here we go into the details of the formalism and the proofs. The logic has some innovative features, as compared to extant versions of deontic logic. For example, because we want to do various forms of automated analysis of specifications, such as model checking, we want our logic to have appropriate meta theorems. So, our logic is not just sound and complete, but also decidable and compact (strongly complete). This is an improvement on the corresponding logic developed by Broersen [Bro03]. This is achieved by means of a number of interesting features. For example, although the idea of distinguishing weak and strong versions of the permission operator were suggested earlier, our formulation enables us to interrelate them in a new and novel manner. The two versions of permission have an existential and universal character, asserting that there is some context for doing an action from the present state, for weak permission, and that every context for doing that action from the present state is allowed, for strong permission, respectively. This notion of "context" for actions is

captured by using the semantics proposed in [KQM91], interpreting an action as the set of *events* in which the action "participates". This also supports our adoption of an open semantics for our specifications (see [Bar87]), in which the environment of the system we are describing may be performing actions in parallel with the system. See [FM92] for an extensive discussion of such open semantics; as in the referenced work, we will eventually want to adopt the idea of specifying system behaviour in parts, in terms of components, and then to combine such components, thus making more concrete the environment for each component in the combination.

Further, though we allow many of the usual combinators on actions, we adopt a restricted version of the complement operator (do something other than the referenced action), interpreting it locally in the state in which the complement is evaluated, instead of globally with respect to all possible actions built from available atomic actions and combinators. These features allow us to characterize the domain of actions, built from basic actions and combinators, as an atomic boolean algebra [Sik69]. It is this characterization that leads to the nice meta properties of the logic that we obtain.

The formal framework defined in the following sections has been influenced by various past ideas; for example, the obligation operator (and its properties) are similar to those defined in [KQM91] and [KM85], though in those works the obligation is not defined using two variants of permission as we do in this work.

As explained above, we are interested in using automatic techniques (like model checking) with our logic; this implies that some properties, such as decidability, are required. Moreover, for expressing properties inherent to fault-tolerance, we need to be able to express temporal assertions, recovery actions, and permission and obligation predicates on actions. The temporal extension of the logic uses some concepts from [Mai93, FM91a], in particular the given semantics using *traces*. Finally, the weak permission operator (see the next section) has similar properties to that defined in [Mey88], though its relationship with the normal, so called strong permission, is new. We shall compare our work with these frameworks at various points in what follows. However, note that some novel properties of the deontic operators will be given (e.g., axiom **A12** below), and the definition of obligation given in section 3.1 is slightly, but significantly, different from those in the literature.

This chapter is organized as follows. In the next section we introduce the basic definitions of the logic, including its syntax and semantics. In section 3.3 a deductive system is described, and some meta theorems of the logic are proven; in particular, we prove the completeness of this deductive system. In section 3.5 we extend the propositional system with temporal notions, and we propose an axiomatic system for this new logical system, which is proved to be sound with respect to the proposed

semantics. (We prove its completeness in chapter 6.) We present some small examples of specification at the end of the chapter, but we leave the description of more complex examples to chapter 4.

3.1 Concocting a Propositional Deontic Logic

As usual, we start defining a propositional version of deontic logic (for the sake of brevity we call it DPL) by introducing its syntax and semantics. DPL is a *modal action logic*, which uses boolean operators for combining action terms. Here, we follow the approach proposed in [KMQ93], in the sense that actions are interpreted as a set of "*events*" (transition labels), in contrast to what is usually done in modal logics and dynamic logics, in particular, where actions are interpreted as relations and action combinators are interpreted as relational operations. We explain later on some benefits that the approach taken here has.

After defining the key components of DPL, we present an axiomatic system which has some similarities to those given for *Dynamic Propositional Logic* [Mey88] and *Modal Boolean Logic* [GP90]. Finally, we prove the soundness of the resulting deductive system.

Definition 6 (language). *A language (or vocabulary) for DPL is a tuple:* $\langle \Phi_0, \Delta_0 \rangle$, *where:*

- Φ_0 *is an enumerable set of propositional letters; we will denote them:* $p_1, p_2, \ldots$.
- Δ_0 *is a (finite) set of primitive actions, denoted by:* $a_1, \ldots, a_m$.
- Φ_0 *and* Δ_0 *are mutually disjoint.*

□

Using the sets Φ_0 and Δ_0, we can define the set of action terms and formulae of a given language.

Definition 7 (action terms). *Given a vocabulary* $\langle \Phi_0, \Delta_0 \rangle$, *we define the set of action terms (called* Δ*) as follows:*

- $\Delta_0 \subseteq \Delta$.

- $\emptyset, \mathbf{U} \in \Delta$.
- *if* $\alpha, \beta \in \Delta$, *then* $\alpha \sqcup \beta \in \Delta$ *and* $\alpha \sqcap \beta \in \Delta$.
- *if* $\alpha \in \Delta$, *then* $\overline{\alpha} \in \Delta$.
- *No other expression belongs to* Δ.

□

We use the Greek letters: $\alpha, \beta, \gamma, ...$ for term variables over Δ. $\emptyset$ and $\mathbf{U}$ are constant action symbols: $\emptyset$ denotes an impossible action and $\mathbf{U}$ denotes the action obtained from the non-deterministic choice between all the actions in the language. In a similar way we define the set of well-formed formulae.

Definition 8 (Formulae). *Given a vocabulary:* $\langle \Phi_0, \Delta_0 \rangle$, *we define the set of well-formed formulae* (Φ) *as follows:*

- $\Phi_0 \subseteq \Phi$.
- $\top, \bot \in \Phi$.
- *if* $\alpha, \beta \in \Delta$, *then* $\alpha =_{act} \beta \in \Phi$.
- *if* $\varphi_1, \varphi_2 \in \Phi$, *then* $\varphi_1 \rightarrow \varphi_2 \in \Phi$.
- *if* $\varphi \in \Phi$, *then* $\neg\varphi \in \Phi$.
- *if* $\varphi \in \Phi$ *and* $\alpha \in \Delta$, *then* $\langle\alpha\rangle\varphi \in \Phi$ *and* $[\alpha]\varphi \in \Phi$.
- *if* $\alpha \in \Delta$ *then* $\mathsf{P}(\alpha) \in \Phi$, $\mathsf{P}_{\mathsf{w}}(\alpha) \in \Phi$ *and* $\mathsf{O}(\alpha) \in \Phi$.
- *No other expression belongs to* Φ.

□

As usual, we can define some derived operators:

- $\phi \vee \psi \overset{\mathsf{def}}{\Longleftrightarrow} (\neg\phi) \rightarrow \psi$.
- $\phi \wedge \psi \overset{\mathsf{def}}{\Longleftrightarrow} \neg(\neg\phi \vee \neg\psi)$.

We call $\mathsf{P}(-)$ permission or *strong permission*, whereas $\mathsf{P}_\mathsf{w}(-)$ is *weak permission*; the differences between the two will become evident with their semantic definitions. Including both versions of permission gives us some freedom in specifying systems, and it is interesting that the two operators are related in a strong way, as we show later on.

The obligation operator will be defined using the two versions of permission, instead of taking the usual definition: $\mathsf{O}(\alpha) \equiv \neg\mathsf{P}_\mathsf{w}(\overline{\alpha})$. We define (by means of axioms, see below):

$$\mathsf{O}(\alpha) \equiv \mathsf{P}(\alpha) \wedge \neg\mathsf{P}_\mathsf{w}(\overline{\alpha}).$$

We will explain this definition later. Before this, we need to introduce the concept of semantic structures.

Definition 9 (structures). *Given a language $L = \langle \Phi_0, \Delta_0 \rangle$, an L-structure is a tuple $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ where:*

- *$\mathcal{W}$ is a set of worlds.*
- *$\mathcal{E}$ is a non-empty set of (names of) events.*
- *$\mathcal{R}$ is an $\mathcal{E}$-labeled relation between worlds. We require that, if $(w, w', e) \in \mathcal{R}$ and $(w, w'', e) \in \mathcal{R}$, then $w' = w''$, i.e., $\mathcal{R}$ is functional when we fix the third element in the tuple.*
- *$\mathcal{I}$ is a function:*
  - *For every $p \in \Phi_0 : \mathcal{I}(p) \subseteq \mathcal{W}$.*
  - *For every $\alpha \in \Delta_0 : \mathcal{I}(\alpha) \subseteq \mathcal{E}$.*

  In addition, the interpretation $\mathcal{I}$ has to satisfy the following properties:

  I.1 *For every $\alpha_i \in \Delta_0$: $|\mathcal{I}(\alpha_i) - \bigcup\{\mathcal{I}(\alpha_j) \mid \alpha_j \in (\Delta_0 - \{\alpha_i\})\}| \leq 1$.*

  I.2 *For every $e \in \mathcal{E}$: if $e \in \mathcal{I}(\alpha_i) \cap \mathcal{I}(\alpha_j)$, where $\alpha_i \neq \alpha_j$ and $\alpha_j, \alpha_i \in \Delta_0$, then: $\cap\{\mathcal{I}(\alpha_k) \mid \alpha_k \in \Delta_0 \wedge e \in \mathcal{I}(\alpha_k)\} = \{e\}$.*

  I.3 *$\mathcal{E} = \bigcup_{\alpha_i \in \Delta_0} \mathcal{I}(\alpha_i)$.*
- *$\mathcal{P} \subseteq \mathcal{W} \times \mathcal{E}$, is a relation which indicates which event is permitted in which world.*

□

We can extend the function $\mathcal{I}$ to well-formed action terms and formulae, as follows:

- $\mathcal{I}(\neg\varphi) \stackrel{\text{def}}{=} \mathcal{W} - \mathcal{I}(\varphi)$.
- $\mathcal{I}(\varphi \rightarrow \psi) \stackrel{\text{def}}{=} \mathcal{I}(\neg\varphi) \cup \mathcal{I}(\psi)$.
- $\mathcal{I}(\alpha \sqcup \beta) \stackrel{\text{def}}{=} \mathcal{I}(\alpha) \cup \mathcal{I}(\beta)$.
- $\mathcal{I}(\alpha \sqcap \beta) \stackrel{\text{def}}{=} \mathcal{I}(\alpha) \cap \mathcal{I}(\beta)$.
- $\mathcal{I}(\overline{\alpha}) \stackrel{\text{def}}{=} \mathcal{E} - \mathcal{I}(\alpha)$.
- $\mathcal{I}(\emptyset) \stackrel{\text{def}}{=} \emptyset$.
- $\mathcal{I}(\mathbf{U}) \stackrel{\text{def}}{=} \mathcal{E}$.

Conditions **I.1** and **I.2** in definition 9 express some requirements on the possible interpretations of primitive actions. **I.1** says that *the isolated application of an action always generates at most one event*; otherwise we will have an undesired nondeterminism in our models, as the different ways of executing a primitive action should arise only because you can execute it together with other actions (perhaps environmental actions). **I.2** establishes that *if an event is a result of the execution of two or more actions, then the concurrent execution of all the actions which generate it will give us only this event.* This condition also ensures that a weird nondeterminism will not occur, in the sense that the existence of nondeterminism must be grounded on the combination of different sets of actions with environmental events; that is, an action can have different behaviours because several different environment (or system) events may happen during its execution, and this is the only cause of the action's nondeterminism. Condition **I.3** says that all the events are generated by the actions of the vocabulary; we revisit this condition in chapter 7 where we extend the models with "external" events.

As explained in [SC06], a useful way of thinking about the semantic structures is seeing them as coloured Kripke structures, where a given transition $w \xrightarrow{e} w'$ is coloured with green if it is allowed (i.e., when $(w, e) \in \mathcal{P}$), and coloured with red otherwise. This allows us to distinguish visually between forbidden and allowed transitions.

Some notation is needed for dealing with the relational part of the structure: we will use the notation $w \xrightarrow{e} w'$ when $(w, w', e) \in \mathcal{R}$. For a given $e \in \mathcal{E}$, we define the relation $\mathcal{R}_e = \{(w, w') \mid (w, w', e) \in \mathcal{R}\}$. Also, given a $w \in \mathcal{W}$ we define: $\mathcal{P}_w = \{e \in \mathcal{E} \mid (w, e) \in \mathcal{P}\}$. These definitions will be useful in the following sections. When convenient we use the relationship $\mathcal{P}$ as a predicate, i.e., $\mathcal{P}(w, e) \stackrel{\text{def}}{\Longleftrightarrow} (w, e) \in \mathcal{P}$. Let us introduce the relation $\models^L$ between models and formulae.

Definition 10 ($\models$). *Given a vocabulary $L = \langle \Phi_0, \Delta_0 \rangle$ and a L-structure $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$, we define the relation $\models^L$ between worlds and formulae as follows:*

- $w, M \models^L p \overset{\text{def}}{\iff} w \in \mathcal{I}(p)$.
- $w, M \models^L \alpha =_{act} \beta \overset{\text{def}}{\iff} \mathcal{I}(\alpha) = \mathcal{I}(\beta)$.
- $w, M \models^L \neg\varphi \overset{\text{def}}{\iff}$ *not* $w, M \models \varphi$.
- $w, M \models^L \varphi \rightarrow \psi \overset{\text{def}}{\iff} w, M \models^L \neg\varphi$ *or* $w, M \models^L \psi$ *or both.*
- $w, M \models^L \langle\alpha\rangle\varphi \overset{\text{def}}{\iff}$ *there exists some* $w' \in \mathcal{W}$ *and* $e \in \mathcal{I}(\alpha)$ *such that* $w \xrightarrow{e} w'$ *and* $w', M \models^L \varphi$.
- $w, M \models^L [\alpha]\varphi \overset{\text{def}}{\iff}$ *for all* $w' \in \mathcal{W}$ *and* $e \in \mathcal{I}(\alpha)$, *if* $w \xrightarrow{e} w'$, *then* $w', M \models^L \varphi$.
- $w, M \models^L \mathsf{P}(\alpha) \overset{\text{def}}{\iff}$ *for all* $e \in \mathcal{I}(\alpha)$, $\mathcal{P}(w, e)$ *holds.*
- $w, M \models^L \mathsf{P_w}(\alpha) \overset{\text{def}}{\iff}$ *there exists some* $e \in \mathcal{I}(\alpha)$ *such that* $\mathcal{P}(w, e)$
- $w, M \models^L \mathsf{O}(\alpha) \overset{\text{def}}{\iff}$ *for all* $e \in \mathcal{I}(\alpha)$, $\mathcal{P}(w, e)$ *holds, and for every* $e' \in \mathcal{E} - \mathcal{I}(\alpha)$, *we have* $\neg\mathcal{P}(w, e')$.

□

When no confusion is possible, we write $\models$ instead of $\models^L$. We have not defined the satisfaction condition for the box modality and the obligation predicate. Note that they can be defined by means of the other operators; see the axiomatic system presented below. As usual, we say that $M \models^L \varphi$ iff, for all worlds $w \in \mathcal{W}$, we have: $w, M \models^L \varphi$. And we say: $\models^L \varphi$, if $M \models^L \varphi$ for all models M. Some intuition about each operator is useful:

- $\langle\alpha\rangle\varphi$, *there is a way of executing* α *so that* φ *holds in the next world.*
- $[\alpha]\varphi$, *after executing* α *in any possible way,* φ *will hold, this formula is equivalent to* $\neg\langle\alpha\rangle\neg\varphi$
- $[\alpha_1 \sqcup \alpha_2]\varphi$, *every way of executing* α_1 *or* α_2 *leads to* φ.
- $[\overline{\alpha}]\varphi$, *after executing an action different from* α, φ *holds.*
- $[\alpha_1 \sqcap \alpha_2]\varphi$, *every way of executing both* α_1 *and* α_2 *leads to* φ.

- $\mathsf{P}(\alpha)$, *all the different ways of executing* α *are allowed.*
- $\mathsf{P}_{\mathsf{w}}(\alpha)$, *there is at least one way of executing* α *which is allowed.*

Note the symmetric definitions of strong and weak permission, reflecting a universal/existential symmetry.

Some comments are in order regarding the definition of obligation. First, note that the permission predicates allow us to partition the transitions of the semantic structures into "allowed" transitions (or "green" transitions) and "not-allowed" transitions (or "red" transitions). For us, an obligatory action is one which is the only one acceptable (i.e., its execution from a given state, in any way, produces a green transition) and the execution of any other action is unwanted (although possible); that is, these transitions are red coloured. This intuition is formalized using both versions of permission, as shown above. The standard definition of dynamic deontic logic does not consider obligated actions as allowed (which we think should be the case). Recall the definition of this notion of obligation: $\mathsf{O}_M(\alpha) \stackrel{\mathsf{def}}{\Longleftrightarrow} \neg[\overline{\alpha}]\mathsf{v}$ (we use the symbol O_M to point out that this is the obligation proposed by Meyer). Formally, in our setting we have that $\mathsf{O}_M(\alpha)$ is true in a world w when for every $e \in \mathcal{I}(\overline{\alpha})$ and world w', if $w \xrightarrow{e} w'$, then $w', M \vDash \mathbf{v}$. ($\mathbf{v}$ is a logical constant pointing out that a violation is true.) In this definition of obligation, we have an instance of Ross's paradox, i.e., $\vDash \mathsf{O}_M(\alpha) \rightarrow \mathsf{O}_M(\alpha \sqcup \beta)$. This follows directly from the definition stated above.

Note that the sentence $\mathsf{O}(\alpha) \rightarrow \mathsf{O}(\alpha \sqcup \beta)$ is not a valid formula in our logic; in figure 3.1 we show a counterexample. In this model we have three states w, w'

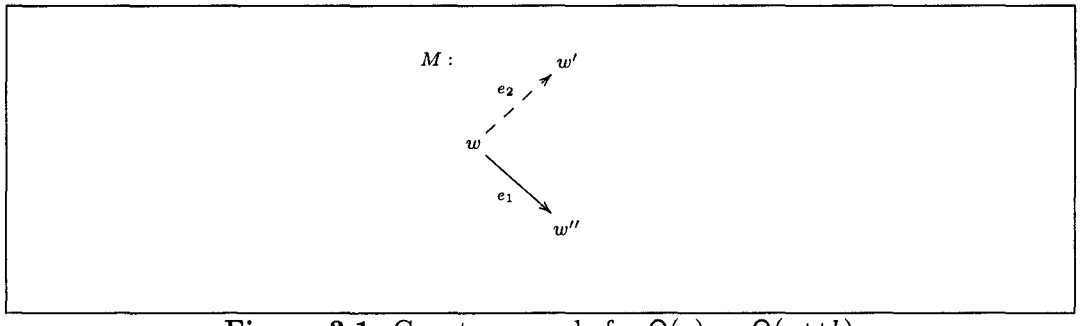


Figure 3.1: Counterexample for $\mathsf{O}(a) \rightarrow \mathsf{O}(a \sqcup b)$

and w''; we use dashed arrows to describe not allowed (red) transitions, and plain arrows for allowed (green) transitions. Consider a vocabulary with two actions a and b; in this model we set $\mathcal{I}(a) = \{e_1\}$ and $\mathcal{I}(b) = \{e_2\}$, and therefore we have $w, M \nvDash \mathsf{O}(a) \rightarrow \mathsf{O}(a \sqcup b)$.

Another option could be to define $\mathsf{O}'(\alpha) \stackrel{\text{def}}{\Longleftrightarrow} \mathsf{P_w}(\alpha) \wedge \neg \mathsf{P_w}(\overline{\alpha})$ (we use the symbol O' to distinguish this variation of obligation from the obligation used in the thesis). The formal semantics of this variation of obligation is as follows. $w, M \models \mathsf{O}'(\alpha)$ iff there exists some $e \in \mathcal{I}(\alpha)$ such that we have $(w, e) \in \mathcal{P}$, and for all $e' \in \mathcal{E} - \mathcal{I}(\alpha)$ we have $(w, e') \notin \mathcal{P}$.

We reject this definition because of underspecification; i.e., this definition says that an obliged action is weakly allowed, and therefore some (not specified) ways of performing it might be forbidden. Our position is that there must be no missing details when we impose an obligation. For example, consider a vocabulary with two actions pr and pb, which are intended to represent the actions of pressing a red button and pressing a blue button, respectively. A possible model of this specification is shown in figure 3.2. In this model, we set $\mathcal{I}(\mathtt{pr}) = \{e_2, e_3\}$ and $\mathcal{I}(\mathtt{pb}) = \{e_1, e_2\}$.

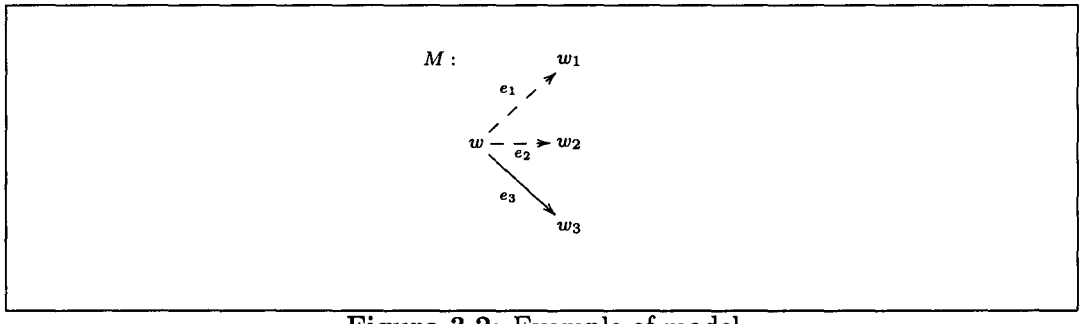


Figure 3.2: Example of model

The dashed arrows illustrate forbidden transitions. In this case we have that $w, M \models \mathsf{O}'(\mathtt{pr})$; however, pressing the red button together with the blue button is forbidden. This variation of obligation does not provide the exact information about the actions that must be performed to fulfil the obligation. This underspecification is avoided by the predicate O(); for example, the semantic structure of figure 3.2 is not a model of $\mathsf{O}(\mathtt{pr})$, since there is some way of pressing the red button that is not allowed. In this case, we have to say explicitly that $\mathsf{O}(\mathtt{pr} \sqcap \overline{\mathtt{pb}})$.

On the other hand, we remark that the obligation introduced above is an immediate obligation, i.e., this operator only predicates about the next transition. As a consequence of this, an obligation is not necessarily kept through time, e.g., if the obliged action is not performed. We think that the dynamics of obligations should be imposed by the designers in their specifications. For example, we can say that, if an obligation is not fulfilled, then we will keep it, with the following formula: $\mathsf{O}(\alpha) \rightarrow [\overline{\alpha}]\mathsf{O}(\alpha)$. This formula says that, if we are in a state w where executing an action α is obliged (in the sense of definition 10), then for any transition $w \xrightarrow{e} w'$ with $e \notin \mathcal{I}(\alpha)$ we have that $w', M \models \mathsf{O}(\alpha)$. In other words, we keep the obligation until we fulfil it.

Another possibility is to say that an obligation to do another action arises (i.e., a contrary to duty obligation) as follows: $\mathsf{O}(\alpha) \rightarrow [\overline{\alpha}]\mathsf{O}(\beta)$. This formula says that, if we are in a state w where the action α is obliged (in the sense of definition 10), then for any transition $w \xrightarrow{e} w'$ with $e \notin \mathcal{I}(\alpha)$, we have that $w', M \vDash \mathsf{O}(\beta)$. That is, when an obligation is not fulfilled, another obligation arises.

The important point here is that these dynamics depend on the scenario to be formalized. In section 3.5 we introduce a temporal extension of the logic. Using the temporal operators, we can define temporal notions of obligation, for example: $\mathsf{A}(\mathsf{O}(\alpha)\ \mathcal{U}\ \varphi)$, which says that we are obliged to perform the action α until φ becomes true. Several deontic temporal notions can also be developed, but we do not investigate this topic in this thesis.

A comment may be useful about the empty action ($\emptyset$); it is an impossible action. Then, why predicate about whether it is permitted or not? In our logic we have $\mathsf{P}(\emptyset)$ and $\neg\mathsf{P}_{\mathsf{w}}(\emptyset)$ (the latter coincides with [Mey88]). From the common sense point of view, this discussion is immaterial; thus, we choose to allow impossible actions since it gives us completeness of our logic. Note that in our semantics both $\mathsf{P}(\emptyset)$ and $\neg\mathsf{P}_{\mathsf{w}}(\emptyset)$ are vacuously true.

3.2 Related Logics.

Several variants of Modal Action Logic with boolean operators (with and without deontic predicates) can be found in the literature. On the one hand, the Boolean Modal Logic introduced by Gargov and Passy in [GP90] does not have deontic operators, and the complement used in their logic is an absolute one; and, as remarked in previous chapters, this is undesirable (in the context of computing systems) because unreachable worlds become reachable using this operator. Note that the complement defined in this chapter is a relative one; that is, for the complement of an action in the scope of a modality we only take into account the enabled transitions in the actual world. Broersen (see [Bro03]) proposes some modifications to the logic of Gargov and Passy, and he presents several variants of relative complement; however, no axiomatic systems for these variants are presented there. Another point to note is that, as proved by Broersen in his thesis, boolean modal logics with relative complement and an infinite number of actions are not compact. The logic presented here is compact (see the next section); this is because having a finite set of actions allows us to obtain an atomic algebra, which can be strongly related to the semantics of the logic (see the completeness proof below). Here it is important to note that the semantics of the action operators is given by means of an algebra of events, in contrast with the cited works where the action operators are interpreted as relational operations. Having a

finite set of actions has other meta theoretical implications, perhaps the most important is that structurality is lost, i.e., change of notation may affect formula validity (see [GB92] and [FS87]), although, perhaps, some weak versions of structurality may be satisfied by this logic (e.g., the weak structurality defined in [FM93]); we come back to these issues in chapter 7 and retrieve the situation.

Furthermore, the deontic part of the logic also has novel features, and (as explained in the introduction) there exists a strong relationship between the two versions of the permission operator, which is expressed by the axioms **A11** and **A12** (see below), and this relationship is important when proving the completeness of the system. Moreover, using the two versions of permission, we define the obligation predicate (as explained above). It is important to remark that an important difference between the logic presented here and dynamic deontic logics ([Mey88] and [Bro03]) is that the deontic operators are independent of modalities. We follow the philosophy introduced in [KM85] in the sense that the prescription and description of systems must be separated concepts. Meyer defines permission using modalities (e.g., $\mathsf{P_w}(\alpha) \equiv \langle\alpha\rangle\neg \mathrm{v}$, for some violation constant v), but this strong relationship is often not desirable in fault-tolerance; for example, violations not only arise from execution of forbidden actions, but they could also be carried forward by allowed actions. (Here, it is important to note the distinction between allowed actions and recovery actions.) Similar arguments are given by Sergot in [SC06], where the deontic component of the language $n\mathcal{C}+$ is presented. In this language, there exists a weak relationship between violations and allowed (or forbidden) transitions; only the so-called Green-Green-Green constraint (or *GGG* for short) is adhered to. In Sergot's words: *from a green state using a green transition we obtain a green state.* We have not included violation constants in the logic. In fault-tolerance (and in other contexts), a different set of violations, with diverse structure, arise in each scenario, and, therefore, which violations exist and how they occur must be defined in each specification by the designers. In chapter 7 we extend the logic presented above with other constructs which are aimed at facilitating the specification of fault-tolerant software. Violation constants are included in the language; however, the independence between modalities and deontic predicates is preserved. On the other hand, the *GGG* constraint can be introduced in the logic as a parameterized (language dependent) logical axiom, i.e., an axiom which must be instantiated for the extra-logical language of each specification (similar to the locality axiom [FM92]). We deal with this in chapter 7, but we note that the characteristics of the logic make it possible to use it to reason about transition systems described with the language $n\mathcal{C}+$.

Finally, the temporal extension of the logic presented in section 3.5 has some similarities to the logic presented in [DK97] (where a temporal extension of dynamic deontic logic is described), but note that there are two important differences: first, the logic described there is an *ought-to-be* logic, i.e., the deontic operators are used

on predicates (in contrast, our logic is an *ought-to-do* logic, i.e., we use the deontic operators to prescribe actions). Second, the temporal logic described in [DK97] is linear and here we introduce a *branching time logic* (see [Eme90]).

3.3 A Deductive System

In this section we present a (*Hilbert style*) deductive system; this is a normal modal system (in the sense that the **K**-axiom can be deduced from it) and the axioms for the box modalities are similar to those given in [GP90] and [HKT00]. Given a language L, we say that $\vdash^L \varphi$ when φ is a theorem of the following axiomatic system (see [BRV01] for the formal definition). Note that we index the deduction with a language since the axiomatic system presented below is dependent on the language used. We follow [LS77] for the definition of the relationship $\vdash^L \subseteq \wp(\Phi) \times \Phi$, i.e., $\Gamma \vdash^L \varphi$ if and only if there exists a sequence of formulae $\varphi_0, \ldots, \varphi_n$ such that $\varphi_n = \varphi$ and for each φ_i $(i < n)$ either $\varphi_i \in \Gamma$, $\vdash^L \varphi_i$, or there exists φ_k and φ_j (where $k, j < i$) such that φ_i is derivable from φ_j and φ_k using *Modus Ponens*. Of course, $\emptyset \vdash^L \varphi$ is equivalent to $\vdash^L \varphi$. Note that with this definition of deducibility, we can only apply the deduction rule **GN** (see below) to theorems (but we cannot use this rule with assumptions). This definition of deduction retains the deduction (meta) theorem, that is: $\Gamma \cup \{\psi\} \vdash^L \varphi$ if and only if $\Gamma \vdash^L \psi \rightarrow \varphi$, see [LS77] for details. However, we need another version of deduction (which can be thought of as being a *global* version of deduction) in which we extend the axiomatic system with additional axioms; we say that $\vdash^L_S \varphi$ if we have $\vdash^L \varphi$ when we add the formulae in S as axioms in the axiomatic system. Note that in this case we can apply the rule **GN** to the formulae in the set S.

An important characteristic of our set of axioms (which, to the author's knowledge, is not shared with previous work) is that it establishes a deep connection between the weak version of permission and the strong version of it. Actually, one of these axioms can be seen as a kind of "*compactness*" property that our models satisfy; this property is implied by the restrictions assumed on the two versions of permission. This is a key fact exploited in the completeness proof. Before going into details, we need to introduce the notions of *canonical action terms* and *boolean algebra of action terms*. In the following we consider a fixed vocabulary:

$$\Phi_0 = \{p_1, p_2, \ldots\} \quad \Delta_0 = \{a_1, \ldots, a_n\}$$

This language induces the set Δ of boolean action terms (see definition 7); we denote by Φ_{BA} some axiomatization of boolean algebras (note that there exist complete axiomatizations, see [Sik69]). Then, the set Δ/Φ_{BA} is the quotient set of the boolean action terms by $=_{act}$ ($=_{act}$ is the congruence defined over actions by the equality

predicate); the point is that, using this set, we can define the (atomic) boolean algebra $\langle \Delta/\Phi_{BA}, \sqcup_{[]}, \sqcap_{[]}, -_{[]}, [\emptyset]_{BA}, [U]_{BA} \rangle$ as follows:

- $-_{[]}[\alpha]_{BA} = [\overline{\alpha}]_{BA}$.
- $[\alpha]_{BA} \sqcup_{[]} [\beta]_{BA} = [\alpha \sqcup \beta]_{BA}$.
- $[\alpha]_{BA} \sqcap_{[]} [\beta]_{BA} = [\alpha \sqcap \beta]_{BA}$.

It is straightforward to prove that this is a boolean algebra. Furthermore, since the terms in Δ are generated by a finite set Δ_0 of primitive actions, the quotient boolean algebra is finite, and therefore atomic. We denote by $at(\Delta/\Phi_{BA})$ (or $at(\Delta)$ when no confusion arises) the set of atoms of the quotient boolean algebra of terms. Note also that we can define $\sqsubseteq_{[]}$ in the usual way.

At this point we are ready to present our axiomatic system.

Definition 11 (Axioms for DPL). *Given a vocabulary $\langle \Phi_0, \Delta_0 \rangle$, where $\Delta_0 = \{a_1, ..., a_n\}$, the axiomatic system is composed of the following axioms:*

1. *The set of propositional tautologies.*
2. *A set of axioms for boolean algebras for action terms (a complete one), including standard axioms for equality.*
3. *The following set of axioms (note that the following are axiom schemas, i.e., the variables α, α', β and γ can be replaced by any action term; and the variables φ and ψ can be replaced by any formula):*

   A1. $[\emptyset]\varphi$

   A2. $\langle\alpha\rangle\varphi \wedge [\alpha]\psi \rightarrow \langle\alpha\rangle(\varphi \wedge \psi)$

   A3. $[\alpha \sqcup \alpha']\varphi \leftrightarrow [\alpha]\varphi \wedge [\alpha']\varphi$

   A4. $[\alpha]\varphi \rightarrow [\alpha \sqcap \alpha']\varphi$

   A5. $\mathsf{P}(\emptyset)$

   A6. $\mathsf{P}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}(\alpha) \wedge \mathsf{P}(\beta)$

   A7. $\mathsf{P}(\alpha) \vee \mathsf{P}(\beta) \rightarrow \mathsf{P}(\alpha \sqcap \beta)$

   A8. $\neg\mathsf{P}_{\mathsf{w}}(\emptyset)$

   A9. $\mathsf{P}_{\mathsf{w}}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}_{\mathsf{w}}(\alpha) \vee \mathsf{P}_{\mathsf{w}}(\beta)$

   A10. $\mathsf{P}_{\mathsf{w}}(\alpha \sqcap \beta) \rightarrow \mathsf{P}_{\mathsf{w}}(\alpha) \wedge \mathsf{P}_{\mathsf{w}}(\beta)$

A11. $\mathsf{P}(\alpha) \wedge \alpha \neq_{act} \emptyset \rightarrow \mathsf{P}_{\mathsf{w}}(\alpha)$

A12. $\mathsf{P}_{\mathsf{w}}(\gamma) \rightarrow \mathsf{P}(\gamma)$, *where* $\gamma \in At(\Delta_0)$

A13. $\mathsf{O}(\alpha) \leftrightarrow \mathsf{P}(\alpha) \wedge \neg \mathsf{P}_{\mathsf{w}}(\overline{\alpha})$

A14. $[\alpha]\varphi \leftrightarrow \neg\langle\alpha\rangle\neg\varphi$

A15. $(a_1 \sqcup \cdots \sqcup a_n) =_{act} \mathbf{U}$

A16. $\langle\beta\rangle(\alpha =_{act} \alpha') \rightarrow \alpha =_{act} \alpha'$

A17. $\langle\gamma\rangle\varphi \rightarrow [\gamma]\varphi$, *where* $\gamma \in At(\Delta_0)$

Subs. $\varphi[\alpha] \wedge (\alpha =_{act} \alpha') \rightarrow \varphi[\alpha/\alpha']$

and the following deduction rules:

MP *if* $\vdash \varphi$ *and* $\vdash \varphi \rightarrow \psi$, *then* $\vdash \psi$

GN *if* $\vdash \varphi$, *then* $\vdash [\alpha]\varphi$

□

Some explanation is needed for the axioms. **A1** formalizes the nature of an impossible action: *after an impossible action everything becomes possible.* **A2** is a basic axiom for *dynamic logics.* **A3** tells us that *if something is true after the execution of a non-deterministic choice between two actions, then it has to be true after the execution of each one of these actions.* **A4** says that parallel execution of actions preserves properties; perhaps one might think of some scenario where this is not true, but this happens when we execute two actions inconsistent with each other, and this is just an impossible action in our framework. **A5**, **A6** and **A7** are similar axioms for strong permission, and **A8**, **A9** and **A10** are the duals for weak permission. Note that **A5** says that the impossible action is strongly permitted in every context, but **A8** says that it is not weakly permitted. So there is no context which allows its execution. It is in this sense that the impossible action can never be executed. The important point is to establish a relationship between the two versions of permission and axiom **A12** expresses an intuitive connection between strong and weak permission: *if an action which can only be executed in one possible context is weakly allowed, then it is strongly allowed.* This axiom implies a kind of compactness property of weak and strong permission: *if in every context an action α is weakly permitted, then α is strongly permitted* (see property **T6** below).

Note that the (schema) axiom **Subs** uses substitution on formulae: the notation $\varphi[\alpha]$ means that the formula, which the meta variable φ denotes, has an occurrence of

the boolean term, which the meta variable α denotes, and we write $\varphi[\alpha/\alpha']$ to mean that the term α is replaced in some of its occurrences by the boolean term α'.

Finally, axiom **A15** says that all the possible actions are *covered* by the choice between all the primitive actions of Δ_0. On the other hand, **A16** is needed to express that modalities do not affect equations (note that the formula $\alpha =_{act} \beta \rightarrow [\gamma]\alpha =_{act} \beta$ can be proven using the properties of equality and axiom **Subs**). Axiom **A17** formalizes the requirement that the transitions must be deterministic with respect to events. (It is important to stress that the action γ in this formula denotes an atom in the boolean atomic term algebra, which implies that the interpretation of this action term can only have at most one event.)

3.4 Soundness and Completeness

Two standard requirements for propositional logics are the soundness and completeness properties; we shall show that the given axiomatic system has both properties. These two theorems give us enough confidence about the adequacy of the basic system, whose axioms will remain in future versions.

In [GP90] and [Bro03], two different complete, and sound, systems are given for the modal part of the logic (that is, action terms and box modality), but both systems lack deontic concepts, and the complement described in those works is the absolute one. As explained above, the one described here is a kind of relative complement.

Theorem 1 (soundness). *The axiomatic system defined in definition 11 is sound with respect to the models defined in definition 9, that is:*

$$\vdash^L \varphi \Rightarrow \models^L \varphi.$$

Proof. *We have to prove that each axiom is valid, and that the deduction rules preserve validity. Axioms **A1-A2** and the deduction rules **MP** and **GN** are very standard and their soundness proofs can be found in the literature. On the other hand, it is clear that boolean algebra axioms are valid, since the interpretation of action operators are given by means of set operators. We prove the validity of axioms **A3-A17** and that the deduction rule **Subs** preserves validity. **A3:** Straightforward by first order properties. See axiom 7's proof.*
A4: *Direct using subset properties and "for all" properties.*
A5: *Straightforward by definition of $\models$ and vacuous domain.*
A6: *Suppose $w, W \models \mathsf{P}(\alpha \sqcup \beta)$, for arbitrary model M and world w. This means that: $\forall e \in \mathcal{I}(\alpha \sqcup \beta) : \mathcal{P}(w, e)$. Using first order logic we get: $(\forall e \in \mathcal{I}(\alpha) : \mathcal{P}(w, e)) \wedge (\forall e \in \mathcal{I}(\alpha) : \mathcal{P}(w, e))$, and this implies: $w, M \models \mathsf{P}(\alpha) \wedge \mathsf{P}(\beta)$.*

A7: *Similar reasoning as before, but using the fact that:* $\mathcal{I}(\alpha \sqcap \beta) = \mathcal{I}(\alpha) \cap \mathcal{I}(\beta)$.
A8: *For every model* M *and world* w*, by first-order reasoning we have:* $\neg(\exists e \in I(\emptyset) : \mathcal{P}(w, e))$*, and this means:* $\models \neg \mathsf{P_w}(\emptyset)$.
A9: *Suppose* $w, M \models \mathsf{P_w}(\alpha \sqcup \beta)$*; by definition we obtain:* $\exists e \in \mathcal{I}(\alpha \sqcup \beta) : \mathcal{P}(w, e)$ *and then using the definition of* $\mathcal{I}$ *and properties of* $\exists$ *we get:* $w, M \models \mathsf{P_w}(\alpha) \vee \mathsf{P_w}(\beta)$.
A10: *Similar to Axiom 10.*
A11: *Suppose that* $w, M \models \mathsf{P}(\alpha) \wedge \alpha \neq \emptyset$*; this means:* $\forall e \in I(\alpha) : e \in \mathcal{P}_w$ *and* $I(\alpha) \neq \emptyset$*; by basic first order reasoning we get:* $\exists e \in I(\alpha) : \mathcal{P}(w, e)$*, but this implies* $w, M \models \mathsf{P_w}(\alpha)$.
A12: *Note that if* γ *is an atom of the term boolean algebra then, by property* ***I.2****,* $\mathcal{I}(\gamma) = \{e\}$ *for some* $e \in \mathcal{E}$ *or* $\mathcal{I}(\gamma) = \emptyset$*; in either case we have that* $w, M \models \mathsf{P_w}(\gamma) \Rightarrow w, M \models \mathsf{P}(\gamma)$*, for every state* w *and* M.
A13, A14, A15: *Straightforward.*
A16: *The result follows from the fact that action interpretations are fixed, and they do not depend on states.*
A17: *If* γ *is an atom of the term boolean algebra, then, by property* ***I.2****,* $\mathcal{I}(\gamma) = \{e\}$ *for some* $e \in \mathcal{E}$ *or* $\mathcal{I}(\gamma) = \emptyset$*; if we have* $w, M \models \langle\gamma\rangle\varphi$*, for some world* w *of a structure* M*, then for some world* w' *and event* e *of* M *we have* $w \xrightarrow{e} w'$ *and* $w', M \models \varphi$*, but as explained above we have* $\mathcal{I}(\gamma) = \{e\}$*, and therefore, since the relation* $\mathcal{R}$ *is deterministic, we have for every* $e \in \mathcal{I}(\gamma)$ *and world* w' *such that* $w \xrightarrow{e} w'$ *we have* $w', M' \models \varphi$*. That is:* $w, M \models [\gamma]\varphi$.
Subs: *The result is straightforward from the fact that if we have* $\alpha =_{act} \alpha'$ *then* $I(\alpha) = I(\alpha')$*; here using the Leibniz equality property deduce that, if* $\models \varphi[\alpha]$*, then* $\models \varphi[\alpha']$. ■

Axiom **A2** and rule **GN** imply that we have a normal modal logic [Che99]. A number of useful standard modal logic properties can be found in [Che99]; we use some of these standard modal properties in proofs, and we will use ML to indicate this.

The following theorems of the axiomatic system defined are used in the completeness proof; actually, in [GP90], theorem **T3** is used for axiomatizing the modal part of boolean logic, and it should be enough for the modal part of our logic. Because we are taking an algebraic view of the logic, in our axiomatic system we focused on operational properties.

Theorem 2. *The following are theorems of DPL:*

T1. $\mathsf{P}(\alpha) \wedge \alpha' \sqsubseteq \alpha \rightarrow \mathsf{P}(\alpha')$.

T2. $\mathsf{P_w}(\alpha') \wedge \alpha' \sqsubseteq \alpha \rightarrow \mathsf{P_w}(\alpha)$.

T3. $[\alpha]\varphi \wedge (\alpha' \sqsubseteq \alpha) \rightarrow [\alpha']\varphi$.

T4. $[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha \sqcup \alpha'](\varphi \vee \psi)$.

T5. $[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha \sqcap \alpha'](\varphi \wedge \psi)$.

T6. $(\bigwedge_{[\alpha]_{BA} \in \Delta/\Phi_{BA} \wedge \alpha \sqsubseteq \alpha'} (P_w(\alpha) \vee (\alpha =_{act} \emptyset))) \rightarrow \mathsf{P}(\alpha')$.

T7. $\alpha =_{act} \alpha' \rightarrow [\beta]\alpha =_{act} \alpha'$.

Proof.
T1:

1.	$\mathsf{P}(\alpha) \rightarrow \mathsf{P}(\alpha \sqcap \alpha')$	*Axiom* ***A7***
2.	$\alpha' =_{act} \alpha \sqcap \alpha' \wedge \mathsf{P}(\alpha) \rightarrow \mathsf{P}(\alpha \sqcap \alpha')$	*PL, 1*
3.	$\alpha' =_{act} \alpha \sqcap \alpha' \wedge \mathsf{P}(\alpha \sqcap \alpha') \rightarrow \mathsf{P}(\alpha')$	***Subs*** *& PL*
4.	$\alpha' =_{act} \alpha \sqcap \alpha' \wedge \mathsf{P}(\alpha) \rightarrow \mathsf{P}(\alpha')$	*PL, 2, 3*

T2: *Similar to* ***T1*** *but using axiom* ***A9****.*
T3: *Similar to* ***T1*** *but using axiom* ***A4****.*
T4:

1.	$[\alpha]\varphi \rightarrow [\alpha](\varphi \vee \psi)$	*ML*
2.	$[\alpha']\psi \rightarrow [\alpha'](\varphi \vee \psi)$	*ML*
3.	$[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha](\varphi \vee \psi) \wedge [\alpha'](\varphi \vee \psi)$	*PL,1 ,2*
4.	$[\alpha](\varphi \vee \psi) \wedge [\alpha'](\varphi \vee \psi) \rightarrow [\alpha \sqcup \alpha'](\varphi \vee \psi)$	*ML,* ***A3****, 3*
5.	$[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha \sqcup \alpha'](\varphi \vee \psi)$	*PL, 3, 4*

T5:

1.	$[\alpha]\varphi \rightarrow [\alpha \sqcap \alpha']\varphi$	***A4***
2.	$[\alpha']\psi \rightarrow [\alpha \sqcap \alpha']\psi$	***A4***
3.	$[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha \sqcap \alpha']\varphi \wedge [\alpha \sqcap \alpha']\psi$	*PL, 1, 2*
4.	$[\alpha]\varphi \wedge [\alpha']\psi \rightarrow [\alpha \sqcap \alpha'](\varphi \wedge \psi)$	*ML, 3*

T6: *We prove* $(\bigwedge_{[\alpha]_{BA} \in \Delta/\Phi_{BA} \wedge \alpha \sqsubseteq \alpha'} (P_w(\alpha) \vee (\alpha =_{act} \emptyset))) \vdash^{L} \mathsf{P}(\alpha)$ *and the result follows by the deduction theorem.*

1.	$(\bigwedge_{[\alpha]_{BA} \wedge \alpha \sqsubseteq \alpha'} (P_w(\alpha) \vee (\alpha =_{act} \emptyset)))$	*Hyp.*
2.	$\alpha =_{act} \emptyset \rightarrow \mathsf{P}(\emptyset)$	*PL,* ***A5***

3.	$\alpha =_{act} \emptyset \rightarrow \mathsf{P}(\alpha)$	*PL*, ***Subs***, *2*
4.	$\alpha =_{act} \gamma_1 \sqcup \cdots \sqcup \gamma_n$	***BA***
5.	$(\mathsf{P_w}(\gamma_1) \vee \gamma_1 =_{act} \emptyset) \wedge \cdots \wedge (\mathsf{P_w}(\gamma_n) \vee \gamma_n =_{act} \emptyset)$	*PL, 1*
6.	$\mathsf{P}(\gamma_1) \wedge \cdots \wedge \mathsf{P}(\gamma_n)$	*PL, 5,* ***A12***
7.	$\mathsf{P}(\gamma_1 \sqcup \cdots \sqcup \gamma_n)$	*PL,* ***A6***
8.	$\mathsf{P}(\alpha')$	*PL*, ***Subs***, *7, 4*

In the above proof we use the word ***BA*** *to indicate that reasoning coming from Boolean Algebra is used in that step of the proof (note that the boolean terms* $\gamma_1, \ldots, \gamma_n$ *used in step 4 are the boolean atom terms which precede* α *in the algebra of terms).*

T7:

1.	$\alpha =_{act} \alpha$	*Ref. of* $=_{act}$
2.	$[\beta]\alpha =_{act} \alpha$	*ML, 1*
3.	$\alpha =_{act} \alpha' \rightarrow [\beta]\alpha =_{act} \alpha'$	*PL*, ***Subs***, *2*

■

Now, we can introduce the following (canonical) model; note that we use the atoms of the boolean term algebra (modulo boolean equations) as labels in the transitions. In other words, each boolean atom (of the action terms) can be mapped to one (and only one) *event* in the model.

Definition 12 (canonical model). *Given an equational (boolean) theory* Γ' *built from* Δ_0*, we define* $\mathcal{C} = \langle \mathcal{E}_\mathcal{C}, \mathcal{W}_\mathcal{C}, \mathcal{R}_\mathcal{C}, \mathcal{P}_\mathcal{C}, \mathcal{I}_\mathcal{C} \rangle$ *as follows:*

- $\mathcal{E}_\mathcal{C} \stackrel{\text{def}}{=} at(\Delta/\Gamma')$.
- $\mathcal{W}_\mathcal{C} \stackrel{\text{def}}{=} \{\Gamma \mid \Gamma$ *is a maximal consistent set of formulae and* $\Gamma' \subseteq \Gamma\}$.
- $\mathcal{R}_\mathcal{C} \stackrel{\text{def}}{=} \bigcup\{\mathcal{R}_{\alpha,w,w'} \mid w, w' \in \mathcal{W}_\mathcal{C} \wedge \alpha \in \Delta \wedge (\forall \varphi \in \Phi :[\,\alpha]\varphi \in w \Rightarrow \varphi \in w')\}$*, where* $R_{\alpha,w,w'} \stackrel{\text{def}}{=} \{w \stackrel{[\alpha']_{BA}}{\rightarrow} w' \mid \forall [\alpha']_{BA} \in \mathcal{I}_\mathcal{C}(\alpha)\}$.
- $\mathcal{P}_\mathcal{C} \stackrel{\text{def}}{=} \bigcup\{\mathcal{P}_{w,\alpha} \mid w \in \mathcal{W}_\mathcal{C} \wedge \mathsf{P}(\alpha) \in w\}$*, where:* $\mathcal{P}_{w,\alpha} \stackrel{\text{def}}{=} \{(w, [\alpha']_{BA}) \mid [\alpha']_{BA} \in \mathcal{I}_\mathcal{C}(\alpha)\}$.
- $\mathcal{I}_\mathcal{C}(\alpha_i) \stackrel{\text{def}}{=} \{[\alpha']_{BA} \in \mathcal{E}_\mathcal{C} \mid\!\vdash_{\Phi_{BA}} \alpha' \sqsubseteq \alpha_i\}$.
- $\mathcal{I}_\mathcal{C}(p_i) \stackrel{\text{def}}{=} \{w \in \mathcal{W}_\mathcal{C} \mid p_i \in w\}$.

□

We use this model to show the completeness of the logic; the usual way to do this is to prove an equivalent result: *each consistent set of formulae has a model.* First, we have to establish a number of useful lemmas:

Lemma 1. $\forall \alpha \in \Delta, \forall [\alpha']_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : \vdash_{\Phi_{BA}} \alpha' \sqsubseteq \alpha$
Proof. *The proof is by induction on the term* α.
Base Case:

- *If* $\alpha = \emptyset$, *then by definition* $\mathcal{I}_{\mathcal{C}}(\emptyset) = \emptyset$ *and the statement is vacuously true. Note that we use the symbol* $\emptyset$ *in two different ways: the first one as an action term, and the second one as the empty set.*
- *If* $\alpha = \alpha_i$, *then the result is straightforward by definition of* $\mathcal{I}_{\mathcal{C}}$.

Inductive Case:

- *case* $(\alpha = \alpha' \sqcup \alpha'')$: *let* $[\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha' \sqcup \alpha'')$ *be an event; then, by definition 12, we have* $[\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha') \cup \mathcal{I}_{\mathcal{C}}(\alpha'')$; *by the hypothesis we obtain:* $\vdash_{\Phi_{BA}} \gamma \sqsubseteq \alpha'$ *or* $\vdash_{\Phi_{BA}} \gamma \sqsubseteq \alpha''$, *and therefore, by properties of boolean algebras we obtain:* $\vdash_{\Phi_{BA}} \gamma \sqsubseteq \alpha' \sqcup \alpha''$.
- *case* $(\alpha = \alpha' \sqcap \alpha'')$: *similar argument as in the last step.*
- *case* $(\alpha = \overline{\alpha'})$: *suppose* $[\gamma]_{BA} \in \mathcal{E} - \mathcal{I}_{\mathcal{C}}(\alpha)$, *then* $\nvdash_{BA} \gamma \sqsubseteq \alpha$; *since* γ *is an atom, by boolean algebra properties we get:* $\vdash_{\Phi_{BA}} \gamma \sqsubseteq \overline{\alpha}$.

■

Now, we can prove a fundamental lemma.

Lemma 2 (truth lemma). $w, \mathcal{C} \models \varphi \Leftrightarrow \varphi \in w$.
Proof. *The proof is by induction on* φ.
Base Case: Using the definition we get

$$w, \mathcal{C} \models p_i \Leftrightarrow w \in \mathcal{I}_{\mathcal{C}}(p_i) \Leftrightarrow p_i \in w.$$

Inductive Case: We have several cases (the standard logical operators are handled as usual):

CASE I. We have to prove $w, \mathcal{C} \models [\alpha]\varphi \Leftrightarrow [\alpha]\varphi \in w$.
$\Rightarrow$) *Suppose* $w, \mathcal{C} \models [\alpha]\varphi$; *we have two possibilities:* $w, \mathcal{C} \models \alpha =_{act} \emptyset$ *or* $w, \mathcal{C} \models \alpha \neq_{act} \emptyset$. *In the former case we have (by axiom 2)* $[\alpha]\varphi \in w$. *If* $w, C \models \alpha \neq_{act} \emptyset$, *then (by definition of* $\mathcal{I}_{\mathcal{C}}$*) we have* $\mathcal{I}_{\mathcal{C}} \neq \emptyset$. *Now, by hypothesis:*

$$\begin{array}{ll} \forall [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha), \forall w' \in \mathcal{W}_{\mathcal{C}} : w \overset{[\gamma]_{BA}}{\rightarrow} w' \Rightarrow w', \mathcal{C} \models \varphi & \\ \equiv & \textit{[inductive hypothesis]} \\ \forall [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha), \forall w' \in \mathcal{W}_{\mathcal{C}} : w \overset{[\gamma]_{BA}}{\rightarrow} w' \Rightarrow \varphi \in w' & (*) \end{array}$$

On the other hand, suppose $[\alpha]\varphi \notin w$*, then (recalling properties of maximal consistent sets)* $\langle\alpha\rangle\neg\varphi \in w$*. Now, consider the set:* $\Gamma = \{\neg\varphi\} \cup \{\psi \mid [\alpha]\psi \in w\}$*. We claim that this set is consistent, if not:*

$$\exists \psi_1, ..., \psi_n \in \Gamma : \{\psi_1, ..., \psi_n, \neg\varphi\} \vdash \bot$$

by definition of contradiction. But using this we can deduce:

$$\begin{array}{ll} \langle\alpha\rangle\neg\varphi \wedge [\alpha]\psi_1 \wedge ... \wedge [\alpha]\psi_n \in w & \\ \Rightarrow & \textit{[axiom 3 and maximal consistent set properties]} \\ \langle\alpha\rangle(\neg\varphi \wedge \psi_1 \wedge ... \wedge \psi_n) \in w & \\ \Leftrightarrow & \textit{[hypothesis]} \\ \langle\alpha\rangle\bot \in w & \\ \Leftrightarrow & \textit{[ML]} \\ \bot \in w\ ! & \end{array}$$

Then Γ *has to be consistent, and therefore it has a maximal consistent extension* Γ^* *(by Lindenbaum's lemma). But by definition of* $\mathcal{R}_{\mathcal{C}}$*:*

$$\forall [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : w \overset{[\gamma]_{BA}}{\rightarrow} \Gamma^* \wedge \Gamma^* \models \neg\varphi$$

which contradicts (*) (recall that $\mathcal{I}_{\mathcal{C}}(\alpha) \neq \emptyset$*) and therefore* $[\alpha]\varphi \in w$*.*
$\Leftarrow$*) Suppose* $[\alpha]\varphi \in w$*; we have to prove* $w \models [\alpha]\varphi$*. Suppose that* $w \not\models [\alpha]\varphi$*, then this means:*

$$\exists [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha), \exists w' \in \mathcal{W}_{\mathcal{C}} : w \overset{[\gamma]_{BA}}{\rightarrow} w' \wedge w', \mathcal{C} \not\models \varphi$$

which is equivalent to (by ind.hyp.):

$$\exists [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha), \exists w' \in \mathcal{W}_{\mathcal{C}} : w \overset{[\gamma]_{BA}}{\rightarrow} w' \wedge \varphi \notin w' \quad (**)$$

But, by definition of $\mathcal{R}_{\mathcal{C}}$*, this means:*

$$\begin{array}{ll} \exists w' \in \mathcal{W}_{\mathcal{C}} : (\forall \psi : [\gamma]\psi \in w \Rightarrow \psi \in w') \wedge \varphi \notin w' & \\ \Rightarrow & \textit{[logic]} \\ \neg([\gamma]\varphi) \in w & \\ \Leftrightarrow & \textit{[max.cons.set properties]} \\ [\gamma]\varphi \notin w & (***) \end{array}$$

But we know by lemma 1 that $\gamma \sqsubseteq \alpha$. *From here and using the hypothesis* ($[\alpha]\varphi \in w$) *and using theorem* ***T3****, we obtain:* $[\gamma]\varphi \in w$, *and therefore:* $w, \mathcal{C} \models [\alpha]\varphi$.

CASE II. We have to prove: $w, \mathcal{C} \models \mathsf{P}(\alpha) \Leftrightarrow \mathsf{P}(\alpha) \in w$.
$\Rightarrow$) *Suppose* $w, \mathcal{C} \models \mathsf{P}(\alpha)$, *this means:*

$$\forall [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : \mathcal{P}_{\mathcal{C}}(w, [\gamma]_{BA}).$$

Because of lemma 1, this implies (using definition of $\mathcal{P}_{\mathcal{C}}$*) that either* $\mathsf{P}(\alpha) \in w$ *or* $\mathsf{P}(\beta) \in w$ *where* $\vdash_{\Phi_{BA}} \alpha \sqsubseteq \beta$, *since there is no other way to introduce this relation in the canonical model. In both cases the result follows, in the first trivially, in the second one by using* ***T1****.*
$\Leftarrow$) *Suppose that* $\mathsf{P}(\alpha) \in w$, *by definition of* $\mathcal{P}_{\mathcal{C}}$ *this means:*

$$\forall [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : \mathcal{P}_{\mathcal{C}}(w, [\gamma]_{BA}).$$

But using the definition of $\models$ *we get:* $w, \mathcal{C} \models P(\alpha)$.

CASE III. $w, \mathcal{C} \models \mathsf{P_w}(\alpha) \Leftrightarrow \mathsf{P_w}(\alpha) \in w$.
For the case $\alpha =_{act} \emptyset$ *the equivalence is trivial; let us prove the other case* ($\alpha \neq_{act} \emptyset$).
$\Rightarrow$) *Suppose* $w, \mathcal{C} \models \mathsf{P_w}(\alpha)$, *this means:*

$$\exists [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : \mathcal{P}_{\mathcal{C}}(w, [\gamma]_{BA}).$$

By definition of $\mathcal{P}_{\mathcal{C}}$ *this only happens if for some* β: $\gamma \sqsubseteq \beta$ *and* $\mathsf{P}(\beta) \in w$. *Then by theorem* ***T1*** *this implies* $\mathsf{P}(\gamma) \in w$, *and therefore, using axiom* ***A11****, we get:* $\mathsf{P_w}(\gamma) \in w$; *from this, by theorem* ***T2****, we obtain* $\mathsf{P_w}(\alpha) \in w$.
$\Leftarrow$) *Suppose* $\mathsf{P_w}(\alpha) \in w$. *We know by properties of atomic boolean algebras that:*

$$\begin{array}{ll} [\alpha]_{BA} = [\gamma_1]_{BA} \sqcup_{[]} \ldots \sqcup_{[]} [\gamma_n]_{BA} & \text{for some } [\gamma_1]_{BA}, \ldots, [\gamma_n]_{BA} \text{ atoms in } \Delta/\Phi_{BA} \\ \Leftrightarrow & \text{[def. of } \Delta/\Phi_{BA}\text{]} \\ [\alpha]_{BA} = [\gamma_1 \sqcup \ldots \sqcup \gamma_n]_{BA}. & \end{array}$$

But this implies by deduction rule BA that $\mathsf{P_w}(\gamma_1 \sqcup \ldots \sqcup \gamma_n) \in w$. *By axiom* ***A9****, this implies:*

$$\mathsf{P_w}(\gamma_1) \vee \ldots \vee \mathsf{P_w}(\gamma_n) \in w.$$

Let γ_i *be some of these action terms such that* $\mathsf{P_w}(\gamma_i) \in w$; *since* $\gamma_i \in at(\Delta)$, *using* ***MP*** *and* ***A12*** *we get* $\mathsf{P}(\gamma_i) \in w$. *By definition of* $\mathcal{P}_{\mathcal{C}}$, *this implies that:*

$$\exists [\gamma]_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha) : \mathcal{P}_{\mathcal{C}}([\gamma]_{BA}, w).$$

and this is just the definition of $w, \mathcal{C} \models \mathsf{P_w}(\alpha)$. ■

Note that we have to prove that the defined interpretation $\mathcal{I}_{\mathcal{C}}$ holds with the restrictions **I.1** and **I.2** (**I.3.** is satisfied by definition). Also we must prove that the transitions in the canonical model are deterministic with respect to events. The following theorems do this.

Theorem 3. *For any $w, w', w'' \in \mathcal{W}_{\mathcal{C}}$ we have that, if $w \xrightarrow{e} w'$ and $w \xrightarrow{e} w''$ are in $\mathcal{R}_{\mathcal{C}}$, then $w' = w''$.*

Proof. *We know that $e = [\gamma]$ for some $\gamma \in At(\Delta/\Gamma')$; now by axiom **A17** we have that $w, \mathcal{C} \vDash \langle\gamma\rangle\varphi \rightarrow [\gamma]\varphi$, this implies that both w and w'' satisfy the same predicates, and therefore $w' = w''$.* ■

Theorem 4. *The function $\mathcal{I}_{\mathcal{C}}$ satisfies conditions **I.1** and **I.2**.*

Proof. *First note that all the atoms of the boolean algebra Δ/Φ_{BA} (the Lindenbaum-Tarski algebra [Sik69]) have the following form (or are equivalent to it):*

$$\alpha_1^1 \sqcap \ldots \sqcap \alpha_m^1 \sqcap \alpha_1^2 \sqcap \ldots \sqcap \alpha_k^2$$

where for all $\alpha_i \in \Delta_0$: $\alpha_i = \alpha_j^1$ or $\overline{\alpha_i} = \alpha_j^2$, for some j. That is, the atoms in the Lindenbaum algebra can be represented by terms which are composed of "intersections" of atomic actions or their negations. It is for this reason that the atoms of the Lindenbaum algebra are suitable for representing labels in the model: each of them point out which primitive actions are executed and which are not.

*That $\mathcal{I}_{\mathcal{C}}$ satisfies conditions **I.1** and **I.2** is implied by the underlying structure of the generated Lindenbaum Algebra:*

I.1: *If $[\gamma] \in \mathcal{I}_{\mathcal{C}}(\alpha_i) - \bigcup_{j \neq i}(\mathcal{I}_{\mathcal{C}}(\alpha_j))$, then $\gamma =_{act} \alpha_i \sqcap (\bigsqcap_{j \neq i}(\overline{\alpha_j}))$, where $\bigsqcap$ is used to denote the application of $\sqcap$ to a finite sequence of boolean terms.*

I.2: *We have to show that, if $[\gamma] \in \mathcal{I}(\alpha_i) \cap \mathcal{I}(\alpha_j)$, for some $i \neq j$, then:*

$$\bigcap\{\mathcal{I}(\alpha_k) \mid [\gamma] \in \mathcal{I}(\alpha_k)\} = \{[\gamma]\}. \tag{3.1}$$

In this case it is easy to see that:

$$\gamma =_{act} \alpha_1^1 \sqcap \ldots \sqcap \alpha_m^1 \sqcap \overline{\alpha_1^2} \sqcap \ldots \sqcap \overline{\alpha_{m'}^2} \tag{3.2}$$

where the α_i^1 are the primitive actions which have the equivalence class $[\gamma]$ in their interpretation, and the α_i^2 are the rest. Since the right term in equation 3.2 is an atom, every other $[\gamma']$ that satisfies condition 3.1 also satisfies: $[\gamma'] = [\gamma]$. The theorem follows. ■

We have proved that the canonical model has the correct behaviour; the completeness follows:

Corollary 1. *For every consistent set Γ of DPL, there is a model which satisfies it.*
Proof. *If Γ is consistent, then there exists a maximal extension of it which is a maximal consistent set, and therefore this set is a world w in the canonical model. By the definition of canonical model we know $w, \mathcal{C} \vDash \Gamma$; this completes the proof.* ■

From it we obtain compactness:

Corollary 2. *If every finite subset of a set Γ of formulae is satisfiable, then Γ is satisfiable.*

Decidability can be proved using a selection argument.

Theorem 5 (decidability). *Satisfiability is decidable in DPL.*
Proof. *Suppose that for a formula φ: $w, M \vDash \varphi$, for some model M and world w. Let $d(\varphi) = m$ be the degree of φ (that is, the maximal depth of nested modalities), and let n be the number of primitive actions in the language.*

First, note that for every world in M we have at most $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$ possible relationships with other worlds (that is, the maximum number of events in the model). Let M' be the model obtained from M by ruling out those worlds not reachable from w in m "steps". Clearly, $M', w \vDash \varphi$, and M' has at most $m * (2^n - 1)$ worlds, where $m = d(\varphi)$ and n is the number of primitive actions.

This gives us a decidability method: given φ, build all the models up to size $m * (2^n - 1)$ and check if φ is true in every one of them. Obviously, this method is exponential in complexity. ■

The following theorems of the axiomatic system give us the first flavor of it.

Theorem 6. *The following sentences are theorems of DPL.*

T8. $\alpha =_{act} \emptyset \leftrightarrow \mathsf{P}(\alpha) \wedge \neg \mathsf{P}_{\mathsf{w}}(\alpha)$.

T9. $\mathsf{O}(\alpha) \wedge \mathsf{O}(\overline{\alpha}) \leftrightarrow \mathbf{U} =_{act} \emptyset$.

T10. $\mathsf{O}(\alpha) \wedge \mathsf{O}(\beta) \rightarrow \mathsf{O}(\alpha \sqcap \beta)$.

T11. $\mathsf{P}(\mathbf{U}) \rightarrow \mathsf{P}(\alpha)$ *for every action* α.

T12. $\mathsf{P}_{\mathsf{w}}(\alpha) \rightarrow \mathsf{P}_{\mathsf{w}}(\mathbf{U})$ *for every action* α.

T13. $\mathsf{O}(\mathbf{U}) \leftrightarrow \mathsf{P}(\mathbf{U})$.

T14. $\mathsf{O}(\emptyset) \leftrightarrow \neg \mathsf{P}_{\mathsf{w}}(\mathbf{U})$.

T15. $\mathsf{O}(\alpha) \rightarrow \mathsf{P}(\alpha)$.

Proof. *We prove property **T8** as an example, the other proofs are similar. Note that the direction $\rightarrow$ is straightforward by axioms **A5** and **A8** and using **Subs**. For the other direction we prove $\mathsf{P}(\alpha) \wedge \neg \mathsf{P}_{\mathsf{w}}(\alpha) \vdash^{L} \alpha =_{act} \emptyset$ and then we use the deduction theorem:*

1.	$\mathsf{P}(\alpha) \wedge \neg \mathsf{P}_{\mathsf{w}}(\alpha)$	*Hyp.*
2.	$\neg \mathsf{P}(\alpha) \vee \alpha =_{act} \emptyset \vee \mathsf{P}_{\mathsf{w}}(\alpha)$	*PL, **A11***
3.	$\alpha =_{act} \emptyset$	*PL, 1, 2*

■

Some intuition about these properties is needed to understand the essence of the logic. Theorem **T8** says that only impossible actions are both strongly allowed and forbidden. **T9** is related with this property, it says that if we are obliged to perform contradictory actions, then every action is impossible. The equation $\emptyset =_{act} \mathbf{U}$ implies that we have a degenerate boolean algebra. Though we have the possibility of having the degenerate boolean algebra in our semantics, usually this can be considered as indicating that our action algebra is inconsistent. Property **T14** is strange at first sight; it says that it is obligated to do an impossible action if no action is allowed. Actually, it only says that in this case we are obliged to do nothing. **T15** says that, if an action is obligated, then it is permitted.

Note that we do not have what is sometimes called *Kant's law* [MWD94], that is: $\mathsf{O}(\alpha) \rightarrow \langle \alpha \rangle \top$. Informally, this can be thought of as saying that ought implies can. We do not believe that this is the case in computing systems. Let us illustrate this with a simple example: it is obligatory for an automatic bank teller to print a receipt when it gives money to a customer; however, if there is no more paper, the receipt cannot be printed. This scenario can be formalized as follows. Consider a vocabulary with actions: `wdr` (it represents the action of withdraw money), `pr` (it is the action of printing a receipt) and `msg` (an error message is displayed). We have a propositional variable `npaper`, which is true when there is no paper in the machine. Consider the following formulae:

- $\mathsf{Done}(\mathtt{wdr}) \rightarrow \mathsf{O}(\mathtt{pr})$
- $\neg \mathtt{npaper} \rightarrow [\mathtt{pr}]\bot$

The first formula says that, if a customer withdrew money, then the machine is obliged to print a receipt. The second formula says that, if there is no paper, then the machine cannot print a receipt. A possible model of this set of sentences is depicted in figure 3.3. In this model, we have two worlds: w, w'. In the two worlds, the predicate

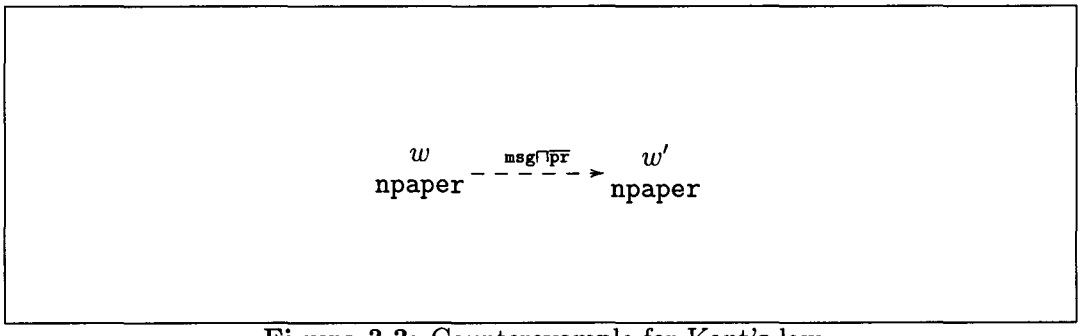


Figure 3.3: Counterexample for Kant's law

npaper is true; the dashed arrow indicates an action which is not allowed (to not print a receipt). If we call this model M, we have that $w, M \not\models \mathsf{O}(\mathtt{pr}) \rightarrow \langle \mathtt{pr} \rangle \top$, i.e., we are obliged to print a receipt but this action is not possible in this state.

In this respect, our definition of obligation differs from the one given in [Mey88]. In that framework Kant's law is a theorem of the logic. Similarly, we have neither $\mathsf{P_w}(\alpha) \rightarrow \langle \alpha \rangle \top$ nor $\mathsf{P}(\alpha) \rightarrow \langle \alpha \rangle \top$.

3.5 Spicing up DPL with Time

We have defined the basic logic, but if we want a good logic for specifying computing systems, the dimension of time is needed. Several types of temporal logics have been used by computer scientists in recent decades. In this section we shall introduce a *Branching Time Logic*; this logic is very similar to CTL (see [EH82]), that is, we allow temporal predicates combined with quantifiers on paths. Although, CTL* logics are more expressive, they are much harder to axiomatize, as is shown in [Rey01]. We leave for further work a CTL* version of the logic presented here.

On the other hand, the temporalization shown below is an *Okhamist* logic since formulae are evaluated with respect to a history and an instant, that is, we evaluate predicates using fixed traces. This semantics allows us to introduce the interesting predicate Done(), which can be used to predicate on the immediate past; this operator is mentioned in [Mey88] and [KQM91], but here we offer an axiomatization with some new axioms, and we show that mixing it with temporal notions allows us to

express interesting properties. Other past operators are not described here, though the extension of the logic to support these is immediate. (In chapter 7 we introduce a variation of the Done() operator which is useful when we consider the notion of component or modular piece of specification.)

First, we will present some changes to the definitions given earlier to be able to introduce time in the language. Let us define the temporal formulae.

Definition 13 (temporal formulae). *Given a DPL vocabulary* $\langle \Phi_0, \Delta_0 \rangle$, *the set of temporal deontic formulae* (Φ_T) *is defined as follows:*

- $\Phi \subseteq \Phi_T$. *That is, the formulae defined in definition 8 are temporal formulae.*
- *if* $\alpha \in \Delta$, *then* $\mathsf{Done}(\alpha) \in \Phi_T$.
- *if* $\varphi, \psi \in \Phi_T$ *and* $\alpha \in \Delta$, *then* $(\varphi \rightarrow \psi) \in \Phi_T, [\alpha]\varphi \in \Phi_T$ *and* $\neg\varphi \in \Phi_T$.
- *if* $\varphi, \psi \in \Phi_T$, *then* $\mathsf{AG}\varphi \in \Phi_T$, $\mathsf{AN}\varphi \in \Phi_T$, $\mathsf{A}(\varphi \,\mathcal{U}\, \psi) \in \Phi_T$ *and* $\mathsf{E}(\varphi \,\mathcal{U}\, \psi) \in \Phi_T$.

The temporal operators are the classic ones in CTL logics; intuitively, the predicate $\mathsf{AN}\varphi$ means *in all possible executions* φ *is true at the next moment*, $\mathsf{AG}\varphi$ means *in all executions* φ *is always true*, $\mathsf{A}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ means *for every possible execution* φ_1 *is true until* φ_2 *becomes true* and $\mathsf{E}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ says there *exists some execution where* φ_1 *is true until* φ_2 *becomes true.* As usual, using these operators we can define their dual versions:

- $\mathsf{AF}\varphi \overset{\mathsf{def}}{\Longleftrightarrow} \mathsf{A}(\top \,\mathcal{U}\, \varphi)$.
- $\mathsf{EF}\varphi \overset{\mathsf{def}}{\Longleftrightarrow} \neg\mathsf{AG}\neg\varphi$.
- $\mathsf{EN}\varphi \overset{\mathsf{def}}{\Longleftrightarrow} \neg\mathsf{AN}\neg\varphi$.

In order to define the semantics of the temporal version of the logic, we need some changes to the structures considered, as well as needing to introduce the definition of traces. Firstly, we define the notion of initial states, then all the possibles traces of execution will be defined with respect to an initial state.

Definition 14 (temporal model). *Given a language* $L = \langle \Phi_0, \Delta_0 \rangle$, $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}, w \rangle$ *is called a temporal structure, where:*

- $\langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ *is a structure as defined in definition 9.*

- $w \in W$ *is the initial state.*

□

Using the initial state we can consider all the traces (or paths) which start in this state.

Definition 15 (traces). *Given a model $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}, w \rangle$ a trace is a (labeled) path $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} ...$, where for every i: $s_i \xrightarrow{e_i} s_{i+1} \in \mathcal{R}$, and $s_0 = w$. The set of all traces with initial state w is $\Sigma(w)$.* □

Note that some paths could be finite; we use maximal traces (i.e., those traces which cannot be extended) to give semantics to the temporal operators.

We need some additional notation; given an infinite trace (or path) $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} ...$, we denote by $\pi^i = s_i \xrightarrow{e_i} s_{i+1} \xrightarrow{e_{i+1}} ...$ the subpath of π starting at position i. The notation $\pi_i = s_i$ is used to denote the i-th element in the path, and we write $\pi[i..j]$ (where $i \leq j$) for the subpath $s_i \xrightarrow{e_i} ... \xrightarrow{e_j} s_{j+1}$. Finally, given a finite path $\pi' = s'_0 \xrightarrow{e'_0} .... \xrightarrow{e'_n} s_{n+1}$, we say $\pi' \preceq \pi$ if π' is an initial subpath of π, that is: $s_i = s'_i$ and $e_i = e'_i$ for $0 \leq i \leq n$, and we denote by $\prec$ the strict version of $\preceq$.

Definition 16 (maximal traces). *Given a structure $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}, w \rangle$, a trace π is called maximal if and only if there is no other trace π' such that: $\pi \prec \pi'$. The set of maximal traces is denoted by $\Sigma^*(w)$. Note that all the infinite traces are maximal. We denote by $\#\pi$ the length of the trace π; if it is infinite we just use an abuse of notation and say $\#\pi = \infty$.* □

The relation $\models^L_{DTL}$ is defined using paths and structures. An interesting point to note is that we also use a given instant to evaluate formulae. It is needed here since the predicate Done(−) allows us to predicate about the immediate past. Note that, in the following definition, we use the relation $\models$ defined in definition 10.

Definition 17 ($\models_{DTL}$). *Given a model $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}, w \rangle$, a trace $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} ... \in \Sigma^*(w)$, we define the relation $\models_{DTL}$ as follows:*

- $\pi, i, M \models_{DTL} \varphi \overset{\text{def}}{\Longleftrightarrow} \pi_i, \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle \models \varphi$, *if φ does not contain any temporal predicates.*
- $\pi, i, M \models_{DTL} \neg\varphi \overset{\text{def}}{\Longleftrightarrow}$ *not* $\pi, i, M \models_{DTL} \varphi$.

- $\pi, i, M \vDash_{DTL} \varphi_1 \rightarrow \varphi_2 \stackrel{\text{def}}{\Longleftrightarrow}$ *either not* $\pi, i, M \vDash_{DTL} \varphi_1$ *or* $\pi, i, M \vDash_{DTL} \varphi_2$.

- $\pi, i, M \vDash_{DTL} \mathsf{Done}(\alpha) \stackrel{\text{def}}{\Longleftrightarrow} i > 0$ *and* $e_{i-1} \in \mathcal{I}(\alpha)$.

- $\pi, i, M \vDash_{DTL} [\alpha]\varphi \stackrel{\text{def}}{\Longleftrightarrow} \forall \pi' = s'_0 \stackrel{e'_0}{\rightarrow} s'_1 \stackrel{e'_1}{\rightarrow} \ldots \in \Sigma^*(w)$ *such that* $\pi[0..i] \prec \pi'$, *if* $e'_i \in \mathcal{I}(\alpha)$, *then* $\pi', i+1, M \vDash_{DTL} \varphi$.

- $\pi, i, M \vDash_{DTL} \mathsf{AN}\varphi \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi' : \pi', i+1, M \vDash \varphi$.

- $\pi, i, M \vDash_{DTL} \mathsf{AG}\varphi \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *we have that* $\forall i \leq j \leq \#\pi' : \pi', j, M \vDash \varphi$.

- $\pi, i, M \vDash_{DTL} \mathsf{A}(\varphi_1 \,\mathcal{U}\, \varphi_2) \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi_2$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *we have that* $\exists i \leq j \leq \#\pi' : \pi', j, M \vDash \varphi_2$ *and* $\forall i \leq k \leq j : \pi', k, M \vDash \varphi$.

- $\pi, i, M \vDash_{DTL} \mathsf{E}(\varphi_1 \,\mathcal{U}\, \varphi_2) \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \vDash \varphi_2$. *If* $i \neq \#\pi$, *then* $\exists \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *we have that* $\exists i \leq j \leq \#\pi' : \pi', j, M \vDash \varphi_2$ *and* $\forall i \leq k \leq j : \pi', k, M \vDash \varphi_1$.

□

Note that, in this semantics, when we are at the end of a maximal (finite) trace those predicates true in the last state are maintained through time.

We say that $M \vDash_{DTL} \varphi$ if $\pi, i, M \vDash_{DTL} \varphi$ for all paths π and instants i. And we say $\vDash_{DTL} \varphi$ if φ holds for all models M. Note that we have, at least, two ways of defining the notion of valid formula; one is saying that a formula is valid if and only if the formula is true for every model, in any path in any position; we denote this by $\vDash \varphi$. The other possibility is to say that a formula is valid if and only if it is true in every model, at the beginning of every path, we denote this validity by $\vDash_A \varphi$, the subindex pointing to the fact that this is a kind of anchored interpretation, anchored at the beginning of time (see [MP89]). In this chapter we use the non-anchored version of satisfiability.

Next, we present an axiomatic system for the semantics just described; some axioms are the classic ones for CTL and the others allow us to exploit the relationship between modal and temporal operators.

Definition 18 (DTL Axioms). *Given a vocabulary* $\langle \Phi_0, \Delta_0 \rangle$, *the axiomatic system is composed of the (substitution instances of) the following axioms:*

- *All the axioms given in definition 11.*

TempAx1. $\langle \mathbf{U} \rangle \top \rightarrow (\mathsf{AN}\varphi \leftrightarrow [\mathbf{U}]\varphi)$

TempAx2. $[\mathbf{U}]\bot \rightarrow (\mathsf{AN}\varphi \leftrightarrow \varphi)$

TempAx3. $\mathsf{AG}\varphi \leftrightarrow \neg\mathsf{E}(\top\, \mathcal{U}\, \neg\varphi)$

TempAx4. $\mathsf{E}(\varphi\, \mathcal{U}\, \psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ENE}(\varphi\, \mathcal{U}\, \psi))$

TempAx5. $\mathsf{A}(\varphi\, \mathcal{U}\, \psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ANA}(\varphi\, \mathcal{U}\, \psi))$

TempAx6. $[\alpha]\mathsf{Done}(\alpha)$

TempAx7. $[\overline{\alpha}]\neg\mathsf{Done}(\alpha)$

TempAx8. $\neg\mathsf{Done}(\emptyset)$

TempAx9. $\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg\mathsf{Done}(\alpha)$

and the following deduction rules:

- *Rules given in definition* 11.

TempRule1. *if* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ *and* $\vdash \varphi \rightarrow \mathsf{AN}\varphi$, *then* $\vdash \varphi$

TempRule2. *if* $\vdash \varphi$, *then* $\vdash \mathsf{AG}\varphi$

TempRule3. *if* $\vdash \varphi \rightarrow (\neg\psi \wedge \mathsf{EN}\varphi)\}$, *then* $\vdash \varphi \rightarrow \neg\mathsf{A}(\vartheta\, \mathcal{U}\, \psi)$

TempRule4. *if* $\vdash \varphi \rightarrow (\neg\psi \wedge \mathsf{AN}(\varphi \vee \neg\mathsf{E}(\vartheta\, \mathcal{U}\, \psi)))$, *then* $\vdash \varphi \rightarrow \neg\mathsf{E}(\vartheta\, \mathcal{U}\, \psi)$

TempRule5. *if* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \mathsf{AG}\varphi$, *then* $\vdash \varphi$

□

Some comments will be useful; note that the formula $\neg\mathsf{Done}(\mathbf{U})$ holds only at the beginning of each trace, and therefore we can think of this as asserting that the actual instant is the beginning of time. Axioms **TempAx1** and **TempAx2** relate the box modal operator with the temporal operators, reflecting the semantics introduced above. On the other hand, **TempAx3 - TempAx5** are classic axioms for CTL logic (given in [EH82]). Axioms **TempAx6-TempAx9** define the Done() operator, mainly using the box modality. The first inference rule is a kind of induction rule, saying: *if something is true at the beginning of time, and it is preserved by every action, we can*

deduce that it holds everywhere. Thus it enables us to establish invariants. On the other hand, **TempRule5** implies that every instant is reachable from the beginning. The other rules are standard for temporal logics.

We prove some useful theorems of this system. Note that, since the CTL system is embedded in the axiomatic system given in definition 18, we can derive all the CTL theorems; we only focus on the new ones.

Theorem 7. *If* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ *and* $\vdash \mathsf{AN}\varphi$, *then* $\vdash \varphi$
Proof. *Suppose:* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ *and* $\vdash \mathsf{AN}\varphi$, *then:*

1.	$\mathsf{AN}\varphi$	*Hyp.*
2.	$\varphi \rightarrow \mathsf{AN}\varphi$	*PL, 1*

Therefore using ***TempRule1*** *we get:* $\vdash \varphi$. ■

Corollary 3. *If* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ *and* $\vdash \varphi \rightarrow [\mathbf{U}]\varphi$, *then* $\vdash \varphi$
Proof. *We suppose* $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ *and* $\vdash \varphi \rightarrow [\mathbf{U}]\varphi$; *then we prove that* $\langle\mathbf{U}\rangle\top \vdash \varphi \rightarrow \mathsf{AN}\varphi$ *and* $[\mathbf{U}]\bot \vdash \varphi \rightarrow \mathsf{AN}\varphi$, *and therefore by the deduction theorem we get* $\vdash \varphi \rightarrow \mathsf{AN}\varphi$. *Finally, using* ***TempRule1*** *we get:* $\vdash \varphi$.
Case 1: $\langle\mathbf{U}\rangle\top \vdash \varphi \rightarrow \mathsf{AN}\varphi$

1.	$\langle\mathbf{U}\rangle\top$	*Assumption*
2.	$\mathsf{AN}\varphi \leftrightarrow [\mathbf{U}]\varphi$	*PL, 1,* ***TempAx2***
3.	$\varphi \rightarrow \mathsf{AN}\varphi$	*PL, Hyp, 2*

Case 2: $[\mathbf{U}]\bot \vdash \varphi \rightarrow \mathsf{AN}\varphi$

1.	$[\mathbf{U}]\bot$	*Assumption*
2.	$\mathsf{AN}\varphi \leftrightarrow \varphi$	***TempAx1***
3.	$\varphi \rightarrow \mathsf{AN}\varphi$	*PL, 2*

■

Both theorem 7 and corollary 3 are two different formulations of the induction principle **TempRule1**. The next theorem allows us to characterize deadlock: *when no action is enabled, we stay in this state forever.* This property is established in [Kro87] as an axiom.

Theorem 8. $\vdash [\mathbf{U}]\bot \wedge \varphi \rightarrow \mathsf{AN}([\mathbf{U}]\bot \wedge \varphi)$
Proof.

1.	$[\mathbf{U}]\bot \rightarrow (\mathsf{AN}\varphi \leftrightarrow \varphi)$	***TempAx2***

2.	$[\mathbf{U}]\bot \wedge \varphi \rightarrow \mathsf{AN}\varphi$	*PL, 1*
3.	$[\mathbf{U}]\bot \rightarrow ((\mathsf{AN}[\mathbf{U}]\bot) \leftrightarrow [\mathbf{U}]\bot)$	***TempAx2***
4.	$[\mathbf{U}]\bot \rightarrow \mathsf{AN}[\mathbf{U}]\bot$	*PL, 3*
5.	$[\mathbf{U}]\bot \wedge \varphi \rightarrow (\mathsf{AN}[\mathbf{U}]\bot \wedge \mathsf{AN}\varphi)$	*PL, 2, 4*
6.	$[\mathbf{U}]\bot \wedge \varphi \rightarrow \mathsf{AN}([\mathbf{U}]\bot \wedge \varphi)$	*PL, CTL property*

■

Now, we can prove some important properties of the Done() operator.

Theorem 9. *The following are theorems of the axiomatic system defined above.*

T16. $\mathsf{Done}(\alpha) \wedge \alpha \sqsubseteq \alpha' \rightarrow \mathsf{Done}(\alpha')$

T17. $\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta)$

T18. $\mathsf{Done}(\alpha \sqcap \beta) \leftrightarrow \mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta)$

T19. $\mathsf{Done}(\alpha \sqcup \beta) \wedge \mathsf{Done}(\overline{\alpha}) \rightarrow \mathsf{Done}(\beta)$

T20. $[\alpha]\varphi \wedge [\beta]\mathsf{Done}(\alpha) \rightarrow [\beta]\varphi$

Proof.
T16 *We use corollary 3 for proving this property (which can be thought of as an induction).*
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\mathsf{Done}(\alpha) \wedge \neg\mathsf{Done}(\alpha'))$	*PL &* ***TempAx9***
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\mathsf{Done}(\alpha') \rightarrow \neg\mathsf{Done}(\alpha))$	*PL, 1*
3.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*PL, 2*
4.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathsf{Done}(\alpha) \wedge \alpha \sqsubseteq \alpha' \rightarrow \mathsf{Done}(\alpha'))$	*PL, 3*

Ind.Case. Suppose: $\alpha \sqsubseteq \alpha'$.

1.	$[\alpha]\mathsf{Done}(\alpha)$	***TempAx6***
2.	$[\alpha']\mathsf{Done}(\alpha')$	***TempAx6***
3.	$[\alpha]\mathsf{Done}(\alpha) \wedge [\alpha']\mathsf{Done}(\alpha') \rightarrow [\alpha \sqcap \alpha'](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\alpha'))$	***T5***
4.	$[\alpha \sqcap \alpha'](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\alpha'))$	*MP, 1, 2, 3*
5.	$[\alpha']\mathsf{Done}(\alpha') \wedge \alpha \sqsubseteq \alpha' \rightarrow [\alpha]\mathsf{Done}(\alpha')$	***T3***
6.	$\alpha \sqsubseteq \alpha'$	*Assumption*
7.	$[\alpha]\mathsf{Done}(\alpha')$	*PL, 2, 5, 6*
8.	$[\alpha]\mathsf{Done}(\alpha') \rightarrow [\alpha](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*ML*

9.	$[\alpha](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*MP, 7, 8*
10.	$[\overline{\alpha}]\neg\mathsf{Done}(\alpha)$	***TempAx7***
11.	$[\overline{\alpha}](\neg\mathsf{Done}(\alpha) \vee \mathsf{Done}(\alpha'))$	*PL, 10*
12.	$[\overline{\alpha}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*PL, 11*
13.	$[\alpha](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha')) \wedge [\overline{\alpha}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$ $\rightarrow [\alpha \sqcup \overline{\alpha}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	***T4***
14.	$[\alpha \sqcup \overline{\alpha}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*PL, 9, 12, 13*
15.	$[\mathbf{U}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	***BA**, 14*
16.	$(\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha')) \rightarrow [\mathbf{U}](\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\alpha'))$	*PL, 15*
17.	$(\mathsf{Done}(\alpha) \wedge \alpha' \sqsubseteq \alpha \rightarrow \mathsf{Done}(\alpha'))$ $\rightarrow [\mathbf{U}](\mathsf{Done}(\alpha) \wedge \alpha' \sqsubseteq \alpha \rightarrow \mathsf{Done}(\alpha'))$	*PL, 16*

***T17:**: We use corollary 3:*
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg\mathsf{Done}(\alpha \sqcup \beta)$	***TempAx9***
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow ((\neg\mathsf{Done}(\alpha) \wedge \neg\mathsf{Done}(\beta)) \rightarrow \neg\mathsf{Done}(\alpha \sqcup \beta))$	*PL, 1*
3.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*PL, 2*

Ind.Case:

1.	$[\alpha \sqcup \beta]\mathsf{Done}(\alpha \sqcup \beta)$	***TempAx6***
2.	$[\alpha]\mathsf{Done}(\alpha)$	***TempAx6***
3.	$[\beta]\mathsf{Done}(\beta)$	***TempAx6***
4.	$[\alpha]\mathsf{Done}(\alpha) \wedge [\beta]\mathsf{Done}(\beta) \rightarrow [\alpha \sqcup \beta](\mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*T4*
5.	$[\alpha \sqcup \beta](\mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*PL, 2, 3, 4*
6.	$[\alpha \sqcup \beta](\mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$ $\rightarrow [\alpha \sqcup \beta](\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*ML*
7.	$[\alpha \sqcup \beta](\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*MP, 5, 6*
8.	$[\overline{\alpha \sqcup \beta}]\neg\mathsf{Done}(\alpha \sqcup \beta)$	***TempAx7***
9.	$[\overline{\alpha} \sqcap \overline{\beta}]\neg\mathsf{Done}(\alpha \sqcup \beta)$	*PL & **BA**, 8*
10.	$[\overline{\alpha} \sqcap \overline{\beta}]\neg\mathsf{Done}(\alpha \sqcup \beta)$ $\rightarrow [\overline{\alpha} \sqcap \overline{\beta}](\neg\mathsf{Done}(\alpha) \wedge \neg\mathsf{Done}(\beta) \rightarrow \neg\mathsf{Done}(\alpha \sqcup \beta))$	*ML*
11.	$[\overline{\alpha} \sqcap \overline{\beta}](\neg\mathsf{Done}(\alpha) \wedge \neg\mathsf{Done}(\beta) \rightarrow \neg\mathsf{Done}(\alpha \sqcup \beta))$	*MP, 9, 10*
12.	$[\overline{\alpha \sqcup \beta}](\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	*ML & **BA**, 11*
13.	$[\mathbf{U}](\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta))$	***BA** & **A3**, 12, 7*

***T18:**$\rightarrow$):*

1.	$\mathsf{Done}(\alpha \sqcap \beta) \wedge \alpha \sqcap \beta \sqsubseteq \alpha \rightarrow \mathsf{Done}(\alpha)$	***T16***
2.	$\alpha \sqcap \beta \sqsubseteq \alpha$	*Def.*$\sqsubseteq$
3.	$\mathsf{Done}(\alpha \sqcap \beta) \rightarrow \mathsf{Done}(\alpha)$	*PL, 1, 2*

4.	$\mathsf{Done}(\alpha \sqcap \beta) \wedge \alpha \sqcap \beta \sqsubseteq \beta \rightarrow \mathsf{Done}(\beta)$	***T3***
5.	$\alpha \sqcap \beta \sqsubseteq \beta$	*Def.*$\sqsubseteq$
6.	$\mathsf{Done}(\alpha \sqcap \beta) \rightarrow \mathsf{Done}(\beta)$	*PL, 4, 5*
7.	$\mathsf{Done}(\alpha \sqcap \beta) \rightarrow \mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta)$	*PL, 3, 6*

$\leftarrow$): *By induction:*
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg\mathsf{Done}(\alpha)$	***TempAx9***
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta) \rightarrow \mathsf{Done}(\alpha \sqcap \beta))$	*PL, 1*

Ind. Case:

1.	$[\alpha]\mathsf{Done}(\alpha) \wedge \alpha \sqcap \beta \sqsubseteq \alpha \rightarrow [\alpha \sqcap \beta]\mathsf{Done}(\alpha)$	***T3***
2.	$[\alpha]\mathsf{Done}(\alpha)$	***TempAx6***
3.	$\alpha \sqcap \beta \sqsubseteq \alpha$	*Def.* $\sqsubseteq$
4.	$[\alpha \sqcap \beta]\mathsf{Done}(\alpha)$	*PL, 1, 2 ,3*
5.	$[\beta]\mathsf{Done}(\beta)$	***TempAx6***
6.	$[\beta]\mathsf{Done}(\beta) \wedge \alpha \sqcap \beta \sqsubseteq \beta \rightarrow [\alpha \sqcap \beta]\mathsf{Done}(\beta)$	***T3***
7.	$\alpha \sqcap \beta \sqsubseteq \beta$	*Def.*$\sqsubseteq$
8.	$[\alpha \sqcap \beta]\mathsf{Done}(\beta)$	*PL,5, 6, 7*
9.	$[\alpha \sqcap \beta]\mathsf{Done}(\alpha \sqcap \beta)$	***TempAx6***
10.	$[\alpha \sqcap \beta]\mathsf{Done}(\alpha \sqcap \beta) \rightarrow [\alpha \sqcap \beta](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta) \rightarrow \mathsf{Done}(\alpha \sqcap \beta))$	*ML*
11.	$[\alpha \sqcap \beta](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta) \rightarrow \mathsf{Done}(\alpha \sqcap \beta))$	*MP, 10, 8, 4*
12.	$[\overline{\alpha}]\neg\mathsf{Done}(\alpha)$	***TempAx7***
13.	$[\overline{\beta}]\neg\mathsf{Done}(\beta)$	***TempAx7***
14.	$[\overline{\alpha} \sqcup \overline{\beta}](\neg\mathsf{Done}(\alpha) \vee \neg\mathsf{Done}(\beta))$	*PL &* ***T4****, 12, 13*
15.	$[\overline{\alpha \sqcap \beta}]\neg(\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta))$	*PL &* ***BA***
16.	$[\overline{\alpha \sqcap \beta}](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta) \rightarrow \mathsf{Done}(\alpha \sqcap \beta))$	*PL & ML, 15*
17.	$[\mathbf{U}](\mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta) \rightarrow \mathsf{Done}(\alpha \sqcap \beta))$	*PL,* ***A3****, BA, 11, 16*

T19:

1.	$\mathsf{Done}(\alpha \sqcup \beta) \wedge \mathsf{Done}(\overline{\alpha}) \rightarrow \mathsf{Done}((\alpha \sqcup \beta) \sqcap \overline{\alpha})$	***T18***
2.	$\mathsf{Done}(\alpha \sqcup \beta) \wedge \mathsf{Done}(\overline{\alpha}) \rightarrow \mathsf{Done}(\beta)$	*PL, BA, 1,* ***T16***

T20:

1.	$\beta \sqcap \overline{\alpha} \sqsubseteq \beta \wedge [\beta]\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\mathsf{Done}(\alpha)$	***T3***
2.	$\beta \sqcap \overline{\alpha} \sqsubseteq \overline{\alpha} \wedge [\overline{\alpha}]\neg\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\neg\mathsf{Done}(\alpha)$	***T3***
3.	$[\beta \sqcap \overline{\alpha}]\neg\mathsf{Done}(\alpha)$	*PL, 2,* ***TempAx7*** *& def.* $\sqsubseteq$
4.	$[\beta]\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\mathsf{Done}(\alpha)$	*PL, 1 & def.* $\sqsubseteq$

5.	$[\beta]\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\mathsf{Done}(\alpha) \wedge [\beta \sqcap \overline{\alpha}]\neg\mathsf{Done}(\alpha)$	*PL, 3, 4*
6.	$[\beta]\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\bot$	*ML, 5*
7.	$[\beta]\mathsf{Done}(\alpha) \rightarrow [\beta \sqcap \overline{\alpha}]\varphi$	*ML, 6*
8.	$\beta \sqcap \alpha \sqsubseteq \alpha \wedge [\alpha]\varphi \rightarrow [\beta \sqcap \alpha]\varphi$	*T3*
9.	$[\alpha]\varphi \rightarrow [\beta \sqcap \alpha]\varphi$	*PL, 8, $\sqsubseteq$ def.*
10.	$[\beta]\mathsf{Done}(\alpha) \wedge [\alpha]\varphi \rightarrow [\beta \sqcap \overline{\alpha}]\varphi \wedge [\beta \sqcap \alpha]\varphi$	*PL, 9, 7*
11.	$[\beta]\mathsf{Done}(\alpha) \wedge [\alpha]\varphi \rightarrow [(\beta \sqcap \overline{\alpha}) \sqcup (\beta \sqcap \alpha)]\varphi$	*PL, T4, 10*
12.	$[\beta]\mathsf{Done}(\alpha) \wedge [\alpha]\varphi \rightarrow [\beta]\varphi$	*PL, BA, 11*

■

Now, we prove the soundness of the system given above.

Theorem 10. *The axiomatic system is sound.*

Proof. *For the modal operators and the propositional part the proof is straightforward, just observing that the the semantics is exactly the same when a formula is evaluated. We give the proof for the novel axioms; the others are standard for temporal logics.*

TempAx1*: Let M be a model, $\pi = w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} \ldots$ a (maximal) path and i a given instant, suppose: $\pi, i, M \vDash \langle \mathbf{U} \rangle \top$. This means that: $\exists w' \in W, e_i \in \mathcal{E} : w_i \xrightarrow{e_i} w'$. But from the semantics, we get that $\pi, i, M \vDash (\mathsf{AN}\varphi \leftrightarrow [\mathbf{U}]\varphi)$ and therefore: $\pi, i, M \vDash \langle \mathbf{U} \rangle \top \rightarrow (\mathsf{AN}\varphi \leftrightarrow [\mathbf{U}]\varphi)$.*

TempAx2*: If $\pi, i, M \vDash [\mathbf{U}]\bot$, then $\nexists e_i \in \mathcal{E}, w' \in W : w \xrightarrow{e_i} w'$, and therefore $\pi[0..i]$ is a maximal trace, and $\#\pi = i$. Using the semantics of* AN *we get: $\pi, i, M \vDash \mathsf{AN}\varphi \leftrightarrow \varphi$.*

TempAx6*: Suppose: $\pi, i, M \nvDash [\alpha]\mathsf{Done}(\alpha)$, i.e., $\exists \pi' : \pi[0..i] \prec \pi'$, where $\pi' = w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{i-1}} w_i \xrightarrow{e_i} w'$ with $e_i \in I(\alpha)$, and $\pi', i+1, M \nvDash \mathsf{Done}(\alpha)$ which is a contradiction.*

TempAx7*: Similar to* ***TempAx6****.*

TempAx8*: We have $\forall e \in \mathcal{E} : e \notin \emptyset$, and therefore $\pi, i, M \vDash \neg\mathsf{Done}(\emptyset)$.*

TempAx9*: Suppose that $\pi, i, M \vDash \neg\mathsf{Done}(\mathbf{U})$; this fact implies that: $i = 0$ and from here we obtain: $\pi, i, M \vDash \neg\mathsf{Done}(\alpha)$.*

TempRule1*: For every path π, the sentence $\neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$ implies that φ must be true at instant 0, the other sentence says that φ must be true in any other instant of π, and therefore $\pi, i, M \vDash \varphi$.* ■

We use the tableaux system presented in chapter 5 to prove the completeness of the axiomatic system described above. A direct proof of completeness of the temporal system can be obtained applying the techniques presented in [GPSS80]; however, for our purposes it is more useful to prove the completeness of the Hilbert system using the tableaux system; we can take advantage of the relationship established between the two formal systems at the time we verify systems.

3.6 Introducing Violation Constants and Several Permissions

In practice we use violation predicates to indicate that a violation has occurred. A natural choice is to consider these predicates as a subset of the propositions of the vocabularies. However, sometimes it is useful to distinguish the violation propositions from the other propositions. We can extend the notion of the vocabularies to take into account this separation, adding a set V_0 of violation propositions (denoted by $\mathsf{v}_1, \ldots, \mathsf{v}_n$). Another useful extension is to consider several versions of permission, which allows us to have stratified norms and to avoid some paradoxes (we illustrate this with the examples of section 4).

Definition 19. *An extended language is a tuple* $\langle \Phi_0, \Delta_0, V_0, I_0 \rangle$ *where:*

- Φ_0 *is an enumerable set of proposition symbols.*
- Δ_0 *is a finite set of action symbols.*
- V_0 *is a finite set of violation propositions.*
- I_0 *is a finite index set for permissions.*

We assume that these sets are mutually disjoint. □

We consider the same formulae as in the earlier section, but we introduce one permission predicate for each index in the vocabulary. The formal definition of the formulae is:

- If $\varphi \in \Phi_0 \cup V_0$, then $\varphi \in \Phi$.
- If φ and ψ are formulae, then $\varphi \rightarrow \psi, \neg\psi \in \Phi$.
- If φ is a formula and α an action, then $[\alpha]\varphi$ is a formula.
- If α is an action and $i \in I_0$, then $\mathsf{P}^i_{\mathsf{w}}(\alpha)$, $\mathsf{P}^i(\alpha)$ and $\mathsf{O}^i(\alpha)$ are formulae.
- If φ and ψ are formulae, then $\mathsf{EN}\varphi$, $\mathsf{A}(\varphi\, \mathcal{U}\, \psi)$ and $\mathsf{E}(\varphi\, \mathcal{U}\, \psi) \in \Phi$.

For example, if we consider the vocabulary $\langle \{a, b\}, \{p\}, \{1, 2\} \rangle$, we have different permissions $\mathsf{P}^1()$ and $\mathsf{P}^2()$. Similarly with weak permission and obligation.

Given a language, because we have several violations in a language, we can define a predicate V_L which, roughly speaking, defines which violations are true and which are false in the current state. A state of violation then is a predicate $V_L = *\mathtt{v}_1 \wedge \cdots \wedge *\mathtt{v}_n$, where $\{\mathtt{v}_1, \ldots, \mathtt{v}_n\} = V_0$ and $*$ is blank or $\neg$, i.e., this predicate describes a state where a subset of the violations are true and other violations are false. On the other hand, we can use the predicate $\mathbf{V}_L = \mathtt{v}_1 \vee \cdots \vee \mathtt{v}_n$ to detect if in a state some violation is true. Note that $\neg\mathbf{V}_L$ says that no violation is true, and therefore this predicate allows us to define normative states (and normative traces).

Obviously, we can build a lattice of violation states (see below) and we can abstract the states of the system using this lattice of violations by forming equivalence classes of states using the predicates V_L.

As explained above, one important requirement to ask for is that allowed actions must not introduce new violations. This is called the *GGG* (Green-Green-Green) condition in [SC06]; as the name of this condition indicates, from a "green" state (without violations), performing an allowed transition (a "green" transition), we must reach a "green" state. This principle can be formally specified by the following (finite) set of axioms:

- $\neg\mathbf{V}_L \wedge \mathsf{P}^j(\alpha) \rightarrow [\alpha]\neg\mathbf{V}_L$ for every permission index j.

We denote by $G(L)$ this set of axioms. We take this principle further, and we say that, if from a state with a violation state V_C we perform an allowed action, then the state of violation is preserved or improved (i.e., it cannot happen that a violation, which is not true in a given state, becomes true after executing an allowed action). This stronger version of *GGG* can be formally specified by the following (finite) set of axioms:

- $\neg\mathsf{v}_j \wedge \mathsf{P}^i(\alpha) \rightarrow [\alpha]\neg\mathsf{v}_i$

We denote this set of axioms by $SG(L)$. Assuming $G(L)$ or its stronger version $SG(L)$ may allow one to simplify proofs and specifications (e.g., see the example of the coolers in the next chapter).

We also introduce some changes to the semantic structures to give the semantics of this variation of the logic. Basically, we consider one relational structure for each index in the vocabulary

Definition 20 (models). *Given a language* $L = \langle \Phi_0, \Delta_0, V_0, I_0 \rangle$, *an L-Structure is a tuple:* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\} \rangle$ *where:*

- $\mathcal{W}$ *is a set of worlds.*
- $\mathcal{E}$ *is a non-empty set of (names of) events.*
- $\mathcal{R}$ *is an* $\mathcal{E}$*-labeled relation between worlds. We require that, if* $(w, w', e) \in \mathcal{R}$ *and* $(w, w'', e) \in \mathcal{R}$*, then* $w' = w''$*, i.e.,* $\mathcal{R}$ *is functional when we fix the third element in the tuple.*
- $\mathcal{I}$*, is a function:*
  - *For every* $p \in \Phi_0 : \mathcal{I}(p) \subseteq \mathcal{W}$
  - *For every* $\alpha \in \Delta_0 : \mathcal{I}(\alpha) \subseteq \mathcal{E}$.

  In addition, the interpretation $\mathcal{I}$ *has to satisfy the following properties:*

  I.1 *For every* $\alpha_i \in \Delta_0$: $|\mathcal{I}(\alpha_i) - \bigcup\{\mathcal{I}(\alpha_j) \mid \alpha_j \in (\Delta_0 - \{\alpha_i\})\}| \leq 1$.

  I.2 *For every* $e \in \mathcal{E}$*: if* $e \in \mathcal{I}(\alpha_i) \cap \mathcal{I}(\alpha_j)$*, where* $\alpha_i \neq \alpha_j$ *and* $\alpha_j, \alpha_i \in \Delta_0$*, then:* $\cap\{\mathcal{I}(\alpha_k) \mid \alpha_k \in \Delta_0 \wedge e \in \mathcal{I}(\alpha_k)\} = \{e\}$.

  I.3 $\mathcal{E} = \bigcup_{\alpha_i \in \Delta_0} \mathcal{I}(\alpha_i)$.
- *each* $\mathcal{P}^i \subseteq \mathcal{W} \times \mathcal{E}$ *is a relation which indicates which event is permitted in which world with respect to permissions with index* i.

□

It is straightforward to extend the axiomatic system of definition 11 to these new formulae: we consider a separate version of axioms **A5 – A13** for each index $i \in I_0$, i.e., we have:

A5. $\mathsf{P}^i(\emptyset)$

A6. $\mathsf{P}^i(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}^i(\alpha) \wedge \mathsf{P}^i(\beta)$

A7. $\mathsf{P}^i(\alpha) \vee \mathsf{P}^i(\beta) \rightarrow \mathsf{P}^i(\alpha \sqcap \beta)$

A8. $\neg \mathsf{P}^i_{\mathsf{w}}(\emptyset)$

A9. $\mathsf{P}^i_{\mathsf{w}}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha) \vee \mathsf{P}^i_{\mathsf{w}}(\beta)$

A10. $\mathsf{P}^i_{\mathsf{w}}(\alpha \sqcap \beta) \rightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha) \wedge \mathsf{P}^i_{\mathsf{w}}(\beta)$

A11. $\mathsf{P}^i(\alpha) \wedge \alpha \neq_{act} \emptyset \rightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha)$

A12. $\mathsf{P}^i_{\mathsf{w}}(\gamma) \rightarrow \mathsf{P}^i(\gamma)$, where $\gamma \in At(\Delta_0)$

A13. $\mathsf{O}^i(\alpha) \leftrightarrow \mathsf{P}^i(\alpha) \wedge \neg \mathsf{P}^i_{\mathsf{w}}(\overline{\alpha})$

where $i \in I_0$.

The different versions of the deontic predicates allow us to have stratified levels of norms. It is straightforward to extend the proofs of soundness and completeness for this variation of the logic. For each index i, we can repeat the proof of soundness given in section 3.3, and in the canonical model we define a relationship:

$$\mathcal{P}^i_{\mathcal{C}} \stackrel{\mathsf{def}}{=} \bigcup \{\mathcal{P}^i_{w,\alpha} \mid w \in \mathcal{W}_{\mathcal{C}} \wedge \mathsf{P}^i(\alpha) \in w\},$$

where: $\mathcal{P}^i_{w,\alpha} \stackrel{\mathsf{def}}{=} \{(w, [\alpha']_{BA}) \mid [\alpha']_{BA} \in \mathcal{I}_{\mathcal{C}}(\alpha)\}$. For each index i, the proof proceeds as before.

Having several versions of permissions is useful in practice, in particular when we have contrary-to-duty statements. Consider, for example, the *gentle killer* paradox:

- it is forbidden to kill.
- if you kill, you ought to kill gently.
- you kill.

In SDL the formalization of this paradox gives us an inconsistent set of sentences [SP94]. We can formalize this scenario as follows:

- $\mathsf{F}^1(\mathtt{k})$
- $\mathtt{kg} \sqsubseteq \mathtt{k}$
- $\mathtt{nk} \sqcap \mathtt{k} =_{act} \emptyset$
- $\mathsf{O}^2(\mathtt{nk} \sqcup \mathtt{kg})$
- $\mathsf{ANDone}(\mathtt{k})$

We consider the actions k (kill), kg (kill gently) and nk (not kill). The first axiom says that it is forbidden to kill. The second formula says that the action of kill gently (kg) is a way of killing. The third axiom expresses that killing and not killing are disjoint actions. The fourth formula says that, if we will kill, then we have to kill gently. In the last formula, we use the Done() operator to state that we will kill

(which is expressed saying that the next action is to kill). In this case we consider in the vocabulary two indexes: 1 and 2 pointing out that there are two different levels of norms in the specification. In contrast to standard deontic logic, these sentences are not contradictory in our setting. For example the structure illustrated in figure 3.4 is a model of this set of sentences. The structure in this figure has three states w, w_1, w_2,

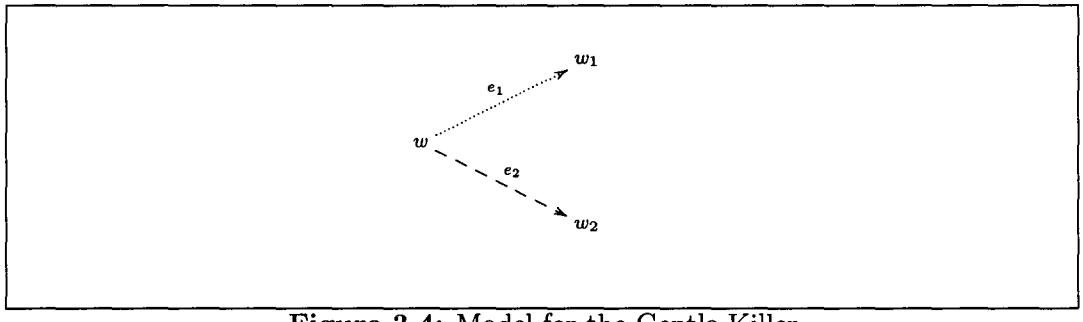


Figure 3.4: Model for the Gentle Killer

and we have $\mathcal{I}(\mathbf{k}) = \{e_1, e_2\}$, $\mathcal{I}(\mathbf{kg}) = \{e_1\}$ and $\mathcal{I}(\mathbf{nk}) = \{e_3\}$. The labels on the transitions indicate which actions are executed and which are not in each transition. The upper dashed arrow denotes a transition that is forbidden with respect to index 1 but not for index 2. The lower dashed arrow indicates an arrow which is forbidden for both indexes.

3.7 Summary

In this chapter we have presented a propositional deontic logic which is a formal basis for the rest of the thesis. This logic has the standard modal predicates (possibility and necessity) over an expressive algebra of actions (a boolean algebra), deontic predicates (permissions and obligations) and we also enriched this logic with branching time operators. The main properties of this formal system were investigated, in particular we proved that the given axiomatic system is *sound*, *strongly complete* and *decidable*. Furthermore, the obtained system is compact, which is an improvement over the deontic logic in boolean operators described in [Bro03] and [GP90].

The completeness proof given in section 3.4 introduces a technique that we also use to prove some properties about the tableaux system presented in chapter 5; we use the atoms of the boolean algebra of terms to label the transitions of the canonical model; these atoms describe which action is performed and which is not during a transition. We take this idea further in the tableaux system to produce *counterexamples* of specifications.

Finally, it is important to stress (once more) that the way in which the deontic predicates are defined differs from what is done in other deontic action logics [Mey88], [Bro03], where deontic predicates are defined using the modalities. Here there is no *a priori* relation between the two. They are intended to model different aspects of a system: modalities are aimed at describing a system in a pre/post-condition way, whereas deontic predicates express prescriptions about how the system should behave. From our point of view, the two notions differ and they must be specified by different formal constructions. (Here we follow the main ideas introduced in [KM85].) This separation between these concepts gives more freedom at the time of specifying systems. Note that, in case they are needed, some relationship between deontic constructions and modalities can be introduced by means of extra axioms, see chapter 7 for some examples.

Chapter 4

Some Examples

The specification of systems using modal logics (and related formalisms) has been well studied. For example, the FOREST reports ([KQM91, Ken91, Rya90, RFM91]) discuss different applications for the *modal action logic* presented there. (For instance in [KQM91] a system for a library is discussed in detail.) On the other hand, *dynamic logics* [HKT00] have been used in practice to reason about complex examples; this logic is similar to *Hoare logic* [Hoa69] and it can be used to specify programs in pre/post-condition style. In this chapter we present five case studies to show how the constructs presented in chapter 3 can be used in practice to specify and verify fault-tolerant systems, going beyond the work cited above.

First, we take the classical example of dining philosophers of Dijkstra [Dij71], but we add some features to it, so that every process has the possibility of crashing for an indefinite amount of time. This modification allows us to illustrate how we can use our logic to reason about fault-tolerance. We have outlined this example in [CM07b]; here we analyze in detail a formalization of this problem, and we focus on proving some of its properties to show that the deductive system is usable in practice.

As a second example, we exhibit a specification of the Muller c-element, a delay insensitive circuit. This circuit is tolerant to delays in its inputs; in this case we show how obligations can be used to distinguish between normal and abnormal behaviour. The third example is a simple train system; fault-tolerance is important in these kinds of systems since failures may cause tragic accidents. In this example, we illustrate how contrary-to-duty statements might arise in practice. We present two further examples: the byzantine generals; this is a classic example of fault-tolerance [LSP82]. We give a specification of this problem where we use the deontic predicates to distinguish between good and bad behaviour, and how stratified norms are used for reasoning

about different kinds of faults that may happen during the execution of the system. The last example is a specification of a microprocessor which has two coolers to keep the temperature low; the redundancy in the system allows it to support failures in one of the coolers. In this example, we use stratified norms to model a contrary-to-duty statement.

4.1 The Diarrheic Philosophers

The problem which we shall investigate below can be described informally as follows:

Example 1. *In some college, a fixed number of philosophers are dedicated to thinking about different problems. Because each philosopher must eat to survive, the college has a (circular) table which contains a big bowl of pasta. Each philosopher has a seat and two forks, one for each hand. But, because of budgetary reasons, neighbouring philosophers have to share forks. In addition, the pasta could be contaminated (usually philosophers think for a long time and so the pasta may develop some dangerous bacteria, probably because they have to share forks!) and therefore philosophers could get a stomach ache and then they must go to the bathroom. A problem arises, of course, when a philosopher goes to the bathroom with some forks in his hands. A philosopher may come back, or not, from the bathroom (the details are left to the reader's imagination).*

The main point to make about the following specification is the way in which the possible faults (i.e., when a given philosopher goes to the bathroom) are modeled. We model this scenario as a system violation; we will see that two possible violations can be defined. First, if a philosopher goes to the bathroom with two forks in his hands, then we obviously have the worst situation that could happen (this situation will be modeled with the predicate $\mathtt{v}_1 \wedge \neg \mathtt{v}_2$, the proposition $\mathtt{v}_1$ becomes true when a philosopher takes one or two forks with him to the bathroom). The other is when a philosopher only takes one fork with him (we use the predicate $\mathtt{v}_2$ to model this); in this case, we will prove that undesirable blocking is not possible. Additionally, we will see that when violation $\mathtt{v}_1$ upgrades to a violation $\mathtt{v}_2$ the system can avoid suffering some undesired blocking. Using this specification, a very interesting set of properties can be proved, some of which will be shown later.

A philosopher in the bathroom is different from a philosopher that is eating. An eating philosopher is a philosopher performing an allowed action, which is not in an error state. Instead, while a philosopher in the bathroom is a philosopher in an error state. In a computing system, an eating philosopher could be a printer printing

a document, while a philosopher in the bathroom could be a printer out of toner, or printing something different from what it is intended to print. In the original problem there is no difference between processes crashing or exhibiting a malicious behaviour, on the one hand, and processes having a normal behaviour, on the other. Note also that the eating philosophers always hold onto the two forks, while the philosophers in the bathroom might hold only one fork; this scenario may prevent deadlock-free solutions from working. For example, solutions introducing asymmetry into the behaviour of philosophers (odd-numbered philosophers take the right fork first, and even-numbered philosophers take the left fork first [MK99]) could fail to avoid deadlock.

We suppose in the example that the eating philosophers will eat for a finite amount of time. The difference between error states and normal states is useful since, in the last cases drastic actions can be taken to restore the normal state of the system (e.g., restarting the printer or replacing the toner).

4.1.1 Axioms

We start defining the language in which the specification is expressed; consider the following set of primitive actions:

$$\Delta_0 = \{\mathtt{i.up_L}, \mathtt{i.up_R}, \mathtt{i.down_R}, \mathtt{i.down_L}, \mathtt{i.getbad}, \mathtt{i.getthk}, \mathtt{i.gethungry}\}$$

where $0 \leq \mathtt{i} \leq n$, for some n. Intuitively, $\mathtt{i.up_L}$ ($\mathtt{i.up_R}$) is the action of philosopher $\mathtt{i}$ picking up the left (right) fork; $\mathtt{i.down_L}$ and $\mathtt{i.down_R}$ are the inverses. On the other hand, $\mathtt{i.getbad}, \mathtt{i.getthk}$ and $\mathtt{i.gethungry}$ are executed when philosopher $\mathtt{i}$ gets sick, starts thinking or gets hungry, respectively. We also consider the following set of predicates:

$$\Phi_0 = \{\mathtt{i.thk}, \mathtt{i.eating}, \mathtt{i.hungry}, \mathtt{i.bath}, \mathtt{i.has_L}, \mathtt{i.has_R}, \mathtt{fork_i.up}, \mathtt{fork_i.down}\}$$

where $\mathtt{i.eating}$ tells us if philosopher $\mathtt{i}$ is eating, $\mathtt{i.hungry}$ if the philosopher is hungry, $\mathtt{i.bath}$ will be true when $\mathtt{i}$ is in the bathroom, $\mathtt{i.has_L}$ and $\mathtt{i.has_R}$ allow us to know if $\mathtt{i}$ has the left or right fork in his hands. And $\mathtt{fork_i.up}, \mathtt{fork_i.down}$ will be true if fork $\mathtt{i}$ is up or down, respectively. We consider two violations per philosopher: $V_0 = \{\mathtt{i.v_1}, \mathtt{i.v_2}\}$

We have several axioms, but note that many of them are *frame axioms* (i.e., they express which part of the system is not changed by the actions, these requirements are usually implicit); stating these axioms can be avoided if a more abstract language of specification is used, and then translated to our logic. Let us establish the initial conditions:

Phil1. $\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge_{0\leq i\leq n} \mathtt{i.thk}) \wedge (\bigwedge_{0\leq i\leq n} \mathtt{fork_i.down}) \wedge (\bigwedge_{0\leq i\leq n} \neg\mathtt{i.v_1} \wedge \neg\mathtt{i.v_2})$

That is, at the beginning all the philosophers are thinking, all the forks are down and there are no violations. We also need to express that some states or some actions are disjoint:

Phil2. $\mathtt{fork_i.down} \oplus \mathtt{fork_i.up}$
Phil3. $\mathtt{i.eating} \oplus \mathtt{i.thk} \oplus \mathtt{i.hungry} \oplus \mathtt{i.bath}$
Phil4. $(\mathtt{i.thk} \vee \mathtt{i.bath}) \rightarrow [\mathtt{i.up_L} \sqcup \mathtt{i.up_R}]\bot$

Here the symbol $\oplus$ denotes the strict version of $\vee$. Axiom **Phil2** says that each fork can either be up or down but not both. **Phil3** is similar but expressing a disjointness condition on philosopher states. **Phil4** says that a thinking or ill philosopher cannot pick up any fork.

Phil5. $([\mathtt{i.up_R}]\mathtt{i.has_R}) \wedge ([\mathtt{i.up_L}]\mathtt{i.has_L})$
$\wedge([\mathtt{i.down_R}]\neg\mathtt{i.has_R}) \wedge ([\mathtt{i.down_L}]\neg\mathtt{i.has_L})$
Phil6. $(\neg\mathtt{i.has_R} \rightarrow \boxed{\mathtt{i.up_R}}\neg\mathtt{i.has_R}) \wedge (\neg\mathtt{i.has_L} \rightarrow \boxed{\mathtt{i.up_L}}\neg\mathtt{i.has_L})$
$(\mathtt{i.has_R} \rightarrow \boxed{\mathtt{i.down_R}}\mathtt{i.has_R}) \wedge (\mathtt{i.has_L} \rightarrow \boxed{\mathtt{i.down_L}}\mathtt{i.has_L})$
$\wedge(\neg\mathtt{i.has_L} \rightarrow [\mathtt{i.down_L}]\bot) \wedge (\neg\mathtt{i.has_R} \rightarrow [\mathtt{i.down_R}]\bot)$
Phil7. $(\mathtt{i.has_L} \rightarrow [\mathtt{i.getthk}]\mathsf{Done}(\mathtt{i.down_L}))$
$\wedge(\mathtt{i.has_R} \rightarrow [\mathtt{i.getthk}]\mathsf{Done}(\mathtt{down_R}))$
Phil8. $([\mathtt{i.getthk}]\mathtt{i.thk}) \wedge (\neg\mathtt{i.thk} \rightarrow \boxed{\mathtt{i.getthk}}\neg\mathtt{i.thk})$

The first axiom models the behaviour of the `i.up` and `i.down` actions. On the other hand, axiom **Phil6** is a frame axiom; it says that only the actions $\mathtt{i.up_L}$ ($\mathtt{i.down_L}$) and $\mathtt{i.up_R}$ ($\mathtt{i.down_R}$) can make predicates $\mathtt{i.has_L}$ and $\mathtt{i.has_R}$ become true (false). **Phil7** and **Phil8** are similar but they model the behaviour of the `i.getthk` action. The next couple of axioms express some behaviour of the forks.

Phil9. $\mathtt{fork_i.up} \rightarrow [\mathtt{i.up_L} \sqcup (\mathtt{i}+1).\mathtt{up_R}]\bot$
Phil10. $\mathtt{fork_i.up} \leftrightarrow ((\mathtt{i}+1).\mathtt{has_R} \vee \mathtt{i.has_L})$

Axiom **Phil9** expresses that, if a fork is up, then none of the corresponding philosophers can take it (here "+" denotes addition modulo $n+1$). The other formula establishes that a fork `i` is up if and only if the philosopher `i`, or $\mathtt{i}+1$, has it in his hands.

We also need to predicate that two neighbouring philosophers cannot take the same fork at the same time; this is expressed by the following axiom:

Phil11. $\mathtt{i.up_L} \sqcap (\mathtt{i}+1).\mathtt{up_R} = \emptyset$

In an implementation, we could use a *semaphore* to ensure this requirement.

The next set of axioms models the behaviour of the i.gethungry action.

Phil12. $(\mathtt{i.gethungry} \sqcap (\mathtt{i}+1).\mathtt{gethungry}) = \emptyset$

Phil13. $\mathtt{i.thk} \wedge \neg(\mathtt{i}-1).\mathtt{hungry} \wedge \neg(\mathtt{i}+1).\mathtt{hungry} \wedge \mathtt{fork_i.down} \wedge \mathtt{fork_{i+1}.down} \rightarrow \langle \mathtt{i.gethungry} \rangle \top$

Phil14. $(\mathtt{i.thk} \wedge (((\mathtt{i}-1).\mathtt{hungry} \vee (\mathtt{i}+1).\mathtt{hungry}) \vee ((\mathtt{i}-1).\mathtt{v_1} \wedge \mathtt{i.has_L}) \vee ((\mathtt{i}+1).\mathtt{v_1} \wedge \mathtt{i.has_R}))) \rightarrow [\mathtt{i.gethungry}]\bot$

Phil15. $\mathtt{i.hungry} \wedge (\mathtt{fork_i.down} \vee \mathtt{i.has_L}) \wedge (\mathtt{fork_{i+1}.down} \vee \mathtt{i.has_R}) \rightarrow \mathsf{ANi.eating}$

Phil16. $\mathtt{i.hungry} \wedge ((\mathtt{i}+1).\mathtt{v_1} \wedge (\mathtt{i}+1).\mathtt{has_R}) \wedge ((\mathtt{i}-1).\mathtt{v_1} \wedge (\mathtt{i}-1).\mathtt{has_L}) \rightarrow \mathsf{ANi.thk}$

Phil17. $([\mathtt{i.gethungry}]\mathtt{i.hungry}) \wedge (\neg\mathtt{i.hungry} \rightarrow \overline{[\mathtt{i.gethungry}]}\neg\mathtt{i.hungry})$

The first formula establishes that no two neighbouring philosophers can get hungry at the same time; if we allow concurrency here, it will give us some problems. Again some mechanism for mutual exclusion is needed in the implementation. **Phil14** expresses that if some philosopher is getting hungry and some neighbour is already in that state, the philosopher has to wait. Obviously, this specification exhibits a starvation problem (and this may be the best reason for food poisoning!); to avoid this, a priority queue is needed. For simplicity, we do not deal with this problem in this example.

Phil13 tells us when a philosopher will have the possibility of getting hungry. Axiom **Phil15** says that if a philosopher is hungry and he can take both forks, then he will start to eat. The last two axioms in this set specify the behaviour of i.gethungry and some frame conditions.

The following set of sentences specify what happens when a philosopher is eating,

Phil18. $\mathtt{i.eating} \leftrightarrow (\mathtt{i.has_L} \wedge \mathtt{i.has_R} \wedge \neg\mathtt{i.bath})$

Phil19. $\mathtt{i.eating} \rightarrow [\mathtt{i.up_L} \sqcup \mathtt{i.up_R} \sqcup \mathtt{i.getbetter} \sqcup \mathtt{i.gethungry}]\bot$

Phil20. $\mathtt{i.eating} \rightarrow [\mathsf{U}]\mathsf{Done}(\mathtt{i.getthk} \sqcup \mathtt{i.getbad})$

Phil21. $\mathtt{i.eating} \leftrightarrow \mathsf{O}(\mathtt{i.down_L} \sqcap \mathtt{i.down_R})$

Axiom **Phil18** says that a philosopher is eating iff he has both forks and he is not in the bathroom. Axiom **Phil19** restricts the actions that can be done when a philosopher is eating. **Phil20** says, if a philosopher is eating, then he can only start thinking again or getting sick. Of course, in a more complicated specification philosophers might eat for an undefined amount of time (here they only eat for one

time unit). The amount of time that philosophers eat is not important for our present purposes. The last axiom establishes an obligation about the release of the forks. Some explanation is needed about this obligation; note that the deontic predicate here (**Phil21**) says what happens ideally or normally, but perhaps this condition may be violated. The important point here is that the obligation predicate allows us to differentiate an ideal or normal scenario from one that is not (note that $\mathsf{O}(\alpha) \rightarrow \neg \mathsf{P}_{\mathsf{w}}(\overline{\alpha})$, where the negation of a weak permission can be read as a prohibition), and this is a strong benefit of deontic logic. Note also that these deontic restrictions will be reflected in the semantic structures, where some arcs will be green colored (allowed transitions) and others red colored (forbidden transitions, e.g., when a philosopher does not put down the forks). This classification of transitions allows us to perform different analyses on the semantic models (e.g., to investigate properties that are preserved by green transitions).

The `i.getbad` action can be modeled as follows:

Phil22. $([\mathtt{i.getbad}]\mathtt{i.bath}) \wedge (\neg \mathtt{i.bath} \rightarrow \boxed{\mathtt{i.getbad}}\neg \mathtt{i.bath})$
Phil23. $\mathtt{i.bath} \rightarrow \boxed{\mathtt{i.getthk}}\mathtt{i.bath}$

We note axiom **Phil23**, which says that if a philosopher gets better, then he goes to the thinking state (and therefore he has to free up those forks that he has in his hands).

Finally, we present a collection of axioms for modeling the notion of violation. The predicates $\mathtt{i.v}_1, \mathtt{i.v}_2$ are used for this purpose; $\mathtt{i.v}_1$ becomes true when philosopher `i` (after eating) does not release both forks. $\mathtt{i.v}_2$ is a refinement of $\mathtt{i.v}_1$; it is true when philosopher `i` only releases one fork, but holds onto the remaining fork. These variables allow us to reason about the situations in which an undesirable blocking becomes possible, because some norm has been violated (e.g., that in **Phil21**), and when we can avoid it (so-called recovery steps). Interestingly, we can predicate about recovery from bad scenarios (for example, when $\mathtt{i.v}_1$ "upgrades" to $\mathtt{i.v}_2$). The axioms are:

V1. $\neg \mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}) \rightarrow$
$(\boxed{\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}}\mathtt{i.v}_1) \wedge ([\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}]\neg \mathtt{i.v}_1)$
V2. $\neg \mathtt{i.v}_1 \wedge \neg \mathsf{O}(\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}) \rightarrow [\mathtt{U}]\neg \mathtt{i.v}_1$
V3. $\mathtt{i.v}_2 \leftrightarrow \mathtt{i.v}_1 \wedge (\neg \mathtt{i.has}_{\mathtt{L}} \oplus \neg \mathtt{i.has}_{\mathtt{R}})$
V4. $(\mathtt{i.v}_1 \rightarrow [\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}]\neg \mathtt{i.v}_1) \wedge (\mathtt{i.v}_1 \wedge \neg \mathtt{i.v}_2 \rightarrow \boxed{\mathtt{i.down}_{\mathtt{L}} \sqcap \mathtt{i.down}_{\mathtt{R}}}i.v_1)$
V5. $\mathtt{i.v}_2 \rightarrow \boxed{\mathtt{i.down}_{\mathtt{L}} \sqcup \mathtt{i.down}_{\mathtt{R}}}\mathtt{i.v}_2$
V6. $((\mathtt{i.v}_2 \wedge \neg \mathtt{i.has}_{\mathtt{L}}) \rightarrow [\mathtt{i.down}_{\mathtt{R}}](\neg \mathtt{i.v}_2 \wedge \neg \mathtt{i.v}_1)) \wedge$
$((\mathtt{i.v}_2 \wedge \neg \mathtt{i.has}_{\mathtt{R}}) \rightarrow [\mathtt{i.down}_{\mathtt{R}}](\neg \mathtt{i.v}_2 \wedge \neg \mathtt{i.v}_1))$

The first axiom defines when $\mathtt{i.v_1}$ can become true, that is, when philosopher i is obligated to put down both forks but he does not do so; otherwise $\mathtt{i.v_1}$ is false. **V2** is needed to say that the other actions do not affect the violation marker $\mathtt{i.v_1}$. **V3** defines $\mathtt{i.v_2}$; it is true if and only if the philosopher has only one fork and violation $\mathtt{i.v_1}$ is true. Intuitively, this occurs when a philosopher only takes one fork with him to the bathroom, or perhaps if he puts down a fork after getting in a violation state. Note that it is possible for a philosopher to put down a fork while in the bathroom, although in this model this philosopher will stay in the bathroom while he puts down a fork; there are no constraints in the specification to prevent this or to encourage it. (This may be considered an example of underspecification.) We may be of the view that he leaves the fork under the bathroom door and somebody will pick it up. (For the sake of simplicity we abstract these details from our model). **V4** establishes that putting down both forks is a recovery action for $\mathtt{i.v_1}$, and, if we are not in a violation of type $\mathtt{i.v_2}$, then doing something different will leave us in the same violation state. This has the important consequence that, when we are in a violation of type $\mathtt{i.v_2}$, we can recover from $\mathtt{i.v_1}$ and $\mathtt{i.v_2}$; exactly this is specified by axiom **V6**.

Note that the deontic part of the specification is simple: we only have obligations when the philosophers are eating. Of course, more complicated scenarios could be thought of; we can add a second obligation (a contrary-to-duty obligation) when a philosopher is in the bathroom (e.g., to release at least one fork). As we said before, we keep our example as simple as possible, and we just illustrate how the logic is used in practice. We only impose one obligation and describe how violations follow when this obligation is not fulfilled.

On the other hand, axioms: **V1** and **V2** impose a relationship between violations and obligations. Although this relationship is not as strong as the ones usually imposed in dynamic deontic logic, see section 3.2, we argue that a weak relationship is better in fault-tolerance (which is also noted by Sergot in [SC06]). For example, note that in each scenario we have a different set of possible violations: $\mathtt{v_1}, ..., \mathtt{v_n}$, and then we can define: $\mathtt{V} \equiv \mathtt{v_1} \vee ... \vee \mathtt{v_n}$, and therefore a state free of violations is one in which $\neg\mathtt{V} \equiv \neg\mathtt{v_1} \wedge ... \wedge \neg\mathtt{v_n}$ is true. However, defining, e.g., $\mathsf{P}(\alpha) \stackrel{\mathsf{def}}{\Longleftrightarrow} \langle\alpha\rangle\neg\mathtt{V}$ (as done by Meyer [Mey88]) is not always a good option: in the example presented above a philosopher could be in the bathroom with the two forks (in a violation state), and with this strong definition putting down only one fork is forbidden (not allowed) because the philosopher will still be in a violation state. The point is that this situation is not desirable in our scenario: the action of putting down only one fork allows the system to make some progress (allowing other philosophers to eat). As noted above, allowed actions, in a violation state, could carry forward violations. More examples of this kind can be found in [SC06].

Note that the *GGG* condition (see 3.2) could be introduced as a specification

dependent axiom, i.e., it must be instantiated in each specification (in a similar way to *locality axioms*, see [FM92]). Using this axiom, formula **V1** can be simplified.

4.1.2 Some Properties

In this section, we prove some important properties about the specification given above. Firstly, we show some important lemmas, which allow us to modularize the proofs. The proofs show how the logic can be used in practice, although it is important to remark that we can build software tools which assist the software designer in this task. In the next chapter we present a tableaux system which can be used for proving properties from specifications.

Our first lemma establishes that no two neighbouring philosophers can have the same fork at the same time.

Lemma 3. $\vdash_{\mathbf{Phil}} \neg((\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \mathtt{i}.\mathtt{has}_{\mathtt{L}})$.
Proof. *We use induction to prove it.*
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \mathtt{fork}_{\mathtt{i}}.\mathtt{down}$	*PL,* ***Phil1***
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \wedge \neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}})$	*PL, 1,* ***Phil2, Phil10***
3.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \vee \neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}})$	*PL,2*

Ind. Case:

1.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [(\mathtt{i}+1).\mathtt{up}_{\mathtt{R}}](\mathtt{i}+1).\mathtt{has}_{\mathtt{R}}$	*PL,* ***Phil5***
2.	$\mathtt{i}.\mathtt{up}_{\mathtt{L}} \sqcap (\mathtt{i}+1).\mathtt{up}_{\mathtt{R}} = \emptyset$	***Phil11***
3.	$(\mathtt{i}+1).\mathtt{up}_{\mathtt{R}} \sqsubseteq \overline{\mathtt{i}.\mathtt{up}_{\mathtt{L}}}$	*BA, 2*
4.	$\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow \overline{[\mathtt{i}.\mathtt{up}_{\mathtt{L}}]}\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}}$	*PL,* ***Phil6***
5.	$\overline{[\mathtt{i}.\mathtt{up}_{\mathtt{L}}]}\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [\mathtt{i}+1.\mathtt{up}_{\mathtt{R}}]\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}}$	*PL,* ***T3****, 3*
6.	$\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [\mathtt{i}+1.\mathtt{up}_{\mathtt{R}}]\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}}$	*PL, 4, 5*
7.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [(\mathtt{i}+1).\mathtt{up}_{\mathtt{R}}]((\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}})$	*ML, 6, 1*
8.	$\neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [\mathtt{i}.\mathtt{up}_{\mathtt{L}}]\mathtt{i}.\mathtt{has}_{\mathtt{L}}$	*PL,* ***Phil5***
9.	$\mathtt{i}.\mathtt{up}_{\mathtt{L}} \sqsubseteq \overline{\mathtt{i}+1.\mathtt{up}_{\mathtt{R}}}$	*BA, 2*
10.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \rightarrow \overline{[(\mathtt{i}+1).\mathtt{up}_{\mathtt{R}}]}\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{L}}$	*PL,* ***Phil6***
11.	$\overline{[(\mathtt{i}+1).\mathtt{up}_{\mathtt{R}}]}\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \rightarrow [\mathtt{i}.\mathtt{up}_{\mathtt{L}}]\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{L}}$	*PL,* ***T3****, 9*
12.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \rightarrow [\mathtt{i}.\mathtt{up}_{\mathtt{L}}]\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}}$	*PL, 10, 11*
13.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow [\mathtt{i}.\mathtt{up}_{\mathtt{L}}](\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \mathtt{i}.\mathtt{has}_{\mathtt{L}})$	*PL, 12, 8*
14.	$\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \rightarrow$ $[\mathtt{i}.\mathtt{up}_{\mathtt{L}} \sqcup (\mathtt{i}+1).\mathtt{up}_{\mathtt{R}}](\neg(\mathtt{i}+1).\mathtt{has}_{\mathtt{R}} \vee \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}})$	*PL,* ***T4****, 6, 12*

15.	$\neg \mathtt{i.has_L} \rightarrow \boxed{\overline{\mathtt{i.up_L}}}\neg \mathtt{i.has_L}$	*PL, **Phil6***
16.	$\neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow \boxed{\overline{(\mathtt{i}+1).\mathtt{up_R}}}\neg(\mathtt{i}+1).\mathtt{has_R}$	*PL, **Phil6***
17.	$\neg \mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow$ $\boxed{\mathtt{i.up_L}} \sqcup \boxed{\overline{(\mathtt{i}+1).\mathtt{up_R}}}(\neg \mathtt{i.has_L} \vee \neg(\mathtt{i}+1).\mathtt{has_R})$	*PL, **T4**, 15, 16*
18.	$\neg \mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow [\mathbf{U}](\neg \mathtt{i.has_L} \vee \neg(\mathtt{i}+1).\mathtt{has_R})$	*PL, **BA**, **T4**, 14, 17*
19.	$\mathtt{i.has_L} \rightarrow \mathtt{fork_i.up}$	*PL, **Phil10***
20.	$\mathtt{fork_i.up} \rightarrow [(\mathtt{i}+1).\mathtt{up_R}]\bot$	*PL, **Phil9**, **T3***
21.	$\mathtt{i.has_L} \rightarrow [(\mathtt{i}+1).\mathtt{up_R}]\bot$	*PL, 19, 20*
22.	$\mathtt{fork_i.up} \rightarrow [\mathtt{i.up_L}]\bot$	*PL, **Phil9**, **T3***
23.	$\mathtt{i.has_L} \rightarrow [\mathtt{i.up_L}]\bot$	*PL, 19, 22*
24.	$\mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow [(\mathtt{i}+1).\mathtt{up_R}]\bot$	*PL, 21*
25.	$\bot \rightarrow \neg(\mathtt{i}+1).\mathtt{has_R}$	*PL*
26.	$\mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow [(\mathtt{i}+1).\mathtt{up_R}]\neg(\mathtt{i}+1).\mathtt{has_R}$	*ML, 24, 25*
27.	$\mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow \boxed{\overline{(\mathtt{i}+1).\mathtt{up_R}}}\neg(\mathtt{i}+1).\mathtt{has_R}$	*PL, 16*
28.	$\neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow \neg(\mathtt{i}+1).\mathtt{has_R} \vee \neg \mathtt{i.has_L}$	*PL*
29.	$\mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow [\mathbf{U}]\neg(\mathtt{i}+1).\mathtt{has_R}$	*PL, **BA**. 26, 27, T4*
30.	$\mathtt{i.has_L} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow [\mathbf{U}](\neg(\mathtt{i}+1).\mathtt{has_R} \vee \neg \mathtt{i.has_L})$	*ML, 28, 29*
31.	$(\mathtt{i}+1).\mathtt{has_R} \rightarrow \mathtt{fork_i.up}$	*ML, Phil10*
32.	$(\mathtt{i}+1).\mathtt{has_R} \rightarrow [\mathtt{i.up_L}]\bot$	*PL, **Phil9**, 31, **T3***
33.	$(\mathtt{i}+1).\mathtt{has_R} \wedge \neg \mathtt{i.has_L} \rightarrow [\mathtt{i.up_L}]\bot$	*PL, 32*
34.	$(\mathtt{i}+1).\mathtt{has_R} \wedge \neg \mathtt{i.has_L} \rightarrow \boxed{\overline{\mathtt{i.up_L}}}\neg \mathtt{i.has_L}$	*PL, 15*
35.	$(\mathtt{i}+1).\mathtt{has_R} \wedge \neg \mathtt{i.has_L} \rightarrow \boxed{\overline{\mathtt{i.up_L}}}(\neg \mathtt{i.has_L} \vee \neg(\mathtt{i}+1).\mathtt{has_R})$	*ML, 34*
36.	$(\mathtt{i}+1).\mathtt{has_R} \wedge \neg \mathtt{i.has_L} \rightarrow [\mathtt{i.up_L}](\neg \mathtt{i.has_L} \vee (\mathtt{i}+1).\mathtt{has_R})$	*ML, 33*
37.	$(\mathtt{i}+1).\mathtt{has_R} \wedge \neg \mathtt{i.has_L} \rightarrow [\mathbf{U}](\neg \mathtt{i.has_L} \vee \neg(\mathtt{i}+1), \mathtt{has_R})$	*BA, PL, 36, 35, T4*
38.	$(\neg(\mathtt{i}+1).\mathtt{has_R} \vee \neg \mathtt{i.has_L}) \rightarrow [\mathbf{U}](\neg \mathtt{i.has_L} \vee \neg(\mathtt{i}+1).\mathtt{has_R})$	*PL, 18, 30, 37*

■

The second lemma says that, if a philosopher is in a violation, then he has some fork in his hands:

Lemma 4. $\vdash_{\mathbf{Phil}} \mathtt{i.v_1} \rightarrow (\mathtt{i.has_R} \vee \mathtt{i.has_L})$
Proof. *See appendix A.* ■

The next lemma tells us that, if it is not the case that philosopher i is in violation, then in the next state either he will not be in a violation or he will not be eating. That is, if a philosopher goes into a violation state, then he cannot be eating.

Lemma 5. $\vdash_{\mathbf{Phil}} \neg \mathtt{i.v_1} \rightarrow [\mathbf{U}](\neg \mathtt{i.v_1} \vee \neg \mathtt{i.eating})$
Proof. *See appendix A.* ■

The following lemma characterizes, in some way, the relationship between violations $\mathtt{i.v_1}$ and $\mathtt{i.v_2}$: if a philosopher is in violation $\mathtt{i.v_1}$ and not in $\mathtt{i.v_2}$ (that is, he has both forks), then none of his neighbouring philosophers can be eating.

Lemma 6. $\vdash_{\mathbf{Phil}} \mathtt{i.v_1} \wedge \neg \mathtt{i.v_2} \rightarrow \neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1).\mathtt{eating}$

The next lemma says that, if a philosopher is not eating, then in the next state he is not eating or he is not in a violation.

Lemma 7. $\vdash_{\mathbf{Phil}} \neg \mathtt{i.eating} \rightarrow [\mathbf{U}](\neg \mathtt{i.v_1} \vee \neg \mathtt{i.eating})$
Proof. *See appendix A.* ■

The following lemma tell us that, if a philosopher is in a violation, then he cannot be eating.

Lemma 8. $\vdash_{\mathbf{Phil}} \mathtt{i.v_1} \rightarrow \neg \mathtt{i.eating}$
Proof. *See appendix A.* ■

It seems obvious that, if a philosopher is in a violation, then he is in the bathroom; the following lemma formalizes this intuition.

Lemma 9. $\vdash_{\mathbf{Phil}} \mathtt{i.v_1} \rightarrow \mathtt{i.bath}$
Proof. *See appendix A.* ■

If a philosopher is thinking, then he has neither the right fork nor the left fork.

Lemma 10. $\vdash_{\mathbf{Phil}} \mathtt{i.thk} \rightarrow \neg \mathtt{i.has_L} \wedge \neg \mathtt{i.has_R}$
Proof. *See appendix A.* ■

No two neighbours can be hungry at the same time.

Lemma 11. $\vdash_{\mathbf{Phil}} \neg \mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry}$
Proof. *See appendix A.* ■

Suppose that a philosopher $\mathtt{i}+1$ is in a violation $\mathtt{v_2}$, but he does not have the right fork in his hands, and, in addition, philosophers $\mathtt{i}$ and $\mathtt{i}-1$ are not in the bathroom, $\mathtt{i}$ is hungry and $\mathtt{i}-1$ is thinking, then there exists the possibility for $\mathtt{i}$ to eat in the future. This fact is expressed by the following lemma.

Lemma 12. $\vdash_{\mathbf{Phil}} \mathsf{AG}((\mathtt{i}+1).v_2 \wedge \neg(\mathtt{i}+1).has_R) \wedge \neg \mathtt{i}.bathroom \wedge \neg(\mathtt{i}-1).bathroom) \wedge \mathtt{i}.hungry \wedge (\mathtt{i}-1).thinking \rightarrow \mathsf{EF}\mathtt{i}.eating$
Proof. *See appendix A.* ■

The next lemma is a variation of the above lemma.

Lemma 13. $\vdash_{\mathbf{Phil}} \mathsf{AG}(((i+1).v_2 \wedge \neg(i+1).has_R) \wedge \neg i.bathroom \wedge \neg(i-1).bathroom) \wedge i.thinking \wedge (i-1).thinking \rightarrow \mathsf{EF}i.eating$
Proof. *See appendix A.* ■

Now, we have the same scenario as before, but, if $(\mathtt{i}-1)$ is eating, then i will have the possibility of eating.

Lemma 14. $\vdash_{\mathbf{Phil}} \mathsf{AG}(((\mathtt{i}+1).\mathtt{v}_2 \wedge \neg(\mathtt{i}+1).\mathtt{has}_\mathtt{R}) \wedge \neg \mathtt{i}.\mathtt{bathroom} \wedge \neg(\mathtt{i}-1).\mathtt{bathroom}) \wedge (\mathtt{i}.\mathtt{thinking} \vee \mathtt{i}.\mathtt{hungry}) \wedge (\mathtt{i}-1).\mathtt{eating} \rightarrow \mathsf{EF}\mathtt{i}.\mathtt{eating}$
Proof. *See appendix A.* ■

We have the following variation of the above lemma.

Lemma 15. $\vdash_{\mathbf{Phil}} \mathsf{AG}(((\mathtt{i}+1).\mathtt{v}_2 \wedge \neg(\mathtt{i}+1).\mathtt{has}_\mathtt{R}) \wedge \neg \mathtt{i}.\mathtt{bathroom} \wedge \neg(\mathtt{i}-1).\mathtt{bathroom}) \wedge \mathtt{i}.\mathtt{thk} \wedge (\mathtt{i}-1).\mathtt{hungry} \rightarrow \mathsf{EF}\mathtt{i}.\mathtt{eating}$
Proof. *See appendix A.* ■

We need one more lemma. The next one says if a philosopher cannot be in the bathroom, and a neighbour is hungry, then he will be thinking in the next state.

Lemma 16.

$$\mathtt{i}.\mathtt{thk} \wedge (\mathtt{i}-1).\mathtt{hungry} \wedge \mathsf{AN}\neg \mathtt{i}.\mathtt{bathroom} \rightarrow \mathsf{EN}\mathtt{i}.\mathtt{thk}$$

Proof. *See appendix A.* ■

At this point we are ready to prove the first important property: if philosopher i is always in violation $\mathtt{i}.\mathtt{v}_1$, but not in violation $\mathtt{i}.\mathtt{v}_2$, then none of his neighbours can eat.

Property 1. $\vdash_{Phil} \mathsf{AG}(\mathtt{i}.\mathtt{v}_1 \wedge \neg \mathtt{i}.\mathtt{v}_2) \rightarrow \mathsf{AG}(\neg \mathtt{i}.\mathtt{eating} \wedge \neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1.\mathtt{eating}))$
Proof.

1.	$\mathtt{i}.\mathtt{v}_1 \rightarrow \neg \mathtt{i}.\mathtt{eating}$	*lemma 8*
2.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_1) \rightarrow \mathsf{AG}(\neg \mathtt{i}.\mathtt{eating})$	*CTL, 1*
3.	$\mathtt{i}.\mathtt{v}_1 \wedge \neg \mathtt{i}.\mathtt{v}_2 \rightarrow$ $\neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1).\mathtt{eating}$	*lemma 6*
4.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_1 \wedge \neg \mathtt{i}.\mathtt{v}_2) \rightarrow$ $\mathsf{AG}(\neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1).\mathtt{eating})$	*CTL, 3*
5.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_1 \wedge \neg \mathtt{i}.\mathtt{v}_2) \rightarrow$ $\mathsf{AG}(\neg \mathtt{i}.\mathtt{eating} \wedge \neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1).\mathtt{eating})$	*CTL, 2, 4*

■

Now, we will prove that a violation $\mathtt{i.v_2}$ is less dangerous than a violation $\mathtt{i.v_1} \wedge \neg\mathtt{i.v_2}$, in the sense that the first type of violation allows neighbours to progress in some cases. Suppose that a philosopher goes to the bathroom with the left fork, then his right neighbour is free to take his left fork (the right fork of the philosopher who has gone to the bathroom); moreover, he will be lucky in the sense that his left neighbour does not compete anymore for that resource. In addition, if the right neighbour of the lucky philosopher will not go to the bathroom, then we can ensure that the lucky philosopher will have the possibility of eating in the future.

Property 2.

$$\vdash_{\mathbf{Phil}} \mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath}) \rightarrow \mathsf{EF}\mathtt{i.eating}$$

Proof.

1.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.thinking} \wedge (\mathtt{i}-1).\mathtt{thinking} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*Lemma 13*
2.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.hungry} \wedge (\mathtt{i}-1).\mathtt{thinking} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*Lemma 12*
3.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.thk} \wedge (\mathtt{i}-1).\mathtt{hungry} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*Lemma 15*
4.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.hungry} \wedge (\mathtt{i}-1).\mathtt{eating} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*PL, Lemma 14*
5.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.thk} \wedge (\mathtt{i}-1).\mathtt{eating} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*PL, Lemma 14*
6.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\wedge\mathtt{i.eating} \rightarrow \mathsf{EF}\mathtt{i.eating}$	*CTL*
7.	$\mathtt{i.eating} \oplus \mathtt{i.bath} \oplus \mathtt{i.hungry} \oplus \mathtt{i.thk}$	***Phil3***
8.	$(\mathtt{i}-1).\mathtt{eating} \oplus (\mathtt{i}-1).\mathtt{bath} \oplus (\mathtt{i}-1).\mathtt{hungry} \oplus (\mathtt{i}-1).\mathtt{thk}$	***Phil3***
9.	$\neg(\mathtt{i.hungry} \wedge (\mathtt{i}-1).\mathtt{hungry}) \wedge \neg(\mathtt{i.eating} \wedge (\mathtt{i}-1).\mathtt{eating})$	*lemma 3, lemma 11*
10.	$\mathsf{AG}(((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R}) \wedge \neg\mathtt{i.bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath})$ $\rightarrow \mathsf{EF}\mathtt{i.eating}$	*CTL, 1-9*

■

The proof of this property is sketched; the idea behind it is to analyze the possible states of the philosophers involved using lines *7*, *8* and *9*, and then by lines *1-6* we can prove that in each possible scenario the property is true, and the result follows.

Note that we cannot ensure that philosopher i will always eat, because this depends on fair scheduling. If we add some kind of fairness restriction to our specification, then we are able to prove it.

Obviously, we can prove the symmetric case.

Property 3.

$$\vdash_{Phil} \mathsf{AG}(((\mathtt{i}-1).\mathtt{v}_2 \wedge \neg(\mathtt{i}-1).\mathtt{has}_{\mathtt{L}}) \wedge \neg\mathtt{i}.\mathtt{bath} \wedge \neg(\mathtt{i}+1).\mathtt{bath}) \rightarrow \mathsf{EF}\mathtt{i}.\mathtt{eating}$$

Using both property 2 and property 3 we can obtain the following corollary.

Corollary 4.
$\vdash_{Phil} \mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge \neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath} \wedge \neg(\mathtt{i}+2).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow$
$\mathsf{EF}((\mathtt{i}-1).\mathtt{eating} \vee (\mathtt{i}+1).\mathtt{eating})$
Proof.

1.	$\mathtt{i}.\mathtt{v}_2 \rightarrow \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \oplus \neg\mathtt{i}.\mathtt{has}_{\mathtt{R}}$	***V3***
2.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}} \wedge \neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}+2).\mathtt{bath}) \rightarrow$ $\mathsf{EF}((\mathtt{i}+1).\mathtt{eating})$	*Prop.3*
3.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge \neg\mathtt{i}.\mathtt{has}_{\mathtt{R}} \wedge \neg(\mathtt{i}-1).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow$ $\mathsf{EF}((\mathtt{i}-1).\mathtt{eating})$	*Prop.2*
4.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge (\neg\mathtt{i}.\mathtt{has}_{\mathtt{R}} \oplus \neg\mathtt{i}.\mathtt{has}_{\mathtt{L}}) \wedge \neg(\mathtt{i}-1).\mathtt{bath} \wedge$ $\neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}+2).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow$ $\mathsf{EF}((\mathtt{i}+1).\mathtt{eating}) \vee \mathsf{EF}((\mathtt{i}-1).\mathtt{eating})$	*CTL, 2, 3*
5.	$\mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge \neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath}$ $\wedge\neg(\mathtt{i}+2).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow$ $\mathsf{EF}((\mathtt{i}+1).\mathtt{eating} \vee (\mathtt{i}-1).\mathtt{eating})$	*CTL, 1, 4*

■

Imposing fairness restrictions (i.e., if a philosopher has the possibility of eating an unbounded number of times, he will eat an unbounded number of times) we should be able to prove the following property:

$$\mathsf{AG}(\mathtt{i}.\mathtt{v}_2 \wedge \neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath} \wedge \neg(\mathtt{i}+2).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow \mathsf{AF}((\mathtt{i}-1).\mathtt{eating} \vee (\mathtt{i}+1).\mathtt{eating})$$

That is, in case of a $\mathtt{v}_2$ violation, one of the two neighbours will eventually eat.

Finally, if we consider the CTL property:

$$(\mathsf{AG}p \rightarrow \mathsf{EF}q) \rightarrow (\mathsf{A}(r\ \mathcal{U}\ \mathsf{AG}p) \rightarrow \mathsf{EF}q)$$

then we obtain the following corollary from theorem 4:

Corollary 5. $\vdash \mathsf{A}(\mathtt{i.v_1}\ \mathcal{U}\ \mathsf{AGi.v_2}) \wedge \mathsf{AG}(\neg(\mathtt{i}+1).\mathtt{bath} \wedge \neg(\mathtt{i}-1).\mathtt{bath} \wedge \neg(\mathtt{i}+2).\mathtt{bath} \wedge \neg(\mathtt{i}-2).\mathtt{bath}) \rightarrow \mathsf{EF}((\mathtt{i}-1).\mathtt{eating} \vee (\mathtt{i}+1).\mathtt{eating})$

This says that, if a violation $\mathtt{v_1}$ upgrades to a violation of type $\mathtt{v_2}$ (for example, if the philosopher puts down a fork), then at least one of the two neighbours will have the possibility of eating in the future.

4.2 The Muller C-element

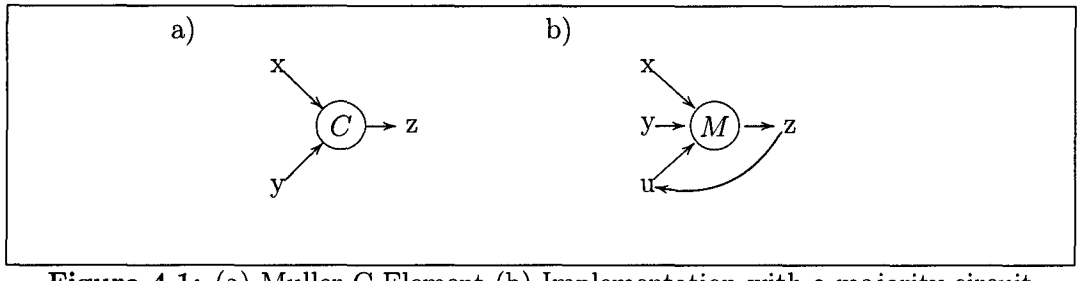


Figure 4.1: (a) Muller C-Element (b) Implementation with a majority circuit

The Muller C-Element [MC79] is a delay-insensitive circuit; the circuit has two boolean inputs and one boolean output (see figure 4.1 (a)). The output becomes true when the two inputs are true, and it becomes false when the two inputs are false. The idea is that the output remains in its state until the two inputs change their states. In [AG92, Aro92], the following (informal) specification of the C-element with inputs x and y and output z is given: "*(i) Input x (respectively y) changes only if $x \equiv z$ (respectively, $y \equiv z$), (ii)Output z becomes true only if $x \wedge y$ holds, and becomes false only if $\neg x \wedge \neg y$ holds; (iii) Starting from a state where $x \equiv y$, eventually a state is reached where z is set to the same value that both x and y have. Ideally, both x and y change simultaneously. Faults may delay changing either x or y.*"

Note that in this (informal) specification the word "ideally" is used to point out an event that should occur in a scenario without faults. We use deontic predicates (see the specification below) to formalize this informal, "ideal", requirement. The predicates of the specification are: $\{\mathtt{x},\mathtt{y},\mathtt{z}\}$, x and y represent the inputs, and z represents the output. The actions are: $\{\mathtt{cx},\mathtt{cy},\mathtt{cz}\}$, cx (respectively, cy) is the

action of changing input x (respectively, y). cz is the action of changing output z. We consider a violation constant v_1.

M1. $\neg\mathsf{Done}(\mathbf{U}) \rightarrow (x \leftrightarrow y) \wedge \neg v_1$
M2. $(x \rightarrow [cx]\neg x) \wedge (\neg x \rightarrow [cx]x)$
M3. $(y \rightarrow [cy]\neg y) \wedge (\neg y \rightarrow [cy]y)$
M4. $(z \rightarrow [cx]\neg z) \wedge (\neg z \rightarrow [cx]z)$
M5. $(x \rightarrow [\overline{cx}]x) \wedge (\neg x \rightarrow [\overline{cx}]\neg x)$
M6. $(y \rightarrow [\overline{cy}]y) \wedge (\neg y \rightarrow [\overline{cy}]\neg y)$
M7. $(z \rightarrow [\overline{cz}]z) \wedge (\neg z \rightarrow [\overline{cz}]\neg z)$
M8. $\langle cx\rangle\top \rightarrow (x \leftrightarrow z)$
M9. $\langle cy\rangle\top \rightarrow (y \leftrightarrow z)$
M10. $\langle cz\rangle\top \leftrightarrow \neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z)$
M11. $(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow \mathsf{O}(cx \sqcap cy)$
M12. $\mathsf{F}(cx \sqcap \overline{xy}) \vee \mathsf{F}(\overline{cx} \sqcap cy) \rightarrow [(cx \sqcap \overline{cy}) \sqcup (\overline{cx} \sqcap cy)]v$
M13. $(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow \neg v_1$
M14. $\langle \mathbf{U}\rangle\top$

Axiom **M1** says that at the beginning x and y have the same value and there is no violation. Axioms **M2-M7** define the behaviour of actions cx, cy and cz. Axioms **M8 – M11** formalize requirement (ii). Axiom **M11** expresses that x and y ought to change simultaneously. Axiom **M12** says that if we perform some forbidden action, then we go into a violation state. Axiom **M13** says that when x, y and z coincide there is no violation. The last axiom expresses that always some action can be executed.

First, we prove the following property from the specification:

$$\mathsf{AG}(\mathsf{O}(\alpha) \rightarrow \mathsf{ANDone}(\alpha)) \vdash_{\mathbf{M}} x \leftrightarrow y$$

(where **M** denotes the set containing all the axioms of the specification.) The formula at the left says that in every path only actions which are obliged are executed (i.e., the deontic constraints are satisfied), the right formula says that in this case $x \leftrightarrow y$. The proof is using the induction rule. First, note that $\vdash_{\mathbf{M}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow x \leftrightarrow y$ holds by axiom **M1**. We prove:

$$\mathsf{AG}(\mathsf{O}(\alpha) \rightarrow \mathsf{ANDone}(\alpha)) \vdash_{\mathbf{M}} x \leftrightarrow y \rightarrow [\mathbf{U}]x \leftrightarrow y$$

as follows:

1. $(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow \mathsf{O}(cx \sqcap cy)$ **M11**

2.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [cz]\bot$	PL, **M10**
3.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [cx \sqcap cy \sqcap \overline{cz}](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	DPL,**M7**, **M2**,**M3**
4.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [cz](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	PL, 2
5.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [cx \sqcap cy](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	PDL, **BA**,3,4
6.	$O(cx \sqcap cy) \rightarrow [\mathbf{U}]Done(cx \sqcap cy)$	Assump.
7.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [\mathbf{U}]Done(cx \sqcap cy)$	PL,6,1
8.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	DPL, 7, 5
9.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cx]\bot$	PL, **M8**
10.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cy]\bot$	PL, **M9**
11.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [\overline{cx} \sqcap \overline{cy}]x \leftrightarrow y$	DPL, **M5**,**M6**
12.	$(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cx \sqcup cy]\bot$	DPL,11,12
13.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [\mathbf{U}]x \leftrightarrow y$	DPL, 11, 12
14.	$(x \leftrightarrow y) \rightarrow [\mathbf{U}](x \leftrightarrow y)$	PL, 8, 13

In the proof we use the acronym DPL to indicate that the formula follows directly using the properties of DPL described in chapter 3. Using this property of the specification we can prove the following property about normative trajectories (i.e., trajectories where the obligations are fulfilled [FM91a]).

$$AG(O(\alpha) \rightarrow ANDone(\alpha)) \vdash_{\mathbf{M}} (x \leftrightarrow y \wedge y \leftrightarrow z) \vee [\mathbf{U}](x \leftrightarrow y \wedge y \leftrightarrow z)$$

i.e., in any instant requirement (iii) is true or it becomes true in the next instant. The proof is as follows.

1.	$x \leftrightarrow y$	Property above
2.	$\neg(x \leftrightarrow y) \wedge (y \leftrightarrow z) \rightarrow (x \leftrightarrow y) \wedge \neg(y \leftrightarrow z)$	PL,1
3.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cx]\bot$	DPL, **M8**
4.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cy]\bot$	DPL, **M9**
5.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cz \sqcap \overline{cy} \sqcap \overline{cx}](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	DPL, **M4**, **M5**,**M6**
6.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [cz \sqcap cy \sqcap cx](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	DPL, 1, 2
7.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [\overline{cz}]\bot$	PL, **M10**
8.	$(x \leftrightarrow y) \wedge \neg(y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	DPL, 7, 6
9.	$(x \leftrightarrow y) \wedge (y \leftrightarrow z) \vee [\mathbf{U}](x \leftrightarrow y) \wedge (y \leftrightarrow z)$	PL, 2, 8

In the general case we can prove that, if a violation occurs, we can reach a state free of errors. This is expressed by the following formula:

$$\vdash_{\mathbf{M}} v_1 \rightarrow AN(x \leftrightarrow y) \wedge (y \leftrightarrow z)$$

The proof is as follows:

1.	$v_1 \rightarrow \neg(x \leftrightarrow z) \vee \neg(y \leftrightarrow z)$	**M13**
2.	$\neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cz \sqcap \overline{cx} \sqcap \overline{cy}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PDL, **M4**, **M5**, **M6**

3.	$\neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cx \sqcup cy]\bot$	PDL, **M8**, **M9**
4.	$\neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cx \sqcup cy](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PDL, 4
5.	$\neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PDL, 2, 4
6.	$\neg(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow [cx \sqcap \overline{cy} \sqcap \overline{cz}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PDL, **M2**,**M5**,**M6**
7.	$\neg(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow [cy \sqcap cz]\bot$	DPL, **M8**, **M10**
8.	$\neg(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	DPL, 7, 6
9.	$(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cy \sqcap \overline{cx} \sqcap \overline{cz}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PDL, **M3**,**M5**,**M6**
10.	$(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z) \rightarrow [cx \sqcap cz]\bot$	DPL, **M9**, **M10**
11.	$(x \leftrightarrow z) \wedge (y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	DPL, 10, 9
12.	$\neg(x \leftrightarrow z) \vee \neg(y \leftrightarrow z) \rightarrow [\mathbf{U}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PL, 11, 8, 5
13.	$v_1 \rightarrow [\mathbf{U}](x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PL, 1, 12
14.	$v_1 \rightarrow AN(x \leftrightarrow z) \wedge (y \leftrightarrow z)$	PL, **TempAx1**, **M14**, 13

Using axiom **M13** straightforwardly, we can prove:

$$\vdash_M v_1 \rightarrow AN\neg v_1$$

This property says that when we go into a violation state, then we can recover from it (i.e., reach a state where there are no faults). In other words, the design is tolerant to faults of type v_1. It is interesting to note that the deontic predicate used in the specification (the obligation) allows us to distinguish an ideal scenario from a faulty one. In addition, the deontic predicates allow us to express naturally some parts of the informal specification, as is shown by axiom **M11**.

4.2.1 Implementing the c-element with a majority circuit

The C-element can be implemented with a majority circuit with three inputs [MC79] as shown in figure 4.1 (b). In this case we have that z is the output of the circuit and u is the extra input of the circuit. The predicate $maj(x, y, u)$ returns the value of the majority circuit (which we suppose works correctly). The definition is:

$$maj(x, y, u) \leftrightarrow (x \wedge y) \vee (x \wedge u) \vee (y \wedge u)$$

We consider a new violation constant v_2. We keep most of the axioms, we replace **M1** by:

$$\textbf{M1.} \quad \neg Done(\mathbf{U}) \rightarrow (x \leftrightarrow y) \wedge (y \leftrightarrow z) \wedge (u \leftrightarrow z) \wedge \neg v_1 \wedge \neg v_2$$

and axiom **M7** is replaced by:

$$\textbf{M7.} \quad \neg(z \leftrightarrow maj(x, y, u)) \leftrightarrow \langle cz \rangle \top$$

which says that z changes according to the value of maj(x, y, z). We add the following axioms

M14. $(u \rightarrow [cu]\neg u) \wedge (\neg u \rightarrow [cu]u)$
M15. $(u \rightarrow [\overline{cu}]u) \wedge (\neg u \rightarrow [\overline{cu}]\neg u)$
M16. $\neg(z \leftrightarrow maj(x, y, u)) \rightarrow O(cz \sqcap cu)$
M17. $F(\overline{cu \sqcap cz}) \rightarrow [\overline{cu \sqcap cz}]v_2$
M18. $u \leftrightarrow z \rightarrow \neg v_2$

Axioms **M14** and **M15** express the behaviour of cu. Axiom **M16** expresses that z and u ought to change simultaneously. This reflects the requirement that the feedback from the output of the circuit to u should work correctly. **M17** says that, if it is forbidden to not change u and z at the same time and we do not change u and z simultaneously, then we go into a violation v_2. **M18** expresses that when u has the same boolean value as z, there is no violation v_2.

In this implementation, we have to verify that the specification satisfies requirements (ii) and (iii). As stated in [AG92], the implementation tolerates delays in inputs x and y (i.e., items (ii) and (iii) hold indeed in the presence of a delay in changing x or y), but it does not tolerate delays in input u. First, let us prove that this specification satisfies item (ii) by proving:

$$AG\neg v_2 \vdash_M z \leftrightarrow u$$

i.e., if there is no violation v_2, the feedback from z to u works correctly. The proof uses induction. It is straightforward to see that $\vdash_M \neg Done(U) \rightarrow \neg(u \leftrightarrow z)$. Now, let us prove $\vdash_M u \leftrightarrow z \rightarrow [U]u \leftrightarrow z$

1.	$(u \leftrightarrow z) \wedge (maj(x,y,u) \leftrightarrow z) \rightarrow [cz]\top$	PL, **M7**
2.	$(u \leftrightarrow z) \wedge (maj(x,y,u) \leftrightarrow z) \rightarrow [cu]\top$	PL, **M16**
3.	$(u \leftrightarrow z) \wedge (maj(x,y,u) \leftrightarrow z) \rightarrow [cu \sqcup cz](u \leftrightarrow z)$	DPL, 1, 2
4.	$(u \leftrightarrow z) \wedge \neg(maj(x,y,u) \leftrightarrow z) \rightarrow O(cz \sqcap cu)$	PL, **M16**
5.	$O(cz \sqcap cu) \rightarrow F(\overline{cz \sqcap cu})$	Def. O()
6.	$(u \leftrightarrow z) \wedge \neg(maj(x,y,u) \leftrightarrow z) \rightarrow [U]\neg v_2$	DPL, assumption
7.	$F(\overline{cz \sqcap cu}) \rightarrow [\overline{cz \sqcap cu}]v_2$	**M17**
8.	$(u \leftrightarrow z) \wedge \neg(maj(x,y,u) \leftrightarrow z) \rightarrow [U]\bot$	DPL,4,5,6
9.	$(u \leftrightarrow z) \wedge \neg(maj(x,y,u) \leftrightarrow z) \rightarrow [U](z \leftrightarrow u)$	DPL, 8
10.	$(u \leftrightarrow z) \rightarrow [\overline{cu} \sqcap \overline{cz}](u \leftrightarrow z)$	DPL, **M7**, **M15**
11.	$(u \leftrightarrow z) \wedge (maj(x,y,u) \leftrightarrow z) \rightarrow [U](u \leftrightarrow z)$	DPL, 3, 10
12.	$(u \leftrightarrow z) \rightarrow [U](u \leftrightarrow z)$	PL, 9, 11

Using this property we can prove that we change z only when x and y have the

same boolean value and z has a different value, i.e.:

$$\mathsf{AG}\neg v_2 \vdash \langle cz\rangle\top \rightarrow \neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z)$$

The proof is as follows:

1.	$\langle cz\rangle\top \rightarrow \neg(z \leftrightarrow maj(x,y,u))$	**M7**
2.	$\neg(z \leftrightarrow maj(x,y,u)) \rightarrow \neg(u \leftrightarrow maj(x,y,z))$	PL, Property above
3.	$(u \leftrightarrow maj(x,y,u)) \rightarrow (maj(x,y,u) \leftrightarrow x) \wedge (maj(x,y,u) \leftrightarrow y)$	PL, Def.maj
4.	$\langle cz\rangle\top \rightarrow (maj(x,y,u) \leftrightarrow x) \wedge (maj(x,y,u) \leftrightarrow y) \wedge (maj(x,y,u) \leftrightarrow z)$	PL, 1, 2, 3
5.	$\langle cz\rangle\top \rightarrow \neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z)$	PL, 4, Def.maj

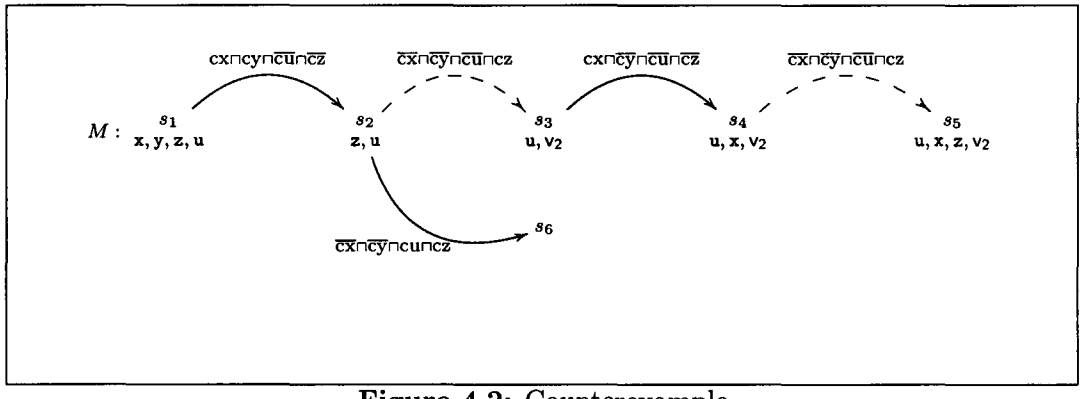


Figure 4.2: Counterexample

However, this property is not true when we have a violation of type v_2, as the model of figure 4.2 shows; in this figure each s_i (for $i \leq 5$) denotes a state of the model, and below each state we put the predicates that are true in that state. On the other hand, each transition is labelled with an event which indicates which action is executed and which is not. Dashed arrows denote transitions whose labels are not allowed to be executed. For example, in state s_4 the event $\overline{cx} \sqcap \overline{cy} \sqcap \overline{cu} \sqcap cz$ is forbidden. In this model, the initial state is s_1, and for the maximal path $\pi = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \xrightarrow{e_3} s_4 \xrightarrow{e_4} s_5$ (where the e_i's are the events of figure 4.2), we have that:

$$\pi, 4, M \not\models \langle cz\rangle\top \rightarrow \neg(x \leftrightarrow z) \wedge \neg(y \leftrightarrow z),$$

i.e., a change in cz is not due to a change in the two inputs x and y, which violates requirement (ii).

4.3 A Simple Train System

We consider a simple example of a train system. Train systems are those systems that control the movement of trains through a network of rail segments. Fault-tolerance is a key aspect of these systems: a fault in the system may cause a train collision and the loss of human life. These kinds of systems are the object of active research in the fault-tolerance community, see [Abr06, HG93, AB08].

Our system is made up of a collection of trains: $\mathtt{t}_1, \ldots, \mathtt{t}_n$ and a set of rail segments $\mathtt{r}_1, \ldots, \mathtt{r}_m$ (we assume $n < m$). Rail segments are connected to other rail segments, in each of these connections the rails have one signal controlling the access to them. The goal of the signal is to prevent trains from entering one segment when another train is already in it. The signals can be green (when the segment is free) or red (when another train is in the segment). We have the following predicates. For each $0 \leq i \leq n$ and $0 \leq j \leq m$, we have a predicate $\mathtt{t}_i.\mathtt{r}_j$ which is true when the train $\mathtt{t}_i$ is in segment $\mathtt{r}_j$; we also have a predicate $\mathtt{t}_i.\mathtt{s}$ (true when the train is stopped). For each $0 \leq j \leq m$ we have a proposition $\mathtt{r}_j.\mathtt{green}$ which is true when the signal of the segment $\mathtt{r}_j$ is green. We have a violation predicate $\mathtt{v}_j$ for every $0 \leq j \leq m$ which is true when we have two trains in the segment $\mathtt{r}_j$. (This is implemented by a sensor in the segment which detects the two trains.) Finally, we have propositions $\mathtt{r}_i R \mathtt{r}_j$ which indicate that $\mathtt{r}_i$ and $\mathtt{r}_j$ are connected.

We have the following actions: $\mathtt{t}_i.\mathtt{move(j)}$ (the train $\mathtt{t}_i$ moves to the segment $\mathtt{r}_j$), $\mathtt{t}_i.\mathtt{stop}$ (this action stops the train), $\mathtt{r}_i.\mathtt{ggreen}$ (the signal of rail $\mathtt{r}_i$ is set to green) and $\mathtt{r}_i.\mathtt{gred}$ (the signal of segment $\mathtt{r}_i$ is set to red).

Recall that in chapter 3 we introduced an extension of the logic with vocabularies that can have several versions of deontic predicates. In this example we consider three versions of deontic predicates for each train, and one version for each segment. We denote by $\mathtt{t}_i.\mathsf{P}^k()$, $\mathtt{t}_i.\mathsf{P}^k_w()$ and $\mathtt{t}_i.\mathsf{O}^k()$ the permissions and obligations corresponding to train $\mathtt{t}_i$. We use the same notation for the segments. Furthermore, we use some syntactic sugar and instead of writing $\mathtt{t}_i.\mathsf{O}^k(\mathtt{t}_i.\mathtt{move(j)})$ we write $\mathtt{t}_i.\mathsf{O}^k(\mathtt{move(j)})$, i.e., we do not repeat twice the trains and the segments when the second occurrence can be deduced from the context.

The axioms are as follows:

$$\textbf{T1.} \quad \bigoplus_{1 \leq j \leq m} \mathtt{t}_i.\mathtt{r}_j$$

for every $1 \leq i \leq n$. ($\bigoplus_{1 \leq j \leq m} \mathtt{t}_i.\mathtt{r}_j$ denotes the exclusive "or" of the predicates $\mathtt{t}_i.\mathtt{r}_j$.)

This axiom states that each train is in one and only one segment.

$$\textbf{T2.}\quad \neg\mathsf{Done}(\mathbf{U}) \rightarrow \bigwedge_{(1\leq i\leq n)} \bigwedge_{(1\leq j\leq n)\wedge(i\neq j)} \neg(\mathtt{t}_i.\mathtt{r}_k \wedge \mathtt{t}_j.\mathtt{r}_k)$$

This axiom says that, at the beginning of time, there are not two trains in the same segment.

$$\textbf{T3.}\quad \bigvee_{1\leq i\leq m} (\mathtt{r}_i R \mathtt{r}_j \wedge \mathtt{t}_k.\mathtt{r}_i) \rightarrow \langle \mathtt{t}_k.\mathtt{moveto(j)}\rangle \top$$

(for every $1 \leq \mathtt{j} \leq \mathtt{m}$ and $1 \leq \mathtt{k} \leq \mathtt{n}$.) Axioms **T3** say that train $\mathtt{t}_k$ can move to rail $\mathtt{r}_j$ if and only if the train k is in a segment that is connected to $\mathtt{r}_j$.

$$\textbf{T4.}\quad \neg \mathtt{r}_i R \mathtt{r}_i$$

(for every i.) Segments are not connected with themselves.

$$\textbf{T5.}\quad (\bigvee_{1\leq i,j\leq n\wedge i\neq j} \mathtt{t}_i.\mathtt{r}_k \wedge \mathtt{t}_j.\mathtt{r}_k) \leftrightarrow \mathtt{v}_k$$

(where $1 \leq \mathtt{k} \leq \mathtt{m}$.) There is a violation in segment $\mathtt{r}_k$ if and only if there are two trains in segment $\mathtt{r}_k$.

$$\textbf{T6.}\quad \mathtt{t}_i.\mathtt{r}_j \rightarrow \mathtt{r}_j.\mathsf{O}^1(\mathtt{gred})$$

(for every $1 \leq \mathtt{j} \leq \mathtt{m}$ and $1 \leq \mathtt{i} \leq \mathtt{n}$.) When there is a train in a segment, the signal for this segment must be red.

$$\textbf{T7.}\quad (\bigwedge_{1\leq i\leq n} \neg \mathtt{t}_i.\mathtt{r}_j) \rightarrow \mathtt{r}_j.\mathsf{O}^1(\mathtt{ggreen})$$

(for every $1 \leq \mathtt{j} \leq \mathtt{m}$.) If there is no train in the segment, then the signal for the segment must be green.

$$\textbf{T8.}\quad \neg \mathtt{t}_i.\mathtt{r}_k \wedge \neg \mathtt{r}_k.\mathtt{green} \rightarrow \mathtt{t}_i.\mathsf{F}^1(\mathtt{move(k)})$$

(for every $1 \leq \mathtt{i} \leq \mathtt{i}$, $1 \leq \mathtt{k} \leq \mathtt{m}$ and $1 \leq \mathtt{j} \leq 2$.) If the signal of a segment is red, then any train is forbidden to move into the segment.

$$\textbf{T9.}\quad \mathtt{t}_i.\mathtt{move(k)} \sqcap \mathtt{t}_j.\mathtt{move(k)} =_{act} \emptyset$$

(for every $1 \leq \mathtt{i}, \mathtt{j} \leq \mathtt{n}$ and $\mathtt{i} \neq \mathtt{j}$.) We suppose that there is some mechanism which prevents two trains from entering the same segment simultaneously, and this mechanism works correctly.

$$\textbf{T10.} \quad \mathtt{t_j}.\mathsf{F}^1(\mathtt{move(k)}) \rightarrow \mathtt{t_j}.\mathsf{O}^2(\overline{\mathtt{move(k)}} \sqcup \mathtt{stop})$$

(for every $1 \leq \mathtt{j} \leq \mathtt{n}$, $1 \leq \mathtt{k} \leq \mathtt{m}$ and $1 \leq \mathtt{i} \leq 3$.) This axiom formalizes a contrary-to-duty statement: if you are forbidden to move to segment $\mathtt{r_k}$, then you are obliged to not move the train to segment $\mathtt{r_k}$, or to stop the train. This statement also can be read as saying: if you are forbidden to move to segment $\mathtt{r_k}$, and you do it, you have to stop the train. This is similar to the *gentle killer* paradox[1].

Note that we are only taking into account the trains that for some reason ignore a red signal and enter into the segment. We must also specify what happens when another train is already in the segment, to avoid train collisions.

$$\textbf{T11.} \quad \mathtt{t_i}.\mathtt{r_j} \wedge \mathtt{r_j}.\mathtt{v} \rightarrow \mathtt{t_i}.\mathsf{O}^2(\mathtt{stop})$$

Another bad scenario is when a train is "locked" in a segment, i.e., when a train is in a segment where all the connected segments have their signal set to red. In this case the train is obliged to stop.

$$\textbf{T12.} \quad \mathtt{t_i}.\mathtt{r_k} \wedge \Big(\bigwedge_{1 \leq \mathtt{j} \leq \mathtt{m}} (\langle \mathtt{t_i}.\mathtt{move(j)} \rangle \top \rightarrow \mathsf{F}^1(\mathtt{t_i}.\mathtt{move(j)}))) \rightarrow \mathtt{t_i}.\mathsf{O}^3(\mathtt{stop})$$

The following axiom says that trains cannot move to a segment and at the same time this segment's signal changes to red; we assume some kind of mechanism which prevents a signal from changing at the same moment that a train is entering the segment.

$$\textbf{T13.} \quad \mathtt{t_i}.\mathtt{move(j)} \sqcap \mathtt{r_j}.\mathtt{gred} =_{act} \emptyset$$

(for every $1 \leq \mathtt{i} \leq \mathtt{n}$ and $1 \leq \mathtt{j} \leq \mathtt{m}$.) We define the behaviour of each action with the following axioms.

$$\textbf{T14.} \quad ([\mathtt{t_i}.\mathtt{take(j)}]\mathtt{t_i}.\mathtt{r_j}) \wedge (\neg \mathtt{t_i}.\mathtt{r_j} \rightarrow \overline{[\mathtt{i}.\mathtt{take(j)}]}\neg \mathtt{t_i}.\mathtt{stop})$$
$$\textbf{T15.} \quad ([\mathtt{r_j}.\mathtt{ggreen}]\mathtt{r_j}.\mathtt{g}) \wedge (\neg \mathtt{r_j}.\mathtt{g} \rightarrow \overline{[\mathtt{r_j}.\mathtt{ggreen}]}\neg \mathtt{r_j}.\mathtt{g})$$
$$\textbf{T16.} \quad ([\mathtt{r_j}.\mathtt{gred}]\neg \mathtt{r_j}.\mathtt{g}) \wedge (\neg \mathtt{r_j}.\mathtt{g} \rightarrow \overline{[\mathtt{r_j}.\mathtt{gred}]}\neg \mathtt{r_j}.\mathtt{g})$$

We can prove some properties of this specification. For example, we can prove that,

[1] It is forbidden to kill, but if you kill, you have to kill gently; you kill. From this one can obtain: you have to kill gently, which is contradictory with respect to the initial obligation.

if obligations of type 2 are fulfilled by trains, then there is no danger of having two trains in the same segment. Let Φ be the following set of formulae:

$$\Phi_1 = \{\mathsf{AG}(\mathtt{t_i}.\mathsf{O}^2(\mathtt{stop}) \rightarrow \mathsf{ANDone}(\mathtt{t_i}.\mathtt{stop})) \mid 1 \leq \mathtt{i} \leq \mathtt{n}\}.$$

These (finite) sets of formulae express that trains fulfil the obligations of type 2. We can consider a similar set of formulae for the segments:

$$\Phi_2 = \{\mathsf{AG}(\mathtt{r_i}.\mathsf{O}^1(\mathtt{gred}) \rightarrow \mathsf{ANDone}(\mathtt{r_i}.\mathtt{gred})) \mid 1 \leq \mathtt{j} \leq \mathtt{m}\}.$$

Using these sets of formulae, we can prove the following property:

$$\Phi_1, \Phi_2 \vdash_{\mathbf{Train}} \neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k})$$

Informally, when trains fulfil their obligations of stopping at a red signal and segments fulfil the obligation of setting their signal to red when there are trains in the segment, then we cannot have two trains in the same segment.

The proof uses the axiom of induction. Using axiom **T2** and propositional logic we obtain $\vdash_{\mathbf{Train}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k})$. Now, we prove:

$$\Phi_1, \Phi_2 \vdash_{\mathbf{Train}} \neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k}) \rightarrow [\mathbf{U}]\neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k}).$$

The proof is a follows:

1.	$\neg\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k} \rightarrow \mathtt{r_k}.\mathsf{O}^2(\mathtt{gred})$	**T6**
2.	$\mathtt{r_k}.\mathsf{O}^1(\mathtt{gred}) \rightarrow [\mathbf{U}]\mathsf{Done}(\mathtt{r_k}.\mathtt{gred})$	DPL, **TempAx1**, Assumption
3.	$\mathtt{r_k}.\mathtt{gred} \sqcap \mathtt{t_i}.\mathtt{move}(\mathtt{k}) =_{act} \emptyset$	**T13**
4.	$\neg\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k} \rightarrow [\mathtt{t_i}.\mathtt{move}(\mathtt{k})]\bot$	PDL, 1,2,3
5.	$\neg\mathtt{t_i}.\mathtt{r_k} \rightarrow \overline{[\mathtt{t_i}.\mathtt{move}(\mathtt{k})]}\neg\mathtt{t_i}.\mathtt{r_k}$	PL, **T14**
6.	$\neg\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k} \rightarrow [\mathbf{U}]\neg\mathtt{t_i}.\mathtt{r_k}$	PDL, 4, 5
7.	$\mathtt{t_i}.\mathtt{r_k} \wedge \neg\mathtt{t_j}.\mathtt{r_k} \rightarrow \mathtt{r_k}.\mathsf{O}^1(\mathtt{gred})$	**T6**
8.	$\mathtt{r_k}.\mathtt{gred} \sqcap \mathtt{t_j}.\mathtt{move}(\mathtt{k}) =_{act} \emptyset$	**T13**
9.	$\neg\mathtt{t_j}.\mathtt{r_k} \wedge \mathtt{t_i}.\mathtt{r_k} \rightarrow [\mathtt{t_j}.\mathtt{move}(\mathtt{k})]\bot$	PDL, 1,2,3
10.	$\neg\mathtt{t_j}.\mathtt{r_k} \rightarrow \overline{[\mathtt{t_j}.\mathtt{move}(\mathtt{k})]}\neg\mathtt{t_i}.\mathtt{r_k}$	PL, **T14**
11.	$\neg\mathtt{t_j}.\mathtt{r_k} \wedge \mathtt{t_i}.\mathtt{r_k} \rightarrow [\mathbf{U}]\neg\mathtt{t_j}.\mathtt{r_k}$	PDL, 4, 5
12.	$\neg\mathtt{t_j}.\mathtt{r_k} \wedge \neg\mathtt{t_i}.\mathtt{r_k} \rightarrow [\mathtt{t_i}.\mathtt{move}(\mathtt{k}) \sqcap \mathtt{t_j}.\mathtt{move}(\mathtt{k})]\bot$	PDL, **T9**
13.	$\neg\mathtt{t_j}.\mathtt{r_k} \wedge \neg\mathtt{t_i}.\mathtt{rk} \rightarrow \overline{[\mathtt{t_i}.\mathtt{r_k}.\mathtt{move}(\mathtt{k}) \sqcap \mathtt{t_j}.\mathtt{r_k}.\mathtt{move}(\mathtt{k})]}\neg\mathtt{t_j}.\mathtt{r_k} \wedge \neg\mathtt{t_i}.\mathtt{rk}$	PDL, **T14**
14.	$\neg\mathtt{t_j}.\mathtt{r_k} \wedge \neg\mathtt{t_i}.\mathtt{rk} \rightarrow [\mathbf{U}]\neg\mathtt{t_j}.\mathtt{r_k} \wedge \neg\mathtt{t_i}.\mathtt{rk}$	PDL, 6, 11, 12, 13
15.	$\neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k} \rightarrow [\mathbf{U}]\neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k}))$	PL, 14, 11, 6

Therefore, using the induction rule, we get $\vdash_{\mathbf{train}} \neg(\mathtt{t_i}.\mathtt{r_k} \wedge \mathtt{t_j}.\mathtt{r_k})$. Another property is that, when the obligations of type 3 are fulfilled, then when we have two trains in

a segment, both will stop. The property can be stated as follows:

$$\mathtt{t_i}.O^3(\mathtt{stop}) \rightarrow \mathsf{ANDone}(\mathtt{t_i.stop}) \vdash_{\mathbf{Train}} \mathtt{t_i.r_k} \wedge \mathtt{t_j.r_k} \rightarrow \mathsf{AN}\mathtt{t_i.stop} \wedge \mathtt{t_j.stop}.$$

The proof is straightforward from the axioms.

We can think of this property as a recovery property, since from a state where there is a (dangerous) violation we go into a state where we still have a violation but it is safe, since it is free of train collisions. As stated in [Aro92], fault-tolerance is not only about reaching a state free of error after a violation. But also, in some cases, it is acceptable to reach a safe state, where no further violations might arise. Of course, this example can be made more realistic, and we can state that after two trains are stopped in the same segment, then one of them can be removed, or an alternative exit can be made available. We keep the example as simple as possible to show how deontic predicates can be used to express requirements over specifications, which, when not fulfilled, yield a violation or bad behaviour.

On the other hand, if obligations of type 2 are not fulfilled, we can reach dangerous states. In figure 4.3, we have a model with two states s_0 and s_1; below each state, we have the predicates that are true at this state. We have two segments which are connected, and we have two trains, $\mathtt{t_1}$ is in segment $\mathtt{r_1}$ and $\mathtt{t_2}$ is in segment $\mathtt{r_2}$. Since segment $\mathtt{r_2}$ is occupied, $\mathtt{t_1}$ is forbidden to move to that segment, but if it moves, then it must stop. The train moves to that segment and it does not stop. We reach a state where the two trains are in the same segment, and $\mathtt{t_1}$ executes any action but $\mathtt{t_1.stop}$, which will produce a collision in the real world. This model also shows that the contrary to duty predicate expressed by axiom **T10** does not introduce any inconsistency in the specification.

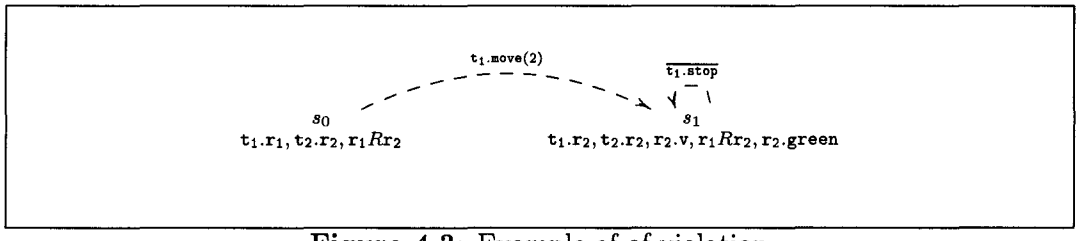


Figure 4.3: Example of of violation

4.4 Byzantine Generals

The Byzantine generals problem was stated originally in [LSP82]; the problem is the following. We have a general with $n-1$ lieutenants. The general and his lieutenants

can communicate with each other using messengers. The general may decide to attack an enemy city or to retreat; then, he sends the order to his lieutenants. Some of the lieutenants might be traitors. Traitors might deliver false messages or perhaps they avoid sending a message that they received. The loyal lieutenants must agree on attacking or retreating. This problem is a classic problem of fault-tolerance and distributed computing. Different solutions have been proposed, for example in [LSP82, DS83, ST87]. These solutions are simpler when an authenticated way of communication is used, i.e, traitors cannot lie. A solution proposed in the original paper is using signed messages in such a way that signatures cannot be forged (using some encryption protocol). The analogy with fault-tolerance is straightforward, the general is a sender process, the lieutenants are processes that have to agree with some decision taken by the sender. The traitors are faulty processes.

The specification that we provide below uses the ideas introduced in [DS83, ST87], where authenticated messages are used. The specification does not assume any form of authentication to prevent forged messages. Instead, deontic predicates are used to express that traitors are forbidden to lie. Of course, they might forge messages anyway. We consider this as a malicious behaviour which is a worse betrayal than to not obey orders. The important point here is that deontic operators allow us to abstract from the mechanisms that are used to prevent traitors from lying.

We have the following actions: $\mathtt{l_i.sendA(j)}$ (lieutenant $\mathtt{l_i}$ sends the message of attack to lieutenant $\mathtt{l_j}$), $\mathtt{l_i.fwd(k, A, j)}$ (lieutenant $\mathtt{l_i}$ forwards to $\mathtt{l_j}$ the message of attack that he received from $\mathtt{l_k}$), $\mathtt{l_i.betray}$ (lieutenant $\mathtt{l_i}$ becomes a traitor). We consider a clock that allows lieutenants to synchronize; the action $\mathtt{tt}$ increments the clock by one unit of time. The specification uses $\mathtt{m} + 1$ rounds of messages, which are coordinated by means of the clock. We have the following predicates. $\mathtt{l_i.A_j}$ (this predicate indicates that $\mathtt{l_i}$ has received a message from $\mathtt{l_j}$ saying that he must attack). We have a violation predicate $\mathtt{l_i.v}$ for each lieutenant (this predicate is true when $\mathtt{l_i}$ is a traitor, i.e., a $\mathtt{l_i}$ is in a violation state) and $\mathtt{l_i.d}$ (this predicate is true when $\mathtt{l_i}$ have decided to attack), $\mathtt{r_i}$ (this predicate is true when we are in round $\mathtt{i}$).

We assume that $\mathtt{l_0}$ is the general, the messages are delivered correctly and all the lieutenants can communicate directly with each other, in such a way that they can recognize who is sending a message. We have $\mathtt{n}$ lieutenants and the specification that is shown below uses a constant $\mathtt{m} < \mathtt{n}$ that indicates that the specification tolerates at most $\mathtt{m}$ traitors.

(For the deontic predicates we use the same notation conventions as in the train example.) The axioms are the following. Note that the following are axiom schemas, each formula denotes a finite collection of axioms.

$$1.\quad \neg \mathsf{Done}(\mathbf{U}) \rightarrow \Big(\bigwedge_{(1\leq i\leq n)} \bigwedge_{(1\leq j\leq n)\wedge(i\neq j)} \neg \mathtt{l}_i.\mathtt{A}_j \Big) \wedge \mathtt{r}_0 \wedge \Big(\bigwedge_{1\leq i\leq n} \neg \mathtt{l}_i.\mathtt{d} \Big)$$

At the beginning, the lieutenants have not received any message, we are in round zero and the lieutenants (with exception of the general) have not taken any decision (by default the decision is to retreat).

$$2.\quad \neg \mathsf{Done}(\mathbf{U}) \wedge \neg \mathtt{l}_0.\mathtt{v} \rightarrow (\mathtt{l}_0.\mathtt{d} \rightarrow \mathsf{AG}\mathtt{l}_0.\mathtt{d}) \wedge (\neg \mathtt{l}_0.\mathtt{d} \rightarrow \mathsf{AG}\neg \mathtt{l}_0.\mathtt{d})$$

If the general is loyal, then he keeps holding the same decision that he has taken at the beginning.

$$3.\quad \bigwedge_{1\leq i\leq n} \mathtt{l}_i.\mathsf{O}^1(\overline{\mathtt{betray}})$$

Lieutenants should not betray.

$$4.\quad \neg \mathsf{Done}(\mathbf{U}) \wedge \mathtt{l}_0.\mathtt{d} \rightarrow \mathtt{l}_0.\mathsf{O}^2\Big(\bigsqcup_{1\leq i\leq n} \mathtt{sendA(i)} \Big)$$

At the beginning, if the general decided to attack, then he ought to send a message with his decision to the other lieutenants.

$$5.\quad (\mathtt{r}_j \rightarrow [\mathtt{tt}]\mathtt{r}_{j+1}) \wedge \neg \mathtt{r}_m \rightarrow \langle \mathtt{tt} \rangle \top$$

These axioms specify the behaviour of the clock.

$$6.\quad \mathsf{Do}(\mathtt{tt})$$

We always increment the clock.

$$7.\quad \mathtt{r}_k \wedge \mathtt{l}_i.\mathtt{A}_{j_1} \wedge \cdots \wedge \mathtt{l}_i.\mathtt{A}_{j_k} \rightarrow \mathtt{l}_i.\mathtt{d}$$

(where $1 \leq \mathtt{k} \leq \mathtt{m}$, $1 \leq \mathtt{i} \leq \mathtt{n}$ and $1 \leq \mathtt{j}_1, \ldots, \mathtt{j}_k \leq 1$ are $\mathtt{k}$ different numbers.) These axioms indicate that, if in round $\mathtt{k}$ the lieutenant $\mathtt{l}_i$ has received $\mathtt{k}$ messages with the order to attack, then he decides to attack.

$$8.\quad \mathtt{r}_k \wedge \mathtt{l}_i.\mathtt{A}_{j_1} \wedge \cdots \wedge \mathtt{l}_i.\mathtt{A}_{j_k} \rightarrow \mathtt{l}_i.\mathsf{O}^2\Big(\bigsqcup_{1\leq j\leq n\wedge j\neq j_1\wedge\cdots\wedge j\neq j_k} \mathtt{sendA(j)} \sqcap \mathtt{fwd}(j_1, A, j) \sqcap \cdots \sqcap \mathtt{fwd}(j_k, A, j) \Big)$$

These axioms indicate that, if in round $\mathtt{r}_k$ the lieutenant $\mathtt{l}_i$ has received $\mathtt{k}$ messages

with the order to attack from k different persons, then he ought to notify all the rest of the lieutenants about the decision to attack, he also forwards all the messages received.

$$\textbf{9.}\quad \mathtt{l_i.v} \wedge \neg \mathtt{l_i.A_j} \rightarrow \mathsf{F}^3(\bigsqcup_{1\leq k\leq n} \overline{\mathtt{fwd(j,A,k)}})$$

If a lieutenant is a traitor, then he is forbidden to lie. This involves contrary-to-duty reasoning. Lieutenants might betray at any moment (which is forbidden), but, if they betray, then they must not lie.

$$\textbf{10.}\quad \mathtt{r_k} \wedge \neg \mathtt{l_i.v} \wedge \neg \mathtt{l_i.A_{j_1}} \wedge \cdots \wedge \neg \mathtt{l_i.A_{j_t}} \rightarrow [\bigsqcup_{1\leq k\leq n} \mathtt{sendA(k)}]\bot \wedge [\bigsqcup_{1\leq k,k'\leq n} \mathtt{fwd(k,A,k')}]\bot$$

(where $1 \leq \mathtt{k} \leq \mathtt{m}$, $1 \leq \mathtt{i} \leq \mathtt{n}$ and $\mathtt{t} > \mathtt{n} - \mathtt{k}$.) These axioms say that, when in round $\mathtt{r_k}$ a loyal lieutenant has not received at least k messages saying that he must attack, then he does not send nor forward any message.

$$\textbf{11.}\quad \mathtt{r_m} \rightarrow (\mathtt{l_i.d} \rightarrow \mathsf{AG}\mathtt{l_i.d}) \wedge (\neg \mathtt{l_i.d} \rightarrow \mathsf{AG}\neg \mathtt{l_i.d})$$

(for any $1 \leq \mathtt{i} \leq \mathtt{n}$.) These axioms expresses that the decision taken in round m is final.

$$\textbf{12.}\quad [\mathtt{l_i.sendA(j)}]\mathtt{l_j.A_i}$$
$$\textbf{13.}\quad [\mathtt{l_i.fwd(k,A,j)}]\mathtt{l_k.A_j}$$
$$\textbf{14.}\quad \neg \mathtt{l_i.A_j} \rightarrow [\mathtt{l_j.sendA(i)} \sqcup \bigsqcup_{1\leq t\leq n} \mathtt{l_t.fwd(i,A,j)}]\neg \mathtt{l_i.A_j}$$
$$\textbf{15.}\quad \neg \mathtt{l_i.v} \wedge \mathtt{l_i.d} \rightarrow [\mathtt{U}]\mathtt{l_i.d}$$
$$\textbf{16.}\quad \mathtt{l_i.A_j} \rightarrow [\mathtt{U}]\mathtt{l_i.A_j}$$

(for every $1 \leq \mathtt{k,i,j} \leq \mathtt{n}$.) These axioms specify the behaviour of the actions $\mathtt{l_i.sendA(j)}$ and $\mathtt{l_t.fwd(i,A,j)}$. Axiom **15** says that, if a loyal lieutenant has decided to attack he keeps his decision, axiom **16** says that lieutenants do not forget the messages received. Finally, we describe the behaviour of the action betray.

$$\textbf{17.}\quad [\mathtt{l_i.betray}]\mathtt{l_i.v}$$
$$\textbf{18.}\quad \neg \mathtt{l_i.v} \rightarrow \overline{[\mathtt{l_i.betray}]}\neg \mathtt{l_i.v}$$

The axioms of the specification depend on a number m which, intuitively, is the number of traitors for which the specification ensures that the loyal lieutenants will agree on a decision. We sketch the proof of the fact that, if we have less than m traitors, then the

loyal lieutenants reach an agreement. Consider, first, the following set of formulae:

$$\Phi_1 = \{\mathtt{l_i}.\mathsf{F}^3(\mathtt{fwd(k,A,j)}) \rightarrow \mathsf{ANDone}(\overline{\mathtt{l_i}.\mathtt{fwd(k,A,j)}}) \mid \text{ for any } 1 \leq \mathtt{i,j,k} \leq \mathtt{n}\}.$$

This set of formulae say that traitors do not lie. The following formulae say that there are at most $\mathtt{m}$ traitors:

$$\Phi_2 = \mathsf{AG}(\neg \mathtt{l}_{\mathtt{j}_1}.\mathtt{v} \wedge \cdots \wedge \neg \mathtt{l}_{\mathtt{j}_{n-m}}.\mathtt{v})$$

(for some different $0 \leq \mathtt{j}_1, \ldots, \mathtt{j}_{n-m} \leq \mathtt{n}$.) Another useful supposition is that loyal lieutenants fulfil their obligations, which is expressed by the following formulae:

$$\Phi_3 = \{\mathtt{l_i}.\mathsf{O}^2(\alpha) \rightarrow \mathsf{ANDone}(\alpha) \mid \text{ for every } 1 \leq \mathtt{i} \leq \mathtt{n}\}.$$

Then, if we suppose that there are at least $\mathtt{n} - \mathtt{m}$ lieutenants who are not traitors, traitors do not lie and that loyal lieutenants fulfil their obligations, we can prove that in round $\mathtt{m}+1$ the loyal lieutenants reach an agreement. This is expressed with the following formulae:

$$\Phi_1, \Phi_2, \Phi_3 \vdash_{\mathbf{Biz_m}} \mathtt{r}_{\mathtt{m}+1} \rightarrow (\mathtt{l}_{\mathtt{j}_1}.\mathtt{d} \leftrightarrow \cdots \leftrightarrow \mathtt{l}_{\mathtt{j}_{n-m}}.\mathtt{d}).$$

(We denote by $\mathbf{Biz_m}$ the specification given above.) This property follows trivially if we prove that any two loyal lieutenants reach an agreement. This is expressed by the following property:

Property 4.

$$\Phi_1, \mathsf{AG}(\neg \mathtt{l}_{\mathtt{j}_1}.\mathtt{v} \wedge \cdots \wedge \neg \mathtt{l}_{\mathtt{j}_{n-m}}.\mathtt{v}) \vdash_{\mathbf{Biz_m}} \mathtt{r}_{\mathtt{m}+1} \rightarrow (\mathtt{l}_{\mathtt{u}_1}.\mathtt{d} \leftrightarrow \mathtt{l}_{\mathtt{j}_{u_2}}.\mathtt{d}).$$

(for any $\mathtt{u}_1, \mathtt{u}_2 \in \{\mathtt{j}_1, \ldots, \mathtt{j}_{n-m}\}$.) We sketch the proof.

Sketch of Proof. *At the beginning we have $\neg \mathtt{l}_{\mathtt{u}_1}.\mathtt{d}$ and $\neg \mathtt{l}_{\mathtt{u}_2}.\mathtt{d}$. If, in any round $\mathtt{r_k}$ with $\mathtt{k} \leq \mathtt{m}$, we have $\mathtt{l}_{\mathtt{u}_1}.\mathtt{d}$ by axiom 8, and since we assume that loyal lieutenants fulfil their obligations, we know that the action $\mathtt{l}_{\mathtt{u}_1}.\mathtt{sendA}(\mathtt{l}_{\mathtt{u}_2})$ will be executed and also $\mathtt{l}_{\mathtt{u}_1}$ will forward all of the $\mathtt{k}$ messages that he received with an attack order. This implies that, in round $\mathtt{r}_{\mathtt{k}+1}$, lieutenant $\mathtt{l}_{\mathtt{u}_2}$ will have received $\mathtt{k}+1$ messages saying attack, and, therefore, by axiom 7, in round $\mathtt{r}_{\mathtt{k}+1}$ we have $\mathtt{l}_{\mathtt{u}_2}.\mathtt{d}$. The same reasoning can be applied to $\mathtt{l}_{\mathtt{u}_2}.\mathtt{d}$ in round $\mathtt{r_k}$ with $\mathtt{k} \leq \mathtt{m}$. If $\mathtt{l}_{\mathtt{k}_1}.\mathtt{d}$ is true in round $\mathtt{r}_{\mathtt{m}+1}$ and false in all the earlier rounds, then this lieutenant has received $\mathtt{m}+1$ messages saying "attack", but since traitors do not lie by assumption and also we assumed that we have at most $\mathtt{m}$ traitors, lieutenant $\mathtt{l}_{\mathtt{u}_1}$ have received an order to attack from some loyal lieutenant, which by axiom 8 sent the same orders to lieutenant $\mathtt{l}_{\mathtt{u}_2}$; this implies that in the next round after receiving the order from the loyal lieutenant, both have decided to attack by axiom 7.* ■

It is interesting to note that when traitors lie, the property shown above is not true. Suppose that we have three lieutenants: $\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2$ and $\mathtt{l}_1$ is a traitor. Consider the specification instanced with $\mathtt{m} = 1$ (only one traitor). The model in figure 4.4 shows a counterexample; we have three states: s_0, s_1, s_2, the initial state is s_0. Below each state the predicates that are true at that state are shown, the predicates which are false are not shown. At the beginning, we have that no lieutenant is a traitor, and that the general $\mathtt{l}_0$ has decided to retreat. Each transition is labelled with the actions that are executed in that transition. In the first transition, $\mathtt{l}_1$ becomes a traitor; the dashed arrow indicates that a forbidden action was executed. After that, $\mathtt{l}_1$ lies to $\mathtt{l}_2$ and he forwards a message that he did not receive; this is also a forbidden action. As a consequence, in round $\mathtt{r}_2$, lieutenants $\mathtt{l}_2$ and $\mathtt{l}_0$ do not agree since one has decided to attack and the other to retreat.

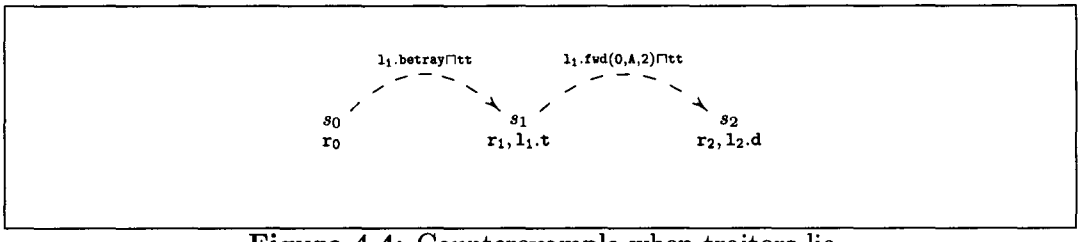


Figure 4.4: Counterexample when traitors lie

4.5 Coolers

Consider a microprocessor which is part of a critical system (perhaps in a space station, where it is not easy to replace it); we have two coolers to keep the temperature of the processor low, and also we have a sensor to measure the temperature. The processor could be in a normal state (that is, working correctly) or on stand by; the latter could occur when the processor is too hot, maybe because the coolers are not working. It is forbidden that the processor is on stand by because this can produce some incorrect behaviour in the system. This example was partially introduced in [CM07c].

We see that we have standard violations (when the coolers are not working), and also a contrary-to-duty scenario: the processor is forbidden to be on stand by, but if the temperature is too high (because of the bad behaviour of some cooler), then we should put the processor on stand by. The vocabulary of the example is given by the following set of actions and predicates with their intuitive meaning:

- $\mathtt{c}_1.\mathtt{start}$, turn on cooler 1.

- c_2.start, turn on cooler 2.
- c_1.stop, cooler 1 stops working.
- c_2.stop, cooler 2 stops working.
- p.sb, the processor goes into stand by.
- p.up, the processor wakes up.
- s.ghigh, the sensor detects high temperature.
- s.glow, the sensor detects low temperature.

and predicates:

- p.on, the processor is working.
- s.high, the sensor is detecting high temperature.
- c_1.on, the cooler 1 is working.
- c_2.on, the cooler 2 is working.

We have the following violation constants:

- v_1, a violation is produced because cooler 1 should be working and it is off.
- v_2, similar than v_1 but produced by cooler 2.
- v_3, a violation is produced because the processor is on stand by.

The following are some of the axioms of the specification:

Ax1. $\neg \mathrm{Done(U)} \rightarrow \neg v_1 \wedge \neg v_2 \wedge \neg v_3 \wedge \mathrm{p.on} \wedge \neg \mathrm{s.high} \wedge \neg c_1.\mathrm{on} \wedge \neg c_2.\mathrm{on}$

At the beginning (of time) there are no violations, the processor is working, the sensor is low, and the coolers are off.

Ax2. $\mathrm{F}^1(\mathrm{p.sb})$

It is forbidden that the processor goes into stand by.

$$\textbf{Ax3.}\quad \neg \mathtt{s.high} \rightarrow \mathsf{P}^i(\overline{\mathtt{p.sb}})$$

(for $i = 1, 2$.) If the sensor is low, then every action, different from putting the processor in stand by, is allowed.

$$\textbf{Ax4.}\quad \mathtt{s.high} \rightarrow \mathsf{O}^1(\mathtt{c}_1.\mathtt{on} \sqcap \mathtt{c}_2.\mathtt{on})$$

If the sensor is detecting high temperature, then the two coolers should be on.

$$\textbf{Ax5.}\quad \mathtt{s.high} \wedge \mathtt{v}_1 \wedge \mathtt{v}_2 \rightarrow \mathsf{O}^2(\mathtt{p.sb})$$

If the sensor is detecting a high temperature and both coolers are not working, then the processor ought to go into stand by.

$$\begin{aligned}\textbf{Ax6.}\quad & \neg \mathtt{v}_i \wedge \mathsf{F}^1(\overline{\mathtt{c}_1.\mathtt{on}}) \rightarrow [\overline{\mathtt{c}_1.\mathtt{on}}](\mathtt{v}_1 \wedge \neg \mathtt{v}_i)) \\ & \wedge(\mathtt{v}_i \wedge \mathsf{F}^1(\overline{\mathtt{c}_1.\mathtt{on}}) \rightarrow [\overline{\mathtt{c}_1.\mathtt{on}}](\mathtt{v}_1 \wedge \mathtt{v}_i))\end{aligned}$$

(For $2 \leq i \leq 3$.) These axioms express that a bad behaviour of cooler 1 yields a violations of type $\mathtt{v}_1$.

$$\begin{aligned}\textbf{Ax7.}\quad & (\neg \mathtt{v}_i \wedge \mathsf{F}^1(\overline{\mathtt{c}_2.\mathtt{on}}) \rightarrow [\overline{\mathtt{c}_2.\mathtt{on}}](\mathtt{v}_2 \wedge \neg \mathtt{v}_i)) \\ & \wedge(\mathtt{v}_i \wedge \mathsf{F}^1(\overline{\mathtt{c}_2.\mathtt{on}}) \rightarrow [\overline{\mathtt{c}_2.\mathtt{on}}](\mathtt{v}_2 \wedge \mathtt{v}_i))\end{aligned}$$

(For $i = 1, 3$.) These axioms are similar to axiom **Ax6** but for cooler 2.

$$\begin{aligned}\textbf{Ax8.}\quad & (\neg \mathtt{v}_i \wedge \mathsf{F}^2(\overline{\mathtt{ps.sb}}) \rightarrow [\overline{\mathtt{ps.sb}}](\mathtt{v}_3 \wedge \neg \mathtt{v}_i)) \\ & \wedge(\mathtt{v}_i \wedge \mathsf{F}^2(\overline{\mathtt{ps.sb}}) \rightarrow [\overline{\mathtt{ps.sb}}](\mathtt{v}_3 \wedge \mathtt{v}_i))\end{aligned}$$

(For $i = 1, 2$.) These axioms indicate that violation $\mathtt{v}_3$ arises when the processor is not put on stand by when it is necessary.

$$\begin{aligned}&\textbf{Ax9.}\quad (\mathtt{v}_1 \rightarrow [\overline{\mathtt{c}_1.\mathtt{on}}]\mathtt{v}_1) \wedge ([\mathtt{c}_1.\mathtt{on}]\neg \mathtt{v}_1) \\ &\textbf{Ax10.}\quad \mathtt{v}_2 \rightarrow [\overline{\mathtt{c}_2.\mathtt{on}}]\mathtt{v}_2 \wedge ([\mathtt{c}_2.\mathtt{on}]\neg \mathtt{v}_2) \\ &\textbf{Ax11.}\quad \mathtt{v}_3 \rightarrow [\overline{\mathtt{p.up}}]\mathtt{v}_3 \wedge ([\mathtt{p.up}]\neg \mathtt{v}_3)\end{aligned}$$

Ax9, **Ax10** and **Ax11** define the recovery actions for each violation; although this example is simple, in more complicated examples the designer has to take care that recovery actions should not cause violations. We add the restriction that if both coolers are on, then it is not possible to have a high temperature in the processor (the system is well designed in this sense).

$$\textbf{Ax12.}\quad \mathtt{c}_1.\mathtt{on} \wedge \mathtt{c}_2.\mathtt{on} \rightarrow \mathtt{s.low}$$

The rest of the axioms specify the behaviour of the actions.

Ax13. $[p.up]p.on$
Ax14. $\neg p.on \rightarrow \overline{[p.up]}\neg p.on$
Ax15. $[c_1.start]c_1.on$
Ax16. $\neg c_1.on \rightarrow \overline{[c_1.start]}\neg c_1.on$
Ax17. $[c_2.start]c_2.on$
Ax18. $\neg c_2.on \rightarrow \overline{[c_2.start]}\neg c_2.on$
Ax19. $[s.ghigh]s.high$
Ax20. $\neg s.high \rightarrow \overline{[s.ghigh]}\neg s.high$
Ax21. $[s.glow]\neg s.high$
Ax22. $s.high \rightarrow \overline{[s.glow]}s.ghigh$

An interesting point about this description is that having two different kinds of deontic predicates adds more structure in the violation lattice; the different violations that may occur in this specification are shown in figure 4.5. In this illustration we can

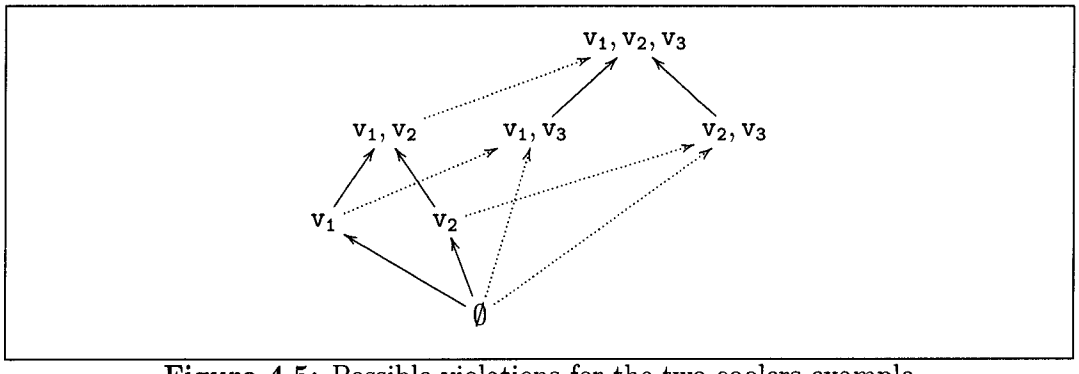


Figure 4.5: Possible violations for the two coolers example

see the different violations that can arise in the example; every node denotes a set of violations which may become true at a certain point in the execution of the system (these are the violation states described in chapter 3). At the beginning we have no violations (the empty set); after that we can go into a violation v_1 (when the cooler 1 is not working and the temperature is high), or to a violation v_2 when we have the same situation but for cooler 2; when both coolers are not working we get the two violations. Now, when we put the processor on stand by we have a violation v_3 which gives us the second dimension in the picture; this violation is needed in some situations to prevent the processor from burning out. These informal relationships between the violation states can be stated formally, we revisit this issue in chapter 7.

On the other hand, using the the *GGG* condition can make easier the proof of

some properties. For example, from this specification we can prove:

$$\vdash_{\mathbf{C} \cup G(\mathbf{C})} \mathsf{AG}\neg\mathtt{high} \rightarrow \mathsf{AG}\neg \mathsf{v}_1 \wedge \neg \mathsf{v}_2 \wedge \neg \mathsf{v}_3$$

i.e., when the temperature is low there are no violations ($\mathbf{C}$ denotes the set of axioms of the specification and $G(\mathbf{C})$ denotes the *GGG* condition instanced for this specification). The proof is straightforward using the *GGG* and **Ax3**.

4.6 Further Comments

In this chapter we have presented some examples of specifications using the logic described in chapter 3. In these examples, deontic predicates are used to express ideal behaviours, and violation predicates to point out that the component or system has not behaved as expected. In contrast to dynamic deontic logics [Mey88, Bro03], in the logic presented in chapter 3 the deontic predicates are not reduced to modalities and violation predicates.

Our view is that the relationship between violations and deontic predicates must be established in each individual specification. In Meyer's approach [Mey88], the definition of permission is $\mathsf{P}_M(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow} \langle\alpha\rangle\neg\mathsf{v}$, i.e., an allowed action is an action such that there is at least one way of executing it such that we reach a state without violations. In fault-tolerance, this is only true for recovery actions; as stated in [SC06], permitted actions may carry forward violations. In our setting, the properties of permitted actions are established in the specification, and they do not necessarily recover from a violation state. For instance, in the train example, by axiom **T8**, trains are obliged to stop; however, if the train stops (which is a permitted action since $\mathsf{O}(\alpha) \rightarrow \mathsf{P}(\alpha)$ is a theorem in our logic), the system is still in a violation state, since two trains are in the same segment. A recovery action would be to remove one train from the segment.

Another problem with Meyer's approach is the Ross's paradox (see chapter 2), with Meyer's obligation we have: $\mathsf{O}_M(\alpha) \rightarrow \mathsf{O}_M(\alpha \sqcup \beta)$. If we take again the train example, we have that $\mathsf{O}_M(\mathtt{t.stop}) \rightarrow \mathsf{O}_M(\mathtt{t.stop} \sqcup \overline{\mathtt{t.stop}})$, i.e., if a train ought to stop, then it ought to stop or not to stop. This property is undoubtedly undesirable. Ross's paradox is not valid in our logic (see chapter 3).

Broersen proposes another reduction of deontic predicates to modalities which avoids Ross's paradox. Broersen proposes, for example, $\mathsf{F}_B(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow} \langle\alpha\rangle\mathsf{v}$, i.e., an action is forbidden if and only if there is a way of executing it which yields a violation.

This relationship between prescriptions and postconditions is undesirable in some scenarios. For instance, in the example of the byzantine generals, if we use the Broersen prohibition to state that the lieutenants ought to not betray, i.e., $\mathsf{F}_B(\mathtt{l_i.betray})$, by the definition of this predicate we have: $\langle \mathtt{l_i.betray} \rangle \mathsf{I}_\mathsf{i}.\mathsf{v}$. Since, in the example, all the lieutenants are forbidden to betray, following this definition all lieutenants may betray. This means that given a lieutenant $\mathtt{l_i}$ and any state w where $\neg \mathtt{l_i.v}$ is true, we have a transition $w \xrightarrow{e} w'$ where $\mathtt{l_i.v}$ is true in w'. In these kinds of models the formula Φ_2 (that expresses that only m lieutenants will betray) will make the specification inconsistent (using the Broersen prohibition). Since in our approach there is no relationship between deontic predicates and transitions, the prohibition to betray does not imply the possibility of betraying.

As shown in the examples, the deontic predicates allow us to distinguish normal or ideal scenarios from those scenarios which are abnormal or faulty. This distinction is reflected in the semantic structures, where we have permitted or allowed transitions and forbidden or not allowed transitions. The use of stratified norms allows us to avoid contrary-to-duty paradoxes, where an obligation which arose after a violation may be in conflict with other obligations; this is illustrated in the train example and the byzantine generals example. In [MWD94], a version of dynamic deontic logic with stratified norms is sketched, but reducing the deontic predicates to modalities (following the original definitions of Meyer). As we argued above, this reduction of deontic predicates to modalities is sometimes not desirable when specifying computing systems, since the notion of prescription and description are mixed up.

Chapter 5

A Tableaux Calculus

Tableaux systems ([Smu68]) are practical proof systems which are representative of an important stream of research in automated theorem proving (see [Fit90]). The basic idea behind this kind of proof system is proof by refutation, i.e., to prove a formula φ, we start with $\neg\varphi$ and then we try to derive a contradiction. Usually, if the formula is not provable, we get a counterexample (a model which satisfies the negation of the formula). Several tableaux systems have been proposed for logics used in computer science, such as *dynamic logics*, *modal logics* and *temporal logics*. For example, in [Pra78], a tableaux system for *propositional dynamic logic* is described, which is also shown to be more efficient than other decision methods. In [Fit72] the method of labeled tableaux is introduced to deal with modal logic. Meanwhile, in [GM96], a tableaux system that incorporates some new characteristics is introduced to deal with *dynamic logic with converse*. These systems allow us to decide these logics, and to find counterexamples in the case of non-valid formulae.

In this chapter, we introduce a tableaux method for the deontic action logic presented in chapter 3. Tableaux methods can help to provide automated theorem provers for this logic, enabling automatation of the analysis of specifications (or the task of finding counterexamples). In [CM08], we have outlined a tableaux system for this deontic logic and in this thesis we fill in the technical details. In section 5.1 we introduce the tableaux system, and we prove that it is sound and complete with respect to the semantics proposed. One important point to stress again about the logic is that we consider a finite number of actions in vocabularies; this then implies that changes of vocabularies could affect the validity of formulae. As a consequence, the logic does not satisfy the satisfaction condition (see [GB92]). Intuitively, some actions in the vocabulary denote environment actions, which might interact with the system being specified. Then, adding more environment actions enlarges the number

of possible models of the specification, and therefore a formula which is valid for a given set of actions is perhaps not valid when more environmental actions are considered. We provide some formal machinery to tackle this problem; theorem 8 states that there is a bound on the number of actions to be considered when proving the validity of formulae. More precisely, given a formula, we can calculate the number of actions that we must consider in the vocabulary to prove its "global" validity (i.e., its validity in every vocabulary, or, in other words, in any environment in which the component being analyzed may be embedded). These kinds of results are important when we need to prove properties of incomplete specifications. For example, when we specify a component, but we do not know, *a priori*, which other components will interact with it, or perhaps we just have partial information about the environment which will interact with the component.

The chapter is organized as follows. In section 5.1 we introduce the tableaux system for the propositional part of the logic, and we prove its soundness and completeness. Then, we extend this system to deal with *branching time* temporal operators, and we prove the soundness and completeness of this new system. In section 5.2 we prove some meta theorems about how formula validity may be preserved when we change notation. Finally, we present some examples and further work.

5.1 Tableaux for DPL

In this section we present a tableaux system for the logic described above; we follow the approach introduced in [Fit72], where standard formulae are enriched with labels; intuitively, each label indicates a state in the semantics where the formula is true. Labeled systems are usual for many logics (see [Gab96]), in particular in [GM96] a tableaux deduction system for dynamic logic with converse is described. Here we adapt these techniques to our modal action logic, showing that deontic operators fit neatly into the system; the duality between the strong and weak permissions resembles in some sense the duality between modal necessity and modal possibility. We prove that this system is complete and sound, and we extend this system with rules for the temporal version of the logic showing that completeness and soundness is preserved.

A labeled, or prefixed, formula has the following structure: $\sigma : \varphi$, where σ is a label made up of a sequence of boolean (action) terms built from a given vocabulary and φ is a formula. We use the following notation for sequences: $\langle\rangle$ (*the empty sequence*), $x \cdot xs$ (*the sequence made up of an element* x *followed by a sequence* xs); we also use the same notation to denote the concatenation of two sequences; i.e., given two sequences xs and ys, by $xs \cdot ys$ we denote the sequence made up of the elements of xs followed by the elements of ys.

From here on we consider a fixed vocabulary: $V = \langle \Phi_0, \Delta_0 \rangle$. Recall that we use some axiomatization of boolean algebras, denoted by Φ_{BA}; note that there exist complete and decidable axiomatizations of boolean algebras.

Now, we can introduce the notion of a tableau.

Definition 21 (Tableaux). *A tableau is a (n-ary) rooted tree where nodes are labeled with prefixed formulae, and a branch is a path from the root to some leaf.* □

Intuitively, a branch is a tentative model for the initial formula (which we are trying to prove valid). Given a branch $\mathcal{B}$, we denote by $EQ(\mathcal{B})$ the equations appearing in $\mathcal{B}$.

In figures 5.1, 5.2 and 5.3 we introduce a classification of formulae which is useful for presenting the rules of the tableaux calculus. In figure 5.1, propositional formulae are classified following Smullyan's unifying notation [Smu68]. This notation is standard in tableaux systems. The figure also defines, for each formula of type A, two formulae (A_1 and A_2), and, for each formula of type B, two formulae (B_1 and B_2). (Note that in the literature α is used instead of A, and β instead of B; here we do not use greek letters to avoid confusion with action terms.) We also introduce the less standard classification for modal logics. (We follow the standard notation for modal logics (see [Fit72]).) Figure 5.2 shows the P and N prefixed formulae (called π and ν, respectively, in the literature); for each of them we define formulae $P(\gamma)$ and $N(\gamma)$, respectively. Here γ is some action term which is needed to define these formulae (see the rules below). Note that, for any formula P, $P(\gamma)$ denotes two formulae. Finally, in figure 5.3 we introduce a new classification for deontic formulae. Although the deontic operators are, in some sense, similar to the modal operators, we need to distinguish them; the deontic predicates state properties over transitions, whereas the modal operators state properties about states related to the actual state. Using the

A	A_1	A_2	B	B_1	B_2
$\sigma : \varphi \wedge \psi$	$\sigma : \varphi$	$\sigma : \psi$	$\sigma : \varphi \vee \psi$	$\sigma : \varphi$	$\sigma : \psi$
$\sigma : \neg(\varphi \vee \psi)$	$\sigma : \neg\varphi$	$\sigma : \neg\psi$	$\sigma : \neg(\varphi \wedge \psi)$	$\sigma : \neg\varphi$	$\sigma : \neg\psi$
$\sigma : \neg\neg\varphi$	$\sigma : \varphi$				

Figure 5.1: Classification for formulae A and B.

N	$N(\gamma)$	P	$P(\gamma)$
$\sigma : [\alpha]\varphi$	$\sigma \bullet \gamma : \varphi$	$\sigma : \langle\alpha\rangle\varphi$	$\sigma \bullet \gamma : \varphi,\ \sigma : \gamma \neq_{act} \emptyset$
$\sigma : \neg\langle\alpha\rangle\varphi$	$\sigma \bullet \gamma : \neg\varphi$	$\sigma : \neg[\alpha]\varphi$	$\sigma \bullet \gamma : \neg\varphi,\ \sigma : \gamma \neq_{act} \emptyset$

Figure 5.2: Classification for formulae P and N.

N_D	$N_D(\gamma)$	P_D	$P_D(\gamma)$
$\sigma : \mathsf{P}(\alpha)$	$\sigma : \mathsf{P}(\gamma)$	$\sigma : \neg\mathsf{P}(\alpha)$	$\sigma : \neg\mathsf{P}(\gamma)$, $\sigma : \gamma \neq_{act} \emptyset$
$\sigma : \neg\mathsf{P}_\mathsf{w}(\alpha)$	$\sigma : \neg\mathsf{P}_\mathsf{w}(\gamma)$	$\sigma : \mathsf{P}_\mathsf{w}(\alpha)$	$\sigma : \mathsf{P}_\mathsf{w}(\gamma)$, $\sigma : \gamma \neq_{act} \emptyset$

Figure 5.3: Classification for deontic formulae.

above classification of formulae, we can introduce the rules of the tableaux method. In the definition of these rules we use *front action* of a P, N, P_D or N_D formula to refer to the nearest action to the root in the syntactical tree corresponding to this formula. For example, for the formula $[\alpha]([\beta]\varphi \wedge \langle\gamma\rangle\psi)$, its front action is α.

In figure 5.4 the classic rules for standard formulae can be found. In figures 5.5 and 5.6, we exhibit the rules for N_D and N formulae, respectively. Rule N is standard for **K** modal logics (see [Fit72]); it does not introduce new labels in the branch, but it adds new formulae to labels already in the branch; intuitively, for all (the states denoted by) the labels reachable from the current state, the N formula must be true. On the other hand, rule N_D for deontic necessity requires that the corresponding action must be allowed for all the possible contexts in the actual state. Note that for modal necessity we only consider the labels already in the branch, while for deontic necessity we do not have this restriction. Rules P and P_D for modal and deontic

$$A : \frac{A}{\begin{array}{l} A_1 \\ A_2 \end{array}} \qquad\qquad B : \frac{B}{B_1 \mid B_2}$$

Figure 5.4: Classic rules for formulae of type A and B

$$N_D : \frac{N_D}{\begin{array}{l} N_D(\gamma_1) \\ \vdots \\ N_D(\gamma_n) \end{array}}$$

for all $\gamma_1, ..., \gamma_n \in At_{\Box\alpha}(\Delta_0/\Gamma)$, for α the front action of N_D and Γ the set of equations already in the branch

Figure 5.5: Rules for deontic necessity

$$N: \quad \frac{N}{\begin{array}{c} N(\gamma_1) \\ \vdots \\ N(\gamma_n) \end{array}}$$

where $\gamma_1, ..., \gamma_n \in At_{\sqsubset\alpha}(\Delta_0/\Gamma)$ for α the front action of N and where the labels of $N(\gamma_i)$ are already in the branch.

Figure 5.6: Rules for modal necessity

possibility, respectively, are shown in figure 5.7; given a P formula, this rule creates one branch for each possible execution of the front action in the formula; although the rule for deontic possibility is very similar, note that deontic possibility does not create new labels, because permissions only predicate over transitions. In the figure we use parentheses to distinguish between the P and P_D rules. Note that in these rules, an inequation saying that the action must not be impossible is added in each branch, allowing us to avoid adding labels that cannot exist in the semantics. In the figure 5.8 we can see the rule *Per*; this rule says that if an action which is atomic (in the sense that it cannot have different executions) is weakly allowed, then it is also strongly allowed. Note that we have not shown any rule for equality; this is because equality reasoning is implicit in our calculus (see below the definition of boolean closed). For simplicity of the presentation of the concepts, we rule out those formulae of the form: $[\alpha](\alpha =_{act} \beta)$, i.e., modal formulae where equality is after a modality. This does not affect the completeness of the method since formulae of this kind are equivalent to formulae where equations do not appear after modalities (see chapter 3). It is straightforward to extend the method described here to manage these kinds of formulae. We can extend this tableaux system to deal with vocabularies with several versions of deontic predicates (i.e., with stratified norms) considering an instance of each rule per each index in the vocabulary. Now we introduce the notions of *closed*, *boolean closed*, *deontic closed* and *open branch*. Keep in mind that a branch is a set of prefixed formulae.

Definition 22 (deontic closed). *Given a branch $\mathcal{B}$ and a boolean theory Γ, we say that $\mathcal{B}$ is deontic closed with respect to Γ if it satisfies at least one of the following conditions:*

- $\sigma : \mathsf{P}(\gamma) \in \mathcal{B}$ *and* $\sigma : \neg\mathsf{P}(\gamma) \in \mathcal{B}$, *for some* $\gamma \in At(\Delta_0/\Gamma)$, *and some label* σ.
- $\sigma : \mathsf{P}_\mathsf{w}(\gamma) \in \mathcal{B}$ *and* $\sigma : \neg\mathsf{P}_\mathsf{w}(\gamma) \in \mathcal{B}$, *for some* $\gamma \in At(\Delta_0/\Gamma)$, *and some label* σ.

$$P\ (P_D) : \frac{P\ (P_D)}{P(\gamma_1)\ (P_D(\gamma_1))\ |\ ...\ |\ P(\gamma_n)\ (P_D(\gamma_n))}$$

with $\{\gamma_1, ..., \gamma_n\} = At_{\sqsubset\alpha}(\Delta_0/\Gamma)$, with α the front action of $P(P_D)$ and Γ is the set of equations in the branch

Figure 5.7: Rules for possibility and permission

$$Per : \frac{\sigma : \mathsf{P_w}(\gamma)}{\sigma : \mathsf{P}(\gamma)}$$

with $\gamma \in At(\Delta_0/\Gamma)$, Γ being the set of equations in the branch

Figure 5.8: Rules for possibility and permission

- $\sigma : \neg\mathsf{P}(\gamma) \in \mathcal{B}$ *and* $\sigma : \mathsf{P_w}(\gamma) \in \mathcal{B}$, *for some* $\gamma \in At(\Delta_0/\Gamma)$, *and some label* σ.

□

Note that we have not included $\sigma : \mathsf{P}(\gamma)$ and $\sigma : \neg\mathsf{P_w}(\gamma)$ as being mutually contradictory; this is because they are not contradictory when $\Gamma \vdash_{BA} \gamma =_{act} \emptyset$. This fact yields the next definition.

Definition 23 (extended boolean theory). *Given a branch* $\mathcal{B}$, *the extended boolean theory (denoted by* $EQ^*(\mathcal{B})$*) of* $\mathcal{B}$ *is defined as follows:*

$$EQ^*(\mathcal{B}) = \{(\gamma =_{act} \emptyset)\ |\ \gamma \in At(\Delta_0) \wedge (\sigma : \mathsf{P}(\gamma), \sigma : \neg\mathsf{P_w}(\gamma) \in \mathcal{B})\} \cup EQ(\mathcal{B})$$

□

It is useful for us to introduce the notion of *boolean closed* branch; intuitively, these branches are inconsistent boolean theories. We denote by $EQ(\mathcal{B})$ the set of equations in the set $\mathcal{B}$.

Definition 24 (boolean closed branch). *A branch* $\mathcal{B}$ *is boolean closed iff* $EQ^*(\mathcal{B}) \vdash_{BA} \emptyset =_{act} \mathbf{U}$, *or* $EQ^*(\mathcal{B}) \vdash_{BA} \alpha =_{act} \beta$ *and* $\alpha \neq_{act} \beta \in \mathcal{B}$ □

Definition 25 (closed branch). *A branch is closed if either it has a propositional variable* $\sigma : p$ *and a negation of it* $\sigma : \neg p$, *or it is deontic closed or boolean closed.* □

An *open branch* is a branch which is not closed. Note that we have not described any particular way to apply the rules, which are, by their nature, non-deterministic. We show that any algorithm which satisfies some requirements provides a complete system for the logic. Let us introduce the notions of "complete" branch and "complete" tableau.

Definition 26. *We say that a branch $\mathcal{B}$ is complete if:*

- *For every labeled formula $\sigma : \varphi \in \mathcal{B}$,*
  - *If it is a B, P or P_D formula, then some of the formulae resulting from applying the corresponding rule to the formula belongs to $\mathcal{B}$.*
  - *If it is a N, A or N_D formula, then all the formulae resulting from applying the corresponding rule belong to $\mathcal{B}$.*

□

A tableau is complete when all its branches are complete. Note that, in the case of an N formula, we require that all the corresponding formulae $N(\gamma_i)$ must belong to the branch. When applying the rules, we must be careful to satisfy this requirement; a problem arises when we apply a rule N and following it we apply a P-rule, adding a label that was not there before, and then invalidating the condition shown above. There are several algorithms which can satisfy these requirements. In particular, variants of the algorithms given in [Smu68] and [Fit72] can be used; we do not describe any of these algorithms in this section (although in section 5.4 an algorithm for the temporal version of this logic is exhibited; this algorithm can be used for the present case too). However, in the next section we prove that any procedure which generates a complete tableaux gives us a complete tableaux system.

5.1.1 Soundness and Completeness

The soundness of the tableaux system is proved by a theorem which ensures that each rule is safe (with respect to satisfability). Towards this goal we introduce the following definitions.

Definition 27 (Mapping). *Given a set S of prefixed formulae (with F being the set of prefixes occurring in it) and a model $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ over a vocabulary $\langle \Delta_0, \Phi_0 \rangle$, an interpretation is a function $\iota : F \rightarrow W$, such that:*

- *For all σ and $\sigma \cdot \gamma$ in F, there exists $e \in \mathcal{I}(\gamma)$ such that $\iota(\sigma) \xrightarrow{e} \iota(\sigma \cdot \gamma)$.*

□

Definition 28 (SAT Branch). *A branch $\mathcal{B}$ is SAT iff there exists a model M and an interpretation ι such that, for every $\sigma : \varphi$, it is the case that $\iota(\sigma), M \models \varphi$.* □

We say that a tableaux $\mathcal{T}$ is SAT if there exists a branch in $\mathcal{T}$ which is SAT. Let us introduce a key theorem.

Theorem 11. *If $\mathcal{T}$ is a SAT tableau, then a tableau $\mathcal{T}'$ obtained by an application of a tableaux rule is also SAT.*

Proof. *Suppose that a branch $\mathcal{B}$ of $\mathcal{T}$ is SAT, and let $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ be the model and ι the interpretation for $\mathcal{B}$. We prove the theorem by induction; for the A and B rules the proof is standard.*

Rule P: Suppose $\sigma : \langle \alpha \rangle \varphi \in \mathcal{B}$, and $\iota(\sigma), M \models \langle \alpha \rangle \varphi$; obviously, $\mathcal{I}(\alpha) \neq_{act} \emptyset$, and also:

$$\exists e \in \mathcal{I}(\alpha) : \exists w' \in W : w \xrightarrow{e} w' \wedge w', M \models \varphi \tag{5.1}$$

If $\mathcal{B} \cup \{\sigma \cdot \gamma_i : \varphi\}$ is not SAT in M, this means for all $\gamma_i \in At(\Delta_0/\Gamma)$ (where Γ is the set of equations in the branch and Δ_0 is the set of atomic actions):

$$\begin{array}{l} \forall e \in \mathcal{I}(\gamma_i) : \forall w' \in W : w \not\xrightarrow{e} w' \vee w', M \not\models \varphi \\ \Rightarrow \\ \forall e \in \mathcal{I}(\gamma_1) \cup ... \cup \mathcal{I}(\gamma_n) : w \not\xrightarrow{e} w' \vee w', M \not\models \varphi \\ \Leftrightarrow \\ \forall e \in \mathcal{I}(\gamma_1 \sqcup ... \sqcup \gamma_n) : w \not\xrightarrow{e} w' \vee w', M \not\models \varphi \\ \Leftrightarrow \\ \forall e \in \mathcal{I}(\alpha) : w \not\xrightarrow{e} w' \vee w', M \not\models \varphi, \end{array}$$

contradicting 5.1.

Rule P_D: If $\sigma : \mathsf{P_w}(\alpha) \in \mathcal{B}$ (we must have $EQ(\mathcal{B}) \nvdash_{BA} \alpha \neq_{act} \emptyset$), then $\iota(\sigma), M \models \mathsf{P_w}(\alpha)$, and this means:

$$\exists e \in \mathcal{I}(\alpha) : \mathcal{P}(\iota(\sigma), e) \tag{5.2}$$

and therefore $e \in \mathcal{I}(\gamma_i)$ for some $\gamma_i \in At(\Delta_0/\Gamma)$, and:

$$\exists e \in \mathcal{I}(\gamma_i) : \mathcal{P}(\iota(\sigma), e) \tag{5.3}$$

and this means: $\iota(\sigma), M \models \mathsf{P_w}(\gamma_i)$.

The cases $\sigma : \neg[\alpha]\varphi \in \mathcal{B}$, $\sigma : \neg\mathsf{P}(\alpha) \in \mathcal{B}$ *are similar to the first and second case, respectively.*

Rule N: *If* $\sigma : [\alpha]\varphi \in \mathcal{B}$, *then* $\iota(\sigma), M \models [\alpha]\varphi$; *this means:*

$$\forall w' \in \mathcal{W}, e \in \mathcal{I}(\alpha) : \iota(\sigma) \xrightarrow{e} w' \Rightarrow w', M \models \varphi \tag{5.4}$$

Then, since for every $\gamma_i \in At(\alpha)$: $\mathcal{I}(\gamma_i) \subseteq \mathcal{I}(\alpha)$, *it is the case that:*

$$\forall w' \in \mathcal{W}, e \in \mathcal{I}(\gamma_i) : \iota(\sigma) \xrightarrow{e} w' \Rightarrow w', M \models \varphi \tag{5.5}$$

Now, if $\sigma . \gamma$ *is in* $\mathcal{B}$ *then we have* $\iota(\sigma) \xrightarrow{e} \iota(\sigma . \gamma)$ *(where* $e \in \mathcal{I}(\gamma_i)$*) by definition, and then* $\iota(\sigma . \gamma_i), M \models \varphi$.

Rule N_D *If* $\sigma : \mathsf{P}(\alpha) \in \mathcal{B}$, *then we have* $\iota(\sigma), M \models \mathsf{P}(\alpha)$. *This means:* $\forall e \in \mathcal{I}(\alpha) : \mathcal{P}(\iota(\sigma), e)$. *Now, if we add* $\sigma : \mathsf{P}(\gamma_i)$ *in* $\mathcal{B}$ *(for all* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/\Gamma)$*), then if* $\iota(\sigma), M \models \neg\mathsf{P}(\gamma_i)$ *for some* i, *it is not hard to see that* $\iota(\sigma), M \models \neg\mathsf{P}(\alpha)$, *which is a contradiction. The only possibility is that* $w, M \models \neg\mathsf{P}_{\mathsf{w}}(\gamma_i)$, *but this just implies that* $\mathcal{I}(\gamma_i) = \emptyset$, *and then* $\iota(\sigma), M \models \mathsf{P}(\gamma_i)$. ■

The soundness of the method follows by a standard argument.

Corollary 6. *If* φ *is tableaux provable (i.e., there exists a closed tableau for* $\neg\varphi$*) then* $\models \varphi$. ■

Towards the proof of completeness, we introduce the notion of *Hintikka sets*:

Definition 29 (Hintikka Sets). *Let* S *be a set of prefixed formulae and* $F(S)$ *the set of prefixes in* S. *We say that* S *is Hintikka iff:*

- *The labels in* S *are sequences made up of elements of* $At(\Delta_0/EQ(S))$.
- S *is not closed.*
- *If* $\sigma : \mathsf{P}(\alpha)$ *and* $\sigma : \neg\mathsf{P}_{\mathsf{w}}(\alpha) \in S$, *then* $EQ(S) \vdash_{BA} \alpha =_{act} \emptyset$
- *If* $A \in S$, *then* $A_1 \in S$ *and* $A_2 \in S$.
- *If* $B \in S$, *then either* $B_1 \in S$ *or* $B_2 \in S$, *or both.*
- *If* $N \in S$ *and* α *is the front action of* N, *then for all labels* $\sigma . \gamma_i \in F(S)$ *(where* σ *is the label of* N*) such that* $EQ(S) \vdash_{BA} \gamma_i \sqsubseteq \alpha$, *we have* $N(\gamma_i) \in S$.
- *If* $P \in S$, *then* $P(\gamma_i) \in S$, *for some* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/EQ(S))$. *(where* α *is the front action in* P*).*

- *If* $N_D \in S$, *then* $N_D(\gamma_i)$ *for all* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/EQ(S))$.
- *If* $\sigma : \mathsf{P_w}(\gamma_i) \in S$ *for some* $\gamma_i \in At(\Delta_0/EQ(S))$, *then* $\sigma : \mathsf{P}(\alpha) \in S$.

□

Now, we prove that any Hintikka set is SAT.

Theorem 12. *Any Hintikka set is SAT.*
Proof. *Given a Hintikka set* S, *we define the following model:*

- $\mathcal{W} = \{\sigma \mid \sigma : \varphi \in S$, *for some formula* $\varphi\}$
- $\mathcal{E} = At(\Delta_0/EQ(S))$.
- $\mathcal{R} = \{\sigma \xrightarrow{[\gamma]} \sigma \centerdot \gamma \mid \sigma, \sigma \centerdot \gamma \in \mathcal{W}\}$
- $p \in \mathcal{I}(w) \Leftrightarrow (\sigma : p) \in S$
- $\mathcal{I}(\alpha) = At_{\sqsubseteq\alpha}(\Delta_0/EQ(S))$
- $\mathcal{P} = \{(\sigma, [\gamma]) \mid (\sigma : \mathsf{P}(\gamma)) \in S \wedge \gamma \in \mathcal{E}\}$

We must prove that this is a model for S. *Proving that* $\mathcal{M}$ *satisfies requirements* ***I1***, ***I2*** *and* ***I3*** *of definition 9 is straightforward using the fact that the events are defined using a canonical boolean algebra (see chapter 3). We define the mapping* ι *as follows:* $\iota(\sigma) = \sigma$ *(the identity function). Let us prove that this structure is really a model of* S. *The proof is by induction.*
Base Case: Obviously, if $\sigma : p \in S$ *then* $\sigma, M \vDash p$. *We cannot have both* p *and* $\neg p$ *in* S, *and therefore the definition for propositional variables is correct. If* $\sigma : \alpha =_{act} \beta \in S$, *then* $EQ(S) \vdash_{BA} \alpha =_{act} \beta$ *and therefore* $At(\alpha) = At(\beta)$.
Ind.Case: We prove this by cases:
A rule: If $A \in S$ *then* A_1 *and* A_2 *are both in* S, *and the result follows by the definition of our model.*
B rule: Similar to the A rule case.
N rule: If $(\sigma : [\alpha]\varphi) \in S$, *and* $EQ(S) \vdash_{BA} \alpha =_{act} \emptyset$, *then* $At_{\sqsubseteq\alpha}(\Delta_0/EQ(S)) = \emptyset$ *and therefore* $\sigma, \mathcal{M} \vDash [\alpha]\varphi$. *Otherwise,* $\sigma \centerdot \gamma_i : \varphi$ *for all* $\gamma_i \in F$ *and* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/EQ(S))$, *and therefore* $\forall e \in \mathcal{I}(\alpha) : \sigma \xrightarrow{e} \sigma \centerdot \gamma_i \Rightarrow \sigma \centerdot \gamma_i, \mathcal{M} \vDash \varphi$.
P rule: If $\sigma : \langle\alpha\rangle\varphi \in S$, *then* $\sigma \centerdot \gamma_i : \varphi$ *for some* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/\Gamma)$, *thus* $\sigma \centerdot \gamma_i, \mathcal{M} \vDash \varphi$ *and then* $\sigma, \mathcal{M} \vDash \langle\alpha\rangle\varphi$.
P_D rule: If $\sigma : \mathsf{P_w}(\alpha) \in S$, *then* $\sigma : \mathsf{P_w}(\gamma_i)$ *for some* $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/\Gamma)$ *and then by definition of Hintikka sets* $\sigma : \mathsf{P}(\gamma_i)$, *which implies by definition of* $\mathcal{M}$: $\sigma, \mathcal{M} \vDash \mathsf{P_w}(\alpha)$.

N_D rule: If $\sigma : \mathsf{P}(\alpha) \in S$, then if $EQ(S) \vdash_{BA} \alpha =_{act} \emptyset$, then $At_{\sqsubseteq\alpha}(\Delta_0/\Gamma) = \emptyset$ and therefore $\sigma, \mathcal{M} \models \mathsf{P}(\alpha)$. Otherwise, $At_{\sqsubseteq\alpha}(\Delta_0/\Gamma) \neq \emptyset$ and then for all $\gamma_i \in At_{\sqsubseteq\alpha}(\Delta_0/\Gamma)$ occurs $\sigma : \mathsf{P}(\gamma_i) \in S$, and this implies $\sigma, \mathcal{M} \models \mathsf{P}(\alpha)$. ■

Using the above theorem we prove that every complete open branch is a Hintikka set.

Theorem 13. *If $\mathcal{B}$ is a complete open branch, then $\mathcal{B} \cup EQ^*(\mathcal{B})$ is a Hintikka set.*
Proof. *The proof is straightforward using the definition of open branch, complete branch and Hintikka set.* ■

This means that any algorithm which applies the rules in such a way that it produces a complete tableau gives us a complete proof method for the logic: given a formula φ, we put $\langle\rangle : \neg\varphi$ at the root and we apply the algorithm. If we get a (complete) open branch, then we have a model of $\neg\varphi$ showing in this way that φ is not valid. If all the branches are closed, then $\neg\varphi$ is not SAT, and therefore φ is valid.

5.2 Open Systems and Partial Specifications

There is a technical point which must be resolved before continuing with the description of the tableaux for the temporal part of the logic. Namely, given a formula φ, how many primitive actions must we consider in the vocabulary? A naive answer is: we just need to consider those primitive actions which appear in φ. However, a simple counter-example of why this does not work is the following. Consider the formula: $\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi$. Obviously, this formula is not valid, but if we build the tableau for it just considering a vocabulary with α as the only action, the final tree has no open branches. This only shows that this formula is valid for a vocabulary with just one primitive action. This problem arises from the fact that we are using a finite alphabet of actions and it is not possible, *a priori*, to know the complete set of actions to be considered. (Note that, usually, in dynamic logics or modal action logics an infinite number of actions is considered.) This occurs mainly for two reasons. First, we consider "Open Systems" in the sense that the component actions interact with environment actions; the point here is that the set of environment actions is not fixed as it may change over time or by context. Secondly, we assume that we are working with partial specifications, i.e., we might know just a part of the entire specification; this could happen since the system is being developed by different teams (as usual in software engineering), or perhaps since the system is evolving constantly.

Summarizing, our specification only gives us a partial picture of a system. Because of this, system properties are hard to verify (using tableaux or other formal systems).

After all, maybe we are not taking into account some actions important for the property to be proven.

The following theorems give us some machinery to attack this difficulty. Corollary 8 says that we can verify a property (using tableaux) restricting our attention just to some part of the system; if for this part of the system, this formula is valid, then it will be valid for any context. Interestingly, the number of actions that we need to consider depends on the formula to be verified. Furthermore, only the number of extra actions is important and not the properties of these extra actions. Some auxiliary notions are needed and we introduce the concepts of *normal form*, *disjunctive normal form*, and *existential degree*, and then we prove the theorems.

The *degree* of a formula φ (denoted by $\mathsf{d}(\varphi)$) is the length of the longest string of nested modalities (taking permission as being of degree 0). For any formula φ we denote by $Pr(\varphi)$ the set of atomic actions appearing in φ. Given a vocabulary $\langle \Delta_0, \Phi_0 \rangle$, we adapt the definition of *normal form of degree n* given in [Fin75] to our logic. We denote by F_i the set of formulae of normal form of degree i, defined as follows:

- F_0 is the set of formulae of the form $*\varphi_1 \wedge ... \wedge *\varphi_h$, where for each i: $\varphi_i \in \Phi_0$ or φ_i is a deontic predicate or φ_i is an equation, and $*$ is $\neg$ or blank.
- F_{n+1} is the set of formulae of the form: $\theta \wedge *\langle\alpha_1\rangle\varphi_1 \wedge .... \wedge *\langle\alpha_k\rangle\varphi_k$, where $\theta \in F_0$, $\varphi_i \in F_n$ for all $1 \leq i \leq k$, $*$ is $\neg$ or blank. (θ may not appear in the formula, in which case we only consider everything but not θ.)

The set of normal form formulae is $F = \bigcup_{i=1}^{\infty} F_i$. If a formula is in normal form, we say that it is a NF formula.

Theorem 14. *Any formulae of degree $\leq n$ is equivalent to $\bot$ or a disjunction of normal forms of degree n.*
Proof *See the proof given in [Fin75] and use the property* $\vdash \langle\alpha\rangle(\varphi \vee \psi) \leftrightarrow \langle\alpha\rangle\varphi \vee \langle\alpha\rangle\psi$.
■

Note that, in general, a NF formula can be expressed using the following schema:

$$\theta \wedge \Delta \wedge \bigwedge_{i=1}^{n} \langle\alpha_i\rangle\varphi \wedge \bigwedge_{j=1}^{m} \neg\langle\beta_j\rangle\beta_j$$

where θ is a conjunction of propositional variables or negations of them, and Δ is a conjunction of deontic predicates or negations of them.

If a formula is a disjunction of normal forms, we say that this formula is in *disjunctive normal form* (or DNF for short). We call $\mathsf{P_w}(\alpha)$, $\neg\mathsf{P}(\alpha)$ and $\langle\alpha\rangle\varphi$ *existential formulae*, i.e., *existential formulae* are those whose semantics is given by an existential quantifier. (Note that $\neg[\alpha]\varphi$ is equivalent to $\langle\alpha\rangle\neg\varphi$.)

Given a NF formula φ, with $\mathsf{d}(\varphi) = n$, we can define a set of formulae $\mathsf{SF}(\varphi, k)$, for every $k \leq n$, called the *subformulae at level k*. For $k = 0$ we define:

- If $\varphi = \bigwedge_{i=1}^{n} *p_i$, i.e., it is a conjunction of propositions or negations of them, then for this case the definition is:
$$\mathsf{SF}(\bigwedge_{i=1}^{n} *p_i, 0) = \bigcup_{i=1}^{n}\{*p_i\}$$
- If $\varphi = \bigwedge_{j=1}^{m} *\mathsf{P}(\alpha_j) \wedge \bigwedge_{k=1}^{t} *\mathsf{P_w}(\beta_k)$, i.e., the formula is a conjunction of deontic formulae or negations of them, then we define:
$$\mathsf{SF}(\bigwedge_{j=1}^{m} *\mathsf{P}(\alpha_j) \wedge \bigwedge_{k=1}^{t} *\mathsf{P_w}(\beta_k), 0) = \bigcup_{j=1}^{m}\{*\mathsf{P}(\alpha_j)\} \cup \bigcup_{k=1}^{t}\{*\mathsf{P_w}(\beta_k)\}$$
- In the case of a conjunction of propositional formulae and deontic formulae we can use the two definitions above, that is:
$$\mathsf{SF}(\theta \wedge \Delta, 0) = \mathsf{SF}(\theta, 0) \cup \mathsf{SF}(\Delta, 0)$$
- In the general case, we define:
$$\mathsf{SF}(\theta \wedge \Delta \wedge (\bigwedge_{i=1}^{n}\langle\alpha_i\rangle\varphi_i) \wedge (\bigwedge_{j=q}^{m}\neg\langle\beta_j\rangle\psi_j), 0) =$$
$$\mathsf{SF}(\theta) \cup \mathsf{SF}(\Delta) \cup \bigcup_{i=1}^{n}\{\langle\alpha_i\rangle\varphi_i\} \cup \bigcup_{j=1}^{m}\{\neg\langle\beta_j\rangle\psi_j\}$$

For the case of $k > 0$, we define:

$$\mathsf{SF}(\theta \wedge \Delta \wedge (\bigwedge_{i=1}^{n}\langle\alpha_i\rangle\varphi_i) \wedge (\bigwedge_{j=q}^{m}\neg\langle\beta_j\rangle\psi_j), k+1) = \bigcup_{i=1}^{n}\mathsf{SF}(\varphi_i, k) \cup \bigcup_{j=1}^{m}\neg\mathsf{SF}(\psi_j, k)$$

Where given a set S of formulae, we denote by $\neg S$, the set containing the negations of the formulae in S. (We also suppose that several negations over a formula are

simplified, i.e., instead of having $\neg\neg p$ we have p.) In some sense, the set SF indicates which set of subformulae must be true at a given level. We use $\#_\exists S$ to denote the number of existential formulae in the set S. Using this definition, we can define the *existential degree* of a NF formula φ, denoted by $\mathsf{D}_\exists$, which is defined as follows:

$$\mathsf{D}_\exists(\varphi) = \mathsf{max}_{0\leq i\leq n}\{\#_\exists \mathsf{SF}(\varphi, i)\}$$

We can extend this definition to CNF formulae, as follows:

$$\mathsf{D}_\exists(\varphi_1 \vee ... \vee \varphi_k) = max\{\mathsf{D}_\exists(\varphi_1), .., \mathsf{D}_\exists(\varphi_k)\}$$

The idea is to use the sets SF to define smaller models of φ. First, we need to define the notion of n-reachable. Given a model M and a state w, we say that a state v is n-reachable (or reachable in n-steps) from w, if there exists a path $w \xrightarrow{e_1} w_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} v$ in M. Note that our logic has the *unraveling property* [BRV01], i.e., if a formula φ is satisfiable in a model M and state w, we can build a model M' unraveling M such that w and M' satisfies φ; this new model is a tree, i.e., it does not have cycles. For the following results we restrict our attention to tree models; The unraveling property guarantees that these theorems extend to any other model.

If $\mathsf{d}(\varphi) = n$, then we can define a mapping L from the states reachable in M in n or less steps, to the subformulae of φ, as follows:

$$L(v) = \{\psi \in \mathsf{SF}(\varphi, k) \mid v \text{ is } k\text{-reachable from } w \text{ and } v, M \models \psi\}$$

For the following definitions, consider a model $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}\rangle$ over a vocabulary $V = \langle \Delta_0, \Phi_0\rangle$ and a NF formula φ (of degree n) such that $w, M \models \varphi$. The labeling L helps us to define a new model $M^\varphi_w = \langle \mathcal{W}^\varphi_w, \mathcal{R}^\varphi_w, \mathcal{E}^\varphi_w, \mathcal{I}^\varphi_w, \mathcal{P}^\varphi_w\rangle$ as follows:

- $\mathcal{E}^\varphi_w = \mathcal{E}$.
- We define $\mathcal{R}^\varphi_w$ in n steps:
  - At step 0, choose for each $\langle\alpha_i\rangle\varphi_i \in L(w)$ an event e_i such that $e_i \in \mathcal{I}(\alpha_i)$ and there exists a state v_i with $w \xrightarrow{e_i} v_i$, and $v_i, M \models \varphi_i$, and define $\mathcal{R}^0 = \bigcup_{e_i}\{w \xrightarrow{e_i} v_i\}$.
  - At step $k+1$, let $v_1, ..., v_m$ be the states k-reachable from w at step 0. For each of these states proceed as was done for state w, and define a relation $R^k_{v_i}$, and then $\mathcal{R}^k = \bigcup_{i\leq m} R^k_{v_i}$

Finally, $R^\varphi_w = \bigcup_{k\leq n} R^k$.

- $\mathcal{P}^{\varphi}_{w} = \mathcal{P}$.
- $\mathcal{W}^{\varphi}_{w} = \{v \in W \mid v \in \mathsf{Dom}(\mathcal{R}^{\varphi}_{w}) \cup \mathsf{Ran}(\mathcal{R}^{\varphi}_{w})\}$.
- $I^{\varphi}_{w}(a_i) = \mathcal{I}(a_i)$, for every e_i. $\mathcal{I}^{\varphi}_{w}(p_i) = \mathcal{I}^{p_i}$, for every $p_i \in \Phi_0$.

Note that this model has an out-degree (the number of transitions coming out of any state) less than or equal to $\mathsf{D}_{\exists}(\varphi)$, since in each state of the new model we only have one transition per existential subformula at the corresponding level. The following theorem says that the new model preserves property φ.

Theorem 15. *Given a NF formula φ and a model M, if $w, M \models \varphi$, then $v, M^{\varphi}_{w} \models L(v)$, for every $v \in \mathcal{W}^{\varphi}_{w}$.*
Proof *Suppose that $d(\varphi) = n$; we prove the result by induction. If v is reachable in n steps from w, then, by definition, $L(v)$ only contains propositional variables and deontic predicates, and therefore, by definition of M^{φ}_{w}, we have that $v, M^{\varphi}_{w} \models L(v)$.*

If v is reachable in k steps (with $k < n$) from w, then for each $\langle\alpha_i\rangle\varphi_i$ in $L(v)$ we have an $e_i \in \mathcal{I}^{\varphi}_{w}(\alpha_i)$ such that $v \xrightarrow{e_i} v'$; by induction we know that $v', M^{\varphi}_{w} \models \varphi_i$, and therefore $v, M^{\varphi}_{w} \models \langle\varphi_i\rangle\varphi_i$.

Now, suppose that $\neg\langle\beta_j\rangle\psi_j \in L(v)$; we know that ψ_j is a NF formula, and therefore $\psi_j = \psi'_1 \wedge ... \wedge \psi'_h$, and by definition of SF, we have that $\neg\psi'_1, ..., \neg\psi'_h \in \mathsf{SF}(\varphi, k+1)$. Then, if for some ψ'_l and state v', we have $v', M^{\varphi}_{w} \models \neg\psi'_l$ (where $v \xrightarrow{e_j} v'$ in $\mathcal{R}^{\varphi}_{w}$ and $e_j \in \mathcal{I}(\beta_j)$), i.e., we have that $\neg\psi'_l \in L(v')$, then by induction we have $v', M^{\varphi}_{w} \models \neg\psi'_l$, which implies that $v, M^{\varphi}_{w} \models \neg\langle\beta_j\rangle\psi_j$. This concludes the proof. ■

Note that $\bigwedge L(w) = \varphi$, and therefore we obtain the following corollary.

Corollary 7. *If $w, M \models \varphi$, then $w, M^{\varphi}_{w} \models \varphi$.*

Summarizing, given a model of a NF formula φ, we can build a new model with branching being at most $\mathsf{D}_{\exists}(\varphi)$. We are close to our original goal; using the model M^{φ}_{w}, we define a model over a restricted vocabulary. First, given a model M over a vocabulary $V = \langle\Delta_0, \Phi_0\rangle$, we denote by $EQ(M)$ the set of equations true in M, and if we have a subset $S \subseteq \Delta_0$, we denote by $EQ^{S}(M)$ the set of equations built from primitive actions in S which are true in M, i.e.,

$$EQ^{S}(M) = \{\alpha =_{act} \beta \mid \alpha =_{act} \beta \in EQ(M) \wedge \alpha, \beta \in T_{BA}(S)\}$$

where $T_{BA}(S)$ denotes the set of boolean terms built from variables in S.

Suppose that $\mathsf{D}_\exists(\varphi) = c$. If $\#\Delta_0 > Pr(\varphi) + c$, then we define a model $M^* = \langle \mathcal{W}^*, \mathcal{R}^*, \mathcal{E}^*, \mathcal{I}^*, \mathcal{P}^* \rangle$ over the vocabulary $V^* = \langle \Delta_0^* = Pr(\varphi) \cup \{b_1, \ldots, b_c\}, \Phi_0 \rangle$, $b_1, ..., b_c$ being fresh primitive actions.

- $\mathcal{E}^* = At(\Delta_0^*/EQ^{Pr(\varphi)}(M))$.
- $\mathcal{W}^* = W_w^\varphi$.
- For each $v \in W_w^\varphi$ let $\{e_1^v, ..., e_k^v\} \subseteq \mathcal{E}^{\varphi}_{w}$ be the set of events such that each e_i^v satisfies either:
  - there exists a state v_i and $v \xrightarrow{e_i} v_i \in \mathcal{R}_w^\varphi$, or
  - there is a $\mathsf{P_w}(\alpha_i) \in L(v)$ such that $e_i^v \in \mathcal{I}_w^\varphi(\alpha_i)$ and $\mathcal{P}_w^\varphi(v, e_i^v)$, or
  - there is a $\neg\mathsf{P}(\alpha_i) \in L(v)$ with $e_i^v \in \mathcal{I}_w^\varphi(\alpha_i)$ and $(v, e_i^v) \notin \mathcal{P}_w^\varphi$.

  We know that $k \leq \mathsf{D}_\exists(\varphi)$, and then define for each such a e_w^v a corresponding event in $\mathcal{E}^*$ as follows:

$$e_i^{v*} = \Big(\bigsqcap_{a \in Pr(\varphi) \wedge e_i^v \in \mathcal{I}(a)} a\Big) \sqcap \Big(\bigsqcap_{a' \in Pr(\varphi) \wedge e_i^v \notin \mathcal{I}(a')} \overline{a'}\Big) \sqcap \Big(\bigsqcap_{b_j \in \Delta_0' \wedge b_j \neq b_i} \overline{b_j}\Big) \sqcap b_i$$

  (where we use some enumeration of the fresh b's to determine each b_i); note that for these e_i^{v*}'s, we have: $e_i^v \in \mathcal{I}_w^\varphi \Leftrightarrow e_i^{v*} \in \mathcal{I}^*(\alpha)$, for each $\alpha \in T_{BA}(Pr(\varphi))$. Now we use these e_i^{v*}'s to define:

  - $\mathcal{R}^v = \{v \xrightarrow{e_i^{v*}} v_i \mid v \xrightarrow{e_i^v} v_i \in \mathcal{R}_w^\varphi\}$.
  - $\mathcal{P}^v = \{(v, e_i^{v*}) \mid \mathcal{P}_w^\varphi(v, e_i^v)\}$.

  Using these sets defined for each state v, we define:

$$\mathcal{R}^* = \bigcup_{v \in \mathcal{W}^*} \mathcal{R}^v$$

  and:

$$\mathcal{P}^* = \Big(\bigcup_{v \in \mathcal{W}^*} \mathcal{P}^v\Big) \cup \{(v, e) \mid v \in \mathcal{W}^* \wedge \mathsf{P}(\alpha) \in L(v) \wedge e \in \mathcal{I}^*(\alpha)\}.$$

- Define $\mathcal{I}^*(a_i) = \{[\gamma] \mid \vdash_{BA} \gamma \sqsubseteq a_i\}$, for every $a_i \in \Delta_0^*$.
- Define $\mathcal{I}^*(p_i) = \mathcal{I}_w^\varphi(p_i)$, for every atomic proposition p_i.

Let us prove that this new model preserves properties of L.

Theorem 16. *Given a vocabulary $V = \langle \Delta_0, \Phi_0 \rangle$, a NF-formula φ such that $\Delta_0 > Pr(\varphi) + \mathsf{D}_\exists(\varphi)$, with $\mathsf{d}(\varphi) = n$ and a model M, if $w, M \models \varphi$, then for the model M^* defined above over the vocabulary $V^* = \langle Pr(\varphi) \cup \{b_1, ..., b_{\mathsf{D}_\exists(\varphi)}\}, \Phi_0 \rangle$ with the b_i's being fresh action terms, we have $v, M^* \models \varphi$.*

Proof. *By the theorem above, we have that $w, M^\varphi_w \models \varphi$. As explained above, if we prove that $v, M^* \models L(v)$ for every v, we have that $w, M^* \models \varphi$. For the states reachable in n steps, we have that $L(v)$ contains only propositional variables or deontic predicates; for the propositional predicates the result is trivial. Now, suppose that $\mathsf{P}(\alpha) \in L(v)$, then $w, M^\varphi_w \models \mathsf{P}(\alpha)$, and then $v, M^* \models \mathsf{P}(\alpha)$, by the definition of M^*. If $\mathsf{P_w}(\alpha) \in L(v)$, then $v, M^\varphi_w \models \mathsf{P_w}(\alpha)$, and therefore there exists an $e_i \in \mathcal{I}^\varphi_w(\varphi)$ such that $\mathcal{P}^\varphi_w(v, e_i)$, but for this e_i we have a corresponding e^v_i such that $(v, e^v_i) \in \mathcal{P}^*$ and $e^v_i \in \mathcal{I}^*(\alpha)$, and therefore $v, M^* \models \mathsf{P_w}(\alpha)$. If $\neg\mathsf{P}(\alpha) \in L(v)$, then we have an $e_i \in I^\varphi_w$ such that $\neg\mathcal{P}(v, e_i)$; for this e_i, we have an $e^v_i \in \mathcal{P}^*$ and by definition of $\mathcal{P}^*$ we have $\neg\mathcal{P}^*(v, e^v_i)$, since $\neg\mathcal{P}^\varphi_w(v, e_i)$ and $\mathsf{P}(\alpha) \notin L(v)$, otherwise $L(v)$ is inconsistent. Therefore, $v, M^* \models \neg\mathsf{P}(\alpha)$. If $\neg\mathsf{P_w}(\alpha) \in L(v)$, then we have have $\neg\mathcal{P}^\varphi_w(e_i, v)$ for every $e_i \in \mathcal{I}^*$; if we have $\mathsf{P}(\alpha) \in L(v)$, then $\alpha =_{act} \emptyset$, an equation which is also true in M^* and therefore $v, M^* \models \neg\mathsf{P_w}(\alpha)$. If $\alpha \neq_{act} \emptyset$, then $\mathsf{P}(\alpha) \notin L(v)$, and there is no way to introduce a tuple (v, e^v_i) in $\mathcal{P}^*$, so $v, M^* \models \neg\mathsf{P_w}(\alpha)$.*

Now, suppose that v is reachable in $k < n$ steps from w; for the deontic predicates and propositional variables the proof proceeds as before. If $\langle\alpha_i\rangle\varphi_i \in L(v)$, then $v, M^\varphi_w \models \langle\alpha_i\rangle\varphi_i$, and so there is an $e_i \in \mathcal{I}^\varphi_w(\alpha)$ such that $v \xrightarrow{e_i} v_i$ and $v_i, M^\varphi_w \models \varphi_i$. Using induction we get $v_i, M^* \models \varphi_i$, and we have, by definition of M^*, that $v \xrightarrow{e^v_i} v_i$; this implies that $v, M^* \models \langle\alpha_i\rangle\varphi_i$. If $\neg\langle\beta_i\rangle\psi_i \in L(v)$, then $v, M^\varphi_w \models \psi_i$, which means that for all e_i such that $e_i \in \mathcal{I}^\varphi_w(\beta_i)$ and $v \xrightarrow{e_i} v_i$, we have $v_i, M^\varphi_w \models \neg\psi$. Since ψ_i is a NF formula it is a conjunction of formulae, i.e., $\psi_i = \psi^1_i \wedge \cdots \wedge \psi^m_i$, and, for some of these ψ_i's, we have $v, M^\varphi_w \models \neg\psi^j_i$, and by definition of L we have $\neg\psi^j_i \in L(v')$ and therefore, by induction, $v', M^* \models \neg\psi^j_i$, which implies $w, M^* \models \neg\langle\beta_i\rangle\psi_i$. The theorem follows. ■

Thus, $\mathsf{D}_\exists(\varphi)$ gives us a bound for the number of new primitive symbols that we need to verify a given formula. From this theorem we get the following corollary:

Corollary 8. *For any DNF formulae φ with $\mathsf{D}_\exists(\varphi) = n$, if we have a vocabulary $V = \langle \Delta_0, \Phi_0 \rangle$ and a model M of V such that $w, M \models \varphi$, then there exists a model M' of a vocabulary $V' = \langle Pr(\varphi) \cup \{b_1, ..., b_k\}, \Phi_0 \rangle$ such that $w', M' \models \varphi$ and $k \leq n$.*

Proof. *Suppose that $w, M \models \varphi$ for some M over a vocabulary V. Since φ is in DNF, we know that $\varphi = \varphi_1 \vee ... \vee \varphi_m$ (each φ_i being a NF formulae), and therefore $w, M \models \varphi_i$ for some i. By theorem 16 we know that there exists a model M' of a vocabulary $V' = \langle Pr(\varphi) \cup \{b_1, ..., b_k\}, \Phi_0 \rangle$ with $k = \mathsf{D}_\exists(\varphi_i) \leq \mathsf{D}_\exists(\varphi)$ such that $w', M' \models \varphi_i$ and then $w', M' \models \varphi$. (Note that we can ensure that each $Pr(\varphi) = Pr(\varphi_i)$, by adding the*

formulae $[a_1 \sqcup ... \sqcup a_t]\top$ *to each* φ_i *with* $Pr(\varphi) = \{a_1, ..., a_t\}$*; these formulae do not modify the truth value of the former one.)* ■

Roughly speaking, this theorem says that if a formula has a model in a language with an extra collection of k new variables (with $k > \mathsf{D}_\exists(\varphi) = n$), then if we add n new primitive actions, we also obtain a model. Or conversely, if we cannot get a model with n new primitive actions, we will not get a model by adding further primitive actions to the language. Because each formula is equivalent to a DNF formula, the above result gives us a bound for checking every formula. The method is as follows: given a formula φ, take its negation and get the DNF equivalent formula φ'; then develop a tableau taking into account at most $D^*_\exists(\varphi') = n$ primitive actions; if the tableau is closed, then the formula φ is valid for any extension of its vocabulary. Corollary 8 can be used to improve the completeness of the method, i.e., the tableaux is not only complete with respect to the semantics, but it is also complete with respect to language enrichment.

It is worth noting that we have two kinds of validities: we have formulae which are valid with respect to one vocabulary (i.e., these formulae are true with respect to all the models of this vocabulary). We can call this notion of validity *local validity*, and we have formulae which are valid with respect to every vocabulary, i.e., a *global validity*. For example, the formula $[a \cup b]\varphi \leftrightarrow [\mathbf{U}]\varphi$ is valid in the vocabulary $\langle\{a, b\}, \{p, q, s, \dots\}\rangle$ but it is not valid in the vocabulary $\langle\{a, b, c\}, \{p, q, s, \dots\}\rangle$. (It is important, when using equivalences, to distinguish between "global" equivalences or "local" equivalences.)

5.3 Some Examples

Now we give some examples. In figure 5.9 we build the tableau for the formula: $([\alpha]\varphi \wedge \langle\alpha\rangle\psi) \rightarrow \langle\alpha\rangle(\varphi \wedge \psi)$. This formula is one of the axioms given for dynamic logic in [HKT00]. The crosses at the end of each branch mean that those branches are closed. Note that here we are using a new action symbol.

We have added numbers in the formulae to better explain the example. Formula (1) is the negation of the formula to be proven. Formulae (2) and (3) are obtained by applying the rule A to the negation of the implication, formulae (4) and (5) are obtained from formula (2) using the A rule. Formulae (6) and (7) follow from formula (3) by application of the N rule. In a similar way, we obtained formulae (8) and (9) from formula 4. After that, we have branching using the P rule. Finally, we apply the B rule to formulae (6) and (7) and we obtain the leaves closing the tableau.

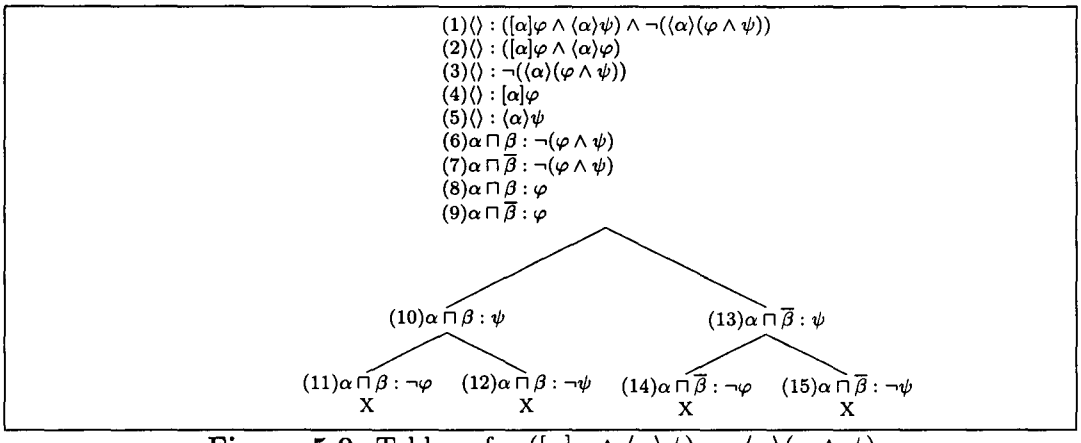


Figure 5.9: Tableau for $([\alpha]\varphi \wedge \langle\alpha\rangle\psi) \rightarrow \langle\alpha\rangle(\varphi \wedge \psi)$

Now, consider the following formula (which is not valid): $\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi$. The tableau for it is shown in figure 5.10. Note that, in this case, we use a new action

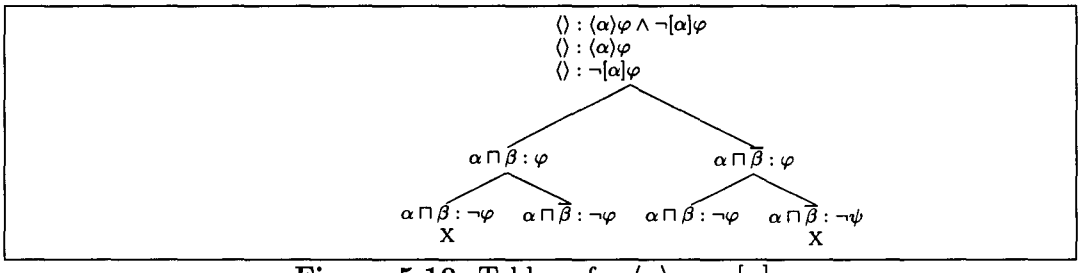


Figure 5.10: Tableau for $\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi$

symbol (following corollary 8). First, we reduce the implication, after that we use the rule P on the second formula, and then we use rule P again in the third formula. We can observe that this tableau has some open branches and using them we can build a "counterexample" (shown in figure 5.11). Note that we can use the labels in the formulae to put the labels on the transitions, indicating in this way which actions were executed and which were not.

Now, we apply the tableaux method to a deontic formula: $\mathsf{P}(\alpha) \wedge (\alpha \neq_{act} \emptyset) \rightarrow \mathsf{P}_{\mathsf{w}}(\alpha)$. We exhibit the corresponding tableau in figure 5.12. For building this tableau, the rule N_D is used together with the A rule, noting that the branch is closed since it is boolean closed; the predicates $\mathsf{P}(\alpha \sqcap \beta)$ and $\neg\mathsf{P}_{\mathsf{w}}(\alpha \sqcap \beta)$ introduce the equation $\alpha \sqcap \beta =_{act} \emptyset$ in the extended boolean theory of the branch. On the other hand, predicates $\mathsf{P}(\alpha \sqcap \overline{\beta})$ and $\neg\mathsf{P}_{\mathsf{w}}(\alpha \sqcap \overline{\beta})$ introduce the equation $\alpha \sqcap \overline{\beta} =_{act} \emptyset$ in the extended boolean theory. Keeping in mind that $\{\alpha \sqcap \beta =_{act} \emptyset, \alpha \sqcap \overline{\beta} =_{act} \emptyset\} \vdash_{BA} \alpha =_{act} \emptyset$, and that $\neg\alpha =_{act} \emptyset$ is in the actual branch, we conclude that the branch is boolean closed.

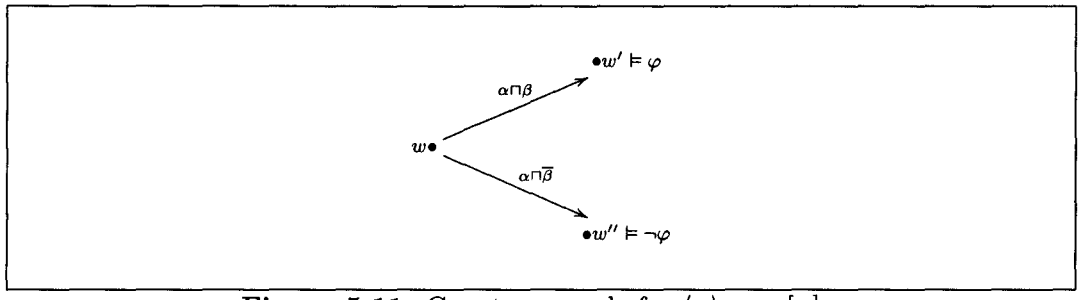


Figure 5.11: Counterexample for $\langle\alpha\rangle\varphi \rightarrow [\alpha]\varphi$

$\langle\rangle : (\mathsf{P}(\alpha) \wedge \neg\alpha =_{act} \emptyset) \wedge \neg\mathsf{P}_{\mathsf{w}}(\alpha)$
$\langle\rangle : (\mathsf{P}(\alpha) \wedge \neg\alpha =_{act} \emptyset)\ \langle\rangle : \mathsf{P}(\alpha)$
$\langle\rangle : \neg(\alpha =_{act} \emptyset)$
$\langle\rangle : \neg\mathsf{P}_{\mathsf{w}}(\alpha)\ \langle\rangle : \mathsf{P}(\alpha \sqcap \beta)$
$\langle\rangle : \mathsf{P}(\alpha \sqcap \overline{\beta})$
$\langle\rangle : \neg\mathsf{P}_{\mathsf{w}}(\alpha \sqcap \beta)$
$\langle\rangle : \neg\mathsf{P}_{\mathsf{w}}(\alpha \sqcap \overline{\beta})$
X

Figure 5.12: Tableau for $\mathsf{P}(\alpha) \wedge (\alpha \neq_{act} \emptyset) \rightarrow \mathsf{P}_{\mathsf{w}}(\alpha)$

We present a last example to illustrate the use in practice of theorem 8. In this example we consider a heating system. This system has two ways of being started: one is pressing a button, and the other one is by means of a sensor, which detects if the room is too cold. Let us analyze the following formula:

$$[\mathtt{get\_cold}]\mathtt{on} \wedge [\mathtt{get\_hot}]\neg\mathtt{on} \wedge \langle\overline{\mathtt{get\_cold}}\rangle\mathtt{on} \wedge \langle\overline{\mathtt{get\_hot}}\rangle\neg\mathtt{on}$$

This formula specifies the behaviour of the sensor (when it turns on the heating system). Also, it has some additional formulae which say that there exists other parts of the system that could turn on (or turn off) the heating mechanism. Note that here we do not know which are the other actions that can interfere, we only know that they exist; this is an *incomplete* specification. It could be the case that some other person is responsible for designing (and specifying) the interaction between the system and the user, and we have only to specify the behaviour of the sensor. However, if we build the tableau for this formula (note that we do not build the tableau for the negation of it, as in this case we are using the tableau to investigate if the given formula is consistent), we learn that this formula is inconsistent (see figure 5.13). Note that we added the equation $\mathtt{get\_cold} \sqcap \mathtt{get\_hot} =_{act} \emptyset$ in the formula, because if the two actions are mutually disjoint, this equation reduces our set of atomic boolean terms. The tableau below was built as follows. First, we apply rule A several times, then we apply rule P to line 7, and then we apply this rule again to line 8; finally, we use rule N twice.

At first sight it seems strange that this formula is inconsistent; the main problem here is that we have not considered some other actions in the specification. Taking

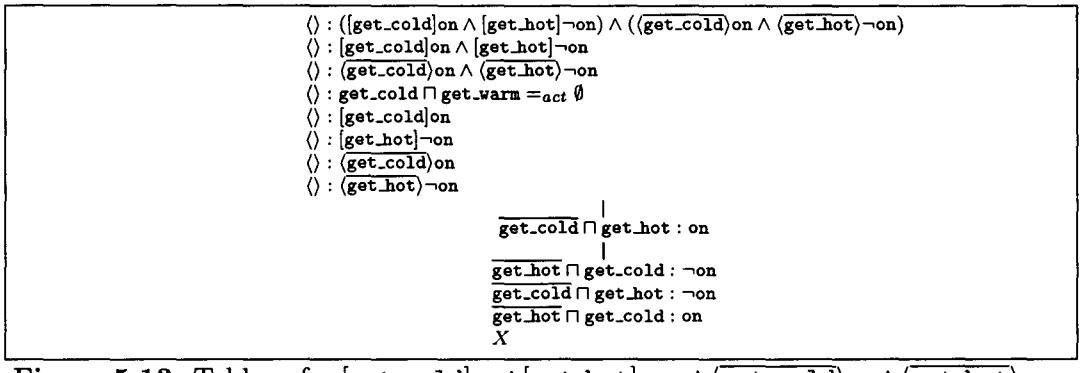


Figure 5.13: Tableau for [get_cold]on∧[get_hot]¬on∧⟨get_cold⟩on∧⟨get_hot⟩¬on

into account corollary 8, we add two more actions. Let us consider the actions: set_on and set_off. Intuitively, the first one turns on the heating system, and the second one turns it off. Again we can add some disjointness restriction on actions, namely: set_on ⊓ set_off $=_{act} \emptyset$ (we cannot press the two buttons at the same time), get_hot ⊓ set_off $=_{act} \emptyset$, get_cold ⊓ set_on $=_{act} \emptyset$, get_hot ⊓ set_on $=_{act} \emptyset$, get_cold ⊓ set_off $=_{act} \emptyset$ (the sensor is blocked when a user presses a button).

Note that these equations impose a strong restriction on the system: the sensor has to be shut off when a user presses a button. (Later on we shall see that we can manage this scenario using deontic predicates, in this way making the specification less restrictive, in a kind of fault-tolerant approach.) For the sake of simplicity, we do not put the equations into the tableau (but remember that they modify the set of atomic boolean terms), and we just show an open branch of the tableau (see figure 5.14). This open branch can be obtained if we apply rule P to formulae in lines 3 and 4 in the tree. Using this branch we can build the model illustrated by figure 5.15.

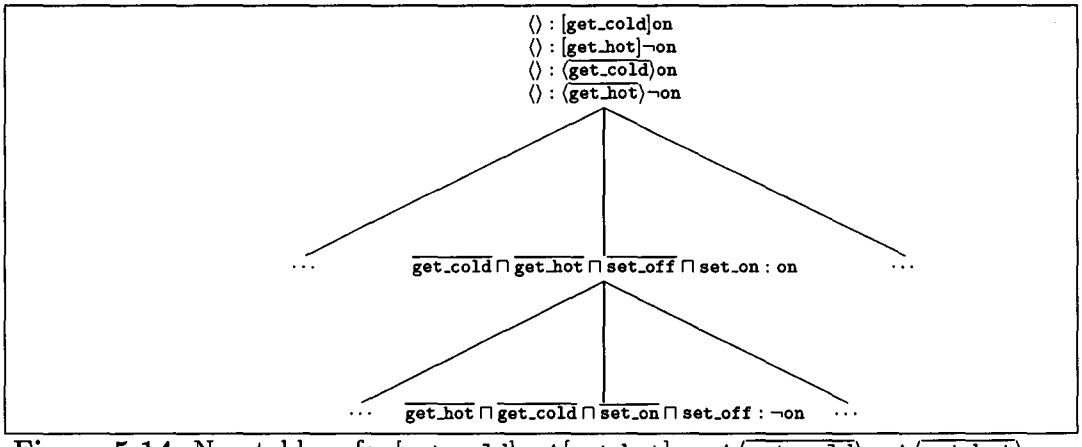


Figure 5.14: New tableau for [get_cold]on∧[get_hot]¬on∧⟨get_cold⟩on∧⟨get_hot⟩¬on

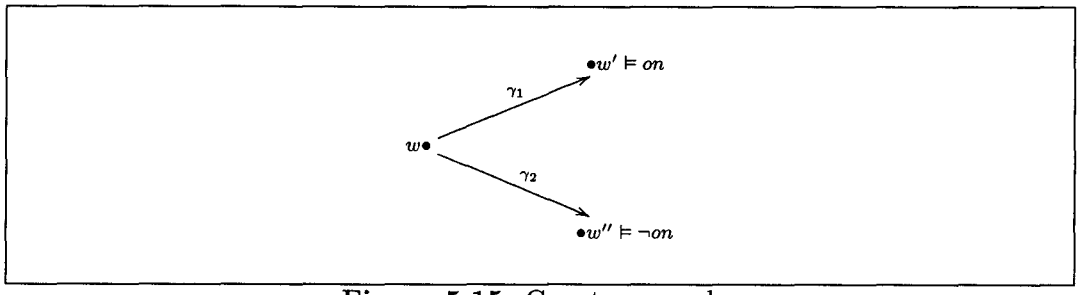


Figure 5.15: Counterexample

In this figure we set:

$$\gamma_1 = \overline{\texttt{get\_cold}} \sqcap \overline{\texttt{get\_hot}} \sqcap \overline{\texttt{set\_off}} \sqcap \texttt{set\_on}$$

and

$$\gamma_2 = \overline{\texttt{get\_hot}} \sqcap \overline{\texttt{get\_cold}} \sqcap \overline{\texttt{set\_on}} \sqcap \texttt{set\_off}$$

Thus, the new actions in the vocabulary allow us to build a model which satisfies the formula. In section 5.6 we extend this example with temporal operators and deontic features, to provide a broader example.

5.4 Extending the tableaux to temporal logics

In the previous sections, we described a tableaux method for the propositional deontic logic introduced in chapter 3; we can extend this method in order to deal with the temporal extension of the deontic logic. The main idea behind this extension is to use the fixed point definition of temporal operators (as is done in [GM96] and [SW91] for dynamic logic and the μ-calculus, respectively). We show that this temporal extension of the tableaux calculus is sound and complete with respect to the given semantics. In addition, in chapter 6, we use this tableaux calculus to prove the completeness of the axiomatic temporal system described in chapter 3; we leave this as a future work.

Recall from chapter 3 that we enrich the language with the following formulae: $\mathsf{AN}\varphi$ (for all paths, in the next instant in time φ is true), $\mathsf{A}(\varphi\, \mathcal{U}\, \psi)$ (for all paths, φ is true until ψ becomes true) and $\mathsf{E}(\varphi\, \mathcal{U}\, \psi)$ (for some path, φ is true until ψ becomes true). Using them we can define the remainder of the usual CTL operators (see [EH82]). As usual in CTL, we can give the following characterization of the temporal operators:

- $\mathsf{A}(\varphi\, \mathcal{U}\, \psi) \equiv \psi \vee (\varphi \wedge \neg\psi \wedge \mathsf{AN}(\mathsf{A}(\varphi\, \mathcal{U}\, \psi)))$

- $\mathsf{E}(\varphi\,\mathcal{U}\,\psi) \equiv \psi \vee (\varphi \wedge \neg\psi \wedge (\mathsf{EN}(\mathsf{E}(\varphi\,\mathcal{U}\,\psi))))$

This means that we can define the temporal operators as fixed points of the appropiate functional (see [Eme90] for details). We use this idea during the development of our extended tableaux. The same technique is used in [SW91] for the μ-calculus, where a given model is used to guide the construction of the tableau. In [GM96], fixed points are also used to develop a tableaux calculus for dynamic logic with converse. In both works, new constants are added to detect branches which cannot satisfy a given formula. Here the nature of the temporal operators (and the fact that we mix them with modal logic operators) allows us to state the rules without these variables. In addition, our tableaux rules for the propositional case allows us to deal with intersection and the complement. We introduce the following rules for temporal logic: One of the benefits of having labels in the formula describing the

$$\mathsf{N}: \frac{\sigma : \mathsf{AN}\varphi}{\begin{array}{l|l}\sigma : \langle \mathbf{U}\rangle \top & \sigma : \neg\langle \mathbf{U}\rangle \top \\ \sigma : [\mathbf{U}]\varphi & \sigma : \varphi\end{array}} \qquad \neg\mathsf{N}: \frac{\neg\mathsf{AN}\varphi}{\begin{array}{l|l}\sigma : \langle \mathbf{U}\rangle \top & \sigma : \neg\langle \mathbf{U}\rangle \top \\ \sigma : \langle \mathbf{U}\rangle \neg\varphi & \sigma : \neg\varphi\end{array}}$$

Figure 5.16: Rules for N

$$\mathsf{E}\,\mathcal{U}: \frac{\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)}{\begin{array}{l|l}\sigma : \neg\psi & \\ \sigma : \psi & \sigma : \varphi \\ & \sigma : \neg\mathsf{AN}\neg\mathsf{E}(\varphi\,\mathcal{U}\,\psi)\end{array}} \qquad \neg\mathsf{E}\,\mathcal{U}: \frac{\sigma : \neg\mathsf{E}(\varphi\,\mathcal{U}\,\psi)}{\begin{array}{ll|ll}\sigma : \neg\psi & & & \sigma : \neg\psi \\ \sigma : \varphi & & | & \sigma : \neg\varphi \\ \sigma : \mathsf{AN}\neg\mathsf{E}(\varphi\,\mathcal{U}\,\psi) & & & \end{array}}$$

Figure 5.17: Rules for $\mathsf{E}\,\mathcal{U}$

$$\mathsf{A}\,\mathcal{U}: \frac{\sigma : \mathsf{A}(\varphi\,\mathcal{U}\,\psi)}{\begin{array}{l|l}\sigma : \neg\psi & \\ \sigma : \psi & \sigma : \varphi \\ & \sigma : \mathsf{ANA}(\varphi\,\mathcal{U}\,\psi)\end{array}} \qquad \neg\mathsf{A}\,\mathcal{U}: \frac{\sigma : \neg\mathsf{A}(\varphi\,\mathcal{U}\,\psi)}{\begin{array}{ll|ll}\sigma : \varphi & & & \sigma : \varphi \\ \sigma : \neg\varphi & & | & \sigma : \neg\psi \\ \sigma : \neg\psi & & & \sigma : \neg\mathsf{ANA}(\varphi\,\mathcal{U}\,\psi)\end{array}}$$

Figure 5.18: Rules for $\mathsf{A}\,\mathcal{U}$

actions performed is that we can check the Done() straightforwardly. We introduce the notion of a branch that is *done-closed.*

Definition 30. *A branch $\mathcal{B}$ is done-closed iff we have either*

- $\sigma \centerdot \gamma : \mathsf{Done}(\alpha) \in \mathcal{B}$ *and* $EQ(\mathcal{B}) \nvdash_{BA} \gamma \sqsubseteq \alpha$, *or*
- $\sigma \centerdot \gamma : \neg\mathsf{Done}(\alpha) \in \mathcal{B}$ *and* $EQ(\mathcal{B}) \vdash_{BA} \gamma \sqsubseteq \alpha$, *or*
- $\langle\rangle : \mathsf{Done}(\alpha) \in \mathcal{B}$, *for some* α.

□

In a similar way, we redefine the notion of closed branch.

Definition 31. *A branch is closed iff either*

- $\sigma : \varphi \in \mathcal{B}$ *and* $\sigma : \neg\varphi \in \mathcal{B}$, *for some formula* φ, *or*
- $\mathcal{B}$ *is boolean closed, or*
- $\mathcal{B}$ *is deontic closed, or*
- $\mathcal{B}$ *is* Done() *closed.*

□

The following definitions are needed to prove the soundness and completeness of the tableaux system; mainly, they are adaptations of similar definitions for other logics (e.g., the several modal logics presented in [Gor95], the dynamic logic introduced in [GM96] and the modal logics of [Fit72]).

Definition 32 (t-mapping). *A t-mapping is a mapping ι between the labels of a set S of formulae and a model M which satisfies:*

- $\iota(\langle\rangle) = w$, *where w is the initial state of M (we suppose that there is a formula labelled with the empty string in S).*
- *for all traces π, if $\sigma : \varphi \in S$, then, if $\pi_i = \iota(\sigma)$, it follows that: $\pi, i, M \models \varphi$*

□

A t-mapping can be used to define when a given branch is satisfiable; we say that a branch $\mathcal{B}$ is SAT iff there exists a mapping ι and a model M such that ι is a t-mapping for $\mathcal{B}$.

Definition 33. $\mathcal{B}/\sigma = \{\varphi \mid \sigma : \varphi \in \mathcal{B}\}$ *(that is, $\mathcal{B}/\sigma$ are the formulae of branch $\mathcal{B}$ with prefix σ).* □

Definition 34. *A prefix σ is reduced if all the rules (except P) have been applied to it. That is, we do not generate new states from it. It is fully reduced if all the rules were applied to it.* □

Definition 35. *A prefix σ' is a copy of a prefix σ if:*

- $\mathcal{B}/\sigma = \mathcal{B}/\sigma'$, *i.e., they have the same formulae.*
- *There exists some $\gamma \in At(\Delta_0/\Gamma)$: $\sigma = \sigma_0 . \gamma$ and $\sigma' = \sigma'_0 . \gamma$; in other words, the last executed action is the same.*

□

Definition 36. *A branch $\mathcal{B}$ is t-completed iff:*

- *all prefixes are reduced.*
- *for every σ' which is not fully reduced there is a shorter copy σ (see the definition above) which is fully reduced.*
- *If it is an N formula, then all the formulae resulting from applying the corresponding rule belong to $\mathcal{B}$.*

□

The difference between this definition and definition 26 is that in the latter we also require the saturation of P rules; instead, above we only require this for fully reduced labels (the reduced labels are a copy of some fully reduced label in the branch). In [GM96] ignorable branches are introduced to capture the scenario of a given branch which cannot satisfy a DPL formula $[\alpha^*]\varphi$ since $\neg\varphi$ is true in every state (i.e., at every label). Here we have two different cases: a branch cannot satisfy a formula $\mathsf{A}(\varphi \mathcal{U} \psi)$ since ψ is never true for some sequence of labels σ, or it cannot satisfy a formula $\mathsf{E}(\varphi \mathcal{U} \psi)$ since for every sequence of labels σ, ψ is never true. That is, we define two kinds of ignorable branches: A-ignorable branches and E-ignorable branches. In these definitions, we say that $\sigma \leq \sigma'$ if there exists some label σ'' such that $\sigma . \sigma'' = \sigma'$, and $\sigma \prec \sigma'$ when for some atomic term γ we have $\sigma . \gamma = \sigma'$.

Definition 37. *A branch $\mathcal{B}$ is* E*-ignorable with respect to a formula $\sigma : \mathsf{E}(\varphi \mathcal{U} \psi) \in \mathcal{B}$ iff:*

- *it is t-complete and not closed.*
- $\sigma : \neg\psi \in \mathcal{B}$.
- *for every prefix $\sigma' \geq \sigma$, we have that $\sigma' : \mathsf{E}(\varphi \mathcal{U} \psi), \sigma' : \neg\psi \in \mathcal{B}$.*

□

In a similar way we can define A-ignorable branches.

Definition 38. *A branch $\mathcal{B}$ is* A*-ignorable with respect to a formula $\sigma : \mathsf{A}(\varphi \mathcal{U} \psi) \in \mathcal{B}$ iff*

- *it is t-complete and not closed.*
- $\sigma : \neg\psi \in \mathcal{B}$.
- *if there exists some $\sigma' \succ \sigma$, then there exists a maximal chain $\sigma \prec \sigma_1 \prec \sigma_2 \prec ...$, such that for all i we have $\sigma_i : \mathsf{A}(\varphi \mathcal{U} \psi), \sigma_i : \neg\psi \in \mathcal{B}$.*

□

Now we redefine the notion of open branch.

Definition 39. *A branch is open if it is t-completed and neither closed,* E*-ignorable nor* A*-ignorable.* □

Using these definitions we can prove soundness:

Theorem 17 (Soundness). *If a branch $\mathcal{B}$ of tableau $\mathcal{T}$ is SAT wrt a model M and a mapping ι, then a branch $\mathcal{B}'$, obtained by some of the given tableaux rules, is SAT wrt the model M and the mapping ι' which coincides with ι over the labels in $\mathcal{B}$.*
Proof. *The proof is by cases; for non-temporal connectives the proof is the same as the one given in theorem 11. For the temporal operators we have the following cases.*
Case AN*: If $\mathcal{B} \cup \{\sigma : \mathsf{AN}\varphi\}$ is SAT, this means that for all π (such that $\pi_i = \iota(\sigma)$) we have: $\pi, i+1, M \vDash \varphi$. If $\iota(\sigma)$ is not related to any other state in M, then we have $\iota(\sigma), M \vDash [\mathbf{U}]\bot$, and therefore we have that $\pi, i, M \vDash \varphi$, and then $\mathcal{B} \cup \{\sigma :$*

$[\mathbf{U}]\varphi, \sigma : \varphi\}$ *is SAT wrt* ι *and* M. *On the other hand, if* $\iota(\sigma), M \models \langle\mathbf{U}\rangle\top$, *then for every path* π' *such that* $\pi[0..i] \prec \pi'$ *we have that* $\pi', i+1, M \models \varphi$, *i.e., we have that* $\mathcal{B} \cup \{\sigma : \langle\mathbf{U}\rangle\top, \sigma : [\gamma_1]\varphi, ..., \sigma : [\gamma_n]\varphi\}$ *(for every* $\gamma_i \in At(\Delta_0/\Gamma)$*) is SAT in* ι *and* M, *and then* $\mathcal{B} \cup \{\sigma : \langle\mathbf{U}\rangle\top, \sigma : [\mathbf{U}]\varphi\}$ *is SAT wrt* ι *and* M.

Case $\neg$AN: *Similar to the case above.*

Case $\mathsf{A}(\varphi\,\mathcal{U}\,\psi)$: *Suppose that* $\mathcal{B} \cup \{\sigma : \mathsf{A}(\varphi\,\mathcal{U}\,\psi)\}$ *is SAT for a model* M *and a mapping* ι, *this means that for all sequences* π *such that* $\iota(\sigma) = \pi_i$ *we have* $\pi, i, M \models \mathsf{A}(\varphi\,\mathcal{U}\,\psi)$, *which by definition means that for every* π' *such that* $\pi[0..i] \preceq \pi'$, *there exists an* $i \leq k$ *for which* $\pi', k, M \models \psi$ *and for every* $i \leq j \leq k$ *we have* $\pi', j, M \models \varphi$. *Using a similar reasoning to the first case, we get that either* $\mathcal{B} \cup \{\sigma : \varphi, \sigma : \neg\psi, \sigma : \mathsf{ANA}(\varphi\,\mathcal{U}\,\psi)\}$ *or* $\mathcal{B} \cup \{\sigma : \psi\}$ *are SAT for* ι *and* M.

Case $\neg\mathsf{A}(\varphi\,\mathcal{U}\,\psi)$: *Similar to the case below.*

Case $\mathsf{E}(\varphi\,\mathcal{U}\,\psi)$: *Suppose that* $\mathcal{B} \cup \{\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)\}$ *is SAT for a model* M *and mapping* ι, *then we have that for every* π *with* $\iota(\sigma) = \pi_i$, $\pi, i, M \models \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$, *i.e., there exists some* π' *such that there exists some* k *for which* $\pi', k, M \models \psi$ *and for every* $i \leq j \leq k$ *we have that* $\pi', j, M \models \varphi$. *If* $\pi, i, M \models \psi$, *then* $\mathcal{B} \cup \{\sigma : \psi\}$ *is SAT wrt* ι *and* M. *Otherwise, we have* $\pi, i, M \models \varphi$ *and* $\pi, i, M \models \mathsf{ENE}(\varphi\,\mathcal{U}\,\psi)$, *and therefore* $\mathcal{B} \cup \{\sigma : \varphi, \sigma : \neg\mathsf{AN}\neg\mathsf{E}(\varphi\,\mathcal{U}\,\psi)\}$ *is SAT wrt* ι *and* M.

Case $\neg\mathsf{E}(\varphi\,\mathcal{U}\,\psi)$: *Similar to the last case.* ■

The key point is to prove that we can discard ignorable branches safely.

Theorem 18. *If a tableau* $\mathcal{T}$ *has a branch* $\mathcal{B}$ *which is SAT for a mapping* ι *and model* M, *and it has a formula* $\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$, *then there is a t-complete tableau* $\mathcal{T}'$, *obtained from* $\mathcal{T}$ *applying the rules, such that it has a SAT branch* $\mathcal{B}'$ *which* $\mathcal{B} \subseteq \mathcal{B}'$ *and it is not* E*-ignorable for the formula* $\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$.

Proof. *Suppose that there exists a model* M *and an assignment* ι *such that* $\pi, i, M \models \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$, *i.e., in* M *we have* π' *such that* $\pi[0..i] \prec \pi'$ *such that* $\pi', k, M \models \psi$ *for some* k, *and* $\pi', j, M \models \varphi$, *for every* $i \leq j \leq k$. *If* $\pi, i, M \models \psi$, *then* $\mathcal{B} \cup \{\sigma : \psi\}$, *which is obtained by rule* $\mathsf{E}\,\mathcal{U}$, *is SAT wrt* ι *and* M. *By definition, this branch is not ignorable. Otherwise, suppose* $\pi^i = \iota(w_i) \xrightarrow{e_1} w_1 \xrightarrow{e2} \ldots \xrightarrow{e_k} w_k \xrightarrow{e_{k+1}} \ldots$, *where* $e_i \in \gamma_i$, *for every* i. *Now, if we apply the rule* $\mathsf{E}\,\mathcal{U}$ *and* $\neg\mathsf{AN}$ *to* $\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$ *we get a branch* $\mathcal{B} \cup \{\sigma : \mathsf{ENE}(\varphi\,\mathcal{U}\,\psi), \sigma : \varphi, \sigma : \langle\mathbf{U}\rangle\varphi, \sigma \centerdot \gamma_i : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)\}$, *which is SAT since we can extend* ι *defining* $\iota(\sigma \centerdot \gamma_i) = w_1$; *moreover, by theorem 17 we can get a t-complete branch* $\mathcal{B}'$ *which extends the branch shown above, and we can repeat this argument until we obtain a SAT t-complete branch* $\mathcal{B}'$ *which contains:*

$$\{\sigma \centerdot \gamma_1 : \mathsf{E}(\varphi\,\mathcal{U}\,\psi), \sigma \centerdot \gamma_1 : \varphi, ..., (\sigma \centerdot \cdots \centerdot \gamma_k) : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)\}.$$

This branch cannot be ignorable for $\sigma : \mathsf{E}(\varphi\,\mathcal{U}\,\psi)$, *otherwise we have* $\pi', k, M \models \psi$ *and* $\pi', k, M \models \neg\psi$, *which is a contradiction.* ■

A similar result can be proven for A-ignorable branches.

Theorem 19. *If a tableau $\mathcal{T}$ has a branch $\mathcal{B}$ which is SAT for a mapping ι and model M, and it has a formula $\sigma : \mathsf{A}(\varphi\, \mathcal{U}\, \psi)$, then there is a t-complete tableau $\mathcal{T}'$, obtained from $\mathcal{T}$ by applying the rules, such that it has a SAT branch $\mathcal{B}'$ for which $\mathcal{B} \subseteq \mathcal{B}'$ and it is not* A*-ignorable for the formula $\sigma : \mathsf{A}(\varphi\, \mathcal{U}\, \psi)$.*
Proof. *Similar to theorem 18.* ■

The soundness of the system follows from the theorems above; if we have a tableau for a set S of formulae which has all closed or ignorable branches, then the set S is not satisfiable. Therefore, if we start with the set $\{\neg\varphi\}$ and we have a closed tableau for it, we know that $\neg\varphi$ is not SAT, hence φ is valid.

5.4.1 Completeness and Decidability

We use the model presented in theorem 12 for proving the completeness; we make some modifications to it to avoid infinite models. (Actually, we have the finite model property, see below.) First, we present some properties about t-completed branches. These properties are useful for the proof of completeness, and to prove that the algorithm shown below terminates. The *closure* of a formula φ ($\mathsf{cl}(\varphi)$) can be defined in a similar way to its definition in [Eme90] and [FL79], i.e., given a formula φ and a set of equations Γ, we define $\mathsf{cl}(\varphi)$ with respect to Γ as the least set satisfying:

- $\varphi \in \mathsf{cl}(\varphi)$.
- If φ' is a subformula of φ, then $\varphi' \in \mathsf{cl}(\varphi)$.
- If $\mathsf{AN}\varphi' \in \mathsf{cl}(\varphi)$, then $\langle \mathbf{U} \rangle \top, [\mathbf{U}]\varphi' \in \mathsf{cl}(\varphi)$.
- If $\mathsf{A}(\varphi'\, \mathcal{U}\, \psi) \in \mathsf{cl}(\varphi)$, then $\mathsf{ANA}(\varphi'\, \mathcal{U}\, \psi) \in \mathsf{cl}(\varphi)$.
- If $\mathsf{E}(\varphi'\, \mathcal{U}\, \psi) \in \mathsf{cl}(\varphi)$, then $\neg\mathsf{AN}\neg\mathsf{A}(\varphi'\, \mathcal{U}\, \psi) \in \mathsf{cl}(\varphi)$.
- If $\mathsf{P}(\alpha)$ or $\mathsf{P}_\mathsf{w}(\alpha) \in \mathsf{cl}(\varphi)$, then $\mathsf{P}(\gamma_i) \in \mathsf{cl}(\varphi)$ or $\mathsf{P}_\mathsf{w}(\gamma_i) \in \mathsf{cl}(\varphi)$, respectively, where $\gamma_i \in At(\Delta_0/\Gamma)$.

The extended closure of φ ($\mathsf{ecl}(\varphi)$) is defined as:

$$\mathsf{ecl}(\varphi) = \mathsf{cl}(\varphi) \cup \neg\mathsf{cl}(\varphi)$$

It is not hard to prove that $\mathsf{ecl}(\varphi)$ is a finite set for any Γ and any φ. It is worth noting that, if we apply any rule described above to φ, we obtain a set of formulae

belonging to $\mathsf{ecl}(\varphi)$; i.e., suppose that we start a tableau with a finite set Φ, then using the rules we can only add formulae belonging to $\mathsf{ecl}(\Phi)$ (where $\mathsf{ecl}(\Phi) = \bigcup_{\varphi\in\Phi} \mathsf{ecl}(\varphi)$). This property is useful when proving the properties stated below.

First, we prove that any infinite chain of labels $\sigma_1 \prec \sigma_2 \prec \sigma_3 \ldots$ in a t-completed branch $\mathcal{B}$ which belongs to a tableau $\mathcal{T}$ (obtained by applying the rules to a finite set Γ) must contain labels which are copies of other labels.

Property 5. *Given a t-completed branch $\mathcal{B}$ of a tableau $\mathcal{T}$ obtained using the rules starting with a finite set of formulae Γ, if we have an infinite chain $\sigma_1 \prec \sigma_2 \ldots$ in $\mathcal{B}$, then there is at least one label σ_j in the chain which is a copy of a label σ_i (where $i < j$).*

Proof. *Given a finite set of formulae Γ, the set $\mathsf{ecl}(\Gamma)$ is finite, and so is the powerset $\wp(\mathsf{ecl}(\Gamma))$; now for any $\sigma \in \mathcal{B}$, we have $\mathcal{B}/\sigma \in \wp(\mathsf{ecl}(\Gamma))$. We also have a finite set of atoms of $\Delta_0/EQ^*(B)$, say $\gamma_1, \ldots, \gamma_n$. Since $\wp(\mathsf{ecl}(\Gamma))$ is finite, some set of formulae $\mathcal{B}/\sigma$, for some label σ, appears infinitely often in the chain. Take all the labels $\sigma^1, \sigma^2, \ldots$ in the chain such that $\sigma^j/\mathcal{B} = \sigma/\mathcal{B}$; we have a finite number of $\gamma_1, \ldots, \gamma_n$, i.e., in the chain there must be an infinite number of labels σ^j with the same set $\sigma^j/\mathcal{B}$ and the same last atom (say γ_k), that is, we have an infinite number of copies in the chain.* ■

Using this theorem we can show that, if we do not apply rule P to formulae with labels which are copies of other labels, then any chain of labels in a t-complete branch is finite.

Property 6. *If $\mathcal{B}$ is a t-completed branch of a tableau $\mathcal{T}$ obtained by applying the rules, such that we have not applied the P rule to formulae labeled with copies of other labels, then any chain of labels $\sigma_1 \prec \sigma_2 \ldots$ in $\mathcal{B}$ is finite.*

Proof. *Suppose that we have an infinite chain $\sigma_1 \prec \sigma_2 \ldots$. By property 5, we have some σ_j which is a copy of some σ_i, but then the label σ_{j+1} is obtained from σ_j by a P-rule (there is no other way to create labels in the calculus). This contradicts the condition in the theorem, and therefore there is no such a chain.* ■

A corollary of these two properties is that any infinite chain of labels of a t-completed branch $\mathcal{B}$ contains only a finite number of labels which are not a copy of other labels in the chain.

Theorem 20. *If $\mathcal{B}$ is a t-completed open branch, then it is SAT.*

Proof. *We build the model in the same way that we did in theorem 12. That is, we define $\mathcal{M}$ as follows:*

- *$\mathcal{W} = F$, where F is the set of prefixes appearing in $\mathcal{B}$.*

- $\mathcal{E} = At(\Delta_0/EQ(\mathcal{B}))$.
- $w = \langle\rangle$ *(initial state).*
- $p \in \mathcal{I}(\sigma) \Leftrightarrow \sigma : p \in \mathcal{B}$.
- $\mathcal{I}(\alpha) = \{[\gamma] \in At(\Delta_0/EQ(\mathcal{B})) \mid \vdash_{BA} \gamma \sqsubseteq \alpha\}$.
- $\mathcal{R} = \{\sigma \xrightarrow{\gamma} \sigma \centerdot \gamma \mid \sigma, \sigma \centerdot \gamma \in F\}$.
- $\mathcal{P} = \{(\sigma, \gamma) \mid \sigma : \mathsf{P}(\gamma) \in \mathcal{B}\}$.

We modify this model to take into account the cycles. If σ' is a copy of a shorter prefix σ then we replace each $\sigma'' \xrightarrow{\gamma} \sigma'$ by $\sigma'' \xrightarrow{\gamma} \sigma$. And the mapping is as follows:

$$\iota(\sigma') = \begin{cases} \sigma & \text{if } \sigma \text{ is a shorter copy of } \sigma'; \\ \sigma' & \text{otherwise} \end{cases}$$

Note that σ' can be safely replaced by σ, since both have the same properties.

Now, we prove that: $\sigma : \varphi \in \mathcal{B} \Rightarrow \pi, i, \mathcal{M} \vDash \varphi$ where $\pi_i = \sigma$. For the standard formulae, the proof follows the same pattern as for theorem 12, by induction. The other cases are as follows:

Suppose that $\sigma : \mathsf{Done}(\alpha) \in \mathcal{B}$. By definition of a done-closed branch, we know that $\sigma \neq \langle\rangle$; then we have $\sigma = \sigma' \centerdot \gamma_i$, and since the branch is open, we have that $EQ^*(\mathcal{B}) \vdash_{BA} \gamma_i \sqsubseteq \alpha$, and therefore, by definition of $\mathcal{M}$, we get $\pi, i, \mathcal{M} \vDash \mathsf{Done}(\alpha)$ where $\pi_i = \sigma$. When we have $\sigma : \neg\mathsf{Done}(\alpha) \in \mathcal{B}$, the proof is similar.

If $\sigma : \mathsf{E}(\varphi \,\mathcal{U}\, \psi) \in \mathcal{B}$, and supposing that $\pi, i, \mathcal{M} \nvDash \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$ for some path π such that $\pi_i = \sigma$, then we have either:

1. *$\pi, i, \mathcal{M} \vDash \neg\varphi \wedge \neg\psi$, or*
2. *for all π' with $\pi[0..i] \preceq \pi$ we have $\pi', j, \mathcal{M} \vDash \neg\psi$ for every $i \leq j$.*

The first case is not possible since in $\mathcal{B}$, by application of rule $\mathsf{E}\,\mathcal{U}$, we have $\sigma : \varphi$ or $\sigma : \psi$. In either case, by induction, we get a contradiction. In the second case, by definition of $\mathcal{M}$, we have a maximal sequence $\sigma_1 \prec \sigma_2 \prec \sigma_3 \ldots$ in $\mathcal{B}$ such that $\pi_i = \sigma, \pi_{i+1} = \sigma_1, \pi_{i+2} = \sigma_2, \ldots$, where at some point in π the states start repeating (since, by property 6, the number of labels which are not copies of other labels is finite). We know that we cannot have $\sigma : \neg\psi, \sigma_1 : \neg\psi, \sigma_2 : \neg\psi, \ldots$ for every label in the chain, since in this case the branch is E-ignorable, and therefore, by several

applications of the E *rule, we have some* σ_j *such that* $\sigma_j : \psi \in \mathcal{B}$*, from which, using induction, we get a contradiction, and therefore,* $\pi, i, \mathcal{M} \models \mathsf{E}(\varphi \mathcal{U} \psi)$.

Suppose that $\sigma : \mathsf{A}(\varphi \mathcal{U} \psi) \in \mathcal{B}$*, and suppose that* $\pi, i, M \models \neg\mathsf{A}(\varphi \mathcal{U} \psi)$*, where* $\pi_i = \sigma$*. Then, by definition of* $\models$*, we have either:*

- $\pi, i, M \models \neg\varphi \wedge \neg\psi$*, or*
- *there exists a path* π' *such that* $\pi[0..i] \preceq \pi$ *and* $\pi', j, M \models \neg\psi$ *for every* $j \geq i$.

The first case, as shown above, is not possible. If the second case is true, then, by definition of $\mathcal{M}$*, we have some maximal sequence* $\sigma \prec \sigma_1 \prec \sigma_2 \ldots$ *such that* $pi'_{i+1} = \sigma_1, \pi'_{i+2} = \sigma_2 \ldots$*, where at some point the states start repeating because of property 6. For this chain in* $\mathcal{B}$*, we cannot have* $\sigma : \neg\psi, \sigma_1 : \neg\psi, \ldots$*, otherwise* $\mathcal{B}$ *is* A*-ignorable; and therefore by application of rule* A *we must have some* σ_j *in the chain such that* $\sigma_j : \psi$ *belongs to the branch, and then,* $\pi', i + j, \mathcal{M} \models \psi$*, which is a contradiction, and therefore* $\pi, i, M \models \mathsf{A}(\varphi \mathcal{U} \psi)$.

If $\sigma : \neg\mathsf{E}(\varphi \mathcal{U} \psi)$ *and suppose* $\pi, i, M \models \mathsf{E}(\varphi \mathcal{U} \psi)$ *where* $\pi_i = \sigma$*, then for some* π' *with* $\pi[0..i] \preceq \pi'$ *and some* k *we have* $\pi', k, M \models \psi$ *and for every* $j \leq k$ *we have* $\pi', j, M \models \varphi$*; that is, we have a sequence* $\sigma \prec \sigma_1 \prec \sigma_2 \ldots$ *such that* $\pi'_i = \sigma, \pi'_{i+1} = \sigma_1, \pi'_{i+2} = \sigma_2, \ldots$*, where at some point the states start repeating by property 6. For* σ *we have either* $\{\sigma : \neg\varphi, \sigma : \neg\psi\} \subseteq \mathcal{B}$ *or* $\{\sigma : \varphi, \sigma : \neg\psi, \sigma : \mathsf{AN}\neg\mathsf{E}(\varphi \mathcal{U} \psi)\} \subseteq \mathcal{B}$*. The first statement is not possible. If we have the second, then applying the rules* AN*,* N *and* $\mathsf{E}\,\mathcal{U}$ *several times we get that* $\sigma : \neg\psi, \sigma_1 : \neg\psi, \sigma_2 : \neg\psi, \ldots$ *for every label in the chain. This implies that* $\pi', k, \mathcal{M} \not\models \psi$*, which is a contradiction, and therefore* $\pi, i, \mathcal{M} \models \psi$.

The case $\sigma : \neg\mathsf{A}(\varphi \mathcal{U} \psi)$ *is similar to the last case above. The cases for standard operators are as explained in theorem 12.* ■

From this theorem, completeness with respect to anchored validity follows.

Theorem 21. *If* $\models_A \varphi$*, then any t-completed tableau obtained from* $\{\neg\varphi\}$ *does not have open branches.*

Proof. *Suppose that we have some open branch in the tableau; then, by theorem 21, we have a model* $\mathcal{M}$ *such that* $\pi, 0, \mathcal{M} \models \neg\varphi$ *for some* π*, and therefore* φ *is not valid.*■

We have proved the completeness of the tableaux rules; however, we have not described any procedure which allows us to apply the rules to obtain t-complete

tableaux from a finite set of formulae. In figure 5.19 (found at the end of the chapter) we describe, informally, an algorithm which applies the rules in a "breadth-first" way; this procedure is a modification of the algorithm given in [Gor95] for modal logics. The important point to keep in mind is that we mark formulae as "*asleep*" (which are not used at the current step in the algorithm, but can be used later on), "*awake*" (formulae which could be used during the current step in the algorithm) or "*finished*" (formulae which will not be used in further steps). This allows us to prevent situations where the rule N is not applied to the correct labels, e.g., when, after an application of the N-rule, there follows the application of the P rule, adding a label which was not taken into account during the application of the N rule. Also, it is worth noting that we can detect the labels which are copies of other labels, preventing the creation of cycles during the execution of the algorithm. On the other hand, we can detect A-ignorable branches and E-ignorable branches since, by property 6, in any chain of labels we get a copy of a label in finite steps; this means that we can detect if a branch is closed, ignorable or open in a finite number of steps. We can prove that the algorithm terminates and returns a t-complete tableau.

Theorem 22. *Given a finite set of formulae Γ, the algorithm of figure 5.19 terminates and returns a t-completed tableau.*

Proof. *First, we show that the algorithm terminates. The algorithm only introduces formulae in* $\mathsf{ecl}(\Gamma)$ *and, therefore, the set of formulae appearing in the tableau is finite. On the other hand, on any level, we have finite branches (i.e., the tree is finitely generated). So, by König's lemma, if the tree is infinite, we have an infinite branch. In this branch, as explained above, the set of different formulae is finite, and since the algorithm does not apply the P-rule to copies of labels, by theorem 6, any maximal chain of labels $\sigma_1 \prec \sigma_2 \prec \sigma_2 \ldots$ is of finite length. Because we only use a finite set $\gamma_1, ..., \gamma_n$ to build the labels, the number of maximal chains is also finite (otherwise, again by König's lemma, we obtain an infinite chain). That is, the number of different elements in the branch is finite and, therefore, the branch is finite. Summarizing, the algorithm cannot go on indefinitely adding new elements to the tableau. The only way that the algorithm does not stop is that it awakes N-formulae an infinite number of times. But, since only P-formulae can awake these formulae, and the number of P-formulae is finite and we only use each of these formulae once, we only awake N formulae a finite number of times. Since the algorithm eventually marks all the formulae in the tableau, it terminates.*

To see that the generated tableau is t-completed, as explained above, all the labels in a branch are fully reduced, or in the case that the label is a copy, it is reduced. And since we awake N-formulae after applying P-formulae, the third condition in the definition of a t-completed branch is satisfied. ■

Summarizing, the tableaux calculus described above is complete, and we have a procedure which allows us to apply the rules in such a way that we can decide if a given formula is valid or not. It is worth remarking that, following the proof of completeness, we can use this algorithm to build counterexamples in the case of a non-valid formula.

5.5 Open Systems and Temporal Logic

In section 5.2 we have proved that the tableaux calculus for the propositional part of the logic is in some sense complete with respect to language extension. In this section we prove similar results for the temporal part of the logic. The idea is to define sets of propositional formulae which are approximations to the temporal operators, and then, since the logic has the *finite model property*, the satisfiability problem for the temporal operator can be reduced to the satisfiability of a propositional formula. For the following results, we consider only formulae without the Done() operator; at the end of this section, we show that we do not lose expressivity with this restriction.

The finite model property for the temporal logic presented above can be proven in the same way that this property is proven for CTL in [Eme90], i.e., we have:

Theorem 23. *For any temporal formula φ, if φ is satisfiable, then it has a model of size less than or equal to $n * 2^s$, where n is the number of eventuality formulae in φ (i.e., E, $\neg\mathsf{A}$, $\langle\alpha\rangle\varphi$, $\mathsf{P_w}(\alpha)$ or $\neg\mathsf{P}(\alpha)$ formulae) and s is the size of φ. We denote the number $n * 2^s$ by $S(\varphi)$.* ■

Now, we define the *n-th* approximation of the $\mathsf{E}\,\mathcal{U}$ temporal operator as follows:

$$\mathsf{E}^n(\varphi\,\mathcal{U}\,\psi) = \begin{cases} \psi & \text{if } n = 0; \\ \psi \vee (\varphi \wedge \langle \mathbf{U} \rangle \mathsf{E}^{n-1}(\varphi\,\mathcal{U}\,\psi)) & \text{if } n > 0 \end{cases}$$

Intuitively, $\mathsf{E}^n(\varphi\,\mathcal{U}\,\psi)$ says that $\mathsf{E}(\varphi\,\mathcal{U}\,\psi)$ is true at most for n steps. We can define the approximation for $\mathsf{A}\,\mathcal{U}$.

$$\mathsf{A}^n(\varphi\,\mathcal{U}\,\psi) = \begin{cases} \psi & \text{if } n = 0; \\ \psi \vee (\varphi \wedge ([\mathbf{U}]\mathsf{A}^{n-1}(\varphi\,\mathcal{U}\,\psi)) \wedge (\langle \mathbf{U} \rangle \top)) & \text{if } n > 0 \end{cases}$$

In summary, $\mathsf{A}^n(\varphi\,\mathcal{U}\,\psi)$ says that $\mathsf{A}(\varphi\,\mathcal{U}\,\psi)$ is true at most for n steps. Approximations are related to the temporal operators by the following theorem:

Theorem 24. *If M is a model with n states, then we have:*

1. $\pi, i, M \vDash \mathsf{A}(\varphi \,\mathcal{U}\, \psi) \Leftrightarrow \pi, i, M \vDash \mathsf{A}^n(\varphi \,\mathcal{U}\, \psi)$.
2. $\pi, i, M \vDash \mathsf{E}(\varphi \,\mathcal{U}\, \psi) \Leftrightarrow \pi, i, M \vDash \mathsf{E}^n(\varphi \,\mathcal{U}\, \psi)$.

Proof.
1: If $\pi, i, M \vDash \mathsf{A}(\varphi \,\mathcal{U}\, \psi)$, then for every $\pi[0..i] \preceq \pi'$ we have that there exists some k such that $\pi', k, M \vDash \psi$, and for every $i \leq j \leq k$ we have $\pi', j, M \vDash \varphi$. For all these paths, take the minimum k which satisfies this condition. Note that it is not possible for any of these π''s to have a cycle between positions i and k (i.e., some state appearing twice), otherwise (since k is a minimum) we have that during this cycle all the positions satisfy $\neg\psi$, and therefore we can make a new sequence (repeating the cycle an infinite number of times) which never makes ψ true, implying that $\pi, i, M \nvDash \mathsf{A}(\varphi \,\mathcal{U}\, \psi)$. Since no state appears twice in these π''s, we have that $k \leq n + i$ for all the sequences π''s, and this implies that $\pi, i, M \vDash \mathsf{A}^n(\varphi \,\mathcal{U}\, \psi)$. The other direction is straighforward by definition of the approximation.
2: If $\pi, i, M \vDash \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$, then there exist some $\pi[0..i] \preceq \pi'$ such that for some $i \leq k$ we have $\pi', k, M \vDash \psi$. Take the sequence π' which satisfies the condition above, and such that this k is the minimum (with respect to the k's associated with the sequences which satisfy the condition); in this sequence we cannot have a cycle, otherwise we can create a sequence π'' deleting the cycle where in position $k' < k$ we have $\pi'', k', M \vDash \psi$, which is a contradiction since the k was the minimum position in a sequence satisfying the condition. Since we do not have cycles in π' for the positions $k \leq n$, and therefore by definition of approximations, we have $\pi, i, M \vDash \mathsf{E}^n(\varphi \,\mathcal{U}\, \psi)$. ■

A direct corollary of the theorem stated above is:

Corollary 9. *If M is a model with n states, then we have:*

1. $\pi, i, M \vDash \neg\mathsf{A}(\varphi \,\mathcal{U}\, \psi) \Leftrightarrow \pi, i, M \vDash \neg\mathsf{A}^n(\varphi \,\mathcal{U}\, \psi)$.
2. $\pi, i, M \vDash \neg\mathsf{E}(\varphi \,\mathcal{U}\, \psi) \Leftrightarrow \pi, i, M \vDash \neg\mathsf{E}^n(\varphi \,\mathcal{U}\, \psi)$.

■

We can define approximations to any other formula. Given a temporal formula φ, an approximation to it is obtained by replacing each temporal operator in φ by an approximation to it. For example, given the formula $\mathsf{A}(\varphi \,\mathcal{U}\, \psi) \wedge \mathsf{E}(\varphi' \,\mathcal{U}\, \psi')$ an approximation of it is $\mathsf{A}^n(\varphi \,\mathcal{U}\, \psi) \wedge \mathsf{E}^m(\varphi' \,\mathcal{U}\, \psi')$, where n and m are natural numbers. However, this notation (having the numbers as superscripts) is not very useful for our purposes; instead we use a postfix notation, putting the numbers in a depth-first order, e.g., instead of the formula above we write $\mathsf{A}(\varphi \,\mathcal{U}\, \psi) \wedge \mathsf{E}(\varphi' \,\mathcal{U}\, \psi')[n, m]$. Using this notation we can state the following theorem.

Theorem 25. *Let φ be a temporal formula, and M a model with n states, then:*

$$\pi, i, M \models \varphi \Leftrightarrow \pi, i, M \models \varphi[n, \ldots, n]$$

where $\varphi[n, \ldots, n]$ is an approximation to φ. ■

The proof is a simple argument using structural induction on φ and theorem 24. Using this property we can prove that each temporal formula has a bound over the number of atomic actions that we need to consider for checking its satisfiability.

Theorem 26. *Given a temporal formula φ, if φ is satisfiable in a model over a vocabulary $V = \langle \Delta_0, \Phi_0 \rangle$, then φ is satisfiable over a language $V' = \langle \mathrm{Pr}(\varphi') \cup \{a_1, ..., a_n\}, \Phi_0 \rangle$, where φ' is a DNF formula equivalent to the approximation $\varphi[k, ..., k]$ to φ for $k = S(\varphi)$, and $n \leq \mathsf{D}_\exists(\varphi')$.*

Proof. *Given a temporal formula φ we know by theorem 23 that there exists a finite model M with $\mathsf{S}(\varphi) = k$ states such that $\pi, i, M \models \varphi$, but this is equivalent to saying that $\pi, i, M \models \varphi[k, ..., k]$, and there exists a DNF formula φ' equivalent to $\varphi[k, ..., k]$, and therefore $\pi, i, M \models \varphi'$. By corollary 8 we know that there exists a model M' over a vocabulary $V = \langle Pr(\varphi') \cup \{a_1, ..., a_n\}, \Phi_0 \rangle$, where $n \leq \mathsf{D}_\exists(\varphi')$, and therefore by theorem 25 we have that $\pi', i', M' \models \varphi$.* ■

In other words, to check a temporal formula, we can use a propositional approximation to it to calculate how many extra actions we need to get a model, if it exists.

The properties above can only be used with formulae without the Done() operator; however, we can show that no expressivity is lost with this restriction. First, consider the following property.

Theorem 27. *Let p be any propositional symbol; then:*

$$\models_A \neg p \wedge \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p) \leftrightarrow \mathsf{AG}(p \leftrightarrow \mathsf{Done}(\alpha)).$$

Proof. *We prove the $\rightarrow$ part. Suppose that $\pi, 0, M \models \neg p \wedge \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p)$, i.e., we have $\pi, 0, M \models \neg p$ and*

$$\pi, 0, M \models \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p) \tag{5.6}$$

On the other hand, by definition of $\models$ we have $\pi, 0, M \models \neg\mathsf{Done}(\alpha)$ and $\pi, 0, M \models \neg p$. Now, for any $\pi[0..0] \preceq \pi'$ such that $\pi', i, M \models \mathsf{Done}(\alpha)$, we know that $i > 0$ and $\pi_{i-1} \xrightarrow{e_i} \pi_i \xrightarrow{e_{i+1}} \cdots = \pi^{i-1}$, and $e_i \in \mathcal{I}(\alpha)$. But then, by 5.6, we get $\pi', i, M \models p$ and then $\pi', i, M \models p \leftrightarrow \mathsf{Done}(\alpha)$. And, since this is for every i and π' (including $i = 0$), we get $\pi, 0, M \models \mathsf{AG}(p \leftrightarrow \mathsf{Done}(p))$. The other direction is similar. ■

Now we prove that the formula in the antecedent of the implication in the above theorem does not destroy models, i.e.,

Theorem 28. *If p does not appear in φ, then $\pi, 0, M \models \varphi$ for a model M and a path π if and only if there exists some model M' such that*

$$\pi', 0, M' \models (\neg p \wedge \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p)) \wedge \varphi$$

for a path π'.

Proof. *One direction is trivial. For the other one suppose that $\pi, 0, M \models \varphi$ and $M = \langle \mathcal{W}, \mathcal{E}, \mathcal{I}, \mathcal{P}, \mathcal{R}, w \rangle$. For each state v in $\mathcal{W}$, we can define a set of sequences of elements of $\mathcal{E}$ denoted by $P(v)$, as follows:*

$$P(v) = \{e_1 \centerdot \cdots \centerdot e_n \mid w \xrightarrow{e_1} \ldots \xrightarrow{e_n} v\}$$

and $P(w) = \{\langle\rangle\}$ (the empty string). That is, for each state we define a collection of paths which yield it. Since $\mathcal{R}$ is deterministic with respect to events, we have that $P(v) \cap P(v') = \emptyset$, when $v \neq v'$.

We define a model $M' = \langle \mathcal{W}', \mathcal{E}', \mathcal{I}', \mathcal{P}', \mathcal{R}' \rangle$ as follows:

- $\mathcal{M}' = \bigcup_{v \in \mathcal{W}} P(v)$.
- $\mathcal{E}' = \mathcal{E}$.
- $\mathcal{R}' = \{s \xrightarrow{e} s' \mid s, s' \in \mathcal{W}' \wedge s' = s \cdot e\}$.
- $\mathcal{P}' = \{(s, e) \mid$ *if* $s \in P(v) \wedge (v, e) \in \mathcal{P}\}$.
- $\mathcal{I}'(a_i) = \mathcal{I}(a_i)$, *for all* $a_i \in \Delta_0$.
- $\mathcal{I}'(p_i) = \mathcal{I}(p_i)$, *for every* $p_i \neq p$.
- $\mathcal{I}'(p) = \{s \in \mathcal{W}' \mid \exists e \in \mathcal{E} : s = s' \centerdot e \wedge e \in \mathcal{I}'(\alpha)\}$.

It is easy to see that we have $\pi, 0, M' \models \neg p \wedge \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p)$ for any π. Note that, for every path $w \xrightarrow{e_1} w_1 \xrightarrow{e_2} w_2 \xrightarrow{e_3} \ldots$, we have a corresponding path π' in M with $\pi' = \langle\rangle \xrightarrow{e_1} e_1 \xrightarrow{e_2} e_1 \cdot e_2 \xrightarrow{e_3} \ldots$, such that for every i, if $\pi, i, M \models \psi$, then $\pi', i, M' \models \psi$, which can be proven by an easy structural induction, and therefore we have $\pi', 0, M' \models \varphi$. ■

Now, consider the CTL property:

$$\models_A \mathsf{AG}(\psi' \leftrightarrow \psi) \rightarrow (\varphi \leftrightarrow \varphi[\psi/\psi'])$$

Where $\varphi[\psi/\psi']$ is the formula obtained by replacing all the occurrences of ψ in φ by occurrences of ψ'. Putting the pieces together, if we have a formula φ with some occurrences of $\mathsf{Done}(\alpha)$, then we can check the formula:

$$\neg p \wedge \mathsf{AG}([\alpha]p \wedge [\overline{\alpha}]\neg p) \wedge \neg\varphi[\mathsf{Done}(\alpha)/p]$$

If we get a model of this formula by the tableaux method, then we have a countermodel of $\varphi[\mathsf{Done}(\alpha)/p]$, and then, using the property stated above, and theorem 27, we get a counterexample for φ. Now, if we do not get a model satisfying the formula above, theorem 28 implies that $\neg\varphi[\mathsf{Done}(\alpha)/p]$ does not have any model, and therefore, again by theorem 27, we obtain that $\neg\varphi$ does not have any model which satisfies it, and therefore φ is valid.

Summarizing, instead of checking formulae with Done() predicates we can check formulae without these predicates and we can use modalities, temporal operators and propositional variables to express them.

In this section we have shown that, given a temporal formula (or a finite set of temporal formulae), we have vocabulary in which it can be checked for global validity. This can be thought of as saying that there is a bound on the number of environment actions that may falsify the given formula. This can be useful when checking *open systems* (i.e., systems that interact with an environment). However, we have to take into account that usually we will get counterexamples, which show scenarios where our formula is false. In these cases, we have to make the formula stronger by adding further assumptions, for example, using a *rely-guarantee* discipline (introduced in [CM81]), assuming some good behaviour of the environment. It is important to note that, although we can check global validities, we have to be careful when we put components together. For example, we can prove using theorem 8 that a given specification has a liveness property, but, when we put together components, the obtained specification has more restrictions and therefore it could be the case that we make the original component specification inconsistent. Roughly speaking, the liveness property is a theorem derived from the specification of the component, but the specification of the system including the component has no model, since perhaps the other components add further restrictions on the possible execution of the component as part of the system. Thus, the liveness property of the component specification is preserved in the system, but as part of an inconsistent specification. Therefore, consistency checking must also be done when we obtain specifications from smaller ones (and this can also be done using the tableaux system).

5.6 A Final Example

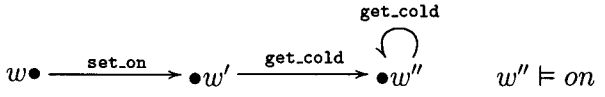
In this section we present a final example in which we use the temporal extension of the tableaux. We take again the example of the heating system outlined in section 5.3. Now, we add the new actions `set_on` and `set_off` in our model, and we try to prove the property:

$$[\texttt{set\_on}]\mathsf{AG}(\langle\texttt{set\_on}\rangle\bot) \rightarrow [\texttt{set\_on}]\mathsf{EF}\neg\texttt{on}$$

where we use the equivalences $\mathsf{AG}\varphi \equiv \neg\mathsf{E}(\top\ \mathcal{U}\ \neg\varphi)$ and $\mathsf{EF}\varphi \equiv \mathsf{E}(\top\ \mathcal{U}\ \varphi)$. This formula says that, *if after setting the heating on, we do not set it on again, then eventually the system will be turned off.* This happens because the sensor should detect that the room is getting warm and it will stop the heating system. However, if there is some fault in the sensor, it will never detect that the room is warming up; as a consequence the room will start to be uninhabitable. Let us use the tableaux system to try to find a countermodel. For the sake of simplicity, we only sketch the tree in figure 5.20. We use the following labels in the tree:

$\gamma_1 = \texttt{set\_on} \sqcap \overline{\texttt{set\_off}} \sqcap \overline{\texttt{get\_cold}} \sqcap \overline{\texttt{get\_warm}}$
$\gamma_2 = \gamma_1 \cdot \gamma_1$
$\gamma_3 = \gamma_1 \cdot \texttt{get\_cold} \sqcap \overline{\texttt{set\_on}} \sqcap \overline{\texttt{set\_off}} \sqcap \overline{\texttt{get\_warm}}$
$\gamma_4 = \gamma_3 \cdot \texttt{get\_cold} \sqcap \overline{\texttt{set\_on}} \sqcap \overline{\texttt{set\_off}} \sqcap \overline{\texttt{get\_warm}}$

Note that, in the illustrated tableau, the last node is a copy of its parent. Because this is not an ignorable branch, the branch is open. The root of the tableau contains the formulae in section 5.3 together with the negation of the property to be refuted, and the formula $\mathsf{AG}\langle\mathbf{U}\rangle\top$, which guarantees that we always have a next state. To obtain this tableau we first apply the P rule to the last formula of the first node, and then we apply the $\neg\mathsf{E}\,\mathcal{U}$ rule several times, together with N rules to inherit the formulae in the upper node; after that we obtain the branches using the P rule over the formula $\langle\mathbf{U}\rangle\top$. Because of space restrictions, we have not included all the formulae in each node, only the relevant formulae are shown. From the branch obtained we can obtain the following model:



Intuitively, we break the property when we press the "on" button, and after that the sensor does not work and it does not detect that the room is getting hot. This is a

classic faulty scenario; we can add some formulae to our specification to prevent this sort of situation from arising. Consider the following set of formulae:

$\{\mathsf{AG}(\mathsf{Done}(\mathtt{set\_on}) \rightarrow (\neg \mathsf{P}_{\mathsf{w}}(\mathtt{get\_cold})\, \mathcal{U}\, \mathsf{Done}(\mathtt{set\_off} \sqcup \mathtt{get\_warm}))),$
$\mathsf{AG}((\langle \mathtt{get\_cold} \rangle \top) \rightarrow \mathsf{P}(\mathtt{get\_cold}))\}$

The first one says that, after setting the heating system to on, it is forbidden for the sensor to detect that the room is cold if it has not detected earlier that the room was warm (or, alternatively, somebody pressed the "off" button). The second formula says that the sensor is working only in those scenarios where it is allowed to be enabled, and then we try to observe what happens with our property. As the reader can check, the open branch shown above is now closed, because either we get that *get_cold* is an impossible action (it is equal to $\emptyset$) in this branch (it is both forbidden and strongly allowed) or the sensor is not enabled since this situation is forbidden.

5.7 Conclusions and Further Work

In this chapter we have described a tableaux system for the deontic action logic introduced in chapter 3. One of the main features of the system is that it uses the underlying algebra of actions to produce tableaux, enabling it to manage successfully the intersection and complement operator on actions, two operators which are normally hard to deal with. Moreover, the algebra of actions allows us to extend the propositional tableaux system to manage temporal predicates. The rules presented for the temporal part of the logic are simple and they reflect the basic properties of the logic, see chapter 3. Moreover, we show that the predicate Done() can be added straightforwardly to the logic and, as we proved above, it is possible to express the Done() operator with the temporal operators and the modal predicates.

A relevant point demonstrated in this chapter is that, though we have a finite number of actions, it is possible to prove properties which are valid in any extension of the actual vocabulary, which seems to be very useful in practice to verify components which could be a part of bigger modules. This kind of completeness with respect to language extension is also preserved for the temporal version of the logic.

It seems very useful to apply the tableaux system described here to specifications. We are pursuing the use of this system to analyze, in an automatic way, the properties of fault-tolerant systems. Here, we do not provide more complicated examples, but, if some software tool is provided, we should be able to prove properties, and to find counterexamples, in an automatic way over complicated and interesting case studies.

Among the benefits of the system proposed here is that it provides, in the case of open tableaux, complete models with labels in the transitions which describe the actions executed (or not) by the system. This allows us to get counterexamples which exhibit an incorrect behaviour of the system or its environment. Analyzing these counterexamples can help to improve the specification and therefore the software produced from it.

As further work, we want to extend the tableaux described here to be able to support some sort of modularization. It seems possible to use concepts coming from category theory to do this (for example, some concept analogous to *Institutions*, introduced by Goguen and Burstall in [GB92]).

It seems that the technique presented here (to use the underlying algebra of actions to guide the tableaux rules) can be used with other algebras, perhaps more expressive than boolean algebras. One of the possible candidates is residuated boolean algebras (see [Jip92]), for which there are also some algebras with complete equational systems and similar properties to boolean algebras. One of the main benefits of using these algebras is the possibility of incorporating the iteration operator over actions into the formalism.

Initial Step: Put the initial formulae in some order and mark them as awake.
Step n+1 ($n \geq 0$)**:** Choose some labelled formula $\sigma : \varphi$ marked awake as near to the root as possible, belonging to a branch which is not ignorable or closed. If there are several such formulae, choose the leftmost one. After that, proceed as follows:

- If φ is an atomic proposition, then mark the formula as finished and finish the step.
- If φ is an A formula, then add the corresponding A_1 and A_2 formulae to all the branches passing trough $\sigma : \varphi$, and mark the formula as finished.
- If φ is a B formula, then for any branch passing through $\sigma : \varphi$, add two new branches, one containing B_1 and the other containing B_2, and mark the formula as finished.
- If φ is a P formula, then, if σ is a copy of another σ' in the branch, then mark the formula as finished. Otherwise, for any branch passing trough $\sigma : \varphi$, add all the branches resulting from applying the P-rule and mark the formula as finished. If there is some N formula with label σ then mark it as awake.
- If φ is a P_D formula, for any branch passing through $\sigma : \varphi$, add all the branches resulting from applying the P_D-rule and mark the formula as finished.
- If φ is a NA, $\neg$NA, A $\mathcal{U}$, $\neg$A $\mathcal{U}$, E $\mathcal{U}$ or $\neg$E $\mathcal{U}$ formula, then apply the corresponding rule and add all the resulting branches to all the branches passing through $\sigma : \varphi$, and then mark $\sigma : \varphi$ as finished.
- If φ is an N_D formula, then apply the N_D-rule and add all the resulting formulae to all the branches passing through $\sigma : \varphi$, and mark this formula as finished.
- If φ is an N formula, then apply the N-rule and add all the resulting formulae to the corresponding branches, and mark the formula as asleep.

End of Step n+1: If all the branches in the tableau are closed or ignorable or all the formulae in the tableau are marked as finished or as asleep, then finish. Otherwise start the next step.

Figure 5.19: Algorithm for applying the tableaux calculus

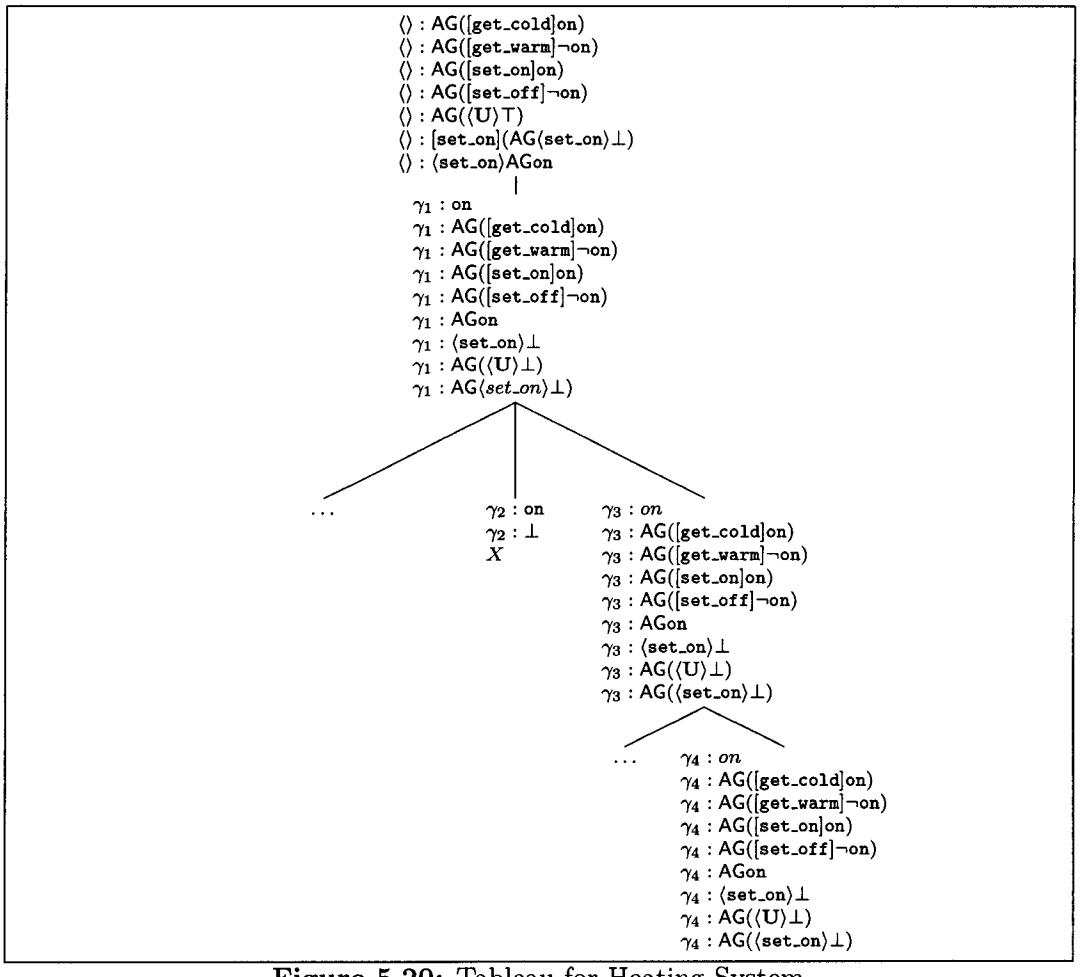


Figure 5.20: Tableau for Heating System

Chapter 6

Relating Tableaux with the Hilbert System

In chapters 3 and 5 we have presented two different deductive systems for the same logic; we proved the completeness and soundness of the propositional part of the logic for these two systems. However, the completeness of the Hilbert system for the temporal extension of the logic was not investigated in chapter 3. Here we use the tableaux system to prove the completeness of the Hilbert system presented in chapter 3. This result implies that we can combine both systems when verifying software, obtaining, on the one hand the benefit of automatic proving and the possibility of getting counterexamples in a case of an unsuccessful proof (using tableaux), and, on the other hand, we have a standard deductive system where the well-known properties of propositional and modal logic can be used. In addition, we show how the labelled and not unlabelled formulae can be related using the atoms of the boolean algebra of terms, which can be a useful technique when proving properties of labelled systems.

6.1 A Proof of the Hilbert-system Completeness

As explained above, we have exhibited two deductive systems. On the one hand, we have described a Hilbert-style system and, in the last chapter, we presented a tableaux deductive system. We shall prove that both systems agree in their theorems. First, it is important to note that in tableaux we use labelled formulae of the type: $\sigma : \varphi$. We define a translation of these formula to standard formulae (without labels), as follows:

- $\mathbf{H}(\langle\rangle : \varphi) = \varphi$.
- $\mathbf{H}(\sigma \centerdot \gamma : \varphi) = \mathbf{H}(\sigma : \langle\gamma\rangle\varphi)$.

If S is a set of labelled formulae, then we denote by $\mathbf{H}(S)$ the application of $\mathbf{H}()$ to each member of S. Note that this function translates a label (consisting of a sequence of atomic boolean terms) to a sequence of modalities whose actions are the corresponding atomic boolean terms.

One important thing to note is that in the tableaux system we consider that degenerate boolean algebras (i.e., when $\mathbf{U} =_{act} \emptyset$) are not possible models of actions (i.e., they are inconsistent). In the Hilbert system we have not ruled out degenerate boolean algebra in our models, though it is easy to do this with the additional axiom:

A19. $\mathbf{U} \neq_{act} \emptyset$.

and stating in the semantics that we only consider non-degenerate boolean algebras. We can also take another approach and modify the tableaux system; however, we think that for specification and verification of systems, degenerate boolean algebras are not needed and can be safely discarded.

In this chapter, it is convenient to distinguish between the tableaux deduction and the Hilbert deduction. We say that $\vdash_{\mathbf{T}} \varphi$ when we have a tableaux built from $\langle\rangle : \neg\varphi$ that is closed, and we say that $\vdash_{\mathbf{H}} \varphi$ when we can prove φ using the axiomatic system presented in chapter 3. (Here we abstract from the language, and we suppose that we are using a fixed arbitrary language L.) We also distinguish between the validity used for tableaux (which is anchored with respect to the beginning of time) and the validity that we use for the Hilbert system; the anchored validity is denoted by $\models_A$, and the non anchored validity by $\models$.

Our first theorem says that every theorem proved with the Hilbert system is provable from tableaux (which is a corollary of the soundness of the Hilbert system).

Theorem 29. *If* $\vdash_{\mathbf{H}} \varphi \Rightarrow \vdash_{\mathbf{T}} \varphi$.
Proof. *Suppose* $\vdash_{\mathbf{H}} \varphi$, *then* $\pi, i, M \models \varphi$ *for every structure* M, *trace* π *and position* i; *but then* $\pi, 0, M \models \varphi$ *for every structure* M *and trace* π. *But this means that* $\models_A \varphi$, *and since the tableaux system is complete we have* $\vdash_{\mathbf{T}} \varphi$. ■

The hard part is to prove the other direction ($\vdash_{\mathbf{T}} \varphi \Rightarrow \vdash_{\mathbf{H}} \varphi$), which implies that the Hilbert system is complete because of the completeness of the tableaux system. To prove this we need a number of properties. In the following, given a finite set of formulae S, we denote by $\bigwedge S$ the conjunction of all its elements.

Theorem 30. *Given a t-completed finite tableau $\mathcal{T}$ and a closed branch $\mathcal{B}$ of $\mathcal{T}$, we have that* $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathbf{H}(\mathcal{B}) \rightarrow \bot)$.
Proof. *We analyze each possibility for which $\mathcal{B}$ is closed.*

- *If $\sigma : p$ and $\sigma : \neg p$ belong to $\mathcal{B}$ for some p, then by PL we have:* $\vdash_{\mathbf{H}} p \wedge \neg p \leftrightarrow \bot$. *Now, using the definition of* $\mathbf{H}()$, *we have:*

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle p$$

for some sequence of atoms $\gamma_1, \ldots, \gamma_n$ in the boolean algebra of terms. (Note that the function $\mathbf{H}()$ *translates labels to (probably empty) sequence of action terms.) In the same way, we obtain:*

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle \neg p.$$

Now, since $\gamma_1, \ldots, \gamma_n$ are atoms in the boolean algebra of terms, we can use axiom ***A17*** *together with properties of boolean algebra and obtain:*

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow [\gamma_1] \ldots [\gamma_n] \neg p$$

and now by axiom ***A2*** *and PL we obtain:*

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle \neg p \wedge p$$

which using PL gives us:

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle \bot$$

and from here using the property $\langle\alpha\rangle\bot \leftrightarrow \bot$ and PL several times we obtain:

$$\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(B) \rightarrow \bot.$$

- *If* $EQ(\mathcal{B}) \vdash_{BA} \mathbf{U} =_{act} \emptyset$, *taking into account axiom* ***A19***, *we have that, using the axioms of boolean algebra and propositional logic,* $\vdash_{\mathbf{H}} \wedge\mathbf{H}(\mathcal{B}) \rightarrow \mathbf{U} =_{act} \emptyset$. *But from here and using the new axiom we get* $\vdash_{\mathbf{H}} \bigwedge \mathbf{H}(\mathcal{B}) \rightarrow \bot$, *and then* $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathbf{H}(\mathcal{B}) \rightarrow \bot)$.

- *If $\mathcal{B}$ is done-closed, then we have the following possibilities:*

  - *If $\langle\rangle : \mathsf{Done}(\alpha) \in \mathcal{B}$, then by axiom* ***tempAx9*** *and PL we have:*

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathsf{Done}(\alpha) \rightarrow \bot).$$

– *Suppose that* $\sigma \bullet \gamma : \mathsf{Done}(\alpha) \in \mathcal{B}$ *with* $\nvdash_{BA} \gamma \sqsubseteq \alpha$. *By definition of* $\mathbf{H}()$, *we have that*

$$\mathbf{H}(\sigma \bullet \gamma : \mathsf{Done}(\alpha)) = \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\langle\gamma\rangle\mathsf{Done}(\alpha).$$

Since $\nvdash_{BA} \gamma \sqsubseteq \alpha$, *by properties of boolean algebras and taking into account that* γ *is an atom in the action term algebra we obtain:* $\vdash_{BA} \gamma \sqsubseteq \overline{\alpha}$. *Now, using axiom* ***TempAx7****, PL and* ***T3*** *we have:* $\vdash_{\mathbf{H}} [\gamma]\neg\mathsf{Done}(\alpha)$, *and from here, using* ***GN****, we get*

$$\vdash_{\mathbf{H}} [\gamma_1] \ldots [\gamma_n][\gamma]\neg\mathsf{Done}(\alpha).$$

Using this, PL and axiom ***A2*** *we get:*

$$\vdash_{\mathbf{H}} \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\langle\gamma\rangle\mathsf{Done}(\alpha) \rightarrow \bot$$

which implies

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\langle\gamma\rangle\mathsf{Done}(\alpha) \rightarrow \bot)$$

which by definition of $\mathbf{H}()$ *gives us:*

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathbf{H}(\sigma \bullet \gamma : \mathsf{Done}(\alpha)) \rightarrow \bot).$$

- *If it is deontic closed, then we have three possibilities:*
  - $\sigma : \mathsf{P}(\gamma)$ *and* $\sigma : \neg\mathsf{P}(\gamma)$.
  - $\sigma : \mathsf{P}_{\mathsf{w}}(\gamma)$ *and* $\sigma : \neg\mathsf{P}_{\mathsf{w}}(\gamma)$.
  - $\sigma : \neg\mathsf{P}(\gamma)$ *and* $\sigma : \mathsf{P}_{\mathsf{w}}(\gamma)$.

The proof for the first two cases is similar to the proof for the propositional variables. For the last case, the proof is as follows. By axiom ***A12*** *we have* $\vdash_{\mathbf{H}} \mathsf{P}_{\mathsf{w}}(\gamma) \rightarrow \mathsf{P}(\gamma)$ *and then, using PL, we have:* $\vdash_{\mathbf{H}} \neg\mathsf{P}(\gamma) \wedge \mathsf{P}_{\mathsf{w}}(\gamma) \rightarrow \bot$, *and using the property* $\langle\alpha\rangle\bot \leftrightarrow \bot$ *we get* $\vdash_{\mathbf{H}} (\langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\neg\mathsf{P}(\gamma) \wedge \mathsf{P}_{\mathsf{w}}(\gamma)) \rightarrow \bot$ *and we know that:*

$$\mathbf{H}(\sigma : \mathsf{P}(\gamma)) = \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\mathsf{P}(\gamma)$$

(for some sequence of atoms $\gamma_1, \ldots, \gamma_n$*), and similarly for* $\mathbf{H}(\sigma : \neg\mathsf{P}_{\mathsf{w}}(\gamma))$. *Note that because of axiom* ***A17****, we have:*

$$\vdash_{\mathbf{H}} \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\mathsf{P}(\gamma) \rightarrow [\gamma_1] \ldots [\gamma_n]\mathsf{P}(\gamma)$$

and by axiom ***A2*** *we have:*

$$\vdash_{\mathbf{H}} [\gamma_1] \ldots [\gamma_n]\mathsf{P}(\alpha) \wedge \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\neg\mathsf{P}_{\mathsf{w}}(\alpha) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle\mathsf{P}(\gamma) \wedge \neg\mathsf{P}_{\mathsf{w}}(\gamma)$$

and then by PL we get:

$$\vdash_{\mathbf{H}} \mathbf{H}(\sigma : \mathsf{P}(\gamma)) \wedge \mathbf{H}(\sigma : \mathsf{P}_{\mathsf{w}}(\gamma)) \rightarrow \bot$$

and therefore: $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathbf{H}(\mathcal{B}) \rightarrow \bot)$.

■

We can prove a variation of the same theorem.

Theorem 31. *Given a t-completed finite tableau $\mathcal{T}$ and a closed branch $\mathcal{B}$ of $\mathcal{T}$, then we have some label σ in $\mathcal{B}$ such that $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathcal{B}/\sigma \rightarrow \bot)$.*
Proof. *Similar to theorem 30.* ■

Note that, in the theorem above, if we restrict our attention only to branches which do not contain a formula $\langle\rangle : \mathsf{Done}(\alpha)$, we have the following theorem:

Theorem 32. *Given a t-completed finite tableau $\mathcal{T}$ and a closed branch $\mathcal{B}$ of $\mathcal{T}$ which do not contain a formulae $\langle\rangle : \mathsf{Done}(\alpha)$, then we have that $\vdash_{\mathbf{H}} \mathcal{B}/\sigma \rightarrow \bot$.*
Proof. *The cases are the same as above taking into account that we do not have the case of a branch wihch is done-closed because of a formula $\langle\rangle : \mathsf{Done}(\alpha)$.* ■

Now, we investigate the nature of tableaux rules; the following theorem says that the rules of the tableaux system only add equivalent formulae to the ones already in the tree.

Theorem 33. *For every rule:*

$$\frac{\varphi}{\begin{array}{cc} \psi_1^1 & \psi_1^n \\ \vdots & \vdots \\ \psi_1^n & \psi_m^n \end{array}}$$

we have that $\vdash_{\mathbf{H}} \mathbf{H}(\varphi) \leftrightarrow \bigvee_{1\leq j\leq n} \bigwedge_{1\leq i\leq m} \mathbf{H}(\psi_i^j)$.
Proof. *For the propositional rules, the proof is straightforward. For the remaining rules, the proof is as follows.*
Rule N_D: From the axioms we can prove $\vdash_{\mathbf{H}} \mathsf{P}(\alpha) \rightarrow \bigwedge \mathsf{P}(\gamma_i)$ where $EQ(\mathcal{B}) \vdash_{BA} \gamma_i \sqsubseteq \alpha$ and γ_i are atoms. Then, using modal logic properties, we get: $\vdash_{\mathbf{H}} \langle\gamma_1'\rangle \ldots \langle\gamma_n'\rangle \mathsf{P}(\alpha) \rightarrow \langle\gamma_1'\rangle \ldots \langle\gamma_n'\rangle \bigwedge \mathsf{P}(\gamma_i)$, where $\langle\gamma_1'\rangle \ldots \langle\gamma_n'\rangle$ is the sequence of modalities obtained from $\sigma : \mathsf{P}(\alpha)$ using the translation $\mathbf{H}()$; therefore: $\vdash_{\mathbf{H}} \mathbf{H}(\sigma : \mathsf{P}(\alpha)) \rightarrow \bigwedge \mathbf{H}(\sigma : \mathsf{P}(\gamma_i))$.
Rule N: Similar to rule N_D.
Rule P_D: It is straightforward to prove: $\vdash_{\mathbf{H}} \mathsf{P_w}(\alpha) \rightarrow \bigvee \mathsf{P_w}(\gamma_i)$ where $\gamma_i \sqsubseteq \alpha$, and now using modal logic we get:

$$\vdash_{\mathbf{H}} \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle \mathsf{P_w}(\alpha) \rightarrow \langle\gamma_1\rangle \ldots \langle\gamma_n\rangle \bigvee \mathsf{P_w}(\gamma_i)$$

and this proves the result.
Rule Per: This follows from axiom ***A12*** *and modal logic.*
Rule N*: By PL we have:*

$$\vdash_{\mathbf{H}} \langle \mathbf{U} \rangle \top \wedge [\mathbf{U}]\varphi \rightarrow \mathsf{AN}\varphi$$

and, by axiom ***A1*** *and PL, we have:*

$$\vdash_{\mathbf{H}} \neg\langle \mathbf{U} \rangle \top \wedge \varphi \rightarrow \mathsf{AN}\varphi$$

and therefore, using PL, we get:

$$\vdash_{\mathbf{H}} (\langle \mathbf{U} \rangle \top \wedge [\mathbf{U}]\varphi) \vee (\neg\langle \mathbf{U} \rangle \top \wedge \varphi) \rightarrow \mathsf{AN}\varphi.$$

On the other hand, using propositional logic we have:

$$\vdash_{\mathbf{H}} \mathsf{AN}\varphi \rightarrow \langle \mathbf{U} \rangle \top \vee \neg\langle \mathbf{U} \rangle \top$$

and then, using PL, the property $\langle \alpha \rangle \bot \leftrightarrow \bot$ *and* ***A1****, we get:*

$$\vdash_{\mathbf{H}} \mathsf{AN}\varphi \rightarrow (\langle \mathbf{U} \rangle \top \wedge [\mathbf{U}]\varphi) \vee (\neg\langle \mathbf{U} \rangle \top \wedge \varphi)$$

which proves the theorem.
Rule $\neg$N*: This can be deduced from the rule above using PL.*
Rule $\mathsf{E}\,\mathcal{U}$ *: By axiom* ***TempAx5****, we have*

$$\vdash_{\mathbf{H}} \mathsf{A}(\varphi \,\mathcal{U}\, \psi) \leftrightarrow \psi \vee (\neg\psi \wedge \varphi \wedge \neg\mathsf{AN}\neg\mathsf{E}(\varphi \,\mathcal{U}\, \psi)).$$

Rule $\neg\mathsf{E}\,\mathcal{U}$ *: Follows from the rule above using PL.*
Rule $\mathsf{A}\,\mathcal{U}$ *: By axiom* ***TempAx4****, we have:*

$$\vdash_{\mathbf{H}} \mathsf{A}(\varphi \,\mathcal{U}\, \psi) \leftrightarrow (\psi \vee (\neg\psi \wedge \varphi \wedge \mathsf{AN}(\varphi \,\mathcal{U}\, \psi)))$$

and using modal logic we get the result.
Rule $\neg$AN*: This follows from the rule above using PL.* ■

From the theorem above, we can prove that the labels that we can obtain by the P rule are implied by formulae already in the actual branch.

Theorem 34. *If* $\mathcal{B}$ *is a branch of a finite t-completed tableau* $\mathcal{T}$ *which contains formulae with labels* σ *and* $\sigma \centerdot \gamma$*, then we have that*

$$\vdash_{\mathbf{H}} \bigwedge \mathcal{B}/\sigma \rightarrow \bigwedge \{\langle \gamma \rangle \varphi \mid \varphi \in \mathcal{B}/(\sigma \centerdot \gamma)\}.$$

Proof. *Let* $\sigma : \varphi_1, \ldots, \sigma : \varphi_n$ *be all the formulae in* $\mathcal{B}$ *labelled with label* σ *and* $\sigma \centerdot \gamma : \psi_1, \ldots, \sigma \centerdot \gamma : \psi_m$ *all the formulae in* $\mathcal{B}$ *labelled with label* $\sigma \centerdot \gamma$*. Since any*

formula $\sigma \centerdot \gamma : \psi_i$ *can only be obtained by using the rule* P *from a formula* $\sigma : \varphi_j$ *in* $\mathcal{B}$*, we have, by theorem 33, that for every* $\sigma \centerdot \gamma : \psi_i$ *there exists a* $\sigma : \varphi_j$ *such that:*

$$\vdash_{\mathbf{H}} \mathbf{H}(\sigma : \varphi_j) \rightarrow \mathbf{H}(\sigma \centerdot \gamma : \psi_i).$$

But then note that we can apply rule P *with* $\sigma = \langle\rangle$*, in which case we obtain:*

$$\vdash_{\mathbf{H}} \varphi_j \rightarrow \mathbf{H}(\gamma : \psi_i)$$

which, by definition of $\mathbf{H}()$ *gives us:*

$$\vdash_{\mathbf{H}} \varphi_j \rightarrow \langle\gamma\rangle\psi_i$$

and, since this is for every ψ_j*, we obtain:*

$$\vdash_{\mathbf{H}} \bigwedge_{j \leq n} \varphi_j \rightarrow \bigwedge_{i \leq m} \langle\gamma\rangle\psi_i$$

The theorem follows. ■

We have proven that closed branches are inconsistent. However, we can have ignorable branches. The following theorem proves that ignorable branches are inconsistent. Here, given a tableaux $\mathcal{T}$, we say that $\mathcal{S}$ is a subbranch of $\mathcal{T}$ if $\mathcal{S}$ is a subpath of a branch $\mathcal{B}$ of $\mathcal{T}$.

Theorem 35. *Let* $\mathcal{T}$ *be a t-complete tableau, and* $\mathcal{S}$ *a subbranch of* $\mathcal{T}$ *with a formula* $\sigma : \mathsf{E}(\varphi \ \mathcal{U} \ \psi) \in \mathcal{S}$ *and such that every* $\sigma' \leq \sigma$ *is reduced in* $\mathcal{S}$*. If every branch* $\mathcal{B}$ *of* $\mathcal{T}$ *with* $\mathcal{S} \subseteq \mathcal{B}$ *is either* E*-ignorable for* $\sigma : \mathsf{E}(\varphi \ \mathcal{U} \ \psi)$ *or closed, then we have* $\vdash_{\mathbf{H}} \bigwedge \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathcal{S}/\sigma \rightarrow \bot)$.

Proof. *If* $\mathcal{S}$ *is done-closed because it contains a formulae* $\langle\rangle : \mathsf{Done}(\alpha)$*, then we straightforwardly have:* $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathcal{S}/\sigma \rightarrow \bot)$*. Otherwise, consider the following set of formulae:*

$$\Gamma = \{\bigwedge \mathcal{B}/\sigma' \mid \sigma' \geq \sigma \wedge \mathcal{S} \subseteq \mathcal{B}\}.$$

First, note that, if $\mathcal{S}$ *does not contain a formula* $\langle\rangle : \mathsf{Done}(\alpha)$*, then none of the branches which extend* $\mathcal{S}$ *can be done-closed because of a formulae* $\langle\rangle : \mathsf{Done}(\alpha)$*, because* $\mathcal{S}$ *is reduced for every* $\sigma'' \leq \sigma$*. Now, let us prove that:*

1. $\vdash_{\mathbf{H}} \bigvee \Gamma \rightarrow \neg\psi$.
2. $\vdash_{\mathbf{H}} \bigvee \Gamma \rightarrow \mathsf{AN} \bigvee \Gamma$.

For item 1, let $\bigwedge \mathcal{B}'/\sigma' \in \Gamma$*; we know that* $\sigma' \geq \sigma$*. If there is some branch extending* $\mathcal{S}$ *which contains the formulae in* $\mathcal{B}'/\sigma'$ *which is ignorable, we know (since* $\mathcal{B}'$ *is completed for* σ' *and definition of* A*-ignorable) that* $\neg\psi \in \mathcal{B}'/\sigma'$*, which implies* $\vdash_{\mathbf{H}} \bigvee \Gamma \rightarrow \neg\psi$*. If all the branches containing* $\mathcal{B}/\sigma'$ *are closed, then all the labels obtained from* σ' *by the tableaux rules are closed branches, and therefore, by theorem 34 and theorem 32, we obtain* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \bot$*, and by propositional logic we obtain* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \neg\psi$*.*

For item 2, we show that for each $\mathcal{B}'/\sigma' \in \Gamma$*:* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \mathsf{AN} \bigvee \Gamma$*. First, since* $\mathcal{T}$ *is t-completed, we have applied to* σ *all the possible rules and then we have branches* $\mathcal{B}_1, \ldots, \mathcal{B}_n$*, so that we have that each* $\sigma \centerdot \gamma_i$ *is a label in a corresponding* $\mathcal{B}_i$ *for all atoms* γ_i*, and since this label can only be obtained from* σ *by the P rule and using theorem 34 we get* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \bigwedge_i \langle\gamma_i\rangle \bigvee \Gamma$*. Using axiom* ***A17*** *gives us:* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \bigwedge_i [\gamma_i] \bigvee \Gamma$*, but then, using the properties of modal logics, we get* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow [\mathbf{U}] \bigvee \Gamma$*. By axioms* ***TempAx1*** *and* ***TempAx2*** *and PL, we obtain* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}'/\sigma' \rightarrow \mathsf{AN} \bigvee \Gamma$*.*

From 1 and 2 and rule ***TempRule4****, we obtain* $\vdash_{\mathbf{H}} \bigvee \Gamma \rightarrow \neg\mathsf{E}(\varphi \,\mathcal{U}\, \psi)$*; but, since* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}/\sigma \rightarrow \bigvee \Gamma$*, we have that* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}/\sigma \rightarrow \neg\mathsf{E}(\varphi \,\mathcal{U}\, \psi)$*, and since* $\mathsf{E}(\varphi \,\mathcal{U}\, \psi) \in \mathcal{B}/\sigma$*, we have that* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}/\sigma \rightarrow \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$*. This implies* $\vdash_{\mathbf{H}} \bigwedge \mathcal{B}/\sigma \rightarrow \bot$*, and, sincSe* $\mathcal{S}$ *is reduced for* σ*, we have that:* $\mathcal{S}/\sigma = \mathcal{B}/\sigma$ *and then* $\vdash_{\mathbf{H}} \bigwedge \mathcal{S}/\sigma \rightarrow \bot$*, which implies* $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathcal{S}/\sigma \rightarrow \bot)$*.* ■

We can prove the same theorem for formulae of the style $\mathsf{A}(\varphi \,\mathcal{U}\, \psi)$.

Theorem 36. *Let* $\mathcal{T}$ *be a t-complete tableau, and* $\mathcal{S}$ *a subbranch of* $\mathcal{T}$ *with a formula* $\sigma : \mathsf{A}(\varphi \,\mathcal{U}\, \psi) \in \mathcal{S}$ *and such that every* $\sigma' \leq \sigma$ *is reduced in* $\mathcal{S}$ *and every branch* $\mathcal{B}$ *of* $\mathcal{T}$ *such that* $\mathcal{S} \subseteq \mathcal{B}$ *is either* A*-ignorable for* $\sigma : \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$ *or closed. Then we have* $\vdash_{\mathbf{H}} \bigwedge \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathcal{S} \rightarrow \bot)$*.*
Proof. *Similar to the proof of theorem 35.* ■

Now, we prove that tableaux theorems can be proven using the Hilbert system.

Theorem 37. *If* $\vdash_{\mathbf{T}} \varphi$*, then* $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$*.*
Proof. *If* $\vdash_{\mathbf{T}} \varphi$*, then, if we start a tableau with* $\langle\rangle : \neg\varphi$*, we obtain all closed or ignorable branches. But, because of theorems 36, 35 and 30, for every branch* $\mathcal{B}_i$ *we have:*

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\bigwedge \mathbf{H}(\mathcal{B}_i) \rightarrow \bot).$$

Now, because of theorem 33 and the fact that the formulae in the branches are obtained by applying the rules to φ, we have that

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\varphi \rightarrow \bigvee(\bigwedge \mathbf{H}(\mathcal{B}_i)))$$

and then:

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow (\neg\varphi \rightarrow \bot)$$

and by PL we obtain:

$$\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow \varphi$$

■

Intuitively, this theorem says that the theorems proven in tableaux can be proven in the Hilbert style system taking into account that the validity of tableaux is an anchored one. Using this result we can prove that the theorems of both systems coincide when we consider AG tableaux formulae.

Theorem 38. $\vdash_{\mathbf{T}} \mathsf{AG}\varphi \Leftrightarrow \vdash_{\mathbf{H}} \varphi$
Proof. *If we have $\vdash_{\mathbf{T}} \mathsf{AG}\varphi$, then by the theorem above we have $\vdash_{\mathbf{H}} \neg\mathsf{Done}(\mathbf{U}) \rightarrow \mathsf{AG}\varphi$, and therefore, from rule **TempRule5**, we obtain: $\vdash_{\mathbf{H}} \varphi$. The other direction is by theorem 29.* ■

Summarizing, if we want to prove a property using tableaux with need to prove that it is true for every path and every instant, if we succeed to do this, then this formula is valid in the sense of $\models$. Another consequence of the theorem above is that the Hilbert system is complete..

Theorem 39. $\models \varphi \Rightarrow \vdash_{\mathbf{H}} \varphi$.
Proof. *Suppose that $\models \varphi$; then we have $\models_A \mathsf{AG}\varphi$, and, therefore, since the tableaux-system is complete, we obtain $\vdash_{\mathbf{T}} \mathsf{AG}\varphi$, but this implies, by the theorem above, that $\vdash_{\mathbf{H}} \varphi$.* ■

6.2 Summary

In this chapter, we have shown that the two deductive systems described in earlier chapters are related. i.e., they allow us to obtain similar theorems. Tableaux systems have been used in earlier work to prove the completeness of Hilbert style systems, in particular in temporal logic. For example, in [EH82] a tableaux system is proposed to demonstrate the decidability of CTL and then this system is used to prove the

completeness of a Hilbert style system, although in this work a tableaux based one graphs is used instead of one based on trees.

It is important to note the technique that we use to relate the tableaux system with the Hilbert system; the properties of the atoms of the boolean algebra of terms are used to translate labelled formulae to standard formulae, and therefore basic facts over the tableaux are proven using the Hilbert calculus. The main properties are that closed and ignorable branches are inconsistent set of formulae, and tableaux rules introduces equivalent formulae to formulae already in the current branch. It seems possible to use this technique with other logics with different algebraization of actions, this seems to be an interesting topic of research.

Chapter 7

An Extended Logic for the Support of Modularity

In this chapter we introduce some modifications to the deontic logic presented in chapter 3 with the aim of obtaining a more general framework where system specifications can be written in a modular way. For this purpose, we mainly follow the philosophy of [BG77], in the sense that a system is specified by putting together smaller specifications (by means of some categorical constructions). The ideas presented below are also inspired by the logical frameworks presented in [FM91b] and [FM92], where Goguen and Burstall's ideas are applied to temporal logics and, therefore, to specifications of concurrent systems and object oriented systems, respectively. In [FM91b] a logic with support for modularization, with temporal constructs, is presented. This logic incorporates deontic predicates (permission, obligation, prohibition) to give a broader language to specify objects (or modular units); deontic predicates (as shown in the earlier chapters) allow designers to separate the concepts of description and prescription of pieces of software. The descriptional part of the logic of a component describes what the effects of the actions are of this component (in a pre/post-condition style). On the other hand, the deontic aspect describes how this module should behave, though a component may exhibit a different behaviour to that which is expected. In [FM91b] a linear temporal logic is used and the deontic constructs are global, in the sense that the prescriptions given in terms of them are shared by all the components of the specification. In the structured version of MAL presented in [RFM91] prescriptions are also global. Here we present a branching time logic (which reflects in some way that the notion of non-determinism is embedded inside of this logic). Moreover, we change some definitions in PDL so that the prescriptions in one component do not affect the other components in the system (i.e., they are intended to be *local*).

The chapter is organized as follows. In section 7.1 we present the logical machinery which allows us to specify different modular units of a system. In particular, we exhibit a formal framework, which is useful when analyzing the violations that may occur during the execution of the system being specified. The deontic operators play a key role here: violations arise since an obligation is not fulfilled, or a prohibition is violated. Interestingly, some natural properties of allowed actions (namely that an allowed action does not introduce more violations in a given state) are useful to facilitate the reasoning about the behaviour of the system in faulty scenarios. We give some simple examples which are intended to show the application of these ideas in practice.

7.1 Modularizing the Deontic Logic

In this chapter we use the notion of vocabulary (or language) introduced at the end of chapter 3, i.e., we use vocabulary to refer to a tuple $L = \langle \Delta_0, \Phi_0, V_0, I_0 \rangle$, where Δ_0 (as before) is a finite set of *primitive actions*: $a_1, ..., a_n$, which represent the possible actions of a part of the system and, perhaps, of its environment. Φ_0 is an enumerable set of propositional symbols denoted by $p_1, p_2, \ldots$. V_0 is a finite subset of $\mathcal{V}$, where $\mathcal{V} = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}_2, \ldots\}$ is an infinite, enumerable set of "violation" propositions, and I_0 is a finite index set of (categories of) permissions. The set of formulae is defined in the same way as chapter 3. We also define a modified version of the Done() operator; this operator can be thought of as being a restriction of the standard Done() operator relativised to a restricted set of actions.

- If α is an action and $S \subseteq \Delta_0$, then $\mathsf{Done}_S(\alpha)$ is a formula.

The intuitive reading of $\mathsf{Done}_S(\alpha)$ is: *if we restrict the actions of the component to those appearing in S, then the last action executed was* α. Note that the classic Done() operator is just $\mathsf{Done}_{\Delta_0}(\alpha)$; for the sake of simplicity, sometimes we write $\mathsf{Done}(\alpha)$ instead of $\mathsf{Done}_{\Delta_0}(\alpha)$. That is, we have similar formulae to chapter 3, but we add indexed permissions and a more general version of the Done().

We also add a logical constant to predicate about the initial state of a system:

- B is a formula.

Note that in the logic defined in chapter 3 the predicate B is not needed since it is expressed by $\neg\mathsf{Done}(\mathbf{U})$; however, here, since we consider transitions which can be

labeled with external events, this formula does not denote the initial instant of the system, but the initial state of the current component.

We also introduce some changes to the semantic structures to give the semantics of this variation of the logic. Basically, we follow the ideas of [FM91b], where transitions can be produced by actions in the language or by external components (i.e., this is an *open system* approach in the sense that is given in [Bar87]). Intuitively, each action produces a (finite) set of events during the execution of the system (the events that this action "observes"), and also there are other events produced by actions from other components or from the environment.

Definition 40 (models). *Given a language* $L = \langle \Phi_0, \Delta_0, V_0, I_0 \rangle$, *an L-Structure is a tuple:* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\} \rangle$ *where:*

- $\mathcal{W}$, *is a set of worlds.*
- $\mathcal{E}$, *is an infinite, enumerable non-empty set, of (names of) events.*
- $\mathcal{R}$, *is an* $\mathcal{E}$*-labeled relation between worlds. We require that, if* $(w, w', e) \in \mathcal{R}$ *and* $(w, w'', e) \in \mathcal{R}$, *then* $w' = w''$, *i.e.,* $\mathcal{R}$ *is functional.*
- $\mathcal{I}$, *is a function:*
  - *For every* $p \in \Phi_0 : \mathcal{I}(p) \subseteq \mathcal{W}$
  - *For every* $\alpha \in \Delta_0 : \mathcal{I}(\alpha) \subseteq \mathcal{E}$, *and* $\mathcal{I}(\alpha)$ *is finite.*

  In addition, the interpretation $\mathcal{I}$ *has to satisfy the following properties:*

  I.1 *For every* $\alpha_i \in \Delta_0$: $|\mathcal{I}(\alpha_i) - \bigcup\{\mathcal{I}(\alpha_j) \mid \alpha_j \in (\Delta_0 - \{\alpha_i\})\}| \leq 1$.

  I.2 *For every* $e \in \mathcal{I}(a_1 \sqcup \cdots \sqcup a_n)$: *if* $e \in \mathcal{I}(\alpha_i) \cap \mathcal{I}(\alpha_j)$, *where* $\alpha_i \neq \alpha_j \in \Delta_0$, *then:*
  $\cap\{\mathcal{I}(\alpha_k) \mid \alpha_k \in \Delta_0 \wedge e \in \mathcal{I}(\alpha_k)\} = \{e\}$.
- *each* $\mathcal{P}^i \subseteq \mathcal{W} \times \mathcal{E}$ *is a relation which indicates which event is permitted in which world with respect to permissions with index* i.

□

Roughly speaking, the structure gives us a labeled transition system, whose labels are events, which are produced by some action or they could also be external events. Note that we have a set of events, but actions are only interpreted over finite subsets, whose intersections satisfy the condition **I.2**, i.e., we require that every one-point set

can be generated from the actions of the component. We call *standard models* those structures where $\mathcal{E} = \bigcup_{\alpha \in \Delta_0} \mathcal{I}(\alpha)$, i.e., when we do not have "outside" events in the structure. Note that the semantics of the logic described in chapter 3 is given only in terms of standard models.

We use maximal traces to give the semantics of the temporal operators, and in the following we use the notation introduced in chapter 3, Since, in a trace, we have events that do not belong to the actual components, we need to distinguish between those events generated by the component being specified and those which are from the environment. Given a language L, an L-structure M and a maximal path π in M, we define the set:

$$\mathrm{Loc}_L(\pi) = \{i \mid \pi(i) \in \mathcal{I}(a_1 \sqcup \cdots \sqcup a_n)\} \cup \{0\}$$

(where $a_1, \ldots, a_n$ are all the primitive actions of L), i.e., this set contains all the positions of π where events occur that are observed by some action in L. Obviously, this set is totally ordered by the usual relationship $\leq$. Also we consider a restricted version of this set; given a set $\{a_1, \ldots, a_m\} \subseteq \Delta_0$, we define:

$$Loc_{\{a_1,\ldots,a_m\}}(\pi) = \{i \mid \pi(i) \in \mathcal{I}(a_1 \sqcup \cdots \sqcup a_m)\}$$

In the following, given a set S of naturals, we denote by $\min_p(S)$ the minimum element in S which satisfies the predicate p, and similarly for $\max_p(S)$. Using these concepts, we define the semantics in a similar way to that used in [CM07a], but taking into account the separation between local and external events.

Definition 41. *Given a trace $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots \in \Sigma^*(w)$, we define the relation $\models_L$ as follows:*

- *If $p_j \in \Phi_0 \cup V_0$, then $\pi, i, M \models_L p_j \overset{\text{def}}{\iff} p_j \in \mathcal{I}(\pi_i)$.*
- $\pi, i, M \models_L \mathsf{P}^i(\alpha) \overset{\text{def}}{\iff} \forall e \in \mathcal{I}(\alpha) : \mathcal{P}^i(w, e)$.
- $\pi, i, M \models_L \mathsf{P}^i_{\mathsf{w}}(\alpha) \overset{\text{def}}{\iff} \exists e \in \mathcal{I}(\alpha) : \mathcal{P}^i(w, e)$.
- $\pi, i, M \models_L \neg\varphi \overset{\text{def}}{\iff}$ *not* $\pi, i, M \models_L \varphi$.
- $\pi, i, M \models_L \varphi_1 \rightarrow \varphi_2 \overset{\text{def}}{\iff}$ *either not* $\pi, i, M \models_L \varphi_1$ *or* $\pi, i, M \models_L \varphi_2$.
- $\pi, i, M \models_L \mathsf{Done}_S(\alpha) \overset{\text{def}}{\iff} \exists j : j = max_{<i}(Loc_S(\pi)) \wedge e_j \in \mathcal{I}(\alpha)$.
- $\pi, i, M \models_L [\alpha]\varphi \overset{\text{def}}{\iff} \forall \pi' = s'_0 \xrightarrow{e'_0} s'_1 \xrightarrow{e'_1} \ldots \in \Sigma^*(w)$ *such that* $\pi[0, i] \prec \pi'$, *if* $j = min_{>i}(Loc(\pi'))$, *and if* $e'_j \in \mathcal{I}(\alpha)$, *then* $\pi', j, M \models_L \varphi$.

- $\pi, i, M \models_L \mathsf{AN}\varphi \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \models \varphi$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi' :$, *if* $j = min_{>i}(Loc(\pi'))$, *then* $\pi', j, M \models \varphi$.

- $\pi, i, M \models_L \mathsf{A}(\varphi_1 \; \mathcal{U} \; \varphi_2) \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \models \varphi_2$. *If* $i \neq \#\pi$, *then* $\forall \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *we have that* $\exists j \in Loc((\pi')^i) : \pi', j, M \models \varphi_2$ *and* $\forall i \leq k \leq j : k \in Loc((\pi')^i)$, *then* $\pi', k, M \models \varphi_1$.

- $\pi, i, M \models_L \mathsf{E}(\varphi_1 \; \mathcal{U} \; \varphi_2) \stackrel{\text{def}}{\Longleftrightarrow}$ *if* $i = \#\pi$, *then* $\pi, i, M \models \varphi_2$. *If* $i \neq \#\pi$, *then* $\exists \pi' \in \Sigma^*(w) : \pi[0..i] \prec \pi'$ *such that* $\exists j \in Loc((\pi')^i) : \pi', j, M \models \varphi_2$ *and* $\forall i \leq k \leq j : k \in Loc((\pi')^i)$, *then* $\pi', k, M \models \varphi_1$.

□

We say that $\models_L \varphi$, if $\pi, i, M \models \varphi$, for every model M and path π. (In this chapter, when we use the symbol $\models$, we refer to this relationship and not the one defined in chapter 3.)

We can think of the propositional variables in L as local variables, which cannot be changed by other components, i.e., we must require (as done in [FM91b] and [FM92]) that external events do not produce changes in local variables. In [FM92] the notion of a *locus* trace is introduced to reflect this property in the logic; a locus (trace) is one in which the external events do not affect the state of local variables. However, the logic used in that work is a linear temporal logic, and this implies that we cannot restrict only to traces to express this requirement, since we have a branching temporal logic. In the following we take further the ideas introduced in [FM92] and we define *locus models* which have the property of generating locus traces. We need to investigate the model theory of our logic more deeply to be able to define this concept.

We have presented an axiomatic system for an ealier version of this temporal logic in chapter 3. We need to add some axioms to that system to deal with the new operators introduced above. The axioms for the propositional part of the logic are:

1. The set of propositional tautologies.
2. A set of axioms for boolean algebras for action terms (a complete one), including standard axioms for equality.
3. The following set of axioms:

   A1. $[\emptyset]\varphi$

   A2. $\langle\alpha\rangle\varphi \wedge [\alpha]\psi \rightarrow \langle\alpha\rangle(\varphi \wedge \psi)$

A3. $[\alpha \sqcup \alpha']\varphi \leftrightarrow [\alpha]\varphi \wedge [\alpha']\varphi$

A4. $[\alpha]\varphi \rightarrow [\alpha \sqcap \alpha']\varphi$

A5. $\mathsf{P}^i(\emptyset)$, for every index i.

A6. $\mathsf{P}^i(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}^i(\alpha) \wedge \mathsf{P}^i(\beta)$, for every index i.

A7. $\mathsf{P}^i(\alpha) \vee \mathsf{P}^i(\beta) \rightarrow \mathsf{P}^i(\alpha \sqcap \beta)$, for every index i.

A8. $\neg\mathsf{P}^i_{\mathsf{w}}(\emptyset)$, for every index i.

A9. $\mathsf{P}^i_{\mathsf{w}}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha) \vee \mathsf{P}^i_{\mathsf{w}}(\beta)$, for every index i.

A10. $\mathsf{P}^i_{\mathsf{w}}(\alpha \sqcap \beta) \rightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha) \wedge \mathsf{P}^i_{\mathsf{w}}(\beta)$, for every index i.

A11. $\mathsf{P}^i(\alpha) \wedge \alpha \neq \emptyset \rightarrow \mathsf{P}^i_{\mathsf{w}}(\alpha)$, for every index i.

A12. $\mathsf{P}^i_{\mathsf{w}}(\gamma) \rightarrow \mathsf{P}^i(\gamma)$, where $[\gamma]$ is an atom in Δ_0/Φ_{BA} and for every index i.

A13. $\mathsf{O}^i(\alpha) \leftrightarrow \mathsf{P}^i(\alpha) \wedge \neg\mathsf{P}^i_{\mathsf{w}}(\overline{\alpha})$, for every index i.

A14. $[\alpha]\varphi \leftrightarrow \neg\langle\alpha\rangle\neg\varphi$

A15. $(a_1 \sqcup \ldots \sqcup a_n) =_{act} \mathbf{U}$

A16. $\langle\beta\rangle(\alpha =_{act} \alpha') \rightarrow \alpha =_{act} \alpha'$

A17. $\langle\gamma\rangle\varphi \rightarrow [\gamma]\varphi$, where $[\gamma]$ is an atom of Δ_0/Φ_{BA}

BA. $\varphi[\alpha] \wedge (\alpha =_{act} \alpha') \rightarrow \varphi[\alpha/\alpha']$

For the temporal extension of the logic consider the axioms above plus:

TempAx1. $\langle\mathbf{U}\rangle\top \rightarrow (\mathsf{AN}\varphi \leftrightarrow [\mathbf{U}]\varphi)$

TempAx2. $[\mathbf{U}]\bot \rightarrow (\mathsf{AN}\varphi \leftrightarrow \varphi)$

TempAx3. $\mathsf{AG}\varphi \leftrightarrow \neg\mathsf{E}(\top\, \mathcal{U}\, \neg\varphi)$

TempAx4. $\mathsf{E}(\varphi\, \mathcal{U}\, \psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ENE}(\varphi\, \mathcal{U}\, \psi))$

TempAx5. $\mathsf{A}(\varphi\, \mathcal{U}\, \psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ANA}(\varphi\, \mathcal{U}\, \psi))$

TempAx6. $[\bigsqcup_{a \in S} a \sqcap \alpha]\mathsf{Done}_S(\alpha)$

TempAx7. $[\bigsqcup_{a \in S} a \sqcap \overline{\alpha}]\neg\mathsf{Done}_S(\alpha)$

TempAx8. $\neg\mathsf{Done}_S(\emptyset)$

TempAx9. $\mathsf{B} \rightarrow \neg\mathsf{Done}_S(\alpha)$

TempAx10. $[\mathbf{U}]\neg\mathsf{B}$

TempAx11. $\mathsf{Done}_S(\alpha) \rightarrow [\overline{\bigsqcup_{a \in S} a}]\mathsf{Done}_S(\alpha)$

TempAx12. $\neg\mathsf{Done}_S(\alpha) \rightarrow [\overline{\bigsqcup_{a \in S} a}]\neg\mathsf{Done}_S(\alpha)$

And the following deduction rules:

- Rules given in [CM07a] for the propositional part of the logic.

TempRule1. if $\vdash \mathsf{B} \rightarrow \varphi$ and $\vdash \varphi \rightarrow [\mathbf{U}]\varphi$, then $\vdash \varphi$

TempRule2. if $\vdash \varphi$, then $\vdash \mathsf{AG}\varphi$

TempRule3. if $\vdash \varphi \rightarrow (\neg\psi \wedge \mathsf{EN}\varphi)\}$, then $\vdash \varphi \rightarrow \neg\mathsf{A}(\varphi' \,\mathcal{U}\, \psi)$

TempRule4. if $\vdash \varphi \rightarrow (\neg\psi \wedge \mathsf{AN}(\varphi \vee \neg\mathsf{E}(\varphi' \,\mathcal{U}\, \psi)))$, then $\vdash \varphi \rightarrow \neg\mathsf{E}(\vartheta \,\mathcal{U}\, \psi)$

TempRule5. if $\vdash \neg\mathsf{Done}(\mathbf{U}) \rightarrow \mathsf{AG}\varphi$, then $\vdash \varphi$

The new axioms are **TempAx6-TempAx12**. Axioms **TempAx9** and **TempAx10** define the basic properties of the B predicate: they imply that no action was performed before, and after executing any action, B becomes false. Note that we also use B instead of $\neg\mathsf{Done}(\mathbf{U})$ in the induction rule. The rest of the axioms define the relativised Done() operator; note that in these axioms $\bigsqcup_{a \in S} a$ denotes the choice between all the actions in S. It is important to remark that in the case that $S = \Delta_0$ (i.e., when S is the set of all the primitive actions of the language), then the properties of $\mathsf{Done}_{\Delta_0}()$ are exactly those of the standard Done() operator as defined in chapter 3.

7.1.1 A Touch of Model Theory

For the next definition we fix two structures $M_1 = \langle \mathcal{W}_1, \mathcal{R}_1, \mathcal{E}_1, \mathcal{I}_1, \mathcal{P}_1, w_1 \rangle$, and $M_2 = \langle \mathcal{W}_2, \mathcal{R}_2, \mathcal{E}_2, \mathcal{I}_2, \mathcal{P}_2, w_2 \rangle$, then we define the notion of morphism between M_1 and M_2.

Definition 42. *A morphism $m : M_1 \rightarrow M_2$ between M_1 and M_2 is a pair of functions $\langle f_W : \mathcal{W}_1 \rightarrow \mathcal{W}_2, f_E : \mathcal{E}_1 \rightarrow \mathcal{E}_2 \rangle$, which satisfies:*

M0 $f_W(w_1) = w_2$.

M1 *For every $p_i \in \Phi_0$ and $w \in \mathcal{W}_1$, if $p_i \in \mathcal{I}_1(w)$, then $p_i \in \mathcal{I}_2(f_W(w))$.*

M2 *For every $e \in \mathcal{E}_1$ and $a_i \in \Delta_0$, if $e \in \mathcal{I}_1(a_i)$, then $f_E(e) \in \mathcal{I}_2(a_i)$.*

M3 *For every* $e \in \mathcal{E}_1$ *and* $w, w' \in \mathcal{W}_1$, *if* $w \xrightarrow{e} w' \in \mathcal{R}_1$, *then* $f_W(w) \overset{f_E(w)}{\rightarrow} f_W(w') \in \mathcal{R}_2$.

M4 *For every* $e \in \mathcal{E}_1$ *and* $w \in \mathcal{W}_1$, *if* $\langle w, e\rangle \in \mathcal{P}_1^i$, *then* $\langle f_W(w), f_E(e)\rangle \in \mathcal{P}_2^i$.

□

We say that a morphism $m = \langle f_W, f_E\rangle$ is surjective, if f_W and f_E are onto, and we say that m is injective if both f_W and f_W are injective. We introduce the concept of *strong morphism* (where we follow the terminology used in the model theory of modal logics [BRV01]).

Definition 43. *A morphism* $m : M_1 \rightarrow M_2$ *is strong iff the conditions* ***M1-M4*** *are equivalences.* □

We say that a morphism $m = \langle f_W, f_E\rangle : M_1 \rightarrow M_2$ is a *bijective morphism* if m is a strong morphism and the components f_W and f_E are bijections; we denote this situation by $M_1 \cong M_2$. Note that, given a morphism $m : M_1 \rightarrow M_2$ and given a trace

$$\pi = w \xrightarrow{e_1} w_1 \xrightarrow{e_2} \ldots,$$

we can define a corresponding trace:

$$m(\pi) = f_W(w) \overset{f_E(e_1)}{\rightarrow} f_W(e_2) \ldots,$$

If $m = \langle f_W, f_E\rangle : M_1 \rightarrow M_2$ is a bijective morphism, then $m^{-1} = \langle f_W^{-1}, f_E^{-1}\rangle : M_2 \rightarrow M_1$ is also a bijective morphism, and it is the inverse of m, i.e., $m \circ m^{-1} = id_{M_1}$ and $m^{-1} \circ m = id_{M_2}$.

It is straightforward to prove that, if we have a bijective morphism between two models, these models are elementarily equivalent, that is:

Theorem 40. *Given two models* M_1 *and* M_2, *if* $M_1 \cong M_2$, *then* $M_1 \models \varphi$ *iff* $M_2 \models \varphi$, *for every formula* φ *of* L.

However, the existence of a bijective morphism is a strong requirement, and we are interested in situations when the models are not exactly isomorphic but where the structure of one of them is somehow preserved by the other. The notion of bisimulation (introduced in the context of process algebra [Mil79]) has been shown to be useful to show equivalence of modal formulae with respect to Kripke semantics [vB76], and with respect to temporal logics [BCG87] and [DV95]. Here, we describe

a notion of bisimulation that is useful for our purposes and is related to branching bisimulation [vGW89] and stuttering bisimulation [MCBG88]; note that in the latter case, the notion of bisimulation is defined over non-labeled systems, while our notion of bisimulation is defined over two different labeled transition systems. Using the terminology of [DV95], we can say that the bisimulation presented later on is a *sensitive divergence* bisimulation, since it distinguishes between processes which diverge by non-local events.

Recall that, in a given structure M over a language L, we say that an event e is non-local if it does not belong to the interpretation of any action of the language; otherwise, we say that it is a local event. We introduce some notation useful for the following sections. We say $w \stackrel{\epsilon}{\Rightarrow} w'$, if there exists a path $w \stackrel{e_0}{\rightarrow} w_1 \stackrel{e_1}{\rightarrow} w_2 \stackrel{e_2}{\rightarrow} \ldots \stackrel{e_n}{\rightarrow} w_n$, such that e_i is non-local for every $0 \leq i \leq n$. We say that $w \stackrel{\infty}{\Rightarrow}$ when there is an infinite path from w: $w \stackrel{e_0}{\rightarrow} w_1 \stackrel{e_1}{\rightarrow} \ldots$, such that every e_i is non-local. Furthermore, we say $w \stackrel{e}{\Rightarrow} w'$ (where e is local) if $w \stackrel{\epsilon}{\Rightarrow} w''$ and $w'' \stackrel{e}{\rightarrow} w'$.

Given two structures M and M', such that $\mathcal{E} = \mathcal{E}'$ (i.e., they have the same events), assume $\mathcal{I}(\alpha) = \mathcal{I}'(\alpha)$ for any α (the interpretation of every action gives us the same events). We say that a relationship $Z \subseteq W \times W'$ is a *local bisimulation* iff:

- If wZv, then $L(w) = L(v)$.
- If wZv, and $w \stackrel{\infty}{\Rightarrow}$, then either $v \stackrel{\infty}{\Rightarrow}$ or there is a v' such that $v \stackrel{\epsilon}{\Rightarrow} v'$ and v' has no successors by $\rightarrow$.
- if wZv and $w \stackrel{e}{\rightarrow} w'$, then $w'Zv$ if e is non-local. Otherwise, we have some v' such that $v \stackrel{e}{\rightarrow} v'$ and $w'Zv'$.
- $Z^{\smile}$ also satisfies the above conditions (where $Z^{\smile}$ is the converse of Z).

Here $L(v)$ denotes the set of all the state formulae (primitive propositions, deontic predicates and equations) true at state v. (Note that the composition of two local bisimulations is a local bisimulation, and the identity relation is a local bisimulation.) In branching bisimulation (as defined in [DV95]), we can "jump" through non-local events; however, here we require a stronger condition: we can move through non-local events, but, if we have the possibility of executing a local event, we must have the same possibility in the related state. We see later on that this notion of bisimulation induces useful properties on the models and that we can characterize this notion in an axiomatic way.

We say that two models M and M' are *bisimilar* iff $w_0 Z w_0'$ (where w_0 and w_0' are the corresponding initial states) for some local bisimulation Z; we denote this situation by $M \sim_Z M'$. We prove later on that two bisimilar models are indistinguishable

by our logic. In [DV95], it is shown that CTL^*-X (CTL^* without the next operator) cannot distinguish between Kripke structures which are DBSB (divergent blind stuttering) bisimilar; however, in the semantics of the temporal logic considered in that work, there are no labels on the transitions and, therefore, the next operator is problematic since it is interpreted as a global next operator. On the other hand, here we can take advantage of the fact that we have the events as labels of transitions, and, therefore, we can distinguish between local and non-local transitions. Furthermore, note that our next operator is a local one (although this implies some subtle technical points when it comes to defining the composition of components, see below).

Using bisimilarity, we define the notion of a *locus* model (following the terminology of [FM92] where locus models are introduced in a linear temporal logic).

Definition 44. *We say that a structure M is a locus iff there is a local bisimulation between M and a standard model M'.* □

It is worth noting that in this work we are not interested in comparing the expressivity of our logic with respect to the notion of bisimilarity introduced above (as done in [DV95]). Instead, we use this notion of bisimulation to formalize the notion of locus structure that, as shown later on, will be essential in defining composition of modules (or components). Roughly speaking, locus models are those which have a behaviour which is, essentially, the same as that of a standard model. Hence, the usual notion of encapsulation, as informally understood in software engineering, applies to our concept of component: only local actions can modify the values of local variables.

We extend the definition of bisimulation to paths.

Definition 45. *Given a path $\pi = w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} \ldots$ in M and a path $\pi' = v_0 \xrightarrow{d_0} v_1 \xrightarrow{d_1} \ldots$ in M', and a local bisimulation between M and M' such that $M \sim_Z M'$, we say that $\pi Z \pi'$, iff when $w_i Z v_i$, then:*

- *if we have $w_i \xrightarrow{e_1} \ldots \xrightarrow{e_n} w_n$ in π, with e_j non-local for $0 \leq j \leq n$, then we have $v_j \xrightarrow{d_1} \ldots \xrightarrow{d_m} v_m$ in π', with d_l non-local for every $0 \leq l \leq m$, such that $w_n Z v_m$.*
- *if we have $w_i \xrightarrow{e} w_{i+1}$ in π, where e is a local action, then we have a (sub)path in π': $v_j \xrightarrow{d_1} \ldots \xrightarrow{e} v_m$ such that for all $1 \leq l < m$, d_l are non-local, and $w_n Z v_m$.*
- *we also have $\pi' Z^{\smile} \pi$.*

□

This is similar to the definition of stuttering equivalence, but taking into account the labels. Our first property about paths and local bisimulation is the following.

Theorem 41. *If $\pi[0..i]Z\pi'[0..j]$, then there exists a $\pi'[0..j] \preceq \pi_2$ such that $\pi Z \pi_2$.*
Proof. *If from position i in π we have an infinite sequence of non-local events, then $\pi_i \stackrel{\infty}{\Rightarrow}$ and therefore, by definition of local bisimulation, $\pi'_j \stackrel{\infty}{\Rightarrow}$. Thus we have some full path π_2 such that $\pi Z \pi_2$. Otherwise, we have some e and k such that $\pi_i \stackrel{e}{\Rightarrow} \pi_k$ in π; but, since $\pi_i Z \pi'_j$ we can find a state $v_{k'}$ in M' such that $\pi'_j \stackrel{e}{\Rightarrow} v_{k'}$. We denote by $\pi''[0..k]$ the extension of $\pi'[0..j]$ obtained by adding the path above. Then, we have $\pi[0..k]Z\pi''[0..k']$. Thus, for any extension of $\pi[0..i]$, we can find a corresponding extension of π', and therefore take π_2 to be the maximal such extension and we have $\pi Z \pi_2$.* ■

It is worth noting that, since $Z^\smile$ satisfies the same conditions as Z, we have that the above theorem also is true when we replace Z by $Z^\smile$. Note that, if $\pi Z \pi'$, we can define a mapping f_π between positions of π and positions of π' as follows, $f_\pi(0) = 0$ and:

$$f_\pi(n+1) = \begin{cases} f_\pi(n) & \text{if } e_n \text{ is non-local} \\ \mathsf{min}_{>f_\pi(n)}(Loc(\pi')) & \text{otherwise} \end{cases}$$

where $\pi = w_0 \stackrel{e_0}{\rightarrow} w_1 \stackrel{e_1}{\rightarrow} \ldots$. In the same way we can define a function $f_{\pi'}$. A useful property of these functions is the following.

Property 7. *If $\pi Z \pi'$, then $\pi_i Z \pi'_{f_\pi(i)}$, for every position i of π.*
Proof. *The proof is by induction; for the basis it is straightforward. For the inductive case, suppose that $\pi_i Z \pi'_{f_\pi(i)}$; if $\pi_i \stackrel{e_i}{\rightarrow} \pi_{i+1}$ and e_i is non-local, then $f_\pi(i+1) = i$ and $\pi_{i+1} Z \pi'_{f_\pi(i)}$ by definition of local bisimulation. Otherwise, $f_\pi(i+1) = min_{>f_\pi(i)}(Loc(\pi'))$, and by definition of bisimulation between paths we get $\pi_{i+1} Z \pi_{f_\pi(i+1)}$.* ■

Property 8. *If $\pi Z \pi'$, $\#Loc(\pi[0..i]) = \#Loc(\pi'[0..f_\pi(i)])$.*
Proof. *The proof is by induction on i; the basis is straightforward: $\#Loc(\pi[0..0]) = 0 = \#Loc(\pi'[0..f_\pi(i)])$. For the inductive case: suppose that:*

$$\#Loc(\pi[0..i]) = \#Loc(\pi[0..f_\pi(i)]).$$

Then, if $\pi_i \stackrel{e_i}{\rightarrow} \pi_{i+1}$ in π and e_i is non-local, then $\#Loc(\pi[0..i+1]) = \#Loc(\pi[0..i])$, and then $f_\pi(i+1) = f_\pi(i)$ and $\#Loc(\pi[0..i+1]) = \#Loc(\pi[0..f_\pi(i+1)])$. If e_i is local, then $\#Loc(\pi[0..i+1]) = \#(Loc(\pi[0..i]) \cup \{e_i\})$ and then we have $\pi'_{f_\pi(i)} \stackrel{e_i}{\Rightarrow} \pi'_{f_\pi(i+1)}$, and then $\#Loc(\pi'[0..f_\pi(i+1)]) = \#(Loc(\pi'[0..f_\pi(i)]) \cup \{e_i\}) = \#Loc(\pi[0..i+1])$. ■

A useful corollary of the above property is the following.

Corollary 10. *If $\pi Z \pi'$, then either:*

- $\pi_i = \pi_{f_{\pi'}(f_\pi(i))}$, *or*
- $\pi_i \stackrel{\epsilon}{\Rightarrow} \pi_{f_{\pi'}(f_\pi(i))}$, *or*
- $\pi_{f_{\pi'}(f_\pi(i))} \stackrel{\epsilon}{\Rightarrow} \pi_i$

■

By definition of f_π and $f_{\pi'}$, we obtain the following properties which resemble the properties of Galois connections.

Property 9. *If* $\pi Z \pi'$, *then:* $\pi'_{f_\pi(i)} \stackrel{\epsilon}{\Rightarrow} \pi'_k$ *in* π' *iff* $\pi_i \stackrel{\epsilon}{\Rightarrow} \pi_{f_{\pi'}(k)}$

Our first important theorem says that bisimilar full paths satisfy the same properties:

Theorem 42. *If* $\pi Z \pi'$, *then, for all position* i, $\pi, i, M \models \varphi \Leftrightarrow \pi', f_\pi(i), M \models \varphi$.
Proof. *The proof is by induction on* φ.
Base Case. We know $L(\pi_i) = L(\pi'_{f_\pi})$, *which implies that* $\pi, i, M \models p_j$ *iff* $\pi', f_\pi(i), M' \models p_j$. *The proof is similar for equations and deontic predicates. For the* $\mathsf{Done}_S()$ *operator, suppose that* $\pi, i, M \models \mathsf{Done}_S(\alpha)$, *then, for* $k = max_{<i}(Loc_S(\pi))$, *we have that* $e_k \in \mathcal{I}(\alpha)$, *and* $\pi_{k-1} Z \pi'_{f_\pi(k-1)}$. *But then we have* $\pi'_{f_\pi(k-1)} \stackrel{\epsilon}{\Rightarrow} \pi'_{f_\pi(k)}$, *and* $\pi'_{f_\pi(k)} \stackrel{\epsilon}{\Rightarrow} \pi'_{f_\pi(i)}$, *thus* $\pi', f_\pi(i), M' \models \mathsf{Done}_S(\alpha)$.

Ind. Case. If $\pi, i, M \models [\alpha]\varphi$, *then suppose* $\pi', f_\pi(i), M' \not\models [\alpha]\varphi$. *Then, for some* $\pi_2 \succeq \pi'[0..f_\pi(i)]$, *we have a* $k = \mathsf{min}_{>i}(Loc(\pi_2))$ *such that* $(\pi_2)_k \in \mathcal{I}(\alpha)$ *and* $\pi_2, k, M' \not\models \varphi$. *By theorem 41, we know that we have a full path* $\pi_1 \succeq \pi[0..i]$ *such that* $\pi_1 Z \pi_2$. *By the definition of bisimulation between paths we know that if* $(\pi_2)_{f_{\pi_1}(i)} \stackrel{\epsilon}{\Rightarrow} (\pi_2)_k$ *in* π_2, *then* $(\pi_1)_i \stackrel{\epsilon}{\Rightarrow} (\pi_1)_{f_{\pi_2}(k)}$ *in* π_1. *Applying induction on the symmetric statement of the theorem, we get* $\pi_2, f_{\pi_2}(k), M \not\models \varphi$ *which is a contradiction. The other direction is similar.*

If $\pi, i, M \models \mathsf{AN}\varphi$ *the argument is as above.*

If $\pi, i, M \models \mathsf{A}(\varphi \,\mathcal{U}\, \psi)$, *suppose* $\pi', f_\pi(i), M' \not\models \mathsf{A}(\varphi \,\mathcal{U}\, \psi)$. *Then, if* $\pi', f_\pi(i), M' \not\models \varphi$, *we get a contradiction. Otherwise, we must have a full path* $\pi_2 \succeq \pi'[0..f_\pi(i)]$, *such that for every* $j \in Loc(\pi_2)$ *we have* $\pi_2, j, M' \not\models \psi$. *Now, as explained above, we have a* $\pi_1 Z \pi_2$ *and for this* π_1 *we have a* $k \in Loc(\pi_1)$ *such that* $\pi_1, k, M \models \psi$, *for this* k *we have that* $\pi_2, f_\pi(k), M' \models \psi$, *by induction. But note that* $\pi_2(f_\pi(k)) \in Loc(\pi_2)$ *which gives us a contradiction, and therefore* $\pi', f_\pi(i), M' \models \mathsf{A}(\varphi \,\mathcal{U}\, \psi)$. *The other direction is similar.*

If $\pi, i, M \models \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$, *and suppose* $\pi', f_\pi(i), M' \not\models \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$, *then if* $\pi', f_\pi(i), M' \not\models \varphi$ *and* $\pi', f_\pi(i), M', \not\models \psi$, *we get a contradiction. Otherwise, we have that for every path* $\pi'' \succeq \pi'[0..f_\pi(i)]$ *and for every* $k \in Loc(\pi_2^{f_\pi(i)})$, $\pi'', k, M' \not\models \psi$ *holds . Note that we have a* π_2 *in* M' *such that* $\pi_1 Z \pi_2$ *(where* π_1 *is that full path mentioned above), and then by induction* $\pi_2, f_\pi(i), M' \models \psi$. *Furthermore, note that* $\pi_2(f_\pi(i))$ *is a local event, which contradicts the assumption above, and therefore* $\pi', f_\pi(i), M' \models \mathsf{E}(\varphi \,\mathcal{U}\, \psi)$. *The other direction is similar.* ■

As a corollary, we get that local bisimilar structures satisfy the same predicates.

Theorem 43. *If* $M \sim_Z M'$, *then* $M \models \varphi$ *iff* $M' \models \varphi$.
Proof. *Suppose that* $M \models \varphi$ *and* $M' \not\models \varphi$; *therefore, we have that* $\pi, i, M' \not\models \varphi$ *for some full path* π *and position* i. *But then we get by the theorem above that* $\pi', f_{\pi'}(i), M \not\models \varphi$ *for some* $\pi' Z \pi$ *(which exists since* $M \sim_Z M'$*). The other direction is similar.* ■

7.1.2 Locus Models.

At the beginning of section 7.1, we introduced non-standard models (i.e., those models which have "external" events). However, not all non-standard models are useful; we want that the external events preserve local variables, that is, the events not generated by any of the actions in the component have to be silent, in some sense. In [FM92], with the same purpose in mind, the notion of locus trace is introduced. A *locus* trace is one in which, after executing a non-local event, the local variables retain their value. However, since we have a branching time logic and a modal logic, in our logic it is not enough to just put restrictions on traces. We need to take into account the branching occurring in the semantic structures, i.e., we need a more general notion of locus model.

Roughly speaking, locus models are those which are local bisimilar to a standard model. In some sense, this definition characterizes those models which behave as standard models, where the external actions are silent with respect to local attributes.

Definition 46. *Given a language* L, *we say that a* L*-structure* M' *is a locus iff there is a standard model* M *such that* $M \sim_Z M'$ *for some local bisimulation* Z. □

Using the result presented above about local bisimulation, we get that locus structures do not add anything new to the logic (w.r.t. formula validity).

Theorem 44. *If* M *is a locus structure, then* $M \models \varphi$ *iff there is some standard structure* M' *such that* $M' \models \varphi$.
Proof. *If* M *is standard the result follows. Otherwise we use theorem 43.* ■

Using this theorem we can prove that the axiomatic system described in section 7.1 is sound with respect to locus structures.

Theorem 45. *The axiomatic system of section 7.1 is sound with respect to locus structures and the relation* $\models$ *defined in section 7.1.*
Proof. *Note that, if we only take into account standard structures, the definition of* $\models$ *coincides with the definition given in chapter 3. Therefore, axioms* **A1 – A17** *are sound with respect to standard models and then, by theorem 44, these axioms are sound with respect to locus models; the same is true for axioms* **TempAx1 – TempAx5** *and the rules; the rest of the axioms are straightforward using the definition of* B *and the relativized* Done(). ■

Summarizing, nothing is gained or lost in using the locus models of a given language. However, we want to use these kinds of models over a wider notion of logical system; we shall consider several languages and translations between them, and therefore we need to have a notion of model which agrees with the locality properties of a language when we embed this language in another. First, let us define what is a translation between two languages.

Definition 47. *A translation* τ *between two languages* $L = \langle \Delta_0, \Phi_0, V_0, I_0 \rangle$ *and* $L' = \langle \Delta'_0, \Phi'_0, V'_0, I'_0 \rangle$ *is given by:*

- *A mapping* $f : \Delta_0 \rightarrow \Delta'_0$ *between the actions of component* C *and the actions of* C'.
- *A mapping* $g : \Phi_0 \rightarrow \Phi'_0$ *between the propositions of* L *and the propositions of* L'.
- *A mapping* $h : V_0 \rightarrow V'_0$, *between the violations of* L *and the violations of* L'.
- *A mapping* $i : I_0 \rightarrow I'_0$, *between the indexes of* L *and the indexes of* L'.

□

For the sake of simplicity, we denote the application of any of these functions using the name of the mapping, e.g., instead of writing $f(a_i)$ we write $\tau(a_i)$.

The collection of all the languages and all the translations between them forms the category **Sign**. It is straightforward to see that it is really a category: identity functions define identity arrows, and composition of functions gives us the composition of translations (which straightforwardly satisfy associativity). Now, given a translation, we can extend this translation to formulae (actually we can describe a functor

grammar which reflects these facts, as done in Institutions [GB92] or π-Institutions [FS87]). Given a translation $\tau : L \rightarrow L'$ as explained above, we extend τ to a mapping between the formulae of L and those of L', as follows. First, we need to define how the translation behaves with respect to action terms:

- $\tau(\alpha \sqcup \beta) = \tau(\alpha) \sqcup \tau(\beta)$.
- $\tau(\alpha \sqcap \beta) = \tau(\alpha) \sqcap \tau(\beta)$.
- $\tau(\overline{\alpha}) = \tau(\mathbf{U}) \sqcap \overline{\tau(\alpha)}$.
- $\tau(\mathbf{U}) = \tau(a_1) \sqcup \cdots \sqcup \tau(a_n)$, where $\{a_1, \ldots, a_n\} = \Delta_0$.
- $\tau(\emptyset) = \emptyset$.

Note that the complement is translated as a relative complement, and the universal action is translated as the non-deterministic choice of all the actions of the original component (which is different from the universal action in the target language). It is important to stress that some extra axioms must be added to the axiomatic system to deal with the fact that the actions are interpreted as being relative to a certain universe. Now, the extension to formulae is as follows:

- $\tau([\alpha]\varphi) = [\tau(\alpha)]\tau(\varphi)$
- $\tau(\neg\varphi) = \neg\tau(\varphi)$.
- $\tau(\varphi \rightarrow \psi) = \tau(\varphi) \rightarrow \tau(\psi)$
- $\tau(\mathsf{AN}\varphi) =$
  $(\langle\tau(\mathbf{U})\rangle\mathsf{T} \rightarrow \mathsf{AN}(\mathsf{Done}(\tau(\mathbf{U}))) \rightarrow \tau(\varphi))) \vee ([\tau(\mathbf{U})]\bot \rightarrow \tau(\varphi))$
- $\tau(\mathsf{A}(\varphi\, \mathcal{U}\, \psi)) = \mathsf{A}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))$
- $\tau(\mathsf{E}(\varphi\, \mathcal{U}\, \psi)) = \mathsf{E}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))$
- $\tau(\mathsf{Done}_S(\alpha)) = \mathsf{Done}_{\tau(S)}(\tau(\alpha))$, where $\tau(S) = \{\tau(a_i) \mid a_i \in S\}$

In other words, using translations between signatures, we can define morphisms between formulae, and therefore we can define interpretations between theories (in the sense of [Eme72]); we deal with this issue in the next section.

Note that, given a translation $\tau : L \rightarrow L'$ and given a L'-structure M, it is straightforward to define the restriction of $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \{\mathcal{P}^i \mid i \in I_0\}\rangle$ with respect to τ (or its reduct as it is called in model theory [CK73]), as follows:

Definition 48. *Given a translation $\tau : L \longrightarrow L'$ and an L'-structure M we can define an L-structure $M|_\tau$ as follows:*

- $\mathcal{W}|_\tau = \mathcal{W}$.
- $\mathcal{E}|_\tau = \mathcal{E} - \{e \in \mathcal{I}(\tau(\mathbf{U})) \cap \mathcal{I}(\Delta_0' - \tau(\Delta_0))\}$.
- $\mathcal{I}|_\tau(a_i) = \{e \in \mathcal{I}(a_i) \mid e \in \mathcal{E}|_\tau\}$, *for every* $a_i \in \Delta_0'$.
- $\mathcal{I}|_\tau(p_i) = \mathcal{I}(\tau(p_i))$, *for every* $p_i \in \Phi_0$.
- $\mathcal{R}|_\tau = \{w \xrightarrow{e} w' \in \mathcal{R} |\ e \in \mathcal{E}|_\tau\}$.
- $\mathcal{P}^i|_\tau(w, e) \Leftrightarrow P^{\tau(i)}(w, e)$.
- $w_0|_\tau = w_0$.

□

It is worth noting that the restriction of a standard structure of L' can be a non-standard structure of L. Note also that we take out of the model those events which belong to translated actions and actions outside of the translation, i.e., we only keep those events which are obtained by executing only actions of L or those which are obtained by executing actions outside of L. Some restrictions added below ensure that no important property of the original model is lost when we take its reduct.

Translations between languages and restrictions between models define a functor which is used in Institutions [GB92] to define logical systems; we investigate the institutional aspects of our logic later on. An important problem with restrictions is that a restriction of a given structure could be a structure which is not a locus, i.e., the obtained semantic entity violates the notion of locality as explained above; Furthermore, perhaps the reduct of a model loses some important properties. For this reason, we introduce the concept of τ-locus structures. We define some requirements on translations; given a translation $\tau : L \longrightarrow L'$, consider the following set of formulae of the form:

- $\langle\tau(\gamma)\rangle\top \rightarrow \langle\tau(\gamma) \sqcap \overline{a_1} \sqcap \cdots \sqcap \overline{a_n}\rangle\top$, where γ is an atom of the boolean term algebra Δ_0/Φ_{BA}, and $a_1, \ldots, a_n \in \Delta_0' - \tau(\Delta_0)$.

These formulae say that the execution of the actions of L when translated to L' are not dependent on any action of L'; we can think of this as an independence requirement,

i.e., the actions of L when translated to L' keep their independence. This is an important modularity notion. In practice, this can be ensured by implementing the two components (which these languages describe) in different processes. We denote this set of formulae by $\mathsf{ind}(\tau)$. Another requirement (which is related to independence) is that the new actions in Δ'_0 (those which are not translations of any action in L) do not add new non-determinism to the translated actions. This fact can be expressed by the set of formulae with the following form:

- $\langle\tau(\gamma)\rangle\tau(\varphi) \rightarrow [\tau(\gamma)]\tau(\varphi)$, $\mathsf{P}_{\mathsf{w}}^{\tau(i)}(\tau(\gamma)) \rightarrow \mathsf{P}^{\tau(i)}(\tau(\gamma))$ for every atom γ of the boolean algebra of terms obtained from L, formula φ of L, and index i of L.

For a given translation $\tau : L \rightarrow L'$, we denote this set of formulae by $\mathsf{atom}(\tau)$, since they reflect the fact that the atomicity of the actions in L is preserved by translation.

Definition 49. *Given a translation $\tau : L \rightarrow L'$ and a L'-structure M, we say that M is a τ-locus iff:*

- $M \models \mathsf{ind}(\tau)$.
- $M \models \mathsf{atom}(\tau)$
- *The restriction $M|_\tau$ is a locus structure for L.*

■

That is, a locus structure with respect to a translation τ is a structure which respects the locality and independence of L. We have obtained a semantical characterization of structures which respect the local behaviour of a language with respect to a given translation; because we wish to use deductive systems to prove properties over a specification, it is important to obtain some axiomatic way of characterizing this class of structures. So, a natural question is: is there some way of characterizing locus models? We shall prove that we have an affirmative answer to this question. Let us investigate some properties of τ-locus models. The first property says that in $M|_\tau$ we have all the paths that are needed.

Property 10. *Given a τ-locus model M, for every full path π' of M such that $\pi' \succeq \pi[0..i]$ (where $\pi[0..i]$ is a subpath in $M|_\tau$), there is full path $\pi'' \succeq \pi[0..i]$ in M such that π'' also is a full path of $M|_\tau$ and for any formula $\tau(\varphi)$: $\pi', i, M \models \tau(\varphi)$ iff $\pi'', i, M \models \tau(\varphi)$.*

Proof. *The proof is direct using the properties of independence and atomicity.* ■

Property 11. *If $\tau : L \to L'$, and M is a L' structure which is a τ-locus, then if $\pi, i, M|_\tau \models \varphi$, and $\pi(i)$ is non-local, then $\pi, i+1, M|_\tau \models \varphi$.*
Proof. *The proof is by induction on φ; the cases are straightforward using the properties of local bisimulation, and the fact that a τ-locus model is bisimilar to a standard model.* ■

Another useful property is the following:

Property 12. *If $\tau : L \to L'$, and M is a L' structure which is a τ-locus, then if for a path π of $M|_\tau$ we have $\pi_i \xrightarrow{e} \pi_{i+1}$ where e is local for $M|_\tau$, then there is no π' such that $\pi' \succeq \pi[0..i]$ and from position i all the events of π' are non-local for $M|_\tau$.*
Proof. *Suppose that we have such a path; then, since $M|_\tau$ is bisimilar to a standard model, we can bisimulate the path π' until i. Thus, we have some state v in the standard model such that $\pi_i Z v$, but from there π' diverges with non-local events, and therefore there is no way to bisimulate it. In addition, v has a successor since π_i has a successor reachable by local events. From here we obtain that M is not a τ-locus model, which is a contradiction.* ■

First, let us prove that local properties are preserved by τ-locus structures.

Theorem 46. *Let $\tau : L \to L'$ be a translation and M an L'-structure. If M is a τ-locus, then for full path π of $M|_\tau$, $\pi, i, M \models \tau(\varphi)$ iff $\pi, i, M|_\tau \models \varphi$, for any formulae φ of L.*
Proof. *The proof is by induction on φ.*
Base Case. It is straightforward using the definition of $M|_\tau$.
Ind. Case. If $\pi, i, M \models \tau([\alpha]\varphi)$ which is equivalent to $\pi, i, M \models [\tau(\alpha)]\tau(\varphi)$, and now suppose that $\pi, i, M|_\tau \not\models [\alpha]\varphi$. From here we have that there exists a path $\pi' \succeq \pi[0..i]$ such that $\pi, i+1, M|_\tau \not\models \varphi$, where $\pi'(i+1) \in \mathcal{I}|_\tau(\alpha)$. Now we have the same trace in M, which gives us a contradiction by induction. If $\pi, i, M|_\tau \models [\alpha]\varphi$, suppose $\pi, i, M \not\models [\tau(\alpha)]\tau(\varphi)$, and then we have a $\pi' \succeq \pi[0..i]$ (noting that, if π' is not a full path of $M|_\tau$ then, applying property 10, we can find an equivalent path which belongs to this model) such that $\pi, i, M \not\models \tau(\varphi)$ and $\pi'(i) \in \mathcal{I}(\tau(\alpha))$. By definition of reduction, we have that $\pi'(i) \in \mathcal{I}|_\tau(\alpha)$, which applying induction, gives us a contradiction, and therefore $\pi, i, M \models [\tau(\alpha)]\tau(\varphi)$.

Suppose $\pi, i, M \models \tau(\mathsf{AN}(\varphi))$ and $\pi, i, M|_\tau \not\models \mathsf{AN}\varphi$. Then, if i is the last position of π, then we have $\pi, i, M|_\tau \not\models \varphi$, which gives us a contradiction, since by induction this implies $\pi, i, M \not\models \varphi$. If i is not the last position of π, then, for $k = min_{>i}(Loc_L(\pi))$, we have $\pi, k, M|_\tau \not\models \varphi$, note that $\pi_i \stackrel{e}{\Rightarrow} \pi_k$ where e is local for L, and then in M we have that it is the next position where an event of $a_1 \sqcup \cdots \sqcup a_n$ is executed, and therefore $\pi, k, M \not\models \mathsf{Done}(a_1 \sqcup \cdots \sqcup a_n) \to \varphi$, which contradicts what we said above, and therefore $\pi, i, M|_\tau \models \mathsf{AN}\varphi$. The other direction is similar.

Suppose that $\pi, i, M \models \tau(\mathsf{A}(\varphi\ \mathcal{U}\ \psi))$ *and* $\pi, i, M|_\tau \not\models \mathsf{A}(\varphi\ \mathcal{U}\ \psi)$. *If* $\pi, i, M|_\tau \not\models \varphi$ *and* $\pi, i, M|_\tau \not\models \psi$ *(and the same reasoning is applied when* φ *and* ψ *are not true at some moment before* ψ *comes true), then by induction we obtain a contradiction. If, for some* $\pi' \succeq \pi[0..i]$, *we have that* $\pi', k, M|_\tau \not\models \psi$, *for every* $k \in Loc_L(\pi^i)$, *then note that for this* π' *in* M *we have* $\pi, i, M \not\models \tau(\psi)$ *(by induction) and from here if a position* $j \leq i$ *is reached by a non-local event for* L *we have, by property 11 and induction, that* $\pi', j, M \models \tau(\psi)$, *and if it is local, then we have by the supposition above that* $\pi, j, M \models \tau(\psi)$, *i.e., for every* $j \geq i$ $\pi, j, M \not\models \tau(\psi)$, *which contradicts our initial assumption, and therefore* $\pi, i, M|_\tau \models \mathsf{A}(\varphi\ \mathcal{U}\ \psi)$. *The other direction is similar.*

If $\pi, i, M \models \tau(\mathsf{E}(\varphi\ \mathcal{U}\ \psi))$, *then we have some* $\pi' \succeq \pi[0..i]$ *such that there is a* k *such that* $\pi, k, M \models \tau(\psi)$ *where* $k \geq i$, *and for every* $j \in Loc_{L'}(\pi')$ *such that* $i \leq j \leq k$, *we have* $\pi', j, M \models \tau(\varphi)$; *then, since* $Loc_L(\pi') \subseteq Loc_{L'}(\pi')$ *and using induction, we have that for every* $j \leq k$ *such that* $j \in Loc_L(\pi')$, $\pi, j, M|_\tau \models \varphi$, *and* $\pi, k, M|_\tau \models \psi$. *Now if* $k \notin Loc_L(\pi')$, *then, by proposition 10, it must be a* $k' \in Loc(\pi')$ *such that* $k' \leq k$ *and* $\pi', k', M|_\tau \models \psi$. *On the other hand, if* $\pi, i, M|_\tau \models \mathsf{E}(\varphi \in \psi)$, *then we have some* $\pi' \succeq \pi[0..i]$ *such that* $\pi', k, M|_\tau \models \psi$, *where* $k \in Loc_L(\pi')$, *and for every* $i \leq j \leq k$ *with* $j \in Loc_L(\pi')$ *we have* $\pi', j, M|_\tau \models \varphi$. *Note that, using property 10, we have that for every position* $j \in Loc_{L'}(\pi')$ *such that* $i \leq j \leq k$, *we have that* $\pi', j, M \models \tau(\varphi)$ *(since, if it is a local event for* L, *we show above that it satisfies* φ, *otherwise it preserves the property), and* $k \in Loc_{L'}(\pi')$ *and then* $\pi, k, M \models \psi$ *by induction.* ■

We have a semantic characterization of τ-locus models, but since we want to use deductive systems, we need a syntactic characterization of this class of models. For a given translation $\tau : L \to L'$, consider the following (recursive) set of formulae:

$$\{\tau(\varphi) \to [\overline{\tau(\mathbf{U})}]\tau(\varphi) \mid \varphi \in \Phi'\}.$$

Roughly speaking, this set of axiom schemes says that if an action of an external component is executed, then the local state of the current module is preserved. Note that, in [FM92], a similar set of axioms is proposed, although in that case it is a finite set, since that work uses a linear temporal logic, and therefore preserving only the propositions is enough for having a good notion of locality. However we need other axioms to express the property that when we embed a module inside another part of the system, we want to ensure that the behaviour of the smaller module is preserved, in the following sense: we can introduce external events in some way in a given trace but we do not want that these external events add divergences that were not in the original trace. The following axiom does this:

$$\langle\tau(\mathbf{U})\rangle\mathsf{T} \to \mathsf{AFDone}(\tau(\mathbf{U})).$$

This axiom expresses one of the conditions of local bisimulation, namely a trace cannot diverge by non-local events unless the component cannot execute any local

action. It is worth noting that, if a local action is enabled in some state, then after executing a non-local action it will continue being enabled (as a consequence of the axiomatic schema described above), i.e., we require a fair scheduling of components, one which will not always neglect a component wishing to execute some of its actions.

Definition 50. *Given two languages* $L = \langle \Delta_0, \Phi_0, V_0, I_0 \rangle$ *and* $L' = \langle \Delta_0', \Phi_0', V_0', I_0' \rangle$ *and a translation* $\tau : L \to L'$ *we define the following set of formulae:*

$$Loc(\tau) = atom(\tau) \cup ind(\tau) \cup \{\tau(\varphi) \to [\widetilde{\tau(\mathbf{U})}]\tau(\varphi) \mid \varphi \in \Phi\} \cup \{\langle \tau(\mathbf{U}) \rangle \top \to \mathsf{AFDone}(\tau(\mathbf{U}))\}$$

□

A nice property is that this set of formulae characterizes the τ-loci L'-structures.

Theorem 47. *Given a translation* $\tau : L \to L'$*, then a* L'*-structure is a* τ*-locus iff* $M \models Loc(\tau)$.

Proof. *First, let us prove that, if* M *is a* τ*-locus, then it satisfies* $Loc(\tau)$*. By definition it satisfies* $\mathsf{ind}(\tau)$ *and* $\mathsf{atom}(\tau)$*), and by property 10 and theorem 46 we have that the model satisfies the axiomatic schema. On the other hand, note that the other axiom is satisfied since we require that* M *is local bisimilar to a standard model, and therefore, if in some state* w *we have the possibility of executing a local event, from this state there cannot be a path which always observes non-local events, since otherwise the standard model does not satisfy the divergence condition of local bisimulation.*

For the other direction, suppose that M *satisfies the axioms; we build a model which is standard and which is bisimilar to the original model. First, we define the following collection of states:*

$$[\epsilon] = \{v \mid w_0 \stackrel{\epsilon}{\Rightarrow} v\} \cup \{w_0\}$$

and:

$$[es \cdot e] = \{v \mid \exists z, v' : (z \in [es]) \wedge (z \stackrel{e}{\Rightarrow} v') \wedge ((v' \stackrel{\epsilon}{\Rightarrow} v) \vee v = v')\}.$$

Then we define the components of the new model as follows:

- $\mathcal{W}^{\#} = \{e_1 \cdot e_n \mid [e_1 \cdot e_n] \neq \emptyset\}$.
- $\mathcal{R}^{\#} = \{es \xrightarrow{e} es \cdot e \mid es, es \cdot e \in \mathcal{W}^{\#}\}$.
- $\mathcal{P}^{\rangle\#} = \{\langle es, e \rangle \mid \exists v \in [es] : \langle v, e \rangle \in \mathcal{P}^{\rangle}|_{\tau}\}$.
- $\mathcal{I}^{\#}(a_i) = \mathcal{I}|_{\tau}(a_i)$.
- $\mathcal{I}^{\#}(p_i) = \{es \in \mathcal{W}^{\#} \mid \exists w \in [es] : w \in \mathcal{I}|_{\tau}(p_i)\}$.

Note that, if $w \in \mathcal{I}|_\tau(p)$ *and* $w \in [es]$, *then, for every* $v \in [es]$, *we have* $v \in \mathcal{I}|_\tau(p)$. *This is because non-local events preserve propositions. This structure is well-defined since it satisfies* ***I1*** *and by definition the transitions are deterministic with respect to a given event. It is straightforward to see that this structure is standard since there are no external events. Now, we define a relationship* Z *as follows:*

$$wZ[es] \Leftrightarrow w \in [es].$$

Let us prove that it is a local bisimulation.

Suppose that $wZ[es]$; *if* $w \stackrel{\infty}{\Rightarrow}$, *then we know that* $[es]$ *cannot diverge by non-local events. The only possibility is that there is no* e *such that* $[es] \xrightarrow{e} [es.e]$; *if there is such a transition, then* w *cannot diverge by non-local events, since any path which passes through* w *will not satisfy the axioms in* $Loc(\tau)$, *and then* $[es]$ *has no successors. Now suppose that* $w \xrightarrow{e} w'$. *If* e *is non-local, then we know that* $w, w' \in [es]$, *and therefore* $w'Z[es]$, *and we know by property 10 that* $L(w) = L(w') = L([es])$. *If* $w \xrightarrow{e} w'$ *and* e *is local, then we know that* $w' \in [es \cdot e]$ *and then* $w'Z[es \cdot e]$, *and furthermore* $[es] \xrightarrow{e} [es \cdot e]$, *by definition. Now, if* $[es]Z^\smile w$, *it is worth noting that* $M^\#$ *does not have any divergence via non-local events. If* $[es] \xrightarrow{e} [es \cdot e]$, *we have some* $w' \in [es \cdot e]$ *(by definition), and note that we have some* $v \in [es]$ *and* $w'' \in [es \cdot e]$ *such that* $v \stackrel{\xi}{\Rightarrow} w''$ *and* $w'' \stackrel{\xi}{\Rightarrow} w'$. *Since* w *and* v *belong to* $[es]$, *both satisfy the same properties of* L *(which can be proved by a straightforward proof by induction) and therefore, since we have* $v \stackrel{e}{\Rightarrow} w''$ *by the axiomatic schema, we must have* $v, M \vDash \langle\gamma\rangle\top$, *where* $I(\gamma) = e$, *and therefore we have* $w, M \vDash \langle\gamma\rangle\top$, *i.e., there is a state* $v' \in [es \cdot e]$ *such that* $w \xrightarrow{e} v''$, *which finishes the proof.* ■

It is worth remarking again that by $\Gamma \vdash^L \varphi$ and $\vdash^L_S \varphi$ we denote two different situations. The first can be thought of as a "local" deduction relationship; this relationship holds when we have a proof, in the standard sense, of φ where some members of Γ may appear, but the only rule that we can apply over them is *modus ponens*. Instead, $\vdash^L_S \varphi$ says that, if we extend our axiomatic system with the formulae of S, then we can prove φ. This can be thought of as a *global* deduction (note that in this case we can apply any rule to the members of S to get a proof of φ). An important difference is that the former notion of deduction preserves the deduction theorem. However, this theorem is not valid for the global version of deduction, an easy counterexample is: $\vdash_{S,\varphi} \mathsf{AG}\varphi$ (where S, φ is an abbreviation for $S \cup \{\varphi\}$). However, we can prove a variation of the deduction theorem:

Theorem 48. $\vdash^L_{S,\varphi} \psi$ *iff* $\vdash^L_S \mathsf{AG}\varphi \rightarrow \psi$.
Proof. *The left direction is trivial.*

For the other direction, we prove the result by induction on the length of the proof. Base Case. *If the proof is of length* 1, *then* $\psi \in S$, *or* ψ *is an axiom. In both cases we*

obtain $\vdash_S^L \mathsf{AG}\varphi \rightarrow \psi$. *(If* $\psi = \varphi$ *it is direct to prove this sentence from the axiomatic system.)*
Ind. Case. If the proof is of length greater than or equal to 1, *then* ψ *was obtained by one of the following rules:*

1. *Via modus ponens from a formula* $\varphi' \rightarrow \psi$ *which appears before.*
2. *Via application of generalization to a formulae which appears before.*
3. *By induction.*
4. ψ *is some axiom or it belongs to* S

The last case is straightforward. The other cases are dealt with as follows:
Case 1: If we obtain it by modus ponens, then $\vdash_S^L \mathsf{AG}\varphi \rightarrow \varphi'$, *and* $\vdash_S^L \mathsf{AG}\varphi \rightarrow (\varphi' \rightarrow \psi)$ *(by induction), which using propositional logic gives us* $\vdash_S^L (\mathsf{AG}\varphi \rightarrow \varphi') \rightarrow (\mathsf{AG}\varphi \rightarrow \psi)$, *and using modus ponens we get* $\vdash_S^L \mathsf{AG}\varphi \rightarrow \psi$.

Case 2: If we obtain ψ *by generalization, then* $\phi = \mathsf{AG}\psi'$. *Then, we have by induction that* $\vdash_S^L \mathsf{AG}\varphi \rightarrow \psi'$; *applying generalization we get* $\vdash_S^L \mathsf{AG}(\mathsf{AG}\varphi \rightarrow \psi')$, *and then it is straightforward using the axioms to prove* $\vdash_S^L \mathsf{AGAG}\varphi \rightarrow \mathsf{AG}\psi'$. *But we have that* $\vdash_S^L \mathsf{AGAG}\varphi \leftrightarrow \mathsf{AG}\varphi$, *then using this property we have* $\vdash_S^L \mathsf{AG}\varphi \rightarrow \mathsf{AG}\psi'$. *For modal generalization the proof is similar*

Case 3: If we obtained ψ *by induction, this means that* $\vdash_{S,\varphi}^L \mathsf{B} \rightarrow \psi$ *and* $\vdash_{S,\varphi}^L [\mathbf{U}]\psi$, *and then by induction we obtain* $\vdash_S^L \mathsf{AG}\varphi \rightarrow (\mathsf{B} \rightarrow \psi)$ *and* $\vdash_S^L \mathsf{AG}\varphi \rightarrow [\mathbf{U}]\psi$, *and then we have that* $\mathsf{AG}\varphi \vdash_S^L \mathsf{B} \rightarrow \psi$ *and* $\mathsf{AG}\varphi \vdash_S^L [\mathbf{U}]\psi$. *But, then, using the induction rule we get* $\mathsf{AG}\varphi \vdash_S^L \psi$ *and then using the deduction theorem for the local notion of deduction we get* $\vdash_S^L \mathsf{AG}\varphi \rightarrow \psi$. ■

Now we can prove that the deductive machinery obtained by adding the locality axioms preserves translations of properties.

Theorem 49. *Given a translation* $\tau : L \rightarrow L'$, *if* $\vdash^L \varphi$, *then* $\vdash^{L'}_{Loc(\tau)} \tau(\varphi)$.
Proof. *We prove that the translation of every axiom of the deductive system of* L *is a theorem of the deductive system of* L', *and for the deduction rules, if we have the translation of the premises, then we can prove the conclusion, and therefore every proof in* L *can be simulated in* L', *modulo translation.*

For the axioms of the propositional part of the logic, only two axioms are dependent on the language: A12 and A17. For A17, note that the translation of the instances of this axiom, $\tau(\langle\gamma\rangle\varphi \rightarrow [\gamma]\varphi)$, *is exactly the axioms of atomicity, and therefore:* $\vdash^{L'}_{Loc(\tau)}$

$\tau(\langle\gamma\rangle\varphi \rightarrow [\gamma]\varphi)$. *For the axiom A12, the proof is similar. And since the other axioms are not dependent on the language, the translation of these axioms are instances of axioms in* L'. *For the deduction rules of the propositional part (modus ponens and modal generalization) the proof is straightforward. For the temporal axioms and rules we proceed by cases.*

TempAx1*: We need to prove:*

$$\vdash^{L'}_{Loc(\tau)} \tau(\langle\mathbf{U}\rangle\top \rightarrow \mathsf{A}\varphi \leftrightarrow [\mathbf{U}]\varphi).$$

Note the following property of Done()*:*

$$\langle\alpha\rangle\top \rightarrow ((\mathsf{ANDone}(\alpha) \rightarrow \varphi) \leftrightarrow [\alpha]\varphi).$$

Using this property, we obtain that the sentence above is equivalent to:

$$\langle\tau(\mathbf{U})\rangle\top \rightarrow ([\tau(\mathbf{U})]\tau(\varphi) \leftrightarrow [\tau(\mathbf{U})]\tau(\varphi))$$

which is obviously a theorem of $\vdash^{L'}_{Loc(\tau)}$.

TempAx2*: It is straightforward that* $[\tau(\mathbf{U})]\bot \vdash^{L'}_{Loc(\tau)} \tau(\varphi) \leftrightarrow \tau(\varphi)$ *and the property follows.*

TempAx3*: The translation of this axiom is an instance of the same axiom in* L'.

TempAx4*: Proving the right direction of the implication is direct; let us prove:*

$$\tau(\psi) \vee \tau(\mathsf{ENE}(\varphi\, \mathcal{U}\, \psi)) \vdash^{L'}_{Loc(\tau)} \mathsf{E}(\varphi\, \mathcal{U}\, \psi)$$

Using the property of Done() *described above; we obtain that the left part of the assertion above is equivalent to:*

$$\tau(\psi) \vee (\tau(\varphi) \wedge (\langle\tau(\varphi)\rangle\top \rightarrow \langle\tau(\varphi)\rangle\mathsf{E}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi)))) \wedge ([\tau(\alpha)]\varphi \rightarrow \tau(\varphi)).$$

Simple calculations (using the axioms for AN*) show that from this formula we can prove* $\mathsf{E}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))$.

TempAx5*: We have to prove:*

$$\tau(\psi) \wedge (\tau(\varphi) \wedge \tau(\mathsf{ANA}(\varphi\, \mathcal{U}\, \psi))) \vdash^{L'}_{Loc(\tau)} \mathsf{A}(\varphi\, \mathcal{U}\, \psi)$$

Using the definition of τ *and properties of* Done()*, the left part is equivalent to:*

$$\tau(\psi) \vee (\tau(\varphi) \wedge (\langle\tau(\mathbf{U})\rangle\top \rightarrow [\tau(\mathbf{U})]\mathsf{A}(\tau(\varphi)\, \mathcal{U}\, \tau(\psi))) \wedge [\tau(\mathbf{U})]\bot \rightarrow \tau(\psi)). \qquad (7.1)$$

Note that by locality we have that: $\vdash^{L'}_{Loc(\tau)} \varphi \rightarrow \mathsf{A}(\varphi\mathsf{WDone}(\tau(\mathbf{U})))$, *and note that:*

$$(\varphi \rightarrow \mathsf{A}(\varphi\mathsf{WDone}(\tau(\mathbf{U})))) \wedge \mathsf{AFDone}(()\tau(\mathbf{U})) \vdash^{L'}_{Loc(\tau)} \varphi \rightarrow \mathsf{A}(\varphi\, \mathcal{U}\, \mathsf{Done}(\tau(\mathbf{U}))).$$

Using the fact that we have the following axiomatic schema in $Loc(\tau)$*:*

$$\langle\tau(\mathbf{U})\rangle\top \rightarrow \mathsf{AFDone}(\tau(\mathbf{U}))$$

and that from the formula we obtain $\mathsf{Done}(\tau(\mathbf{U})) \rightarrow \mathsf{A}(\tau(\varphi)\,\mathcal{U}\,\tau(\psi))$, *we have that:*

$$\varphi \wedge \langle\tau(\mathbf{U})\rangle\top \vdash^{L'}_{Loc(\tau)} \mathsf{A}(\tau(\varphi)\,\mathcal{U}\,(\mathsf{A}(\tau(\varphi)\,\mathcal{U}\,\tau(\psi))))$$

which is equivalent to:

$$\varphi \wedge \langle\tau(\mathbf{U})\rangle\top \vdash^{L'}_{Loc(\tau)} \mathsf{A}(\tau(\varphi)\,\mathcal{U}\,\tau(\psi)).$$

The result follows.
Axioms ***TempAx6-TempAx9*** *are straightforward as their translations are instances of the same axioms.*
TempAx8*: The translation of this axiom is* $[\tau(\mathbf{U})]\neg\mathsf{B}$, *which can be proven using the properties of modalities.*
TempAx11 *and* ***TempAx12*** *are direct.*

For the induction rule we can proceed as follows. Note that, if we have $\vdash^{L'}_{Loc(\tau)} \mathsf{B} \rightarrow \tau(\varphi)$, *and* $\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow [\tau(\mathbf{U})]\tau(\varphi)$, *then we have:*

$$\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow [\tau(a_1) \sqcup \cdots \sqcup \tau(a_n)]\tau(\varphi)$$

and by locality we have:

$$\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow [\overline{\tau(a_1) \sqcup \cdots \sqcup \tau(a_n)}]\tau(\varphi)$$

and then using the properties of the logic we get:

$$\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow [\mathbf{U}]\tau(\varphi)$$

and then using the induction rule we get: $\vdash^{L'}_{Loc(\tau)} \tau(\varphi)$.

The temporal rule ***TempRule2*** *is straightforward. For the rule* ***TempRule3****, if we have:*

$$\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow (\neg\tau(\psi) \wedge \tau(\varphi))$$

this is equivalent to:

$$\vdash^{L'}_{Loc(\tau)} \tau(\varphi) \rightarrow \neg\tau(\psi) \wedge (\langle\tau(\mathbf{U})\rangle\tau(\varphi) \vee [\tau(\mathbf{U})]\bot \rightarrow \tau(\varphi)).$$

It is not hard to prove that this formula implies $\tau(\varphi) \rightarrow (\neg\tau(\psi) \vee \mathsf{EN}\tau(\varphi))$, *and then, applying* ***TempRule3****, we obtain* $\neg\mathsf{A}(\tau(\varphi)\,\mathcal{U}\,\tau(\psi))$. *For the rule* ***TempRule4****, the proof is similar using the locality axioms.* ■

Recall that a theory presentation is a tuple $P = \langle L, A\rangle$ where L is a language and A is a set of axioms; it defines a theory, which is the set of consequences of the axioms. We say that $\vdash_P \varphi$ when $\vdash^L_A \varphi$. We use the notion of theory presentation to define components. To that end, it is important to define interpretations between theory presentations.

Definition 51. *Given two theory presentations $P = \langle L, A\rangle$ and $P' = \langle L', A'\rangle$, then an interpretation is given by a translation $\tau : L \rightarrow L'$ such that $\vdash_{P', Loc(\tau)} \tau(\varphi)$, for every $\varphi \in A$.* □

As shown in [FS87], interpretations between theory presentations in structural logics preserve consequence; we can recast this result here if we add locality axioms to deductions. In this sense, it is important to understand that, from the logical point of view, we are considering a different deduction system per component. The resulting deduction system (of the final system) is determined by the way in which the different components are put together (as described below, a component is basically a theory presentation). First, let us prove that interpretations preserve consequences.

Theorem 50. *Given two theory presentations $P = \langle L, A\rangle$ and $P' = \langle L', A'\rangle$ and an interpretation $\tau : P \rightarrow P'$, we have that:*

$$\vdash_P \varphi \Rightarrow \vdash_{P', Loc(\tau)} \tau(\varphi)$$

Proof. *Suppose that we have $\vdash_P \varphi$; then we have a proof which uses some finite number of axioms of P (by the definition of proof). Let us say the proof is $\varphi_1, \ldots, \varphi_n$; but then we have that:*

$$\vdash^L \mathsf{AG}\varphi_1 \wedge \ldots \mathsf{AG}\varphi_n \rightarrow \varphi$$

by theorem 48, and then by theorem 49 we have that

$$\vdash^{L'}_{Loc(\tau)} \tau(\mathsf{AG}\varphi_1 \wedge \ldots \mathsf{AG}\varphi_n \rightarrow \varphi).$$

But using the definition of translation we get:

$$\vdash^{L'}_{Loc(\tau)} \mathsf{AG}(\tau(\varphi_1)) \wedge \cdots \wedge \mathsf{AG}(\tau(\varphi_n)) \rightarrow \tau(\varphi).$$

We know that $\vdash_{P', Loc(\tau)} \tau(\varphi_i)$ for every $0 \leq i \leq n$, and therefore by generalization we obtain that $\vdash_{P', Loc(\tau)} \mathsf{AG}(\tau(\varphi_i))$ for every $0 \leq i \leq n$, and then using modus ponens we have that $\vdash_{P', Loc(\tau)} \tau(\varphi)$. ■

This theorem implies that we have a "weakly structural" logic [FM93], i.e., we have to add locality assumptions to preserve consequence.

7.1.3 Putting Together Deontic Specifications

In this section we focus on the notion of component and the concepts needed to enable compositions of different components. First, we define the notion of "upgrading"

and "degrading" formulae. These formulae are built from violation propositions; an upgrading formula says that one or more violations will become false in the future, and a degrading formula says that one or more violations will become true in the future. These kind of formulae are useful to analyze when, in a given state, a component will go into a worse violation state or will go into a better violation state (with perhaps an absence of violations). This is an important aspect when analyzing fault-tolerant software, as we want to know from which error states we cannot recover and from which ones we can make some improvement. An order between violation states is needed to formalize upgrading and degrading formulae. Note that, for every violation state (i.e., predicates $V_L = *\mathtt{v_1} \wedge \cdots \wedge *\mathtt{v_n}$ over a language L), we can define a set:

$$\mathcal{U}(V_L) = \{\mathtt{v_i} \mid \mathtt{v_i} \text{ appears without a negation in } V_L\}.$$

These sets induce an order $\leq_v$ over violation states as follows:

$$V_L \leq_v V'_L \Leftrightarrow U(V_L) \supseteq U(V'_L)$$

Note that $\leq_v$ is contravariant with respect to $\subseteq$. Intuitively, only the violations which are true at that point appear in $\mathcal{U}(V_L)$. The relationship $\leq_v$ denotes a relationship of "improvement", the set of sets $U(V_L)$ form a lattice; we study this fact in detail later on. We denote by $<_v$ the strict version of $\leq_v$.

Given a component (see below), an upgrading formula in this component is a formula which specifies some way of recovering from a violation, i.e., an upgrading formula identifies a recovery action for a given violation. Taking into account the order defined over the violation states, we can say that a recovery action improves the state of violation of a component. These facts inspire the following definition.

Definition 52. *Given a language L, the set of upgrading formulae for L is defined as follows:*

- *If V_L and V'_L are violation states in L, φ is a formulae in L and $V_L <_v V'_L$ then,*

$$(V_L \rightarrow ([\alpha_1; \ldots; \alpha_n]V'_L) \wedge (\langle\alpha_1; \ldots; \alpha_n\rangle\top))$$

is an upgrading formula, where $\alpha_1, \ldots, \alpha_n$ are actions in L.

□

Here we define $[\alpha;\beta]\varphi = [\alpha][\beta]\varphi$. Also note that we require that there must exist some way of executing the sequence of actions in the upgrading formulae (otherwise an impossible action will be a recovery action). One might think that the condition $(V \rightarrow$

$\langle \alpha_1; \ldots; \alpha_n \rangle V'_L$) is a better formalization of upgrading formulae, but it is too weak; it says that, sometimes, by executing $\alpha_1, \ldots, \alpha_n$ we eliminate some violations, and it does not give us any details about those scenarios corresponding to bad situations from which we can effectively recover. Instead, using box modalities, we can ensure that executing the sequence of actions in the predicate we can always recover from a violation state. At first sight, this could be too strong a requirement, however, we can refine the sequence of actions $\alpha_1; \ldots; \alpha_n$ as much as required to describe exactly the actions needed to upgrade the actual state of violations.

In the same way, we can define the set of degrading formulae, which intuitively define actions which introduce violations.

Definition 53. *Given a language L, the set of degrading formulae is defined as follows:*

- *If V_L and V'_L are violation states in L, φ is a formula in L and $V'_L <_v V_L$ then,*

$$(V_L \rightarrow ([\alpha_1; \ldots; \alpha_n]V'_L) \wedge \langle \alpha_1; \ldots; \alpha_n \rangle \top))$$

is a degrading formula, where $\alpha_1, \ldots, \alpha_n$ are actions in the language L.

□

A component is a piece of specification which is made up of a language, a finite set of axioms, and a set of additional axioms which formalize implicit assumptions on the components (e.g., locality axioms). These implicit axioms are not intended to be defined by a designer; instead, they are automatically obtained from the structure of our system (using the relationships between the different components).

Definition 54. *A component is a tuple $\langle L, A, S \rangle$ where:*

- *L is a language as described in earlier sections.*
- *A is a finite set of axioms (the axioms given by the designers).*
- *S is a set of axioms (the system axioms).*

□

Given a component $C = \langle L, A, S \rangle$ we denote by $\vdash_C \varphi$ the assertion $\vdash^L_{A,S} \varphi$. Usually we consider that $G(L) \subseteq S$ or $SG(L) \subseteq S$(i.e., the GGG predicate is in the system axioms). A mapping between two components is basically an interpretation between the theory presentations that define them.

Definition 55. *A mapping $\tau : C \rightarrow C'$ between two components $C = \langle L, A, S \rangle$ and $C' = \langle L', A', S' \rangle$ is a translation $\tau : L \rightarrow L'$ such that:*

- $\vdash_{C'} \tau(\varphi)$, *for every* $\varphi \in A \cup S$.
- $\vdash_{C'} Loc(\tau)$.

□

It is worth noting that we require that the locality axioms must be theorems in the target component to ensure that the properties of the smaller component are preserved (as proved by theorem 50). This is expressed by the following corollary.

Corollary 11. *If $\tau : C \rightarrow C'$ is a mapping between components C and C', then:* $\vdash_C \varphi \Rightarrow \vdash_{C'} \tau(\varphi)$.

In this thesis we focus on *horizontal structuring* of components, i.e., we study how components can be put together to form a system. Vertical structuring (i.e., notions like refinement) is not investigated in this thesis, interesting further work is to introduce other kinds of morphisms between components to capture vertical structuring (as done in [FM97] and [LF97] to formalize vertical structuring). It is worth remarking that, although the properties of components are preserved by the system, some properties of a component might be too strong, in the sense that they may restrict the behaviour of the environment, and, as a consequence, we may obtain an inconsistent specification when we put together the components. A way of dealing with this issue is using techniques like those introduced in [LF97], where the notion of *co-property* is introduced in a modal action logic without complement to formalize properties that a component is willing to have when working with an environment. Note that the tableaux method introduced in chapter 5 can be used to check the consistency of a (finite) specification.

Now that we have a notion of component, we need to have some way to put components together. We follow Goguen's ideas [BG77], where concepts coming from category theory are used to put together components of a specification. The same ideas are used in [FM91b] and [FM92], where temporal theories are used for specifying pieces of concurrent programs, and translations between them are used for specifying the relationships between these components. The idea then is to define a category where the objects are components (specifications) and the arrows are translations between them; therefore, putting together components is achieved by using the construction of colimits. Of course, some prerequisites are required. Firstly, the category

of components has to be finitely cocomplete and, secondly, the notion of deduction has to be preserved by translations (which is exactly what we proved above).

First, note that the collection of all the languages and all the translations between them form the category **Sign**. It is straightforward to see that it is really a category: identity functions define identity arrows, and composition of functions gives us the composition of translations (which straightforwardly satisfies associativity). Components and mappings between them also constitute a category.

Theorem 51. *The collection of all components* **Comp** *and all the arrows between them form the category* ***Comp***.

Proof. *The identity arrow is the identity translation, which obviously satisfies all the requisites. And the composition between mappings is just the composition of the functions which define these mappings. In addition, we must prove that* $\vdash_C Loc(id_C)$ *(where* id_C *is the identity translation). And, if we have translations* $\tau : C_1 \to C_2$ *and* $\tau' : C_2 \to C_3$*, then* $\vdash_{C_3} Loc(\tau' \circ \tau)$.

To prove $\vdash_C Loc(id)$*, we have to prove (i)* $\vdash_C \langle\gamma\rangle\top \to \langle\gamma\rangle\top$*, (ii)* $\vdash_C \varphi \to [\overline{\mathbf{U}}]\varphi$*, (iii)*$\langle\mathbf{U}\rangle\top \to \mathsf{AFDone}(\mathbf{U})$ *and (iv)*$\vdash_C \langle\gamma\rangle\varphi \to [\gamma]\varphi$*. (i), (ii) and (iv) are straightforward from the axioms. For (iii) we have that* $\vdash_C \langle\mathbf{U}\rangle\top \to \langle\mathbf{U}\rangle\mathsf{Done}(\mathbf{U})$ *by definition of* Done()*, and also* $\vdash_C [\mathbf{U}]\mathsf{Done}(\mathbf{U})$*, by the temporal axioms we have* $\vdash_C \mathsf{AN}\varphi \wedge \mathsf{EN}\varphi \to \mathsf{AF}\varphi$*, and then using* ***TempAx*** *we get* $\vdash_C \langle\mathbf{U}\rangle\top \to \mathsf{AFDone}(\mathbf{U})$.

Now, we have to prove $\vdash_{(C_3)} Loc(\tau' \circ \tau)$*. We have that:* $\vdash_{C_2} \langle\tau(\gamma)\rangle\top \to \langle\tau(\gamma) \sqcap a_1 \sqcap \cdots \sqcap a_n\rangle\top$ *(where* $a_1, \ldots, a_n$ *are the primitive action which are not images of any symbol by* τ*). Therefore, by properties of translations we have,* $\vdash_{C_3} \langle(\tau' \circ \tau)(\gamma)\rangle\top \to \langle\rangle\tau'(\overline{a_1}) \sqcap \cdots \sqcap \tau'(\overline{a_n})\rangle\top$*, where* $b_1, \ldots, b_n$ *are the primitive action of* Φ_0^3 *which are not in the image of* τ'*. Since* $[(\tau' \circ \tau)(\gamma) \sqcap \tau'(\overline{a_1}) \sqcap \cdots \sqcap \tau'(\overline{a_n})]$ *is an atomic action term in the language of* C_2*, we have that:*

$$\vdash_{C_3} \langle(\tau' \circ \tau)(\gamma) \sqcap \tau'(\overline{a_1}) \sqcap \ldots \tau'(\overline{a_n})\rangle\top \to \langle(\tau' \circ \tau)(\gamma) \sqcap \tau' \circ \tau(\overline{a_1}) \sqcap \cdots \sqcap \tau' \circ \tau(\overline{a_n}) \sqcap b_1 \sqcap \cdots \sqcap b_m\rangle\top$$

let $b'_1, \ldots, b'_k$ *be the primitive actions in the language of* C_3 *which are not translations of any primitive action of* C *through* $\tau' \circ \tau$*. Note that, if some of these* b'_j *is a translation of a primitive action* a_j *of* C_2*, then* $\vdash_{C_3} \tau'(a_j) \sqcap \overline{b'_j}$*, otherwise* b'_j *is some of the* b_i*'s. In any case we have:*

$$\vdash_{C_3} (\tau' \circ \tau)(\gamma) \sqcap \tau' \circ \tau(\overline{a_1}) \sqcap \cdots \sqcap \tau' \circ \tau(\overline{a_n}) \sqcap b_1 \sqcap \cdots \sqcap b_m \sqsubseteq (\tau' \circ \tau)(\gamma) \sqcap b'_1 \sqcap \cdots \sqcap b'_k \sqcap b_1 \sqcap \cdots \sqcap b_m.$$

Therefore we have:

$$\vdash_{C_3} \langle\tau' \circ \tau(\gamma)\rangle\top \to \langle(\tau' \circ \tau)(\gamma) \sqcap b'_1 \sqcap \cdots \sqcap b'_k \sqcap b_1 \sqcap \cdots \sqcap b_m\rangle\top$$

On the other hand, we have $\vdash_{C_2} \tau(\varphi) \to [\tau(\overline{\mathbf{U}})]\tau(\varphi)$*, and properties of translation we have:* $\vdash_{C_3} \tau' \circ \tau(\varphi) \to [\tau' \circ \tau(\overline{\mathbf{U}})]\tau' \circ \tau(\varphi)$*. We also have:* $\vdash_{C_2} \langle\tau(\gamma)\rangle\tau(\varphi) \to$

$[\tau(\gamma)]\tau(\varphi)$, *and by theorem 46 we have* $\vdash_{C_3} \langle\tau' \circ \tau(\gamma)\rangle\tau' \circ \tau(\gamma) \rightarrow [\tau' \circ \tau(\gamma)]\tau' \circ \tau(\gamma)$. *The same reasoning can be used to prove:* $\vdash_{C_3} \langle\tau' \circ \tau(\mathbf{U})\rangle\top \rightarrow \mathsf{AF}(\tau' \circ \tau(\mathsf{Done}(\mathbf{U})))$.■

The initial element of this category is the component with an empty language. We first prove that the category of signatures is finitely cocomplete (its elements are just tuples of finite sets):

Theorem 52. ***Sign*** *is finitely cocomplete.*
Proof. *A signature is a tuple* $\langle\Phi_0, \Delta_0, V_0, I_0\rangle$, *where* Δ_0, V_0, I_0 *are finite sets and* Φ_0 *is a set. The categories* $\mathbf{Set_f}$ *(of finite sets and functions between them) and* **Set** *(of small sets and functions between them) are finitely cocomplete [MM92]. The result follows from the fact that in product categories the colimits can be calculated componentwise [Mac98].* ■

The category of components is also finitely cocomplete; the forgetful functor from components to signatures reflects finite colimits (as shown for different logics in [GB92] and [FS87]; however, these logics are Institutions).

Theorem 53. *The category* ***Comp*** *is finitely cocomplete.*
Proof. *We prove that the functor* $Sign : \mathbf{Comp} \rightarrow \mathbf{Sign}$ *reflects colimits, and since* ***Sign*** *is finitely cocomplete, hence* ***Comp*** *is finitely cocomplete too.*

Suppose that $D : I \rightarrow \mathbf{Comp}$ *is a diagram in* ***Comp***. *Therefore, we have a diagram* $D' = SignD : I \rightarrow \mathbf{Sign}$. *Say* $C_i = \langle L_i, A_i, S_i\rangle$ *are the components of the diagram. Let* $\langle L, \alpha : D' \rightarrow L\rangle$ *be a colimit cocone in* ***Sign****; then we assert that*

$$C = \langle L, \bigcup_{i\in I} \alpha_i(A_i), \bigcup_{i\in I} \alpha_i(Loc(C_i)) \cup \bigcup_{i\in I} \alpha_i(S_i)\rangle$$

is a colimit object in ***Comp***. *For each component* C_i, *the translation to* C *is given by* α_i. *We prove that* α_i *is a morphism between components. We know that* $\vdash^L_{\bigcup_{i\in I}\alpha_i(A_i)} \alpha_i(A_i)$ *and* $\vdash^L_{\bigcup_{i\in I}\alpha_i(Loc(C_i))} \alpha_i(Loc(C_i))$ *and* $\vdash^L_{\bigcup_{i\in I}\alpha_i(G(L_i))} \alpha_i(G(L_i))$, *and by theorem 50 we have that*

$$\vdash_\Gamma \alpha_i(\varphi)$$

where $\Gamma = \bigcup_{i\in I}\alpha_i(A_i) \cup \bigcup_{i\in I}\alpha_i(Loc(C_i)) \cup \bigcup_{i\in I}\alpha_i(S_i)$, *for every* $\vdash_{C_i} \varphi$, *and therefore* α_i *is a morphism between components. These morphisms make the corresponding diagram commute in* ***Sign****, and therefore their extension make the corresponding diagram commute in* ***Comp***. *Now, if we have another cocone* $\langle C', \beta : C_i \rightarrow C'\rangle$, *then in* ***Sign*** *we have an unique morphism* $\psi : L \rightarrow L'$ *(where* L' *is the language of* C'*). It is straightforward to check that* ψ *can be extended to an unique* $\psi : C \rightarrow C'$, *extending the mapping of languages to mapping between formulae. This finishes the proof.* ■

Putting together components is therefore achieved by taking the colimit of a given diagram of components; an important point here is that the colimit of a given diagram of specifications preserves the separation of deontic predicates. In the next section we exhibit an example. First, we describe how the lattice of violations of a system can be approximated from the lattice of violations of each of its components.

7.2 Calculating Violations

In each component, in a given state of execution of that component, we have a set of violation predicates which are valid. This set of violation predicates can be illustrated as a partially ordered set (using a classic graphical illustration of partially ordered sets). Having a visual representation of how the violations in an execution of a program behave is useful to analyze specifications to determine what can go wrong and what to do to fix it. Each state of violation can be thought of as the set of violations which are true at that state; then the inclusions between these sets give us a diagram of degrading, whereas the opposite arrows of inclusion give us an upgrading diagram of violations. For each set of violation predicates we can calculate a corresponding diagram of sets of violations and inclusions (or opposites of inclusions), which form a category. The important point is that these diagrams illustrate how violation states are related with respect to upgrading and degrading actions and, furthermore, given a collection of violation diagrams, a colimit of them gives us a good approximation to the violation diagram of the system obtained when the components are put together, when some conditions are satisfied.

As explained above, we consider a set $\mathcal{V} = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}_3 \dots\}$ of violations. Then we can define the small category $\mathcal{C}(\mathcal{V})$ which has as objects subsets of $\mathcal{V}$ and as arrows functions between these sets. We want a particular part of this category which corresponds to upgrading and degrading actions. First, consider the category **Pos** whose objects are partially ordered sets (which are categories) and whose morphism are functors between them (order preserving mappings). This category is complete and cocomplete [AHS09]. We call a functor $F : I \to \mathcal{C}(\mathcal{V})$ a *degrading diagram*, where I is a partially ordered set such that to each arrow $i \to j$ between two elements of I, F maps it to an inclusion $F(i) \hookrightarrow F(j)$ in $\mathcal{C}(\mathcal{V})$. Now, a morphism $G : D \to D'$ between two degrading diagrams $D : I \to \mathcal{C}(\mathcal{V})$ and $D' : J \to \mathcal{C}(\mathcal{V})$ is a functor (an order preserving mapping) $F : I \to J$ between I and J and a natural transformation

$\alpha : D \stackrel{\cdot}{\rightarrow} D'F$. Naturality means that the following diagram commutes:

$$\begin{array}{ccc} i & D(i) \xrightarrow{\alpha_i} & D'F(i) \\ \uparrow & \uparrow & \uparrow \\ j & D(j) \xrightarrow{\alpha_j} & D'F(j) \end{array}$$

The category **Deg** is the category whose objects are violation diagrams and whose arrows are pairs $\langle F : I \rightarrow J, \alpha : D \stackrel{\cdot}{\rightarrow} D'F \rangle : D \rightarrow D'$ as explained above. Since the category **Pos** and the category $\mathcal{C}(\mathcal{V})$ are finitely cocomplete, then for **Deg** colimits can be calculated pointwise (see [MM92]); therefore **Deg** is finitely cocomplete.

Theorem 54. *The category* **Deg** *is finitely cocomplete.*

Proof. *Let us show how to calculate the coproducts pointwise, the coequalizers can be calculated using the same technique, and therefore the property of finitely cocompleteness follows. Let $D_1 : I_1 \rightarrow \mathcal{C}(\mathcal{V})$ and $D_2 : I_2 \rightarrow \mathcal{C}(\mathcal{V})$ be two degrading diagrams. Since* ***Pos*** *has coproducts, we have a poset $I_1 + I_2$ and morphisms $f_1 : I_1 \rightarrow I_1 + I_2$ and $f_2 : I_2 \rightarrow I_1 + I_2$. Now, let i be an element of $I_1 + I_2$, by definition of coproducts in* ***Pos****, we have some $i_1 \in I_1$ such that $f_1(i_1) = i$ or we have an element $i_2 \in I_2$ such that $f_2(i_2) = i$, but not both. In the first case, we define $D_1 + D_2(i) = D_1(i)$, otherwise $D_1 + D_2(i) = D_2(i)$. Now, let $d : i \rightarrow i'$ be an arrow in $I_1 + I_2$, then by definition of coproducts in* ***Pos****, for this arrow we some arrow $d_1 : i_1 \rightarrow i'_1$ in D_1 such that $d : f_1(i_1) \rightarrow f_1(i'_1)$, or some arrow $d_2 : i_2 \rightarrow i'_2$ in D_2 such that $d : f_1(i_2) \rightarrow f_2(i'_2)$, but not both. In the first case, we define $D_1 + D_2(d) = D_1(d_1)$, otherwise we define it: $D_1 + D_2(d) = D_2(d_2)$. This defines a functor $D_1 + D_2 : I_1 + I_2 \rightarrow \mathcal{C}(V)$, and arrows $\langle f_1, \alpha_1 : D_1 \stackrel{\cdot}{\rightarrow} (D_1 + D_2)f_1 \rangle$ and $\langle f_2, \alpha_2 : D_2 \stackrel{\cdot}{\rightarrow} (D_1 + D_2)f_2 \rangle$, where we have $(\alpha_1)_{i'}(\mathsf{v}) = \mathsf{v}$, and similarly for α_2.*

Now, for any other degrading diagram $D : I \rightarrow \mathcal{C}(\mathcal{V})$ and arrows $\langle f'_1, \alpha'_1 : D_1 \rightarrow Df_1 \rangle$ and $\langle f'_2, \alpha_2 : D_2 \stackrel{\cdot}{\rightarrow} Df'_2 \rangle$. Since $I_1 + I_2$ is a coproduct in ***Pos****, we an unique arrow $f : I_1 + I_2 \rightarrow I$ such that $f \circ f_1 = f'_1$ and $f \circ f_2 = f'_2$. Using this arrow we define the following morphism in* ***Deg****: $\langle f : I_1 + I_2 \rightarrow I, \alpha : D_1 + D_2 \stackrel{\cdot}{\rightarrow} Df \rangle$, where the components of α are defined as follows: $\alpha_i(\mathsf{v}) = (\alpha'_1)_{i'_1}(\mathsf{v})$, when $f_1(i'_1) = i$, otherwise $\alpha_i(\mathsf{v}) = (\alpha'_1)_{i'_2}(\mathsf{v})$ for the i'_2 such that $f_2(i'_2) = i$. This definition satisfies the requirement of commutativity with α'_1 and α'_2, and is unique since f is unique.* ■

On the other hand, an *upgrading diagram* is a functor $F : I^{op} \rightarrow \mathcal{C}(\mathcal{V})$, where I is a partially ordered set. Note that we take the dual of this category since upgrading diagrams are contravariant with respect to inclusion. However, the opposite of partially ordered set is a partially ordered set; therefore, an upgrading diagram is essentially the same as a degrading diagram, but we draw the arrows in the other direction (see the example below). A morphism between two upgrading diagrams $D : I^{op} \rightarrow \mathcal{C}(\mathcal{V})$

and $D' : J^{op} \rightarrow \mathcal{C}(\mathcal{V})$ is a tuple $\langle F : I^{op} \rightarrow J^{op}, \alpha : D \stackrel{\cdot}{\rightarrow} D'F \rangle$. Since the dual of a partially ordered set is a partially ordered set, we have the following property:

Theorem 55. *The category* ***Upg*** *is finitely cocomplete.*

To illustrate the idea of how colimits are built over violation diagrams, consider the degrading diagrams of figure 7.1 Note that diagrams D_2 and D_3 are isomorphic,

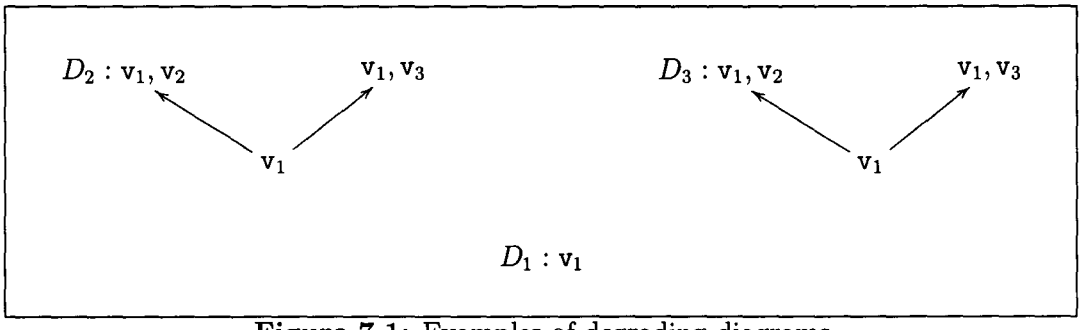


Figure 7.1: Examples of degrading diagrams

while the diagram D_1 only has $\{\mathrm{v}_1\}$ (i.e., it only has an isolated point). If we consider the following morphisms between degrading diagrams (recall that they are made up of a natural transformation and a functor): $F_1 : D_1 \rightarrow D_2$ and $F_2 : D_1 \rightarrow D_3$, which map $\{\mathrm{v}_1\}$ to the the same set in each diagram, then the pushout object obtained $(D_2 + _{D_1} D_3)$ from the above violation diagrams is shown in figure 7.2. In other words,

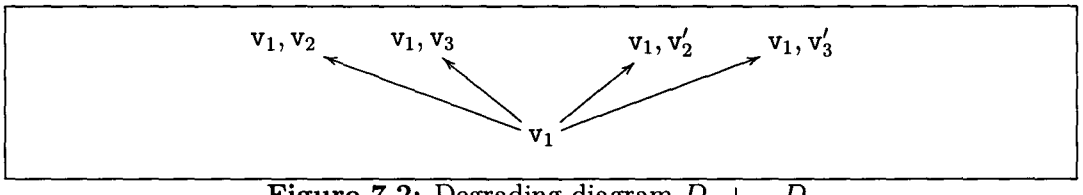


Figure 7.2: Degrading diagram $D_2 + _{D_1} D_3$

colimits join the common parts indicated by the given diagram and they separate the other parts (we create new names or violation predicates to distinguish between unrelated propositions). The same applies to upgrading diagrams.

Given a component $C = \langle L, A, S \rangle$, we can define its degrading and upgrading diagrams formally as follows. The degrading diagram is a functor $D_C : I_C \rightarrow \mathcal{C}(\mathcal{V})$, where the elements of I_C are defined as follows. If V, V' are two violation states of C and $\vdash_C V \rightarrow ([\alpha_1; \ldots; \alpha_n]V') \wedge (\langle \alpha_1; \ldots \alpha_n \rangle \top)$ is a degrading formula of C, then the pair $\langle V, V' \rangle$ is in I_C (and V, V' are elements of I_C). We also add the pairs $\langle V, V \rangle$ to satisfy reflexivity (and note that the defined relationship is transitive and antisymmetric). The functor $D_I : I_C \rightarrow \mathcal{C}(\mathcal{V})$ is defined as follows.

- For each violation state V which is an object of I_C, we have $D_I(V) = \mathcal{U}(V)$.
- For each arrow $V \rightarrow V'$ (pair) in I_C, it returns the inclusion $\mathcal{U}(V) \hookrightarrow \mathcal{U}(V')$ in $\mathcal{C}(V)$.

Similarly, we define an upgrading diagram $U_C : J_C^{op} \rightarrow \mathcal{C}(\mathcal{V})$; in this case we draw the arrows following the direction in J_C^{op}.

Note that though the potential upgrading and degrading predicates in a component are possibly infinite, the diagram is finite, since we have a finite number of violation states, which implies that we have equivalence classes of degrading and upgrading functions that represent the same transition between two violation state.

We present a simple example to illustrate these notions (a more complex example is given in section 7.3). Consider the following scenario. We have a system where we have two coolers which must maintain the low temperature of a processor. We specify the coolers as two instances of the specification shown in figure 7.3: In this

C1.B $\rightarrow \neg$v $\wedge \neg$on $\wedge \neg$high	**C11**.v $\rightarrow [\overline{on}]$v
C2.$\neg$on $\rightarrow [\overline{on}]\neg$on	**C12**.[ghigh]high
C3.[off]$\neg$on	**C13**.$\neg$high $\rightarrow [\overline{ghigh}]\neg$high
C4.on $\rightarrow [\overline{off}]$on	**C14**.[on]$\neg$high
C5.[on]on	**C15**.high $\rightarrow [\overline{on}]$high
C6.$\neg$high $\rightarrow$ P(U)	**C16**.$\langle$ghigh$\rangle\top$
C7.high $\rightarrow$ O(on)	**C17**.$\langle$off$\rangle\top$
C8.F($\overline{on}$) $\rightarrow [\overline{on}]$v	**C18**.$\langle$on$\rangle\top$
C9.v $\rightarrow$ [on]$\neg$v	**C19**.off $\sqcap$ on $=_{act} \emptyset$
C10.P(on)	

Figure 7.3: Specification of a cooler

specification we have actions: on (to turn on the cooler), ghigh (this action indicates when the temperature of the cooler is high) and off (this action turns the cooler off). Most of the axioms specify the behaviour of these actions, we can highlight the following axioms. Axiom **C6** says that, if the temperature is low, then any action is allowed. Axiom **C7** says that, if the temperature is high, then the cooler ought to be on. Axioms **C8** indicates when a violation arises. On the other hand, axiom **C9** says that on is a recovery action for violation v. Axioms **C15-C16** say that the actions ghigh, on and off can always be executed. Finally, axiom **C19** says that actions on and off are disjoint, and axiom **C10** says that the action on is always allowed (which implies that it never causes a violation).

Now suppose that, perhaps since we want some redundancy in the system, we are interested in having two coolers; a specification of two coolers can be obtained by taking the coproduct $\mathbf{C} + \mathbf{C}$. In this case, different variables, actions and deontic predicates are created to distinguish the two instances of the same specification. see figure 7.4. Let us investigate the degrading diagram of the component $\mathbf{C}$. First, we

$\mathbf{C1}_i.\mathsf{B} \rightarrow \neg \mathtt{v}_i \wedge \neg \mathtt{on}_i \wedge \neg \mathtt{high}_i$	$\mathbf{C11}_i.\mathtt{v}_i \rightarrow [\overline{\mathtt{on}_i}]\mathtt{v}_i$
$\mathbf{C2}_i.\neg \mathtt{on}_i \rightarrow [\overline{\mathtt{on}_i}]\neg \mathtt{on}_i$	$\mathbf{C12}_i.[\mathtt{ghigh}_i]\mathtt{high}_i$
$\mathbf{C3}_i.[\mathtt{off}_i]\neg \mathtt{on}_i$	$\mathbf{C13}_i.\neg \mathtt{high}_i \rightarrow [\overline{\mathtt{ghigh}_i}]\neg \mathtt{high}_i$
$\mathbf{C4}_i.\mathtt{on}_i \rightarrow [\overline{\mathtt{off}_i}]\mathtt{on}_i$	$\mathbf{C14}_i.[\mathtt{on}_i]\neg \mathtt{high}_i$
$\mathbf{C5}_i.[\mathtt{on}_i]\mathtt{on}_i$	$\mathbf{C15}_i.\mathtt{high}_i \rightarrow [\overline{\mathtt{on}_i}]\mathtt{high}_i$
$\mathbf{C6}_i.\neg \mathtt{high}_i \rightarrow \mathsf{P}^i(\mathsf{U})$	$\mathbf{C16}_i.\langle \mathtt{ghigh}_i \rangle \top$
$\mathbf{C7}_i.\mathtt{high}_i \rightarrow \mathsf{O}^i(\mathtt{on}_i)$	$\mathbf{C17}_i.\langle \mathtt{off}_i \rangle \top$
$\mathbf{C8}_i.\mathsf{F}(\overline{\mathtt{on}_i}) \rightarrow [\overline{\mathtt{on}_i}]\mathtt{v}_i$	$\mathbf{C18}_i.\langle \mathtt{on}_i \rangle \top$
$\mathbf{C9}_i.\mathtt{v}_i \rightarrow [\mathtt{on}_i]\neg v_i$	$\mathbf{C19}_i.\mathtt{off}_i \sqcap \mathtt{on}_i =_{act} \emptyset$
$\mathbf{C10}_i.\mathsf{P}^i(\mathtt{on}_i)$	where $i = 1, 2$

Figure 7.4: Coproduct $\mathbf{C} + \mathbf{C}$

can prove $\vdash_{\mathbf{C}} \neg \mathtt{v} \rightarrow [\mathtt{ghigh}; \overline{\mathtt{on}}]\mathtt{v}$.

1.	$\vdash_{\mathbf{C}} [\mathtt{ghigh}]\mathtt{high}$	**C11**
2.	$\vdash_{\mathbf{C}} \mathtt{high} \rightarrow \mathsf{O}(\mathtt{on})$	**C7**
3.	$\vdash_{\mathbf{C}} \mathsf{O}(\overline{\mathtt{on}}) \rightarrow \mathsf{F}(\overline{\mathtt{on}})$	Def.O()
4.	$\vdash_{\mathbf{C}} \mathsf{F}(\overline{\mathtt{on}}) \rightarrow [\overline{\mathtt{on}}]\mathtt{v}$	**C5**
5.	$\vdash_{\mathbf{C}} \mathtt{high} \rightarrow [\overline{\mathtt{on}}]\mathtt{v}$	PL,2,3,4
6.	$\vdash_{\mathbf{C}} [\mathtt{ghigh}][\overline{\mathtt{on}}]\mathtt{v}$	ML, 5, 1
7.	$\vdash_{\mathbf{C}} [\mathtt{ghigh}; \overline{\mathtt{on}}]\mathtt{v}$	Def.;

That is, the degrading diagram of specification $\mathbf{C}$ is the diagram (a) shown in figure 7.5. Meanwhile, the degrading diagram of the coproduct $\mathbf{C} + \mathbf{C}$ is shown in figure 7.5 (b). To prove that this is correct, we have to prove that there exist action expressions α_1, α_2, α_3 and α_4 which make the following statements true:

1. $\vdash_{\mathbf{C}+\mathbf{C}} \neg \mathtt{v}_1 \wedge \neg \mathtt{v}_2 \rightarrow ([\alpha_1](\mathtt{v}_1 \wedge \neg \mathtt{v}_2) \wedge \langle \alpha_1 \rangle \top)$.
2. $\vdash_{\mathbf{C}+\mathbf{C}} \neg \mathtt{v}_1 \wedge \mathtt{v}_2 \rightarrow ([\alpha_2](\neg \mathtt{v}_1 \wedge \neg \mathtt{v}_2) \wedge \langle \alpha_2 \rangle \top)$.
3. $\vdash_{\mathbf{C}+\mathbf{C}} \mathtt{v}_1 \wedge \neg \mathtt{v}_2 \rightarrow ([\alpha_3](\mathtt{v}_1 \wedge \mathtt{v}_2) \wedge \langle \alpha_3 \rangle \top)$.
4. $\vdash_{\mathbf{C}+\mathbf{C}} \neg \mathtt{v}_1 \wedge \mathtt{v}_2 \rightarrow ([\alpha_4](\mathtt{v}_1 \wedge \mathtt{v}_2) \wedge \langle \alpha_3 \rangle \top)$.

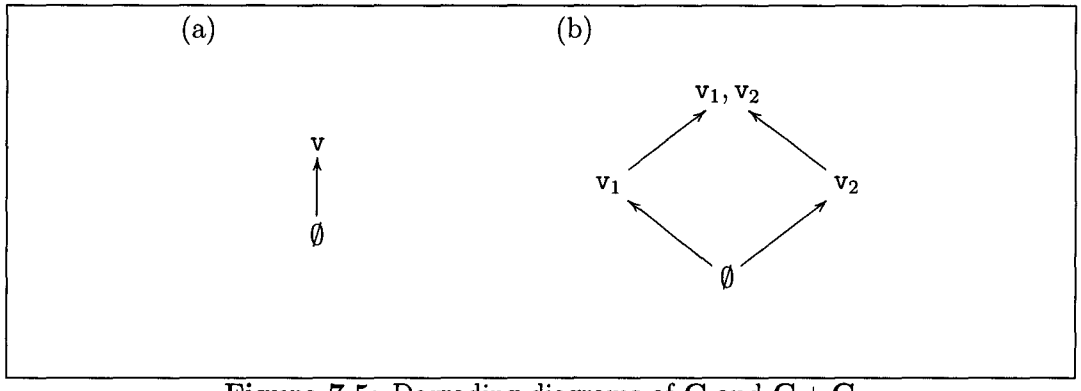


Figure 7.5: Degrading diagrams of $\mathbf{C}$ and $\mathbf{C}+\mathbf{C}$

For example, for item 1 we can prove:

$$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_1 \wedge \neg v_2 \rightarrow [\mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1]\neg v_2 \wedge v_1$$

as follows:

1.	$\vdash_{\mathbf{C}} \neg v \rightarrow [\mathtt{ghigh}; \overline{\mathtt{on}}]v$	See above.
2.	$\vdash_{\mathbf{C}+\mathbf{C}} \tau_1(\neg v) \rightarrow [\tau_1(\mathtt{ghigh}); \tau_1(\overline{\mathtt{on}})]\tau_1(v)$	Theorem 50
3.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_1 \rightarrow [\mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1]v_1$	Def.τ_1
4.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_2 \rightarrow [\mathtt{ghigh} \sqcup \mathtt{off}_1 \sqcup \mathtt{on}_1]\neg v_2$	Loc.
5.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_2 \rightarrow [\mathtt{ghigh}_1]\neg v_2$	DPL 4
6.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_2 \rightarrow [\mathtt{ghigh}_1 \sqcup \mathtt{off}_1]\neg v_2$	DPL 4
7.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_2 \rightarrow [\mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1]\neg v_2$	DPL, 5, 6
8.	$\vdash_{\mathbf{C}+\mathbf{C}} \neg v_1 \wedge \neg v_2 \rightarrow [\mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1]\neg v_2 \wedge v_1$	DPL, 4, 7

Note that, in this proof, we have used the theorem proven in component $\mathbf{C}$ to prove the statement in the specification $\mathbf{C}+\mathbf{C}$; this shows the way in which the theorems proven in the components can be reused to prove properties about the entire system. We can also prove: $\vdash_{\mathbf{C}+\mathbf{C}} \neg v_1 \wedge \neg v_2 \rightarrow \langle \mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1\rangle \top$, as follows:

1.	$\vdash_{\mathbf{C}+\mathbf{C}} \langle \mathtt{ghigh}_1\rangle \top$	$\mathbf{C16}_1$
2.	$\vdash_{\mathbf{C}+\mathbf{C}} \langle \mathtt{off}_1\rangle \top$	$\mathbf{C17}_1$
3.	$\vdash_{\mathbf{C}+\mathbf{C}} \langle \mathtt{ghigh}_1 \sqcup \mathtt{off}_1\rangle \top$	DPL, 1, 2
4.	$\vdash_{\mathbf{C}+\mathbf{C}} [\mathtt{ghigh}_1]\langle \mathtt{ghigh}_1 \sqcup \mathtt{off}_1\rangle \top$	**GN**, 3
5.	$\vdash_{\mathbf{C}+\mathbf{C}} \langle \mathtt{ghigh}_1\rangle\langle \mathtt{ghigh}_1 \sqcup \mathtt{off}_1\rangle \top$	DPL, 3, 4
6.	$\vdash_{\mathbf{C}+\mathbf{C}} \langle \mathtt{ghigh}_1; \mathtt{ghigh}_1 \sqcup \mathtt{off}_1\rangle \top$	Def.;

and therefore we obtain the arrow $\emptyset \rightarrow v_1$ in the degrading diagram. In the same way, the remaining items in the list can be proven, and therefore, we obtain the degrading diagram of figure 7.5 (b).

In practice we can coordinate the different instances of the specification via some actions. However, in this case (depending on which actions or variables we coordinate across the specifications) the independence of the degrading and upgrading diagrams of each component might not be preserved when putting the components together. An obvious example is to coordinate the two coolers via actions `ghigh` and `on` (i.e., they have the same sensor and the same button turning on both coolers). In this case, both sensors go into violation at the same time. But note that, if we coordinate the two coolers only for action `on`, then the independence of the degrading or upgrading diagram is preserved. This can be proven using the *GGG* condition and the fact that the action `on` is always allowed in both components, i.e., this action never introduces a violation.

In the following, we investigate some scenarios where we can ensure some independence between the violations in the components. For the following theorems, we need to define formally what it means for two components to be coordinated via a variable or an action. Given a diagram $D : I \to \mathbf{Comp}$ with components $C_1, \ldots, C_n$, and colimit $\langle C, \tau_i : C_i \to C \rangle$, we say that two components C_i and C_j coordinate via an action c of C if we have an action c_i of C_i and an action c_j of C_j such that $\tau_i(c_i) = c = \tau_j(c_j)$, and we say that C_i and C_j coordinate via a variable p of C if there are variables p_i in C_i and p_j in C_j such that $\tau_i(p_i) = p = \tau(p_j)$.

Our first theorem says that, when we have two components which do not coordinate via any action, then the degrading diagrams of each component are respected by the degrading diagram of the specification obtained when we put both components together.

Theorem 56. *Consider two components C_1 and C_2, with degrading diagrams $D_{C_1} : I_{C_1} \to \mathcal{C}(V)$ and $D_{C_2} : I_{C_2} \to \mathcal{C}(V)$, respectively. Let $D_{C_1+C_2} : I_{C_1+C_2} \to \mathcal{C}(V)$ be the degrading diagram of $C_1 + C_2$ (the coproduct of C_1 and C_2), then there is a morphism $\langle F, \alpha \rangle : D_{C_1} + D_{C_2} \to D_{C_1+C_2}$, such that all the components of α are iso and F is faithful.*

Proof. *We prove that, if we have an arrow $V \to V'$ in the coproduct of the violation diagrams, then we have a degrading action in the coproduct of the components, which identifies this arrow. From here, we can use the identities to map $D_{C_1} + D_{C_2}$ to $D_{C_1+C_2}$. Suppose that we have $V \to V'$ in $D_{C_1}+D_{C_2}$, then, by properties of coproducts, this arrow belongs to D_1 or D_2; if $V \to V'$ belongs to D_1, then we have that $\vdash_{C_1} (V \to [\alpha_1; \ldots \alpha_n]V' \wedge \langle \alpha_1; \ldots; \alpha_n \rangle \top$, but then we have:*

$$\vdash_{C_1+C_2} \tau_1(V) \to [\tau_1(\alpha_1); \ldots; \tau_1(\alpha_n)]\tau_1(V') \wedge \langle \tau_1(\alpha_1); \ldots \tau_1(\alpha_n) \rangle \top.$$

But note that $\tau(V)$ is not necessarily a violation state of $C_1 + C_2$, since the violations of C_2 are not considered there. But since C_1 and C_2 do not coordinate via any action, we know that $\tau_1(V) \wedge \tau_2(\neg \boldsymbol{V}_{C_2})$, is a violation state of $C_1 + C_2$. From this, using the

properties of locality, we obtain:

$$\vdash_{C_1+C_2} \tau_1(V) \wedge \neg\tau_2(\boldsymbol{V}_{C_2}) \rightarrow \langle \tau_1(\alpha_1) \sqcap \overline{\tau_2(\mathbf{U})}; \ldots; \tau_1(\alpha_n) \sqcap \overline{\tau_2(\mathbf{U})} \rangle \tau_1(V') \wedge \neg\tau_2(\boldsymbol{V}_{C_2})$$

and

$$\vdash_{C_1+C_2} \tau_1(V) \wedge \neg\tau_2(\boldsymbol{V}_{C_2}) \rightarrow [\tau_1(\alpha_1) \sqcap \overline{\tau_2(\mathbf{U})}; \ldots; \tau_1(\alpha_n) \sqcap \overline{\tau_2(\mathbf{U})}] \tau_1(V') \wedge \neg\tau_2(\boldsymbol{V}_{C_2})$$

and therefore the degrading transition $V \rightarrow V'$ belongs to the degrading diagram of $C_1 + C_2$. ■

It is important to analyze in detail what this theorem says. If we have two components and we put them together without coordinating them via any action (they are totally disjoint), then the violation state of one component does not affect the other component and viceversa. The isomorphism of the components of the natural transformation indicate that the number of violations is preserved in each violation state of the components, and the faithfulness of the functor indicates that the "shape" of the degrading or upgrading diagrams of each component is preserved by the degrading diagram of the entire system. Obviously, this result can be extended for a more general setting, where we have many components which do not coordinate via any actions.

We can generalize this result to situations where the components with violations do not coordinate via any action (see below the example of the diarrheic philosophers where philosophers do not coordinate via any actions; they only coordinate with forks, but forks do not have violations).

Theorem 57. *Given a finite diagram $D : I \rightarrow$ **Comp** with components $C_1, \ldots, C_n$, let $\langle C, \tau_i : C_i \rightarrow C \rangle$ be a colimit of D. If $C_{i_1}, \ldots, C_{i_k}$ are all the components with violations in the language, and let $D_{i_1}, \ldots, D_{i_k}$ be their degrading diagrams. If components $C_{i_1}, \ldots, C_{i_k}$ do not coordinate via any action nor variable, then there is a morphism $\langle F, \alpha \rangle : D_{i_1} + \cdots + D_{i_k} \rightarrow D_C$ where the components of α are iso and F is faithfull.*

Proof. *The diagram $D_{i_1} + \cdots + D_{i_k}$ is the coproduct of the diagrams $D_{i_1}, \ldots, D_{i_k}$; we show that each arrow $V \rightarrow V'$ is also an arrow of D_C, and therefore, the morphism between $D_{i_1} + \cdots + D_{i_k}$ and D_C is given by identities.*

Let $V \rightarrow V'$ be an arrow in some D_{i_j}; then we have:

$$\vdash_{C_{i_j}} V \rightarrow [\alpha_1; \ldots; \alpha_n] V' \wedge \langle \alpha_1; \ldots; \alpha_n \rangle \top.$$

Now , we have that:

$$\vdash_C \tau_{i_j}(V) \rightarrow [\tau_{i_j}(\alpha_1); \ldots; \tau_{i_j}(\alpha_n)] \tau_{i_j}(V') \wedge \langle \tau_{i_j}(\alpha_1); \ldots; \tau_{i_j}(\alpha_n) \rangle \top$$

Note that $\tau_{i_j}(V)$ is not necessarily a violation state in C. Let $\mathsf{v}_1, \ldots, \mathsf{v}_n$ be the violations which do not appear in V (these are translations of violations in other components), then $\tau_{i_j} \wedge \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m$ is a violation state of C. Let $a_1, \ldots, a_q$ be the actions which are translations of actions of C_{i_j}. Since C_{i_j} does not coordinate via any action nor variable with the other of components with violations, by locality we have:

$$\vdash_C \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m \rightarrow [\overline{a_1} \sqcap \cdots \sqcap \overline{a_q}] \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m$$

and therefore by properties of DPL we have:

$$\vdash_C \tau_{i_j}(V) \wedge \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m \rightarrow [\tau_{i_j}(\alpha_1) \sqcap \overline{a_1}; \ldots; \tau_{i_j}(\alpha_n); \overline{a_q}] V' \wedge \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m$$

and by the independence axioms we have:

$$\vdash \tau_{i_j}(V) \wedge \neg \mathsf{v}_1 \wedge \cdots \wedge \neg \mathsf{v}_m \rightarrow \langle \tau_{i_j}(\alpha_1) \sqcap \overline{a_1}; \ldots; \tau_{i_j}(\alpha_n); \overline{a_q} \rangle \top$$

which shows that we have an arrow $V \rightarrow V'$ in D_C. The result follows. ■

Note that the theorems above are also valid for upgrading diagrams; we only need to change the direction of the arrows in the proofs.

However, in practice components usually coordinate via some actions, and therefore it is important to have some result which can be applied to wider cases where components interact in some way. Note that the strong version of the *GGG* predicate says that an execution of an allowed action cannot introduce a violation into a violation state. Then, if we coordinate two components on actions which are always allowed (i.e., they are safe) and we have the axioms $SG(L)$ in the components, then we can ensure that no violations are introduced when we execute a recovery (or a degrading) action on one of the components. We need some extra notation to present these results. Given a language L, we say that $\mathbf{P}(\alpha)$ (α is in general allowed) iff $\mathsf{P}^1(\alpha) \wedge \cdots \wedge \mathsf{P}^n(\alpha)$ where $\{1, \ldots, n\}$ are the permission indexes of L, and we say that α is safe in a component C if $\vdash_C \mathbf{P}(\alpha)$.

Theorem 58. *Given a diagram $C_1 \leftarrow C \rightarrow C_2$, and the pushout of this diagram, denoted by $C_1 +_C C_2$, if (i) C_1 and C_2 do not coordinate via any action, (ii) the actions in C when translated into actions of $C_1 +_C C_2$, say $c_1, \ldots, c_n$, are safe in $C_1 +_C C_2$ (i.e., $\vdash_{C_1 +_C C_2} \mathbf{P}(c_i)$ for every i) and (iii) in the system axioms of C_1 (respectively, C_2) we have $SG(C_1)$ (respectively, $SG(C_2)$), then there is a morphism $\langle F, \alpha \rangle : U_{C_1} + U_{C_2} \rightarrow U_{C_1 +_C C_2}$, such that all the components of α are iso and F is faithful.*

Proof. *The proof is similar to the proof of theorem 56. Suppose that we have an upgrading transition $V \rightarrow V'$ in $U_{C_1} + U_{C_2}$. For the case that $V \rightarrow V'$ belongs to U_{C_1},*

we proceed as follows: let $a_1, \ldots, a_k$ *be the primitive actions of* C_1 *and* $b_1, \ldots, b_m$ *be the primitive actions of* C_2*. Let us use* $C_1 - C_2$ *for the expression:*

$$\bigsqcup_{\tau_2(b_i) \notin \{\tau_1(a_1), \ldots, \tau_1(a_n)\}} \tau_2(b_i).$$

and similarly for $C_2 - C_1$*. We have that:*

$$\vdash_{C_1 +_C C_2} \tau_1(V) \rightarrow [\tau_1(\alpha_1); \ldots; \tau_1(\alpha_n)]\tau(V') \wedge \langle \tau_1(\alpha_1); \ldots; \tau_1(\alpha_n) \rangle \top.$$

Note that $\tau_1(V)$ *and* $\tau_1(V')$ *are not necessarily violation states of* $C_1 +_C C_2$*. Now let* $\mathtt{v}_1, \ldots, \mathtt{v}_t$ *be the violation predicates which do not appear in* $\tau_1(V)$*. Obviously, these violation predicates are translations of violation predicates of component* C_2*. Now by locality we have:*

$$\vdash_{C_1 +_C C_2} \neg \mathtt{v}_1 \wedge \ldots \neg \mathtt{v}_t \rightarrow [\overline{\tau_2(\mathbf{U})}] \neg \mathtt{v}_1 \wedge \cdots \wedge \neg \mathtt{v}_t.$$

Also we know that:

$$\vdash_{C_1 +_C C_2} \neg \mathtt{v}_1 \wedge \ldots \neg \mathtt{v}_t \rightarrow [c_1 \sqcup \cdots \sqcup c_n] \neg \mathtt{v}_1 \wedge \cdots \wedge \neg \mathtt{v}_t.$$

since $c_1, \ldots, c_n$ *are safe actions by hypothesis and by the translations of the axioms* $SG(C_2)$ *and therefore* $\mathbf{P}(c_i)$ *for any* i*. Now using the formulae above and the properties of the logic we get:*

$$\vdash_{C_1 +_C C_2} \neg \mathtt{v}_1 \wedge \ldots \neg \mathtt{v}_t \rightarrow [(c_1 \sqcup \cdots \sqcup c_n) \sqcup \overline{\tau_2(\mathbf{U})}] \neg \mathtt{v}_1 \wedge \cdots \wedge \neg \mathtt{v}_t$$

and $(c_1 \sqcup \cdots \sqcup c_n) \sqcup \overline{\tau_2(\mathbf{U})}$ *is just* $\overline{C_2 - C_1}$*.*

$$\vdash_{C_1 +_C C_2} \tau_1(V) \wedge \neg \mathtt{v}_1 \wedge \cdots \wedge \neg \mathtt{v}_t \rightarrow [\alpha_1 \sqcap \overline{C_2 - C_1}; \ldots; \alpha_n \sqcap \overline{C_2 - C_1}] \tau_1(V') \wedge \neg \mathtt{v}_1 \wedge \cdots \wedge \neg \mathtt{v}_t$$

We find here part of the formula that we must prove. For the other part we have:

$$\vdash_{C_1 +_C C_2} \tau_1(V) \rightarrow \langle \alpha_1; \ldots; \alpha_n \rangle \top$$

Consider that $C_2 - C_1$ *are exactly the choice of the action which belongs to* $C_1 +_C C_2$ *and do not belong to the translation of primitive actions in* C_1*, and therefore by independence we get:*

$$\vdash_{C_1 +_C C_2} \tau_1(V) \rightarrow \langle \alpha_1 \sqcap \overline{C_2 - C_1}; \ldots; \alpha_n \sqcap \overline{C_2 - C_1} \rangle \top.$$

The case that $V' \rightarrow V$ *in* U_{C_2} *uses a similar argument. This finishes the proof.* ■

This theorem can be expressed by means of a slogan:

Coordination on safe actions is safe.

This property can be generalized when we have a finite number of components and they only interact (or coordinate) by means of safe actions. Note that in the theorem we require that component C does not have any violations, i.e., in other words, we require that components C_1 and C_2 do not coordinate via any violation. In the case that components coordinate via violations, the independence between the violation diagrams of each component are not respected any longer; it is possible that in this case the violation diagram of the system can be approximated using the colimits of the violation diagrams of the components. We do not investigate this in this thesis. It is worth remarking that, in a concurrent setting, we want to keep the components as independent as possible, and coordination by means of violation constants may not be a good practice, to the extent that this is not strictly necessary.

7.3 Revisiting the Diarrheic Philosophers

Now, we show an example to illustrate the application of these theorems in practice. We revisit the example of the diarrheic philosophers (which was introduced without the notion of components in chapter 4). Here we follow the main ideas introduced in [FM92] to modularize the design; note that the design obtained by modularization is clearer that the original one. First, let us consider the specification of a fork. The actions of a fork are:

$$\{\texttt{l.up}, \texttt{l.down}, \texttt{r.up}, \texttt{r.down}\}$$

and the predicates are

$$\{\texttt{l.up?}, \texttt{r.up?}\}$$

Intuitively, we have two ports by means of which we can use the forks; one is for the left philosopher and the other one is for the right philosopher. Note that this implies that the philosophers do not coordinate directly via any action (also note that these actions are mutually disjoint), this allows us to use theorem 57 to prove that the recovery of one philosopher does not cause any violations in the other philosophers. The axioms of the fork are shown in figure 7.6. The axioms specify the behaviour of a fork. As explained above, a fork can be held onto by the philosopher on the left or by the philosopher on the right. Therefore, we have two actions that reflect this action: l.up and r.up. Obviously they are disjoint (as stated by axiom $\mathbf{f_6}$), meaning that only one of the philosophers can be holding onto the fork. The rest of the specification describes what is the behaviour of each action.

In chapter 4 we have described a complete specification of the diarrheic philosophers. The specification of a philosopher that we provide below is slightly different

XFork:

$f_1.B \rightarrow \neg 1.up? \wedge \neg r.up?$
$f_2.[1.up]1.up?$
$f_3.[1.down]\neg 1.up?$
$f_4.\neg 1.up? \rightarrow \boxed{1.up}\neg 1.up?$
$f_5.1.up? \rightarrow \boxed{1.down}1.up?$
$f_6.\neg(1.up? \wedge r.up?)$
$f_7.\neg 1.up? \rightarrow [1.down]\bot$
$f_8.[r.up]r.up?$
$f_9.[r.down]\neg r.up?$
$f_{10}.\neg r.up? \rightarrow \boxed{r.up}\neg r.up?$
$f_{11}.r.up? \rightarrow \boxed{r.down}r.up?$
$f_{12}.\neg r.up? \rightarrow [r.down]\bot$
$f_{14}.1.up? \rightarrow \langle 1.down\rangle\top$
$f_{15}.r.up? \rightarrow \langle r.down\rangle\top$

Figure 7.6: XFork specification

to the one given earlier, in part since we abstract from some details to focus on the specification of the violations and their properties. Note that here we take a simpler approach: a philosopher can take both forks or none. The actions of the specification are the following.

$$\{\mathtt{getthk}, \mathtt{getbad}, \mathtt{gethungry}, up_L, up_R\}$$

and they are intended to denote the same actions as chapter 4. We have the following propositions:

$$\{has_L, has_R, \mathtt{thk}, \mathtt{eating}, \mathtt{hungry}, \mathtt{bath}, has_L, has_R\}$$

and two violations $\{v_1, v_2\}$. The set of axioms of this specification is shown in figure 7.7. We consider the $SG(\mathbf{Phil})$ predicates on the system axioms, i.e.: $\neg v_i \wedge P^1(\alpha) \rightarrow$

Phil :

$p_1 : B \rightarrow \neg v_1 \wedge \neg v_2 \wedge \mathtt{thk} \wedge \neg \mathtt{hungry} \wedge \neg \mathtt{bath}$
$p_2 : \mathtt{thk} \veebar \mathtt{hungry} \veebar \mathtt{eating} \veebar \mathtt{sick}$
$p_3 : \mathtt{eating} \leftrightarrow has_L \wedge has_R \wedge \neg \mathtt{sick}$
$p_4 : \neg \mathtt{hungry} \rightarrow AF\mathtt{hungry}$
$p_5 : \neg \mathtt{eating} \rightarrow P^1(\mathbf{U})$
$p_6 : \mathtt{eating} \rightarrow O^1(down_L \sqcap down_R)$
$p_7 : F^1(\boxed{down_L}) \rightarrow \boxed{down_L}v_1$
$p_8 : F^1(\boxed{down_R}) \rightarrow \boxed{down_R}v_2$
$p_9 : v_1 \rightarrow \boxed{down_L}v_1$
$p_{10} : v_2 \rightarrow \boxed{down_R}v_2$
$p_{11} : v_1 \rightarrow [down_L]\neg v_1$
$p_{12} : v_2 \rightarrow [down_R]\neg v_2$
$p_{13} : [\mathtt{getthk}]\mathtt{thk}$
$p_{15} : \neg \mathtt{thk} \rightarrow [\mathtt{getthk}]\neg \mathtt{thk}$
$p_{16} : [\mathtt{getbad}]\mathtt{bath}$
$p_{17} : \neg \mathtt{bath} \rightarrow \boxed{\mathtt{getbad}}\neg \mathtt{bath}$
$p_{18} : [\mathtt{gethungry}]\mathtt{hungry}$
$p_{19} : \neg \mathtt{hungry} \rightarrow [\mathtt{gethungry}]\neg \mathtt{hungry}$
$p_{20} : \mathtt{thk} \rightarrow down_1 \wedge down_R$
$p_{21} : \mathtt{hungry} \rightarrow down_L \wedge down_R$
$p_{22} : \mathtt{hungry} \wedge \langle up_L \sqcup up_R\rangle\top \rightarrow AN\mathtt{eating} \vee AN\mathtt{hungry}$
$p_{23} : \mathtt{bad} \rightarrow \boxed{\mathtt{getthk}}\mathtt{bad}$
$p_{24} : \mathtt{eating} \rightarrow AN(\mathtt{thk} \vee \mathtt{bad})$

Figure 7.7: Specification of a philosopher

$[\alpha]\neg \mathtt{v}_i$, for every i are in the system axioms of **Phil**. Most of the axioms have already been introduced in the example of chapter 4. We dicuss the remaining axioms. Axiom $\mathbf{p_4}$ says that a philosopher which is not hungry will become hungry in the future; axiom $\mathbf{p_5}$ states that, when the philosopher is not eating, then everything is allowed. Axioms $\mathbf{p_7}$-$\mathbf{p_{12}}$ specify how the violations occur in a given execution of this specification and which are the recovery actions. Note that, in axiom $\mathbf{p_6}$, we say that, if a philosopher is eating, then it will be obliged to return both forks. In the same way that we did in chapter 3, we simplify the problem by requiring that philosophers can only eat for a unit of time (axiom $\mathbf{p_{24}}$); the amount of time that the philosophers eat is not important for our current purposes. However, note that the specification can be extended establishing that a philosopher can only eat a finite amount of time (for this we need to have a component specifying a clock). In this case we will have a new violation in case the philosopher continues eating beyond the prescribed limit. (We can consider this a preferable violation to going to the bathroom!)

Now we focus on the analysis of violations. We suppose that some properties like $\vdash_{\mathbf{Phil}} \mathtt{v}_1 \rightarrow \mathtt{has}_{\mathtt{L}}$ (if the philosopher is in violation $\mathtt{v}_1$, then he has the left fork) and $\vdash_{\mathbf{Phil}} \mathtt{v}_1 \rightarrow \neg\mathtt{eating}$ (if he is in a violation, then he is not eating) can be proven from the specification. (We proved these properties for the specification of chapter 4.)

Suppose that we want to obtain the specification of a unique philosopher with two forks. We need to define some way of connecting the different components. With this goal in mind, we define a component **Chan** which only has actions $\{\mathtt{port}_1, \mathtt{port}_2\}$ with no predicates and no violations. Using channels, we can connect the forks with the philosopher taking the colimit of the diagram shown in figure 7.8. The components

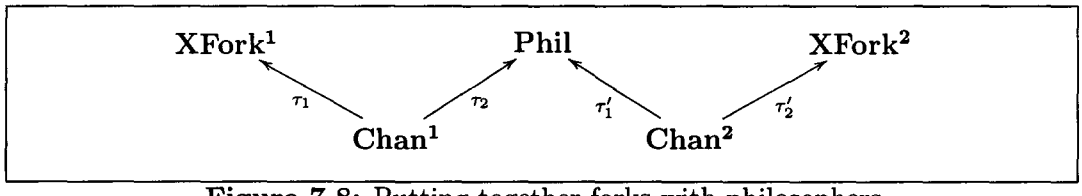
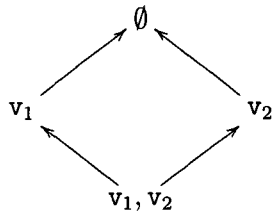


Figure 7.8: Putting together forks with philosophers

$\mathbf{XFork^1}$ and $\mathbf{XFork^2}$ are "instances" of the specification **XFork** (i.e., they obtained from **XFork** renaming the symbols with the subindex i), and $\mathbf{Chan^1}$ and $\mathbf{Chan^2}$ are "instances" of **Chan**. Here τ_1 : **Chan** $\rightarrow$ **XFork** maps $\mathtt{port}_1^1 \mapsto \mathtt{lup}$ and $\mathtt{port}_2^1 \mapsto \mathtt{ldown}$, whereas τ_2 : **Chan** $\rightarrow$ **Phil** maps $\mathtt{port}_1^1 \mapsto \mathtt{up}_{\mathtt{L}}$ and $\mathtt{port}_2^1 \mapsto \mathtt{down}_{\mathtt{L}}$ and similarly for τ_1' and τ_2'. In other words, these morphisms connect the right and the left fork with the philosopher. Let us call the colimit object of this specification **FPhil**, where the morphism f_1 : **XFork** $\rightarrow$ **FPhil**, f_2 : **XFork** $\rightarrow$ **FPhil**, p_1 : **Phil** $\rightarrow$ **FPhil**, c_1 : **Chan** $\rightarrow$ **FPhil** and c_2 : **Chan** $\rightarrow$ **FPhil**, are the required morphisms from the base of the cocone to the colimit object. Now, the upgrading diagram of **FPhil** is as shown in figure 7.9. The formulae at the right in this figure



- $\vdash_{\mathbf{FPhil}} v_1 \wedge v_2 \rightarrow [\mathtt{down}_L]\neg v_1 \wedge v_2$
- $\vdash_{\mathbf{FPhil}} v_1 \wedge v_2 \rightarrow [\mathtt{down}_R]v_1 \wedge \neg v_2$
- $\vdash_{\mathbf{FPhil}} v_1 \wedge \neg v_2 \rightarrow [\mathtt{down}_L]\neg v_1 \wedge \neg v_2$
- $\vdash_{\mathbf{FPhil}} \neg v_1 \wedge v_2 \rightarrow [\mathtt{down}_R]\neg v_1 \wedge \neg v_2$

Figure 7.9: Upgrading diagram of **FPhil**

indicate the properties that we need to prove to show that this diagram is correct. Intuitively, the worst state is when a philosopher is in the bathroom with both forks. He can recover from this scenario by putting one of the forks down, and then he can go into a normal state by putting the other fork down. To prove the arrow from v_1 to $\emptyset$, we proceed as follows.

1.	$\vdash_{\mathbf{FPhil}} v_1 \rightarrow \neg\mathtt{eating}$	Property of **FPhil**
2.	$\vdash_{\mathbf{FPhil}} \neg\mathtt{eating} \rightarrow \mathsf{P}^1(\mathbf{U})$	$\mathbf{p_5}$
3.	$\vdash_{\mathbf{FPhil}} \mathsf{P}^1(\mathbf{U}) \rightarrow \mathsf{P}^1(\mathtt{down}_L)$	DPL
4.	$\vdash_{\mathbf{FPhil}} \neg v_2 \wedge \mathsf{P}^1(\mathtt{down}_L) \rightarrow [\mathtt{down}_L]\neg v_2$	SG(**Phil**)
5.	$\vdash_{\mathbf{FPhil}} v_1 \wedge \neg v_2 \rightarrow [\mathtt{down}_L]\neg v_2$	ML, 1, 2, 4
6.	$\vdash_{\mathbf{FPhil}} v_1 \wedge \neg v_2 \rightarrow [\mathtt{down}_L]\neg v_1 \wedge \neg v_2$	PL, $\mathbf{p_{11}}$, 5
7.	$\vdash_{\mathbf{FPhil}} v_1 \rightarrow \mathtt{has}_L$	Property of **Phil**
8.	$\vdash_{\mathbf{XFork}} \mathtt{lup?} \rightarrow \langle\mathtt{ldown}\rangle\top$	$\mathbf{f_{14}}$
9.	$\vdash_{\mathbf{FPhil}} \mathtt{has}_L \rightarrow \langle\mathtt{down}_L\rangle\top$	Def.f_1 & theorem 50
10.	$\vdash_{\mathbf{FPhil}} v_1 \rightarrow \langle\mathtt{down}_L\rangle\top$	PL, 7, 9
11.	$\vdash_{\mathbf{FPhil}} v_1 \wedge \neg v_2 \rightarrow \langle\mathtt{down}_L\rangle\top$	PL, 10
12.	$\vdash_{\mathbf{FPhil}} v_1 \wedge \neg v_2 \rightarrow [\mathtt{down}_L](\neg v_1 \wedge \neg v_2) \wedge \langle\mathtt{down}_L\rangle\top$	PL,10, 6

In this proof, the acronym DPL means that we can obtain the corresponding line using basic properties of the logic, similarly for PL (propositional logic) and ML (modal logic). Note that, in line 4, we use the *GGG* property. The other transitions between violation states can be proven in a similar way.

We can build a complete specification with forks and philosophers interacting. Let us keep this simple and consider only two philosophers. We can use the channels to coordinate the two philosophers. Consider the diagram shown in figure 7.10. The colimit of this diagram gives us the final design (say **TPhils**), and note that

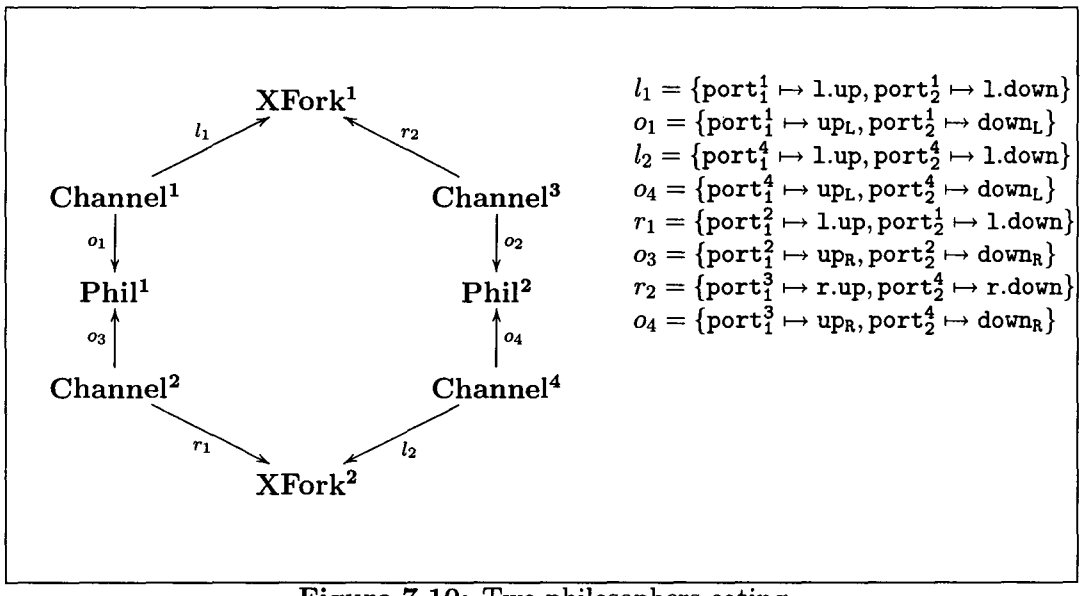


Figure 7.10: Two philosophers eating

the colimit produces the corresponding specification with all the needed renaming of clashing symbols. Note that at the right of this figure the different mappings appearing in the diagram are defined. These mappings define how the different parts of the design interconnect (as explained in [FM92]). The interesting point here is to analyze what happens with the upgrading diagram in this system, when we add an extra philosopher. Note that the two instances of **Phil** do not coordinate via any action (both coordinate with **XFork**, but using different channels), and therefore theorem 56 can be applied here, obtaining that this specification preserves the coproduct of the upgrading diagrams of each philosopher. Note that the coproduct of the upgrading diagrams of each philosopher with forks is the one illustrated in figure 7.11 This means that each of the (formulae which act as witnesses of a) transition of

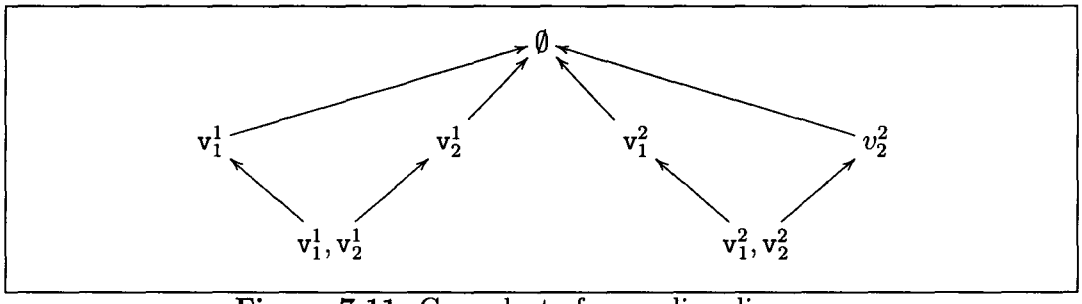


Figure 7.11: Coproduct of upgrading diagrams.

this diagram can be proven from the specification **TPhils**. It is worth investigating if there are other transitions (since the theorem above says that we have a faithful

(injective) functor, however we cannot ensure that this functor is full (surjective)). Note that when we have two philosophers, if one of them has a fork, then the other cannot start eating, and therefore there is no way to reach a state violation of the type $v_1^1 \wedge v_2^2$. This kind of extra-transition depends on how many philosophers we have in this specification; for example, if we have three philosophers, we can obtain further violation states.

Summarizing, theorems 56 and 58 allow us to deduce some basic transitions between violation states. However, some other transitions could be dependent on the specification being developed and have to be investigated by the designer (although it is worth noting that these theorems give us a good starting point to analyze the violation structure of a specification built from several components).

7.4 Further Remarks

In this chapter, we have taken further the formalism presented in the earlier chapters, using the ideas of [FM92] to allow the specification of different modules which can be put together to build a whole system. Also, we have provided a theoretical basis for analyzing the structure of the violations that can arise when executing an implementation of a specification. The main benefit of this formal machinery is that it allows us to deduce part of the violation structure of the system using the properties of the components.

It is important to stress that we have introduced a notion of bisimulation which allows us to characterize certain semantic structures which, in some way, reflect the locality of the components. Moreover, the deontic part of the original logic was generalized to allow a local version of permission and obligation, which is coherent with modularization, in the sense that the permissions and obligations of a given component do not affect (at least as far as this is desired) other parts of the system.

Chapter 8

Concluding Remarks

In this thesis we have investigated the utilization of deontic logics to specify concepts related to fault-tolerance. The main idea is to use axiomatic theories to specify computing systems at the design level. While theories describe components or modules, translations between them express the relationships between the different modules. This methodology can be traced back to [MT84] where theories and interpretations between them are used as a basis to specify and structure computing systems. In this setting, we use deontic predicates to capture the notion of prescription, which is highly related to the notion of violation: a violation occurs since the system exhibits a non-desired behaviour. Deontic predicates are intended to capture the distinction between ideal or normal behaviour of systems, on the one hand, and the non-ideal or abnormal behaviour, on the other. We proposed a logic which is enough expressive for expressing interesting examples and differs from the other deontic systems proposed in the literature. Furthermore, we investigated the properties that are needed to modularize the deontic predicates in such a way that they fit neatly in the methodology described above. Then we concluded that some axiomatic schemas must be added to reflect reasonable assumptions on permissions and obligations.

In the next section we state the contributions made in this thesis; we also describe some further work that seems interesting to develop using the formalisms described in the preceding chapters.

8.1 Contributions

Critical systems are very common today: airplane systems, software in nuclear power stations, health care related systems, bank systems, etc. Formal methods provide a rigorous way of producing software without errors. However, there are cases where the possibility of having bugs or faults is present for various reasons: a malicious environment, some component which changes over time or low level components which are not reliable. Because of this, formal frameworks that provide mechanisms to reason about faulty behaviour combine the benefits of both formal methods and fault-tolerance and allow us to develop more reliable software. The formalism presented in chapter 3 allows us to express the idea of normativeness, i.e., which behaviour of the system is desirable. Using these logical constructs, the concept of violation can be introduced into logical theories, and therefore recovery actions can also be expressed and related properties can be proven.

This deontic logic is different from previous formalisms presented in the literature; the deontic predicates are interpreted as properties of transitions and not over states as done in dynamic deontic logics; as a consequence, there are no relationships between the deontic predicates and the modalities. This provides some freedom when specifying systems, and also this approach fits better with fault-tolerance. For example, it allows us to separate clearly the notions of recovery action and allowed action (sometimes these two concepts are mixed up in the deontic literature where it is stated that an allowed action is one which yields a state free of violations). Furthermore, since we intend to use our logic for a rigorous analysis of software, we have proved that our logic is sound and complete, and we also proved that it is compact, which is an improvement with respect to other boolean modal logics proposed in the literature [GP90] and [Bro03]. Furthermore, the boolean modal logic that we present uses a relative version of the complement operator which, as we argued in chapter 3, is more suitable when specifying computing systems. It seems that axiomatizations for modal logics with relative complement have not been investigated deeply in the literature. (Broersen proposes an axiomatic system in [Bro03], but the completeness and the soundness of this system are not dealt with there.)

We described an extension of the basic logic introduced in chapter 3, which supports several versions of deontic predicates; this allows us to introduce stratified norms which can be used to avoid contrary-to-duty paradoxes, which are common in practice (as shown in the examples of chapter 4). Stratified norms allow designers to introduce several levels of norms and violations which, intuitively, distinguish the different classes of bad behaviour that could occur during the execution of the system. This classification allows us to prove that some faults are tolerated by the system and others not, as shown in the byzantine generals, where one kind of fault is tolerated

and others not.

Temporal predicates have been shown to be useful to specify and verify concurrent and reactive systems, and, because most of fault-tolerant systems use concurrency or they interact with an environment, then we have added temporal operators to our logic. We also extended the axiomatic system to cover the temporal operators, and we have proved the soundness and completeness of this extended system. It is worth remarking that we consider an operator Done() which is used to predicate about the last action executed; this predicate has been proposed in other works, [DK97] and [KQM91], however, a complete set of axioms for it cannot be found in the literature. (But, note that we prove in chapter 5 that the Done() operator can be expressed in terms of the other temporal operators in an anchored temporal logic.)

On the other hand, if we want to use this logic in practice, we need automated tools; tableaux deduction systems have been shown to be useful for automatic deduction [Fit90]. We present a tableaux system for our logic in chapter 5; the interesting point about this system is how the algebra of terms is used to label the formulae. The algebraic properties of the actions allow us to obtain a complete and sound tableaux system; furthermore this tableaux system has interesting properties. Our logic is dependent on the language, in the sense that when we change the symbols, the theorems that we can prove from the axiomatic system may differ; however, we have proven that there exists a bound on the number of actions that we need to consider when verifying a formula; if the formula is valid for this number of actions, then it will be valid for any language possibly containing more actions. This can also be read as saying that if a formula has some countermodel (i.e. a model which makes this formula false), then this model can be obtained using the tableaux with a language with the number of actions indicated by the bound. It is important to note that these extra actions can be thought of as environment actions, and therefore the theorem gives (for a given formula) the number of environment actions that must be considered to falsify the formula (if this is possible). This result seems to be relevant to finding counterexamples of specifications that interact with an environment.

With the purpose of making this logic applicable to large specifications, we have provided an extension of the logic with support for modularization; an important issue here is how the notion of component is reflected at the syntactic and semantical levels, i.e, what is the relationship between the model of a module and the model of the entire system. For example, in [FM92] a linear temporal logic is proposed, and the model of the modules are traces where the external actions (the actions outside of the module) do not affect the local variables. However, this characterization does not work in branching logics, where, in addition to preserving the variables, we have to preserve the possible choices of a component. We use the notion of bisimulation (which has been widely used in modal logic) to characterize local models. Further-

more, we provided a set of axioms which enables us to capture this locality at the syntactic level. One consequence of this characterization of locality by means of axioms is that structurality is satisfied when we consider a deduction relation augmented with the axioms of locality, i.e., the theorems of the components are preserved by the entire system. However, it is important to note that, when putting together several components, it is possible that the specification obtained is inconsistent. An important point here is that the tableaux system can be used to check the inconsistency of a specification (when it is made up of a finite number of axioms).

The modularization of the logic raises the question of how to preserve the locality of prescriptions, i.e., if we pursue the development of specifications in a modular way, prescriptions also have to be modularized, otherwise the local reasoning about a specification becomes hard because the prescriptions in other components may affect the component being analyzed. In chapter 7 we propose the use of several versions of permission and obligation, and we show that using colimits we can compose specifications in such a way that the locality of prescription is preserved. Furthermore, since we include violation constants in the language, the different violations in a specification make up a lattice-like structure. We show some results which allow us to approximate the violations structure of the entire system using the violations of the components; moreover, we have shown that, if the interaction between the components is coordinated only by means of safe actions (actions which are always allowed), then the violation structure of the components is preserved in the final system. A consequence of this is that the action of one component cannot cause violations in other components. We have observed in chapter 3 that, when working with deontic specification, it is useful to have some requirements about the behaviour of allowed actions; roughly speaking, allowed action do not introduce violations. This is called the *GGG* requirement in [SC06], where it is introduced to reason about transition systems specified with the language $\eta C+$, and we have shown that we can introduce this requirement by means of parametrizable axioms in a similar way to the use of frame axioms in [FM92]. This property is useful when proving properties over specifications with prescriptions, as we illustrated in the examples of chapter 7.

Summarizing, we have provided a deontic action logic which differs from the deontic logics presented in the literature, and we have illustrated with some examples why we think that this deontic logic is appropriate for use in specifying fault-tolerance. In addition, we have provided an extension of this deontic logic to allow designers to provide specifications in a modular way; we designed this extension in a way which allows us to keep the locality of the deontic predicates (i.e., that the prescriptions in a module do not necessarily affect other modules). Deontic logics have been proposed in some work [CJ96, LS04, Kho88] as an adequate logic for the study of fault-tolerance, in particular since the notion of prescription and violation are naturally embedded in these logics. We have provided a set of sufficiently complex examples to ground

this thinking; by means of these examples, we have studied some properties that can be proven from specifications with prescriptions. Part of the work presented in this thesis has been published in some earlier papers: [CM07a, CM08, CM07b, CM09].

8.2 Future Work

The logic that we presented in chapter 3 uses an atomic boolean algebra of actions; we take advantage of the atoms in the algebra of terms to obtain a canonical model and then to prove the completeness of the axiomatic system shown in that chapter, and we use the same idea to define the labels that we use with the formulae in the tableaux system. There are several algebras related to boolean algebras which have a similar structure. For example, boolean algebras with operators [JT51, JT52] are algebras which have the boolean operators and in addition they have other operators with several interesting properties. In particular, there are several classes of boolean algebras with residuated operators that also have atom-like elements; in particular, residuated algebras seem to be very expressive, and perhaps the notion of iteration can be captured here (see [Jip92] for an introduction to residuated boolean algebras). It seems to be interesting further work to extend our axiomatic systems to these more expressive algebras.

We have described an algorithm in chapter 5 to apply the tableaux rules; further work should be done to implement a software tool which allows us to apply this method in an automatic way. Such an algorithm would be useful to allow specifiers to get counterexamples (expressed by means of traces) of properties while analyzing a specification; these counterexamples can be used to investigate which possible runs of the system to be built can be dangerous and must be taken into account when improving the design. Here it is interesting to note that the counterexamples obtained by means of the tableaux system are traces made up of action terms that express which actions were executed and which were not; this level of detail is undoubtedly useful when analyzing specifications.

The logics presented throughout this thesis have propositional operators; a first-order extension of this logic would allow us to express interesting properties. However, first-order predicates bring undecidability, and furthermore (as proven in [Aba89] for linear temporal logic), many first-order temporal logics do not admit a complete calculus (although a more general notion of semantic structures can be considered to obtain completeness). To investigate how to keep the good meta properties of our logic (except, of course, decidability) in a first-order logic seems to constitute a interesting further work.

Institutions and π-Instituitions are abstract formulations of logical systems; while Institutions focus on the semantical aspects of a logical system, π-Institutions are intended to capture the notion of syntactical entailment. Note that we have taken a syntactical approach, in the sense that we use deduction systems to prove properties of specifications, and therefore π-Institutions seem to be a good theoretical setting where the properties of the deductive systems described in earlier chapters can be investigated. However, π-Institutions do not capture some important aspects of tableaux systems. For example, in many tableaux systems (including those exhibited in chapter 5), in the case where a formula is not valid, we can obtain a counterexample (a model which satisfies the negation of this formula). An abstract formulation of tableaux that captures these properties would be useful when reasoning about several tableaux systems and the connection between them. In particular, this abstract formulation could help to understand how different counterexamples coming from related tableaux are connected. For example, it should be possible to extract counterexamples for the components from counterexamples obtained for the entire system (this will allow us to analyze the counterexamples with respect to a simpler specification). Also, the universal constructions could be used to obtain tableaux from smaller tableaux, allowing in this way compositional reasoning about (tableaux) proofs. For example, it seems plausible that from two tableaux, one for a formula φ and another for a formula ψ (built from the same language), the product of these tableaux gives us a tableaux for the formula $\varphi \wedge \psi$; this construction is called clashing tableaux in [Smu68], but it is not expressed in categorical terms. Other categorical constructions (limits in general) will allow us to put many tableaux together. We think that these ideas deserve a deeper investigation.

In chapter 7 we have investigated mechanisms to put together several components. In particular, we provided a theoretical framework for the analysis of violations. As we argued in that chapter, an important issue is how the structure of the violations occurring in the components are reflected in the final system. We shown that, in some cases, components can be put together safely, i.e., preserving the properties of the local violations. It is important to investigate further these kinds of properties; providing an important set of these style of results will allow designers to reason in a easier way about the composition of specifications with support for violations. In particular, it seems possible that rely/guarantee mechanisms can be used to extend the properties described in chapter 7 in such a way that we can apply them in more general situations.

The theoretical framework exhibited in earlier chapters is intended to be used to prove properties related to fault-tolerance. We provided some examples that are complex enough to show how these logics can be used in practice. However, it seems interesting to produce specifications of existing fault-tolerant mechanisms or protocols and, therefore, investigate their properties. Of course, the formalisms described in this

thesis are intended to be used to specify new fault-tolerant systems or protocols and, then, their properties can be investigated in a rigorous way. It is worth remarking that the logical frameworks used in this thesis can be thought of as being at a low level, in the sense that a more design-oriented language can be developed and, then, the specifications made with this language can be "compiled" to the formalisms described in chapter 3, and so the deductive systems introduced in chapters 3 and 5 can be used to prove the properties of these specifications. Summarizing, this formalism can be utilized to provide the semantics (and the logical calculus) of languages more oriented to system design.

Finally, as we remarked above, the logics introduced in this thesis are dependent on vocabularies, this implies that, when we add more actions, the set of theorems changes, perhaps some properties are no longer a consequence of the specification. Intuitively, this says that the specifications are in some degree dependent on the actions of the environment (it seems intuitive that if we add more behaviour to the environment, the possible faults of the system may increase). However, we proved that it is possible to obtain a bound in the number of the "extra" actions needed to prove a given property for any language. This result might be useful at time of proving important system properties like liveness properties (where the behaviour of the environment is relevant). It seems an interesting stream of research to investigate in which scenarios this result can be used, and if this kind of meta property can be implemented in a software tool in such a way that important properties of components can be proven in an automatic way. In particular, the idea of an automatic tool that finds environments which may falsify a property of a component (using properties as the one shown in chapter 5) seems very attractive.

Appendix A

Proof of lemma 4.
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg \mathtt{i.v}_1$	PL, **Phil1**
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathtt{i.v}_1 \rightarrow (\mathtt{i.has}_L \vee \mathtt{i.has}_R))$	PL, 1, **Phi10**

Ind. Case:

1.	$\neg\mathtt{i.has}_L \wedge \neg\mathtt{i.has}_R \rightarrow \neg\mathtt{i.eating}$	PL, **Phil18**
2.	$\neg\mathtt{i.eating} \rightarrow \mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R)$	PL, **Phil21**
3.	$\neg\mathtt{i.has}_L \wedge \neg\mathtt{i.has}_R \rightarrow \neg\mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R)$	PL, 1, 2
4.	$\neg\mathtt{i.v}_1 \wedge \neg\mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R) \rightarrow [\mathbf{U}]\neg\mathtt{i.v}_1$	PL, **V2**
5.	$\neg\mathtt{i.v}_1 \wedge \neg(\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow [\mathbf{U}]\neg\mathtt{i.v}_1$	PL, 3, 4
6.	$\mathtt{i.v}_1 \wedge \neg(\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow [\mathbf{U}](\neg\mathtt{i.v}_1 \vee (\mathtt{i.has}_L \vee \mathtt{i.has}_R))$	PL, 5
7.	$\mathtt{i.v}_1 \wedge (\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow [\mathtt{i.down}_R \sqcap \mathtt{i.down}_L]\neg\mathtt{i.v}_1$	PL, **V4**
8.	$\mathtt{i.v}_1 \wedge (\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow$ $[\mathtt{i.down}_R \sqcap \mathtt{i.down}_L](\neg\mathtt{i.v}_1 \vee (\mathtt{i.has}_L \vee \mathtt{i.has}_R))$	PL, 7
9.	$\mathtt{i.has}_L \rightarrow \overline{[\mathtt{i.down}_L]}\mathtt{i.has}_L$	PL, **Phil6**
10.	$\mathtt{i.has}_R \rightarrow \overline{[\mathtt{i.down}_R]}\mathtt{i.has}_R$	PL, **Phil6**
11.	$\mathtt{i.has}_L \vee \mathtt{i.has}_R \rightarrow \overline{[\mathtt{i.down}_L \sqcup \mathtt{i.down}_R]}(\mathtt{i.has}_L \vee \mathtt{i.has}_R)$	
12.	$\neg\mathtt{i.v}_1 \wedge (\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow$ $\overline{[\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]}(\neg\mathtt{i.v}_1 \vee (\mathtt{i.has}_L \vee \mathtt{i.has}_r))$	PL, 11 PL, **T4**, 9, 10
13.	$\neg\mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R) \rightarrow [\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1$	**V1**
14.	$\neg\mathtt{i.v}_1 \wedge \neg\mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R) \rightarrow [\mathbf{U}]\neg\mathtt{i.v}_1$	**V3**
15.	$[\mathbf{U}]\neg\mathtt{i.v}_1 \wedge \mathtt{i.down}_L \sqcap \mathtt{i.down}_R \sqsubseteq \mathbf{U} \rightarrow [\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1$	**T3**
16.	$[\mathbf{U}]\neg\mathtt{i.v}_1 \rightarrow [\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1$	PL, 15
17.	$\neg\mathtt{i.v}_1 \wedge \neg\mathsf{O}(\mathtt{i.down}_L \sqcap \mathtt{i.down}_R) \rightarrow [\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1$	PL, 14, 16
18.	$\neg\mathtt{i.v}_1 \rightarrow [\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1$	PL, 13, 17
19.	$\neg\mathtt{i.v}_1 \wedge (\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow$ $[\mathtt{i.down}_L \sqcap \mathtt{i.down}_R]\neg\mathtt{i.v}_1 \vee (\mathtt{i.has}_L \vee \mathtt{i.has}_R)$	PL, 18
20.	$\neg\mathtt{i.v}_1 \wedge (\mathtt{i.has}_L \vee \mathtt{i.has}_R) \rightarrow [\mathbf{U}](\neg\mathtt{i.v}_1 \vee (\mathtt{i.has}_L \vee \mathtt{i.has}_R))$	PL, 12, 19

21.	$i.v_1 \wedge i.has_L \wedge \neg i.has_R \rightarrow i.v_2$	PL, **V3**
22.	$i.v_1 \wedge \neg i.has_R \rightarrow [i.down_L]\neg i.v_1$	PL, **V6**
23.	$i.v_1 \wedge i.has_L \wedge \neg i.has_R \rightarrow [i.down_L]\neg i.v_1$	PL, 21, 22
24.	$i.v_1 \wedge i.has_R \wedge \neg i.has_L \rightarrow i.v_2$	PL, **V3**
25.	$i.v_2 \wedge \neg i.has_R \rightarrow [i.down_R]\neg i.v_1$	PL, **V6**
26.	$i.v_1 \wedge i.has_R \wedge \neg i.has_L \rightarrow [i.down_R]\neg i.v_1$	PL, 25
27.	$i.has_L \rightarrow \boxed{i.down_L} i.has_L$	**Phil6**
28.	$i.v_1 \wedge i.has_L \wedge \neg i.has_R \rightarrow$ $[i.down_L \sqcup \overline{i.down_L}](\neg i.v_1 \vee i.has_L)$	ML, **T4**, 23, 26 27
29.	$i.v_1 \wedge i.has_L \wedge \neg i.has_R \rightarrow [\mathbf{U}](\neg i.v_1 \vee (i.has_L \vee i.has_R))$	PL, 28
30.	$i.has_R \rightarrow \boxed{i.down_R} i.has_R$	PL, **Phil6**
31.	$i.v_1 \wedge i.has_R \wedge \neg i.has_L \rightarrow \boxed{i.down_R}(\neg i.v_1 \vee (i.has_R \vee i.has_L))$	PL, 26, 30
32.	$i.v_1 \wedge i.has_R \wedge \neg i.has_L \rightarrow [\mathbf{U}](\neg i.v_1 \vee i.has_R \vee i.has_L)$	PL, 26, 31, **T4**
33.	$i.v_1 \wedge i.has_R \wedge i.has_L \rightarrow \neg i.v_2$	PL, **V3**
34.	$i.has_R \rightarrow \boxed{i.down_R} i.has_R$	PL, **Phil6**
35.	$i.has_L \rightarrow \boxed{i.down_L} i.has_L$	PL, **Phil6**
36.	$i.has_R \wedge i.has_L \rightarrow \boxed{i.down_R \sqcup i.down_L}(i.has_R \vee i.has_L)$	**BA**, 34, 35
37.	$i.v_1 \rightarrow [i.down_L \sqcap i.down_R]\neg i.v_1$	PL, **V7**
38.	$i.has_R \wedge i.has_L \wedge i.v_1 \rightarrow [\mathbf{U}](\neg i.v_1 \vee (i.has_r \vee i.has_L))$	PL, **T4**, 36, 35
39.	$\neg i.v_1 \vee (i.has_R \vee i.has_R) \rightarrow [\mathbf{U}](\neg i.v_1 \vee (i.has_R \vee i.has_L))$	PL, 20, 29 32, 38

Proof of lemma 5

1.	$\neg i.v_1 \wedge \neg O(i.down_L \sqcap i.down_R) \rightarrow [\mathbf{U}]\neg i.v_1$	**V2**
2.	$\neg i.v_1 \wedge \neg i.eating \rightarrow [\mathbf{U}]\neg i.v_1$	PL, **Phil21**, 1
3.	$\neg i.v_1 \wedge \neg i.eating \rightarrow [\mathbf{U}]\neg i.v_1$	PL, **V1**
4.	$\neg i.v_1 \wedge O(i.down_L \sqcap i.down_R) \rightarrow [i.down_L \sqcap i.down_R]\neg i.v_1$	PL, **V1**
5.	$\neg i.v_1 \wedge i.eating \rightarrow \boxed{i.down_L sqcap i.down_R} i.v_1$	PL, **Phil21**,**V2**
6.	$\neg i.v_1 \wedge i.eating \rightarrow [i.down_L \sqcap i.down_R]\neg i.v_1$	PL, **V1**
7.	$[i.down_L]\neg i.has_L \wedge [i.down_R]\neg i.has_R$	**Phil6**
8.	$[i.down_L \sqcap i.down_R](\neg i.has_L \wedge \neg i.has_R)$	PL, **T5**, 7
9.	$[i.down_L \sqcap i.down_R]\neg i.eating$	PL, 8, **Phil18**
10.	$\neg i.v_1 \wedge i.eating \rightarrow [i.down_L \sqcap i.down_R]\neg i.eating$	PL, 9
11.	$i.eating \rightarrow [\mathbf{U}]Done((i.down_L \sqcap i.down_R) \sqcup i.getbad)$	**Phil20**
12.	$i.eating \rightarrow \boxed{i.down_L \sqcap i.down_R} Done(i.getbad)$	PL,**T16**, **T17**, 11
13.	$([i.getbad]i.bathroom) \wedge (i.bathroom \rightarrow \neg i.eating)$	PL, **Phil3** **Phil18**, **Phil22**
14.	$i.eating \rightarrow \boxed{i.down_L \sqcap i.down_R}\neg i.eating$	PL, **T17**, 13, 14
15.	$\neg i.v_1 \wedge i.eating \rightarrow \boxed{i.down_L \sqcap i.down_R}\neg i.eating$	PL, 14
16.	$\neg i.v_1 \wedge i.eating \rightarrow [\mathbf{U}]\neg i.eating$	PL, 10, **T4**, 15
17.	$\neg i.v_1 \wedge (\neg i.eating \vee i.eating) \rightarrow [\mathbf{U}](\neg i.eating \vee \neg i.v_1)$	PL, 2, 16
18.	$\neg i.v_1 \rightarrow [\mathbf{U}](\neg i.v_1 \vee \neg i.eating)$	PL, 17

Proof of lemma 6. The proof is by induction. We prove $\mathtt{i.v_1} \wedge \neg \mathtt{i.v_2} \rightarrow \neg(\mathtt{i}+1).\mathtt{eating}$, the another is similar.
Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg \mathtt{i.v_1}$	PL, **Phil1**
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathtt{i.v_1} \wedge \neg \mathtt{i.v_2} \rightarrow \neg(\mathtt{i}+1).\mathtt{eating})$	PL, 1

Ind.Case:

1.	$(\mathtt{i}+1).\mathtt{eating} \rightarrow (\mathtt{i}+1).\mathtt{has_L} \wedge (\mathtt{i}+1).\mathtt{has_R}$	PL, **Phil18**
2.	$(\mathtt{i}+1).\mathtt{has_R} \rightarrow \neg \mathtt{i.has_L}$	Lemma 3
3.	$(\mathtt{i}+1).\mathtt{eating} \rightarrow \neg \mathtt{i.has_L}$	PL, 10, 11
4.	$(\mathtt{i}+1).\mathtt{eating} \rightarrow \neg(\neg \mathtt{i.v_2} \wedge \mathtt{i.v_1})$	PL, **V3**, 1
5.	$[\mathbf{U}](\neg(\mathtt{i}+1).\mathtt{eating} \vee \neg(\neg \mathtt{i.v_2} \wedge \mathtt{i.v_1}))$	**GN**, 4
6.	$\neg(\mathtt{i.v_1} \wedge \neg \mathtt{i.v_2}) \vee (\neg(\mathtt{i}+1).\mathtt{eating}) \rightarrow$ $[\mathbf{U}](\neg(\mathtt{i}+1).\mathtt{eating} \vee \neg(\neg \mathtt{i.v_2} \wedge \mathtt{i.v_1}))$	PL, 14

Proof of lemma 7:

1.	$\neg \mathtt{i.eating} \leftrightarrow \neg \mathtt{i.has_L} \vee \neg \mathtt{i.has_R} \vee \mathtt{i.bathroom}$	**Phil18**
2.	$\neg \mathtt{i.v_1} \wedge \neg \mathsf{O}(\mathtt{i.down_L} \sqcap \mathtt{i.down_R}) \rightarrow [\mathbf{U}]\neg \mathtt{i.v_1}$	**V2**
3.	$\neg \mathtt{i.eating} \rightarrow \neg \mathsf{O}(\mathtt{i.down_L} \sqcap \mathtt{i.down_R})$	**Phil21**
4.	$\neg \mathtt{i.v_1} \wedge \neg \mathtt{i.eating} \rightarrow [\mathbf{U}]\neg \mathtt{i.v_1}$	PL, 2, 3
5.	$\neg \mathtt{i.v_1} \wedge \neg \mathtt{i.eating} \rightarrow [\mathbf{U}](\neg \mathtt{i.v_1} \vee \neg \mathtt{i.eating})$	PL, 4
6.	$\mathtt{i.v_1} \rightarrow [\mathtt{i.down_L} \sqcap \mathtt{i.down_R}]\neg \mathtt{i.v_1}$	**V4**
7.	$\mathtt{i.v_1} \rightarrow \mathtt{i.has_L} \vee \mathtt{i.has_R}$	Lemma 3
8.	$\mathtt{i.v_1} \rightarrow \mathtt{i.has_L} \vee \mathtt{i.has_R}$	Lemma 4
9.	$\mathtt{i.bathroom} \wedge \mathtt{i.has_L} \rightarrow \overline{[\mathtt{i.getbetter}]}\mathtt{i.bathroom}$	**Phil23**
10.	$\mathtt{i.bathroom} \wedge \mathtt{i.has_L} \rightarrow [\mathtt{i.getbetter}]\mathsf{Done}(\mathtt{i.down_L})$	**Phil19**
11.	$[\mathtt{i.down_L}]\neg \mathtt{i.has_L}$	**Phil4**
12.	$\mathtt{i.bathroom} \wedge \mathtt{i.has_L} \rightarrow [\mathtt{i.getbetter}]\neg \mathtt{i.has_L}$	PL, **T17**, 10, 11
13.	$\mathtt{i.bathroom} \wedge \mathtt{i.has_R} \rightarrow [\mathtt{i.getbetter}]\mathsf{Done}(\mathtt{i.down_R})$	**Phil19**
14.	$[\mathtt{i.down_R}]\neg \mathtt{i.has_R}$	**Phil4**
15.	$\mathtt{i.bathroom} \wedge \mathtt{i.has_L} \rightarrow [\mathtt{i.getbetter}]\neg \mathtt{i.has_L}$	PL, 13, 14
16.	$\mathtt{i.bathroom} \wedge (\mathtt{i.has_R} \vee \mathtt{i.has_L}) \rightarrow$ $[\mathtt{i.getbetter}](\neg \mathtt{i.bathroom} \wedge (\neg \mathtt{i.has_R} \vee \neg \mathtt{i.has_L}))$	PL, 12, 15
17.	$\mathtt{i.bathroom} \wedge (\mathtt{i.has_R} \vee \mathtt{i.has_L}) \rightarrow$ $[\mathtt{i.getbetter} \sqcup \overline{\mathtt{i.getbetter}}](\neg \mathtt{i.bathroom} \wedge (\neg \mathtt{i.has_R} \vee \neg \mathtt{i.has_L}))$	PL, 9, 16
18.	$\mathtt{i.bathroom} \wedge (\mathtt{i.has_R} \vee \mathtt{i.has_L}) \rightarrow [\mathbf{U}]\neg \mathtt{i.eating}$	PL, 1, 17
19.	$\mathtt{i.v_1} \wedge \neg \mathtt{i.eating} \rightarrow [\mathbf{U}](\neg \mathtt{i.v_1} \vee \neg \mathtt{i.eating})$	PL, 1, 8, 18
20.	$\neg \mathtt{i.eating} \rightarrow [\mathbf{U}](\neg \mathtt{i.v_1} \vee \neg \mathtt{i.eating})$	PL, 19, 5

Proof of lemma 8. The proof is by induction. Base Case:

1.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg \mathtt{i.v}_1 \wedge \neg \mathtt{i.eating}$	PL, **Phil1**
2.	$\neg\mathsf{Done}(\mathbf{U}) \rightarrow (\mathtt{i.v}_1 \rightarrow \neg \mathtt{i.eating})$	PL, 1

Ind.Case:

1.	$\neg \mathtt{i.v}_1 \rightarrow [\mathbf{U}]\neg(\mathtt{i.v}_1 \wedge \mathtt{i.eating})$	lemma 5
2.	$\neg \mathtt{i.eating}] \rightarrow [\mathbf{U}]\neg(\mathtt{i.v}_1 \wedge \mathtt{i.eating})$	lemma 7
3.	$\neg \mathtt{i.v}_1 \vee \neg \mathtt{i.eating} \rightarrow [\mathbf{U}]\neg(\mathtt{i.v}_1 \wedge \mathtt{i.eating})$	PL, 1, 2

Proof of lemma 9. The proof is by induction: the base case is straightforward using **Phil1**. The inductive case is as follows:

1.	$\neg \mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow [\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]\neg \mathtt{i.v}_1$	**V1**
2.	$\neg \mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow \overline{[\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]}\mathtt{i.v}_1$	**V1**
3.	$\mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow \mathtt{i.eating}$	**Phil21**
4.	$\mathtt{i.eating} \rightarrow [\mathbf{U}]\mathsf{Done}(\mathtt{i.getthk} \sqcup \mathtt{i.getbad})$	**Phil20**
5.	$\mathtt{i.eating} \rightarrow \mathtt{i.has}_\mathtt{L} \wedge \mathtt{i.has}_\mathtt{R}$	**Phil18**
6.	$\mathtt{i.has}_\mathtt{L} \wedge \mathtt{i.has}_\mathtt{R} \rightarrow [\mathtt{i.getthk}](\mathsf{Done}(\mathtt{i.down}_\mathtt{L}) \wedge \mathsf{Done}(\mathtt{i.down}_\mathtt{R}))$	**Phil7**
7.	$\mathtt{i.eating} \rightarrow [\mathbf{U}]\mathsf{Done}(\mathtt{i.gethtk}) \vee [\mathbf{U}]\mathsf{Done}(\mathtt{i.getbad})$	ML, **T14**, 4
8.	$\mathtt{i.eating} \rightarrow$ $[\mathbf{U}](\mathsf{Done}(\mathtt{i.down}_\mathtt{L} \sqcap \mathsf{Done}(\mathtt{i.down}_\mathtt{R}))) \vee [\mathbf{U}](\mathsf{Done}(\mathtt{i.getbad}))$	**T17**, 6, 7
9.	$\mathtt{i.eating} \rightarrow \overline{[\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]}\neg\mathsf{Done}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R})$	PL, **TempAx8**
10.	$\mathtt{i.eating} \rightarrow \overline{[\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]}\mathsf{Done}(\mathtt{i.getbad})$	**T4**, **T16**, 8, 9
11.	$[\mathtt{i.getbad}]\mathtt{i.bathroom}$	**Phil22**
12.	$\mathtt{i.eating} \rightarrow \overline{[\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]}\mathtt{i.bathroom}$	**T17**, 10, 11
13.	$\neg \mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow$ $\overline{[\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}]}(\mathtt{i.bathroom} \vee \neg \mathtt{i.v}_1)$	ML, 12
14.	$\neg \mathtt{i.v}_1 \wedge \mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow [\mathbf{U}](\mathtt{i.bathroom} \vee \neg \mathtt{i.v}_1)$	**BA**, PL, 1, 3
15.	$\neg \mathtt{i.v}_1 \wedge \neg\mathsf{O}(\mathtt{i.down}_\mathtt{L} \sqcap \mathtt{i.down}_\mathtt{R}) \rightarrow [\mathbf{U}](\mathtt{i.bathroom} \vee \neg \mathtt{i.v}_1)$	**V2**
16.	$\neg \mathtt{i.v}_1 \rightarrow [\mathbf{U}](\mathtt{i.bathroom} \vee \neg \mathtt{i.v}_1)$	**BA**, PL, 14, 15
17.	$\mathtt{i.bathroom} \rightarrow \overline{[\mathtt{i.getthk}]}\mathtt{i.bathroom}$	**Phil22**
18.	$\mathtt{i.v}_1 \wedge \mathtt{i.has}_\mathtt{L} \rightarrow [\mathtt{i.down}_\mathtt{L}]\neg \mathtt{i.v}_1$	**Phil5**
19.	$\mathtt{i.v}_1 \wedge \mathtt{i.has}_\mathtt{R} \rightarrow [\mathtt{i.down}_\mathtt{R}]\neg \mathtt{i.v}_1$	**Phil5**
20.	$\mathtt{i.v}_1 \rightarrow \mathtt{i.has}_\mathtt{R} \vee \mathtt{i.has}_\mathtt{L}$	lemma 4
21.	$\mathtt{i.bathroom} \wedge \mathtt{i.has}_\mathtt{L} \rightarrow [\mathtt{i.getthk}]\neg \mathtt{i.v}_1$	**T17**, **Phil7**, **V6**
22.	$\mathtt{i.bathroom} \wedge \mathtt{i.has}_\mathtt{R} \rightarrow [\mathtt{i.getthk}]\neg \mathtt{i.v}_1$	**T17**, **Phil7**, **V6**
23.	$\mathtt{i.v}_1 \wedge \mathtt{i.bathroom} \rightarrow [\mathtt{i.getthk}]\neg \mathtt{i.v}_1$	ML, 20, 21, 22
24.	$\mathtt{i.bathroom} \wedge \mathtt{i.v}_1 \rightarrow [\mathbf{U}](\neg \mathtt{i.v}_1 \vee \mathtt{i.bathroom})$	**BA**, PL, 16, 17, 23

Proof of lemma 10. The proof is by induction, the base case is direct using **Phil1**,

we prove the inductive case:

1.	$\mathtt{i.thk} \rightarrow \boxed{\mathtt{i.getthk}}\neg\mathtt{i.thk}$	**Phil8**
2.	$\mathtt{i.has_L} \rightarrow [\mathtt{i.getthk}]\mathtt{Done(i.down_L)}$	**Phil7**
3.	$[\mathtt{i.down_L}]\neg\mathtt{i.has_L} \wedge [\mathtt{i.getthk}]\mathtt{Done(i.down_L)} \rightarrow [\mathtt{i.getthk}]\neg\mathtt{i.has_L}$	**T17**
4.	$\mathtt{i.has_R} \rightarrow [\mathtt{i.getthk}]\mathtt{Done(i.down_R)}$	**Phil7**
5.	$[\mathtt{i.down_R}]\neg\mathtt{i.has_R} \wedge [\mathtt{i.getthk}]\mathtt{Done(i.down_R)} \rightarrow [\mathtt{i.getthk}]\neg\mathtt{i.has_R}$	**T17**
6.	$\neg\mathtt{i.thk} \wedge (\mathtt{i.has_L} \vee \mathtt{i.has_R}) \rightarrow$ $[\mathtt{i.geththk}](\mathtt{i.thk} \wedge (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}))$	PL, 3, 5, **Phil5**
7.	$(\neg\mathtt{i.has_L} \rightarrow \boxed{\mathtt{i.up_L}}\neg\mathtt{i.has_L}) \wedge (\neg\mathtt{i.has_R} \rightarrow \boxed{\mathtt{i.up_R}}\neg\mathtt{i.has_R})$	**Phil7**
8.	$(\mathtt{i.getthk} \sqsubseteq \overline{\mathtt{i.up_L}}) \wedge (\mathtt{i.getthk} \sqsubseteq \overline{\mathtt{i.up_R}})$	**BA**, **Phil7**
9.	$\mathtt{i.getthk} \sqsubseteq \overline{\mathtt{i.up_L} \sqcup \mathtt{i.up_R}}$	**BA**, 8
10.	$\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R} \rightarrow [\mathtt{i.getthk}](\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R})$	**T3**, 7, 8
11.	$\neg\mathtt{i.thk} \wedge (\neg\mathtt{i.has_R} \wedge \neg\mathtt{i.has_L}) \rightarrow$ $[\mathtt{i.getthk}](\mathtt{i.thk} \wedge (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}))$	ML, 10, **Phil8**
12.	$\neg\mathtt{i.thk} \rightarrow [\mathbf{U}](\neg\mathtt{i.thk} \vee (\neg\mathtt{i.has_L} \wedge \mathtt{i.has_R}))$	**BA**, PL, 1, 6, 11
13.	$\mathtt{i.thk} \wedge (\mathtt{i.has_L} \wedge \mathtt{i.has_R}) \rightarrow [\mathtt{i.up_L} \sqcup \mathtt{i.up_R}]\bot$	**Phil9**
14.	$\mathtt{i.thk} \wedge (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}) \rightarrow$ $[\mathtt{i.up_L} \sqcup \mathtt{i.up_R}](\neg\mathtt{i.thk} \vee (\neg\mathtt{i.has_R} \wedge \neg\mathtt{i.has_L}))$	ML, 13
15.	$\mathtt{i.thk} \wedge (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}) \rightarrow$ $\boxed{\mathtt{i.up_L} \sqcup \mathtt{i.up_R}}(\neg\mathtt{i.thk} \vee (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}))$	ML, **Phil6**
16.	$\neg\mathtt{i.thk} \vee (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}) \rightarrow [\mathbf{U}](\neg\mathtt{i.thk} \vee (\neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}))$	**BA**, PL, 1, 12, 15

Proof of lemma 11. The proof is by induction, the base case is, as usual, using **Phil1**, the inductive case is as follows:

1.	$\neg\mathtt{i.hungry} \wedge (\mathtt{i}+1).\mathtt{hungry} \rightarrow [\mathtt{i.gethungry}]\bot$	**Phil14**
2.	$\neg\mathtt{i.hungry} \rightarrow \boxed{\mathtt{i.gethungry}}\neg\mathtt{i.hungry}$	**Phil17**
3.	$\neg\mathtt{i.hungry} \wedge (\mathtt{i}+1).\mathtt{hungry} \rightarrow [\mathbf{U}](\neg\mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry})$	**BA**, PL, 1, 2
4.	$\mathtt{i.gethungry} \sqcap (\mathtt{i}+1).\mathtt{gethungry} = \emptyset$	**Phil12**
5.	$\mathtt{i.gethungry} \sqsubseteq \overline{\mathtt{i}+1.\mathtt{gethungry}}$	**BA**, 4
6.	$\neg(\mathtt{i}+1).\mathtt{hungry} \rightarrow \boxed{(\mathtt{i}+1).\mathtt{gethungry}}\neg(\mathtt{i}+1).\mathtt{gethungry}$	**Phil17**
7.	$(\mathtt{i}+1).\mathtt{hungry} \rightarrow [\mathtt{i.gethungry}](\mathtt{i}+1.\mathtt{hungry})$	PL, **T4**, 5, 6
8.	$\neg\mathtt{i.hungry} \wedge \neg(\mathtt{i}+1).\mathtt{hungry} \rightarrow$ $[\mathtt{i.gethungry}](\neg\mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry})$	**T4**, 6, 7
9.	$\neg\mathtt{i.hungry} \rightarrow \boxed{\mathtt{i.gethungry}}\neg\mathtt{i.hungry}$	**Phil17**
10.	$\neg\mathtt{i.hungry} \wedge \neg(\mathtt{i}+1).\mathtt{hungry} \rightarrow$ $\boxed{\mathtt{i.gethungry}}(\neg\mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry})$	ML, 9
11.	$\mathtt{i.hungry} \wedge \neg(\mathtt{i}+1).\mathtt{hungry} \rightarrow$ $[\mathbf{U}](\neg\mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry})$	Symmetrically from 3
12.	$\neg\mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry} \rightarrow$	

$[\mathbf{U}](\neg \mathtt{i.hungry} \vee \neg(\mathtt{i}+1).\mathtt{hungry})$ **BA**, PL, 3, 10, 11

Proof of lemma 12.

1. $(\mathtt{i.hungry}) \wedge (\mathtt{fork_i.down} \vee \mathtt{i.has_R}) \wedge (\mathtt{fork_{i+1}.down} \vee \mathtt{i.has_L})$
   $\rightarrow \mathsf{AN}\mathtt{i.eating}$ CTL, **Phil15**
2. $(\mathtt{i}-1).\mathtt{thk} \rightarrow \neg(\mathtt{i}-1).\mathtt{has_L} \wedge \neg(\mathtt{i}-1).\mathtt{has_R}$ lemma 10
3. $\neg(\mathtt{i}-1).\mathtt{has_L} \rightarrow \mathtt{fork_i.down} \vee \mathtt{i.has_R}$ PL, **Phil10**
4. $\neg(\mathtt{i}+1).\mathtt{has_R} \rightarrow \mathtt{fork_{i+1}.down} \vee \mathtt{i.has_L}$ PL, **Phil10**
5. $\mathsf{AG}((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \wedge \neg\mathtt{i.bathroom} \wedge \neg(\mathtt{i}-1).\mathtt{bathroom})$
   $\wedge \mathtt{i.hungry} \wedge (\mathtt{i}-1).\mathtt{thk} \rightarrow \mathsf{EN}(\mathtt{i.eating})$ CTL, 1, 2, 3 ,4

Proof of lemma 13.

1. $(\mathtt{i}-1).\mathtt{thk} \rightarrow \neg(\mathtt{i}-1).\mathtt{has_L} \wedge \neg(\mathtt{i}-1).\mathtt{has_R}$ lemma 10
2. $\mathtt{i.thk} \rightarrow \neg\mathtt{i.has_L} \wedge \neg\mathtt{i.has_R}$ lemma 10
3. $\neg(\mathtt{i}-1).\mathtt{has_L} \wedge \neg\mathtt{i.has_R} \rightarrow \mathtt{fork_i.down}$ **Phil10**
4. $\neg(\mathtt{i}+1).\mathtt{v_2} \wedge (\mathtt{i}+1).\mathtt{has_L} \rightarrow \neg(\mathtt{i}+1).\mathtt{has_R}$ PL, **V3**
5. $\mathtt{i.thk} \rightarrow \neg\mathtt{i.bathroom}$ PL, **Phil3**
6. $\neg(\mathtt{i}+1).\mathtt{has_R} \wedge \neg\mathtt{i.has_L} \rightarrow \mathtt{fork_{i+1}.down}$ **Phil10**
7. $(\mathtt{i}+1).\mathtt{v_2} \rightarrow (\mathtt{i}+1).\mathtt{bathroom}$ lemma 9
8. $(\mathtt{i}+1).\mathtt{bathroom} \rightarrow \neg(\mathtt{i}+1).\mathtt{hungry}$ PL, **Phil3**
9. $(\mathtt{i}+1).\mathtt{v_2} \rightarrow \neg(\mathtt{i}+1).\mathtt{hungry}$ PL, 7, 8
10. $\mathtt{i.thk} \wedge \neg(\mathtt{i}-1).\mathtt{hungry} \wedge \neg(\mathtt{i}+1).\mathtt{hungry} \wedge$
    $\mathtt{fork_i.down} \wedge \mathtt{fork_{i+1}.down} \rightarrow \langle \mathtt{i.gethungry} \rangle \top$ PL, **Phil13**
11. $\langle \mathtt{i.gethungry} \rangle \mathtt{i.hungry} \rightarrow \langle \mathbf{U} \rangle \mathtt{i.hungry}$ **T3**
12. $\langle \mathbf{U} \rangle \mathtt{i.hungry} \rightarrow \mathsf{EN}\mathtt{i.hungry}$ PL, **TempAx1**
13. $(\mathtt{i.gethungry} \sqcap \overline{(\mathtt{i}-1).\mathtt{gethungry}}) = \emptyset$ **Phil12**
14. $\mathtt{i.gethungry} \sqsubseteq \overline{(\mathtt{i}-1).\mathtt{gethungry}}$ **BA**, 13
15. $(\mathtt{i}-1).\mathtt{thk} \wedge \mathsf{AN}(\neg(\mathtt{i}-1).\mathtt{bathroom}) \rightarrow$
    $[\overline{(\mathtt{i}-1).\mathtt{gethungry}}](\mathtt{i}-1).\mathtt{thk}$ lemma 16
16. $(\mathtt{i}-1).\mathtt{thk} \wedge \mathsf{AN}(\neg(\mathtt{i}-1).\mathtt{bathroom}) \rightarrow [\mathtt{i.gethungry}](\mathtt{i}-1).\mathtt{thk}$ PL, **T3**, 16
17. $[\mathtt{i.gethungry}](\mathtt{i}-1).\mathtt{thk} \wedge \langle \mathtt{i.gethungry} \rangle \mathtt{i.hungry} \rightarrow$
    $\langle \mathtt{i.gethungry} \rangle ((\mathtt{i}-1).\mathtt{thk} \wedge \mathtt{i.hungry})$ ML
18. $\mathsf{AN}((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \wedge \neg\mathtt{i.bathroom} \wedge$
    $\neg(\mathtt{i}-1).\mathtt{bathroom}) \wedge \mathtt{i.thk} \wedge (\mathtt{i}-1).\mathtt{thk} \rightarrow$
    $\langle \mathtt{i.gethungry} \rangle ((\mathtt{i}-1).\mathtt{thk} \wedge \mathtt{i.hungry})$ CTL, 10, 17
19. $\mathsf{AG}((\mathtt{i}+1).\mathtt{v_2} \wedge \neg(\mathtt{i}+1).\mathtt{has_R} \wedge \neg\mathtt{i.bathroom} \wedge$
    $\neg(\mathtt{i}-1).\mathtt{bathroom}) \wedge \mathtt{i.thk} \wedge (\mathtt{i}-1).\mathtt{thk} \rightarrow \mathsf{EN}\mathtt{i.eating}$ CTL, 18, lemma 12

Proof of lemma 14.

1.	$(i-1).eating \rightarrow ANDone(i.getthk \sqcup i.getbad)$	**Phil20**
2.	$(i-1).eating \rightarrow AN(Done(i.getthk) \vee Done(i.getbad))$	**T14**
3.	$[(i-1).getbad](i-1).bad$	**Phil22**
4.	$ANDone((i-1).getbad) \wedge$ $[(i-1).getbad](i-1).bathroom \rightarrow AN(i-1).bathroom$	**T17**,**TempAx1**, **TempAx2**
5.	$(i-1).eating \rightarrow ANDone((i-1).getthk) \vee ANDone((i-1).getbad)$	CTL, 2
6.	$(i-1).eating \rightarrow ANDone((i-1).getthk) \vee AN(i-1).bathroom$	PL, 3, 4, 5
7.	$AG\neg(i-1).bathroom \rightarrow AN\neg(i-1).path$	CTL
8.	$(i-1).eating \wedge AG\neg(i-1).bathroom \rightarrow ANDone(i.getthk) \vee AN\bot$	CTL, 6, 7
9.	$(i-1).eating \wedge AG\neg(i-1).bathroom \rightarrow ANDone(i.getthk)$	CTL, 8
10.	$[(i-1).getthk](i-1).thk \wedge [\mathbf{U}]Done((i-1).getthk) \rightarrow (i-1).thk$	**T17**
11.	$(i-1).eating \wedge AN\neg(i-1).bathroom \rightarrow AN(i-1).thk$	CTL, 9 ,10
12.	$AN\neg i.bathroom \rightarrow A(i.hungry \vee i.thk \vee i.eating)$	PL, **Phil3**
13.	$AG((i+1).v_2 \wedge \neg(i+1).has_R \wedge \neg i.bathroom\neg(i-1).bathroom) \wedge$ $i.hungry \vee i.thk \vee i.eating) \wedge$ $(i-1).eating \rightarrow EFi.eating$	CTL 11, 12, Lemma 16 Lemma 12

Proof of lemma 15.

1.	$(i-1).hungry \wedge (fork_{i-1}.down \vee (i-1).has_L) \wedge (fork_i.down \wedge (i-1).has_R)$ $\rightarrow AN(i-1).eating$	**Phil15**
2.	$i.thk \rightarrow \neg i.has_L \wedge \neg i.has_R$	Lemma 10
3.	$\neg i.has_R \rightarrow (i-1).has_L \vee fork_{i-1}.down$	**Phil10**
4.	$i.thk \rightarrow (i-1).has_L \vee fork_{(i-1)}.down$	PL, 2, 3
5.	$i.thk \wedge AN\neg i.bathroom \wedge (i-1).hungry \rightarrow ENi.thk$	lemma 16
6.	$i.thk \wedge AN\neg i.bath \wedge (i-1).hungry \rightarrow EN(i.thk \wedge (i-1).eating)$	CTL, 1, 4, 5
7.	$AG((i+1).v_2 \wedge \neg(i+1).has_R \wedge \neg i.bathroom \wedge \neg(i-1).bathroom) \wedge$ $i.thk \wedge (i-1).hungry \rightarrow EN(i.thk \wedge (i-1).eating)$	CTL, 6
8.	$AG((i+1).v_2 \wedge \neg(i+1).has_R \wedge \neg i.bathroom \wedge \neg(i-1).bathroom) \wedge$ $i.thk \wedge (i-1).hungry \rightarrow EF(i.eating)$	Lemma 14

Proof of lemma 16.

1.	$\neg((i-1).hungry \wedge (i+1).hungry)$	lemma 11
2.	$(i+1).hungry \rightarrow [(i-1).gethungry]\bot$	PL, **Phil14**
3.	$AN\neg(i-1).bathroom \rightarrow [\mathbf{U}]\neg(i-1).bathroom$	PL, **TempAx1**
4.	$[\mathbf{U}]\neg(i-1).bathroom \wedge [i.getbad]i.bathroom \rightarrow [i.getbad]\bot$	**BA**, **T5**
5.	$[i.getbad]i.bathroom$	**Phil22**
6.	$AN\neg(i-1).bathroom \rightarrow [i.getbad]\bot$	PL, 3, 4, 5
7.	$[(i-1).gethungry]\bot \rightarrow [(i-1).gethungry]\neg(i-1), gethungry$	ML
8.	$\neg(i-1).hungry \rightarrow \overline{[i.gethungry]}\neg(i-1).hungry$	**Phil17**

9.	$\neg(\mathtt{i}-1).\mathtt{hungry} \wedge \mathsf{AN}(\neg(\mathtt{i}-1).\mathtt{bathroom}) \rightarrow [\mathbf{U}]\neg(\mathtt{i}-1).\mathtt{hungry}$	**BA**, PL, 6, 8
10.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow \neg(\mathtt{i}-1).\mathtt{hungry}$	PL, **Phil3**
11.	$(\mathtt{i}-1).\mathtt{thk} \wedge (\mathtt{i}+1).\mathtt{hungry} \wedge \mathsf{AN}\neg(\mathtt{i}-1).\mathtt{bathroom} \rightarrow$ $\mathsf{AN}\neg(\mathtt{i}-1).\mathtt{hungry}$	PL, 9, 10
12.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow [(\mathtt{i}-1).\mathrm{up_L} \sqcup (\mathtt{i}+1).\mathrm{up_L}]\bot$	**Phil4**
13.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow$ $\overline{[(\mathtt{i}-1).\mathrm{up_L} \sqcup (\mathtt{i}-1).\mathrm{up_L}]}(\neg(\mathtt{i}-1).\mathrm{has_L} \wedge (\mathtt{i}-1).\mathrm{has_R})$	PL, **T5**, **Phil6**
14.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow [\mathbf{U}]\neg\mathtt{i}.\mathtt{eating}$	PL, **BA**, 12, 13
15.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow \neg\mathtt{i}.\mathtt{eating}$	PL, **Phil4**
16.	$(\mathtt{i}-1).\mathtt{thk} \rightarrow \mathsf{AN}\neg\mathtt{i}.\mathtt{eating}$	PL, **TempAx1**, 14, 15
17.	$\neg(\mathtt{i}+1).\mathtt{eating} \wedge \neg(\mathtt{i}-1).\mathtt{hungry} \wedge \neg(\mathtt{i}-1).\mathtt{bathroom}$ $\rightarrow (\mathtt{i}-1).\mathtt{thk}$	PL, **Phil4**
18.	$(\mathtt{i}-1).\mathtt{thk} \wedge (\mathtt{i}+1).\mathtt{hungry} \wedge \mathsf{AN}\neg(\mathtt{i}-1).\mathtt{bathroom} \rightarrow$ $\mathsf{AN}((\mathtt{i}-1).\mathtt{thk})$	CTL, 11, 16, 17

Bibliography

[AB08] Zair Abdelouahab and Isaias Braga. An adaptive train traffic controller. In *An Adaptive Train Traffic Controller*, pages 550–555. Springer Netherlands, 2008.

[Aba89] M. Abadi. The power of temporal proofs. In *Theorical Computer Science*, volume 65, 1989.

[Abr06] Jean-Raymond Abrial. Train systems. In *RODIN Book*. Springer, 2006.

[AG92] Anish Arora and Mohamed G. Gouda. Closure and convergence: A formulation of fault-tolerant computing. In *FTCS*, 1992.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77:1–28, 2007.

[AHS09] Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and Concrete Categories: The Joy of Cats*. John Wiley and Sons, 2009. Corrected version of the 1990 book of the same name, available online at http://katmat.math.uni-bremen.de/acc/.

[AK98] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Trans. Software Eng.*, 24:63–78, 1998.

[AL81] Thomas Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice*. Prentice Halls, 1981.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17:507–534, 1995.

[Ang08] Albert J. J. Anglberger. Dynamic deontic logic and its paradoxes. *Studia Logica*, 89, 2008.

[AP94] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. In *Foundations of Software Technology and Theoretical Computer Science, 14th Conference, Madras, India,*, 1994.

[Aqv84] Lennart Aqvist. Deontic logic. In D.M.Gabbay and F.Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 605–714. Kluwer Academic Publishers, 1984.

[Aro92] Anish Arora. *A Foundation of Fault-Tolerant Computing.* PhD thesis, The University of Texas at Austin, 1992.

[ASS94] Adnan Aziz, Thomas R. Shiple, and Vigyan Singhal. Formula-dependent equivalence for compositional CTL model checking. *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA*, 1994.

[Avi67] Algirdas Avizienis. Design of fault-tolerant computers. *Fall Joint Computer Conference AFIPS*, 31:733–743, 1967.

[Avi95] Algirdas Avizienis. The methodology of N-version programming. *R. Lyu, Editor, Software Fault Tolerance, John Wiley and Sons*, 1995.

[Bac87] Ralph-Johan Back. A calculus of refinement for program derivations. Technical report, Abo Akademi, 1987.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. *POPL*, pages 164–176, 1981.

[Bar87] Howard Barringer. The use of temporal logic in the compositional specification of concurrent systems. In A.Galton, editor, *Temporal Logic and their Applications.* Academic Press, 1987.

[BCG87] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing kripke structures in temporal logic. *APSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy.*, 1987.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *LICS*, pages 428–439, 1990.

[Bel87] M. Belzer. Legal reasoning in 3-d. In *ICAIL*, pages 155–163, 1987.

[Ber91] Paul Bernays. *Axiomatic Set Theory.* Dover Publications, 1991.

[BG77] R. Burstall and J. Goguen. Putting theories together to make specifications. In R.Reddy, editor, *Porc. Fifth International Joint Conference on Artificial Intelligence*, 1977.

[Bro03] J. Broersen. *Modal Action Logics for Reasoning about Reactive Systems.* PhD thesis, Vrije University, 2003.

[BRV01] P. Blackburn, M.de Rijke, and Y.de Venema. *Modal Logic.* Cambridge Tracts in Theoretical Computer Science 53, 2001.

[BV01] Philippe Balbiani and Dimiter Vakarelov. Iteration-free PDL with intersection: a complete axiomatization. *Fundam. Inform.*, 45, 2001.

[BW95] Michael Barr and Charles Wells. *Category Theory for Computer Science.* Prentice Halls, 1995.

[CC96] Antonio Cau and Pierre Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Inf.*, 33:153–176, 1996.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. *POPL*, 1983.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8, 1986.

[CH03] Zhou Chaochen and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems.* Springer-Verlag, 2003.

[Che99] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1999.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40:269–276, 1991.

[CJ96] Jose Carmo and Andrew J. I. Jones. Deontic database contraints, violation and recovery. *Studia Logica*, 57(1):139–165, 1996.

[CJ07] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17:807–841, 2007.

[CK73] C. C. Chang and H. J. Keisler. *Model Theory.* North-Holland, 1973.

[CLM89] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. *Fourth Annual Symposium on Logic in Computer Science, LICS*, 1989.

[CM81] K. Many Chandy and Jayadev Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering 7*, pages 417–426, 1981.

[CM88] K. Many Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CM07a] P.F. Castro and T.S.E. Maibaum. A complete and compact deontic action logic. In *The 4th International Colloquium on Theoretical Aspects of Computing.* Springer Berlin, 2007.

[CM07b] P.F. Castro and T.S.E. Maibaum. An ought-to-do deontic logic for reasoning about fault-tolerance: The diarrheic philosophers. In *5th IEEE International Conference on Software Engineering and Formal Methods.* IEEE, 2007.

[CM07c] P.F. Castro and T.S.E. Maibaum. Reasoning about system-degradation and fault-recovery with deontic logic. In *Workshop on Methods, Models and Tools for Fault-Tolerance*, 2007.

[CM08] P.F. Castro and T.S.E. Maibaum. A tableaux system for deontic action logic. In *Deontic Logic in Computer Science, 9th International Conference, DEON 2008, Luxembourg, Luxembourg, July 15-18, 2008. Proceedings.* Springer, 2008.

[CM09] Pablo F. Castro and T.S.E. Maibaum. Deontic action logic, atomic boolean algebra and fault-tolerance. *Accepted for publication in Journal of Applied Logic (Feb 27)*, 2009.

[Coe94] Jos Coenen. *Formalisms for Program Reification and Fault Tolerance.* PhD thesis, Tenische Universiteit Eindhoven, 1994.

[Cri85] Flaviu Cristian. A rigorous approach to fault-tolerant programming. *IEEE Trans. Software Eng.*, 11:23–31, 1985.

[Dan84] Ryszard Danecki. Non-deterministic propositional dynamic logic with intersection is decidable. In *A. Skowron (ed.), Computation Theory, Springer-Verlag*, 1984.

[dBCG92] Frank S. de Boer, J. Coenen, and Rob Gerth. Exception handling in process algebra. In *NAPAW 92, Proceedings of the First North American Process Algebra Workshop*, 1992.

[Den76] Peter Denning. Fault tolerant operating systems. *ACM Computing Surverys*, 8:359–389, 1976.

[Dij71] E.W. Dijkstra. Hirarchical ordering of sequential processes. In *Acta Informatica*, volume 1, pages 115–138. Springer-Verlag, 1971.

[Dij72] Edsger Dijkstra. Notes on structured programming. *Structured Programming, O.-J Dahl, E.W. Dijkstra, and C.A.R. Hoare eds. Academic Press.*, pages 1–82, 1972.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[Dix83] Trevor I. Dix. Exceptions and interrupts in CSP. *Sci. Comput. Program.*, 3:189–204, 1983.

[DK97] F. Dignum and R. Kuiper. Combining dynamic deontic logic and temporal logic for the specification of deadlines. In *Proceedings of the thirtieth HICSS*, 1997.

[DM00] Carlos H. C. Duarte and T. S. E. Maibaum. A rely-guarantee discipline for open distributed systems design. *Inf. Process. Lett.*, 74:55–63, 2000.

[dR98] Maarten de Rijke. A system of modal logic. *Journal of Philosophical Logic*, 27:109–142, 1998.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12:656–666, 1983.

[DV95] Rocco DeNicola and Frits Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42:458–487, 1995.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland.*, 1980.

[EH82] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *14th Annual Symposiun on Theory of Computing (STOC)*, 1982.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33:151–178, 1986.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1985.

[Eme72] E.A. Emerson. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Eme90] E.A. Emerson. Temporal and modal logic. In *Handbook of Theorical Computer Science*, volume Formal Methods and Semantics (B), 1990.

[Fia05] José Luiz Fiadeiro. *Categories for Software Engineering.* Springer-Verlag, 2005.

[Fin75] Kit Fine. Normal forms in modal logics. In *Notre Dame Journal of Formal Logic*, volume XVI, pages 229–237, 1975.

[Fit72] M. Fitting. Tableau methods of proof for modal logics. In *Notre Dame Journal of Formal Logic*, volume XIII, April 1972.

[Fit83] Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics.* Volume 169 of Synthese Library, 1983.

[Fit90] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Jacob T. Schwartz (ed.), Mathematical Aspects of Computer Science, American Mathematical Society Proc. Symposia in Applied Mathematics*, 19, 1967.

[FM91a] José Luiz Fiadeiro and T. S. E. Maibaum. Temporal reasoning over deontic specifications. *J. Log. Comput.*, 1:357–395, 1991.

[FM91b] José Luiz Fiadeiro and T.S.E. Maibaum. Towards object calculi. In Sernadas A Saake G, editor, *Information Systems; Correctness and Reusability.* Technische Universität Braunschweig, 1991.

[FM92] José Luiz Fiadeiro and T.S.E. Maibaum. Temporal theories as modularization units for concurrent system specification. In *Formal Aspects of Computing*, volume 4, pages 239–272, 1992.

[FM93] José Luiz Fiadeiro and T. S. E. Maibaum. Generalising interpretations between theories in the context of (pi-) institutions. In *Theory and Formal Methods*, pages 126–147, 1993.

[FM97] José Luiz Fiadeiro and T. S. E. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[FS87] José Luiz Fiadeiro and Amílcar Sernadas. Structuring theories on consequence. In *Recent Trends in Data Type Specification, 5th Workshop on Abstract Data Types, Gullane, Scotland, Selected Papers*, pages 44–72, 1987.

[FvG99] W.H.J. Feijen and A.J.M. van Gasteren. *On a Method of Multiprogramming.* Springer-Verlag, 1999.

[Gab96] Dov M. Gabbay. *Labelled Deductive Systems, Volume 1.* Oxford University Press, 1996.

[Gär98] Felix Gärtner. Specification for fault-tolerance: A comedy of failures. Technical report, Darmstadt University of Technology, 1998.

[GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. In *Journal of the Association of Computing Machinery*, 1992.

[Gen69] Gentzen. Investigation into logical deduction. *M. E. Szabo (ed.), The Collected Papers of Gerhard Gentzen, North-Holland*, 1969.

[GLL+00] Stefania Gnesi, Diego Latella, Gabriele Lenzini, C. Abbaneo, Arturo M. Amendola, and P. Marmo. An automatic SPIN validation of a safety critical railway control system. In *International Conference on Dependable Systems and Networks*, 2000.

[GM96] G. Giacomo and F. Massacci. Tableaux and algorithms for propositional dynamic logic with converse. In *Conference on Automated Deduction*, 1996.

[Gol82] Rob Goldblatt. *Axiomatising the Logic of Computer Programming.* Springer, 1982.

[Gor95] Rajeev Goré. Tableau methods for modal and temporal logics. Technical Report TR-ARP-15-95, Australian National University, 1995.

[GP90] G. Gargov and S. Passy. A note on boolean logic. In P.P.Petkov, editor, *Proceedings of the Heyting Summerschool.* Plenum Press, 1990.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal basis of fairness. *POPL*, pages 163–173, 1980.

[HC96] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic.* Routledge, 1996.

[HG93] Claude Hennebert and Gérard D. Guiho. SACEM: A fault tolerant system for train speed control. In *The Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, pages 624–628, 1993.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. *ICALP*, pages 299–309, 1980.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computing programming. *Commun. ACM*, 12:576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23:279–295, 1997.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[Jan95] Tomasz Janowski. *Bisimulation and Fault-Tolerance.* PhD thesis, Department of Computer Science, University of Warwick, 1995.

[Jip92] Peter Jipsen. *Computer Aided Investigations of Relational Algebras.* PhD thesis, Vanderbilt University, 1992.

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[JT51] Bjarni Jonsson and Alfred Tarski. Boolean algebras with operators i. *Amer. J, Math.*, 73:891–939, 1951.

[JT52] Bjarni Jonsson and Alfred Tarski. Boolean algebras with operators ii. *Amer. J, Math.*, 74:127–162, 1952.

[Ken91] Stuart Kent. A deduction calculus for modal action logic with action combinators. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, FOREST Project, 1991.

[Kho88] Samit Khosla. *System Specification: A Deontic Approach.* PhD thesis, Imperial College, 1988.

[KJ08] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in alloy. In *Abstract State Machines, B and Z.* Springer Berlin / Heidelberg, 2008.

[KM85] S. Khosla and T.S.E. Maibaum. The prescription and description of state-based systems. In H.Barringer B.Banieqnal and A.Pnueli, editors, *Temporal Logic in Computation.* Springer-Verlag, 1985.

[KMQ93] S. Kent, T.S.E. Maibaum, and W. Quirk. Formally specifying temporal constraints and error recovery. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 208–215, 1993.

[KP93] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[KQM91] S. Kent, B. Quirk, and T.S.E. Maibaum. Specifying deontic behaviour in modal action logic. Technical report, Forest Research Project, 1991.

[Kro87] Fred Kroger. *Temporal Logic of Programming.* Springer-Verlag, 1987.

[LA90] P.A. Lee and T. Anderson. *Fault-Tolerance, Principles and Practice.* Springer-Verlag, 1990.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 2:125–143, 1977.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5:190–222, 1983.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.

[LF97] Antonia Lopes and José Luiz Fiadeiro. Preservation and reflection in specification. *AMAST*, pages 380–394, 1997.

[LJ92] Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Asp. Comput.*, 4:442–469, 1992.

[LM94] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems - ProCoS*, pages 41–76, 1994.

[LMJ93] Luiz A. Laranjeira, Miroslaw Malek, and Roy M. Jenevein. Nest: A nested-predicate scheme for fault tolerance. *IEEE Trans. Computers*, 42:1303–1324, 1993.

[LR93] Patrick Lincoln and John M. Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In *5th International Conference os Computer Aide5th International Conferenced Verification, CAV '93, Elounda, Greece*, 1993.

[LS77] E. J. Lemmon and Dana Scott. *The "Lemmon Notes": An Introduction to Modal Logic.* Oxford: Blakwell, 1977.

[LS93] P. J. A. Lentfert and S. Doaitse Swierstra. Towards the formal design of self-stabilizing distributed algorithms. In *STACS 93, 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 440–451, 1993.

[LS04] Alessio Lomuscio and Marek J. Sergot. A formalisation of violation, error recovery, and enforcement in the bit transmission problem. *Journal of Applied Logic*, 2:93–116, 2004.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[LZ75] Barbara H. Likov and Stephen N. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, 1, 1975.

[Mac98] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1998.

[Mai87] T. S. E. Maibaum. A logic for the formal requirements specification. Technical report, Imperial College, London.Deliverable R3 for FOREST, 1987.

[Mai93] T.S.E. Maibaum. Temporal reasoning over deontic specifications. In John & Wiley Sons, editor, *Deontic Logic in Computer Science*, 1993.

[MC79] Caver Mead and Lynn Conway. *Introduction to VLSI systems.* Addison-Wesley, 1979.

[MCBG88] E. A. Emerson M. C. Browne and O. Grumberg. Characterizing finite kripke structures in propositional temporal logics. *Theoret. Comput. Sci.*, 59:115–131, 1988.

[McM00] Kenneth L. McMillan. The SMV system. Technical report, available at http://www.cs.cmu.edu/ modelcheck/smv.html, 2000.

[McN06] Paul McNamara. Deontic logic. Technical report, The Stanford Encyclopedia of Philosophy, 2006.

[Men79] Elliott Mendelson. *Introduction to Mathematical Logic.* D. van Nostrand Company, 1979.

[Mey88] J.J. Meyer. A different approach to deontic logic: Deontic logic viewed as variant of dynamic logic. In *Notre Dame Journal of Formal Logic*, volume 29, 1988.

[MG00] Heiko Mantel and Felix C. Gärtner. A case study in the mechanical verification of fault tolerance. In *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference*, 2000.

[Mil79] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1979.

[Mil89] Robin Milner. *Comunication and Concurrence.* Prentice-Hall, 1989.

[MK99] Jeff Magee and Jeff Kramer. *Concurrence: State Models and Java Programs.* Wiley, 1999.

[MM92] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic.* Springer-Verlag, 1992.

[MM06] J. Magee and T.S.E. Maibaum. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceeding of the 2006 international workshop on self-adaptation and self-managing systems*, 2006.

[Mon76] J.D. Monk. *Mathematical Logic.* Graduate Texts in Mathematics. Springer-Verlag, 1976.

[Mor02] Luc Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *17th ACM Symposium on Applied Computing*, 2002.

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. *POPL*, pages 141–154, 1983.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. De Bakker, W. P. De Roover, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, pages 201–284, 1989.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. *PODC*, pages 377–410, 1990.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inf. Comput.*, 100:1–40, 1992.

[MT84] T. S. E. Maibaum and Wladyslaw M. Turski. On what exactly is going on when software is developed step-by-step. *ICSE*, pages 528–533, 1984.

[MWD94] J.J. Meyer, R.J. Wieringa, and F.P.M. Dignum. The paradoxes of deontic logic revisited: A computer science perspective. Technical Report UU-CS-1994-38, Utrecht University, 1994.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *CADE*, 1992.

[Par72] D. Parnas. A technique for software module specification with exampes. In *Communications ACM 15*, pages 330–336, 1972.

[Par78] Rohit Parikh. The completeness of propositional dynamic logic. *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland.*, 64, 1978.

[Pel91] Jan Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5:95–106, 1991.

[PJ94] Doron Peled and Mathai Joseph. A compositional framework for fault tolerance by specification transformation. *Theor. Comput. Sci.*, 128:99–125, 1994.

[Pnu77] Amir Pnueli. The temporal logic of programs. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science.*, pages 46–67, 1977.

[Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. *FOCS*, pages 109–121, 1976.

[Pra78] V.R. Pratt. *A Practical Decision Method for Propositional Dynamic Logic.* ACM Symposium on Theory of Computing, 1978.

[PS05] I. S. W. B. Prasetya and S. Doaitse Swierstra. Formal design of self-stabilizing programs. *J. High Speed Networks*, 14:59–83, 2005.

[QS98] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, 1998.

[Rey01] M. Reynolds. An axiomatization of full computation tree logic. In *Journal of Symbolic Logic*, volume 11, pages 1011–1057, 2001.

[RFM91] M. Ryan, José Luiz Fiadeiro, and T.S.E. Maibaum. Sharing actions and attributes in modal action logic. In *Theoretical Aspects of Computer Software.* Springer-Verlag, 1991.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Automata, Languages and Programming, 24th International Colloquium*, 1997.

[Rya90] Mark Ryan. Structured MAL. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, FOREST Report, 1990.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32:733–749, 1985.

[SC06] Marek J. Sergot and Robert Craven. The deontic component of action language nC+. *DEON*, pages 222–237, 2006.

[Sch04] Klaus Schneider. *Verification of Reactive Systems, Formal Methods and Algorithms.* Springer, 2004.

[SECH98] Francis Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *3rd International Conference on Requirements Engineering (ICRE '98)*, 1998.

[Sik69] R. Sikorski. *Boolean Algebras.* Springer-Verlag, 1969.

[Smu68] R.M. Smullyan. *First-Order Logic.* Springer-Verlag New York, 1968.

[SP94] Marek J. Sergot and Henry Prakken. Contrary-to-duty obligations. In *DEON 94 (Proc.Second International Workshop on Deontic Logic in Computer Science)*, 1994.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1:222–238, 1983.

[SS98] Saniel Siewiorek and Robert Swarz. *Reliable Computer Systems: Design and Evaluation.* A.K. Peters, 1998.

[ST87] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[SW91] C. Stirling and D. Walker. Local model checking in modal mu-calculus. In *Theorical Computer Science*, volume 89, pages 161–177, 1991.

[Tar56] Alfred Tarski. *On the Concept of Logical Consequence.* Translation in : Logic, Semantics, Metamathematics. Oxford University Press, 1956.

[TP00] Wilfredo Torres-Pomales. Software fault-tolerance: A tutorial. *NASA Technical Memorandum TM-2000-210616*, 2000.

[vB76] J. van Benthem. *Modal Correspondence Theory.* PhD thesis, Mathematisch Instituut & Instituut voor Gronslagenonderzoek, University of Amsterdam, 1976.

[vdM96] Ron van der Meyden. The dynamic logic of permission. *J. Log. Comput.*, pages 465–479, 1996.

[vGW89] Rob J. van Glabbeek and W. P. Weijland. Refinement in branching time semantics. In *Proceedings of the AMAST Conference. Iowa*, 1989.

[WM93] Roel J. Wieringa and John-Jules Meyer. Applications of deontic logic in computer science: A concise overview. *Deontic Logic in Computer Science, Normative System Specification*, 1993.

[YB09] Divakar Yadav and Michael Butler. Formal development of a total order broadcast for distributed transactions using Event-B. In *Methods, Models and Tools for Fault Tolerance.* Springer, 2009.

[YTK01] T. Yokogawa, T. Tsuchiya, and T. Kikuno. Automatic verification of fault tolerance using model checking. In *Pacific Rim International Symposium on Dependable Computing.*, 2001.

[Zha08] Bo Zhang. Formal analysis of a distributed fault tolerant clock synchronization algorithm for automotive communication systems. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:393–400, 2008.

Index of Symbols

Index