TÉCNICO LISBOA

Model Checking with the Event-Based Time-Stamped Claim Logic

Candidate Simão Ferreira Rodrigues Graça Leal

Thesis to obtain the Master of Science Degree in

Applied Mathematics and Computation

Supervisors: Prof. Jaime Arsénio de Brito Ramos
Prof. João Filipe Quintas dos Santos Rasga

Examination Committee

Chairperson: Prof. TBD
Supervisor: Prof. TBD
Member of the Committee: Prof. TBD

Month Year

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First I would like to thank my supervisors – Professors Jaime Ramos and João Rasga – for helping me during the development process of this thesis, and especially for having set deadlines for me. I could not have done this without them.

I want to thank my family for all the support they gave me throughout the years. In particular, I would like to thank my father, mother and sister who have to bear with me every day, my grandmothers who were key for my education, and my aunts uncles and cousins, who make my life so much more interesting.

I want to thank my music teachers – Mafalda Nascimento, Hugo Fernandes, Ricardo Mota and Cristina Almeida – and my quartet colleagues (and friends) – Maria Pinho, Teresa Nogueira, Inês Chambel and Miguel Amil. Playing the cello brings me joy every day, and it is a delight to get to do it with all of them. A special thanks also goes to my mother, who enrolled me in cello classes at the age of six without really asking me if I wanted.

I want to thank my teachers both in school and university for sparking my interest in their respective areas. I am fully convinced I chose to pursuit mathematics because I had the good fortune of never having a bad mathematics teacher.

I want to thank my lifelong friends Luís Santos and Hugo Pereira, who have been there for me my whole life (give or take). I would also like to thank my friend Vojin Crnjanski, who became one of my best friends in a short amount of time.

I would like to thank all the great friends and colleagues I met at university. They were crucial for the development of this thesis, as one needs to have a deep knowledge of mathematics to understand the struggles of studying it. In no particular order: Lourenço Abecasis, Guilherme Rodrigues, Francisco Araújo, Henrique Navas, Maria Madrugo, Luís Maia, Bruno Pires, António Gouveia, Eduardo Skapinakis, Ricardo Quintas and Patrícia Roxo.

Finally, I would like to thank and apologize to everyone who I forgot to thank.

All these people made me who I am today. Thank you.

Resumo

A Lógica de Afirmações Temporalmente Etiquetadas e Eventos permite raciocinar sobre afirmações proferidas por múltiplos agentes (alguns mais confiáveis que outros) em diferentes alturas. Esta lógica é particularmente útil em áreas em que a informação é providenciada de forma assíncrona por diversas fontes, como a História ou o sistema judicial. Nesta tese nós desenvolvemos verificação de modelos para esta lógica. Ou seja, dado um sistema ou estrutura de interesse, queremos verificar se satisfaz uma dada propriedade exprimida pela linguagem da Lógica de Afirmações Temporalmente Rotuladas Baseada em Eventos. Começamos por provar que a satisfiabilidade e validade de uma fórmula é decidível e implementamos os respetivos processos de decisão em Java. Definimos uma estrutura de modelação – o sistema de transição – e usamo-lo como a representação final do sistema que queremos modelar. Definimos outras estruturas de modelação mais complexas. Usamos as fórmulas da lógica para exprimir as propriedades que os modelos devem satisfazer. Provamos que a satisfiabilidade de uma fórmula por um sistema de transição é decidível e implementamos o respetivo procedimento de decisão. Desenvolvemos a linguagem de programação ClaimLang e o respetivo compilador, o que facilita consideravelmente o processo de modelação e a especificação de propriedades.

Palavras-chave: Lógica de Afirmações Temporalmente Rotuladas Baseada em Eventos, Verificação de Modelos, Decidibilidade, Sistema de Transição, ClaimLang

Abstract

The Event-Based Time-Stamped Claim Logic allows one to reason about claims asserted by multiple agents (some more trustworthy than others) at different points in time. This logic is particularly useful for areas where evidence is asynchronously provided by several sources, like History or the judicial system. In this thesis we develop model checking for this logic. That is, given some system or structure of interest, we want to check if it satisfies some property expressed in the language of the Event-Based Time-Stamped Claim Logic. We start by proving that the satisfiability and validity of a formula is decidable and implement the decision procedures in Java. We define a modelling structure – the transition system – and use it as the final representation of the system we are modelling. We define other more complex modelling structures. We use formulas to express the properties that the models should satisfy. We prove that the satisfiability of a formula by a transition system is decidable and implement the decision procedure. We develop the ClaimLang programming language and its compiler which considerably facilitates the modelling process and property specification.

Keywords: Event-Based Time-Stamped Claim Logic, Model Checking, Decidability, Transition System, ClaimLang

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Figures xiii
List of Algorithms xv
List of Codes xvii
Nomenclature xix

1 Introduction **1**
1.1 The Event Based Time Stamped Claim Logic 1
1.2 Model Checking 2
1.3 Outline 4

2 The Event-Based Time-Stamped Claim Logic **5**
2.1 Syntax 5
2.2 Semantics 7

3 The Decidability of the Event-Based Time-Stamped Claim Logic **11**
3.1 Local Satisfiability 11
3.2 Global Satisfiability 22

4 Model Checking **35**
4.1 Transition System 35
4.2 Defining a Transition System 41
4.3 Decidability 45

5 Implementation **51**
5.1 Algorithms 51
5.1.1 Generating Time Chains 52
5.1.2 Satisfaction of a Formula 53
5.1.3 Path-Satisfaction of a Formula by a Transition System 53
5.2 The State Explosion Problem 56
5.2.1 Transition System Complexity 56

5.2.2 Formula Complexity . . . 57
5.2.3 Time Order . . . 58
5.3 ClaimLang . . . 60
5.4 Application Examples . . . 62
5.4.1 Three Friends . . . 62
5.4.2 Smug Thieves . . . 66

6 Conclusion **73**
6.1 Achievements . . . 73
6.2 Future Work . . . 74

Bibliography **75**

A ClaimLang Code **77**
A.1 Documentation . . . 77
A.2 Three Friends . . . 81
A.2.1 ClaimLang file . . . 81
A.2.2 Output . . . 83
A.3 Smug Thieves . . . 84
A.3.1 Output . . . 86

B Lexer and Grammar specification **91**
B.1 Formula Parser . . . 91
B.1.1 Lexer Specification . . . 91
B.1.2 Grammar Specification . . . 92

C Java Code **95**
C.1 World.java . . . 95
C.2 Sat.java . . . 96

List of Figures

1.1 The different phases of the model checking process 3

3.1 Abstract representation of the graph structure used to verify the satisfaction of Ω_0 22
3.2 The two inequalities of definition 3.14 visualized. 25
3.3 Graph structure for deciding if $\{\mathsf{F}\varphi\}$ is satisfiable. 25
3.4 The "short circuit" step illustrated. 33

4.1 A very simple transition system. 36
4.2 Two transition systems and their product. 42
4.3 An event transition system. 43
4.4 The conversion of the event transition system of figure 4.3 into a transition system. 44

5.1 A transition system. 56
5.2 A family of transition systems. 57
5.3 The number of branches induced by the formula τ_Σ of definition 3.10 versus the number of branches induced by (1) in function of $n = \#(\mathbb{T})$. 59
5.4 A lengthy conversation between Alice, Bob and $\mathrm{Charlie}$, modeled as a transition system. . 64
5.5 The three relevant agents modeled as event transition systems. 68

List of Algorithms

1 Algorithm for generating all ordered partitions of a set of time-stamps. 52
2 Algorithm for deciding if a set of formulas is decidable using BFS. 54
3 Algorithm for deciding if a transition system path-satisfies a set of formulas using BFS. . . 55

List of Codes

5.1 Part of the Java code from the `Or` class used to implement the $\vee$ connective. 52
5.2 Java code for defining the transition system of figure 4.1 60
5.3 ClaimLang code for defining the transition system of figure 4.1 61
A.1 documentation.claim 77
A.2 three_friends.claim 81
A.3 three_friends_output.txt 83
A.4 smug_thieves.claim 84
A.5 smug_thieves_output.txt 86
B.1 Formula parser: scanner.flex 91
B.2 Formula parser: parser.cup 92
C.1 World.java 95
C.2 Sat.java 96

Nomenclature

Symbols

Σ	Signature
$\mathbb{P}$	Set of propositional symbols
$\mathbb{E}$	Set of event symbols
$\mathbb{T}$	Set of time-stamps
$\mathbb{A}$	Set of agent names

Formulas

K_Σ	Set of claims, i.e., formulas of the type $t \cdot p$ or $-(t \cdot p)$
AK_Σ	Set of agent claims, i.e., formulas of the type $a : t \cdot p$ or $a : -(t \cdot p)$
L_Σ	language of the formulas of the Event-Based Time-Stamped Claim Logic
ϕ	A claim, i.e., an element of K_Σ
φ, ψ	A formula, i.e., an element of L_Σ

Interpretation Structure

M	An interpretation structure
I	A local interpretation structure
$\mathcal{I}$	A sequence of local interpretation structures

Formula Satisfiability

Ω	A set of formulas
Ψ	A set of sets of formulas
Δ, Λ	A set of formulas belonging to $C(\Omega)$
ε_Σ	The “event formula” of definition 3.5.
τ_Σ	The “time-stamp formula” of definition 3.10.

$\longrightarrow$ The “is accessible from” relation between sets of $C(\Omega)$

Model Checking

T A transition system

E An event transition system

S The set of states of a(n) (event) transition system

I The set of initial states of a(n) (event) transition system

L The labelling function of a(n) (event) transition system

$\longrightarrow$ The transition relation of a(n) (event) transition system

ε The null event

π A path over a transition system

Chapter 1

Introduction

Our work can be divided into two main topics: the Event Based Time Stamped Claim Logic and Model Checking.

1.1 The Event Based Time Stamped Claim Logic

The ability to come to conclusions based on multiple, possibly incomplete and contradictory sources is very important to many disciplines, from the judicial system, to History. Different sources might make claims about some occurrence. These sources might not be equally trustworthy and different sources may make contradictory claims. The sources might make their claims at different points in time. The nature of the sources can be vastly different. For example, in the judicial system, a source is typically a witness, an expert or some piece of physical evidence. In History, a source is typically a written document.

The Event-Based Time-Stamped Claim Logic (EBTSCL) introduced in [1] allows us to reason formally with agents which make claims at different instants. It builds upon the Time-Stamped Claim Logic of [2] by adding events and a linear temporal component. The claims used are always of the form $t \cdot p$ or $-(t \cdot p)$, where t is a time-stamp and p is a propositional symbol. The meaning of $t \cdot p$ is that p happened at time-stamp t and the meaning of $-(t \cdot p)$ is that p did not happen at time-stamp t. An agent a might claim that $t \cdot p$ (which we represent as $a : t \cdot p$) or that $-(t \cdot p)$ or a might not make either claim. However, a cannot claim that both $t \cdot p$ and $-(t \cdot p)$ as these are contradictory claims.

It is very important to note that there are two different time concepts are at play. The first one are the time-stamps. These correspond to time when the propositional symbols happen. These can be referred to explicitly with EBTSCL formulas. We can also express that some time-stamp refers to a time point prior to that of another time-stamp, and that two different time-stamps refer to the same time-point. The second one is the time points when the claims are made (we shall refer to them as instants). We refer to these instants using the usual Linear Time Logic (LTL) connectives. To better differentiate the different times, one can imagine that the time-stamps point to a time in a distant past, about which agents make claims along some instants in the current time.

Some agents may be more trustworthy than others depending on the topic being discussed. Events might happen that condition what each agent claims. All these properties can also be expressed with an EBTSCL formula.

One of the main features of the EBTSCL from a set of multiple claims said by specific agents, it can assign a truth value to a claim depending on which agents said about it and how trustworthy the agents are in relation to one another.

In [1] a Hilbert calculus for the EBTSCL is provided and proven to be complete. The satisfiability and validity of an EBTSCL formula is also proven to be decidable. We will not discuss the former, but will focus heavily on the latter.

In a real-life situation the number of agents, time-stamps and claims might be very large, and so it makes sense to have a computational tool which can deal with and assess the validity and satisfiability of some EBTSCL formula. In [3], a Python program which decides the validity of a EBTSCL formula was developed.

In this work we introduced a slightly altered version of the Event-based Time-Stamped Claim Logic. We define its syntax and semantics and prove that the satisfiability and validity of a formula is decidable. The proof is heavily inspired on [1] (i.e., using a tableau-like technique). However, our condition for satisfiability is somewhat more tangible which allows our proof to be translated into an algorithm seamlessly.

We also implemented the logic in Java, representing each formula as a Java object with sub-objects for each subformula. Over this implementation we implemented algorithms which decide the satisfiability and validity of an EBTSCL formula. Finally, a parser was implemented so that one can easily input an EBTSCL formula.

1.2 Model Checking

The goal of model checking is to verify that some structure satisfies some property. Given some structure or system of interest (which from now on we shall call environment), we construct a finite model which fully characterizes the evolution of the environment over time. This model can be non-deterministic (and it should, else model checking is not very interesting).

Given a model for the environment, we will use an EBTSCL formula to encode some property which we want to check if the environment satisfies. The aim of model checking is to create some algorithm which receives an environment (encoded as a finite model) and a property (encoded as an EBTSCL formula) and returns either `Satisfies!` if the environment satisfies the property and `Does not satisfy!` otherwise. In the latter case, it should also provide a counterexample which proves that the environment does not satisfy the property. The phases of the model checking process are illustrated in figure 1.1.

In [4], transition systems are used to model the environment. Some other more intricate structures are also presented. We give two examples

- One can model the environment as a program graph. This enables (among others) the use of variables and conditional statements on the basis of those.

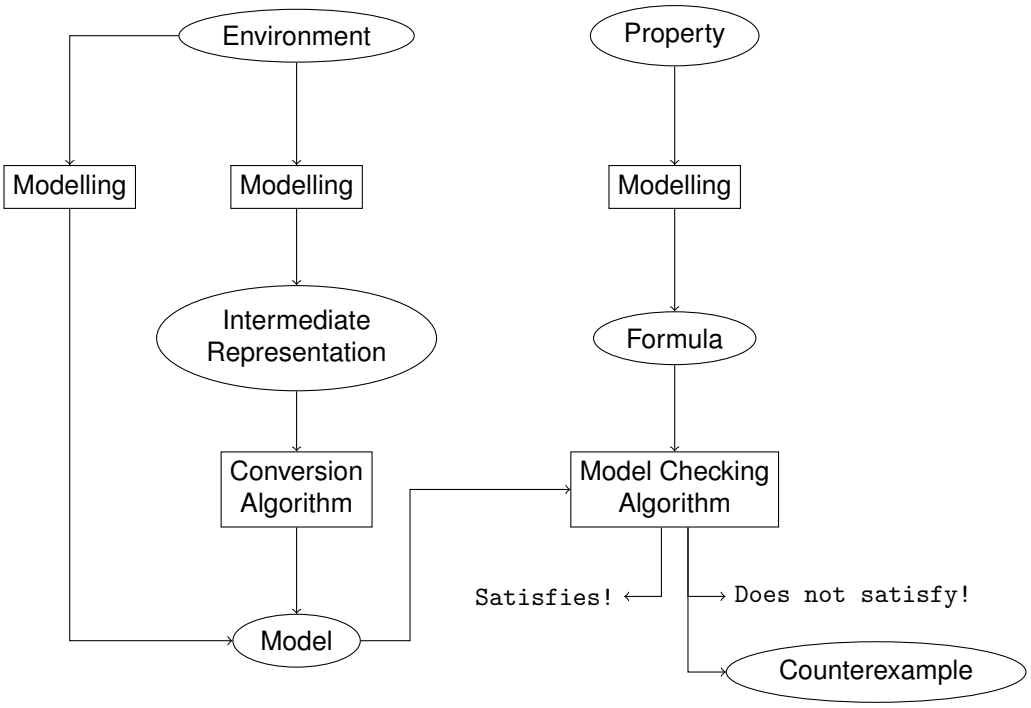


Figure 1.1: The different phases of the model checking process for checking whether “Environment” satisfies “Property”. The nodes in ellipses represent objects, while the nodes in rectangles represent processes. “Environment” and “Property” are abstract objects, while the remaining ones are concrete objects with a computational representation which describes fully them. The “Modelling” process is done manually by the user, while the remaining ones are done automatically.

- On an environment composed of multiple interacting components, one might choose to model each part separately. Then one obtains the model for the whole environment by composing the models for each component in some way. Multiple ways of composing models are presented in [4].

In both cases we are modelling the environment using some intermediate representation which is then automatically converted into the final representation, i.e., the model (in our case, a transition system). This is a good paradigm as modelling with an intermediate representation is typically easier (and less error-prone) than modelling directly with the final representation. Of course, the model checking algorithm can only be as good as the model it is given. If an environment is badly modeled, the conclusions that the model checking algorithm reaches are meaningless. Modelling an environment correctly can be a strenuous task in of itself, so using any tools available to simplify that process is good practice.

In [4], an automata-based LTL model checking procedure is given.

In this work we will introduce the notion of a transition system for environments which deal with agents and claims. We will define the path-satisfiability and satisfiability of an EBTSCL formula by a transition system and prove that we can decide whether a transition system path-satisfies or satisfies a formula.

We also define the event transition system, the product of transition systems and the product of event transition systems. These structures allow for a more natural modelling of the environment in the case of the event transition system and for a complex system to be defined in terms of its more simple components in the case of the product of transition systems. We also define their conversion into a plain transition system. That is, these structures are intermediate representations.

These concepts are all also implemented in Java, along with the algorithm which decides whether a transition system satisfies an EBTSCL. Finally, we created and developed a compiler for a programming language – ClaimLang. This is a simple, very focused programming language which essentially allows the user to input formulas, transition systems and queries to the model-checking algorithm in a very concise manner. This was very important as defining transition systems directly in Java was very time-consuming.
All the code can be found in the GitHub repository:

https://github.com/Simao-Leal/Model-Checking-with-the-EBTSCL

1.3 Outline

- In Chapter 2 we define the syntax and semantics of the Event-Based Time-Stamped Claim Logic.
- In Chapter 3 we define the World function and prove that the satisfiability and validity of a EBTSCL formula is decidable.
- In Chapter 4 we define the transition system structure, define the path-satisfiability and satisfiability of an EBTSCL by a transition system. We define an event transition system and its conversion to a plain transition system and define the product of transition systems and event transition systems. Finally, we prove that the path-satisfiability and the satisfiability of an EBTSCL formula by a transition system is decidable.
- In Chapter 5 we give an overview of how formulas are implemented in Java and present three of the more interesting algorithms used during the development process. We address the State Explosion Problem. We present the ClaimLang programming language and briefly explain how its compiler works. Finally, we present two examples of environments involving agents and claims, model them using the tools we developed and use the model checking algorithm to prove some properties about each environment.

Chapter 2

The Event-Based Time-Stamped Claim Logic

In this chapter we introduce the syntax and semantics of the Event-Based Time-Stamped Claim Logic from [1]. The semantics of some formula components were adapted so that the logic is better suited for model checking We changed the semantics of some formula components, as it made sense in the context of model-checking. These changes will be discussed later as they are introduced.

2.1 Syntax

This section presents the definitions of the Event-Based Time-Stamped Claim Logic (EBTSCL) defined in [1]. We simplified some of these definitions where such simplifications did not meaningfully affect the aim of our work. We also use some slightly different notation for the logical connectives.

Definition 2.1. A signature is a tuple

$$\Sigma = (\mathbb{P}, \mathbb{E}, \mathbb{T}, \mathbb{A})$$

comprising four pairwise disjoint non-empty finite[1] sets: $\mathbb{P}$ is a set of propositional symbols, $\mathbb{E}$ is a set of event symbols, $\mathbb{T}$ is a set of time-stamps and $\mathbb{A}$ is a set of agent names.

From now on we assume fixed a signature Σ.

Definition 2.2. The set K_Σ of time-stamped propositional claims is defined as

$$K_\Sigma = \{-(t \cdot p),\ t \cdot p \mid t \in \mathbb{T} \text{ and } p \in \mathbb{P}\}$$

We use ϕ to denote elements of K_Σ.

Definition 2.3. The language AL_Σ of atomic assertions is defined as follows:

[1] In [1], only $\mathbb{E}$ and $\mathbb{A}$ are finite. $\mathbb{P}$ and $\mathbb{T}$ are allowed to be countably infinite. Since our main goal is to develop a computational model checking system, these sets will forcibly be finite.

- $K_\Sigma \subset AL_\Sigma$;
- $t_1 < t_2 \in AL_\Sigma$ whenever $t_1, t_2 \in \mathbb{T}$;
- $t_1 \cong t_2 \in AL_\Sigma$ whenever $t_1, t_2 \in \mathbb{T}$;
- $a_1 \trianglelefteq_p a_2 \in AL_\Sigma$ whenever $a_1, a_2 \in \mathbb{A}$ and $p \in \mathbb{P}$;
- $a : \phi \in AL_\Sigma$ whenever $a \in \mathbb{A}$ and $\phi \in K_\Sigma$;
- $a : \boxdot \phi \in AL_\Sigma$ whenever $a \in \mathbb{A}$ and $\phi \in K_\Sigma$.

We present the intuitive meaning of these formulas:

- $t \cdot p$ states that p holds at time-stamp t;
- $-(t \cdot p)$ states that p does not hold at time point t;
- $t_1 < t_2$ states that time-stamp t_1 happens before time-stamp t_2;
- $t_1 \cong t_2$ states that t_1 and t_2 represent the same time-stamp;
- $a_1 \trianglelefteq_p a_2$ states that a_1 is less or as trustworthy as a_2 with respect to p;
- $a : t \cdot p$ states that agent a claims that p holds at time-stamp t;
- $a : -(t \cdot p)$ states that agent a claims that p does not hold at time-stamp t;
- $a : \boxdot\, t \cdot p$ states that there is no agent more or as trustworthy as a which claims $-(t \cdot p)$. If you have $a : t \cdot p$ and $a : \boxdot\, t \cdot p$, this means that a claims $t \cdot p$ and no agent at least as trustworthy as a contradicts a.

The following definition will be useful in the future:

Definition 2.4. We denote by AK_Σ the set of all agent claims. That is,

$$AK_\Sigma = \{a : \phi \mid a \in \mathbb{A} \text{ and } \phi \in K_\Sigma\}$$

We now extend AL_Σ by adding propositional and temporal operators to obtain the full language of the Event-Based Time-Stamped Claim Logic.

Definition 2.5. The language L_Σ of the formulas of the Event-Based Time-Stamped Claim Logic is defined as follows:

- $\mathbb{E} \subset L_\Sigma$;
- $AL_\Sigma \subset L_\Sigma$;
- $\top \in L_\Sigma$;
- $\bot \in L_\Sigma$;

- $\neg\varphi \in L_\Sigma$ whenever $\varphi \in L_\Sigma$;
- $\varphi_1 \wedge \varphi_2 \in L_\Sigma$ whenever $\varphi_1, \varphi_2 \in L_\Sigma$;
- $\varphi_1 \vee \varphi_2 \in L_\Sigma$ whenever $\varphi_1, \varphi_2 \in L_\Sigma$;
- $\varphi_1 \implies \varphi_2 \in L_\Sigma$ whenever $\varphi_1, \varphi_2 \in L_\Sigma$;
- $\varphi_1 \iff \varphi_2 \in L_\Sigma$ whenever $\varphi_1, \varphi_2 \in L_\Sigma$;
- $\mathsf{X}\varphi \in L_\Sigma$ whenever $\varphi \in L_\Sigma$;
- $\mathsf{G}\varphi \in L_\Sigma$ whenever $\varphi \in L_\Sigma$;
- $\mathsf{F}\varphi \in L_\Sigma$ whenever $\varphi \in L_\Sigma$;
- $\varphi_1 \mathsf{U} \varphi_2 \in L_\Sigma$ whenever $\varphi_1, \varphi_2 \in L_\Sigma$;

Although the logic allows for double negation, for convenience sake, we will not consider that case. That is, we will treat $\neg\neg\varphi$ to be syntactically equal to φ. The connectives $\top$, $\bot$, $\wedge$, $\vee$, $\iff$, G and F can be defined as abbreviations from the other ones, and indeed that is how they are defined in [1]. However, since we implemented these formulas directly, we consider them to be primitive.

The connectives $\neg$, $\wedge$, $\vee$, $\implies$, $\iff$, $\top$, $\bot$ represent negation, conjunction, disjunction, implication, equivalence, *verum* and *falsum* respectively. As for the temporal connectives,

- $\mathsf{X}\varphi$ states that φ holds in the next instant;
- $\mathsf{G}\varphi$ states that φ always holds in the future;
- $\mathsf{F}\varphi$ states that there is some instant in the future where φ holds;
- $\varphi_1 \mathsf{U} \varphi_2$ states that there is some instant in the future where φ_2 holds and from now until then φ_1 holds.

It is important to mention once again that the time-stamps are not related in any way with the linear temporal instants. Thus, these temporal connectives do not act upon time-stamps. Let $t_1, t_2 \in \mathbb{T}$ and $\varphi \in L_\Sigma$. We write $[\varphi]_{t_2}^{t_1}$ to denote the formula obtained from φ by replacing every occurrence of t_1 by t_2. Similarly, if $a_1, a_2 \in \mathbb{A}$, we write ${}^{p}[\varphi]_{a_2}^{a_1}$ to denote the formula obtained from φ by replacing every occurrence of a_1 by a_2 in matters pertaining to p. That is, we make the replacement in subformulas of the type $a \trianglelefteq_{p'} a'$, $a : t \cdot p'$ and $a : \boxdot t \cdot p'$ only if $p' = p$. Furthermore, when using ϕ to refer to a formula of the type $t \cdot p$ or $-(t \cdot p)$, we use $-\phi$ to refer to its symmetric. That is, if $\phi = t \cdot p$ then $-\phi = -(t \cdot p)$ and if $\phi = -(t \cdot p)$ then $-\phi = t \cdot p$. We also might use $a_1 \trianglelefteq_\phi a_2$ to refer to $a_1 \trianglelefteq_p a_2$.

2.2 Semantics

We now define the semantics of the EBTSCL.

Definition 2.6. An interpretation structure M over Σ is a tuple

$$M = \left(<^M, \cong^M, \{ \trianglelefteq_p^M \}_{p \in \mathbb{P}}, \mathcal{I} \right)$$

where

- $<^M$ is an irreflexive, transitive and connected relation over $\mathbb{T}$;
- $\cong^M$ is an equivalence relation over $\mathbb{T}$, congruent with $<^M$, i.e., let $t_1 \cong^M t_2$, then,
  - if $t_1 <^M t_3$ then $t_2 <^M t_3$,
  - if $t_3 <^M t_1$ then $t_3 <^M t_2$;
- $\trianglelefteq_p^M$ is a transitive and reflexive relation over $\mathbb{A}$ for each $p \in \mathbb{P}$;
- $\mathcal{I}$ is a sequence of local interpretation structures over M (which we define bellow)

$$\mathcal{I} = I_0 \; I_1 \; I_2 \; \ldots$$

Definition 2.7. A local interpretation structure I over an interpretation structure M is a tuple

$$I = (e, C)$$

where

- $e \in E$ is an event symbol;
- $C \subset AK_\Sigma$ is a set of formulas of type agent claim, that is, of the form $a : t \cdot p$ or $a : -(t \cdot p)$, such that
  - if $t_1 \cong^M t_2$ then $a : t_1 : p \in C$ if and only if $a : t_2 : p \in C$;
  - if $t_1 \cong^M t_2$ then $a : -(t_1 : p) \in C$ if and only if $a : -(t_2 : p) \in C$;
  - if $a_1 \trianglelefteq_p^M a_2$ and $a_2 \trianglelefteq_p^M a_1$ then $a_1 : t : p \in C$ if and only if $a_2 : t : p \in C$;
  - if $a_1 \trianglelefteq_p^M a_2$ and $a_2 \trianglelefteq_p^M a_1$ then $a_1 : -(t : p) \in C$ if and only if $a_2 : -(t : p) \in C$

The local interpretation structure corresponds to the atemporal interpretation structure of [1]. Our approach is different in two main ways.
Firstly, in [1] the $<$, $\cong$ and $\trianglelefteq_p$ relations are placed in the atemporal interpretation structure rather than in the interpretation structure. That is, [1] allows for the interpretation of the time-stamp and trust orders to change over time. We decided to not allow this for two reasons

1. Since agents cannot make assertions about the time-stamp and trust orders ($a : t_1 < t_2$ is not a valid formula for example), there is not much of a reason to ever change the time-stamp and trust orders;

2. Allowing the time order to change over time greatly increases the time complexity of the model-checking algorithm. This point will be discussed further in section 5.2.

Secondly, [1] defines an interpretation structure as a tuple (ε, μ) where ε is a sequence of events and μ is a sequence of atemporal interpretation structures. Essentially we pushed the events into the corresponding atemporal interpretation structures.
Thirdly, in [1] the time symbols are interpreted. That is, there is a set $D_{\mathbb{T}}$ of possible time-stamp interpretations, each time symbol t gets an interpretation $t^M \in D_{\mathbb{T}}$ and the relation $<^M$ is over $D_{\mathbb{T}}$ instead of $\mathbb{T}$. We understand that this might make sense if we want, for example, to assign a natural number to each time-stamp. However, for our purposes this would entail unnecessary complications and provide us little benefit.
We now define the satisfaction of a formula by an interpretation structure.

Definition 2.8. Let $M = \left(<^M, \cong^M, \left\{\trianglelefteq_p^M\right\}_{p\in\mathbb{P}}, \mathcal{I}\right)$ be an interpretation structure over Σ, let e_i and C_i be such that $\mathcal{I}(i) = (e_i, C_i)$ for all i. Let $k \in \mathbb{N}$. The satisfaction of a formula φ by M at instant k, denoted by

$$M, k \Vdash \varphi$$

is defined as follows

- $M, k \Vdash t_1 < t_2$ whenever $t_1 <^M t_2$;
- $M, k \Vdash t_1 \cong t_2$ whenever $t_1 \cong^M t_2$;
- $M, k \Vdash a_1 \trianglelefteq_p a_2$ whenever $a_1 \trianglelefteq_p^M a_2$;
- $M, k \Vdash a : \phi$ whenever $a : \phi \in C_k$;
- $M, k \Vdash a : \boxdot\, \phi$ whenever for every $b \in \mathbb{A}$ such that $a \trianglelefteq_\phi^M b$, we have $M, k \nVdash b : -\phi$;
- $M, k \Vdash \phi$ whenever
  - there is $a \in \mathbb{A}$ such that $M, k \Vdash a : \phi$ and $M, k \Vdash a : \boxdot\, \phi$,
  - for each $b \in \mathbb{A}$, if $M, k \Vdash b : -\phi$, then $M, k \nVdash b : \boxdot\, \phi$;
- $M, k \Vdash e$, with $e \in \mathbb{E}$, whenever $e_k = e$;
- $M, k \Vdash \top$ always;
- $M, k \nVdash \bot$ always;
- $M, k \Vdash \neg\varphi'$ whenever $M, k \nVdash \varphi'$;
- $M, k \Vdash \varphi_1 \wedge \varphi_2$, whenever $M, k \Vdash \varphi_1$ and $M, k \Vdash \varphi_1$;
- $M, k \Vdash \varphi_1 \vee \varphi_2$, whenever $M, k \Vdash \varphi_1$ or $M, k \Vdash \varphi_2$;
- $M, k \Vdash \varphi_1 \implies \varphi_2$, whenever either $M, k \nVdash \varphi_1$ or $M, k \Vdash \varphi_2$;

- $M, k \Vdash \varphi_1 \iff \varphi_2$, whenever either $M, k \Vdash \varphi_1$ and $M, k \Vdash \varphi_2$ or $M, k \nVdash \varphi_1$ and $M, k \nVdash \varphi_2$;
- $M, k \Vdash \mathsf{X}\varphi'$ whenever $M, (k+1) \Vdash \varphi'$;
- $M, k \Vdash \mathsf{G}\varphi'$ whenever for every $k' \geq k$ we have $M, k' \Vdash \varphi'$;
- $M, k \Vdash \mathsf{G}\varphi'$ whenever there is $k' \leq k$ such that $M, k' \Vdash \varphi'$;
- $M, k \Vdash \varphi_1 \mathsf{U} \varphi_2$, whenever there is $k' \geq k$ such that $M, k' \Vdash \varphi_2$ and for every $k \leq k'' < k'$ we have $M, k'' \Vdash \varphi_1$.

We extend this definition to sets of formulas in the usual way. Let Ω be a set of formulas. Then,

$M, k \Vdash \Omega$ iff $M, k \Vdash \varphi$ for all $\varphi \in \Omega$

We use anchored semantics to define $M \Vdash \varphi$.

Definition 2.9. For $k \in \mathbb{N}$, we say that an interpretation structure M k-satisfies a formula φ if $M, k \Vdash \varphi$. We say that M satisfies φ, and write $M \Vdash \varphi$ whenever M 0-satisfies φ.

These definitions are extended to sets of formulas in the usual way.
In [1], a floating semantics is used. That is, $M \Vdash \varphi$ whenever $M, k \Vdash \varphi$ for all k. We decided to use anchored semantics, as it makes more sense in the context of model-checking. If needed, one can emulate floating semantics by prepending every formula with the G connective.

Definition 2.10. A formula φ is said to be satisfiable if there is an interpretation structure M such that $M \Vdash \varphi$.

Definition 2.11. We say that a formula φ is valid, and write $\vDash \varphi$, if all interpretation structures M satisfy φ.

The next result establishes a relation between satisfiability and validity.

Proposition 2.12. *Let $\varphi \in L_\Sigma$. Then,*

φ is valid iff *$\neg\varphi$ is not satisfiable.*

Proof.

φ is valid
iff $M \Vdash \varphi$ for every interpretation structure M
iff not(There is an interpretation structure M such that $M \nVdash \varphi$)
iff not(There is an interpretation structure M such that $M \Vdash \neg\varphi$)
iff $\neg\varphi$ is not satisfiable

□

Chapter 3

The Decidability of the Event-Based Time-Stamped Claim Logic

The goal of this chapter is to present computable procedures which allow us to decide whether some EBTSCL formula is satisfiable or valid. If we find that some formula is satisfiable, we must present a witness which attests this – an interpretation structure which satisfies such formula. On the other hand, if a formula is not valid, we must present a counter-example – an interpretation structure which does not satisfy said formula. The chapter follows section 7 of [1], although some definitions and proofs had to be adapted to our case. For one, the definition of World had to be changed to accommodate the different semantics regarding the time and trust relations. Furthermore, instead of using the recursive definition of compatible set of formulas, we used the equivalent concept of a compatible path of sets of formulas, as extracting an interpretation structure from a compatible path is immediate. Although not the main goal of this thesis, we felt that it was important to present and fully prove these results, as the chapter on model-checking relies heavily on some of these definitions and propositions.

3.1 Local Satisfiability

Before defining the world generated by a formula, we discuss its intuitive meaning. Our goal is to decompose each formula into its components, preserving its semantics. $\mathrm{World}(\varphi)$ is a set of set of formulas. One can look at this structure as if it were the decomposition of φ into disjunctive normal form. That is, in order for φ to hold (locally), all the formulas in one of the sets of $\mathrm{World}(\varphi)$ must hold. Contrary to [1], formulas which make claims about time or trust orders are prepended with an X to ensure they are propagated forwards to every following instant. Recall that under our semantics, the time and trust orders remain unchanged at every time instant. Since we are considering the $\top$, $\bot$, $\wedge$, $\vee$, $\iff$, G and F connectives as primitive and not abbreviations, we need to define the behavior of World for them.

Definition 3.1. The world generated by a formula is a map

$$\mathrm{World} : L_\Sigma \to \mathcal{P}(\mathcal{P}(L_\Sigma))$$

defined as follows

1. $\mathrm{World}(t_1 < t_2) = \{\{t_1 < t_2, \mathsf{X}(t_1 < t_2)\}\}$;
2. $\mathrm{World}(\neg t_1 < t_2) = \{\{\neg t_1 < t_2, \mathsf{X}(\neg t_1 < t_2)\}\}$;
3. $\mathrm{World}(t_1 \cong t_2) = \{\{t_1 \cong t_2, \mathsf{X}(t_1 \cong t_2)\}\}$;
4. $\mathrm{World}(\neg t_1 \cong t_2) = \{\{\neg t_1 \cong t_2, \mathsf{X}(\neg t_1 \cong t_2)\}\}$;
5. $\mathrm{World}(a_1 \trianglelefteq_p a_2) = \{\{a_1 \trianglelefteq_p a_2, \mathsf{X}(a_1 \trianglelefteq_p a_2)\}\}$;
6. $\mathrm{World}(\neg a_1 \trianglelefteq_p a_2) = \{\{\neg a_1 \trianglelefteq_p a_2, \mathsf{X}(\neg a_1 \trianglelefteq_p a_2)\}\}$;
7. $\mathrm{World}(a : \phi) = \{\{a : \phi\}\}$;
8. $\mathrm{World}(\neg(a : \phi)) = \{\{\neg(a : \phi)\}\}$;
9. $\mathrm{World}(a : \boxdot\phi) = \left\{\bigcup_{b \in \mathbb{A}} \{a \trianglelefteq_\phi b \implies \neg(b : -\phi)\}\right\}$;
10. $\mathrm{World}(\neg(a : \boxdot\phi)) = \{\{a \trianglelefteq_\phi b, b : -\phi\} \mid b \in \mathbb{A}\}$;
11. $\mathrm{World}(\phi) = \left\{\left(\bigcup_{b \in \mathbb{A}} \{b : -\phi \implies \neg b : \boxdot - \phi\}\right) \cup \{a : \phi, a : \boxdot\phi\} \;\middle|\; a \in \mathbb{A}\right\}$;
12. $\mathrm{World}(\neg\phi) = \left\{\bigcup_{a \in \mathbb{A}} \{a : \phi \implies \neg a : \boxdot\phi\}\right\} \bigcup \{\{b : -\phi, b : \boxdot - \phi\} \mid b \in \mathbb{A}\}$;
13. $\mathrm{World}(\top) = \{\{\}\}$;
14. $\mathrm{World}(\neg\top) = \{\{\bot\}\}$;
15. $\mathrm{World}(\bot) = \{\{\bot\}\}$;
16. $\mathrm{World}(\neg\bot) = \{\{\}\}$;
17. $\mathrm{World}(\varphi_1 \wedge \varphi_2) = \{\{\varphi_1, \varphi_2\}\}$;
18. $\mathrm{World}(\neg(\varphi_1 \wedge \varphi_2)) = \{\{\neg\varphi_1\}, \{\neg\varphi_2\}\}$;
19. $\mathrm{World}(\varphi_1 \vee \varphi_2) = \{\{\varphi_1\}, \{\varphi_2\}\}$;
20. $\mathrm{World}(\neg(\varphi_1 \vee \varphi_2)) = \{\{\neg\varphi_1, \neg\varphi_2\}\}$;
21. $\mathrm{World}(\varphi_1 \implies \varphi_2) = \{\{\neg\varphi_1\}, \{\varphi_2\}\}$;
22. $\mathrm{World}(\neg(\varphi_1 \implies \varphi_2)) = \{\{\varphi_1, \neg\varphi_2\}\}$;
23. $\mathrm{World}(\varphi_1 \iff \varphi_2) = \{\{\varphi_1, \varphi_2\}, \{\neg\varphi_1, \neg\varphi_2\}\}$;
24. $\mathrm{World}(\neg(\varphi_1 \iff \varphi_2)) = \{\{\varphi_1, \neg\varphi_2\}, \{\neg\varphi_1, \varphi_2\}\}$;
25. $\mathrm{World}(e) = \{\{e\}\}$;
26. $\mathrm{World}(\neg e) = \{\{e'\} \mid e' \in \mathbb{E} \backslash \{e\}\}$;[1]

[1] In the case where $\mathbb{E} = \{e\}$ then $\mathrm{World}(\neg e) = \{\{\bot\}\}$ instead of $\{\{\}\}$.

27. $\mathrm{World}\,(\mathsf{X}\varphi) = \{\{\mathsf{X}\varphi\}\}$;

28. $\mathrm{World}\,(\neg\mathsf{X}\varphi) = \{\{\mathsf{X}\neg\varphi\}\}$;

29. $\mathrm{World}\,(\mathsf{G}\varphi) = \{\{\varphi, \mathsf{X}\mathsf{G}\varphi\}\}$;

30. $\mathrm{World}\,(\neg\mathsf{G}\varphi) = \{\{\neg\varphi\}, \{\varphi, \mathsf{X}\neg\mathsf{G}\varphi\}\}$;

31. $\mathrm{World}\,(\mathsf{F}\varphi) = \{\{\varphi\}, \{\neg\varphi, \mathsf{X}\mathsf{F}\varphi\}\}$;

32. $\mathrm{World}\,(\neg\mathsf{F}\varphi) = \{\{\neg\varphi, \mathsf{X}\,\neg\mathsf{F}\varphi\}\}$;

33. $\mathrm{World}\,(\varphi_1\mathsf{U}\varphi_2) = \{\{\varphi_2\}, \{\varphi_1, \neg\varphi_2, \mathsf{X}\,(\varphi_1\mathsf{U}\varphi_2)\}\}$;

34. $\mathrm{World}\,(\neg(\varphi_1\mathsf{U}\varphi_2)) = \{\{\neg\varphi_1, \neg\varphi_2\}, \{\varphi_1, \neg\varphi_2, \mathsf{X}\neg\,(\varphi_1\mathsf{U}\varphi_2)\}\}$;

We extend this map to sets of formulas as follows. Let $\Omega = \{\varphi_1, \ldots, \varphi_n\}$. Then

$$\mathrm{World}(\Omega) = \left\{ \left(\left\{\varphi_1^{\checkmark}\right\} \cup \Omega_1\right) \cup \cdots \cup \left(\left\{\varphi_n^{\checkmark}\right\} \cup \Omega_n\right) \;\middle|\; \text{for each } \Omega_i \in \mathrm{World}(\varphi_i), \text{for each } i = 1, \ldots, n \right\}$$

where we write $\varphi^{\checkmark}$ to mean that the formula should be in the set, but we do not apply rules to it, i.e., $\mathrm{World}(\varphi^{\checkmark}) = \left\{\left\{\varphi^{\checkmark}\right\}\right\}$. The map also extends to a finite set of finite sets of formulas as follows:

$$\mathrm{World}(\{\Omega_1 \ldots \Omega_n\}) = \mathrm{World}(\Omega_1) \cup \cdots \cup \mathrm{World}(\Omega_n)$$

For a finite set Ψ of finite sets of formulas, we denote by $\mathrm{World}^n(\Psi)$ the successive application of World to Ψ n times.

Example 3.2. Let $\mathbb{A} = \{a_1, a_2\}$. As an example, let us compute $\mathrm{World}^3(\{\{a_1 : \boxdot(t \cdot p)\}\})$.
We start by computing

$$\mathrm{World}(a_1 : \boxdot(t \cdot p)) = \{\{(a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))), (a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p)))\}\}$$

Thus,

$$\mathrm{World}(\{\{a_1 : \boxdot(t \cdot p)\}\}) = \{\{(a_1 : \boxdot(t \cdot p))^{\checkmark}, (a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))),\; (a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p)))\}\}$$

Now we compute

$$\mathrm{World}(a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))) = \{\{\neg\,(a_2 : (-t \cdot p))\}, \{\neg a_1 \trianglelefteq_p a_2\}\}$$

Similar for $a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))$. Thus,

$$\mathrm{World}^2(\{\{a_1 : \boxdot(t \cdot p)\}\}) = \{\{\neg\,(a_1 : (-t \cdot p)), ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark},\; \neg a_1 \trianglelefteq_p a_2, ((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\},$$

$$\{\neg\,(a_2 : (-t \cdot p))\,, ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,,$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\,, \neg a_1 \trianglelefteq_p a_1\},$$
$$\{\neg\,(a_2 : (-t \cdot p))\,, \neg\,(a_1 : (-t \cdot p))\,, ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark},$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\},$$
$$\{((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,, \neg a_1 \trianglelefteq_p a_2,$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\,, \neg a_1 \trianglelefteq_p a_1\}\}$$

Finally,

$$\mathrm{World}(a_1 \trianglelefteq_p a_2) = \{\{a_1 \trianglelefteq_p a_2, \mathsf{X}(a_1 \trianglelefteq_p a_2)\}\}$$

Similar for $\neg a_1 \trianglelefteq_p a_2$. Hence,

$$\mathrm{World}^3(\{\{a_1 : \boxdot(t \cdot p)\}\}) =$$
$$= \{\{(\neg\,(a_2 : (-t \cdot p)))^{\checkmark}\,, (\neg\,(a_1 : (-t \cdot p)))^{\checkmark}\,, ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,,$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\},$$
$$\{((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,, (\neg a_1 \trianglelefteq_p a_2)^{\checkmark}\,, ((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\,,$$
$$(\neg a_1 \trianglelefteq_p a_1)^{\checkmark}\,, \mathsf{X}\neg a_1 \trianglelefteq_p a_2, \mathsf{X}\neg a_1 \trianglelefteq_p a_1\},$$
$$\{(\neg\,(a_1 : (-t \cdot p)))^{\checkmark}\,, ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,, (\neg a_1 \trianglelefteq_p a_2)^{\checkmark}\,,$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\,, \mathsf{X}\neg a_1 \trianglelefteq_p a_2\},$$
$$\{(\neg\,(a_2 : (-t \cdot p)))^{\checkmark}\,, ((a_1 \trianglelefteq_p a_2 \implies \neg\,(a_2 : (-t \cdot p))))^{\checkmark}\,,$$
$$((a_1 \trianglelefteq_p a_1 \implies \neg\,(a_1 : (-t \cdot p))))^{\checkmark}\,, (\neg a_1 \trianglelefteq_p a_1)^{\checkmark}\,, \mathsf{X}\neg a_1 \trianglelefteq_p a_1\}\}$$

Now notice that $\mathrm{World}^3(\{\{a_1 : \boxdot(t \cdot p)\}\})$ is a set of sets of indecomposable formulas. That is, formulas φ such that $\mathrm{World}(\varphi) = \{\{\varphi\}\}$. As we will see next, this means that $\mathrm{World}^4(\{\{a_1 : \boxdot(t \cdot p)\}\}) = \mathrm{World}^3(\{\{a_1 : \boxdot(t \cdot p)\}\})$.

This example also illustrates the need for a system which can preform these calculations automatically. Working out World^3 for this very simple formula was already quite cumbersome. [2]

We now prove that the successive application of World to a finite set of finite formulas eventually stabilizes.

Proposition 3.3. *If Ψ is a finite set of finite sets of formulas, then there is an n such that*

$$\mathrm{World}^{n+1}(\Psi) = \mathrm{World}^n(\Psi)$$

We define $\Psi^{\ddagger} = \mathrm{World}^n(\Psi)$ for such an n.

[2] In reality this computation was done automatically. Even the LaTeX code for this example was computer generated, but it was still quite cumbersome to copy-paste it and make it fit the page width.

Proof. Recall that a formula φ is said to be indecomposable if $\mathrm{World}(\varphi) = \{\{\varphi\}\}$. First, note that if Ω is a set of indecomposable formulas then $\mathrm{World}(\Omega) = \{\Omega\}$
Let $\Omega = \{\varphi_1, \ldots, \varphi_n\}$ be a set of indecomposable formulas. That is, $\mathrm{World}(\varphi_i) = \{\{\varphi_i\}\}$. Notice that since we consider φ and $\varphi^{\checkmark}$ to be equal, $\{\varphi\} \cup \{\varphi^{\checkmark}\}$ is equal to $\{\varphi^{\checkmark}\}$. Hence,

$$\mathrm{World}(\{\varphi_1, \ldots, \varphi_n\}) = \{\{\varphi_1^{\checkmark}\} \cup \{\varphi_1\}, \ldots \{\varphi_n^{\checkmark}\} \cup \{\varphi_n\}\} = \{\{\varphi_1^{\checkmark}, \ldots, \varphi_n^{\checkmark}\}\} = \{\Omega\}$$

Secondly, if Ψ is a set of sets of indecomposable formulas, then $\mathrm{World}(\Psi) = \Psi$.
Let $\Psi = \{\Omega_1, \ldots, \Omega_n\}$ where each Ω_i is a set of indecomposable formulas. Then

$$\mathrm{World}(\{\Omega_1, \ldots, \Omega_n\}) = \mathrm{World}(\Omega_1) \cup \cdots \mathrm{World}(\Omega_n) = \{\{\Omega_1\}\} \cup \cdots \cup \{\{\Omega_n\}\} = \{\Omega_1, \ldots, \Omega_n\} = \Psi$$

Now looking at the definition of World for formulas note that cases 7, 8, 15, 25 and 27 consist of indecomposable formulas; cases 1 through 6, 26 and 28 result in a set of sets of indecomposable formulas in one step; case 10 results in a set of sets of indecomposable formulas in two steps; case 9 results in a set of sets of indecomposable formulas in three steps (as seen in example 3.2); cases 11 and 12 result in a set of sets of indecomposable formulas in five steps; all the other cases consist of propositional or temporal formulas whose World results in a set of sets of formulas with lower complexity, so we can conclude that after a finite number of applications of World, a set of sets of indecomposable formulas is reached. □

There are still some aspects of the logic which have not been captured: properties of events, the time and trust relations. We introduce now a notion of closure.

Definition 3.4. Given a set of formulas Ω, we define its $\mathrm{OperatorClosure}$ as the least set for which the following conditions hold. For any $t, t_1, t_2, t_3 \in \mathbb{T}$, $a, a_1, a_2 \in \mathbb{A}$, $p \in \mathbb{P}$ and $\varphi \in L_\Sigma$,

- $\Omega \subset \mathrm{OperatorClosure}(\Omega)$;
- if $t_1 < t_2,\ t_2 < t_3 \in \mathrm{OperatorClosure}(\Omega)$ then $t_1 < t_3 \in \mathrm{OperatorClosure}(\Omega)$;
- $t \cong t \in \mathrm{OperatorClosure}(\Omega)$;
- if $t_1 \cong t_2 \in \mathrm{OperatorClosure}(\Omega)$ then $t_2 \cong t_1 \in \mathrm{OperatorClosure}(\Omega)$;
- if $t_1 \cong t_2,\ t_2 \cong t_3 \in \mathrm{OperatorClosure}(\Omega)$ then $t_1 \cong t_3 \in \mathrm{OperatorClosure}(\Omega)$;
- if $t_1 \cong t_2, \varphi \in \mathrm{OperatorClosure}(\Omega)$ then $[\varphi]_{t_2}^{t_1} \in \mathrm{OperatorClosure}(\Omega)$;
- $a \trianglelefteq_p a \in \mathrm{OperatorClosure}(\Omega)$;
- if $a_1 \trianglelefteq_p a_2,\ a_2 \trianglelefteq_p a_3 \in \mathrm{OperatorClosure}(\Omega)$ then $a_1 \trianglelefteq_p a_3 \in \mathrm{OperatorClosure}(\Omega)$;
- if $a_1 \trianglelefteq_p a_2, a_2 \trianglelefteq_p a_1, \varphi \in \mathrm{OperatorClosure}(\Omega)$ then ${}^{p}[\varphi]_{a_2}^{a_1} \in \mathrm{OperatorClosure}(\Omega)$.

Definition 3.5. We denote by ε_Σ the following formula

$$\varepsilon_\Sigma = \bigvee_{e \in \mathbb{E}} e$$

This formula will get added to guarantee that at least one event happens at every time point. Of course, we will also have to guarantee that at most one event happens, but this is handled later.

Definition 3.6. Let Ω be a set of formulas, and let

$$(\Omega \cup \{\varepsilon_\Sigma\})^\ddagger = \{\Delta_1, \ldots, \Delta_n\}$$

We define by Ω^* the set of sets of formulas

$$\Omega^* = \{\mathrm{OperatorClosure}(\Delta_1), \ldots, \mathrm{OperatorClosure}(\Delta_n)\}$$

If Ω is finite, then we can compute Ω^* in a finite number of steps.
We now prove that the Ω^* construction preserves satisfiability.

Proposition 3.7. *Let Ω be a non-empty finite set of formulas and M an interpretation structure. Then*

$$M, k \Vdash \Omega \text{ iff } \textit{there is } \Delta \in \Omega^* \textit{ such that } M, k \Vdash \Delta$$

Proof. The proof is rather long and consists of five main steps

1. We show that World preserves satisfiability. We must do it for formulas, sets of formulas and sets of sets of formulas. That is, we must show that

$$M, k \Vdash \varphi \text{ iff there is } \Delta \in \mathrm{World}(\varphi) \text{ such that } M, k \Vdash \Delta \tag{1}$$

   We proceed by analyzing each case in the definition of World (definition 3.1). Note that whenever $\mathrm{World}(\varphi)$ consists of only one element $\mathrm{World}(\varphi) = \{\Lambda\}$, "there is $\Delta \in \mathrm{World}(\varphi)$ such that $M, k \Vdash \Delta$" is equivalent to just "$M, k \Vdash \Lambda$".

   - For case 1, $\mathrm{World}(t_1 < t_2) = \{\{t_1 < t_2, \mathsf{X}(t_1 < t_2)\}\}$. Then,

$$\begin{aligned} & M, k \Vdash t_1 < t_2 \\ \text{iff } & t_1 <^M t_2 \\ \text{iff } & M, k \Vdash t_1 < t_2 \text{ and } M, k+1 \Vdash t_1 < t_2 \\ \text{iff } & M, k \Vdash t_1 < t_2 \text{ and } M, k \Vdash \mathsf{X}(t_1 < t_2) \\ \text{iff } & M, k \Vdash \{t_1 < t_2, \mathsf{X}(t_1 < t_2)\} \end{aligned}$$

   - Cases 2-6 are similar to the previous one;

- For cases 7, 8, 15, 25 and 27, $\mathrm{World}(\varphi) = \{\{\varphi\}\}$. Then, $M, k \Vdash \varphi$ iff $M, k \Vdash \{\varphi\}$;
- For case 9, $\mathrm{World}(a : \boxdot\phi) = \left\{\bigcup_{b\in\mathbb{A}}\{a \trianglelefteq_\phi b \implies \neg(b : -\phi)\}\right\}$. Then,

$$
\begin{aligned}
& M, k \Vdash a : \boxdot\phi \\
\text{iff}\ \ & \text{for every } b \in \mathbb{A}\text{, if } a \trianglelefteq_\phi^M b \text{ then } M, k \nVdash b : -\phi \\
\text{iff}\ \ & \text{for every } b \in \mathbb{A}\text{, } M, k \Vdash a \trianglelefteq_\phi^M b \implies \neg(b : -\phi) \\
\text{iff}\ \ & M, k \Vdash \bigcup_{b\in\mathbb{A}}\{a \trianglelefteq_\phi b \implies \neg(b : -\phi)\}
\end{aligned}
$$

- Case 10 is similar to the previous one;
- For case 11, we must prove each implication separately

$\longrightarrow$

$$
\begin{aligned}
& M, k \Vdash \phi \\
\text{implies}\ \ & \text{– there is } a \in \mathbb{A} \text{ such that } M, k \Vdash a : \phi \text{ and } M, k \Vdash a : \phi \\
& \text{– for each } b \in \mathbb{A} \text{ if } M, k \Vdash b : -\phi \text{ then } M, k \nVdash b : \boxdot - \phi \\
\text{implies}\ \ & \text{– there is } a \in \mathbb{A} \text{ such that } M, k \Vdash \{a : \phi, M, k \Vdash a : \phi\} \\
& \text{– } M, k \Vdash \bigcup_{b\in\mathbb{A}}\{b : -\phi \implies \neg b : \boxdot - \phi\} \\
\text{implies}\ \ & \text{there is } a \in \mathbb{A} \text{ such that} \\
& M, k \Vdash \left(\bigcup_{b\in\mathbb{A}}\{b : -\phi \implies \neg b : \boxdot - \phi\}\right) \cup \{a : \phi, a : \boxdot\phi\} \\
\text{implies}\ \ & \text{there is} \\
& \Delta \in \left\{ \left(\bigcup_{b\in\mathbb{A}}\{b : -\phi \implies \neg b : \boxdot - \phi\}\right) \cup \{a : \phi, a : \boxdot\phi\} \;\middle|\; a \in \mathbb{A} \right\} \\
& \text{such that } M, k \Vdash \Delta
\end{aligned}
$$

$\longleftarrow$ We skip some steps, as they are similar to what we did in $\longrightarrow$.

$$
\begin{aligned}
& \text{there is } \Delta \in \mathrm{World}(\phi) \text{ such that } M, k \Vdash \Delta \\
\text{implies}\ \ & \text{there is } a \in \mathbb{A} \text{ such that} \\
& M, k \Vdash \left(\bigcup_{b\in\mathbb{A}}\{b : -\phi \implies \neg b : \boxdot - \phi\}\right) \cup \{a : \phi, a : \boxdot\phi\} \\
& \vdots \\
\text{implies}\ \ & M, k \Vdash \phi
\end{aligned}
$$

- Case 12 is similar to the previous one;
- For case 13, $\mathrm{World}(\top) = \{\{\}\}$. $\top$ is always satisfied by any in interpretation structure. The empty set is also vacuously satisfied by any interpretation structure. Then $M, k \Vdash \top$ iff $M, k \Vdash \{\}$ as both sides of the equivalence are always true;
- Cases 14-16 are similar to the previous one;
- For case 17, $\mathrm{World}(\varphi_1 \wedge \varphi_2) = \{\{\varphi_1, \varphi_2\}\}$. Then

$$
\begin{aligned}
& M, k \Vdash \varphi_1 \wedge \varphi_2 \\
\text{iff } & M, k \Vdash \varphi_1 \text{ and } M, k \Vdash \varphi_2 \\
\text{iff } & M, k \Vdash \{\varphi_1, \varphi_2\}
\end{aligned}
$$

- For case 19, $\mathrm{World}(\varphi_1 \vee \varphi_2) = \{\{\varphi_1\}, \{\varphi_2\}\}$. Then

$$
\begin{aligned}
& M, k \Vdash \varphi_1 \vee \varphi_2 \\
\text{iff } & M, k \Vdash \varphi_1 \text{ or } M, k \Vdash \varphi_2 \\
\text{iff } & M, k \Vdash \{\varphi_1\} \text{ or } M, k \Vdash \{\varphi_2\} \\
\text{iff } & \text{there is } \Delta \in \{\{\varphi_1\}, \{\varphi_2\}\} \text{ such that } M, k \Vdash \Delta
\end{aligned}
$$

- Cases 18, 20-24 are similar to the two previous ones;
- For case 26, $\mathrm{World}(\neg e) = \{\{e'\} \mid e' \in \mathbb{E}\backslash\{e\}\}$. Let $e_k \in \mathbb{E}$ be such that $\mathcal{I}(k) = (e_k, \cdots)$. Then,

$$
\begin{aligned}
& M, k \Vdash \neg e \\
\text{iff } & M, k \nVdash e \\
\text{iff } & e_k \neq e \\
\text{iff } & e_k \in \mathbb{E}\backslash\{e\} \\
\text{iff } & \text{there is } e' \in \mathbb{E}\backslash\{e\} \text{ such that } M, k \Vdash \{e'\} \\
\text{iff } & \text{there is } \Delta \in \{\{e'\} \mid e' \in \mathbb{E}\backslash\{e\}\} \text{ such that } M, k \Vdash \Delta
\end{aligned}
$$

- For case 28, $\mathrm{World}(\neg \mathsf{X}\varphi) = \{\{\mathsf{X}\neg\varphi\}\}$. Then,

$$
\begin{aligned}
& M, k \Vdash \neg \mathsf{X}\varphi \\
\text{iff } & M, k \nVdash \mathsf{X}\varphi \\
\text{iff } & M, k+1 \nVdash \varphi \\
\text{iff } & M, k+1 \Vdash \neg\varphi \\
\text{iff } & M, k \Vdash \mathsf{X}\neg\varphi
\end{aligned}
$$

$$\text{iff}\ \ M,k \Vdash \{\mathsf{X}\neg\varphi\}$$

- For case 29, $\mathrm{World}(\mathsf{G}\varphi) = \{\{\varphi, \mathsf{XG}\varphi\}\}$. Then,

$$\begin{aligned}
& M,k \Vdash \mathsf{G}\varphi \\
\text{iff}\ \ & M,k' \Vdash \varphi \text{ for all } k' \geq k \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ and } \ M,k' \Vdash \varphi \text{ for all } k' \geq k+1 \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ and } \ M,k+1 \Vdash \mathsf{G}\varphi \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ and } \ M,k \Vdash \mathsf{XG}\varphi \\
\text{iff}\ \ & M,k \Vdash \{\varphi, \mathsf{XG}\varphi\}
\end{aligned}$$

- For case 31, $\mathrm{World}(\mathsf{F}\varphi) = \{\{\varphi\}, \{\neg\varphi, \mathsf{XG}\varphi\}\}$. Then,

$$\begin{aligned}
& M,k \Vdash \mathsf{F}\varphi \\
\text{iff}\ \ & \text{there is } k' \geq k \text{ such that } M,k' \Vdash \varphi \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ or } \ \big(M,k \nVdash \varphi \ \text{ and } \ \text{there is } k' \geq k+1 \text{ such that } M,k' \Vdash \varphi\big) \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ or } \ \big(M,k \Vdash \neg\varphi \ \text{ and } \ M,k+1 \Vdash \mathsf{F}\varphi\big) \\
\text{iff}\ \ & M,k \Vdash \varphi \ \text{ or } \ \big(M,k \Vdash \neg\varphi \ \text{ and } \ M,k \Vdash \mathsf{XF}\varphi\big) \\
\text{iff}\ \ & M,k \Vdash \{\varphi\} \ \text{ or } \ M,k \Vdash \{\neg\varphi, \mathsf{XF}\varphi\} \\
\text{iff}\ \ & \text{there is } \Delta \in \{\{\varphi\}, \{\neg\varphi, \mathsf{XF}\varphi\}\} \text{ such that } M,k \Vdash \Delta
\end{aligned}$$

- Cases 30 and 32 are similar to the previous one;
- For case 33, $\mathrm{World}(\varphi_1\mathsf{U}\varphi_2) = \{\{\varphi_2\}, \{\varphi_1, \neg\varphi_2, \mathsf{X}(\varphi_1\mathsf{U}\varphi_2)\}\}$. Then,

$$\begin{aligned}
& M,k \Vdash \varphi_1\mathsf{U}\varphi_2 \\
\text{iff}\ \ & \text{there is } k' \geq k \text{ such that } M,k' \Vdash \varphi_2 \ \text{ and } \ M,k'' \Vdash \varphi_1 \text{ for all } k \leq k'' < k' \\
\text{iff}\ \ & M,k \Vdash \varphi_2 \ \text{ or } \ \big(M,k \Vdash \varphi_1 \ \text{ and } \ M,k \nVdash \varphi_2 \\
& \quad \text{and } \ \text{there is } k' \geq k+1 \text{ such that } M,k' \Vdash \varphi_2 \\
& \quad \text{and } \ M,k'' \Vdash \varphi_1 \text{ for all } k+1 \leq k'' < k'\big) \\
\text{iff}\ \ & M,k \Vdash \varphi_2 \ \text{ or } \ \big(M,k \Vdash \varphi_1 \ \text{ and } \ M,k \Vdash \neg\varphi_2 \ \text{ and } \ M,k+1 \Vdash \varphi_1\mathsf{U}\varphi_2\big) \\
\text{iff}\ \ & M,k \Vdash \varphi_2 \ \text{ or } \ \big(M,k \Vdash \varphi_1 \ \text{ and } \ M,k \Vdash \neg\varphi_2 \ \text{ and } \ M,k \Vdash \mathsf{X}(\varphi_1\mathsf{U}\varphi_2)\big) \\
\text{iff}\ \ & M,k \Vdash \{\varphi_2\} \ \text{ or } \ M,k \Vdash \{\varphi_1, \neg\varphi_2, \mathsf{X}(\varphi_1\mathsf{U}\varphi_2)\} \\
\text{iff}\ \ & \text{there is } \Delta \in \{\{\varphi_2\}, \{\varphi_1, \neg\varphi_2, \mathsf{X}(\varphi_1\mathsf{U}\varphi_2)\}\} \text{ such that } M,k \Vdash \Delta
\end{aligned}$$

- Case 34 is similar to the previous one.

2. We show that if Ω is a finite set of formulas,

$$M, k \Vdash \Omega \text{ iff there is } \Delta \in \mathrm{World}(\Omega) \text{ such that } M, k \Vdash \Delta \tag{2}$$

Let $\Omega = \{\varphi_1, \ldots, \varphi_n\}$. Recall that each set $\Delta \in \mathrm{World}(\Omega)$ is of the form

$$\Delta = \{\varphi_1^{\checkmark}\} \cup \Delta_1 \cup \cdots \cup \{\varphi_n^{\checkmark}\} \cup \Delta_n \text{ where each } \Delta_i \in \mathrm{World}(\varphi_i)$$

Furthermore, every set of that form is in $\mathrm{World}(\Omega)$. Thus,

$M, k \Vdash \Omega$

iff $M, k \Vdash \varphi_i$ for all $i \in \{1, \ldots, n\}$

iff for each $i \in \{1, \ldots, n\}$

– $M, k \Vdash \{\varphi_i^{\checkmark}\}$

– there is $\Delta_i \in \mathrm{World}(\varphi_i)$ such that $M, k \Vdash \Delta_i$ (by(1))

iff there are $\Delta_1, \ldots, \Delta_n$ with $\Delta_i \in \mathrm{World}(\varphi_i)$ such that

$M, k \Vdash \{\varphi_1^{\checkmark}\} \cup \Delta_1 \cup \cdots \cup \{\varphi_n^{\checkmark}\} \cup \Delta_n$

iff there is $\Delta \in \mathrm{World}(\Omega)$ such that $M, k \Vdash \Delta$

3. We show that if Ψ is a finite set of finite sets of formulas, then

$$\Big(\text{there is } \Omega \in \Psi \text{ such that } M, k \Vdash \Omega\Big) \text{ iff } \Big(\text{there is } \Delta \in \mathrm{World}(\Psi) \text{ such that } M, k \Vdash \Delta\Big) \tag{3}$$

Recall that if $\Psi = \{\Omega_1, \ldots, \Omega_n\}$ then $\mathrm{World}(\Psi) = \mathrm{World}(\Omega_1) \cup \cdots \cup \mathrm{World}(\Omega_n)$. Thus,

there is $\Omega_i \in \Psi$ such that $M, k \Vdash \Omega_i$

iff there is $i \in \{1, \ldots, n\}$ such that there is $\Delta \in \mathrm{World}(\Omega_i)$ such that $M, k \Vdash \Delta$ (by (2))

iff there is $\Delta \in \mathrm{World}(\Omega_1) \cup \cdots \cup \mathrm{World}(\Omega_n)$ such that $M, k \Vdash \Delta$

iff there is $\Delta \in \mathrm{World}(\Psi)$ such that $M, k \Vdash \Delta$

One can now easily prove that

$$M, k \Vdash \Omega \text{ iff there is } \Delta \in \mathrm{World}^n(\Omega) \text{ such that } M, k \Vdash \Delta, \text{ for all } n$$

by induction using 3 in the step. In particular,

$$M, k \Vdash \Omega \text{ iff there is } \Delta \in \Omega^{\ddagger} \text{ such that } M, k \Vdash \Delta \tag{4}$$

4. Notice that if M is an interpretation structure, then $M, k \Vdash \varepsilon_{\Sigma}$.

5. We show that $\mathrm{OperatorClosure}$ preserves satisfiability. For any set of formulas Δ,

$$M, k \Vdash \Delta \text{ iff } M, k \Vdash \mathrm{OperatorClosure}(\Delta) \tag{5}$$

$\longrightarrow$ By contradiction, assume that $M, k \Vdash \Delta$, but there is $\varphi \in \mathrm{OperatorClosure}(\Delta)$ such that $M, k \nVdash \varphi$. Then φ encodes some property (transitivity of $<$, reflexivity of $\cong$, etc.) which M does not have, and so M is not an interpretation structure. Contradiction.

$\longleftarrow$ Because, by definition, $\Delta \subset \mathrm{OperatorClosure}(\Delta)$.

Finally, we can prove the main result.

$M, k \Vdash \Omega$

iff $M, k \Vdash \Omega \cup \{\varepsilon_{\Sigma}\}$

iff there is $\Delta \in (\Omega \cup \{\varepsilon_{\Sigma}\})^{\ddagger} = \{\Delta_1, \ldots, \Delta_n\}$ such that $M, k \Vdash \Delta$ (by (4))

iff there is $\Delta \in \{\mathrm{OperatorClosure}(\Delta_1), \ldots, \mathrm{OperatorClosure}(\Delta_n)\}$ such that $M, k \Vdash \Delta$ (by (5))

iff there is $\Delta \in \Omega^{*}$ such that $M, k \Vdash \Delta$

□

Now we present a mechanism to verify syntactically if a set of formulas is satisfiable.

Definition 3.8. A set $\Delta \subset L_{\Sigma}$ of formulas is said to be *locally incompatible* if at least one of the following holds:

1. there are a and ϕ such that $a : \phi, a : -\phi \in \Delta$;
2. there is t such that $t < t \in \Delta$
3. there is a formula φ such that $\varphi, \neg\varphi \in \Delta$;
4. $\bot \in \Delta$;
5. $|\Delta \cap \mathbb{E}| \neq 1$;

If Δ is not locally incompatible, we say that it is locally compatible.

The next result establishes that local compatibility is a necessary condition for satisfiability.

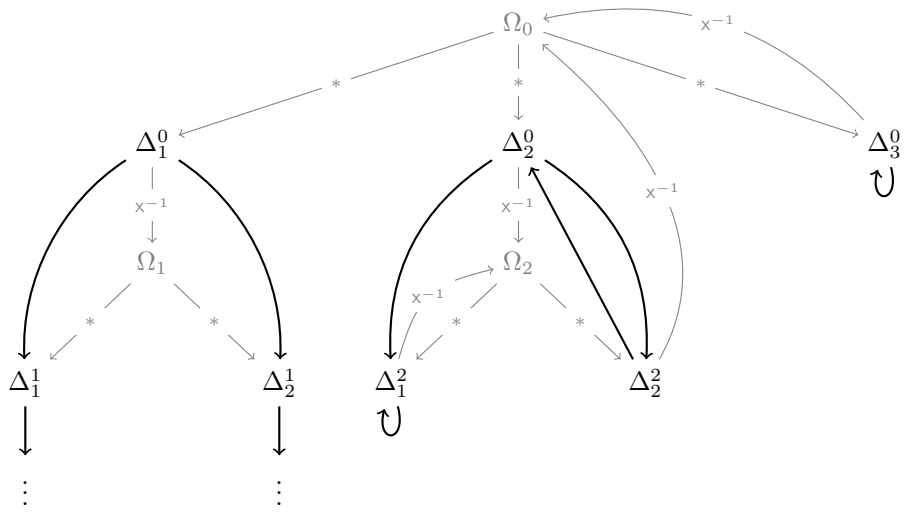


Figure 3.1: Abstract representation of the graph structure used to verify the satisfaction of Ω_0. In black, we represent the nodes and transitions of the actual graph structure. In gray, we represent the middle steps in generating such graph.

Proposition 3.9. *Let $\Delta \subset L_\Sigma$. Then, for any $k \in \mathbb{N}$,*

$$\Delta \textit{ is } k\textit{-satisfiable} \quad \text{implies} \quad \Delta \textit{ is locally compatible}$$

Proof. By contradiction assume that Δ is locally incompatible and M k-satisfies Δ. Then one of the conditions of definition 3.8 holds. But then M is not a valid interpretation structure. Contradiction. □

3.2 Global Satisfiability

In this section we deal with deciding the satisfiability and validity of formulas with temporal operators. In summary, our method to check if some set of formulas Ω is satisfiable consists of building set $C(\Omega)$ of indecomposable sets of formulas, each representing a possible instant where all the formulas in Ω hold. Then, using the X formulas as a temporal guide, we form a graph structure which encodes the possible interpretation structures that satisfy some formula. An abstract representation of this graph structure is shown in figure 3.1.

Recall that the $<$ relation between time symbols is connected with respect to $\cong$. That means that for any two time symbols t_1 and t_2 we have either $t_1 < t_2$, $t_1 \cong t_2$ or $t_2 < t_1$. We must ensure that this property is captured by the graph construction. For that we add a formula which encodes this property into whichever set whose satisfiability we are checking.

Definition 3.10. We denote by τ_Σ the following formula

$$\tau_\Sigma = \bigvee_i \chi_i$$

Where each χ_i is a "chain", that is, a formula of the type

$$t_{i_1} \leqq t_{i_2} \wedge t_{i_2} \leqq t_{i_3} \wedge \cdots \wedge t_{i_{n-1}} \leqq t_{i_n}$$

where each $\leqq$ is either $<$ or $\cong$ and such that each time symbol in $\mathbb{T}$ is featured exactly once. Note that τ_Σ does not contain all possible chains, but only those which represent unique interpretations of the symbols.

We may represent time chains using a more concise notation, by omitting the repeated symbols and the $\wedge$ connective: $t_1 < t_2 \wedge t_2 \cong t_3$ can be represented as $t_1 < t_2 \cong t_3$. On this last note, we clarify the notion of unique interpretation with an example: $t_1 < t_2 \cong t_3$ and $t_1 < t_3 \cong t_2$ are different chains, but they effectively represent the same time interpretation. Thus, only one of these would be featured in τ_Σ. How we generate all the chains corresponding to unique time interpretations will be discussed in section 5.1.

Definition 3.11. Given a set Ω of formulas, we denote by $\mathsf{X}^{-1}(\Omega)$ the set

$$\mathsf{X}^{-1}(\Omega) = \{\varphi \mid \mathsf{X}\varphi \in \Omega\}$$

Definition 3.12. The completion $C(\Omega)$ of a non-empty finite set $\Omega \subset L_\Sigma$ is inductively defined as follows:

- $(\Omega \cup \{\tau_\Sigma\})^* \subset C(\Omega)$;
- if $\Delta \in C(\Omega)$ then $\big(\mathsf{X}^{-1}(\Delta)\big)^* \subset C(\Omega)$.

Given two sets of formulas $\Delta_1, \Delta_2 \in C(\Omega)$, we say that Δ_2 is accessible from Δ_1, and write $\Delta_1 \longrightarrow \Delta_2$ whenever $\Delta_2 \in \big(\mathsf{X}^{-1}(\Delta_1)\big)^*$.

Proposition 3.13. *$C(\Omega)$ is a finite set.*

Proof. Intuitively the proof is based on the fact that the formulas introduced during the closure operation are (mostly) subformulas of Ω. Some formulas may be prepended with X or $\neg$, but it always results in a finite amount of new formulas.
We first establish a very loose (but finite) bound on the set of formulas which can appear in some $\Delta \in C(\Omega)$.
Let W be the following set of formulas

$$\begin{aligned} W = &\{a : \phi \mid a \in \mathbb{A}, \phi \in K_\Sigma\} \\ &\cup \{a : \boxdot\phi \mid a \in \mathbb{A}, \phi \in K_\Sigma\} \\ &\cup \{a \trianglelefteq_p b \mid a, b \in \mathbb{A}, p \in \mathbb{P}\} \\ &\cup \{t_1 < t_2 \mid t_1, t_2 \in \mathbb{T}\} \\ &\cup \{t_1 \cong t_2 \mid t_1, t_2 \in \mathbb{T}\} \\ &\cup \{a \trianglelefteq_\phi b \implies \neg(b : \phi) \mid a, b \in \mathbb{A}, \phi \in K_\Sigma\} \\ &\cup \{a : \phi \implies \neg a : \boxdot\phi \mid a \in \mathbb{A}, \phi \in K_\Sigma\} \end{aligned}$$

$\cup \{\bot\}$

$\cup\ \mathbb{E}$

W consists of formulas which might appear after the closure operation even if they are not subformulas of Ω. Since, $\mathbb{P}$, $\mathbb{E}$, $\mathbb{T}$, $\mathbb{A}$ and K_Σ are finite, W also is.
Let S be the set of all subformulas present in all formulas of $\Omega \cup W$. Ω and W are finite. Hence, S is also finite.
Finally, we can establish our bound. Let,

$$L_\Omega = \{\varphi, \neg\varphi, \mathsf{X}\varphi, \mathsf{X}\neg\varphi \mid \varphi \in S\}$$

Clearly L_Ω is finite. Then,

$$\Delta \in C(\Omega)\ \text{ implies }\ \Delta \subset L_\Omega$$

Indeed, looking at definition 3.1, whenever we apply World to a formula φ, we get either a formula of W, a subformula of φ (including itself) or a subformula of φ prepended with X, $\neg$ or X$\neg$. Now, X formulas are treated as monoliths and $\neg$ formulas also do not introduce new formulas besides the ones defined in W. Hence, the implication holds. Thus, $C(\Omega) \subset \mathcal{P}(L_\Omega)$. Since L_Ω is finite, so is $C(\Omega)$. □

The $\longrightarrow$ relation induces a graph structure over the elements of $C(\Omega)$. In order to assess whether Ω is satisfiable, we search within this graph structure for a path starting at $\Delta \in \Omega^*$ which at some point loops back to some previous state.
We now define a compatible path.

Definition 3.14. A compatible path is a finite sequence of sets of formulas,

$$[\Delta_0, \Delta_1, \cdots, \Delta_n]$$

such that each Δ_i is in $C(\Omega)$ and all the following conditions are met

1. $\Delta_0 \longrightarrow \Delta_1 \longrightarrow \cdots \longrightarrow \Delta_n$;
2. All Δ_i are locally compatible;
3. There is $x : 0 \leq x \leq n$ such that $\Delta_n \longrightarrow \Delta_x$.
4. The final conditions are met. That is,
   (a) if $\varphi_1 \mathsf{U} \varphi_2 \in \Delta_i$ for some i, then there must be a j such that $\varphi_2 \in \Delta_j$;
   (b) if $\mathsf{F}\varphi' \in \Delta_i$ for some i, then there must be a j such that $\varphi' \in \Delta_j$;
   (c) if $\neg\mathsf{G}\varphi' \in \Delta_i$ for some i, then there must be a j such that $\neg\varphi' \in \Delta_j$;

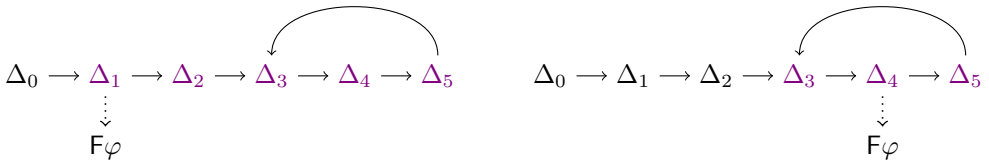


Figure 3.2: The two inequalities of definition 3.14 visualized. If $\mathsf{F}\varphi$ is found in Δ_1 or Δ_4 respectively, then φ must be found in one of the sets colored purple in order for the path to be compatible.

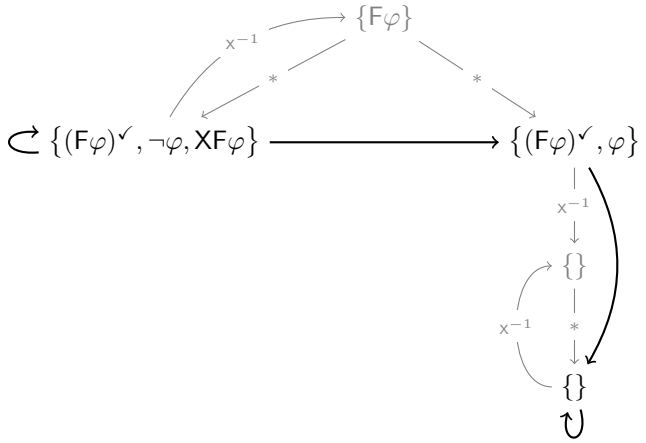


Figure 3.3: Graph structure for deciding if $\{\mathsf{F}\varphi\}$ is satisfiable.

In (a), (b) and (c) j must be such that

$$\begin{cases} i \leq j \leq n, & \text{if } i < x \\ x \leq j \leq n, & \text{if } x \leq i \leq n \end{cases}$$

The last two inequalities might seem a bit counterintuitive, but it is just a matter of where we find the formula which requires a final condition, and where we are allowed to search for the formula which fulfills said condition. Suppose that we have found a path which fulfills points 1-3. If we find $\mathsf{F}\varphi$ in some Δ_i before the cycle (i.e., $i < x$), then we must find φ in any set after Δ_i. On the other hand, if Δ_i is inside the cycle (i.e., $i : x \leq i \leq n$), then we must find φ in any set within the whole cycle. An example is shown in figure 3.2. Note that although we typically represent a compatible path by $[\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$, there is no minimum length for a compatible path, nor does the "looping set" Δ_x need to be different from Δ_0 or Δ_n. The sequence $[\Delta_0]$ might very well be a compatible path, so long as $\Delta_0 \longrightarrow \Delta_0$ and all the other conditions hold.

We illustrate the need for the final conditions in the following example.

Example 3.15. Suppose we are trying to check whether $\{\mathsf{F}\varphi\}$ is satisfiable. We present the graph structure in figure 3.3. Looking at the graph structure, it is easy to see a case where the final condition is necessary. If not, then the sequence $[\{(\mathsf{F}\varphi)^{\checkmark}, \neg\varphi, \mathsf{XF}\varphi\}]$ would be considered a compatible path, since $\{(\mathsf{F}\varphi)^{\checkmark}, \neg\varphi, \mathsf{XF}\varphi\}$ is locally compatible and $\{(\mathsf{F}\varphi)^{\checkmark}, \neg\varphi, \mathsf{XF}\varphi\} \longrightarrow \{(\mathsf{F}\varphi)^{\checkmark}, \neg\varphi, \mathsf{XF}\varphi\}$. This would mean that an interpretation structure which satisfies $\neg\varphi$ at every time point would also satisfy $\mathsf{F}\varphi$, which does not make sense.

If the reader is familiarized with the automata-based approach to model checking, they might realize that the graph of figure 3.3 is essentially a Büchi automaton whose accepted language is equal to the set of interpretation structures which satisfy $\mathsf{F}\varphi$. Indeed, our approach is not very different. If the graph were a Büchi automaton, then its set of final states would be $\{\{(\mathsf{F}\varphi)^{\checkmark}, \varphi\}, \{\}\}$ as for each set either $\mathsf{F}\varphi$ is not in it or φ is in it. Thus, any word accepted by the automaton must include one of those states infinitely often (in this case, the only option is that $\{\}$ is featured infinitely often). That is, if we have a formula which requires a witness (in this case, $\mathsf{F}\varphi$ requires φ to hold at some point) the final state condition of a Büchi automaton prevents a path in which the witness never holds to be accepted. That is, it serves the same purpose of our final conditions. A detailed explanation of the automata-based approach to mode checking can be found in [4].
We will need the following result in a proof later.

Proposition 3.16. *Let $\Delta_1 \in C(\Omega)$. Then,*

$$M, k \Vdash \Delta_1 \text{ implies } \textit{there is } \Delta_2 \in C(\Omega) \textit{ such that } \Delta_1 \longrightarrow \Delta_2 \text{ and } M, k+1 \Vdash \Delta_2$$

Proof. Assume, $M, k \Vdash \Delta_1$, then in particular $M, k \Vdash \{\mathsf{X}\varphi \mid \mathsf{X}\varphi \in \Delta_1\}$. Then, $M, k{+}1 \Vdash \{\varphi \mid \mathsf{X}\varphi \in \Delta_1\} = \mathsf{X}^{-1}(\Delta_1)$. By proposition 3.7, there is $\Delta_2 \in (\mathsf{X}^{-1}(\Delta_1))^*$ such that $M, k+1 \Vdash \Delta_2$. By definition of $\longrightarrow$, $\Delta_1 \longrightarrow \Delta_2$. □

We are now ready to prove the main result of this chapter.

Proposition 3.17. *Let Ω be a set of formulas. Then,*

$$\Omega \textit{ is satisfiable } \text{iff } \textit{ there is a compatible path starting at some } \Delta \in (\Omega \cup \{\tau_\Sigma\})^*$$

Proof.

$\longrightarrow$ In this proof, for convenience sake, we introduce the generic temporal connective T. A formula φ is of the type $\mathsf{T}\psi$ if φ is either $\mathsf{F}\psi$, $\varphi'\mathsf{U}\psi$ or $\neg\mathsf{G}\neg\psi$. Intuitively, if $\mathsf{T}\psi$ holds at instant k, then there is an instant $k' \geq k$ where ψ holds. We denote by T_Ω the set of all T formulas present in any set of $C(\Omega)$. Notice that because $C(\Omega)$ is a finite set of finite sets, T_Ω is also finite. Using the generic connective, one can define the final conditions of definition 3.14 in just one item:

A finite sequence $[\Delta_0, \ldots \Delta_x, \ldots \Delta_n]$ with $\Delta_n \longrightarrow \Delta_x$ satisfies the final conditions if whenever $\mathsf{T}\varphi \in \Delta_i$, there is j such that $\varphi \in \Delta_j$ and such that

$$\begin{cases} i \leq j \leq n, & \text{if } i < x \\ x \leq j \leq n, & \text{if } x \leq i \leq n \end{cases}$$

We now prove the result. Assume that there is an interpretation structure M such that $M, 0 \Vdash \Omega$. Since any valid interpretation structure satisfies τ_Σ, $M, 0 \Vdash \Omega \cup \{\tau_\Sigma\}$. Using M, we define an infinite sequence of sets $s = \Delta_0, \Delta_1, \ldots$ inductively as follows:

- $\Delta_0 \in C(\Omega)$ is such that $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$ and $M, 0 \Vdash \Delta_0$. Proposition 3.7 guarantees its existence.
- $\Delta_{k+1} \in C(\Omega)$ is such that $\Delta_k \longrightarrow \Delta_{k+1}$ and $M, k+1 \Vdash \Delta_{k+1}$. Proposition 3.16 guarantees its existence.

Thus, s is an infinite sequence of sets of formulas such that $M, k \Vdash \Delta_k$ and $\Delta_k \longrightarrow \Delta_{k+1}$ for all k. If s is periodic, i.e., $s = \Delta_0 \cdots \Delta_{x-1}(\Delta_x \cdots \Delta_n)^\omega$, then the sequence $[\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$ is a compatible path with $\Delta_n \longrightarrow \Delta_x$. However, s might not be periodic.

Notice that because each $\Delta_k \in s$ is in $C(\Omega)$ and $C(\Omega)$ is finite, there must be some set $\widetilde{\Delta}$ which appears in s infinitely often. Thus, s can be broken up into an infinite family of finite blocks as follows:

$$s = \underbrace{\Delta_0, \cdots, \Delta_{k_0}}_{\mathbf{w_0}}, \underbrace{\widetilde{\Delta}, \cdots, \Delta_{k_1}}_{\mathbf{w_1}}, \underbrace{\widetilde{\Delta}, \cdots, \Delta_{k_2}}_{\mathbf{w_2}}, \underbrace{\widetilde{\Delta}, \cdots, \Delta_{k_3}}_{\mathbf{w_3}}, \cdots$$

We say that a block $\mathbf{w}$ contains a formula φ if there is $\Delta \in \mathbf{w}$ such that $\varphi \in \Delta$. We now build a periodic sequence s' from these blocks using the following procedure

1. We begin by setting $s' \leftarrow \mathbf{w_0}(\mathbf{w_1})^\omega$.
2. Let D start as the empty set. This set represents the formulas which have been dealt with.
3. If there is $\Delta_i \in s'$ such that $\mathsf{T}\varphi \in \Delta_i$, but $\varphi \notin \Delta_j$ for all Δ_j in s' with $j \geq i$, then
   (a) Let n be such that $s' = \mathbf{w_0}(\mathbf{w_1} \cdots \mathbf{w_n})^\omega$.
   (b) Let r the smallest number such that $\mathbf{w_{n+r}}$ contains φ. We can always find such an r. Notice that $M, i \Vdash \Delta_i$ and so, $M, i \Vdash \mathsf{T}\varphi$. Thus, there is $j \geq i$ such that $M, j \Vdash \varphi$. By the definition of World for T formulas, there is $j \geq i$ such that $\varphi \in \Delta_j$.
   (c) Add all the blocks from $n+1$ to $n+r$ to s'. That is, $s' \leftarrow \mathbf{w_0}(\mathbf{w_1} \cdots \mathbf{w_n}\mathbf{w_{n+1}} \cdots \mathbf{w_{n+r}})^\omega$
   (d) Add $\mathsf{T}\varphi$ to D.
   (e) Go to 3.
4. Else, we are done.

We must prove that this procedure always halts. Whenever we enter 3, by the end of step (c) s' will have φ in its periodic part. That is, φ will appear in s' infinitely often. Thus, whenever we enter 3, $\mathsf{T}\varphi$ is not in D, because if it were, then φ already appears in s' infinitely often, and so the guard of 3 cannot hold. Furthermore, notice that every time we enter 3, we add an element to D. However, $D \subset \mathsf{T}_\Omega$, and so its size is bounded by the size of T_Ω, which is finite. Thus, we enter 3 a finite number of times and so the procedure halts.

Thus, we have built an infinite sequence $s' = \Delta_0 \cdots \Delta_{x-1}(\Delta_x \cdots \Delta_n)^\omega$. From it, we define the finite sequence

$$p = [\Delta_0, \cdots, \Delta_{x-1}, \Delta_x, \cdots \Delta_n]$$

We must prove that p is a compatible path. Indeed,

1. $\Delta_0 \longrightarrow \Delta_1 \longrightarrow \cdots \longrightarrow \Delta_n$;
2. Notice that p consists of the first n elements of s, and so $M, k \Vdash \Delta_k$ for every $k : 0 \leq k \leq n$. Thus, each element of p is locally compatible by proposition 3.9.
3. $\Delta_n \longrightarrow \Delta_x$.
4. By construction of s', it is clear that p fulfills the final conditions.

Thus, we have found a compatible path.

$\longleftarrow$ Assume there is a compatible path $p = [\Delta_0, \ldots, \Delta_x, \ldots \Delta_n]$ such that $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$ and $\Delta_n \longrightarrow \Delta_x$. From such a path we extract an interpretation structure $M = \left(<^M, \cong^M, \left\{\trianglelefteq_p^M\right\}_{p\in\mathbb{P}}, \mathcal{I}\right)$ which satisfies Ω. M is defined as follows:

- The time and trust orders are extracted from Δ_n. Note that we cannot do this using Δ_0 because time and trust order formulas are propagated only forwards. Of course any set within the cycle would be admissible. That is, $<^M$, $\cong^M$ and $\{\trianglelefteq_p^M\}_{p\in\mathbb{P}}$ are the smallest relations such that
  - if $t_1 < t_2 \in \Delta_n$ then $t_1 <^M t_2$;
  - if $t_1 \cong t_2 \in \Delta_n$ then $t_1 \cong^M t_2$;
  - if $a_1 \trianglelefteq_p a_2 \in \Delta_n$ then $a_1 \trianglelefteq_p^M a_2$.
- We extract the local interpretation sequence $\mathcal{I}$ from the compatible path. Let $I_0, \ldots, I_n$ be local interpretation structures such that, for all $i \in \{0, \ldots, n\}$,
  - $I_i = (e_i, C_i)$;
  - C_i is the smallest set such that if $a : \phi \in \Delta_i$ then $a : \phi \in C_i$;
  - if $e \in \Delta_i$ then $e_i = e$. Recall that there is exactly one event formula present in Δ_i, since Δ_i is locally compatible.

  And now define

$$\mathcal{I} = I_0\ I_1\ \ldots\ I_{x-1}\,(I_x\ I_{x+1}\ \ldots\ I_n)^\omega$$

  That is, $\mathcal{I} = \lambda k.I_{f(k)}$, where $f = \lambda k.\begin{cases} k & \text{if } k \leq x-1 \\ x + \mathrm{mod}(k-x, n-x+1) & \text{otherwise} \end{cases}$.

Now we must prove that M satisfies Ω. First, some properties of function f which we might need:

P1: $0 \leq f(k) \leq n$ for all k;

P2: if $0 \leq k \leq n$ then $f(k) = k$;

P3: if $k \geq x$ then $x \leq f(k) \leq n$;

P4: for all $k \geq x$, for $x \leq j \leq n$ there is $j' \geq k$ such that $f(j') = j$;

P5: f is such that $\Delta_{f(k)} \longrightarrow \Delta_{f(k+1)}$ for all k. In particular, $f(n+1) = x$.

We start by proving that for all $k \in \mathbb{N}$,

$$\varphi \in \Delta_{f(k)} \text{ implies } M, k \Vdash \varphi \tag{6}$$

The proof is by induction on the structure of φ.

For the basis of the induction, either φ is in AL_Σ or in $\mathbb{E}$. In both cases (6) holds either by definition of M or Ω^*.

For the induction step we omit the details for the propositional operators (including negation) and consider only the temporal operators. Note that property P1 guarantees that $\Delta_{f(k)}$ is part of p for all k.

- φ is $\mathsf{X}\varphi'$, where we assume (6) holds for φ'. By P5, $\Delta_{f(k+1)} \in (\mathsf{X}^{-1}(\Delta_{f(k)}))^*$ for all k. Then

  $\mathsf{X}\varphi' \in \Delta_{f(k)}$
  implies $\varphi' \in \mathsf{X}^{-1}(\Delta_{f(k)})$
  implies $\varphi' \in \Lambda$, for all $\Lambda \in (\mathsf{X}^{-1}(\Delta_{f(k)}))^*$ (by definition of World for sets)
  implies $\varphi' \in \Delta_{f(k+1)}$
  implies $M, k+1 \Vdash \varphi'$ (by induction hypothesis)
  implies $M, k \Vdash \mathsf{X}\varphi'$

- φ is $\mathsf{G}\varphi'$, where we assume (6) holds for φ'. Assume that $\mathsf{G}\varphi' \in \Delta_{f(k)}$. We first prove that $\mathsf{G}\varphi' \in \Delta_{f(i)}$ for every $i \geq k$ by induction.

  The basis of the induction is immediate, as by hypothesis, $\mathsf{G}\varphi' \in \Delta_{f(k)}$. For the step, assume that $\mathsf{G}\varphi' \in \Delta_{f(i)}$. Since $\Delta_{f(i)} \in C(\Omega)$, by definition of World, $\mathsf{XG}\varphi' \in \Delta_{f(i)}$. By P5 and the definition of $\longrightarrow$, $\mathsf{G}\varphi' \in \Delta_{f(i+1)}$.

  Now, notice that by definition of World, for any $\Delta \in C(\Omega)$, if $\mathsf{G}\varphi' \in \Delta$ then $\varphi' \in \Delta$.

  Finally,

  $\mathsf{G}\varphi' \in \Delta_{f(k)}$
  implies $\mathsf{G}\varphi' \in \Delta_{f(i)}$ for all $i \geq k$
  implies $\varphi' \in \Delta_{f(i)}$ for all $i \geq k$
  implies $M, i \Vdash \varphi'$ for all $i \geq k$ (by induction hypothesis)
  implies $M, k \Vdash \mathsf{G}\varphi'$

- φ is $\mathsf{F}\varphi'$, where we assume (6) holds for φ'. There are two cases,

– Case $0 \leq k < x$

$$
\begin{aligned}
& \mathsf{F}\varphi' \in \Delta_{f(k)} \\
\text{implies}\ & \mathsf{F}\varphi' \in \Delta_k && \text{(by P2)} \\
\text{implies}\ & \text{there is } i : k \leq i \leq n \text{ such that } \varphi' \in \Delta_i && \text{(because } p \text{ is compatible)} \\
\text{implies}\ & \text{there is } i \geq k \text{ such that } \varphi' \in \Delta_{f(i)} && \text{(by P2)} \\
\text{implies}\ & \text{there is } i \geq k \text{ such that } M, i \Vdash \varphi' && \text{(by induction hypothesis)} \\
\text{implies}\ & M, k \Vdash \mathsf{F}\varphi'
\end{aligned}
$$

– Case $k \geq x$

$$
\begin{aligned}
& \mathsf{F}\varphi' \in \Delta_{f(k)} && \text{(note that by P3, } x \leq f(k) \leq n\text{)} \\
\text{implies}\ & \text{there is } i : x \leq i \leq n \text{ such that } \varphi' \in \Delta_i && \text{(because } p \text{ is compatible)} \\
\text{implies}\ & \text{there is } i' \geq k \text{ such that } \varphi' \in \Delta_{f(i')} && \text{(by P4)} \\
\text{implies}\ & \text{there is } i' \geq k \text{ such that } M, i' \Vdash \varphi' && \text{(by induction hypothesis)} \\
\text{implies}\ & M, k \Vdash \mathsf{F}\varphi'
\end{aligned}
$$

- φ is $\varphi_1 \mathsf{U} \varphi_2$, where we assume (6) holds for both φ_1 and φ_2. The proof consists of two steps.

  1. We prove that there is $i \geq k$ such that $M, i \Vdash \varphi_2$. The proof is very similar to the previous case where φ was $\mathsf{F}\varphi'$.
     There are two cases,

     – Case $0 \leq k < x$

$$
\begin{aligned}
& \varphi_1 \mathsf{U} \varphi_2 \in \Delta_{f(k)} \\
\text{implies}\ & \text{there is } i : k \leq i \leq n \text{ such that } \varphi_2 \in \Delta_i && \text{(because } p \text{ is compatible)} \\
\text{implies}\ & \text{there is } i \geq k \text{ such that } \varphi_2 \in \Delta_{f(i)} && \text{(by P2)} \\
\text{implies}\ & \text{there is } i \geq k \text{ such that } M, i \Vdash \varphi_2 && \text{(by induction hypothesis)}
\end{aligned}
$$

     – Case $k \geq x$

$$
\begin{aligned}
& \varphi_1 \mathsf{U} \varphi_2 \in \Delta_{f(k)} && \text{(note that by P3, } x \leq f(k) \leq n\text{)} \\
\text{implies}\ & \text{there is } i : x \leq i \leq n \text{ such that } \varphi_2 \in \Delta_i && \text{(because } p \text{ is compatible)} \\
\text{implies}\ & \text{there is } i' \geq k \text{ such that } \varphi_2 \in \Delta_{f(i')} && \text{(by P4)} \\
\text{implies}\ & \text{there is } i' \geq k \text{ such that } M, i' \Vdash \varphi_2 && \text{(by induction hypothesis)}
\end{aligned}
$$

2. Let $i \geq k$ be the smallest number such that $M, i \Vdash \varphi_2$. We prove that $M, j \Vdash \varphi_1$ for all $j : k \leq j < i$.

   Assume that $i > k$ (if $i = k$, the statement is vacuously true). Then $M, k \nVdash \varphi_2$. By definition of World, if $\varphi_1 \mathsf{U} \varphi_2 \in \Delta_{f(k)}$, then either $\varphi_2 \in \Delta_{f(k)}$ or $\varphi_1, \neg\varphi_2, \mathsf{X}(\varphi_1 \mathsf{U} \varphi_2) \in \Delta_{f(k)}$. Assume $\varphi_2 \in \Delta_{f(k)}$. Then, by induction hypothesis, $M, k \Vdash \varphi_2$. Contradiction. Thus, $\varphi_1, \neg\varphi_2, \mathsf{X}(\varphi_1 \mathsf{U} \varphi_2) \in \Delta_{f(k)}$ and in particular $\varphi_1 \in \Delta_{f(k)}$. Hence, by induction hypothesis, $M, k \Vdash \varphi_1$. Additionally, by P5 and the definition of $\longrightarrow$, we can conclude that $\varphi_1 \mathsf{U} \varphi_2 \in \Delta_{f(k+1)}$. Thus, we can repeat the argument to conclude that $M, k+1 \Vdash \varphi_1$. Making this argument for $j = k, k+1, \ldots, i-2, i-1$ we conclude that $M, j \Vdash \varphi_1$ for all $j : k \leq j < i$.

By 1 and 2, we conclude that $M, k \Vdash \varphi_1 \mathsf{U} \varphi_2$.

Thus, we have proven (6). Finally, we are ready to prove that $M \Vdash \Omega$. By 6, $M, 0 \Vdash \varphi$ for all $\varphi \in \Delta_0 = \Delta_{f(0)}$. Hence, $M, 0 \Vdash \Delta_0$. Since $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$, by proposition 3.7, $M, 0 \Vdash \Omega \cup \{\tau_\Sigma\}$ and so, in particular, $M, 0 \Vdash \Omega$. Thus, $M \Vdash \Omega$

□

The previous result is still not sufficient to prove that the satisfiability problem for sets of formulas is decidable, as there is still an infinite number of finite paths over the graph structure of $C(\Omega)$. The next result will establish a bound on the maximum number of times a set Δ must appear in a compatible path.

Proposition 3.18. *Let Ω be a set of formulas. Let $b = \#(\mathsf{T}_\Omega) + 2$. Then, Ω is satisfiable iff there is a compatible path $p = [\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$ with $\Delta_n \longrightarrow \Delta_x$ such that $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$ and each Δ_i appears in p at most b times.*

Proof.

$\longleftarrow$ Immediate by proposition 3.17.

$\longrightarrow$ Assume that Ω is satisfiable. Then, by proposition 3.17, there is a compatible path

$$p = [\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$$

such that $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$. However, p might contain sets which appear more than b times. The following procedure produces a compatible path p' which satisfies this restriction.

1. Set $p' \leftarrow p$.
2. Let D start as the empty set. This set represents the formulas which have been dealt with.
3. If there is a set $\widetilde{\Delta}$ which appears in p' more than b times, then
   (a) Set $p'' \leftarrow [\,]$.

(b) Divide p' into blocks as described in the proof of proposition 3.17. That is, such that

$$p' = [\ \underbrace{\Delta_0, \cdots, \Delta_{k_0}}_{\mathbf{w_0}},\ \underbrace{\widetilde{\Delta}, \cdots, \Delta_{k_1}}_{\mathbf{w_1}},\ \cdots,\ \underbrace{\widetilde{\Delta}, \cdots, \Delta_n}_{\mathbf{w_m}}\ ]$$

(c) For each block $\mathbf{w_i} = \mathbf{w_0}, \ldots, \mathbf{w_m}$

i. If $\mathbf{w_i}$ contains a formula φ such that $\mathsf{T}\varphi \in \mathsf{T}_\Omega \backslash D$ or $\mathbf{w_i}$ contains either the first appearance of Δ_x in p' or the last set of p', then

A. Append $\mathbf{w_i}$ to p''.

B. For every formula $\mathsf{T}\varphi \in \mathsf{T}_\Omega \backslash D$ such that $\mathbf{w_i}$ contains φ, add $\mathsf{T}\varphi$ to D.

ii. Else, no block is appended to p''.

(d) Set $p' \leftarrow p''$.

(e) Go to 3.

4. Else, we are done.

Intuitively, the procedure skips every block except the ones which contain witnesses. The bound is then determined by the number of witnesses needed.

We must prove that this procedure always halts and that at the end of this procedure p' is still a compatible path with no set in it appearing more that b times.

First, we prove that at the end of step 3, $\widetilde{\Delta}$ appears in p' at most b times. Each block $\mathbf{w_i}$ (generated in step 3b) contains $\widetilde{\Delta}$ exactly once. Looking at the guard of 3ci, it is easy to see that the number of times we append a block to p'' is bounded by the number of formulas in T_Ω plus two (to account for the cases where we add the blocks that contain the first appearance of Δ_x in p' or the last set of p'). That is, we add at most b blocks, and so $\widetilde{\Delta}$ appears in p' at most b times.

Second, we prove that this procedure halts. This is easy to see, as every time we enter step 3, the number of sets which appear in p' more than b times decreases by (at least) one. Thus, it eventually reaches zero. At that point the procedure halts.

Thus, the procedure always halts and at the end of it, p' is a sequence of sets where each set appears at most b times.

Finally, we prove that at the end of the procedure p' is still a compatible path which starts at Δ_0. Let

$$p' = [\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$$

where Δ_x and Δ_n correspond to the first appearance of Δ_x in p and the last set of p (notice that these sets are never removed in the procedure). Indeed, it must start at Δ_0 because every time we enter 3, either $\widetilde{\Delta} \neq \Delta_0$, and then at the end of 3 the first block is not skipped, or $\widetilde{\Delta} = \Delta_0$, and then at the end of 3 the first set of p' is the first set of a block, which is Δ_0. As for being a compatible path:

1. We prove that all the sets of p' are accessible from the previous one. Every time we enter step 3, we are skipping some blocks. However, by construction of the blocks, the first set of each block (which is $\widetilde{\Delta}$) is always accessible from the last set of any block, thus we may "short circuit" the

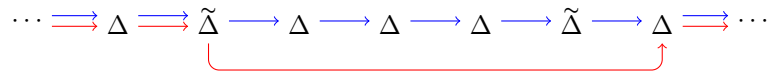


Figure 3.4: The "short circuit" step illustrated. The blue arrows represent p' at the beginning of step 3 and red arrows represent p' at the end of step 3.

sequence while retaining the accessibility relations between each set of p'. This is illustrated in figure 3.4.

2. Each Δ of p' is locally compatible since p' starts as a compatible path.

3. $\Delta_n \longrightarrow \Delta_x$.

4. By construction of p', it fulfills the final conditions.

□

We are ready to prove the decidability result.

Proposition 3.19. *The satisfiability problem for sets of formulas is decidable.*

Proof. Let Ω be a set of formulas. Compute $C(\Omega)$. This can be done, at it is a finite set of finite sets. Then, search within it for a compatible path starting at some $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\})^*$ such that no set appears more than $b = \#(\mathsf{T}_\Omega) + 2$ times. Since the number of sets which can appear in a sequence is finite and each set might appear at most a finite number of times, the number of sequences that we need to test is finite. Thus, our search eventually halts. By proposition 3.18, Ω is satisfiable iff a compatible path is found. □

This procedure can be somewhat optimized, as $C(\Omega)$ can be generated "on the fly" and the bound b is not tight. We will discuss this in section 5.1.

Proposition 3.20. *The validity problem for formulas is decidable.*

Proof. Let $\varphi \in L_\Sigma$. By proposition 2.12, φ is valid iff $\{\neg\varphi\}$ is not satisfiable. This problem is decidable by proposition 3.19. □

Chapter 4

Model Checking

Recall that the objective of model checking is to model a certain environment using some standard structure, and then use it to check if the environment satisfies some property expressed by an EBTSCL formula. In this chapter we define our modelling structure, the transition system, and define the path-satisfaction and satisfaction of an EBTSCL formula by a transition system. Then, we define the event transition system. It allows us to more naturally model some environments. Afterwards, we define the product of two transition systems and of two event transition systems, which allows us to model the environment by modelling each of its component separately. Finally, we prove that the path-satisfiability and the satisfiability of an EBTSCL formula by a transition system is decidable.

4.1 Transition System

In this section we define a transition system, introduce the product operation over transition systems and define satisfiability of an EBTSCL formula by a transition system.

Definition 4.1. A transition system T over a signature $\Sigma = (\mathbb{P}, \mathbb{E}, \mathbb{T}, \mathbb{A})$ is a tuple

$$T = \left(<^T, \cong^T, \{ \trianglelefteq_p^T \}_{p \in \mathbb{P}}, S, I, \longrightarrow, L \right)$$

where

- $<^T$ is a relation over $\mathbb{T}$;
- $\cong^T$ is a relation over $\mathbb{T}$;
- $\trianglelefteq_p^T$ is a trust relation over $\mathbb{A}$ for each $p \in \mathbb{P}$;
- S is a set of states;
- $I \subset S$ is a set of initial states;
- $\longrightarrow \subset S \times S$ is a transition relation. For convenience, we write $s \longrightarrow s'$ instead of $(s, s') \in \longrightarrow$;

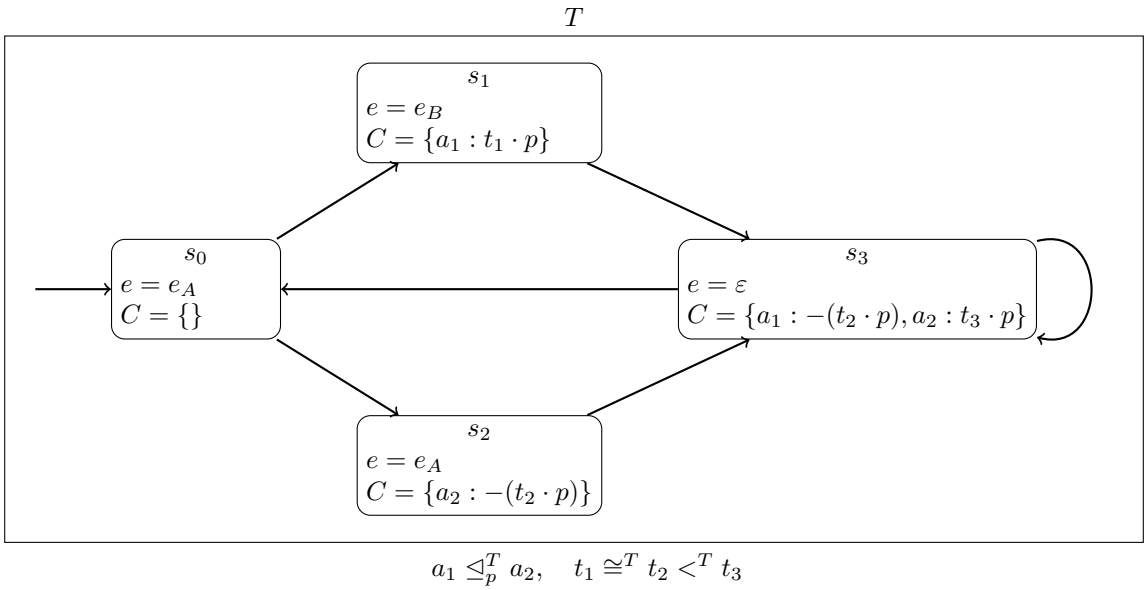


Figure 4.1: A very simple transition system. We use the notation $e = e'$ and $C = C'$ within a set s to mean that $L(s) = (e', C')$

- $L : S \to (E \cup \{\varepsilon\}) \times \mathcal{P}(AK_\Sigma)$ is a labeling function, which assigns a pair (e, C) to each state of T. C is a set of agent claims and e is either an event symbol or ε – the null event.

We can represent a transition system graphically, as can be seen in figure 4.1.

Given a transition system T with state set S and $s \in S$, we denote by $\mathrm{Reach}_T(s)$ the set $\mathrm{Reach}_T(s) = \{s' \in S \mid s \longrightarrow s'\}$. A state s is said to be terminal if $\mathrm{Reach}_T(s) = \varnothing$. From now on we assume that all our transition systems do not have terminal states.

From a transition system we are able to extract multiple interpretation structures following the various paths present in the transition system. One might then think that a transition system is somewhat like an interpretation structure, but with an automaton-like structure. Indeed, that is what a transition system is for Linear Time Logic (as presented in [4]). However, in our case there are some key differences between a transition system and an interpretation structure. Since the user defines the transition system, we are less restrictive about the conditions it must meet.

Note that although similar, the labels of each state are not local interpretation structures. The user is free to declare the transition system disregarding any of the sensible restrictions that a pair (e, C) must fulfill to be considered a local interpretation structure. Of course this will usually yield a transition system which satisfies no formula (we call it a contradictory transition system). More specifically, the user can define a label of a state with $C = \{a : t \cdot p, a : -(t \cdot p)\}$. Of course this results in a state of contradiction. If this state is reachable, then we can say that the transition system is contradictory. Regarding the time and trust orders, we also allow the user to declare, for example, $t <^T t$, thus violating irreflexivity. This immediately results in a contradictory transition system.

There are other, more intricate, ways in which a transition system is different from an interpretation structure. For that we must distinguish between descriptions and restrictions.

Remark 4.2. [Descriptions vs Restrictions] A transition system is a representation of the environment.

It is then crucial that we understand how not only what we *declare* is interpreted, but also what we *do not declare*. For this we divide the declarations into two groups: descriptions and restrictions. These concepts might seem abstract, but will become clear as we provide examples.

A **description** is a declaration which intends to describe exactly how the environment is. This means that whenever a description is not declared, we must consider that it is not part of the system.

A **restriction** is a declaration which intends to present a restriction that our model must fulfil in order to model the environment. This means that whenever a restriction is not declared, we consider that it might or not be part of the environment.

We now categorize the declarations that compose a transition system. There are five types of declarations possible:

1. **Transition**, i.e., of the type $s_1 \longrightarrow s_2$. It is a **description**. If $(s_1, s_2) \notin \longrightarrow$ then we are saying our transition system must never transition from s_1 to s_2.

2. **Time Order**, i.e., of the type $t_1 <^T t_2$ or $t_1 \cong^T t_2$. It is a **restriction** for two reasons

   - the transitivity of $<$ and $\cong$. If we declare only $\{t_1 <^T t_2, t_2 <^T t_3\}$ then our model checking algorithm can (and will) assume that $t_1 <^T t_3$, although this was not declared;
   - the connectedness of $<$. Suppose $\mathbb{T} = \{t_1, t_2\}$, but the user has not specified the time relation between t_1 and t_2 (because it is not known for example). Suppose we are trying to decide whether T path-satisfies some formula φ (we will define path-satisfiablility later). Although there is no relation specified, we know that any two time-stamps must be in relation, so there are only three possibilities: $t_1 <^T t_2$, $t_1 \cong^T t_2$ and $t_2 <^T t_1$. The model-checking algorithm might then output, for example, "T path-satisfies φ so long as $t_1 <^T t_2$".

   That is, if $(t_1, t_2) \notin <^T$ that does not mean that our system does not allow that $t_1 < t_2$, just that it does not demand that $t_1 < t_2$. If the user then realizes that it is not admissible that $t_1 < t_2$ then they must declare that either $t_2 <^T t_1$ or $t_1 \cong^T t_2$.

3. **Event**, within the label of a state. It is a **restriction**. Labeling a state with ε (i.e. not declaring an event) represents the absence of an event restriction on that state. The transition system may take any event in that state.

4. **Trust Order**, i.e., of the type $a_1 \trianglelefteq_p^T a_2$. It is mostly a **description**, except for the transitivity of $\trianglelefteq_p$. Indeed, if $a_1 \trianglelefteq_p^T a_2$ and $a_2 \trianglelefteq_p^T a_3$ then we can assume that $a_1 \trianglelefteq_p^T a_3$. However, that is where it ends. We cannot do as we did for the time order, as there may be a pair of agents which is not in relation! For example, assume $\mathbb{A} = \{a_1, a_2\}$. We ask the algorithm to infer whether T path-satisfies φ. The algorithm returns "T path-satisfies φ so long as $a_1 \trianglelefteq_p^T a_2$". But what if the user wants to specify a system where neither $a_1 \trianglelefteq_p a_2$ nor $a_2 \trianglelefteq_p a_1$? The only way to declare such a thing would be to declare that $\neg a_1 \trianglelefteq_p a_2$ and $\neg a_2 \trianglelefteq_p a_1$. However, consider now another case where we have many agents and propositional symbols but no two agents are in relation. The user would then have to actively declare that $\neg a_1 \trianglelefteq_p a_2$ for all $a_1, a_2 \in \mathbb{A}$. In our opinion this is not desirable, so we

chose to interpret trust order declarations as descriptions. That is, whenever a statement of the type $a_1 \trianglelefteq_p^T a_2$ is *not* declared, the statement $\neg a_1 \trianglelefteq_p^T a_2$ is implicitly declared except when we can infer $a_1 \trianglelefteq_p^T a_2$ with the transitivity of $\trianglelefteq_p^T$.

5. **Agent Claim**, i.e., of the type $a : \varphi$ within the label of a state. It is mostly a **description** except for the congruent closure with the time and trust orders. We can interpret it like this: what is declared in the transition system is an exact description of what was said, while the statements obtained by congruent closure correspond to what was thought. As an example, suppose that in some transition system the user has declared $\{t_1 \cong^T t_2, a_1 \trianglelefteq_p^T a_2, a_2 \trianglelefteq_p^T a_1\}$ and in some state's label we have $a_1 : t_1 \cdot p$. Because t_1 and t_2 represent the same time point, we can conclude that even though a_1 did not say it, they must surely think that $t_2 \cdot p$. Similarly, because a_1 and a_2 are equally trustworthy with regard to p, we can infer that a_2 agrees with a_1 on statements regarding p, i.e., we can deduce $a_2 : t_1 \cdot p$ and $a_2 : t_2 \cdot p$. That is, whenever a statement of the type $a : \varphi$ is *not* declared, the statement $\neg a : \varphi$ is implicitly declared except when we can infer $a : \varphi$ with the congruent closure of the time and trust orders.

We are now ready to define the satisfiability of a formula by a transition system. First we define a path

Definition 4.3. A path π over a transition system $T = \left(<^T, \cong^T, \left\{\trianglelefteq_p^T\right\}_{p\in\mathbb{P}}, S, I, \longrightarrow, L\right)$ is a map

$$\pi : \mathbb{N} \to S$$

such that

(a) $\pi(0) \in I$

(b) $\pi(i) \longrightarrow \pi(i+1)$ for all $i \in \mathbb{N}$

A path is periodic if it can be written as

$$\pi = s_0\ s_1\ \ldots\ s_n(s'_0\ s'_1\ \ldots\ s'_m)^\omega$$

We denote by $\mathrm{Paths}(T)$ the set of all paths over T.

There are two meaningful ways in which we can check if the environment satisfies some property – path-satisfaction and satisfaction. Intuitively, if a transition system path-satisfies a formula, then there is some interpretation consistent with the given description of the environment such that the interpretation satisfies said formula. On the other hand, if a transition system satisfies a formula, then all possible interpretations which are consistent with the environment description must satisfy the formula. The latter one is the usual goal of model-checking: we want to make sure that whatever happens, our environment always satisfies some key property.

A transition system is a description of the environment. That is, it encodes what types of behaviors we can expect from such an environment. That is, from a transition system we can extract a multitude of interpretation structures, which represent one way in which the environment can behave. We must then

define what interpretation structures can be extracted from some transition system. If an interpretation structure is a valid extraction from a transition system, we say it is consistent with it. We define this concept now.

Definition 4.4. Let T be a transition system with state set S and labeling function L, and let $s \in S$ with $L(s) = (e_s, C_s)$. Let M be an interpretation structure and $I = (e_I, C_I)$ be a local interpretation structure over M. Let $R^T_{\trianglelefteq}$ and $R^M_{\cong}$ be the sets of formulas which encode the $\trianglelefteq^T_p$ relation for each p and the $\cong^M$ relation. That is,

$$R^T_{\trianglelefteq} = \bigcup_{p \in \mathbb{P}} \left\{ a_1 \trianglelefteq_p a_2 \;\middle|\; a_1 \trianglelefteq^T_p a_2 \right\} \qquad R^M_{\cong} = \left\{ t_1 \cong t_2 \;\middle|\; t_1 \cong^M t_2 \right\}$$

We say that a local interpretation structure $I = (e_I, C_I)$ over an interpretation structure M is consistent with s, and write $I \parallel s$ if

- $e_I = e_s$ or $e_s = \varepsilon$;
- $\mathrm{OperatorClosure}\left(C_s \cup R^T_{\trianglelefteq} \cup R^M_{\cong}\right) \cap AK_\Sigma = C_I$.

Recall that by the definition of local interpretation structure, the set C_I is closed under the congruent closure of $\trianglelefteq^M_p$ and $\cong^M$. Notice that we use the $\trianglelefteq_p$ relation from T but the $\cong$ relation from M. The reason for this difference comes from what we have discussed in remark 4.2. Intuitively, a local interpretation structure is consistent with a state if their events match and their set of claims match up to the congruent closure of $\trianglelefteq^T_p$ and $\cong^M$.
Returning to a previous example, if there is a state s such that $\{a : t \cdot p, a : -(t \cdot p)\} \subset C_s$, then there is no local interpretation structure which is consistent with s, as this would violate the definition of local interpretation structure.

Definition 4.5. Given a transition system $T = \left(<^T, \cong^T, \left\{\trianglelefteq^T_p\right\}_{p \in \mathbb{P}}, S, I, \longrightarrow, L\right)$ and a path π over T, we say that an interpretation structure $M = \left(<^M, \cong^M, \left\{\trianglelefteq^M_p\right\}_{p \in \mathbb{P}}, \mathcal{I}\right)$ is consistent with π, and write $M \parallel \pi$, if

- $<^T \subset <^M$;
- $\cong^T \subset \cong^M$;
- $\trianglelefteq^T_p$ is equal to $\trianglelefteq^M_p$ for each $p \in \mathbb{P}$, up to the transitive and congruent closure of $\trianglelefteq^T_p$. That is, $\trianglelefteq^T_p \subset \trianglelefteq^M_p$ and if $a_1 \trianglelefteq^M_p a_2$ then one can obtain $a_1 \trianglelefteq^T_p a_2$ from $\trianglelefteq^T_p$ using transitive or congruent closure;
- the local interpretation structure $\mathcal{I}(i)$ is consistent with the state $\pi(i)$ for all $i \in \mathbb{N}$.

Definition 4.6. We say that a path π over a transition system T satisfies a formula φ, and write $\pi \Vdash \varphi$, if there is an interpretation structure M consistent with π such that $M \Vdash \varphi$.
We say that a transition system T path-satisfies φ, and write $T \Vdash \varphi$ if there is $\pi \in \mathrm{Paths}(T)$ such that $\pi \Vdash \varphi$.
These definitions extend to sets of formulas in the usual way.

We now precisely define of a contradictory transition system.

Definition 4.7. We say that a transition system T is non-contradictory if for all $\pi \in \mathrm{Paths}(\pi)$ there is an interpretation structure M such that $M \parallel \pi$.

Because the environment itself cannot be contradictory, a contradictory transition system is a sign of bad modelling. However, a badly modeled environment might be enough to assert path-satisfaction. For satisfaction this is not possible.

Definition 4.8. Let T be a non-contradictory transition system. We say that a path π over T entails a formula φ, and write $\pi \vDash \varphi$ if $M \Vdash \varphi$ for all $M \parallel \pi$.
We say that T satisfies a formula φ, and write $T \vDash \varphi$, if for all paths $\pi \in \mathrm{Paths}(T)$, $\pi \vDash \varphi$.

Note that if we did not require T to be non-contradictory then any path with a contradictory state would entail any formula vacuously. In particular, if T was such that $t <^T t$, then T would satisfy any formula.
We establish some properties of $\Vdash$ and $\vDash$ for paths and transition systems.

Proposition 4.9. *Let T be a non-contradictory transition system and let $\pi \in \mathrm{Paths}(T)$. Then,*

1. $\pi \nVdash \varphi$ implies $\pi \Vdash \neg\varphi$. *The converse, in general, is not true.*
2. $\pi \vDash \neg\varphi$ implies $\pi \nvDash \varphi$. *The converse, in general, is not true.*
3. $\pi \Vdash \neg\varphi$ iff $\pi \nvDash \varphi$.
4. $\pi \vDash \neg\varphi$ iff $\pi \nVdash \varphi$.

Proof.

1.

$\pi \nVdash \varphi$
implies for all $M \parallel \pi, M \nVdash \varphi$
implies there is $M \parallel \pi$ such that $M \nVdash \varphi$ (T is non-contradictory)
implies there is $M \parallel \pi$ such that $M \Vdash \neg\varphi$
implies $\pi \Vdash \neg\varphi$

2. Similar to 1.

3.

$\pi \Vdash \neg\varphi$
iff there is $M \parallel \pi$ such that $M \Vdash \neg\varphi$
iff there is $M \parallel \pi$ such that $M \nVdash \varphi$

$$\text{iff } \text{not}(\text{for all } M \parallel \pi, M \Vdash \varphi)$$

$$\text{iff } \pi \nVdash \varphi$$

4. Similar to 3.

□

As a counterexample for the converse of point 1, let T be a transition system which does not declare any relation between the time-stamps. Let π be any path of T. Let φ be $t_1 < t_2$. Let M_1 be an interpretation structure such that $M_1 \parallel \pi$ and $t_2 <^{M_1} t_1$. Let M_2 be an interpretation structure such that $M_1 \parallel \pi$ and $t_1 <^{M_2} t_2$. Then, $M_1 \Vdash \neg\varphi$ and $M_2 \Vdash \varphi$, thus, $\pi \Vdash \neg\varphi$ and $\pi \Vdash \varphi$.

4.2 Defining a Transition System

Defining a transition system can be a tedious task. Often, we would like to express a big complex system in terms of its smaller components. Other times it is not natural to model the environment as a transition system. It would be more practical to model the environment as another, more convenient structure which is then converted to a transition system. [4] presents multiple, more meaningful ways to define a complex transition system in simpler terms. In this section we present three ways: the product of transition systems, the event transition system and the product of event transition systems.

Definition 4.10. Let $T_i = \left(<^{T_i}, \cong^{T_i}, \left\{\trianglelefteq_p^{T_i}\right\}_{p\in\mathbb{P}}, S_i, I_i, \longrightarrow_i, L_i\right), i = 1, 2$ be two transition systems. The product $T_1 \otimes T_2$, is given by

$$T_1 \otimes T_2 = \left(<^{T_1} \cup <^{T_2},\ \cong^{T_1} \cup \cong^{T_2},\ \left\{\trianglelefteq_p^{T_1} \cup \trianglelefteq_p^{T_2}\right\}_{p\in\mathbb{P}},\ S,\ I,\ \longrightarrow,\ L\right)$$

where

- $S = \{\langle s_1, s_2\rangle \in S_1 \times S_2 \mid e_1 = e_2$ or $e_1 = \varepsilon$ or $e_2 = \varepsilon\}$ where e_1, e_2 are such that $L_1(s_1) = (e_1, \cdots)$ and $L_2(s_2) = (e_2, \cdots)$
- $I = \{\langle s_1, s_2\rangle \in S \mid s_1 \in I_1 \wedge s_2 \in I_2\}$
- the transition relation is defined by the following rule

$$\frac{s_1 \longrightarrow_1 s_1' \qquad s_2 \longrightarrow_2 s_2' \qquad \langle s_1, s_2\rangle, \langle s_1', s_2'\rangle \in S}{\langle s_1, s_2\rangle \longrightarrow \langle s_1', s_2'\rangle}$$

- and the labelling function is defined by $L(\langle s_1, s_2\rangle) = (e, C)$ where
  - $\langle s_1, s_2\rangle \in S$
  - Let $L_1(s_1) = (e_1, C_1)$ and $L_2(s_2) = (e_2, C_2)$

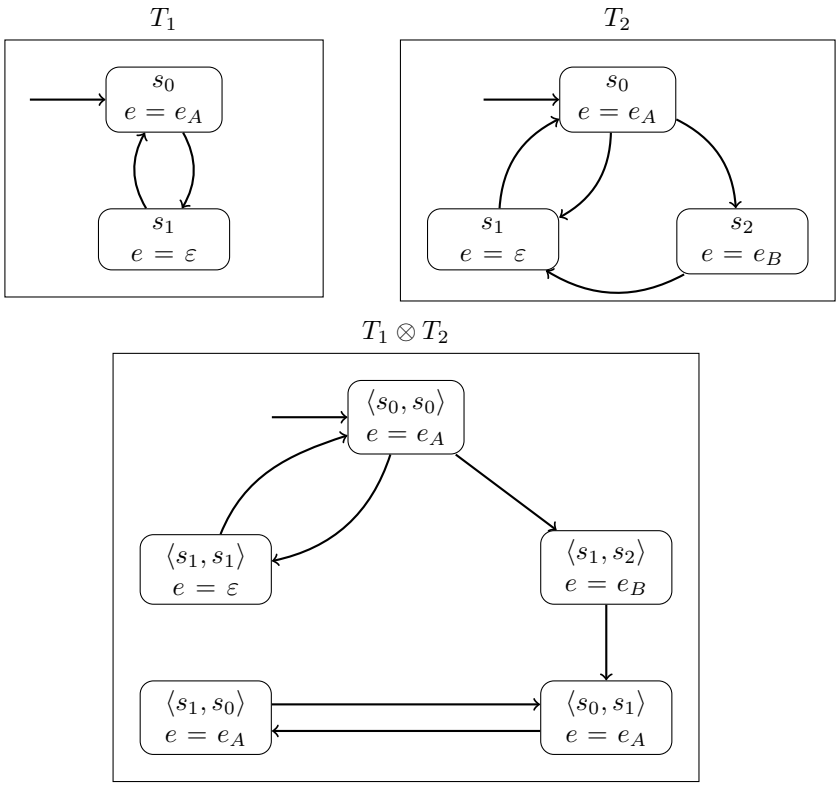


Figure 4.2: Two transition systems and their product. We omit the time and trust relations, as well as the set of agent claims for each set. Notice that the events of T_1 and T_2 condition the structure of $T_1 \otimes T_2$. In particular, state $\langle s_0, s_2 \rangle$ does not exist as the events of the two original states do not match.

- $e = \max(e_1, e_2)$
- $C = C_1 \cup C_2$

Here, $\max(e_1, e_2)$ represents the most restrictive between e_1 and e_2. If both e_1 and e_2 are not ε then they must be equal and so $\max(e_1, e_2) = e_1 = e_2$. If either e_1 or e_2 is ε then $\max(e_1, e_2)$ equals the non-ε event. If both e_1 and e_2 are ε, then $\max(e_1, e_2)$ is ε.

We present a product of two transition systems in figure 4.2. The product of two transition systems is useful when modelling a system which is composed of two independent subsystems. However, one must be very careful when taking the product of two transition systems as it might lead to

- unwanted interactions, for example if we use "a" as an agent symbol in T_1 and T_2 but representing different people, then these agents are considered the same in $T_1 \otimes T_2$. This problem can be avoided by following some better naming conventions or by preforming a variable name change in one of the transition systems.
- non-satisfying transition systems, for example if T_1 contains $t_1 <^{T_1} t_2$ and T_2 contains $t_2 <^{T_2} t_1$, then $T_1 \otimes T_2$ will contain both $t_1 <^{T_1 \otimes T_2} t_2$ and $t_2 <^{T_1 \otimes T_2} t_1$ and thus will not satisfy any formula.

One complaint one might have about transition systems is that it is a bit strange that events are declared in the states' labels. It seems to imply that events happen at the same time as the states. This was motivated by the semantics of the EBTSCL — recall that an interpretation structure consists of a sequence

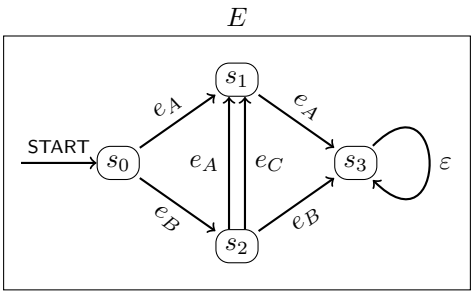


Figure 4.3: An event transition system. We omit the time and trust relations, as well as the set of agent claims for each set.

of pairs (one for each time instant) of an event and a set of claims. However intuitively it makes more sense that events are the driving force that makes a system transition from one state to another. That is, it makes more sense for events to be placed in the transitions of a transition system, rather than in the states. We now define the event transition system which achieves just that.

Definition 4.11. An event transition system T over a signature $\Sigma = (\mathbb{P}, \mathbb{E}, \mathbb{T}, \mathbb{A})$ is a tuple

$$E = \left(<^E, \cong^E, \left\{\trianglelefteq_p^E\right\}_{p\in\mathbb{P}}, S, I, \longrightarrow, L\right)$$

where

- $<^E \subset \mathbb{T} \times \mathbb{T}$ is a relation over $\mathbb{T}$;
- $\cong^E \subset \mathbb{T} \times \mathbb{T}$ is a relation over T;
- $\trianglelefteq_p^E$ is a trust relation over $\mathbb{A}$ for each $p \in \mathbb{P}$;
- S is a set of states;
- $I \subset S$ is a set of initial states;
- $\longrightarrow \subset S \times (E \cup \{\varepsilon\}) \times S$ is a transition relation. For convenience, we write $s \xrightarrow{e} s'$ instead of $(s, e, s') \in \longrightarrow$;
- $L : S \to \mathcal{P}(AK_\Sigma)$ is a labeling function, which assigns a set of agent claims C to each state.

We present a transition system in figure 4.3.
In order to use the event transition system we must provide a procedure that converts it into a regular transition system. The underlying idea consists of pushing the events from the transition to the state to which it gets us. Of course one might reach the same state using transitions with different events. Thus, we will have to expand each state with multiple ones by annotating each state with the event of the transition that takes us to such state.

Definition 4.12. Given an event transition system $E = \left(<^E, \cong^E, \left\{\trianglelefteq_p^E\right\}_{p\in\mathbb{P}}, S_E, I_E, \longrightarrow_E, L_E\right)$, its conversion $T = \mathrm{convert}(E)$ is a transition system defined as follows

$$T = \left(<^T, \cong^T, \left\{\trianglelefteq_p^T\right\}_{p\in\mathbb{P}}, S_T, I_T, \longrightarrow_T, L_T\right)$$

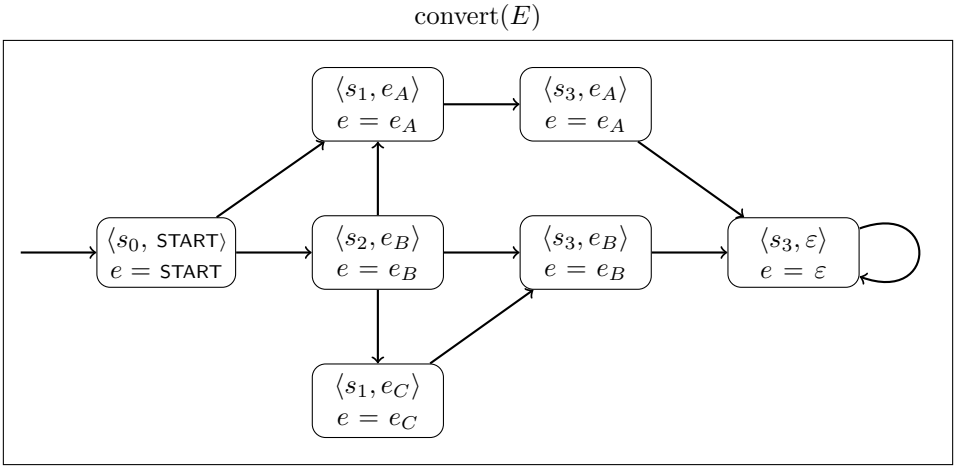


Figure 4.4: The conversion of the event transition system of figure 4.3 into a transition system. We omit the time and trust relations, as well as the set of agent claims for each set.

where

- $<^T$ is equal to $<^E$;
- $\cong^T$ is equal to $\cong^E$;
- $\trianglelefteq_p^T$ is equal to $\trianglelefteq_p^E$ for each $p \in \mathbb{P}$;
- $I_T = \{\langle s, \mathsf{START}\rangle \mid \langle s, e\rangle \in S_T \text{ and } s \in I_E\}$;
- $S_T = I_T \cup \left\{\langle s, e\rangle \;\middle|\; s \in S_E, e \in E \text{ and there is } s' \in S_E \text{ such that } s' \xrightarrow{e} s\right\}$;
- the transition relation is defined by the following rule

$$\frac{s \xrightarrow{e'}_E s' \qquad \langle s, e\rangle \in S_T}{\langle s, e\rangle \longrightarrow_T \langle s', e'\rangle}$$

  that is, if a state $s \in S_E$ gets expanded into $\{\langle s, e_1\rangle, \ldots, \langle s, e_n\rangle\}$ and E contains the transition $s \xrightarrow{e'}_E s'$, then we must add the transitions $\{\langle s, e_1\rangle \longrightarrow \langle s', e'\rangle, \ldots, \langle s, e_n\rangle \longrightarrow \langle s', e'\rangle\}$ to T;

- $L_T(\langle s, e\rangle) = (e, L_E(s))$.

Because states get their event from an incoming transition, the initial states would get no event. Thus, we introduce the special event symbol START. This symbol can be referred to in formulas like any other symbol. For this reason, when we represent an event transition system in a diagram, we annotate the initial states with the event symbol START. We present an event transition system and its conversion in figure 4.4.

Now it is easy to define the satisfiability of a formula by an event transition system:

Definition 4.13. We say that an event transition system E satisfies an EBTSCL formula φ, and write $E \vDash \varphi$, if $\mathrm{convert}(E) \vDash \varphi$. We say that E path-satisfies φ, and write $E \Vdash \varphi$, if $\mathrm{convert}(E) \Vdash \varphi$.

We can also define the product of two event transition systems

Definition 4.14. Let $E_i = \left(<^{E_i}, \cong^{E_i}, \left\{\trianglelefteq_p^{E_i}\right\}_{p\in\mathbb{P}}, S_i, I_i, \longrightarrow_i, L_i\right), i = 1, 2$ be two event transition systems. The product $E_1 \otimes E_2$, is given by

$$E_1 \otimes E_2 = \left(<^{E_1} \cup <^{E_2},\ \cong^{E_1} \cup \cong^{E_2},\ \left\{\trianglelefteq_p^{E_1} \cup \trianglelefteq_p^{E_2}\right\}_{p\in\mathbb{P}},\ S,\ I,\ \longrightarrow,\ L\right)$$

where

- $S = S_1 \times S_2$;
- $I = \{\langle s_1, s_2\rangle \in S \mid s_1 \in I_1 \text{ and } s_2 \in I_2\}$;
- the transition relation is defined by the following rule

$$\frac{s_1 \xrightarrow{e_1}_1 s_1' \qquad s_2 \xrightarrow{e_2}_2 s_2' \qquad e_1 = e_2 \text{ or } e_1 = \varepsilon \text{ or } e_2 = \varepsilon}{\langle s_1, s_2\rangle \xrightarrow{\max(e_1,e_2)} \langle s_1', s_2'\rangle}$$

- $L(\langle s_1, s_2\rangle) = L_1(s_1) \cup L_2(s_2)$

4.3 Decidability

In this section we prove that we can decide in a finite number of steps whether a transition system path-satisfies or satisfies a formula. The procedure is very similar to the one used to decide whether a formula was satisfiable or valid.
We start by defining the set R_T.

Definition 4.15.

$$R_T = \left\{t_1 < t_2 \;\middle|\; (t_1, t_2) \in <^T\right\} \cup \left\{t_1 \cong t_2 \;\middle|\; (t_1, t_2) \in \cong^T\right\} \cup A_T$$

where A_T is the smallest set such that for each $p \in \mathbb{P}$, for each $a_1, a_2 \in \mathbb{A}$, if $a_1 \trianglelefteq_p^T a_2$ or we can obtain $a_1 \trianglelefteq_p^T a_2$ using transitivity, then $a_1 \trianglelefteq_p a_2 \in A_T$, otherwise $\neg a_1 \trianglelefteq_p a_2 \in A_T$. One should not confuse the statement $a_1 \trianglelefteq_p^T a_2$, meaning the pair a_1, a_2 is in the $\trianglelefteq_p^T$ relation of T with the formula $a_1 \trianglelefteq_p a_2 \in L_\Sigma$.

Intuitively, R_T is the encoding of the relations declared in T into formulas. Whenever we want to verify if $T \Vdash \Omega$ we start by adding R_T to Ω to ensure that the time and trust relations declared in T always hold. Note that we do not add the negated formulas in the case of the time relations because of the different nature of the declarations of time and trust orders (see remark 4.2).
We start by defining the local satisfaction of a formula by a state in a transition system.

Definition 4.16. Let T be a transition system and s be a state of such system such that $L(s) = (e, C)$. Let $\Delta \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ be a set of formulas (in particular, $\mathrm{OperatorClosure}(\Delta) = \Delta$). Let R_Δ be the set of formulas in Δ of the type $t_1 \cong t_2$ or $a_1 \trianglelefteq_p a_2$. We say that s locally satisfies Δ, and write $s \Vdash^\ell \Delta$, whenever

- if $e' \in \Delta$ with $e' \in \mathbb{E}$ then either $e = \varepsilon$ or $e = e'$.
- For all $a : \phi \in AK_\Sigma$,

$$a : \phi \in \Delta \text{ implies } a : \phi \in \text{OperatorClosure}(C \cup R_\Delta)$$
$$\text{and}$$
$$\neg(a : \phi) \in \Delta \text{ implies } a : \phi \notin \text{OperatorClosure}(C \cup R_\Delta)$$

- The set $\text{OperatorClosure}(C \cup R_\Delta)$ is locally compatible.

Intuitively, s locally satisfies Δ if it contains all the agent claims in Δ and does not contain any agent claim which contradicts the claims in Δ (up to the closure of the time and trust relations). Recall that a set of claims of a transition system might contain contradictory claims. The last condition accounts for that.
The following proposition relates the concepts of consistency and local satisfaction.

Proposition 4.17. *Let M be an interpretation structure with local interpretation structure sequence $\mathcal{I}$, let $\Delta \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ and s be a state of a transition system T. For all k, if $M, k \Vdash \Delta$ then,*

$$\mathcal{I}(k) \parallel s \text{ implies } s \Vdash^\ell \Delta$$

Proof. Assume $M, k \Vdash \Delta$. Let $\mathcal{I}(k) = (e_k, C_k)$ and $L(s) = (e_s, C_s)$.

- Regarding events, if $e_s = \varepsilon$, it is immediate. Assume $e_s \neq \varepsilon$. Because Δ is satisfied by M at k, then it can only contain one event, say e', and $e' = e_k$. Since $\mathcal{I}(k) \parallel s$ then $e_s = e_k = e'$.
- Notice that since $M, k \Vdash \Delta$, by construction of R_T and τ_Σ, and because World propagates all relation formulas to every set $\Delta \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$, we have that $R_\Delta = R^T_{\trianglelefteq} \cup R^M_{\cong}$. Thus, we must prove that $\text{OperatorClosure}\,(C_s \cup R_\Delta) \cap AK_\Sigma = C_k$ implies $((a : \phi \in \Delta \text{ implies } a : \phi \in \text{OperatorClosure}(C_s \cup R_\Delta)) \text{ and } (\neg(a : \phi) \in \Delta \text{ implies } a : \phi \notin \text{OperatorClosure}(C_s \cup R_\Delta)))$. Assume that $\text{OperatorClosure}\,(C_s \cup R_\Delta) \cap AK_\Sigma = C_k$. If $a : \phi \in \Delta$ then, $a : \phi \in C_k$ because $M, k \Vdash \Delta$, and so $a : \phi \in \text{OperatorClosure}\,(C_s \cup R_\Delta)$. If $\neg(a : \phi) \in \Delta$ then, $a : \phi \notin C_k$ because $M, k \Vdash \Delta$, and so $a : \phi \notin \text{OperatorClosure}\,(C_s \cup R_\Delta)$, since $a : \phi \in AK_\Sigma$.
- By the last point, $\text{OperatorClosure}\,(C_s \cup R_\Delta) \cap AK_\Sigma = C_k$. Since C_k is a locally consistent set of formulas (if it were not, then M would not be a valid interpretation structure), then so is $\text{OperatorClosure}\,(C_s \cup R_\Delta)$.

□

Now we define the product between a transition system and the completion of a set of formulas.

Definition 4.18. Let $T = \left(<^T, \cong^T, \{\trianglelefteq^T_p\}_{p \in \mathbb{P}}, S_T, I_T, \xrightarrow{T}, L_T\right)$ be a transition system, Ω be a set of formulas and $C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ be its completion. We use $\xrightarrow{T}$ to represent the transition relation of T and $\xrightarrow{\Omega}$ to represent the "is accessible from" relation between sets of $C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$. The product

between T and $C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$, denoted by $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$, is a transition system without relations nor labels:

$$T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T) = \left(S_\otimes, I_\otimes, \xrightarrow{\otimes}\right)$$

where,

- $S_\otimes = \left\{\langle s, \Delta\rangle \;\middle|\; s \in S_T \text{ and } \Delta \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T) \text{ and } s \Vdash^\ell \Delta\right\}$
- $I_\otimes = \{\langle s, \Delta\rangle \in S_\otimes \mid s \in I_T, \Delta \in (\Omega \cup \{\tau_\Sigma\} \cup R_T)^*\}$
- the transition relation is defined by the following rule

$$\frac{\langle s_1, \Delta_1\rangle, \langle s_2, \Delta_2\rangle \in S_\otimes \qquad s_1 \xrightarrow{T} s_2 \qquad \Delta_1 \xrightarrow{\Omega} \Delta_2}{\langle s_1, \Delta_1\rangle \xrightarrow{\otimes} \langle s_2, \Delta_2\rangle}$$

Our objective now is to find a finite path within $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ which loops back to itself at some point. We now define satisfying path.

Definition 4.19. We say that a finite sequence $[\langle s_0, \Delta_0\rangle, \langle s_1, \Delta_1\rangle \ldots \langle s_n, \Delta_n\rangle]$ of elements of $S_\otimes$, is a satisfying path over $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ if

1. $\langle s_0, \Delta_0\rangle \in I_\otimes$;
2. $\langle s_0, \Delta_0\rangle \xrightarrow{\otimes} \langle s_1, \Delta_1\rangle \xrightarrow{\otimes} \cdots \xrightarrow{\otimes} \langle s_n, \Delta_n\rangle$;
3. There is $x : 0 \leq x \leq n$ such that $\langle s_n, \Delta_n\rangle \xrightarrow{\otimes} \langle s_x, \Delta_x\rangle$;
4. $[\Delta_0, \ldots, \Delta_x, \ldots \Delta_n]$ is a compatible path with $\Delta_n \xrightarrow{\Omega} \Delta_x$.

Proposition 4.20. *Let T be a transition system, and Ω a set of formulas. Then,*

$$T \Vdash \Omega \text{ iff } \textit{there is a satisfying path over } T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$$

Proof. The proof is very similar to that of proposition 3.17. Thus, we present just an outline.

$\longrightarrow$ Assume $T \Vdash \Omega$. Then there is some $\pi \in \mathrm{Paths}(T)$, $\pi = s_0, s_1, \ldots$ and some interpretation structure M such that $M \parallel \pi$ and $M \Vdash \Omega$. Since $M \parallel \pi$, $M \Vdash R_T$. We build the following infinite sequence of states $u = \langle s_0, \Delta_0\rangle, \langle s_1, \Delta_1\rangle, \ldots$ inductively as follows:

- $\Delta_0 \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ is such that $\Delta_0 \in (\Omega \cup \{\tau_\Sigma\} \cup R_T)^*$ and $M, 0 \Vdash \Delta_0$. Proposition 3.7 guarantees its existence. By proposition 4.17, $s_0 \Vdash^\ell \Delta_0$ and so $\langle s_0, \Delta_0\rangle \in S_\otimes$. In particular, $\langle s_0, \Delta_0\rangle \in I_\otimes$.
- $\Delta_{k+1} \in C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ is such that $\Delta_k \xrightarrow{\Omega} \Delta_{k+1}$ and $M, k+1 \Vdash \Delta_{k+1}$. Proposition 3.16 guarantees its existence. By proposition 4.17, $s_{k+1} \Vdash^\ell \Delta_{k+1}$ and so $\langle s_{k+1}, \Delta_{k+1}\rangle \in S_\otimes$. Since $s_k \xrightarrow{T} s_{k+1}$ then $\langle s_k, \Delta_k\rangle \xrightarrow{\otimes} \langle s_{k+1}, \Delta_{k+1}\rangle$.

Thus, we have built a sequence $u = \langle s_0, \Delta_0\rangle, \langle s_1, \Delta_1\rangle, \ldots$ such that $\langle s_0, \Delta_0\rangle \xrightarrow{\otimes} \langle s_1, \Delta_1\rangle \xrightarrow{\otimes} \cdots$ and $M, k \Vdash \Delta_k$ and $s_k \Vdash^{\ell} \Delta_k$, for all k. This sequence might be non-periodic, but since $S_\otimes$ is finite, there must be some $\left\langle \widetilde{s}, \widetilde{\Delta} \right\rangle$ which appears in u infinitely often. Thus, we use a procedure similar to that of proposition 3.17 to create a periodic sequence

$$u' = \langle s_0, \Delta_0\rangle, \ldots, \langle s_{x-1}, \Delta_{x-1}\rangle, \Big(\langle s_x, \Delta_x\rangle, \ldots, \langle s_n, \Delta_n\rangle\Big)^{\omega}$$

such that $\langle s_n, \Delta_n\rangle \xrightarrow{\otimes} \langle s_x, \Delta_x\rangle$ and such that $[\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$ is a compatible path. Then, it is easy to see that the finite sequence $[\langle s_0, \Delta_0\rangle, \ldots, \langle s_x, \Delta_x\rangle, \ldots, \langle s_n, \Delta_n\rangle]$ is a satisfying path.

$\longleftarrow$ Assume that $[\langle s_0, \Delta_0\rangle, \ldots, \langle s_x, \Delta_x\rangle, \ldots, \langle s_n, \Delta_n\rangle]$ is a satisfying path with $\langle s_n, \Delta_n\rangle \xrightarrow{\otimes} \langle s_x, \Delta_x\rangle$. We must prove that there is some $\pi \in \mathrm{Paths}(T)$ and an interpretation structure M such that $M \parallel \pi$ and $M \Vdash \Omega$. Let $\pi = s_0, \ldots, s_{x-1}\big(s_x, \ldots, s_n\big)^{\omega}$. Clearly, $\pi \in \mathrm{Paths}(T)$.

Let $M = \Big(<^M, \cong^M, \big\{\trianglelefteq_p^M\big\}_{p\in\mathbb{P}}, \mathcal{I}\Big)$ be an interpretation structure defined as follows:

- The time and trust orders $<^M$, $\cong^M$ and $\trianglelefteq_p^M$ are extracted from Δ_n exactly as described in the proof of proposition 3.17.
- Let $I_0, \ldots, I_n$ be local interpretation structures such that, for all $i \in \{0, \ldots, n\}$,
  - Let $L(s_i) = (e_i, C_i)$. We will not use e_i, but rather take the event from Δ_i;
  - The set of claims of I_i is $\mathrm{OperatorClosure}(C_i \cup R_{\Delta_i}) \cap AK_\Sigma$;
  - The event of I_i is equal to the only event formula of Δ_i.

  And now define

$$\mathcal{I} = I_0\ I_1\ \ldots\ I_{x-1}\,(I_x\ I_{x+1}\ \ldots\ I_n)^{\omega}$$

The proof that M satisfies Ω is the same as in the proof of proposition 3.17. Notice that in that proof we set each I_i's set of claims equal to $\Delta_i \cap AK_\Sigma$. But in this one we set it equal to $\mathrm{OperatorClosure}(C_i \cup R_{\Delta_i}) \cap AK_\Sigma$. Since each s_i is locally compatible with Δ_i, this does not alter the fact that $M \Vdash \Omega$.

All that remains is to prove that $M \parallel \pi$. It is easy to see that $<^T \subset <^M$, $\cong^T \subset \cong^M$ and that $\trianglelefteq_p^T$ is equal to $\trianglelefteq_p^M$ (up to transitive closure) for each $p \in \mathbb{P}$ because all the relations of T are encoded into formulas (this corresponds to the set R_T) which are added to Ω in the beginning. By the definition of World these formulas are then propagated to every Δ in the path and so will be extracted into M. By construction of $\mathcal{I}$ it is also clear that $\mathcal{I}(k) \parallel s_k$ for all k.

□

Similarly to what was proved in section 3.2, this result is still not sufficient to prove that the path-satisfiability problem is decidable. Although the state set $S_\otimes$ is finite, the number of finite sequences within it is infinite. Like before, we achieve a finite search space by establishing a bound on the number of times a state might appear in a sequence.

Proposition 4.21. *Let T be a transition system, and Ω a set of formulas. Let $b = \#(\mathsf{T}_\Omega) + 2$. Then, $T \Vdash \Omega$ iff there is a satisfying path $p = [\langle s_0, \Delta_0\rangle, \dots \langle s_x, \Delta_x\rangle, \dots \langle s_n, \Delta_n\rangle]$ over $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ with $\langle s_n, \Delta_n\rangle \xrightarrow{\otimes} \langle s_x, \Delta_x\rangle$ such that each Δ_i appears in $[\Delta_0, \dots, \Delta_x, \dots, \Delta_n]$ at most b times.*

Proof. The proof is analogous to the proof of proposition 3.18. □

We can now prove the two main results of this thesis.

Proposition 4.22. *Let T be a transition system and Ω be set of formulas. Then, we can decide whether $T \Vdash \Omega$ in a finite number of steps.*

Proof. Compute the state set $S_\otimes$ of $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$. This can be done as $S_\otimes$ is finite. Search within it for a satisfying path $p = [\langle s_0, \Delta_0\rangle, \dots \langle s_x, \Delta_x\rangle, \dots \langle s_n, \Delta_n\rangle]$ over $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ such that no set appears in $[\Delta_0, \dots, \Delta_x, \dots, \Delta_n]$ more than b times. Since the number of states which can appear in a sequence is finite and each state might appear at most a finite number of times, the number of sequences we need to test is finite. Thus, our search eventually halts. By proposition 4.21, T path-satisfies Ω iff a satisfying path is found. □

This procedure can be somewhat optimized as we do not need to compute the entire $S_\otimes$ *a priori* and the bound b is not tight. This will be discussed further in section 5.1.
The second result comes as a corollary of the first one. It proves that we can decide whether $T \vDash \varphi$ in a finite number of steps.

Proposition 4.23. *Let T be a non-contradictory transition system, and φ a formula. Then, we can decide whether $T \vDash \varphi$ in a finite number of steps.*

Proof.

$$
\begin{aligned}
& T \vDash \varphi \\
\text{iff}\ \ & \pi \vDash \varphi \text{ for all } \pi \in \mathrm{Paths}(T) \\
\text{iff}\ \ & \mathrm{not}\Big(\text{there is } \pi \in \mathrm{Paths}(T) \text{ such that } \pi \nvDash \varphi\Big) \\
\text{iff}\ \ & \mathrm{not}\Big(\text{there is } \pi \in \mathrm{Paths}(T) \text{ such that } \pi \Vdash \neg\varphi\Big) && \text{(by point 3 of proposition 4.9)} \\
\text{iff}\ \ & T \nVdash \{\neg\phi\}
\end{aligned}
$$

By proposition 4.21, we can decide $T \nVdash \{\neg\phi\}$ in a finite number of steps. □

Chapter 5

Implementation

In this chapter we discuss the implementation of the concepts that we defined in the previous chapters. We start by presenting an overview of the main algorithms used. We then discuss the State Explosion Problem – which is the main limitation of our algorithm. Next, we present the ClaimLang programming language. It is a language which we developed that allows us to define transition systems and EBTSCL formulas concisely. Finally, we present some examples of real-world situations which we model and test for some important properties.

5.1 Algorithms

In this section we explain how we represented the relevant objects (formulas, transition systems, etc.) and give an overview of the algorithms used to decide the satisfaction and path-satisfaction of a formula by a transition system. This was a complex project spanning over multiple files and thousands of lines of code. This section cannot ever fully explain all the intricacies, details and edge cases that had to be considered just so that everything would function correctly. The Java files for two of the main algorithms can be found in appendix C. All the code can be found in the GitHub repository:

https://github.com/Simao-Leal/Model-Checking-with-the-EBTSCL

We decided to implement the formulas of the logic in an object-oriented fashion. Each logical connective is an object of some class and the attributes of each object (if any) are the subformulas that compose the connective. We decided to work in Java, as it is a famously suitable language for object-oriented programming. The fact that it is statically typed allowed us to use overloading and helped us not make mistakes[1] during the development process. This has the added benefit of all the developed code being cross-platform. As an example, in code 5.1 we present part of the definition of the class `Or`, which implements the $\vee$ connective.

We now briefly outline some of the more intricate algorithms that were used, presenting them in pseudocode.

[1] Many mistakes were made, but very rarely these were related with an object's type.

```java
final public class Or extends Formula {
    private Formula inner1;
    private Formula inner2;

    public Or(Formula inner1, Formula inner2) {
        this.inner1 = inner1;
        this.inner2 = inner2;
    }
    ...
}
```

Code 5.1: Part of the Java code from the `Or` class used to implement the $\vee$ connective.

5.1.1 Generating Time Chains

Recall that in definition 3.10, it was mentioned that τ_Σ only included a set of time-stamp chains which represented unique interpretations of the symbols. To achieve this we notice that this problem is equivalent to generating all ordered partitions of the set of time stamps, $\mathbb{T}$. For example, assume that $\mathbb{T} = \{t_1, t_2, t_3, t_4, t_5\}$. Then the ordered partition $[\{t_1, t_4\}, \{t_5\}, \{t_2, t_3\}]$ gets translated into the chain[2] $t_1 \cong t_4 < t_5 < t_2 \cong t_3$. We use a recursive algorithm to calculate all ordered partitions of $\mathbb{T}$ which we present in algorithm 1. We also present the Java implementation of this algorithm in the function `generatePartitions` of class `World` (check the file `World.java` in appendix C.1).

Algorithm 1: Algorithm for generating all ordered partitions of a set of time-stamps.

input : A set T of time-stamps.
output: A set of all ordered partitions of T.

Function `GeneratePartitions`(T)**:**
  if $\#(T) = 1$ **then**
    Let t be the only time-stamp in T
    `// There is only one possible partition`
    return $\{[\{t\}]\}$
  else
    result $\leftarrow \{\}$
    Let $T = \{t_1, \ldots, t_n, t\}$
    `/* Assume we already have all ordered partitions for` $\{t_1, \ldots, t_n\}$`. Let` P `be one of these partitions. We now want to add the time-stamp` t `to` P`. We can either add it to one of the existing sets of` P`, or it can be added in a set by itself,` $\{t\}$`, in which case this set can be inserted into` P `in any position. */`
    foreach *partition* P *in* `GeneratePartitions`($\{t_1, \ldots, t_n\}$) **do** `// Recursion!`
      foreach i *in* `Range(Length(`*P*`))` **do**
        $P' \leftarrow P$ `// make a copy of` P
        $P'[i] \leftarrow P'[i] \cup \{t\}$
        result $\leftarrow$ result $\cup \{P'\}$
      foreach i *in* `Range(Length(`*P*`) + 1)` **do**
        $P' \leftarrow P$ `// make a copy of` P
        P'.insert($\{t\}, i$) `// insert` $\{t\}$ `in` P' `at position` i
        result $\leftarrow$ result $\cup \{P'\}$
  return *result*

[2] We write it in a more concise manner, omitting the $\wedge$ connectives

5.1.2 Satisfaction of a Formula

Recall that by proposition 3.18 Ω is satisfiable if and only if there is some compatible path (composed of sets of $C(\Omega \cup \{\tau_\Sigma\})$) which starts at some set in $(\Omega \cup \{\tau_\Sigma\})^*$. Furthermore, the number of times any set might appear in Δ is bounded, and this bound is determined by the formulas present in Ω. Our algorithm tries to mitigate the cost of this search by not generating the whole of $C(\Omega \cup \{\tau_\Sigma\})$ *a priori*. Instead, we generate the sets $C(\Omega \cup \{\tau_\Sigma\})$ on the fly. We follow the graph structure using either Depth First Search (DFS) or Breadth First Search (BFS). Moreover, we do not use the bound b defined in proposition 3.18. Instead, we keep expanding the current path whenever either the current Δ does not appear previously in the current path or the number of times it appears in the current path is strictly smaller than the number of formulas in the current path which require a witness. The intuition is that one needs at most the same number of blocks after the first appearance of Δ as there are formulas which require witnesses. Thus, if the number of blocks equals the number of formulas which require a witness and a compatible path was not found, then it will never be found, so we stop expanding that path. We present the algorithm (BFS variant) in pseudocode in algorithm 2. The DFS variant is identical, except that we replace the queue of line 3 by a stack. We also present the Java implementation of this algorithm in the function `sat` of class `Sat` (check the file `Sat.java` in appendix C.2). There are certain advantages and disadvantages of using BFS over DFS which we briefly mention. Assume that the set Ω is satisfiable. Thus, there is a compatible path within $C(\Omega \cup \{\tau_\Sigma\})$. On the one hand, BFS will find a minimal interpretation structure which satisfies Ω. By this we mean a periodic interpretation structure whose length is minimal. On the other hand, if the graph structure of $C(\Omega \cup \{\tau_\Sigma\})$ is very wide, then searching for a witness with BFS may take significantly longer. However, if Ω is not satisfiable, then all the sets of $C(\Omega \cup \{\tau_\Sigma\})$ must be visited, and so the algorithm will take the same amount of time to verify this using BFS or DFS.

5.1.3 Path-Satisfaction of a Formula by a Transition System

The algorithm for deciding if a formula is path-satisfied by a transition system is very similar to the previous one. Recall that by proposition 4.21, a transition system T satisfies a set of formulas Ω if and only if there is a satisfying path over $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$. We do not actually generate the whole product structure $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$, but rather we visit the states of T and the sets of $C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ in parallel. We can also choose to preform the search using BFS or DFS. We use the same strategy as in the previous algorithm to decide when to stop expanding the current path. We present the algorithm (BFS variant) in pseudocode in algorithm 3. The DFS variant is identical, except that we replace the queue of line 3 by a stack. We also present the Java implementation of this algorithm in the function `pathSat` of class `TransitionSystem` (check the file `TransitionSystem.java` in the repository in `src`▸`model_checking`▸`TransitionSystem.java`).

The same advantages and disadvantages apply when choosing DFS or BFS. BFS always gives us the smallest witness but might take more time if the product structure is very wide. Furthermore, if Ω is not path-satisfied by T, then all the states in $T \otimes C(\Omega \cup \{\tau_\Sigma\} \cup R_T)$ must be visited and so the search will take the same amount of time using either BFS or DFS. Recall that in order to check if $T \vDash \varphi$, we check

Algorithm 2: Algorithm for deciding if a set of formulas is decidable using BFS.

input : A set $\Omega \subset L_\Sigma$ of formulas.
output: Either $\langle$**false**, **null**$\rangle$ if Ω is not satisfiable or $\langle$**true**, $M\rangle$ if Ω is satisfiable, where M is a witness, that is, a (periodic) interpretation structure which satisfies Ω.

Function `Sat(`Ω`)`**:**
  $\Omega \leftarrow \Omega \cup \{\tau_\Sigma\}$
  `// Initializing queue.`
  $Q \leftarrow$ Queue() `// Each element of` Q `is a list of sets of formulas.`
  foreach Δ *in* Ω^* **do**
    if Δ *is locally compatible* **then**
      Q.push($[\Delta]$)
  `// Begin search`
  found $\leftarrow$ **false**
  while not *found* **and** Q *is not empty* **do**
    path $\leftarrow Q$.pop()
    $\Delta_{\text{last}} \leftarrow$ the last element of path
    foreach Δ *in* $(\mathsf{X}^{-1}(\Delta_{\text{last}}))^*$ **do**
      if Δ *is locally compatible* **then**
        if Δ *is an element of path* **then**
          `// We found a cycle.`
          Let "requirements" be the set of formulas φ such that $\mathsf{T}\varphi$ is in some set of path.
          Let m be the number of times Δ appears in path.
          if *path satisfies the final conditions* **then**
            `// We found a compatible path!`
            found $\leftarrow$ **true**
            compatiblePath $\leftarrow$ path
            $x \leftarrow$ the index such that path$[x]$ equals Δ
          else if $\#($*requirements*$) > m$ **then**
            `// Keep expanding path`
            path.append(Δ)
            Q.push(path)
        else
          path.append(Δ)
          Q.push(path)
  if not *found* **then**
    return $\langle$**false**, **null**$\rangle$
  else
    Let $\Delta_0, \ldots, \Delta_n$ be such that compatiblePath $= [\Delta_0, \ldots, \Delta_n]$
    From $[\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$ with $\Delta_n \longrightarrow \Delta_x$ extract a periodic interpretation structure, M, which satisfies Ω using the procedure which was outlined in the proof of proposition 3.17.
    return $\langle$**true**, $M\rangle$

Algorithm 3: Algorithm for deciding if a transition system path-satisfies a set of formulas using BFS.

input : A transition system T and a set $\Omega \subset L_\Sigma$ of formulas.

output: Either $\langle$**false**, **null**, **null**$\rangle$ if Ω is not path-satisfied by T or $\langle$**true**, $\pi, M\rangle$ if Ω is path-satisfied by T, where $\pi \in \mathrm{Paths}(T)$, $M \parallel \pi$ and M is a (periodic) interpretation structure which satisfies Ω.

Function `PathSat`(T, Ω)**:**
- $\Omega \leftarrow \Omega \cup \{\tau_\Sigma\} \cup R_T$
- `// Initializing queue.`
- $Q \leftarrow$ Queue() `// Each element of` Q `is a list of pairs` $\langle s, \Delta\rangle$.
- **foreach** Δ *in* Ω^* **do**
  - **if** Δ *is locally compatible* **then**
    - **foreach** s *in* I **do**
      - **if** s *locally satisfies* Δ **then**
        - Q.push($[\langle s, \Delta\rangle]$)
- `// Begin search`
- found $\leftarrow$ **false**
- **while not** *found* **and** Q *is not empty* **do**
  - path $\leftarrow Q$.pop()
  - Let s_{last} and Δ_{last} be such that $\langle s_{\text{last}}, \Delta_{\text{last}}\rangle$ is the last element of path
  - **foreach** Δ *in* $(\mathsf{X}^{-1}(\Delta_{\text{last}}))^*$ **do**
    - **if** Δ *is locally compatible* **then**
      - **foreach** s *in* $\mathrm{Reach}_T(s_{\text{last}})$ **do**
        - **if** $s \Vdash^\ell \Delta$ **then**
          - **if** $\langle s, \Delta\rangle$ *is an element of path* **then**
            - `// We found a cycle.`
            - Let $\Delta_0, \ldots, \Delta_n$ be such that path $= [\langle s_0, \Delta_0\rangle, \ldots, \langle s_n, \Delta_n\rangle]$
            - Let "requirements" be the set of formulas φ such that $\mathsf{T}\varphi$ is in some set of $[\Delta_0, \ldots, \Delta_n]$.
            - Let m be the number of times Δ appears in $[\Delta_0, \ldots, \Delta_n]$.
            - **if** $[\Delta_0, \ldots, \Delta_n]$ *satisfies the final conditions* **then**
              - `// We found a satisfying path!`
              - found $\leftarrow$ **true**
              - satisfyingPath $\leftarrow$ path
              - $x \leftarrow$ the index such that path$[x]$ equals $\langle s, \Delta\rangle$
            - **else if** $\#($*requirements*$) > m$ **then**
              - `// Keep expanding path`
              - path.append($\langle s, \Delta\rangle$)
              - Q.push(path)
          - **else**
            - path.append($\langle s, \Delta\rangle$)
            - Q.push(path)
- **if not** *found* **then**
  - **return** $\langle$**false**, **null**, **null**$\rangle$
- **else**
  - Let $s_0, \ldots, s_n$ and $\Delta_0, \ldots, \Delta_n$ be such that satisfyingPath $= [\langle s_0, \Delta_0\rangle, \ldots, \langle s_n, \Delta_n\rangle]$
  - $\pi \leftarrow s_0, \ldots, s_{x-1}(s_x \ldots s_n)^\omega$
  - From $[\Delta_0, \ldots, \Delta_x, \ldots, \Delta_n]$ with $\Delta_n \longrightarrow \Delta_x$ and π extract a periodic interpretation structure, M, which satisfies Ω and such that $M \parallel \pi$ using the procedure which was outlined in the proof of proposition 4.20.
  - **return** $\langle$**true**, $\pi, M\rangle$

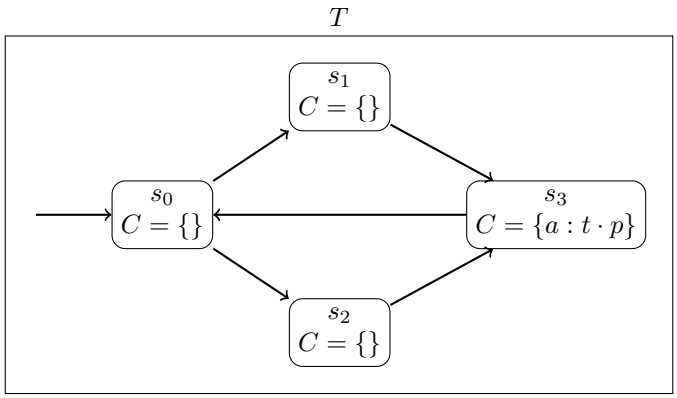


Figure 5.1: A transition system. For each state we omit its event.

if T does not path-satisfy $\{\neg\varphi\}$. Thus, if $T \nvDash \varphi$ then DFS might be faster to find a counterexample than BFS, although BFS always finds the smallest counterexample. But, if $T \vDash \varphi$ then using BFS or DFS is equivalent.

5.2 The State Explosion Problem

One of the shortcomings of our work (and of model-checking in general) is that the techniques we use to verify the satisfaction of a formula under a transition system are very computationally expensive and do not scale well with the complexity of the formula or the transition system. In order to verify that $T \vDash \varphi$ we must exhaustively check all the states in $T \otimes C(\{\neg\varphi\})$ for some potential counter-example. As the number of reachable states of $T \otimes C(\{\neg\varphi\})$ increases, the more time and space is needed to go through all the states. As we will see, the number of states can increase exponentially. We discuss the two types of complexity.

5.2.1 Transition System Complexity

In general, the environment can be a very complex system, which requires an equally complex transition system to model it. Here, we present a minimal example of a transition system exploding in the number of states. Let T be the transition system of figure 5.1. From this small component we make a complex system by taking the product of T with itself many times. Denote by T^n this repeated product, i.e., $T^n = \bigotimes_{i=1}^{n} T$. It is easy to see that T^n has $2 + 2^n$ reachable states – at instant 0 every transition system is in state s_0; at instant 1 each transition system may be in either state s_1 or s_2 (there are 2^n possibilities); at instant 3 all the transition systems are in state s_3 and at instant 4 they all return to state s_0. Thus, the size of T^n increases exponentially. It is obvious that $T^n \vDash \mathsf{F}(a : t \cdot p)$ for any n. However, the time it takes to check this increases very rapidly with n.

Also notice that when modeling the environment with an event transition system, the number of states increases with the conversion to a transition system. This problem is even more exacerbated if we model the environment using a product of event transition systems. Furthermore, for real life situations, event transition systems are usually not complex *enough*. In regular LTL model-checking one would typically

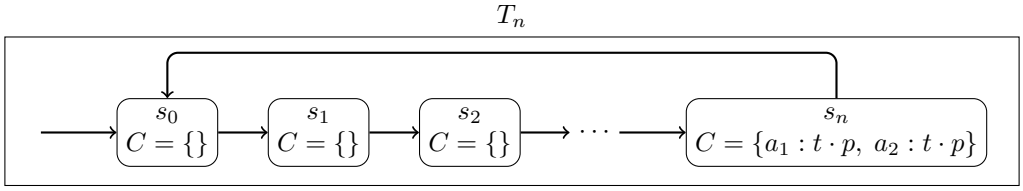


Figure 5.2: A family of transition systems. For each state we omit its event.

model the environment using a program graph (see [4]). This would also increase the number of states, as there we have variables which might take multiple values under a certain domain.

5.2.2 Formula Complexity

Certain formulas φ can greatly increase the size of $T \otimes C(\{\neg\varphi\})$ if there are multiple ways in which φ can be satisfied. For example if $\neg\varphi = \mathsf{G}(\varphi_1 \vee \varphi_2)$, then at every instant our search within $T \otimes C(\{\neg\varphi\})$ can go into two branches – the one where φ_1 holds and the one where φ_2 holds. A constant branching factor induced by the formula, coupled with the multiple paths in the transition system inevitably leads to state explosion. As an example, let $\{T_n\}_{n\in\mathbb{N}_1}$ be the a family of transition systems as defined in figure 5.2, where n is the number of states of the system. Let $\varphi_\wedge$ and $\varphi_\vee$ be the formulas

$$\varphi_\wedge = \mathsf{F}(a_1 : t \cdot p \wedge a_2 : t \cdot p) \qquad \varphi_\vee = \mathsf{F}(a_1 : t \cdot p \vee a_2 : t \cdot p)$$

It is obvious that $T_n \vDash \varphi_\wedge$ and $T_n \vDash \varphi_\vee$ for any n. However, to check that $T_n \vDash \varphi_\wedge$ algorithmically, we need to check that $\mathsf{G}(\neg a_1 : t \cdot p \vee \neg a_2 : t \cdot p)$ is not path-satisfiable. This formula induces a branching factor of 2 over a path of length n in the transition system. Thus, the number of states in $T_n \otimes C(\{\neg\varphi_\wedge\})$ is 2^n, i.e., it increases exponentially. Notice that the state explosion arises not from complexity of the transition system, but from the specific logical connectives present in the formula. Indeed, checking that $T_n \vDash \varphi_\vee$ is much less computationally expensive. In fact, the number of states is equal to the number of states of the transition system, so it increases linearly.

This problem is exacerbated by a specific aspect of the Event-Based Time-Stamped Claim Logic – the hidden complexity of apparently small formulas. One of the main features of the EBTSCL is its ability to assign a truth value to a claim. Recall that we defined a claim $\phi \in K_\Sigma$ to be true whenever

- there is an agent a which claims φ and there is no agent as trustworthy as a which contradicts a;
- for every agent b which claims $-\phi$, there is some other agent as trustworthy as b which contradicts b.

Assume that $\mathbb{A} = \{a, b\}$ then, proving that ϕ is true is equivalent to proving that

$$(a : \phi \wedge a : \boxdot\phi \wedge (b : -\phi \implies \neg b : \boxdot\phi)) \vee (b : \phi \wedge b : \boxdot\phi \wedge (a : -\phi \implies \neg a : \boxdot\phi))$$

is true. Now notice that proving that $a : \boxdot\phi$ is true is equivalent to proving that

$$a \trianglelefteq_\phi b \implies \neg(b : -\phi)$$

is true, which is equivalent to proving that

$$\neg(a \trianglelefteq_\phi b) \vee \neg(b : -\phi)$$

is true. Also notice that proving an implication is true is equivalent to proving that a disjunction is true. The point is that hidden behind the formula ϕ are many disjunctions. The number of disjunctions increases with the number of agent symbols in the signature. As we have seen, disjunctions make the state space explode. Thus, trying to check if a model satisfies a single claim formula can become very computationally expensive, even for a small number of agents involved.

5.2.3 Time Order

Recall that although very similar, our logic is not exactly the same as the logic of [1]. The main difference is that we do not allow the time-stamp and trust orders to change over time. We justified this difference by asserting that since agents cannot make claims about these orders, there is not much of a reason for these relations to change over time. This is a valid reason, but in reality the main motivation came from the state explosion problem. As it turns out, allowing the time order to change instant leads to state explosion, even for very simple transition systems and formulas.

According to [1], in order to guarantee the connectivity of the time relation, the following formula must be added at every instant, that is, every time we apply the closure operation Ω^* to some set of formulas Ω.

$$\bigwedge_{t_1,t_2\in\mathbb{T}} t_1 < t_2 \vee t_1 \cong t_2 \vee t_2 < t_1 \tag{1}$$

Firstly, (1) has many "or" operators which will make the graph structure of $C(\Omega)$ split into many branches. Say, into b branches. For the sake of argument assume that $C(\Omega)$ without (1) does not branch, i.e., it consists of a linear path with depth d. Then, $C(\Omega)$ consists of d sets. If we add (1) at every instant, then the number of sets in $C(\Omega)$ increases to b^d. That is, it leads to state explosion. If $C(\Omega)$ without (1) already contained branches, then this problem is even more exacerbated. On the other hand, by adding (1) only at the beginning, we increase the number of sets in $C(\Omega)$ to just $b \cdot d$. Very early in development we noticed that it was impossible to compute these structures in a sensible amount of time, and so decided to change the semantics of the logic.

Secondly, although 1 encompasses all possible time-stamp orders, it also includes some which are invalid. For example, with three time-stamps, (1) would give us a set $\Delta \in \Omega^*$ with

$$\{t_1 < t_2, t_2 < t_3, t_3 < t_1\} \subset \Delta$$

Applying transitivity of $<$ we would then get $t_1 < t_1$ which would make Δ incompatible. This does not

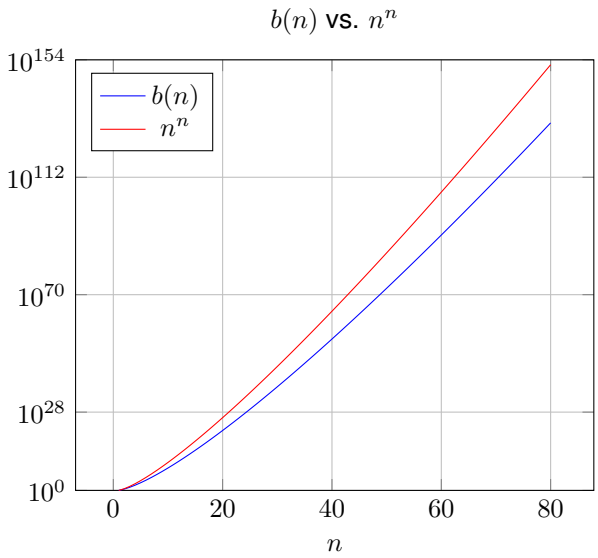


Figure 5.3: The number of branches induced by the formula τ_Σ of definition 3.10 versus the number of branches induced by (1) in function of $n = \#(\mathbb{T})$. Notice that the Y-axis in on a log scale.

affect in any way the proof of the decidability of the logic, but it affects the complexity of the model-checking algorithm. We aim at introducing in Δ only the formulas which encode valid time-stamp orders. For this reason we decided to use τ_Σ, as described in definition 3.10. Let n be such that $\#(\mathbb{T}) = n$. Notice that (1) gives us n^n possible time-stamp orders. The next result establishes that the number of valid time-stamp orders is much smaller.

Proposition 5.1. *If $\mathbb{T}$ contains n time-stamps, then the number of valid time-stamp orders is*

$$b(n) = \sum_{k=1}^{n} k!\ S_2(n,k)$$

Where S_2 represents the Stirling numbers of the second kind.

Proof. Fix k. We assign to each time-stamp a natural number in $\{1,\ldots,k\}$ and extract a time-stamp order by using the usual $<$ and $=$ orders from the natural numbers. This is a time-stamp order with k distinct instants. We obtain all the valid time-stamp orders over $\mathbb{T}$ by combining all the time-stamp orders with k distinct instants for $k = 1,\ldots,n$. Now, the number of time-stamp orders with k distinct instants equals the number of surjective functions from $\mathbb{T}$ to $\{1,\ldots,k\}$, where the number of elements of $\mathbb{T}$ is n. It can be shown that this number is $k!\ S_2(n,k)$ (see [5, section 1.9] for details). Summing over k we get the result. □

By plotting n^n and $b(n)$ together (see figure 5.3), we see that n^n grows much faster than $b(n)$. We also notice that the growth of $b(n)$ is hyper-exponential. We then conclude not only that it is important to introduce only the formulas which encode valid time-stamp orders, but also that in general one should always try to minimize the number of time-stamps used.

```
//create transition system
TransitionSystem T = new TransitionSystem();

//add trust and time relation
T.addTrustRelation(new AgentSymbol("a1"), new PropositionalSymbol("p"), new AgentSymbol("
    a2"));
T.addTimeEqualsRelation(new TimeSymbol("t1"), new TimeSymbol("t2"));
T.addTimeLessThanRelation(new TimeSymbol("t2"), new TimeSymbol("t3"));

//adding state s0
LocalIntStruc s0 = new LocalIntStruc();
s0.setEvent(new EventSymbol("eA"));
T.addInitialState("s0", s0);

//adding state s1
LocalIntStruc s1 = new LocalIntStruc();
s1.addPositiveClaim(new AgentSymbol("a1"), new TimeSymbol("t1"), new PropositionalSymbol(
    "p"));
s1.setEvent(new EventSymbol("eB"));
T.addState("s1", s1);

//adding state s2
LocalIntStruc s2 = new LocalIntStruc();
s2.addNegativeClaim(new AgentSymbol("a2"), new TimeSymbol("t2"), new PropositionalSymbol(
    "p"));
s2.setEvent(new EventSymbol("eA"));
T.addState("s2", s2);

//adding state s3
LocalIntStruc s3 = new LocalIntStruc();
s3.addPositiveClaim(new AgentSymbol("a2"), new TimeSymbol("t3"), new PropositionalSymbol(
    "p"));
s3.addNegativeClaim(new AgentSymbol("a1"), new TimeSymbol("t2"), new PropositionalSymbol(
    "p"));
T.addState("s3", s3);

//adding transitions
T.addTransition("s0", "s1");
T.addTransition("s1", "s3");
T.addTransition("s3", "s2");
T.addTransition("s2", "s3");
T.addTransition("s3", "s3");
```

Code 5.2: Java code for defining the transition system of figure 4.1

5.3 ClaimLang

A problem with working in an object-oriented paradigm is that the code we write can very easily become quite verbose. As an example, observe how complex it is to define the simple formula $a_1 : \boxdot(t_1 \cdot p_1)$:

```
Formula phi = new MostTrustworthy(new AgentSymbol("a1"), new Claim(new TimeSymbol("t1"),
    new PropositionalSymbol("p1")));
```

Although it is true that this code could be a bit simplified by including the calls to the symbols' classes within the constructor of `MostTrustworthy` and `Claim`, there is no escaping the lengthy class names for each of the logical connectives. Defining a transition system is even more verbose. In code 5.2 we present the code required to define the transition system of figure 4.1.

There is no point in creating the model-checking algorithm if we have to waste so much time defining a transition system as simple as that of figure 4.1. User-friendliness should always be an important factor when working in model-checking. Hence, the need arose to create some tool which allowed us to write formulas define transition systems succinctly. And thus ClaimLang was born.

ClaimLang is a programming language for defining transition systems, EBTSCL formulas and querying

```
TransitionSystem T {
    //Defining time and trust orders
    TrustOrder  a1 <[p] a2;
    TimeOrder t1 = t2 < t3;

    //Defining states
    Initial State s0 {
        Event(eA);
    }

    State s1 {
        Event(eB);
        a1 : t1 . p;
    }

    State s2 {
        Event(eA);
        a2 : - t2 . p;
    }

    State s3 {
        a1 : - t2 . p;
        a2 :   t3 . p;
    }

    //Defining transitions
    s0 -> s1 -> s3 -> s2 -> s3 -> s3;
}
```

Code 5.3: ClaimLang code for defining the transition system of figure 4.1

our algorithm to check the satisfiability or validity of a formula or to check if a transition system satisfies or path-satisfies a formula. Although we call it a programming language, its scope and its computing power are rather limited. It certainly is not Turing complete.

Two Java-based compilers were created to parse this language. The first one parses EBTSCL formulas. The second one parses proper ClaimLang files, using the first compiler whenever a formula is declared. The compiler parses the ClaimLang code and converts it into Java code which it then runs. It is a very simple compiler – of all the usual tasks a compiler does, it essentially only preforms lexical and syntactic analysis and intermediate code generation. All the type checking and symbol handling is done by the Java compiler. It also is not very helpful when dealing with errors – the most expressive error message it will ever present is “Syntax Error”. Despite these limitations, this compiler saves us a lot of trouble. In code 5.3 we present the ClaimLang code equivalent to code 5.2. Notice how concise the ClaimLang code is, compared with the equivalent Java code.

We shall refrain from delving extensively into the intricacies of compiler construction, as that would be outside the scope of this thesis. The interested reader can check, for example, [6]for more details. Very briefly, assume we have some ClaimLang file which we want to parse. We start by writing a lexer specification. It consists essentially of a set of regular expressions which divide the file, a sequence of characters, into a sequence of tokens. Here we also remove comments and whitespace and distinguish between the keywords of our language and identifiers for example. Then, we write a grammar. It consists of a set of production rules which unambiguously describe how the sequence of tokens must be organized, extracting a tree structure which fully describes the file – the abstract syntactic tree. Finally, we traverse this tree and write the appropriate equivalent Java code. The lexer specification files for the formula and ClaimLang compilers can be found in the repository in `src‣parser‣scanner.flex`

and `src‣claimlang‣TSscanner.flex`, respectively. These were then automatically compiled into a Lexer/Scanner – that is, the Java program which divides a file into tokens – by the JFlex tool (see [7]). The grammar for both compilers can be found in appendixes in the repository in `src‣parser‣parser.cup` and `src‣claimlang‣TSparser.cup`, respectively. These were then automatically compiled into a Parser – that is, the Java program which builds a tree from the sequence of tokens – by the CUP tool (see [8]). We also present the lexer and parser specifications in appendixes B.1.1 and B.1.2.
The full ClaimLang compiler can be found in the repository in `src‣claimlang‣ClaimLang.jar`. One can use the compiler on some ClaimLang (`.claim`) file by running the following command on the console

```
java -jar <path to ClaimLang.jar> <path to file.claim>
```

Note that one must have a version of the Java Development Kit (JDK) installed to run the compiler as it needs to compile the Java file it generates. I.e. one must have the `javac` command installed. The compiler was developed for Java version 8 or higher.
Furthermore, a VSCode extension was also developed so that we could have syntax highlighting when writing a ClaimLang file. It can be found in the repository, in the directory `src‣claimlang‣claimlang_extension`. In appendix A.1 we present a concise ClaimLang documentation.

5.4 Application Examples

In this section we present some examples of (somewhat) real world environments, we use the tools which we developed to model said environments and finally use our model-checking algorithm to verify that each one satisfies some key property, expressed in the EBTSCL. In order to showcase the different aspects of our work, two examples are presented. The first one is less focused on modelling and more focused on the interaction of agents, some more trustworthy than others. In the second one, we show how a system can be modelled by a product of event transition systems. The ClaimLang code used for each example and its output can be found in appendixes A.2 and A.3 respectively.

5.4.1 Three Friends

Three friends Alice, Bob and $\mathrm{Charlie}$ are having a long-winded conversation about which city is best. All three of them were born and raised in Lisbon (LIS), and so all consider it a serious contender. However, under the Erasmus program each did a semester abroad. For six months,

- Alice lived in Copenhagen (CPH);
- Bob lived in Munich (MUC);
- $\mathrm{Charlie}$ lived in Berlin (BER).

Furthermore, each friend has visited some of the other cities:

- Alice has visited Munich and Berlin;

- Bob has visited Berlin;
- $\mathrm{Charlie}$ has visited Copenhagen.

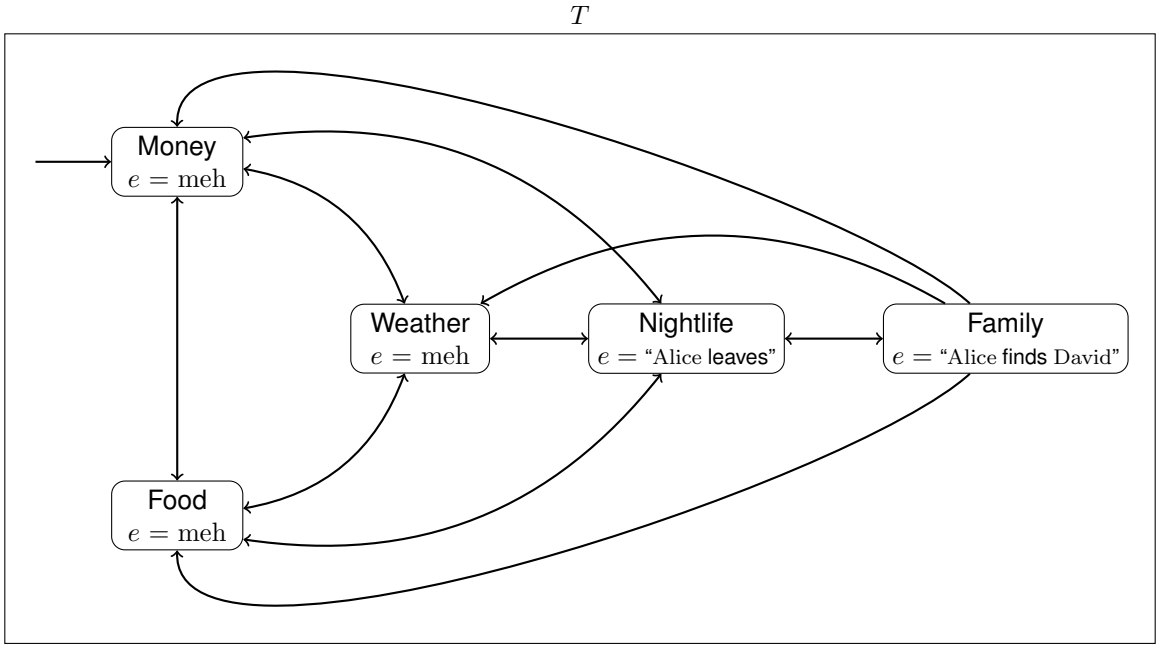
The friends are discussing whether Lisbon is the best city out of those four cities. At every instant each friend might make a statement of the type "Lisbon is better than city X" or "Lisbon is not better than city X". We represent these claims by $t \cdot \mathrm{LIS} \gg \mathrm{CTX}$ and $-(t \cdot \mathrm{LIS} \gg \mathrm{CTX})$, respectively, where CTX is replaced by CPH, MUC or BER. The time-stamp t represents the time of the conversation. We will not use any more time-stamps for this example. We establish that someone who has lived in some city CTX is more trustworthy with regard to $t \cdot \mathrm{LIS} \gg \mathrm{CTX}$ than someone who has just visited CTX. And someone who visited CTX is more trustworthy than someone who has not visited nor lived in CTX. For example $\mathrm{Charlie} \trianglelefteq_{\mathrm{LIS} \gg \mathrm{CPH}} \mathrm{Bob} \trianglelefteq_{\mathrm{LIS} \gg \mathrm{CPH}} \mathrm{Alice}$. However, we cannot assert any trust relation between Alice and Bob with regard to $\mathrm{LIS} \gg \mathrm{BER}$ as we do not know how thorough were their visits. However, we can say that both are certainly less trustworthy than $\mathrm{Charlie}$. The three friends discuss the matter for a long time and sometimes Alice might leave the conversation for one time instant to get some food for everyone. While getting food she may also find her friend David (who studies at Bonn). In that case the two of them spend another time instant talking about the good and the bad about living in Germany. Hence, Alice might leave the main conversation by one or two time instants.

There are many upsides and downsides to living abroad. Thus, the conversation between the three friends approaches five topics which we name "Money", "Weather", "Food", "Nightlife" and "Family". The transition system in figure 5.4 models how the conversation progresses from topic to topic. We now explain each topic, detailing which claims are made by each friend.

Money Both $\mathrm{Charlie}$ and Bob know that the prospective salary of a German worker is much higher than that of a Portuguese one, and thus affirm that Lisbon is not better than their respective cities. When he visited Copenhagen, $\mathrm{Charlie}$ was appalled by how expensive everything was. He also knows that Danish people pay a very high tax rate. He knows salaries are high in Denmark, but he thinks that that cannot outweigh costs. Thus, he affirms that Lisbon is better than Copenhagen. Of course, unlike Alice, $\mathrm{Charlie}$ never went into a Danish supermarket and always had his meals in restaurants. Alice knows that in reality Denmark is not that expensive and that the high tax rate is compensated by Denmark's robust social welfare system. Hence, she states that Lisbon is not better than Copenhagen. That is,

$$C = \{\mathrm{Charlie} : -(t \cdot \mathrm{LIS} \gg \mathrm{BER}), \quad \mathrm{Bob} : -(t \cdot \mathrm{LIS} \gg \mathrm{MUC}),$$
$$\mathrm{Charlie} : t \cdot \mathrm{LIS} \gg \mathrm{CPH}, \quad \mathrm{Alice} : -(t \cdot \mathrm{LIS} \gg \mathrm{CPH})\}$$

Weather When Alice, Bob and $\mathrm{Charlie}$ visited Munich, Berlin and Copenhagen respectively, they had the misfortune of getting rainy weather. Thinking about it, the three friends never got that much rain during their whole life in Lisbon. Each friend then affirms that Lisbon is better than the city they visited. The friend who lived in each city prefers to not make any statement regarding this topic. They obviously acknowledge Lisbon's good weather, but also admit that the infrastructure in Germany



$\text{Bob} \trianglelefteq_{\text{LIS} \gg \text{CPH}} \text{Charlie} \trianglelefteq_{\text{LIS} \gg \text{CPH}} \text{Alice}, \quad \text{Charlie} \trianglelefteq_{\text{LIS} \gg \text{MUC}} \text{Alice} \trianglelefteq_{\text{LIS} \gg \text{MUC}} \text{Bob},$
$\text{Alice} \trianglelefteq_{\text{LIS} \gg \text{BER}} \text{Charlie}, \quad \text{Bob} \trianglelefteq_{\text{LIS} \gg \text{BER}} \text{Charlie}.$

Figure 5.4: A lengthy conversation between Alice, Bob and Charlie, modeled as a transition system. We use the event symbol meh (Mostly Everything Habitual) to say that no relevant event happened. For space reasons we omit from this diagram the set of agent claims for each set and use edges with arrows on both ends to represent two transitions, one in each way.

and Denmark is better well-equipped to deal with rain and cold weather. That is,

$$C = \{\text{Alice} : t \cdot \text{LIS} \gg \text{MUC}, \quad \text{Bob} : t \cdot \text{LIS} \gg \text{BER}, \quad \text{Charlie} : t \cdot \text{LIS} \gg \text{CPH}\}$$

Food When Alice and Bob visited Berlin they had the best *bratwurst* of their lives. And thus affirm that Lisbon is not better than Berlin. Being an avid codfish enthusiast, Charlie makes the opposite claim. When she visited Munich, Alice ate on a budget, and so she did not have the opportunity to enjoy Bavarian cuisine. Hence, she affirms that Lisbon is better than Munich. Bob makes no claim about Munich – although generally enjoying German food, Bob cooked most of the dishes he ate while in Munich, and he only knows how to cook Portuguese food. Regarding Copenhagen, Alice enjoyed the pastries sold there, but she also remembers how bland *smørrebrød* is and the sheer thought of ever tasting licorice candy again makes her want to vomit. Thus, she affirms that Lisbon is better than Copenhagen. That is,

$$\begin{aligned} C = \{&\text{Alice} : -(t \cdot \text{LIS} \gg \text{BER}), \quad \text{Bob} : -(t \cdot \text{LIS} \gg \text{BER}), \quad \text{Charlie} : t \cdot \text{LIS} \gg \text{BER}, \\ &\text{Alice} : t \cdot \text{LIS} \gg \text{MUC}, \quad \text{Alice} : t \cdot \text{LIS} \gg \text{CPH}\} \end{aligned}$$

Nightlife Bob and Charlie start talking about the nightlife in Germany. Never having been too keen on clubbing, Alice takes this opportunity to go get some food for everyone, leaving Bob alone with

$\mathrm{Charlie}$. Both having been to Berlin, they have visited some of that city's world renowned techno discos. Truly, the nights are wild in Berlin. The clubs in Lisbon do not stand a chance. Hence, Bob and $\mathrm{Charlie}$ affirm that Lisbon is not better than Berlin. That is,

$$C = \{\mathrm{Bob} : -(t \cdot \mathrm{LIS} \gg \mathrm{BER}), \quad \mathrm{Charlie} : -(t \cdot \mathrm{LIS} \gg \mathrm{BER})\}$$

Family Once Bob and $\mathrm{Charlie}$ realize Alice will take a while longer to return, they start approaching more sensitive topics. They were a bit embarrassed to admit it in front of Alice, but both of them felt genuinely homesick while they were abroad. Being so far away from their families, friends and anyone who spoke their language was harsh. Six months away was bearable, but moving permanently to Germany is a whole different matter. Truly there is nothing like family, and no amount of money will ever fix that problem. $\mathrm{Charlie}$ affirms that Lisbon is better than Copenhagen, Munich and Berlin. Bob nods silently in agreement.

$$C = \{\mathrm{Charlie} : t \cdot \mathrm{LIS} \gg \mathrm{CPH}, \quad \mathrm{Charlie} : t \cdot \mathrm{LIS} \gg \mathrm{MUC}, \quad \mathrm{Charlie} : t \cdot \mathrm{LIS} \gg \mathrm{BER}\}$$

When they see Alice coming back, they wipe away their tears and resume the conversation as normal. If only they knew Alice felt the same way...

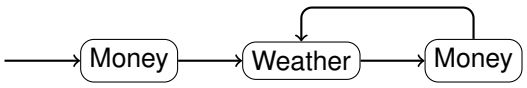
Do the friends come to any conclusion? For each city – Copenhagen, Munich and Berlin – we want to check if the friends at some point converge[3] to the opinion that Lisbon is better than that city. If they converge to that opinion for the three cities, then we conclude that Lisbon is the best city. For example, for Copenhagen we represent the property of converge with the formula $\mathsf{FG}(t \cdot \mathrm{LIS} \gg \mathrm{CPH})$. Thus, we encode the property "The three friends converge to the opinion that Lisbon is the best city" with the following formula:

$$\varphi_{\mathrm{LIS}} = \mathsf{FG}(t \cdot \mathrm{LIS} \gg \mathrm{CPH}) \wedge \mathsf{FG}(t \cdot \mathrm{LIS} \gg \mathrm{MUC}) \wedge \mathsf{FG}(t \cdot \mathrm{LIS} \gg \mathrm{BER})$$

Our objective now is to check if $T \vDash \varphi_{\mathrm{LIS}}$. Recalling the definition of satisfiability for formulas of the type $t \cdot p$, it is not very hard to see that $t \cdot \mathrm{LIS} \gg \mathrm{CPH}$, $t \cdot \mathrm{LIS} \gg \mathrm{MUC}$ and $t \cdot \mathrm{LIS} \gg \mathrm{BER}$ all hold in states "Weather", "Food" and "Family", but do not hold in states "Money" and "Nightlife" (as one would expect). Thus, it is not hard to find paths in T which do not satisfy φ_{LIS}. This is already quite a complex formula to try to model-check. We run the query

$$T \vDash \varphi_{\mathrm{LIS}} \ ?$$

Here we use $?$ to denote a call to the model-checking algorithm to check if $T \vDash \varphi_{\mathrm{LIS}}$. The algorithm returns `Does not satisfy!` and gives you the following counterexample:

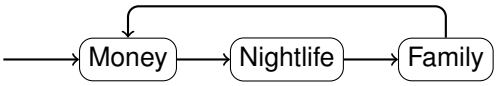


[3]This notion of convergence is inspired by Learning Theory. See [9] for example.

We impose the condition that the topic of family must be discussed. It would be unfair to Lisbon to deprive it of its biggest advantage. We do this by making use of the event "Alice finds David". That is, we want to check if $T \vDash \mathsf{F}(\text{“}\mathrm{Alice}\ \text{finds}\ \mathrm{David}\text{”}) \implies \varphi_{\mathrm{LIS}}$. This is still not enough. For example, if we discuss the topic of Family infinitely often, then we will also discuss the topic of Nightlife infinitely often and so will never converge to the opinion that Lisbon is the best city. We run the query

$$T \vDash \mathsf{F}(\text{“}\mathrm{Alice}\ \text{finds}\ \mathrm{David}\text{”}) \implies \varphi_{\mathrm{LIS}}\ ?$$

Alas, this formula is already too complex for our algorithm (and the author's laptop) to handle. Checking the previous property already took a substantial amount of running time, and in that case there was a counterexample with length 2. In this case, the smallest counter example has length 3. Recall that the number of paths to check increases exponentially with their length. We present a smallest counterexample



Bob and $\mathrm{Charlie}$ are very persuadable individuals. If at some point an opinion is widely accepted, they will defend (or at least refuse to make claims contradicting) that opinion. They do this only for the cities they have lived in. For our purposes we can encode this personality trait as follows:

$$\varphi_{\mathrm{Bob}} = \mathsf{G}(t \cdot \mathrm{LIS} \gg \mathrm{MUC} \implies \mathsf{G}(\neg(\mathrm{Bob} : -(\mathrm{LIS} \gg \mathrm{MUC}))))$$
$$\varphi_{\mathrm{Charlie}} = \mathsf{G}(t \cdot \mathrm{LIS} \gg \mathrm{BER} \implies \mathsf{G}(\neg(\mathrm{Charlie} : -(\mathrm{LIS} \gg \mathrm{BER}))))$$

That is, if at any instant $t \cdot \mathrm{LIS} \gg \mathrm{MUC}$ holds, then for every instant after that, Bob does not affirm that Lisbon is not better than any Munich. Similar for $\mathrm{Charlie}$. Hence, we run the query

$$T \vDash \varphi_{\mathrm{Bob}} \wedge \varphi_{\mathrm{Charlie}} \wedge \mathsf{F}(\text{“}\mathrm{Alice}\ \text{finds}\ \mathrm{David}\text{”}) \implies \varphi_{\mathrm{LIS}}\ ?$$

The algorithm should return `Satisfies!`. Once again, we could not verify with our algorithm. It is not very hard to see that T satisfies this property – if the topic of family eventually is discussed, then in that instant both $t \cdot \mathrm{LIS} \gg \mathrm{MUC}$ and $t \cdot \mathrm{LIS} \gg \mathrm{BER}$ hold, and so $\mathrm{Bob} : -(\mathrm{LIS} \gg \mathrm{MUC})$ and $\mathrm{Charlie} : -(\mathrm{LIS} \gg \mathrm{BER})$ will not hold from that point on. Thus, any path satisfying the antecedent will eventually enter a Food – Weather cycle and consequently φ_{LIS} will hold from some point on.

5.4.2 Smug Thieves

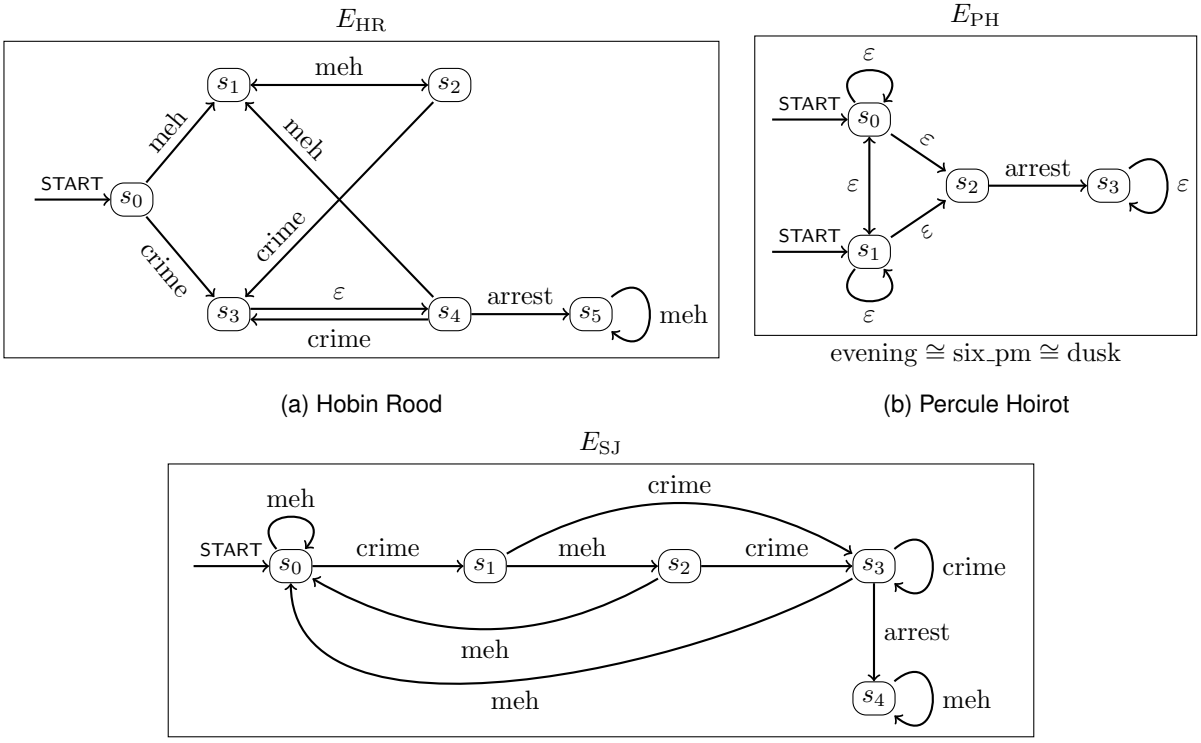
Two infamous thieves – Hobin Rood (HR) and Spack Jarrow (SJ) – have stolen an artifact of great value: the Necklace of Saint Cadoc. Thankfully, the best detective in the world – the great *Monsieur* Percule Hoirot (PH) – has been hired to solve this case and arrest the two thieves. Analyzing the crime scene, PH quickly concludes that HR and SJ are the culprits. This is not the first time PH encountered this evil

duo, and so he immediately recognized their *modus operandi*. However, HR and SJ are very cunning. They have left the crime spotless, and there is no substantial proof that they committed the crime (no fingerprints, no DNA, no video surveillance footage). Although PH is sure that HR and SJ stole the necklace, he needs concrete evidence to present to a judge, and so wiretaps the thieves' phones.
The thieves have no clue their phones are tapped and so go on with their lives as usual. Ever since the theft, HR and SJ have grown increasingly arrogant. They think they are the best robbers in the world (they might be right, it is the Necklace of Saint Cadoc after all). Their smugness shall be their demise. Of course, they know they should not ever mention anything about the theft, but from time to time, given the right circumstances, they cannot resist it and will reveal some important information which incriminates them. We can categorize each robber's smugness into one of three levels:

- **Silent** – the thief makes no remark regarding their criminal activity;
- **Boasting** – the thief makes various claims about their accomplished crime career, but not about the theft of the Necklace of Saint Cadoc;
- **Revealing** – in the middle of his bragging, the thief makes some remark which incriminates him in the theft of the Necklace of Saint Cadoc.

Furthermore, although the Necklace of Saint Cadoc is worth millions, it will take a while before the criminal duo can sell it without alerting the authorities. But these robbers need to survive, so they will commit some small crimes from time to time in order to sustain their lavish lifestyle. Although they have separate lives, they always get together when to commit some crime. Their smugness level will increase for each successful crime they commit. Of course these crimes are nothing compared to the theft of the Necklace of Saint Cadoc. The police will never arrest the duo for these crimes as if they did, there would be no chance they would admit to their major theft.
The detective plans to just wait until both the thieves are recorded admitting their wrongdoing. There is just one small problem with that plan: the wiretap devices have very limited storage – they can only retain recordings of the last 24 hours. Thus, PH must record both criminals admit their crime on the same day as if he arrests only one criminal, the other will flee the country.
In figures 5.5a and 5.5c we present the criminals' behavior throughout PH's investigation. Each state corresponds to some set of claims each thief makes according to their smugness level. This is explained in detail below. Each linear temporal instant represents one day, with events happening during the night. Each night, the duo either commits a small crime or refrains from doing so (which is represented by the event symbols crime and meh respectively). If both of the thieves admit to the theft of the Necklace of Saint Cadoc on the same day, PH will arrest them in that night (which is represented by the event symbol arrest).
We now explain each thief's states in detail.

- **Hobin Rood (HR):** His disposition changes in cycles of two instants. For every two days he is either he is calm (states s_1 and s_2) or ecstatic. If it is the first day when he is calm, he will refuse to commit a crime, in every state other than s_1 and s_5 he might or not commit a crime. Whenever



(a) Hobin Rood

(b) Percule Hoirot

(c) Spack Jarrow

Figure 5.5: The three relevant agents modeled as event transition systems.

he commits a crime, his smugness level will rise to **Boasting** in the next day and to **Revealing** on the following day (regardless of whether a second crime was committed). State s_5 corresponds to his time in prison after his arrest (if it ever happens). Thus, HRis in level

- **Silent** in states s_0, s_1, s_2 and s_5. These states all have claim set $C = \{\mathrm{HR} : -\mathrm{evening} \cdot \mathrm{HR\_guilty}\}$;
- **Boasting** in state s_3 which has claim set $C = \{\mathrm{HR} : -\mathrm{evening} \cdot \mathrm{HR\_guilty}\}$;
- **Revealing** in state s_4 which has claim set $C = \{\mathrm{HR} : \mathrm{evening} \cdot \mathrm{HR\_guilty}\}$;

- **Spack Jarrow (**SJ**):** His disposition is more volatile. Whenever he commits a crime, his smugness level will rise to **Boasting**. Then, if a crime is committed in the following two nights, his level rises to **Revealing**, else, it goes back to **Silent**. If he is at level **Revealing**, then he must commit a crime every night to keep the smugness level. Else, he loses confidence and his level goes back to **Silent**. State s_4 corresponds to his time in prison after his arrest (if it ever happens). Here is the set of claims for each state:
  - **Silent** in states s_0 and s_4. These states all have claim set $C = \{\mathrm{SJ} : -\mathrm{evening} \cdot \mathrm{SJ\_guilty}\}$;
  - **Boasting** in states s_1 and s_3. These states all have claim set $C = \{\mathrm{SJ} : -\mathrm{evening} \cdot \mathrm{SJ\_guilty}\}$;
  - **Revealing** in state s_3 which has claim set $C = \{\mathrm{SJ} : \mathrm{evening} \cdot \mathrm{SJ\_guilty}\}$;

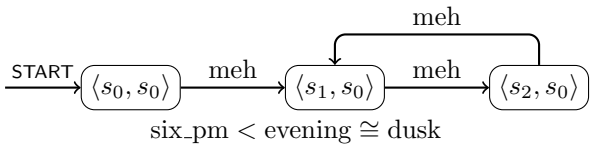
This investigation proceeds during quite some time. After a year from its start and after countless small crimes committed, the head of the precinct – Captain Haymond Rolt – hires you – a proficient model-

checking expert – to access if this investigation will ever come to term. Detective PH is offended that his abilities are being questioned, but agrees that the investigation has been taking more time than usual, and so decides to help you.

Detective PH has provided you with a model for each thief's behavior in the form of an event transition system, so you take their product to model the whole investigation. You want to check if PH will ever arrest the thieves and so you run the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \vDash \mathsf{F}(\mathrm{arrest}) \ ?$$

The algorithm returns `Does not satisfy!` and gives you the following counterexample:



Detective PH immediately protests, saying that the counter example considers that $\mathrm{evening}$ happens after $\mathrm{six\_pm}$, when he perfectly knows that both represent the same time. Furthermore, as it is currently modeled, the criminals will never be arrested, because even if the system is in state $\langle s_4, s_3 \rangle$ (i.e., when both thieves admit to the crime at the same time), there is nothing that forces the system to take the arrest transition.

Indeed, that is missing. You realize that the actions of the detective are also relevant, and so decide to model the behavior of PH into an event transition system. It can be found in figure 5.5b. The purpose of this transition system is twofold:

1. It introduces the restriction $\mathrm{evening} \cong \mathrm{six\_pm} \cong \mathrm{dusk}$.
2. It models the arresting action of the detective. That is, whenever both thieves admit to the crime at the same time, then PH will necessarily take the arresttransition, and so force the whole system to take that transition. This is achieved using a modelling trick which we explain by enumerating the set of claims for each state:

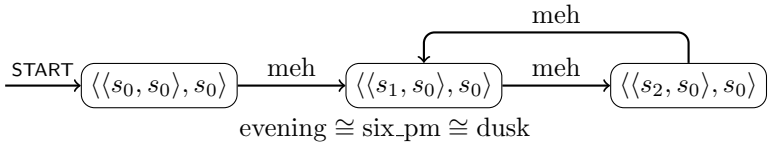
   - State s_0: $C = \{\mathrm{HR} : -\mathrm{six\_pm} \cdot \mathrm{HR\_guilty}\}$;
   - State s_1: $C = \{\mathrm{SJ} : -\mathrm{six\_pm} \cdot \mathrm{SJ\_guilty}\}$;
   - State s_2: $C = \{\mathrm{HR} : \mathrm{six\_pm} \cdot \mathrm{HR\_guilty}, \mathrm{SJ} : \mathrm{six\_pm} \cdot \mathrm{SJ\_guilty}\}$;
   - State s_3: $C = \{\}$.

   Essentially the set of claims for each state makes it so that E_{PH} is in state s_2 if and only if HR and SJ both have admitted their crimes (at the same time) and either in state s_0 or s_1 if none or only one of them have admitted his crime. Note that, for example, the state $\langle \langle s_4, s_1 \rangle, s_2 \rangle$ still exists, but its claim set is $C = \{\mathrm{HR} : \mathrm{six\_pm} \cdot \mathrm{HR\_guilty}, \mathrm{SJ} : \mathrm{six\_pm} \cdot \mathrm{SJ\_guilty}, \mathrm{HR} : \mathrm{evening} \cdot \mathrm{HR\_guilty}, \mathrm{SJ} : -\mathrm{dusk} \cdot \mathrm{SJ\_guilty}\}$, which is inconsistent, when combined with the time-stamp order $\mathrm{evening} \cong \mathrm{six\_pm} \cong \mathrm{dusk}$. Thus, we cannot reach this state. Whenever in s_2, E_{PH} will necessarily transition to s_3 with the arrest event, thus creating the desired effect.

You now run the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}} \vDash \mathsf{F}(\mathrm{arrest})\ ?$$

The algorithm returns `Does not satisfy!` and gives you the following counterexample:

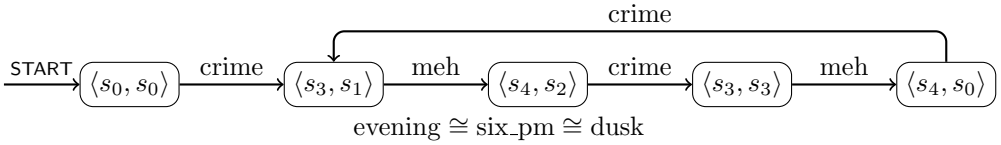


Note that now the counterexample also contains the state of detective PH. One might wonder why the state names are like $\langle\langle s_1, s_0\rangle, s_0\rangle$ and not like $\langle s_1, s_0, s_0\rangle$. This is because $E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}}$ is considered to be two products, each between two transition systems rather that a product between three transition systems. That is, $E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}}$ is parsed as $(E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}}) \otimes E_{\mathrm{PH}}$. From now on we shall omit PH's state, as it can be inferred from the thieves' states. Notice that this counterexample is essentially equal to the previous one.

PH now mentions that in that counterexample, no crime is ever committed. This does not match reality – the duo has not stopped committing crimes, and will continue to do so as they need money to survive. You add the condition that a crime must be committed infinitely often. That is, you run the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}} \vDash (\mathsf{GF}(\mathrm{crime})) \implies (\mathsf{F}(\mathrm{arrest}))\ ?$$

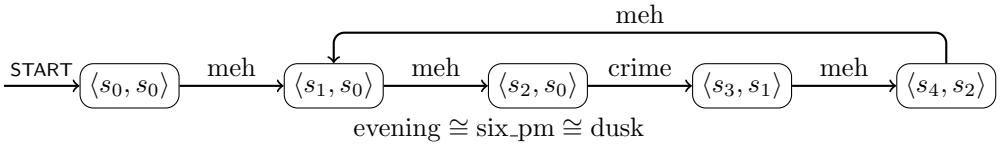
The algorithm returns `Does not satisfy!` and gives you the following counterexample:



Detective PH is appalled. He now conjectures that both thieves admit their crimes infinitely often. You run the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}} \vDash (\mathsf{GF}(\mathrm{crime})) \implies (\mathsf{GF}(\mathrm{HR} : \mathrm{six\_pm} \cdot \mathrm{HR\_guilty})) \wedge (\mathsf{GF}(\mathrm{SJ} : \mathrm{six\_pm} \cdot \mathrm{SJ\_guilty}))\ ?$$

The algorithm returns `Does not satisfy!` and gives you the following counterexample:



In this example, SJ never admits to his crimes. You point out that even if this property were satisfied, this would not mean that both criminals confessed at the same time.

It is at this point in the story where we must imagine that the police station has access to some super-computer, as the author's laptop does not have enough computing power to run the next two queries.

However, it is not very hard to see what the expected result is.
Detective PH notices that in that example, a crime is never committed in two consecutive nights. You consider this scenario by running the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}} \vDash (\mathsf{F}(\mathrm{crime} \wedge \mathsf{X}\mathrm{crime})) \implies (\mathsf{F}(\mathrm{arrest}))\ ?$$

The algorithm should return `Satisfies!`. We could not verify this with our algorithm. Finally, the detective has some strategy to catch the thieves. You now notice that there is a stronger property that also holds. You run the query

$$E_{\mathrm{HR}} \otimes E_{\mathrm{SJ}} \otimes E_{\mathrm{PH}} \vDash \mathsf{G}((\mathrm{crime} \wedge \mathsf{X}(\mathrm{crime})) \implies (\mathsf{XX}(\mathrm{arrest})))\ ?$$

The algorithm should return `Satisfies!`. We could not verify this with our algorithm. That is, whenever crimes are committed in two consecutive nights, PH will arrest the thieves in the third night.
For the next two nights detective PH places some very valuable items in stores close to the thieves' homes. They commit a crime for two nights in a row and PH finally arrests the perpetrators and the Necklace of Saint Cadoc is returned to their rightful owners.
The moral of this story is that crime pays, but only if you do not do it too often.

Chapter 6

Conclusion

6.1 Achievements

We presented the definition of the Event-Based Time-Stamped Claim Logic, making small changes to its semantics to better accommodate the end goal of model checking.

We proved that the satisfiability and validity of a EBTSCL formula is decidable. Our proof is similar to the one in [1]. However, in [1] in order to prove that a formula is satisfiable, we must find a compatible set (which is defined recursively). Our condition for satisfiability involves finding a compatible path. We find our condition to be more intuitive and tangible as the path we find is trivially converted into an interpretation structure. Furthermore, we also proved that whenever a formula is satisfiable, then there is a periodic interpretation structure which satisfies it. This is extremely relevant, as it guarantees that whenever a formula is satisfiable, we can algorithmically produce and output a witness which has a finite definition.

We defined the transition system. Although the concept was heavily inspired by the transition systems of [4], many adaptations had to be made so it could be used as models whose properties are encoded in the EBTSCL. Time-stamp and trust relations were added. There was a significant effort made to define exactly which interpretation structures a transition system encompasses, by explaining not only how a statement present in a transition system is interpreted, but also how a statement which is absent is interpreted.

We defined the event transition system which allows us to model environments in a more natural way by assigning events to the transitions of a system, rather than its states. We defined the conversion procedure between an event transition system and a plain transition system.

We defined the product of two transition systems and of two event transition systems. This allows us to model environments by modelling each of its components separately. Although a very simple composition mechanism with many limitations, it is a powerful tool for modelling a complex system.

We proved that the path-satisfiability and satisfiability of a formula by a transition system is decidable by mimicking the proof that the satisfiability of a formula is decidable. Once again our condition for path-satisfiability involves finding a satisfying path, which is a tangible concept that is seamlessly translated

into an algorithm.
All the concepts which were discussed were implemented in Java. Over that implementation we developed the model checking algorithms.
We discussed the state explosion problem and its impact on our ability to use model checking. This problem is the main hurdle for any model checking algorithm, but it is also exacerbated by specific aspects of the EBTSCL.
We developed the ClaimLang programming language and a Java compiler for it. This was very important as modelling transition systems directly in Java was inelegant and time-consuming. There is no point in developing a model checking algorithm if there is no user-friendly way of using such an algorithm. We also developed a VSCode extension which provides syntax highlighting support for ClaimLang.
Two application examples which highlight the features of the EBTSCL and of our modelling structures were presented. Although these were made up toy-examples, they served their purpose well. These examples also draw attention to the state explosion problem, since even for such small examples the number of states to check was already too overwhelming for our algorithm to reach a conclusion in a sensible amount of time.

6.2 Future Work

There are many aspects which could be improved upon in future works.
Regarding the EBTSCL, it could be expanded as to allow agents to make statements about the time and trust relations. As an example, the statement "some suspect a says that time t_{alibi} for which he has an alibi is the same as the time t_{crime} when the crime was committed" could be represented by the formula $a : t_{\text{alibi}} \cong t_{\text{crime}}$. It would also be interesting to relate the time-stamps when the propositional symbols happened with the time instants when the agents make claims.
Regarding the model checking structures presented, we would benefit from the introduction of more complex structures, like program graphs. More intricate ways of composing transition systems should also be implemented like interleaving or handshaking (see [4]). Ultimately, one could develop an extension to the Promela language (see [10]) so that it could support aspects of the EBTSCL.
Regarding our Java implementation of the decidability procedures, it is clear that it is painfully slow. State explosion will always be a problem in model checking, but what we have verified is that our algorithm takes a significant amount time to check each state. We conjecture that this arises from the vast size of the sets of formulas which have to be analyzed in each state. Although we tried at all times to keep the algorithm as efficient as possible, there might be some simplifications which would significantly improve the time efficiency of our algorithm. In a future work we would also look into parallelization techniques and implement them in our algorithms.

Bibliography

[1] J. Ramos, J. Rasga, C. Sernadas, and L. Viganò. Event-based time-stamped claim logic. *Journal of Logical and Algebraic Methods in Programming*, 121:100684, 2021.

[2] J. Rasga, C. Sernadas, E. Karafili, and L. Viganò. Time-stamped claim logic. *Logic Journal of the IGPL*, 29(3):303–332, 2021.

[3] B. M. C. Almeida. Event-based time-stamped claim logic. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2021.

[4] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[5] R. P. Stanley. *Cambridge studies in advanced mathematics enumerative combinatorics: Series number 49: Volume 1*. Cambridge University Press, Cambridge, England, 2 edition, Dec. 2011.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques, and Tools*. Pearson Addison-Wesley, 2007.

[7] G. Klein, C. S. Ananian, and F. Flannery. *JFlex: The Fast Scanner Generator for Java*. JFlex Project, 2023. URL http://jflex.de/.

[8] C. S. Ananian and F. Flannery. *CUP: LALR Parser Generator for Java*. CUP Project, 2023. URL http://www2.cs.tum.edu/projects/cup/.

[9] D. N. Osherson, M. Stob, and S. Weinstein. *Systems that learn: An introduction to learning theory for cognitive and computer scientists.* The MIT Press, 1986.

[10] G. J. Holzmann. *Design And Validation Of Computer Protocols*. Prentice Hall, Philadelphia, PA, Oct. 1990.

[11] M. Mukund. Linear-time temporal logic and büchi automata. *Tutorial talk*, 1997.

[12] J. Ramos. Lecture notes in logic and model checking, Fall 2021.

Appendix A

ClaimLang Code

A.1 Documentation

```
/*
Claim Lang Documentation

This file contains some examples which should be enough to grasp the main aspects
of ClaimLang syntax. In case of doubt consult the full grammar file.
*/

//=========== Comments ===========

//A line comment!

/*
A multi-
-line comment
*/

//=========== Defining a Transition System ===========

TransitionSystem T1 {

    //=========== Defining the trust and time orders ===========

    TimeOrder{
        //each line is a time-stamp chain separeted by a ";"
        t1 < t2;
        t2 = t3;
        t4 < t5 = t6 < t7;
        //this is interpreted as t4 < t5 and t5 = t6 and t6 < t7
    }
    /*
    If the order consists of only one statement, then the curly
    brackets are not necessary. For example,
```

```
    TimeOrder t1 < t2 = t3;
    */

    TrustOrder{
        /*
        Similar to time order. We write a1 <[p] a2 to denote the
        relation "a2 is less or as trustworthy as a1 with respect
        to p".
        */
        a1 <[p1] a2 <[p2] a3;
        //this is interpreted as a1 <[p1] a2 and a2 <[p2] a3
    }

    //=========== Defining states ===========

    State s0 {
        //each line is a claim made in that state by some agent or the event declaration
        Event(eA);
        a1 : t1 . p1;
        a2 : - t2 . p2;
    }

    //Use the keyword "Initial" to declare a state as initial
    Initial State s1 {
        Event(eB);
        a3 : - t1 . p1;
    }

    //A state can have no claims
    State s2 {
        Event(eC);
    }

    //If no event is declared, the state gets the null event (epsilon)
    State s3 {
        a2 : t2 . p2;
    }

    //=========== Defining transitions between states ===========
    //each line is a chain of transitions
    s1 -> s3;
    s1 -> s1; //a state can have a transition to itself
    s0 -> s2 -> s1 -> s0;
    //this is interpreted as s0 -> s2 and s2 -> s1 and s1 -> s0
}

//=========== Defining an Event Transition System ===========

/*
```

```
Similar to the plain Transtition System, but the events are declared in
the transitions instead of in the states
*/

EventTransitionSystem E1 {
    /*
    Defining the Time Order or Trust Order is optional for
    both the TransitionSystem and EventTransitionSystem
    */

    State s0 {
        a1 : t . p;
    }

    State s1 {
        a2 : t . p;
    }

    State s2 {}

    //we use the notation [e] sx -> sy to denote a transition from sx to sy with event e:
    [eA] s0 -> s1;

    //this is interpreted as before, where the two transitions are with event eB
    [eB] s1 -> s0 -> s2;

    //if no event is declared, the transition gets the null event (epsilon)
    s2 -> s0;
}

//=========== Product and Conversion Declarations ===========

//Assume we have declared some other transition systems T2, E2 and E3.

TransitionSystem Tprod1 = T1 * T2;

EventTransitionSystem Eprod = E1 * E2 * E3;

/*
You cannot multiply a Transition System with an Event Transition System.
*/

TransitionSystem Tprod2 = T1 * E1; //this will result in an error

//=========== Formulas ===========
/*
Within a ClaimLang file we write formulas between two double quotes.
The text inside the quotes is interpreted by the formula compiler.
Some keywords have alternate versions. For example the "square" operator can
```

```
also be represented by "boxdot". Check the lexer specification for more details.
*/

//Atomic
Formula phi1 = "a1 : t . p";
Formula phi2 = "a2 : - t . p";
Formula phi3 = "a2 : square t . p";
Formula phi4 = "- t . p"
Formula phi5 = "t1 < t2";
//We can also use chains for formulas
Formula phi6 = "t2 = t3 < t4";
//This is interpreted as t2 = t3 and t3 < t4
Formula phi7 = "a1 <[p] a2";

//Event
//Just type the name. It gets parsed as an event by the context.
Formula phi8 = "eA"; //in this context, eA is interpreted as an event name.

//Propositional
Formula phi8 = "true";
Formula phi9 = "false";
Formula phi10 = "not p"; //notice that p is parsed as events
Formula phi11 = "p and q";
Formula phi12 = "p or q";
Formula phi13 = "p implies q";
Formula phi14 = "p iff q";

//Linear Temporal
Formula phi15 = "X p";
Formula phi16 = "G p";
Formula phi17 = "F p";
Formula phi18 = "p U q";
/*
Important note:
Names of agent, time, propositional and event symbols must not start
with X, G, or U, as to not be confused with the time operators.
*/

//Operator precedence
/*
Whenever multiple operators are placed in succession, the expression
is evaluated from left to right and following the precedence values of
each operator

+------------+--------------+
| Precedence | Operators    |
+------------+--------------+
| 1          | not          |
+------------+--------------+
```

```
| 2           | and, or       |
+-------------+---------------+
| 3           | iff, implies  |
+-------------+---------------+
| 4           | X, G, F, U    |
+-------------+---------------+
*/
//As an example:
Formula phi21 = "X p and not q implies r implies s";
//Gets interpreted as X(((p and (not q)) implies r) iff s)

//=========== Check Statements ===========

Check phi1 Satisfiable;
Check phi1 Valid;
Check T1 Satisfies phi1;

//One can define the product of transition systems and formulas "on the spot"
Check T1 * T2 Satisfies "G(a1 : t . p)";

/*
If preforming a check on an EventTransitionSystem, it gets converted
into a plain TransitionSystem automatically.
*/
Check E1 Satisfies phi1;
//is equivalent to
Check Convert(E1) Satisfies phi1;
```

A.2 Three Friends

A.2.1 ClaimLang file

```
//Three friends

TransitionSystem T {
    TrustOrder{
        bob <[lis_bt_cph] charlie <[lis_bt_cph] alice;
        charlie <[lis_bt_muc] alice <[lis_bt_muc] bob;
        alice <[lis_bt_ber] charlie;
        bob <[lis_bt_ber] charlie;
    }

    Initial State money {
        Event(meh);
        charlie : - t . lis_bt_ber;
        bob     : - t . lis_bt_muc;
        charlie :   t . lis_bt_cph;
        alice   : - t . lis_bt_cph;
```

```
    }

    State weather {
        Event(meh);
        alice   :   t . lis_bt_muc;
        bob     :   t . lis_bt_ber;
        charlie :   t . lis_bt_cph;
    }

    State food {
        Event(meh);
        alice   : - t . lis_bt_ber;
        bob     : - t . lis_bt_ber;
        charlie :   t . lis_bt_ber;
        alice   :   t . lis_bt_muc;
        alice   :   t . lis_bt_cph;
    }

    State nightlife {
        Event(alice_leaves);
        charlie : - t . lis_bt_ber;
        bob     : - t . lis_bt_ber;
    }

    State family {
        Event(alice_finds_david);
        charlie:    t . lis_bt_cph;
        charlie:    t . lis_bt_muc;
        charlie:    t . lis_bt_ber;
    }
    money     -> weather   -> money;
    money     -> food      -> money;
    food      -> weather   -> food;
    food      -> nightlife -> food;
    weather   -> nightlife -> weather;
    money     -> nightlife -> money;
    nightlife -> family    -> nightlife;
    family    -> money;
    family    -> food;
    family    -> weather;
}

Formula phi_lis     = "((F G t . lis_bt_cph) and (F G t . lis_bt_muc) and (F G t .
    lis_bt_ber))";
Formula phi_bob     = "G(t . lis_bt_muc implies (G(not (bob : - t . lis_bt_muc))))";
Formula phi_charlie = "G(t . lis_bt_ber implies (G(not (charlie : - t . lis_bt_ber))))";

Check T Satisfies phi_lis;

```

```
//Takes too much time to run:
/*
Check T Satisfies "(F alice_finds_david) implies ((F G t . lis_bt_cph) and (F G t .
    lis_bt_muc) and (F G t . lis_bt_ber))";
//This is equivalent to
//Check T Satisfies "(F alice_finds_david) implies $phi_lis";

Check T Satisfies "(G(t . lis_bt_muc implies (G(not (bob : - t . lis_bt_muc))))) and (G(t
     . lis_bt_ber implies (G(not (charlie : - t . lis_bt_ber))))and (F alice_finds_david
    ) implies ((F G (t . lis_bt_cph)) and (F G (t . lis_bt_muc)) and (F G (t . lis_bt_ber
    )))";
// This is equivalent to
// Check T Satisfies "($phi_bob and $phi_charlie and (F alice_finds_david)) implies
    $phi_lis";
*/
```

A.2.2 Output

```
Checking whether the transition system T satisfies the formula ((FG(t · lis_bt_cph) ∧ FG(
    t · lis_bt_muc)) ∧ FG(t · lis_bt_ber))
Does not satisfy!
Counter example:
Path:
Time Order:
        t
Trust Relations:
        bob ⊴[lis_bt_ber] bob, bob ⊴[lis_bt_ber] charlie, alice ⊴[lis_bt_ber] alice,
    alice ⊴[lis_bt_ber] charlie, charlie ⊴[lis_bt_ber] charlie,
        bob ⊴[lis_bt_muc] bob, alice ⊴[lis_bt_muc] alice, alice ⊴[lis_bt_muc] bob,
    charlie ⊴[lis_bt_muc] charlie, charlie ⊴[lis_bt_muc] alice, charlie ⊴[lis_bt_muc] bob
    ,
        bob ⊴[lis_bt_cph] alice, bob ⊴[lis_bt_cph] charlie, bob ⊴[lis_bt_cph] bob, alice
    ⊴[lis_bt_cph] alice, charlie ⊴[lis_bt_cph] charlie, charlie ⊴[lis_bt_cph] alice,
--------------------------
Instant: 0
State: money
Local Interpretation Structure:
Event: meh
Claims: (bob : -(t · lis_bt_muc)), (alice : -(t · lis_bt_cph)), (charlie : -(t ·
    lis_bt_ber)), (charlie : (t · lis_bt_cph)),
--------------------------

<<<START OF CYCLE>>>

--------------------------
Instant: 1
State: weather
Local Interpretation Structure:
```

```
Event: meh
Claims: (bob : (t · lis_bt_ber)), (alice : (t · lis_bt_muc)), (charlie : (t · lis_bt_cph)
    ),
--------------------------
Instant: 2
State: money
Local Interpretation Structure:
Event: meh
Claims: (bob : -(t · lis_bt_muc)), (alice : -(t · lis_bt_cph)), (charlie : -(t ·
    lis_bt_ber)), (charlie : (t · lis_bt_cph)),
--------------------------

<<<END OF CYCLE>>>
```

A.3 Smug Thieves

```
//Thief 1: Hobin Rood
EventTransitionSystem E_HR {

    Initial State s0 {HR : - dusk . HR_guilty;} //Silent

            State s1 {HR : - dusk . HR_guilty;} //Silent

            State s2 {HR : - dusk . HR_guilty;} //Silent

            State s3 {HR : - dusk . HR_guilty;} //Boasting

            State s4 {HR :   dusk . HR_guilty;} //Revealing

            State s5 {HR : - dusk . HR_guilty;} //Silent - arrested

    //Transitions
    //with the meh event
    [meh] s0 -> s1 -> s2 -> s1;
    [meh] s4 -> s1;
    [meh] s5 -> s5;
    //with the null event (epsilon)
    s3 -> s4;
    //with the crime event
    [crime] s0 -> s3;
    [crime] s2 -> s3;
    [crime] s4 -> s3;
    //with the arrest event
    [arrest] s4 -> s5;
}

//Thief 2: Spack Jarrow
EventTransitionSystem E_SJ {
```

```
33
34      Initial State s0 {SJ : - evening . SJ_guilty;} //Silent
35
36              State s1 {SJ : - evening . SJ_guilty;} //Boasting
37
38              State s2 {SJ : - evening . SJ_guilty;} //Boasting
39
40              State s3 {SJ :   evening . SJ_guilty;} //Revealing
41
42              State s4 {SJ : - evening . SJ_guilty;} //Silent - arrested
43
44      //Transitions
45      //with the meh event
46      [meh] s1 -> s2 -> s0 -> s0;
47      [meh] s3 -> s0;
48      [meh] s4 -> s4;
49
50      //with the crime event
51      [crime] s0 -> s1 -> s3 -> s3;
52      [crime] s2 -> s3;
53
54      //with the arrest event
55      [arrest] s3 -> s4;
56  }
57
58  //Detective: Percule Hoirot
59  EventTransitionSystem E_PH {
60
61      TimeOrder evening = dusk = six_pm;
62
63      Initial State s0 {HR : - six_pm . HR_guilty;}
64
65      /*Initial*/ State s1 {SJ : - six_pm . SJ_guilty;}
66
67              State s2 {HR : six_pm . HR_guilty; SJ : six_pm . SJ_guilty;}
68
69              State s3 {} //arrested criminals
70
71      //Transitions
72      //with the null event (epsilon)
73      s0 -> s0;
74      s1 -> s1;
75      s0 -> s1 -> s0;
76      s0 -> s2;
77      s1 -> s2;
78      s3 -> s3;
79      //with the arrest event
80      [arrest] s2 -> s3;
81  }
```

```
82
83  //Defining formulas to check:
84  Formula phi1 = "F(arrest)";
85
86  Formula phi2 = "(G F crime) implies (F(arrest))";
87
88  Formula phi3 = "(G F crime) implies (G F(HR : six_pm . HR_guilty)) and (G F(SJ : six_pm .
        SJ_guilty))";
89
90  Formula phi4 = "(F (crime and (X crime))) implies (F(arrest))";
91
92  Formula phi5 = "G( (crime and (X crime)) implies (X X (arrest)) )";
93
94  //Checking formulas
95  Check E_HR * E_SJ Satisfies phi1;
96
97  Check E_HR * E_SJ * E_PH Satisfies phi1;
98
99  Check E_HR * E_SJ * E_PH Satisfies phi2;
100
101 Check E_HR * E_SJ * E_PH Satisfies phi3;
102
103 // Check E_HR * E_SJ * E_PH Satisfies phi4; //takes too long to verify
104
105 // Check E_HR * E_SJ * E_PH Satisfies phi5; //takes too long to verify
```

A.3.1 Output

```
1  Checking whether the transition system E_HR ∧ E_SJ satisfies the formula Farrest
2  Does not satisfy!
3  Counter example:
4  Path:
5  Time Order:
6          six_pm < evening ≅ dusk
7  Trust Relations:
8  --------------------------
9  Instant: 0
10 State: <<s0, s0>, START>
11 Local Interpretation Structure:
12 Event: START
13 Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)),
14 --------------------------
15
16 <<<START OF CYCLE>>>
17
18 --------------------------
19 Instant: 1
20 State: <<s1, s0>, meh>
```

```
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)),
--------------------------
Instant: 2
State: <<s2, s0>, meh>
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)),
--------------------------

<<<END OF CYCLE>>>


Checking whether the transition system E_HR ∧ E_SJ ∧ E_PH satisfies the formula Farrest
Does not satisfy!
Counter example:
Path:
Time Order:
        evening ≅ six_pm ≅ dusk
Trust Relations:
--------------------------
Instant: 0
State: <<<s0, s0>, s0>, START>
Local Interpretation Structure:
Event: START
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------

<<<START OF CYCLE>>>

--------------------------
Instant: 1
State: <<<s1, s0>, s0>, meh>
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------
Instant: 2
State: <<<s2, s0>, s0>, meh>
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------

<<<END OF CYCLE>>>
```

```
67
68
69  Checking whether the transition system E_HR ∧ E_SJ ∧ E_PH satisfies the formula (GFcrime
        => Farrest)
70  Does not satisfy!
71  Counter example:
72  Path:
73  Time Order:
74          evening ≅ six_pm ≅ dusk
75  Trust Relations:
76  --------------------------
77  Instant: 0
78  State: <<<s0, s0>, s0>, START>
79  Local Interpretation Structure:
80  Event: START
81  Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
        HR_guilty)),
82  --------------------------
83
84  <<<START OF CYCLE>>>
85
86  --------------------------
87  Instant: 1
88  State: <<<s3, s1>, s0>, crime>
89  Local Interpretation Structure:
90  Event: crime
91  Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
        HR_guilty)),
92  --------------------------
93  Instant: 2
94  State: <<<s4, s2>, s1>, meh>
95  Local Interpretation Structure:
96  Event: meh
97  Claims: (SJ : -(evening · SJ_guilty)), (SJ : -(six_pm · SJ_guilty)), (HR : (dusk ·
        HR_guilty)),
98  --------------------------
99  Instant: 3
100 State: <<<s3, s3>, s0>, crime>
101 Local Interpretation Structure:
102 Event: crime
103 Claims: (HR : -(dusk · HR_guilty)), (HR : -(six_pm · HR_guilty)), (SJ : (evening ·
        SJ_guilty)),
104 --------------------------
105 Instant: 4
106 State: <<<s4, s0>, s1>, meh>
107 Local Interpretation Structure:
108 Event: meh
109 Claims: (SJ : -(evening · SJ_guilty)), (SJ : -(six_pm · SJ_guilty)), (HR : (dusk ·
        HR_guilty)),
```

```
--------------------------

<<<END OF CYCLE>>>


Checking whether the transition system E_HR ∧ E_SJ ∧ E_PH satisfies the formula (GFcrime
    => (GF(HR : (six_pm · HR_guilty)) ∧ GF(SJ : (six_pm · SJ_guilty))))
Does not satisfy!
Counter example:
Path:
Time Order:
        evening ≅ six_pm ≅ dusk
Trust Relations:
--------------------------
Instant: 0
State: <<<s0, s0>, s0>, START>
Local Interpretation Structure:
Event: START
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------

<<<START OF CYCLE>>>

--------------------------
Instant: 1
State: <<<s1, s0>, s0>, meh>
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------
Instant: 2
State: <<<s2, s0>, s0>, meh>
Local Interpretation Structure:
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------
Instant: 3
State: <<<s3, s1>, s0>, crime>
Local Interpretation Structure:
Event: crime
Claims: (SJ : -(evening · SJ_guilty)), (HR : -(dusk · HR_guilty)), (HR : -(six_pm ·
    HR_guilty)),
--------------------------
Instant: 4
State: <<<s4, s2>, s1>, meh>
Local Interpretation Structure:
```

```
Event: meh
Claims: (SJ : -(evening · SJ_guilty)), (SJ : -(six_pm · SJ_guilty)), (HR : (dusk ·
    HR_guilty)),
--------------------------

<<<END OF CYCLE>>>
```

Appendix B

Lexer and Grammar specification

B.1 Formula Parser

B.1.1 Lexer Specification

```
1  package parser;
2  import java_cup.runtime.*;
3
4  import java.io.IOException;
5  import java.io.StringReader;
6
7  %%
8
9  %class Scanner
10 %public
11 %unicode
12 %cup
13
14 %{
15   void print(String s){
16     System.out.println(s);
17   }
18
19   private Symbol sym(int sym) {
20     return new Symbol(sym);
21   }
22
23   private Symbol sym(int sym, Object val) {
24     return new Symbol(sym, val);
25   }
26
27    public static void main(String[] args) {
28   String s = "";
29
30     Scanner scanner = new Scanner(new StringReader(s));
31     while ( !scanner.zzAtEOF )
32     try {
33       scanner.next_token();
34     } catch (IOException e) {
35       e.printStackTrace();
36     }
37   }
38 %}
39
40 WhiteSpace     = [ \t\f\r\n]
41
42
43 %%
44 /*comment*/
45
46 "/*" ~"*/"             {/*ignore, comments don't nest*/}
47
```

```
/* keywords */

"-" | "minus"          {return sym(sym.MINUS);}
"." | "·" | "dot"       {return sym(sym.DOT);}
"timelt" | "<"          {return sym(sym.LT);}
"agentlt" | "◁"         {return sym(sym.AGENTLT);}
"=" | "≃"            {return sym(sym.EQ);}
":"                {return sym(sym.COLON);}
"square" | "boxdot" | "⊡"   {return sym(sym.SQUARE);}
"not" | "¬"            {return sym(sym.NOT);}
"iff" | "<=>"          {return sym(sym.IFF);}
"implies" | "=>"       {return sym(sym.IMPLIES);}
"and" | "∧" | "&&"       {return sym(sym.AND);}
"or" | "∨"  | "||"       {return sym(sym.OR);}
"true" | "⊤"           {return sym(sym.TRUE);}
"false" | "⊥"          {return sym(sym.FALSE);}
"always" | "G"          {return sym(sym.ALWAYS);}
"eventually" | "F"       {return sym(sym.EVENTUALLY);}
"next" | "X"           {return sym(sym.NEXT);}
"until" | "U"          {return sym(sym.UNTIL);}
"(" | "{"            {return sym(sym.LPAREN);}
")" | "}"            {return sym(sym.RPAREN);}
"["                {return sym(sym.LBRACK);}
"]"                {return sym(sym.RBRACK);}


[[:jletter:]--[XGFU]] [:jletterdigit:]*       {return sym(sym.IDENTIFIER, yytext()); /*identifiers mustn't start with a letter which represents a time
      operator*/}
[\"\'][[:jletter:]] [:jletterdigit:]* [\"\']    {return sym(sym.IDENTIFIER, yytext());}

{WhiteSpace}           {/*ignore*/}
[^]                { throw new Error("Lexical error: " + yytext()); }
```

B.1.2 Grammar Specification

```
package parser;

import java_cup.runtime.*;

import formulas.*;
import formulas.atemporal.*;
import formulas.LTL.*;
import formulas.propositional.*;
import symbols.*;


/* Terminals (tokens returned by the scanner). */
terminal       MINUS, DOT, LT, AGENTLT, EQ, COLON, SQUARE;
terminal       NOT, IFF, IMPLIES, AND, OR, TRUE, FALSE;
terminal       ALWAYS, EVENTUALLY, NEXT, UNTIL;
terminal       LPAREN, RPAREN, LBRACK, RBRACK;
terminal String    IDENTIFIER;

/* Non terminals */
non terminal Formula       formula;
non terminal Formula       atempform; /* this has to be of type formula, as it might include chains (with And)*/
non terminal Formula       timeChain;
non terminal Formula       trustChain;
non terminal Claim         claim;

/* precedence */
precedence left ALWAYS, EVENTUALLY, NEXT, UNTIL;
precedence left IFF, IMPLIES;
precedence left AND, OR;
precedence left NOT;


/* The grammar rules */
start with formula;

formula ::= LPAREN formula:f RPAREN        {: RESULT = f; :}
    | TRUE                 {: RESULT = new True(); :}
    | FALSE                {: RESULT = new False(); :}
    | IDENTIFIER:id            {: RESULT = new Event(new EventSymbol(id)); :}
```

```
    | atempform:af              {: RESULT = af; :}
    | NOT formula:f             {: RESULT = f.negate(); :}
    | ALWAYS formula:f            {: RESULT = new Always(f); :}
    | EVENTUALLY formula:f          {: RESULT = new Eventually(f); :}
    | NEXT formula:f              {: RESULT = new Next(f); :}
    | formula:f1 IFF formula:f2       {: RESULT = new Equivalent(f1, f2); :}
    | formula:f1 IMPLIES formula:f2     {: RESULT = new Implies(f1, f2); :}
    | formula:f1 AND formula:f2       {: RESULT = new And(f1, f2); :}
    | formula:f1 OR formula:f2        {: RESULT = new Or(f1, f2); :}
    | formula:f1 UNTIL formula:f2       {: RESULT = new Until(f1, f2); :}
    ;

atempform ::= claim:c                               {: RESULT = c; :}
      | IDENTIFIER:id COLON claim:c                     {: RESULT = new AgentClaim(new AgentSymbol(id), c); :}
      | IDENTIFIER:id COLON LPAREN claim:c RPAREN              {: RESULT = new AgentClaim(new AgentSymbol(id), c); :}
      | IDENTIFIER:id COLON SQUARE claim:c                  {: RESULT = new MostTrustworthy(new AgentSymbol(id), c); :}
      | IDENTIFIER:id COLON SQUARE LPAREN claim:c RPAREN           {: RESULT = new MostTrustworthy(new AgentSymbol(id), c); :}
      | timeChain:tc                            {: RESULT = tc; :}
      | trustChain:tc                           {: RESULT = tc; :}
      ;

trustChain  ::= IDENTIFIER:id1 AGENTLT LBRACK IDENTIFIER:id2 RBRACK IDENTIFIER:id3  {: RESULT = new LessTrustworthy(new AgentSymbol(id1),
                                      new PropositionalSymbol(id2), new AgentSymbol(id3)); :}
      | IDENTIFIER:id1 LT LBRACK IDENTIFIER:id2 RBRACK IDENTIFIER:id3   {: RESULT = new LessTrustworthy(new AgentSymbol(id1),
                                      new PropositionalSymbol(id2), new AgentSymbol(id3)); :}
      | trustChain:tc LT LBRACK IDENTIFIER:id1 RBRACK IDENTIFIER:id2    {:
                                    LessTrustworthy previous;
                                    if(tc instanceof And){
                                      previous = (LessTrustworthy)((And) tc).getInner2();
                                    } else {
                                      previous = (LessTrustworthy) tc;
                                    }
                                    RESULT = new And(tc , new LessTrustworthy(previous.getAgent2(),
                                      new PropositionalSymbol(id1), new AgentSymbol(id2)));
                                    :}
      | trustChain:tc AGENTLT LBRACK IDENTIFIER:id1 RBRACK IDENTIFIER:id2   {:
                                    LessTrustworthy previous;
                                    if(tc instanceof And){
                                      previous = (LessTrustworthy)((And) tc).getInner2();
                                    } else {
                                      previous = (LessTrustworthy) tc;
                                    }
                                    RESULT = new And(tc , new LessTrustworthy(previous.getAgent2(),
                                      new PropositionalSymbol(id1), new AgentSymbol(id2)));
                                    :}
      ;


timeChain ::= IDENTIFIER:id1 LT IDENTIFIER:id2  {: RESULT = new TimeLessThan(new TimeSymbol(id1), new TimeSymbol(id2)); :}
      | IDENTIFIER:id1 EQ IDENTIFIER:id2  {: RESULT = new TimeEquals(new TimeSymbol(id1), new TimeSymbol(id2)); :}
      | timeChain:tc LT IDENTIFIER:id   {:
                          BinaryTimeRelation previous;
                          if(tc instanceof And){
                            previous = (BinaryTimeRelation)((And) tc).getInner2();
                          } else {
                            previous = (BinaryTimeRelation) tc;
                          }
                          RESULT = new And(tc , new TimeLessThan(previous.getTime2(), new TimeSymbol(id)));
                        :}
      | timeChain:tc EQ IDENTIFIER:id   {:
                          BinaryTimeRelation previous;
                          if(tc instanceof And){
                            previous = (BinaryTimeRelation)((And) tc).getInner2();
                          } else {
                            previous = (BinaryTimeRelation) tc;
                          }
                          RESULT = new And(tc , new TimeEquals(previous.getTime2(), new TimeSymbol(id)));
                        :}
      ;

claim ::= IDENTIFIER:id1 DOT IDENTIFIER:id2           {: RESULT = new Claim(new TimeSymbol(id1),
                                  new PropositionalSymbol(id2)); :}
    | MINUS IDENTIFIER:id1 DOT IDENTIFIER:id2         {: RESULT = new Claim(new TimeSymbol(id1),
                                  new PropositionalSymbol(id2)).invert(); :}
    | MINUS LPAREN IDENTIFIER:id1 DOT IDENTIFIER:id2 RPAREN {: RESULT = new Claim(new TimeSymbol(id1),
                                  new PropositionalSymbol(id2)).invert(); :}
    ;
```

Appendix C

Java Code

We present only the relevant part of each file.

C.1 World.java

```
124  private static List<List<List<TimeSymbol>>> generatePartitions(List<TimeSymbol>
       timeList){
125    /*
126     * Returns all the ordered partitions of the set timeList
127     * The order of the sets of the partition is relevant but
128     * the order within each set is not (i.e. are considered the same)
129     */
130    if(timeList.size() == 1) {
131      List<TimeSymbol> L1 = new LinkedList<>();
132      L1.add(timeList.get(0));
133      List<List<TimeSymbol>> L2 = new LinkedList<>();
134      L2.add(L1);
135      List<List<List<TimeSymbol>>> L3 = new LinkedList<>();
136      L3.add(L2);
137      return L3;
138    } else {
139      List<List<List<TimeSymbol>>> res = new LinkedList<>();
140      List<TimeSymbol> previous = new LinkedList<>(timeList);
141      TimeSymbol addNow = previous.remove(previous.size() - 1);
142      for(List<List<TimeSymbol>> partition : generatePartitions(previous)) { //
       recursion!
143
144        //adding addNow to each partion not changing the number of sets
145        for(int i = 0 ; i < partition.size(); i++) {
146          List<List<TimeSymbol>> newPartition = deepCopy(partition);
```

```
147             newPartition.get(i).add(addNow);
148             res.add(newPartition);
149           }
150
151           //adding another set to the partition (it must be inserted in every index
      possible)
152           for(int index = 0 ; index < partition.size() + 1; index++) {
153             List<List<TimeSymbol>> newPartition = deepCopy(partition);
154             List<TimeSymbol> newSet = new LinkedList<>();
155             newSet.add(addNow);
156             newPartition.add(index, newSet);
157             res.add(newPartition);
158           }
159         }
160         return res;
161       }
162     }
```

C.2 Sat.java

```
136   public static BoolModelPair<PeriodicIntStruc> sat(Formula phi) {
137     /*
138      * Tries to find a periodic int struc that satisfies phi
139      * by following the tree-like structure of fig 3 (p.20) in
140      * a BFS style algorithm until we find a cycle of compatible deltas.
141      * From these we extract a periodic int struc that satisfies phi.
142      */
143     Deque<List<SetFormulas>> queue = new LinkedList<>(); //rename this to stack if
      you want DFS
144     //each element of the stack is a path. This is done because we always need the
      whole path
145     SetFormulas omega = new SetFormulas();
146     omega.add(phi);
147     omega.add(World.timeChains(Formula.sigma.getTimeSymbols()));
148
149     boolean found = false;
150     for(SetFormulas delta : omega.star()) {
151       if(!locallyIncompatibleQ(delta)) {
152         List<SetFormulas> l = new LinkedList<SetFormulas>();
153         l.add(delta);
154         queue.addLast(l); //replace by addFirst if you want DFS
```

```
155       }
156     }
157
158     List<SetFormulas> path = null;
159     int periodStart = 0;
160     while(!found && !queue.isEmpty()) {
161       path = queue.removeFirst();
162       SetFormulas last = path.get(path.size() - 1);
163       for(SetFormulas delta : nextStep(last).star()) {
164         if(!locallyIncompatibleQ(delta)) {
165           if(path.contains(delta)) {
166             //found potential cycle
167             periodStart = path.indexOf(delta);
168             //Checking for liveness conditions
169             /*
170              * Important note:
171              * The liveness conditions are important because World(F phi) = {{phi},
    {XF phi}}
172              * and so we can have a chain F phi -> F phi -> F phi -> ...
173              * which would be considered sat, but it would never satisfy phi.
    However, if such a
174              * chain exists, then F phi must be present in every delta of the period
    . That is,
175              * to extract the liveness conditions, it is enough to look at any delta
     in the period
176              * (in periodStart for example). The condition must be met at some point
     in the period.
177              */
178             SetFormulas requirements = new SetFormulas();
179             //adding requirements for period
180             for(Until until : until Formulas(path.get(periodStart))) {
181               requirements.add(until.getInner2());
182             }
183             for(Eventually eventually : eventuallyFormulas(path.get(periodStart))) {
184               requirements.add(eventually.getInner());
185             }
186             for(Always always : notAlwaysFormulas(path.get(periodStart))) {
187               requirements.add(always.getInner().clone().negate());
188             }
189
190             //removing requirements in period (including those added in prefix)
191             SetFormulas toRemove = new SetFormulas();
```

```
192             for(Formula beta : requirements) {
193               for(int i = periodStart; i < path.size(); i++) {
194                 if(path.get(i).contains(beta)) toRemove.add(beta);
195               }
196             }
197             /*
198              * We keep expanding this path if the number of requirements is strictly
    bigger
199              * than the number of blocks after the first appearance of delta.
200              * The number of blocks is equal to the number of times delta has
    appeared in delta.
201              * Intuitively, if you have 1 requirement you have 1 block to deal with
    it.
202              * If you could not do it in 1 block, there is no point in continuing
    this path
203              */
204             if(requirements.size() > Collections.frequency(path, delta)) {
205               List<SetFormulas> l = new LinkedList<>(path);
206               l.add(delta);
207               queue.addLast(l); //replace by addFirst if you want DFS
208             }
209
210             requirements.removeAll(toRemove);
211
212             //IF requirements is empty then we've found a satisfying path
213             found = requirements.isEmpty();
214           } else {
215             //if a cycle cannot be formed we keep expanding the path
216             List<SetFormulas> l = new LinkedList<>(path);
217             l.add(delta);
218             queue.addLast(l); //replace by addFirst if you want DFS
219           }
220         }
221       }
222     }
223     //At this point, IF we found a satisfying path, it is stored in "path" and "
    periodStart"
224     BoolModelPair<PeriodicIntStruc> o;
225     if(found) {
226       o = new BoolModelPair<PeriodicIntStruc>(true, new PeriodicIntStruc());
227       /*Building time and trust orders:
228        *They are fixed along time, but we might only infer them at the end
```

```
229         *So we pick the last delta in the path to infer the orders
230         */
231       SetFormulas delta = path.get(path.size() - 1);
232       for(Formula psi : delta) {
233         if(!psi.isNegation()) {
234           if(psi instanceof LessTrustworthy) {
235             LessTrustworthy lessTrustworthy = (LessTrustworthy) psi;
236             o.model.addTrustRelation(lessTrustworthy.getAgent1(), lessTrustworthy.
    getProp(), lessTrustworthy.getAgent2());
237           } else if(psi instanceof TimeLessThan) {
238             TimeLessThan timeLessThan = (TimeLessThan) psi;
239             o.model.addTimeLessThanRelation(timeLessThan.getTime1(), timeLessThan.
    getTime2());
240           } else if(psi instanceof TimeEquals) {
241             TimeEquals timeEquals = (TimeEquals) psi;
242             o.model.addTimeEqualsRelation(timeEquals.getTime1(), timeEquals.getTime2
    ());
243           }
244         }
245       }
246
247       //Building the prefix and period
248       for(int i = 0; i < periodStart; i++) {
249         LocalIntStruc l = localIntStrucBuilder(path.get(i));
250         o.model.addPrefixInstant(l);
251       }
252       for(int i = periodStart; i < path.size(); i++) {
253         LocalIntStruc l = localIntStrucBuilder(path.get(i));
254         o.model.addPeriodInstant(l);
255       }
256     } else {
257       o = new BoolModelPair<PeriodicIntStruc>(false, null);
258     }
259     return o;
260   }
```