AAPO ALASUUTARI

PROCESS CONTROL USING PARACONSISTENT LOGIC PROGRAMMING

Master of Science thesis

Examiner: Prof. Esko Turunen
Examiner and topic approved by the
Faculty Council of the Faculty of
Science and Engineering
on 26th September 2018

ABSTRACT

AAPO ALASUUTARI: Process Control using Paraconsistent Logic Programming
Tampere University of Technology
Master of Science thesis, 71 pages, 1 Appendix page
October 2018
Master's Degree Program in Science and Engineering
Major: Mathematics
Examiner: Prof. Esko Turunen
Keywords: paraconsistent, logic programming, process control, finite state automata

In this Master's thesis the author's aim is to give model building tools for the temporal paraconsistent logic program called Before-After Extended Vector Annotated Logic Program with Strong Negation (bf-EVALPSN), originally created by Professor Kazumi Nakamatsu of the Prefectural University of Hyogo, Japan. The tools bridge the paraconsistent logic program language with finite state automata.

TIIVISTELMÄ

AAPO ALASUUTARI: Prosessihallinta parakonsistentilla logiikalla
Tampereen teknillinen yliopisto
Diplomityö, 71 sivua, 1 liitesivu
Lokakuu 2018
Teknis-luonnontieteellinen koulutusohjelma
Pääaine: Matematiikka
Tarkastajat: Prof. Esko Turunen
Avainsanat: parakonsistentti, logiikkaohjelmointi, prosessiohjaus, tilakone

Tämän diplomityön tavoite on luoda käsitteelliset ja teoreettiset työvälineet Japanin Hyogon prefektuurin yliopistossa, professori Kazumi Nakamatsun kehittämän Before-After Extended Vector Annotated Logic Program with Strong Negation (bf-EVALPSN, Ennen-jälkeen, laajennetun vektoriannotaation, vahvalla negaatiolla varustettu logiikkaohjelma) -logiikkaohjelmointikielen mallien rakentamiseen. Nämä työvälineet yhdistävät kielen äärellisten tilakoneiden teoriaan.

PREFACE

This thesis has been a long time coming. The preparatory work began in 2013 under the tutelage and expert guidance of Professor Kazumi Nakamatsu at the Prefectural University of Hyogo.

Further work was undertaken under on-and-off during the next few years under Professor Esko Turunen at the Tampere University of Technology, then again under Prof. Nakamatsu in 2016. It is clear that this thesis would never have seen the light of day if it weren't for the two. My deepest gratitude as well as my apologies for taking so long in finishing what I started. As it is said: "The slower it drops."

Further thanks belong to my friends and family. If it were not for you, I would not be writing this. Finally I wish to thank Valmet Automation for giving me the final push that sent me on a course to complete my degree.

Tampere, 10.10.2018

Aapo Alasuutari

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ALP	Annotated Logic Program
bf-EVALPSN	Before-After Extended Vector Annotated Logic Program with Strong Negation
GHC	generalised Horn clause
GHP	generalised Horn program
SLD resolution	Selective Linear Definite clause resolution

The abbreviations and symbols used in the thesis are collected into a list in alphabetical order. In addition, they must be explained upon first usage in the text.

1. INTRODUCTION

Computer science and programming are young fields that, although originating from algorithmic problem solving methods that span thousands of years into the early history of human civilization, only began to fully take form in the latter half of the 20th century. The advent of high-level programming languages has brought forth a multitude of different paradigms to programming while at the same time obscuring most of the actual operations done by a computer in order to perform some action. Thus it has become less and less important for a programmer to know any logic or know intimately about the inner workings of the machine he works with. Moreover, computers may seem to many to be mystical boxes where a single command invokes complex actions as if by magic.

The field of logic programming was formed in the 1970s from research efforts into theory proving, question-answer systems, and artifical intelligence and given form by the drive to bring the field of computation into the house of logic. That is to say that computation can be modeled by logic and by extension programming should be reasoned about using logic. Further logic programming attempts to unify different fields of computing such as programming, databases, and artificial intelligence by finding a logical system general enough to describe all fields of computing. Concurrent computation has been shown to not modelable using Turing machines and thus they exceed the expressivity of traditional logic programming, but this setback has simply brought about the research of actor and multi-actor models to expand the field.

Modern logic programming is based upon implication clauses called Horn clauses. These clauses are used to show deductive rules that form the basis of a given program, and consequently a set of clauses is called a Horn program. A clause is formed by a head and a body, connected with a logical implication from the body to the head. The body of a clause is formed by conjoining multiple logical statements whereas the head of a clause must contain at most one statement.

If the body of a clause is empty, then the clause is read as stating a fact. A clause with an empty head is read as a query, or a statement requirnaking proof or refutation by the system. In this thesis we will not be answering queries or deriving proofs for clauses from some program and thus will not be seeing any query clauses.

Instead, this thesis focuses on program model creation. A model of a program is a set of statements for which all rules of the program are fulfilled. Various different model semantics exists, including stable model semantics, well-founded model semantics and least fixpoint model semantics. We will use the stable model semantics and, when required, perfect model semantics to deal with negation as failure.

The purpose of model creation is to essentially define the answer set of a program and thus program model creation, specifically stable model semantics, forms the basis of a form of declarative programming called answer set programming. Given a program, the stable model or models generated from it correspond with the answer or answers to the program. This can be used to f.ex. solve difficult search problems. Program models are also but a small step away away from formal automata and the resulting models can indeed be used to generate automaton designs to solve the original program.

The logic programming language that we will use in this thesis is the Before-After Extended Vector Annotated Logic Program with Strong Negation (abbr. bf-EVALPSN). It was developed by Professor Kazumi Nakamatsu of the Prefectural University of Hyogo, Japan [7] and is the latest extension in a long string of logic programming languages building on each other, starting with Annotated Logic Program (abbr. ALP). It has been used in multiple engineering challenges such as factory pipe control, program livelock and deadlock checking and e-business modeling. [5] [6]

Bf-EVALPSN, as the name gives a hint of, is a rather complex and involved logic programming language first extends classical logic programming languages with paraconsistent logic. This means that the language accepts not only true and false attributions of truth value but also unknown and inconsistent or contradictory attributions as well. Furthermore, the language is extended with deontic attributions, giving it the power to talk not only of facts but also of allowance and forbiddance as well as paraconsistent combinations or even lack of these. Finally, the latest addition of before-after semantics defines a new functional predicate that allows the language to describe the temporal relations of ground level atoms. Usually temporal logics describe temporal relations with regards to possible or inevitable futures and only on a granularity of ”next” or ”eventually”. Bf-EVALPSN takes a different approach, describing time as a running integer and temporal relations as possibly overlapping lengths of time defined by the start and end time of the atom in question.

The focal point presented in this thesis to bf-EVALPSN is the perfect model semantics for bf-EVALPSN. In Nakamatsu's previous works bf-EVALPSN has been adapted to on-line process control but the introduction of model creation into bf-EVALPSN's toolbox allows the language to be used for off-line process control verification as well.

2. LOGIC PROGRAMMING

Logic programming is a programming paradigm based upon formal logic. Although its written history begins in the 1970s, the seeds for it were sown beginning in the 1930s and culminating in the 1960s debate between declarative and procedural representations of knowledge in Artificial Intelligence research. The procedural representation of knowledge interprets clauses in a logic program as a procedure, that is that in order to solve the resultant or head of the clause, the program must solve all the items in the body of the clause. The declarative approach instead views the clause as an implication, that is to say that if all the items in the body of a clause are shown, then the head of the clause is also shown. The difference is that a declarative interpretation does not dictate how the program should be solved, only giving rules that must be fulfilled at the end, where as the procedural interpretation gives only procedures or steps that the program can use.

In the 1970s actual logic computer programming languages began to appear. The first in the procedural camp was Planner[3]. On the declarative camp this triggered work that eventually lead to the creation of the programming language Prolog. An important step on this journey was the development of the SLD resolution inference rule by Robert Kowalski[4]. It was based upon what was already at the time an established basis of logic programming; Horn clauses.

A Horn clause is rule-like logical formula made up of a disjunction of literals with at most one positive literal and any number of negated literals.

$$\neg p \lor \neg q \lor \ldots \lor \neg t \lor u \tag{2.1}$$

This can also be interpreted and written as an implication, which is by tradition written in reverse order:

$$u \leftarrow p \land q \land \ldots \land t \tag{2.2}$$

In this format the literal at the head of the implication arrow is called the head of the Horn clause, while the conjunction of literals on the right side of the arrow are called the body. If the body of a clause is empty then the clause is called a fact statement and if the head of the clause is empty then it is called a goal clause or query. A Horn clause is definite if it is not a goal clause.

A Horn program is then a set of definite Horn clauses. A Horn program defines the knowledge of a program, or alternatively the procedures the program can undertake to answer a query. In addition to this both a computation mechanism and a control is needed. The control of a logic program is, as the name suggests, used to control the steps the program takes in searching for an answer to a given query. It could be, for instance, a depth- or breadth-first search of the query clause's members or something else entirely. The computation mechanism of logic programming is a logical inference system called SLD resolution.

SLD resolution is an inference rule for unifying disparate Horn clauses. The basis of it is in resolution inference rules. A resolution inference rule is a valid inference rule where two clauses containing complementary literals produce a new clause. As an example *Modus ponens* can be interpreted as a resolution, as

$$\frac{\neg p \vee q, p}{q} \tag{2.3}$$

does readily have two clauses with complementary literals, $\neg p \vee q$ and p, and produces a new clause q. SLD resolution works in the similar vein. Given a goal clause

$$\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n \tag{2.4}$$

and an input clause

$$A \leftarrow B_1 \wedge \ldots \wedge B_m, \tag{2.5}$$

where A_1 and A are unifiable with the substitution θ, SLD resolution produces the new clause

$$(\leftarrow B_1 \wedge \ldots B_m \wedge A_2 \wedge \ldots \wedge A_n)\theta. \tag{2.6}$$

A resolution proof is then a linear sequence of SLD resolutions where each subsequent resolution uses the previous clause as one of its parent clauses. If the resolution proof ends with an empty clause, it means that all of the initial literals of the query have been solved. This is called a refutation as it refutes the query clause in the sense that a query with an empty body clause can be interpreted as being an implication from truth ($\bigwedge B$ where B is the empty set) to falsehood ($\bigvee A$ where A is the empty set). However, in logic programming the linear sequence of SLD resolutions leading to an empty clause is interpreted as an answer to the query as it lists the unifications under which the body of the query resolves to true. This can then be used to answer eg. questions of variable binding for the given query to be true.

2.1 Semantics of logic programs

The semantics of a Horn program are next defined. To this end let us presume that we have defined for us the language L used in our program. This means that we can speak of well-formed formulas in the chosen language. Then we can define the Herbrand interpretation of a logic program P.

Definition 1 (Herbrand interpretation)**.** Assume P is a logic program. The *Herbrand universe* H of P is the set of all well-formed formulas constructed from the constants and function symbols in L.

The *Herbrand base* $\tilde{H}$ of P is then the set of all ground atomic formulas of the form $R(t_1, \ldots, t_n)$ where R is an n-ary relation symbol in L and t_i are elements of the Herbrand universe.

A *Herbrand interpretation* of P is any subset $I \subseteq \tilde{H}$.

A Herbrand interpretation can be read of as denoting a set of ground atoms that are true in the interpretation. The truth of a formula is then formally defined as follows.

Definition 2 (Truth in a Herbrand interpretation)**.** Assume that I is a Herbrand interpretation for a logic program P. We denote the *truth* of a formula F in I by $I \vDash F$, or otherwise worded that I satisfies F. The *truth* of a formula is defined as follows: as follows:

1. a ground atomic formula $R(t_1, \ldots, t_n)$ is true in I iff $R(t_1, \ldots, t_n) \in I$;
2. a ground atomic negative formula $\neg R(t_1, \ldots, t_n)$ is true in I iff $R(t_1, \ldots, t_n) \notin I$;
3. a ground conjunction $L_1 \wedge \ldots \wedge L_n$ is true in I iff $\forall i \in [1, n] : L_i \in I$;
4. a ground Horn clause $A \leftarrow B_1 \wedge \ldots \wedge B_n$ is true in I iff $I \nvDash B_1 \wedge \ldots \wedge B_n$, or $I \vDash A$;

5. a generalised Horn clause $A \Leftarrow B_1 \wedge \ldots \wedge B_n$ is true in I iff every ground instance, also known as closed instance, $A\theta \leftarrow B_1\theta \wedge \ldots B_n\theta$, where θ is a substitution from the set of variables is true in I

Now that we can speak of truth with regards to an interpretation of a program, we may begin to talk of models. A model of a logic program is an interpretation that satisfies all clauses of the program. Thus it is, in essence, a subset of the program's Herbrand base $\tilde{H}$ such that for each fact statement in P the head of the clause is in the model and for each Horn clause in P with a non-empty body, if the body clause is satisfied by the model then the head must be in the model as well.

The question however is, how can a model be constructed for a given program P? And moreover, we can see no guarantee that there is only a single model for any given program. So what kind of models are we interested in, if there are multiple to choose from? The previous paragraph holds a clue to both those questions. First lets regard a model in which there exists a formula that is neither the head of a fact statement, nor the head of a clause otherwise satisfied by the model. Let us even presume that this formula does not appear in any bodies of Horn clauses satisfied by the model. (This last part is unnecessary but makes the logic here easier, ignoring a recursion into the argument.) If we now remove this formula from the model, we are left with another model that is strictly smaller than the original but should still definitely be a model.

This is because, as defined, the formula we removed did not appear as a fact in the program, nor did it appear as the head of a true Horn clause, so removing the formula from the model did not invalidate any immediately obvious requirements for a model. Furthermore, as we defined that this particular formula did not appear in any body of a satisfied Horn clause either, removing this formula from the model does not unsatisfy any previously satisfied Horn clauses either. We can thus be sure that all Horn clauses in P are still satisfied by this new, strictly smaller model and the removed formula can thus be regarded as extraneous, perhaps even superfluous to our model. It comes clear that our models should be as small as possible to avoid any unnecessary bloat seeping in. This is what is called the least model.

Definition 3 (Herbrand model)**.** A Herbrand interpretation for a logic prgram P is a *Herbrand model* if for all clauses F in P, $I \vDash F$. If P has a Herbrand model, it is called consistent.

Definition 4 (Least model)**.** Assume a logic program P is consistent and that $M(P)$ is the set of Herbrand models of P. Then $\bigcap_{I \in M(P)} I$ is a Herbrand model of P, and more specifically it is the *least model* of P.

Knowing how to define the model we want, that is the least model, does not help with creating one, though. The intersection of all models is one more clue towards the method of creating it, though. If by narrowing down from the set of all models we can arrive at the least model, then perhaps building up using the definition of truth in a Herbrand interpretation we can arrive at a Herbrand model that is the least model.

Presume we start with an empty set of formulas as our Herbrand interpretation I for a given program P and we aim to constructively create a model for the program. The first step is obvious; for any fact statement in P, we must include the head of that clause in our interpretation. Now we have an interpretation that satisfies all the facts in P. Looking at the other Horn clauses in P now, there may be ground clauses for which all the formulas in the body of the clause are satisfiable with the set of facts alone. It becomes evident that the heads of these clauses must be included into our interpretation as otherwise the clause would run afoul of Definition 2.4. For generalised Horn clauses we must determine all ground instances of the clauses and interpret their truth using the same method as for the ground clauses above.

Adding formulas to the interpretation like this, we work our way up towards an interpretation that satisfies each and every Horn clause in the program P and only just. This method cannot, after all, add any formulas into the interpretation without them being explicitly derived from the definition of truth in a Herbrand interpretation and the clauses of the given program. No extraneous formulas can appear and as such, if the program is consistent, we should arrive necessarily at the least model, or possibly a least model, of the program. This generation of a next interpretation from a previous interpretation is called the immediate consequence operator T_P.

Definition 5 (Immediate consequence operator)**.** Let P be a logic program and I be a Herbrand interpretation of it. Then the *immediate consequence operator* T_P over I is defined as

$$T_P(I) := I \cup \{A \in \tilde{H} \mid A \leftarrow B_1 \wedge \ldots \wedge B_n \text{ is a closed instance of a rule in } P \text{ and } B_1, \ldots, B_n \in I\}$$

As Herbrand interpretations are just sets, we can define a partial order between the interpretations of a program through set inclusion.

$$I_1 \leq I_2 \Leftrightarrow I_1 \subseteq I_2 \tag{2.7}$$

With this definition we can ascertain that for two interpretations I_1, I_2, $I_1 \leq I_2$ it must be that $T_P(I_1) \leq T_P(I_2)$, as adding more items to the set I_1 to get I_2 can only cause more Horn clauses to be satisfied and thus further enlarge $T_P(I_2)$ in comparison to $T_P(I_1)$. [1] This property of the function is called monotonicity and it is the basis of finding the least model of the program P using least fixpoint semantics.

x is a pre-fixpoint of a function $f : I \to I$, where I is a partially ordered set (poset) under the ordering $\leq$, if and only if

$$f(x) \leq x, \tag{2.8}$$

x is a post-fixpoint of f if and only if

$$x \leq f(x), \tag{2.9}$$

and finally x is a fixpoint of f if and only if

$$f(x) = x. \tag{2.10}$$

Let us now show that the pre-fixpoints of T_P are models of P.

Theorem 1. *I is a model of the program P iff $T_P(I) \leq I$.*
Proof: $T_P(I) \leq I$ if and only if for all $A \in \tilde{H}$

$$T_P(I) \vDash A \Rightarrow I \vDash A.$$

Using Definition 5 the left side can be stated with regards to a single formula A as

$$\begin{aligned} I \vDash A \text{ or } \{A \leftarrow \mathit{Body} \text{ is a closed instance} \\ \text{of a rule in } P \text{ and } I \vDash \mathit{Body}\}. \end{aligned}$$

Combining this with the original equation, the left side of the *or* becomes a tautology and the latter half gives us

$$\{A \leftarrow \mathit{Body} \text{ is a closed instance of a rule in } P \text{ and } I \vDash \mathit{Body}\} \Rightarrow I \vDash A.$$

[1] This does not necessarily hold if the Horn clauses are non-definitive. If negated literals are allowed in the bodies of the Horn clauses, then the program must be stratified to show the monotonicity of the immediate consequence operator.

Combining this all, we have the statement that $T_P(I) \leq I$ if and only if

$$\forall A \in \tilde{H} : \; I \vDash A \text{ or } \{(A \leftarrow \mathit{Body}) \in P \text{ and } I \vDash \mathit{Body}\} \Rightarrow I \vDash A.$$

So if I is a pre-fixpoint of T_P, then necessarily for all clauses in P, if the body of the clause is satisfied by I then the head must be satisfied as well. Otherwise the head alone must be satisfied. This satisfies all clauses in P, forming a Herbrand model. Further, other formulas not in the heads of clauses may be satisfied by the model, allowing for not only the least model to fulfill this statement. □

As $M(P)$ is a set of sets over the set $\tilde{H}$, it can be viewed as a partially ordered set where all subsets have a join and meet through normal set operations $\cup$ and $\cap$. This is the definition of a complete lattice and for a complete lattice the Knaster-Tarski theorem guarantees that the least pre-fixpoint of a function coincides with its least fixpoint. This combined with Theorem 1 gives us the result that among the least pre-fixpoint of T_P is also the least fixpoint of T_P, and it is a model of P.

Theorem 2. Any consistent logic program P has a least model $\mathcal{M}_P$, which is identical to the least fixpoint of T_P.

Definition 6 (Upward iteration)**.** The *upward iteration* of T_P is

$$T_P \uparrow 0 = \emptyset, \quad T_P \uparrow \alpha = T_P(T_P \uparrow (\alpha - 1)), \quad T_P \uparrow \lambda = \cup\{T_P \uparrow \nu \mid \nu \leq \lambda\},$$

where a and ν are successor ordinals and λ is a limit ordinal.

Now we have the necessary knowledge to construct our least model.

Theorem 3. The least model of a program P is the limit of upward iteration of T_P

$$T_P \uparrow \omega = \mathcal{M}_P.$$

The least model of a program in unique for any given definite program, that is a program containing no clauses with negated formulas in the body. Negated formulas in the head of the clause are naturally not allowed as that would effectively be a query clause. If a program contains negated formulas in the body, then the search for a least model is determined by stratification of the program.

A stratified program is such a program that its clauses can be split into numbered strata, or layers. Then for each positively occurring formula F in the body of a clause, all clauses with F as the head of the clause (definition of F) occur on either equal of lower strata. For all negatively occurring formula $\neg F$ in the body of a clause, all clauses with F as the head of the clause occur on strictly lower strata. If such a stratification is possible, then a program is called stratified and it has a unique least model.

2.2 Annotated logic programs

Bf-EVALPSN has its foundations in the field of annotated logic programs. A good overview into the field is given in [1]. The basic idea of annotated logics is that formulas are bound with annotations. These annotated terms then look something like $A : \mu$. As an example of annotated logic programs, we will present the paraconsistent annotated logic $P\mathcal{T}$.

The theory along with soundness and completeness proofs for $P\mathcal{T}$ is well established in various books such as [1]. Here the main points shall be quickly re-established to show the base on which bf-EVALPSN stands, originally proposed by da Costa, et al. [2].

A truth value in $P\mathcal{T}$ is called an *annotation* and is attached to each atomic formula with the truth values constituting an arbitrary fixed finite complete lattice τ with the ordering $\leq$ and the negation operator $\sim: |\tau| \to |\tau|$. We use $\top$ and $\bot$ to denote the top and bottom elements of the lattice, respectively. In addition $\vee$ and $\wedge$ operators are defined as the least upper bound and the greatest lower bound. We shall only concern ourselves with a lattice of cardinality larger than 2, making the logic nontrivial.

Definition 7 (Symbols)**.** The symbols of $P\mathcal{T}$ are defined as follows:

1. Propositional symbols: p, q, ...
2. Annotated constants: $\mu, \lambda, \ldots \in |\tau|$
3. Logical connectives: $\wedge$, $\vee$, $\to$ and $\neg$.
4. Parentheses: (and).

Definition 8 (Formulas)**.** Formulas are defined recursively according to the following rules:

1. If A is a propositional symbol and $\mu \in \mathcal{T}$ is a an annotation, then $(A : \mu)$ is a formula called an *annotated atomic formula* or annotated atom for short.
2. If F and P are formulas, then $(\neg F)$, $(F \wedge G)$, $(F \vee G)$ and $(F \to G)$ are formulas.

Extraneous parentheses are omitted where no chance of misinterpretation is possible.

The negation $\neg$ in $P\mathcal{T}$ is an *epistemic negation*, meaning a negation of some type of truth but not the knowledge of the truth itself. This means that as long as the lattice ordering $\leq$ is likewise defined as an epistemic order (the highest element holds most knowledge, lowest the least), as is usual, then the epistemic negation does not change the order of elements.

$$\text{If } a \leq b \text{ then } \neg a \leq \neg b. \tag{2.11}$$

Another negation $\neg^*$ is defined. This negation is known as the *strong* or *ontological* negation. Ontological refers to existence and thus the ontological negation is usually used to refute the existence of the formula it is applied on. Another interpretation is to understand the ontological negation as negating the knowledge represented in an annotation.

Definition 9 (Strong Negation)**.** Let F be any formula. Then the *strong negation* of F is defined as

$$\neg^* F := F \rightarrow ((F \rightarrow F) \wedge \neg(F \rightarrow F)) \tag{2.12}$$

The strong negation can be read as the existence of F implying a contradiction.

The semantics for $P\mathcal{T}$ is defined through interpretations.

Definition 10 (Interpretation)**.** Let ν be the set of all propositional symbols, $\mathcal{T}$ be the set of all annotations and $\mathcal{F}$ be the set of all formulas. An *interpretation* I is a function

$$I : \nu \rightarrow \mathcal{T} \tag{2.13}$$

and to each interpretation I we can associate a valuation function

$$\nu_I : \mathcal{F} \rightarrow \{0, 1\}. \tag{2.14}$$

The valuation function is defined as:

1. Let A be a propositional symbol and μ an annotation. Then

$$\nu_I(A : \mu) = 1 \text{ iff } \mu \leq I(A), \text{ else } \nu_I(A : \mu) = 0,$$

$$\nu_I(\neg^k A : \mu) = \nu_I(\neg^{k-1} A : {\sim}\mu), \text{ where } k \geq 1.$$

2. Let F be any complex formula. Then

$$\nu_I(\neg F) = 1 - \nu_I(F).$$

Other formulas with logical connectives are valuated as expected.

The paraconsistent logic $P\mathcal{T}$ can be proven to be sound and complete. Additionally $P\mathcal{T}$ can be extended into a propositional paraconsistent annotated logic $Q\mathcal{T}$ which is still sound and complete. The proof for both of these along with the algebraic semantics using Curry algebras can be found in [1]. Logic programming with annotated logics is next given a definition using generalised Horn programs.

2.3 Generalised Horn Program

Paraconsistent logic programming bases on an extension of the same Horn programs introduced earlier, called generalised Horn programs (GHP). Generalised Horn programs are made out of, similarly, generalised Horn clauses. The generalisation here comes from the formulas in the clauses not being Horn clauses in the traditional sense of containing logical atoms and negations of thereof.

Instead, in generalised Horn clauses, the atoms of the clause can essentially be anything that is Horn-like. In the field of paraconsistent annotated logic programming, the Horn-like atoms are the annotated atomic formulas. These clauses will later serve as the basis of bf-EVALPSN programs. Again a complete lattice of annotations $\tau = \langle |\tau|, \leq, \sim \rangle$ is used. Formulas and the strong negation are defined as in the previous section.

Definition 11 (Generalised Horn Program)**.** A *generalised Horn clause* (GHC) is constructed with annotated literals $A : \mu$, $B_i : \mu_i$ and an implication. In the notation of logic programming an implication is denoted from right to left. The parts of a GHC

$$A : \mu \Leftarrow B_1 : \mu_1 \wedge \ldots \wedge B_n : \mu_n$$

are called the head and the body of the clause respectively, separated by the implication. A *generalised Horn program* is a finite set of GHCs.

GHCs are interpreted by a function I from the Herbrand base $\tilde{H}$ of formulas under consideration to the set of all annotations $\mathcal{T}$.

$$I : \tilde{H} \to \mathcal{T}. \tag{2.15}$$

Each formula is either satisfied or not by the interpretation I according to the following definition.

Definition 12 (Satisfaction)**.** We write $I \models F$ to say that I satisfies F. An interpretation I satisfies

1. the general formula F iff I satisfies each of its closed instances, i.e. for each variable symbol x occurring free in F and each variable term t, the replacement of occurrences of x by t, the ground formula $F(t/x)$, is satisfied by I,
2. the closed annotated atom $A : \mu$ iff $I(A) \geq \mu$,
3. the closed annotated literal $\neg A : \mu$ iff it satisfies $A : \sim\mu$,
4. the closed formula $(\exists x)F$ iff for some variable free term t, $I \models F(t/x)$,
5. the closed formula $(\forall x)F$ iff for every variable free term t, $I \models F(t/x)$,
6. the closed formula $F_1 \Leftarrow F_2$ iff $I \not\models F_2$ or $I \models F_1$,
7. the closed formula $F_1 \wedge \ldots \wedge F_n$ iff for all $i = 1, \ldots, n$, $I \models F_i$,
8. the closed formula $F_1 \vee \ldots \vee F_n$ iff for some $1 \leq i \leq n$, $I \models F_i$,
9. the closed formula $F \Leftrightarrow G$ iff $I \models F \Leftarrow G$ and $I \models G \Leftarrow F$.

An interpretation I satisfies a GHP P if it satisfies every GHC $C \in P$. Then I is called a model of P.

Definition 13 (Positive Counterpart)**.** If C is a GHC in P then the result of replacing all negated literals $\neg A : \mu$ in C by $A : \sim\mu$ is called the *positive counterpart* C^{pos} of C. The GHP P^{pos} in which all GHCs C are replaced by their respective counterparts C^{pos} is called the *positive counterpart* of P.

It is self-evident that the using this definition any epistemic negations in a GHC can be internalised without affecting the logical meaning of the clause. Thus likewise any model of P is also a model of G^{pos} and vice versa.

It is generally known that a logic program can have multiple models. With paraconsistent logic programming it is possible to use fixpoint semantics to define the least model amongst them. First an order of interpretations is defined.

Definition 14 (Order of Interpretations)**.** Given a GHP P and Herbrand interpretations I_1 and I_2 the order $\leq_I$ is defined as

$$I_1 \leq_I I_2 \text{ iff } (\forall A \in \tilde{H}) : I_1(A) \leq I_2(A) \tag{2.16}$$

where $\tilde{H}$ is the Herbrand base of P. Note that the set of interpretations forms a complete lattice.

The completeness of the lattice can be briefly shown by taking a Herbrand base $P = \{A\}$, ie. a single atom base. Now as the lattice of possible annotations is a complete lattice, the interpretation of P is exactly the same lattice as the lattice of annotations but with each point being, instead of a truth value, a mapping from A to that truth value. This lattice is then verily seen to be complete and to correspond completely with the lattice of annotations in terms of order.

Now adding another atom to the base, $P = \{A, B\}$, we can view the interpretation lattice as a point couple $[I(A), I(B)]$. Both $I(A)$ and $I(B)$ correspond with the lattice of annotations in terms of order and thus form complete lattices. The combination of these two lattices is then nothing but a Cartesian product $I(A) \times I(B)$ which is evidently a complete lattice.

The immediate consequence operator for a paraconsistent annotated logic with a complete lattice of annotations is defined. It is a monotone, recursive operator as expected and needed to establish fixpoint semantics.

Definition 15 (Immediate consequence operator T_P)**.** Suppose P is a GHP. Then T_P is a mapping from the Herbrand interpretations of P to the Herbrand interpretations of P defined by

$$T_P(I)(A) = \sqcup\{\mu \mid A : \mu \Leftarrow B_1 : \mu_1 \wedge \ldots \wedge B_n : \mu_n\}$$

where $\sqcup$ is the least upper bound in τ, the clause is a ground instance of a GHC in P and $I \vDash B_1 : \mu_1 \wedge \ldots \wedge B_n : \mu_n$.

Since interpretations form a complete lattice it is clear that the above defined T_P is indeed monotonic as intended.

As earlier, a pre-fixpoint for a function f is some x for which under some ordering $\sqsubseteq$ it holds that $f(x) \sqsubseteq x$. Similarly, as earlier it can be shown that I is a model of the GHP P if and only if I is a pre-fixpoint, ie. $T_P(I) \leq I$.

This is readily understandable from taking into account that if I is a model of P then all GHCs in P must be satisfied. Now operating on I with T_P is calculating unions of each atom's interpretations in the heads of clauses in P whose body clause is satisfied by I. This operation may leave out some unsupported annotations (annotations that do not appear at the head of satisfied clauses) from I, thereby lessening the union of annotations. This in turn may turn some interpretations of atoms from true to false but never from false to true. This then guarantees that a generalised Horn clause whose body was not satisfied will stay unsatisfied, and a Horn clause whose body was satisfied either stays unsatisfied or becomes unsatisfied, thus facilitating only the lessening of the interpretation, never its growth. As any negative atoms that might've been present in the program can (and should have been) also be removed by converting them to their positive counterparts, we can be sure that all contradictions in the logic of the program are internalised into the interpretation lattice. Thus $T_P(I)$ is still a a model of P and can be less or equal to I.

From the monotonicity and pre-fixpoint properties of T_P it can be shown that T_P has a model. Additionally because the set of interpretations is a complete lattice the least fixpoint and the least pre-fixpoint coincide, giving the following result.

Theorem 4. Any GHP P has a least model $\mathcal{M}_P$. In addition this least model is identical to the least fixpoint $lfp(T_P)$ of T_P.

Additional properties of GHPs and T_P can be proven but are not immediately relevant to this paper and are thus omitted.

If an annotated logic program admits strong negation, then any strongly negated symbols in bodies of clauses can be handled using stable model semantics similarly to that of traditional logic programs'. This means that for an interpretation I, for any strongly negated atom $\neg^* F$ present in the body of a formula for which $I \models F$, the formula is removed and any strongly negated atom from which $I \not\models F$, the strongly negated atom is removed from the body of the clause. In this case, the generation of the interpretation I must be made using stratification of the program as described in 2.1, with the strata separated by strong negation.

It is worth noting that althought the strong negation in annotated logics is defined not on annotations but as a formula, it could be devised in terms of annotations as well. This definition would, however, require changes to the way annotations are interpretated and thus would make stratification difficult and creation of the immediate consequence operator possibly impossible. Yet as far as on-line interpretation of the status of a program goes, the satisfaction of a strongly negated atom may be immediately checked by verifying that the value attributed to the annotated propositional symbol is not equal or greater than that of the annotation given to it in the strongly negated atom.

This means that it on-line verification of processes using strongly negated annotated logics offers very expressive ways to deal with both unknown data, contradictions in inputs as well as logic of restricted value. It is, for instance, possible to attribute to a propositional symbol the meaning of pressure in a system and write a verification rule that begins an emergency shutdown if the pressure is not within set bounds.

These sorts of capabilities make annotated logic programs highly desirable in a multitude of engineering solutions.

3. BF-EVALPSN

In chapter 2 the basic notions of logic programming and paraconsistent annotated logics were given. Formulas were formed from a combination of a literal and an annotation.

$$A : \mu, \tag{3.1}$$

and benefit was found in the expressiveness of the logic and its possible applications to, for example, process control. However, many processes as well as most modern programming is no longer entirely describable using only fixed relations. The need to reason about the timing of events and processes is a must in many applications. For this purpose, among others, we show the annotated logic program bf-EVALPSN, created by Nakamatsu [7], which stands for before-after Extended Vector Annotated Logic Program with Strong Negation.

In bf-EVALPSN formulas are defined recursively as before but a particular new literal is also given. This is the *bf-literal R*. The bf-literal, or before-after literal, is a functional literal taking three arguments. The first two arguments are ground literals, usually used to refer to processes, while the third is a time variable. The bf-literal

$$R(A, B, t)$$

is read: "The before-after relation between A and B at the time t." Additionally we define some types of formulas.

Definition 16 (Hyper-literal and complex formulas)**.** A formula of the form

$$\neg^k F : \mu$$

where $k \geq 0$ and F is a ground level formula is called a hyper-literal formula, or hyper-literal for short. A formula which is not a hyper-literal is a complex formula.

Annotations in bf-EVALPSN are formed from the combination of two vector annotations. The first part is the epistemic tuple (a, b) denoting positive (a) and negative knowledge (b) that we have already seen in the previous chapter. The second part is a deontic annotation. These are combined in a fashion that for every deontic annotation there exists a full epistemic tuple.

The tuple (a, b) belongs to a complete lattice $\mathcal{T}_v(n)$ of annotations with a lowest element $\perp = (0, 0)$, highest element $\top = (n, n)$ and a partial order $\leq$. Similarly the deontic annotation belongs to a complete lattice $\mathcal{T}_d$ of annotations with a set of elements

$$\perp,\ \alpha,\ \beta,\ \gamma,\ *_1,\ *_2,\ *_3,\ \top,$$

where $*_1 = \alpha \wedge \beta$, $*_2 = \beta \wedge \gamma$ and $*_3 = \alpha \wedge \gamma$. The partial order is defined naturally as seen in figure 3.1.

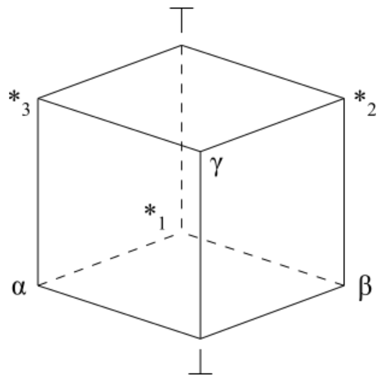


Figure 3.1 *Deontic lattice of bf-EVALPSN*

Combining these two in the above mentioned fashion we get an 8-tuple of epistemic tuples:

$$[(a_1, b_1)_\perp, (a_2, b_2)_\alpha, (a_3, b_3)_\beta, (a_4, b_4)_\gamma, (a_5, b_5)_{*_1}, (a_6, b_6)_{*_2}, (a_7, b_7)_{*_3}, (a_8, b_8)_\top] \quad (3.2)$$

This presentation of the annotations is of course very cumbersome to write and could not reasonably be used as a basis for bf-EVALPSN. Instead we use the form that comes naturally from the cartesian product formula of the two lattices

$$[\mathcal{T}_v(n) \times \mathcal{T}_d] \quad (3.3)$$

Thus a complete annotated atom of bf-EVALPSN is presented as the following:

$$A : [(a, b), \delta] \tag{3.4}$$

This presentation is reasonable given that in logic programming languages we allow only well-formed annotations to be used as annotations in clauses. As any annotation with

$$\delta \in \{\bot,\ *_1,\ *_2,\ *_3,\ \top\}$$

is not a well-formed annotation, we're left to deal with only those annotations where

$$\delta \in \{\alpha,\ \beta,\ \gamma\}$$

and combinations of these. Combinations would seem to require us to at least use a format of annotations like

$$[(a_1, b_1)_\alpha, (a_2, b_2)_\beta, (a_3, b_3)_\gamma]$$

but we regard these to not be well-formed formulas either. Instead we require these types of combinations to be split into individual annotations where only one of the three epistemic lattices contains non-zero values (or indeed contain only one non-zero value) and the atoms annotated with these individual annotations are conjoined by the logical $\wedge$. In the case where such a combined annotation would be at the head of a clause, the clause needs to be repeated for each of the individual annotations.

Finally the order of annotations is defined as

$$[(a, b), \sigma] \leq [(c, d), \theta] \Leftrightarrow (a \leq c) \wedge (b \leq d) \wedge (\sigma \leq \theta). \tag{3.5}$$

This ordering may be slightly unorthodox with regards to the underlying 8-tuple of epistemic annotations. It, however, makes intuitive sense and clearly defines a proper complete lattice. We are thus happy with the definition.

The deontic annotations are given natural language meaning as follows. α stands for epistemic fact or known truth and can be viewed as the default annotation in the sense that if deontic annotations were removed from the language, then every interpretation of an atom in the program would traditionally in be interpreted as an epistemic statement. β stands for obligation or requirement. It is normally used to state a requirement for the atom in question to receive the stated epistemic annotation immediately or very soon. Finally, γ stands for non-obligation or permission and is used to state permission for a given interpretation to change to the given epistemic annotation.

Any logical relations between the different deontic meanings is, naturally, part of the program they're used in. Bf-EVALPSN as a language itself does not force any semantics on these relations. Although it may seem useful and natural to, for example, require that an obligation for the falseness of an atom would imply that the atom becomes false it is not necessarily that such a forcing would be useful. As part, it would even remove from the deonticity of the logic by binding an obligation to the action and thus it becomes clear that such rules must be left alone.

The introduction of the deontic annotations requires us to define new epistemic negations for bf-EVALPSN. The first of these, $\neg_1$, is defined as

$$\neg_1 A : [(a, b), \delta] := A : [(b, a), \delta]. \tag{3.6}$$

Clearly this negation is an epistemic negation, as no knowledge of the statement itself is lost. The epistemic annotations are flipped, negative knowledge becomes positive and vice versa. However, the deontic annotation stays unchanged, as it does not represent any epistemic quantity of the statement but only the deontic nature of it.

The second negation is an epistemic negation of the deontic annotations. This negation negates the ontological, that is moral knowledge of the statement. The moral negation of fact is still fact, but permission becomes obligation and vice versa. The epistemic annotation of the statement stays unchanged. The negations of the deontic annotations are shown separately first.

$$\sim_2 \perp := \perp \tag{3.7}$$

$$\sim_2 \alpha := \alpha \tag{3.8}$$

$$\sim_2 \beta := \gamma \tag{3.9}$$

$$\sim_2 \gamma := \beta \tag{3.10}$$

$$\sim_2 *_1 := *_3 \tag{3.11}$$

$$\sim_2 *_2 := *_2 \tag{3.12}$$

$$\sim_2 *_3 := *_1 \tag{3.13}$$

$$\sim_2 \top := \top \tag{3.14}$$

With these, the definition of the second epistemic negation $\neg_2$ can be stated as

$$\neg_2 A : [(a, b), \delta] := A : [(a, b), \sim_2 \delta] \tag{3.15}$$

When the interpretation of bf-EVALPSN is presented later, the formulas and definitions using negation (such as the definition of strong negation and various definitions concerning the interpretation of negations) should be considered to refer to both or either of the epistemic negations presented here. For example, the negation of a complex formula can be done using either of the negations and the interpretation is the same; if the formula is not satisfied, then its negation is.

Finally we define well-formed formulas for bf-EVALPSN. It is normal in annotated logic to require that any annotations present in a program are well-formed annotations. Well-formed annotations are a subset of possible annotations that are usually constrained to show non-contradictory evidence. In our case, well-formed annotations are of the form

$$[(a, b), \delta], \text{ where either } a \text{ or } b \text{ (not both) is zero and } \delta \in \{\alpha,\ \beta\,\gamma\}, \tag{3.16}$$

and well-formed formulas are then atoms annotated with well-formed annotations or combinations of thereof.

This requirement is not essential to the language itself, and is actually egregiously broken by the before-after annotations introduced later. When dealing with normal annotated atoms, however, the requirement makes natural language reasoning of the programs simpler. It would make little sense if the program itself was written to, for example, set some ground atom's epistemic truth value to $\top$ directly. If the value becomes $\top$ or otherwise contradictory from multiple input sources declaring differing data, then the result makes sense. If, however, an atom's value is set to be contradictory based on some Horn clause, it makes for a bizarre system where the knowledge of one atom is decided to be contradictory by some completely other one.

With this system of annotations bf-EVALPSN is capable of deontic reasoning in addition to defeasilbe reasoning, and further can express temporal relations while still retaining all the features of a complete lattice based annotated logic. This means that, if not strong negations are present in the system, the least model of the program is unique and easy to construct. And even with strong negations in the program, model creation is made easy using the strong negation perfect model semantics.

3.1 Interpretation of bf-EVALPSN

With the language of bf-EVALPSN set in place, the interpretation of the language must be defined. As the language consist of vastly more complex literals than simple constants, the interpretation is likewise much more involved than the simple interpretation given for $P\mathcal{T}$ in the previous chapter.

Definition 17 (Interpretation)**.** An interpretation I for the language L of bf-EVALPSN consists of a non-empty set, denoted by $dom(I)$ and called the *domain* together with

1. a function η_I that maps constants of L to $dom(I)$
2. a function ζ_I that assings to each function symbol f or arity n in L a function from $(dom(I))^n$ to $dom(I)$
3. a function χ_I that assigns to each predicate symbol of arity n in L a function from $(dom(I))^n$ to τ.

Definition 18 (Variable assignment)**.** Suppose I is an interpretation for L. Then a *variable assignment* v for L with respect to I is a map from the set of variable symbols of L to $dom(I)$.

Definition 19 (Denotation)**.** The *denotation* $d_{I,v}(t)$ of a term t with reference to an interpretation I and variable assignment v is defined inductively as follows:

1. If t is a constant symbol then $d_{I,v}(t) = \eta(t)$.
2. If t is a variable symbol then $d_{I,v}(t) = \nu(t)$.
3. If t is a function symbol then $d_{I,v}(t) = \zeta(f)(d_{I,v}(t_1), \dots, d_{I,v}(t_n))$.

Definition 20 (Satisfaction)**.** Let I and ν be an interpretation of L and a variable assignment with reference to I, respectively. Also suppose that A is a ground literal and that F, G and H are any formulas whatsoever. Then the *satisfaction*

$$I,\, \nu \models F$$

standing for F being satisfied with regards to interpretation I and a variable assignment ν, is defined as follows:

1. $I,\, \nu \models A : \mu$ iff $d_{I,\nu}(A) \geq \mu$.
2. $I,\, \nu \models P(t_1, \dots, t_n) : \mu$ iff $\zeta_I(P)(d_{I,\nu}(t_n)) \geq \mu$.
3. $I,\, \nu \models \neg^k A : \mu$ iff $I,\, \nu \models \neg^{k-1} A : {\sim}\mu$.

4. $I,\, \nu \vDash (F \wedge G)$ iff $I,\, \nu \vDash F$ an $I,\, \nu \vDash G$.
5. $I,\, \nu \vDash (F \vee G)$ iff $I,\, \nu \vDash F$ or $I,\, \nu \vDash G$.
6. $I,\, \nu \vDash (F \rightarrow G)$ iff $I,\, \nu \nvDash F$ or $I,\, \nu \vDash G$.
7. $I,\, \nu \vDash \neg F$ iff $I,\, \nu \nvDash F$ where F is a complex formula.
8. $I,\, \nu \vDash \exists H$ iff for some variable assignment $\nu^{'}$ such that for all variables y different from x, $\nu(y) = \nu(y^{'})$ we have that $I,\, \nu^{'} \vDash H$.
9. $I,\, \nu \vDash \forall H$ iff for all variable assignment $\nu^{'}$ such that for all variables y different from x, $\nu(y) = \nu(y^{'})$ we have that $I,\, \nu^{'} \vDash H$.
10. $I \vDash H$ iff for all variable assignment ν associated with I, $I,\, \nu \vDash H$.
11. $I,\, \nu \vDash s = t$ iff $d_{I,\nu}(s) = d_{I,\nu}(t)$.

Let $\Gamma \bigcup \{H\}$ be a set of formulas. We write $\vDash H$ and say that H is *valid* if for every interpretation I, $I \vDash H$. If $I \vDash A$ for each $A \in \Gamma$ then I is a *model* of Γ. Finally H is a *semantic consequence* of Γ if and only if for any interpretation I such that $I \vDash G$ for all $G \in \Gamma$ it is the case that $I \vDash H$.

Lemma 1. For any complex formula A and B and any formula F the valuation ν satisfies the following:

1. $\vDash A \leftrightarrow B$ iff $\vDash A \rightarrow B$ and $\vDash B \rightarrow A$.
2. $\nvDash (A \rightarrow A) \wedge \neg(A \rightarrow A)$.
3. $\vDash \neg^{*} A$ iff $\nvDash A$.
4. $\vDash \neg F \leftrightarrow \neg^{*} F$.

Lemma 2. Let $A(t_1, \ldots, t_n) : \mu$ be an annotated atom and $\mu, \lambda \in |\tau|$. Then the following hold:

1. $\vDash A(t_1, \ldots, t_n) : \bot)$.
2. $\vDash A(t_1, \ldots, t_n) : \mu \rightarrow A(t_1, \ldots, t_n) : \lambda$ iff $\mu \geq \lambda$.
3. $\vDash \neg^{k} A(t_1, \ldots, t_n) : \mu \leftrightarrow \neg^{k-1} A(t_1, \ldots, t_n) : \sim\mu$ where $k \geq 0$.

3.2 Before-After Relations

The before-after relations of bf-EVALPSN apply solely to the annotated *bf-literals*, that is to say literals of the form

$$R(p_i, p_j, t) : [(a, b), \delta] \tag{3.17}$$

where p_i and p_j are ground terms and t stands for some point in time. Although the ground terms are not be anything more than basic ground terms, it is usual to speak of them as processes. This is because speaking of the bf-relations between to ground terms only makes sense if the two ground terms both have some definable temporal span of activity, when they start and when they end.

To define the start and end time of a ground term, we prescribe functions for the starting and ending of a process.

$$start(x,t),\ finish(x,t) \tag{3.18}$$

The *start* function's valuation starts as false ($[(0,n),\alpha]$ where n is the cardinality of the complete lattice) and becomes true ($[(n,0),\alpha]$) when the process x begins, whereas the $finish$ function's valuation starts as false and becomes true when the process finishes. With these two functional predicates it is possible to keep tabs on process start and end times, and thus calculate the before-after relations between processes. The time moment t when $start(x,t)$ becomes true is labeled x_s and the end time is labeled x_f for start and finish, respectively.

The number of possible temporal relations between two processes is 15, relating to their relative start and finish times. The relations are as follows:

- Before (be) / After (af)
- Disjoint Before (db) / After (da)
- Immediate Before (mb) / After (ma)
- Joint Before (jb) / After (ja)
- S-included Before (sb) / After (sa)
- Included Before (ib) / After (ia)
- F-included Before (fb) / After (fa)
- Paraconsistent Before-after (pba)

Thus the annotations of bf-literals, called *bf-annotations* form a complete bi-lattice $\mathcal{T}_v(12)_{bf}$. The lattice can be seen in figure 3.2. If further simplicity is wanted, or it is unreasonable or unnecessary to consider the possibility of two events exactly overlapping, then *sb*, *sa*, *fb*, *fa* and *pba* annotations can be left out from the annotations, leading to a complete bi-lattice $\mathcal{T}_v(7)_{bf}$.

The relations and relevant annotations are cleared next. For each relation, with the exception of the paraconsistent before-after, there are two relations that are reflections of each other. The first relation mentioned is one that places the first process x earlier in time than the second process y. Somewhat counter-intuitively the annotations are in order that the higher the "negative evidence" of the annotation,

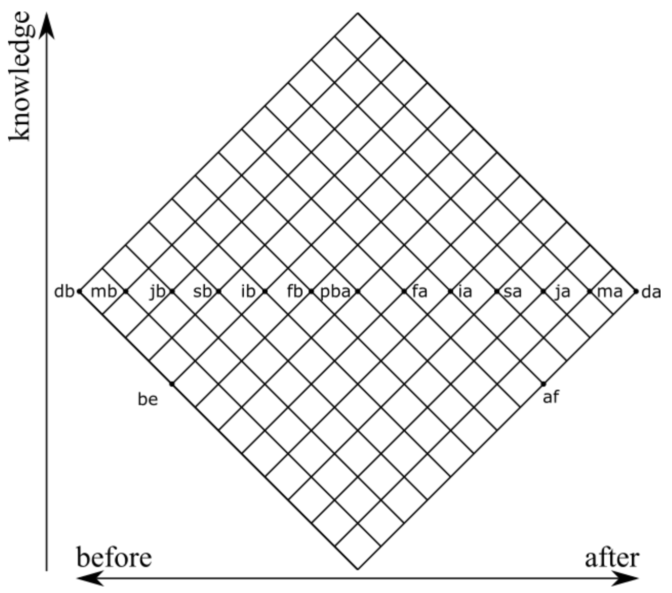


Figure 3.2 *Before-after annotations lattice* $\mathcal{T}_v(12)_{bf}$

the earlier the first process is compared to the second, and the higher the "positive evidence", the earlier the second becomes compared to the first. With the first pair, both of these annotations will be presented but later on only the relation where x is earlier than y will written out implicitly.

As mentioned earlier in this chapter, these annotations are not well-formed annotations as defined by 3.16. However, this does not pose a problem as all the annotations can be split into well-formed annotations. As an example the following (nonsense) formula

$$R(p_2, p_3, t) : [(2, 10), \alpha] \rightarrow R(p_1, p_2, t) : [(8, 4), \alpha] \tag{3.19}$$

can be split into two well-formed formulas that together are entirely equivalent to the original as shown below. Thus we choose to write out the bf-rules in their original forms knowing that in an actual implementation the rules will be converted to well-formed bf-rules.

$$R(p_2, p_3, t) : [(2, 0), \alpha] \wedge R(p_2, p_3, t) : [(0, 10), \alpha] \rightarrow R(p_1, p_2, t) : [(8, 0), \alpha] \tag{3.20}$$

$$R(p_2, p_3, t) : [(2, 0), \alpha] \wedge R(p_2, p_3, t) : [(0, 10), \alpha] \rightarrow R(p_1, p_2, t) : [(0, 4), \alpha] \tag{3.21}$$

3.2.1 Before / After

A *be* / *af* -relation states that the process x starts before (after) y. Formally means that the starting time of process x is smaller than that of y, $x_s < y_s$. As an annotation we represent this as

$$R(x, y, t) : [(0, 8), \delta] \tag{3.22}$$

for *be* or

$$R(x, y, t) : [(8, 0), \delta] \tag{3.23}$$

for *af*.

This relation does not say anything about the end times of the two processes. Thus the relation can be understood as giving imperfect knowledge.

3.2.2 Disjoint Before / After

A *db* / *da* relation means that the first process finishes before (after) the second one starts. Formally for *db* this can be stated as $x_f < y_s$ or as an annotation represented as

$$R(x, y, t) : [(0, 12), \delta]. \tag{3.24}$$

3.2.3 Immediate Before / After

An *mb* / *ma* relations means that the second process starts the moment the first finishes. Formally for *mb* this can be stated as $x_f = y_s$ or as an annotation represented as

$$R(x, y, t) : [(1, 11), \delta]. \tag{3.25}$$

3.2.4 Joint Before / After

A *jb* / *ja* relation means that the two processes overlap in time such that the first process starts and finishes first but the second process starts before the first one has finished. Formally for *jb* this can be stated as $x_s < y_s < x_f < y_f$ or as an annotation represented as

$$R(x, y, t) : [(2, 10), \delta]. \tag{3.26}$$

3.2.5 S-included Before / After

An *sb* / *sa* relation means that the first process starts before the second but they end at the same time. Formally for *sb* this can be stated as $x_s < y_s < x_f = y_f$ or as an annotation represented as

$$R(x, y, t) : [(3, 9), \delta]. \tag{3.27}$$

3.2.6 Included Before / After

An *ib* / *ia* relation means that the second process starts and finishes between the first process starting and finishing. Formally for *ib* this can be stated as $x_s < y_s < y_f < x_f$ or as an annotation represented as

$$R(x, y, t) : [(4, 8), \delta]. \tag{3.28}$$

3.2.7 F-included Before / After

An *fb* / *fa* relation means that the two processes start at the same time and the second process finishes before the first one. Formally for *fb* this can be stated as $x_s = ys < y_f < x_f$ or as an annotation represented as

$$R(x, y, t) : [(5, 7), \delta]. \tag{3.29}$$

3.2.8 Paraconsistent Before-After

A *pba* relation means that the two processes start and finish at the same time. Formally this can be stated as $x_s = y_s < x_f = y_f$ or as an annotation represented as

$$R(x, y, t) : [(6, 6), \delta]. \tag{3.30}$$

3.3 Before-After Inference Rules

As earlier given, bf-EVALPSN record the start and end times of processes using special function symbols *start* and *end*. Calculating the bf-relations from this data, while simple, would require pairwise comparisons between each pair of process atoms.

This would take way too much time, even if it was cut down by half by utilizing the $\neg_1$ epistemic negation to avoid doing two comparisons between each pair of atoms ($R(x, y, t)$ and $R(y, x, t)$).

To avoid the exponential growth in processing needed, bf-EVALPSN defines basic and transitive inference rules with which one can reason about temporal relations between processes. In [7] these rules are introduced in more detail via giving examples of how they would run in a system as it computes each state online. We shall here only show the broad outline of both the basic rules that govern the system.

In the following, let p_i and p_j be some processes. At the beginning neither of them has yet started. We shall then explain all the possible bf-annotations that can occur between these two processes during the runtime of the system. Again only the before-side of the rules are presented, as the after-side is simply the negation of the first.

3.3.1 (0,0)-rules

Suppose that neither process has yet started and thus the annotation associated with $R(p_i, p_j, t)$ is $(0, 0)$. There are two possibilities in this case. Either one process starts before the other, leading to annotation before $(0, 8)$ or after $(8, 0)$, or both processes start at the same time, leading to greatest lower bound of *fa*, *fb* and *pba* which is $(5, 5)$.

3.3.2 (0,8)-rules

Suppose that p_i has already started. If p_i finishes and p_j does not start at the same time, then the annotation should become *db* $(0, 12)$. If p_j starts at the same moment then the annotation is *mb* $(1, 11)$. If process p_j finishes before p_i has finished then the annotation becomes the greatest lower bound of *jb*, *sb* and *ib* which is $(2, 8)$

3.3.3 (5,5)-rules

Suppose that the processes have started at the same time. If now p_i finishes before p_j the annotation should become *sb* $(5, 7)$. If the order is reversed then the annotation becomes *sa* $(7, 5)$ and if the processes finish at the same time then the annotation becomes *pba* $(6, 6)$.

3.3.4 (2,8)-rules

Suppose that process p_i has started before process p_j starts and process p_j has started before p_i finishes. If now p_i finishes first then the annotation becomes *jb* $(2, 10)$. If the processes finish at the same time then the annotation becomes *fb* $(3, 9)$ and if process p_j finishes first then the annotation becomes *ib* $(4, 8)$.

Naturally we see that from any of the given lower bounds, all of the before-after annotations included within that bound are accessible and logically sound. These basic rules give the backbone of the before-after reasoning system. The true strength, however, comes from the transitive rules.

3.3.5 Transitive Before-After Inference Rules

Next we shall delve into the transitive rules. We shall add a process p_k into our considerations. The question at hand is that given the bf-relations between processes p_i and p_j, and p_j and p_k at some point in time then what is the relation between processes p_i and p_k? These transitive rules answer that question and are used to minimize the computations the system needs to do at each time step in order to keep tabs of bf-relations.

In [7] the rules are given in a simple format with just two bf-literals implying a third. This is enough for a simple implementation of bf-EVALPSN but a rigorous language meant for model building cannot work with just that. The problem is that these simple representations of the rules are too general and can often be applied in any or nearly any situation. The resulting annotations would be nonsensical.

As an example, the simplest rule to be looked at is of a situation where p_i has already started but is the only one of the three processes to have done so. In this situation the transitive rule given in [7] is

$$R(p_i, p_j, t) : [(0, 8), \alpha] \wedge R(p_j, p_k, t) : [(0, 0), \alpha] \to R(p_i, p_k, t) : [(0, 8), \alpha]. \tag{3.31}$$

But from Lemma 2.1 the $R(p_j, p_k, t) : [(0, 0), \alpha]$ is always satisfied (the absence of knowledge is always a known deontic fact) and as such can be left out from the rule:

$$R(p_i, p_j, t) : [(0, 8), \alpha] \to R(p_i, p_k, t) : [(0, 8), \alpha]. \tag{3.32}$$

This rule would be madness, however. It would apply to all possible situations where a bf-relation *be* or higher is found for some pair of processes, and that then any third process is always after the earlier process of the two. It would apply even if the third process had actually ended already way before the pair of processes on the left side of the rule began. This should indeed not be the case and thus these rules need additional specifications to keep them from being as generally applicable as they currently are. For this end we use the start and finish time predicates from 3.18.

Using these predicates we rewrite the rules as exclusive, or maximal. The rule 3.31 thus becomes

$$\begin{aligned} R(p_i, p_j, t) : [(0, 8), \alpha] \wedge R(p_j, p_k, t) : [(0, 0), \alpha] \wedge start(p_j, t) : [(0, n), \alpha] \wedge \\ start(p_k, t) : [(0, n), \alpha] \rightarrow R(p_i, p_k, t) : [(0, 8), \alpha]. \end{aligned} \quad (3.33)$$

All the inference rules shall now be listed in a simplified format using with the added *start* and *finish* literals left out and the bf-literals likewise simplified to only their epistemic knowledge levels, that is the bf-annotations. Thus

$$\begin{aligned} &R(p_i, p_j, t) : [(n_1, n_2), \alpha] \wedge R(p_j, p_k, t) : [(n_3, n_4), \alpha] \wedge \\ &start(p_i, t) : \mu_1 \wedge start(p_j, t) : \mu_2 \wedge start(p_k, t) : \mu_3 \wedge \\ &finish(p_i, t) : \lambda_1 \wedge finish(p_j, t) : \lambda_2 \wedge finish(p_k, t) : \lambda_3 \\ &\qquad \rightarrow R(p_i, p_k, t) : [(n_5, n_6), \alpha] \end{aligned} \quad (3.34)$$

is listed as

$$(n_1, n_2) \wedge (n_3, n_4) \rightarrow (n_5, n_6). \quad (3.35)$$

The inference rules are nested within each other in such a way that a nested rule of a lower level may be applied only after the parent rule has been applied on the same triple of processes at an earlier point in time. That is to say that the rule $TR1 - 1$ can be applied at a time t only if the rule $TR1$ has already been applied at some $t' < t$ to the same processes, or formally using a rule predicate eg.

$$TR_1(p_i, p_j, p_k, t)$$

that is set as true for a triplet of processes on the same exact premises as the corresponding rule, in this case **TR 1**, is applied. Then any subsequent rules, in the above example for instance **TR 1-3**, also require the preceding rule prefix to be true for the same triplet of processes. These prefixes are left out of the simplified format.

Transitive bf-Inference Rules

TR 0: $(0,0) \wedge (0,0) \rightarrow (0,0)$
TR 1: $(0,8) \wedge (0,0) \rightarrow (0,8)$
- **TR 1-1:** $(0,12) \wedge (0,0) \rightarrow (0,12)$
- **TR 1-2:** $(1,11) \wedge (0,8) \rightarrow (0,12)$
- **TR 1-3:** $(1,11) \wedge (5,5) \rightarrow (1,11)$
- **TR 1-4:** $(2,8) \wedge (0,8) \rightarrow (0,8)$
  - **TR 1-4-1:** $(2,10) \wedge (0,8) \rightarrow (0,12)$
  - **TR 1-4-2:** $(4,8) \wedge (0,12) \rightarrow (0,8)$
  - **TR 1-4-3:** $(2,8) \wedge (2,8) \rightarrow (2,8)$
    - **TR 1-4-3-1:** $(2,10) \wedge (2,8) \rightarrow (2,10)$
    - **TR 1-4-3-2:** $(4,8) \wedge (2,10) \rightarrow (2,8)$
    - **TR 1-4-3-3:** $(2,8) \wedge (4,8) \rightarrow (4,8)$
    - **TR 1-4-3-4:** $(3,9) \wedge (2,10) \rightarrow (2,10)$
    - **TR 1-4-3-5:** $(2,10) \wedge (4,8) \rightarrow (3,9)$
    - **TR 1-4-3-6:** $(4,8) \wedge (3,9) \rightarrow (4,8)$
    - **TR 1-4-3-7:** $(3,9) \wedge (3,9) \rightarrow (3,9)$
  - **TR 1-4-4:** $(3,9) \wedge (0,12) \rightarrow (0,12)$
  - **TR 1-4-5:** $(2,10) \wedge (2,8) \rightarrow (1,11)$
  - **TR 1-4-6:** $(4,8) \wedge (1,11) \rightarrow (2,8)$
  - **TR 1-4-7:** $(3,9) \wedge (1,11) \rightarrow (1,11)$
- **TR 1-5:** $(2,8) \wedge (5,5) \rightarrow (2,8)$
  - **TR 1-5-1:** $(4,8) \wedge (5,7) \rightarrow (2,8)$
  - **TR 1-5-2:** $(2,8) \wedge (7,5) \rightarrow (4,8)$
  - **TR 1-5-3:** $(3,9) \wedge (5,7) \rightarrow (2,10)$
  - **TR 1-5-4:** $(2,10) \wedge (7,5) \rightarrow (3,9)$

TR 2: $(5,5) \wedge (0,8) \rightarrow (0,8)$
- **TR 2-1:** $(5,7) \wedge (0,8) \rightarrow (0,12)$
- **TR 2-2:** $(7,5) \wedge (0,12) \rightarrow (0,8)$
- **TR 2-3:** $(5,5) \wedge (2,8) \rightarrow (2,8)$

TR 2-3-1: $(5,7) \wedge (2,8) \rightarrow (2,10)$
TR 2-3-2: $(7,5) \wedge (2,10) \rightarrow (2,8)$
TR 2-3-3: $(5,5) \wedge (4,8) \rightarrow (4,8)$
TR 2-3-4: $(7,5) \wedge (3,9) \rightarrow (4,8)$
TR 2-4: $(5,7) \wedge (2,8) \rightarrow (1,11)$
TR 2-5: $(7,5) \wedge (1,11) \rightarrow (2,8)$
TR 3: $(5,5) \wedge (5,5) \rightarrow (5,5)$
TR 3-1: $(7,5) \wedge (5,7) \rightarrow (5,5)$
TR 3-2: $(5,7) \wedge (7,5) \rightarrow (6,6)$

The transitive rules $TR1-4-2$, $TR1-4-3-2$, $TR1-4-6$, $TR1-5-1$, $TR2-2$, $TR2-3-2$, $TR2-5$, and $TR3-1$ do not result in a definite before-after relation, despite having no subsequently applicable transitive rules listed. These rules all end up at a state where the relation between the first and third processes is solvable using one of the basic inference rules. Thus one of those basic inference rules is to be applied to find the final before-after relation being sought.

These rules conclude the basic engine of bf-EVALPSN. With these concepts and rules it is possible to construct complex generalised Horn programs expressing temporal logic on a realtime-like level of accuracy. The language is also capable of expressing deontic notations of permission and obligation, as well as of course normal annotated logic.

4. MODEL BUILDING

Building stable models from annotated logic programs is a solved problem.[8] There is no question of whether EVALPSN can or cannot be for model building, as any general annotated logic necessarily allows for least model semantics. Even the usage of strong negation does not cause problems, as the strong negation can be dealt with using stratification and thus perfect model semantics become usable with EVALPSN.

The question in for this thesis is how to build models from bf-EVALPSN programs. As a general question there is little doubt if it is possible or not. In general, bf-EVALPSN's before-after relations only extend upon the EVALPSN language and the added temporal dimension is dealt away as just another variable in the formulas. But even though the question of "can it be done" is easily answerable in the positive, the question of "how can it be done" does not have an immediately obvious answer.

A naïve approach would take all time points t of the system and go through them in order, trying to build a separate model for each time point. Yet this would soon run into problems with the fact that bf-relations do not, in general, specify the moment of time when they become active. Unless the program is written with only fixed points of time it is mostly meaningless to look at the time points as a progression of integers from 0 to some N that is known in advance where for each $t \in [0, N]$ a known event occurs.

Most likely the program is written only as some promises of processes following each other in some sequence but nothing about the actual points of time they occur on. In this situation most of the time points t hold absolutely no special meaning, and only on some specific time points that cannot be known in advance does anything of interest to the system happen. This is what real time systems mostly look like anyway.

This thesis here proposes a system to deduce those specific time points of interest in advance from the program. This does not mean that we divine the exact time point that an event occurs but instead we deduce from the program the sequence of time points that will occur during the run time of the program. In most interesting programs it is likely that no single sequence can be inferred and instead the possible sequences spread out as a tree into possible futures. The final part in the system is then, naturally, to map this tree into formal automata.

4.1 Transitive Before-After Deduction Rules

In the previous chapter the basic and transitive inference rules of bf-EVALPSN were outlined. Those rules can be used to reason about a system at runtime, updating information about the system's state online as new information becomes available. It is also used to transfer real knowledge into knowledge at the logic language level. These, however, do not aid us in deducing information about the capabilities and pitfalls of a system as it is described to us as a collection of clauses.

The basic deductive rules of bf-EVALPSN were outlined in the previous section. The natural next step we must take is to explore the impact of bf-clauses cause on the model to be built from a Horn program. In the transitive inference rules it was possible to only reason about a smaller set of bf-EVALPSN relations, essentially choosing to always view bf-relations from the earlier process' direction, thus only acting on the 'before' side of bf-EVALPSN relations. When acting in the deductive side of transitive bf-EVALPSN rules, it is not possible to choose preferable bf-EVALPSN relations to reason with. Instead, we must do deduction on those rules that we're given. The only reasonable expectation we can choose is that for any triple of processes, Pr_1, Pr_2 and Pr_3, there always exists some way to connect the three temporally for if there is none, then we can split the three processes into two sets, one of which has no temporal relations with the other and these two can thus be reasoned about separately.

Furthermore, as we are interested in building a model of the system from a set of rules we are given about it, we do not cocern with imperfect bf-relations nor about the exact moment of time at which these rules become true. We only concern ourselves with rules that our system is eventually forced to conform to. This means that when writing these deduction rules we can do away with both the deontic annotation as well as the time parameter in the bf-relation. The deontic annotation can be done away with because only the annotation of fact, α, is used in the rules as only the factual state of the system impacts our further deductions on what the state of the system will eventually be. The time parameter for the bf-EVALPSN relation can, again, be omitted as the exact time when a rule becomes true is generally not important to our rules. We say generally because the expectation is that most bf-relation rules given to a system are actually existential with regards to the time parameter, that is to say that they do not define an exact time when the rule becomes true but posit that there is a time when the rule will become true and thus due to the nature of bf-EVALPSN relations will be true when the system reaches a ‘terminal‘ position. The rules are then likewise written with an implied existential time parameter. If the system is given bf-relations that are not existential, they can either be universal which makes no sense and should be removed, or they can state the time parameter exactly. An exact time parameter will simply create a set point of time in the system that can help in other comparisons but otherwise does not change the implications in comparison to the existential rule. Thus the deduction rules are given in the format

$$R(Pr_1, Pr_2) : (a, b) \wedge R(Pr_2, Pr_3) : (c, d) \rightarrow R(Pr_1, Pr_3) : (e, f) \tag{4.1}$$

Each bf-EVALPSN relation must now be compared with every other to come up with all the possible transitive before-after deduction rules for bf-relations. As exhaustively outlining all the rules would be too intensive, only the head and body of a deduction rule clause are given for each rule. Each rule has a body containing two bf-EVALPSN formulas, one pertaining to the bf-relation between processes some Pr_1 and Pr_2, the other to that of Pr_2 and Pr_3. Finally the head of the clause establishes the resultant bf-relation of Pr_1 and Pr_3 deducible from the body.

It is possible that a rule will not have a definite bf-relation solution. For instance it may be that the result is $(0, 0)$ or $\perp$, that is to say that all bf-relations are possible. It may also be that only some bf-relations are possible but they cannot be all represented by a single annotation without including extra annotations that are not valid possible solutions. In these cases the head of the rule clause shows multiple annotations separated by $\vee$. However, that is not a valid GHC head and when the rule is to be used in an actual system, the annotations must be converted to their greatest lower bound. This despite the greatest lower bound also satisfying other bf-relations that are not truly possible from the setting. The reason for showing only the possible annotations and leaving out the impossible ones in the rule representation is that this information will be used later to create a separate model for each of the possible annotations.

We shall go through the deduction rules so that each following subsection defines the bf-relation between Pr_1 and Pr_2, and within is a list of clauses with each possible bf-relation of Pr_2 and Pr_3 used in the rule body in order. An overview of the rules is interspersed within the list. The rules are condensed into sets represented by the greatest lower bound when possible

4.1.1 Disjoint before - db (0,12)

The first process starts and finishes before the second.

- $R(Pr_1, Pr_2) : (0, 12) \wedge R(Pr_2, Pr_3) : (0, 5) \rightarrow R(Pr_1, Pr_3) : (0, 12)$

As long as the third process starts after or at the same time as the second, the relation between the disjoint before first process and the third process is always likewise disjoint before.

- $R(Pr_1, Pr_2) : (0, 12) \wedge R(Pr_2, Pr_3) : (8, 1) \rightarrow$
  $R(Pr_1, Pr_3) : (0, 10) \vee (5, 7) \vee (8, 4)$

Once the start time of the third process is freed from the bound of the starting time of the second, then we instantly lose our definite knowledge of the bf-relation between the two. The choices become that either the first is still disjointly before, is immediately before, jointly before or f-included before the third process if the start time of the third process is at least not before the start time of the first, or included after if the third process starts before the first.

- $R(Pr_1, Pr_2) : (0, 12) \wedge R(Pr_2, Pr_3) : (12, 0) \rightarrow R(Pr_1, Pr_3) : (0, 0)$

Finally if the third process is disjointly before the second process, then we cannot deduce anything about the relation between the first and the third process.

4.1.2 Immediate before - mb (1,11)

The first process finishes immediately as the second process starts.

- $R(Pr_1, Pr_2) : (1, 11) \wedge R(Pr_2, Pr_3) : (0, 8) \rightarrow R(Pr_1, Pr_3) : (0, 12)$

Again as expected, when the third process comes after the second, the resulting annotation is a disjoint before.

- $R(Pr_1, Pr_2) : (1, 11) \wedge R(Pr_2, Pr_3) : (5, 5) \rightarrow R(Pr_1, Pr_3) : (1, 11)$

If the start times of the second and third process coincide, then the annotation becomes immediate before.

- $R(Pr_1, Pr_2) : (1, 11) \wedge R(Pr_2, Pr_3) : (8, 2) \rightarrow$
  $R(Pr_1, Pr_3) : (2, 10) \vee (5, 7) \vee (8, 4)$

When the start time of the third process is released form the bound of the second process' start time, we see the annotation become joint before, f-included before or included after.

- $R(Pr_1, Pr_2) : (1, 11) \wedge R(Pr_2, Pr_3) : (11, 1) \rightarrow$
  $R(Pr_1, Pr_3) : (3, 9) \vee (6, 6) \vee (9, 3)$
- $R(Pr_1, Pr_2) : (1, 11) \wedge R(Pr_2, Pr_3) : (12, 0) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 2)$

If the second process is immediately after the third, then the annotation becomes s-included after or before, or paraconsistent before-after. Finally if the third process is disjointly before the second process, the annotation can be included before, f-included before or joint after.

4.1.3 Joint before - jb (2,10)

The first process starts before the second and finishes after the second starts but before it finishes.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (0, 1) \rightarrow R(Pr_1, Pr_3) : (0, 12)$

If the third process starts after or as the second is ending then the relation between the first and third is clearly disjoint before.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (2, 9) \rightarrow R(Pr_1, Pr_3) : (0, 10)$

If the third is jointly or s-included after the second, then the resultant relation can either be disjoint, immediate or joint before.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (4, 8) \rightarrow R(Pr_1, Pr_3) : (0, 8)$
- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (5, 6) \rightarrow R(Pr_1, Pr_3) : (2, 10)$
- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (7, 5) \rightarrow R(Pr_1, Pr_3) : (2, 8)$

If the third process is included within the second (included after) then the result can be anything between disjoint before and included before, effectively resulting in the before relation *be*.

If the third process is f-included after or paraconsistent before-after the second, its starting time is the same as the second process and its finishing time is after (*fa*) or equal (*pba*) of the second, thus resulting in the joint before relation. In the f-included before relation the finishing time is before the second process and thus the resulting relation can be joint before, s-included before or included before.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (8, 4) \rightarrow$
  $R(Pr_1, Pr_3) : (2, 10) \vee (5, 7) \vee (8, 4)$
- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (9, 3) \rightarrow$
  $R(Pr_1, Pr_3) : (2, 10) \vee (5, 7) \vee (8, 4)$

If the second process is included or s-included within the third, meaning that the third process starts before the second and finishes later (*ia*) or at the same time (*sa*), then the relation between the first and third can be joint before, f-included before or included after. This somewhat bizarre trio occurs relatively often in annotations. Essentially it describes the situation where one process is known to end after the other and they are known to overlap each other, but the process finishing after the other has an otherwise unbound start time.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (10, 2) \rightarrow R(Pr_1, Pr_3) : (2, 2)$

If both the first and third processes are jointly before the second (in the rule written as 'second is jointly after third') then we only know that their finish times are within the running time of the second process and their start times are before the second process starts. This makes disjoint before / after and immediate before / after relations between the first and third process impossible but otherwise any bf-relation is possible.

- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (11, 1) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 2)$
- $R(Pr_1, Pr_2) : (2, 10) \wedge R(Pr_2, Pr_3) : (12, 0) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 0)$

If the third process is immediately before the second, it means that the third process finishes before the first. Their start times may still have any order and thus the resulting bf-relation is a combination of included before (first process starts first, finishes after), f-included after (first and third processes start together, first finishes after) and joint after (first process starts, finishes after). If the third process is disjointly before the second, then it is also possible that the third process finishes at the same time as the first starts, or even before that. Thus the immediate after and disjoint after relations get added to the mix.

4.1.4 S-included before - sb (3,9)

The first process starts before the second process and they finish at the same time.

- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (0, 12) \rightarrow R(Pr_1, Pr_3) : (0, 12)$

- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (1, 11) \to R(Pr_1, Pr_3) : (1, 11)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (2, 10) \to R(Pr_1, Pr_3) : (2, 10)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (3, 9) \to R(Pr_1, Pr_3) : (3, 9)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (4, 8) \to R(Pr_1, Pr_3) : (4, 8)$

In s-included before the finish times of the two processes coincide and the first process starts before the first. Thus as long as the third process is strictly after the second, then the resulting bf-relation between the first and third process is exactly the same as the second and third.

- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (5, 7) \to R(Pr_1, Pr_3) : (2, 10)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (6, 6) \to R(Pr_1, Pr_3) : (3, 9)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (7, 5) \to R(Pr_1, Pr_3) : (4, 8)$

When the start times of the second and third process are bound together, then if the third process finishes after the second the relation between the first and third becomes joint before. If the third process finishes before the second, then the relation becomes included before. Of course, as usual, if the second and third process coincide completely then the relation between the first and third is the same as the first and second processes.

- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (8, 4) \to$
  $R(Pr_1, Pr_3) : (2, 10) \vee (5, 7) \vee (8, 4)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (9, 3) \to$
  $R(Pr_1, Pr_3) : (3, 9) \vee (6, 6) \vee (9, 3)$

If the second process is included within the third, then the relation between the first and third can be either joint before, f-included before, or included after. If both first and third are s-included before the second, it means that the first and third processes' finish times coincide. Thus they may either be themselves s-included before or after each other, or completely coincide in paraconsistent before-after relation.

- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (10, 1) \to$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 2)$
- $R(Pr_1, Pr_2) : (3, 9) \wedge R(Pr_2, Pr_3) : (12, 0) \to$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 0)$

If the third process is immediately or jointly before the second, then the relation between the first and third can be included before, f-included after or joint after, same as with the previous joint before relationship. And similarly, when disjoint after is considered, the immediate after and disjoint after relations become possible between Pr_1 and Pr_3.

4.1.5 Included before - ib (4,8)

The first process starts before and finishes after the second.

- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (0, 12) \rightarrow R(Pr_1, Pr_3) : (0, 8)$
- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (1, 10) \rightarrow R(Pr_1, Pr_3) : (2, 8)$
- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (3, 8) \rightarrow R(Pr_1, Pr_3) : (4, 8)$
- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (5, 7) \rightarrow R(Pr_1, Pr_3) : (2, 8)$
- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (6, 5) \rightarrow R(Pr_1, Pr_3) : (4, 8)$

When the second process is included within the first and the relation between Pr_2 and Pr_3 is at most f-included after (the processes start at the same time, Pr_3 finishes first) then the third process is always strictly after the first. If start time of the third process is disjoint from the finish of the second, then any strictly before relation between Pr_1 and Pr_3 is possible. If the start time is bound between the start or finish of Pr_2 without binding the finish time (*mb*, *jb*, *fb*) then immediate and disjoint before become impossible. If both the start and finish time of Pr_3 are bound within Pr_2 one way or another (*sb*, *ib*, *pba*, *fa*) then the resulting relation between Pr_1 and Pr_2 is necessarily included before.

- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (8, 4) \rightarrow R(Pr_1, Pr_3) : (2, 2)$

If the second process is included within both the first and the third processes, then the relation between Pr_1 and Pr_3 can be anything except disjoint and immediate before / after.

- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (9, 1) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 2)$
- $R(Pr_1, Pr_2) : (4, 8) \wedge R(Pr_2, Pr_3) : (12, 0) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 0)$

Again if the relation between Pr_2 and Pr_3 is such that the start time of Pr_3 is unbound with relation to Pr_2 but the finish time is bound to the start time of it, then the relation between Pr_1 and Pr_3 can included before, f-included after or joint after, and if the finish time is likewise unbound then immediate and disjoint after become possible.

4.1.6 F-included before - fb (5,7)

The first and second processes start at the same time and the first process finishes before the second.

- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (0,11) \rightarrow R(Pr_1, Pr_3) : (0,12)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (2,9) \rightarrow R(Pr_1, Pr_3) : (0,10)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (4,8) \rightarrow R(Pr_1, Pr_3) : (0,8)$

If the third process is disjointly or immediately after the second, then the resultant relation is of course disjoint before. When the third process starts within the runtime of Pr_2 it becomes possible for the resultant relation to be immediate or joint before as well. Then if the third process is included within the run time of the second, then Pr_1 is known to be before Pr_3 and nothing more.

- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (5,6) \rightarrow R(Pr_1, Pr_3) : (5,7)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (7,5) \rightarrow R(Pr_1, Pr_3) : (5,5)$

If the start times of Pr_2 and Pr_3 coincide, then if the third process finishes after or at the same time as the second the resultant relation is necessarily f-included before. Else it can also be paraconsistent before-after or f-included after.

- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (8,3) \rightarrow R(Pr_1, Pr_3) : (8,4)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (10,2) \rightarrow R(Pr_1, Pr_3) : (8,2)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (11,1) \rightarrow R(Pr_1, Pr_3) : (11,1)$
- $R(Pr_1, Pr_2) : (5,7) \wedge R(Pr_2, Pr_3) : (12,0) \rightarrow R(Pr_1, Pr_3) : (12,0)$

If Pr_3 contains Pr_2 within its run time, then necessarily also Pr_1 is contained within it. If the third process is jointly before the second, then the first process is either included after, s-included after or joint after the third. Finally if immediate after and disjoint after relations between Pr_2 and Pr_3 also map into the relation between Pr_1 and Pr_2 as Pr_1 and Pr_2 share their start time.

4.1.7 Paraconsistent before-after - pba (6,6)

If Pr_1 and Pr_2 completely coincide in their run times, then the bf-relation between Pr_2 and Pr_3 will also be the bf-relation between Pr_1 and Pr_3.

4.1.8 F-included after - fa (7,5)

The first and second processes start at the same time and the first process finishes after the second.

- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (0,12) \rightarrow R(Pr_1, Pr_3) : (0,8)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (1,10) \rightarrow R(Pr_1, Pr_3) : (2,8)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (3,8) \rightarrow R(Pr_1, Pr_3) : (4,8)$

When the third process is disjointly after the second, it is known to be after the first. If it comes immediately or jointly after the second, then it can be joint, s-included or included after the first. If it comes s-included after or included after the second, then it is known to be included within the first process.

- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (5,7) \rightarrow R(Pr_1, Pr_3) : (5,5)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (6,5) \rightarrow R(Pr_1, Pr_3) : (7,5)$

If both the second and third processes are f-included before / after or paraconsistent before-after each other it means that Pr_1 and Pr_3 share their starting time. If the third process finishes after the second (*fb*) then the resultant relation is any of f-included before, paraconsistent before-after or f-included after. If Pr_3 finishes at the same time or before as Pr_2, then the result is f-included after.

- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (8,4) \rightarrow R(Pr_1, Pr_3) : (8,2)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (9,2) \rightarrow R(Pr_1, Pr_3) : (10,2)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (11,1) \rightarrow R(Pr_1, Pr_3) : (11,1)$
- $R(Pr_1, Pr_2) : (7,5) \wedge R(Pr_2, Pr_3) : (12,0) \rightarrow R(Pr_1, Pr_3) : (12,0)$

If the second process is included within the third, then the first is either included, s-included or joint after of Pr_3. If Pr_2 finishes at the same time as or within, and starts after Pr_3, then the relation between Pr_1 and Pr_3 is joint after. Finally if Pr_3 is immediately or disjointly before Pr_2, then the same relation exists between Pr_3 and Pr_1.

4.1.9 Included after - ia (8,4)

The first process starts after the second and finishes before the second finishes, being completely subsumed within the second.

- $R(Pr_1, Pr_2) : (8,4) \wedge R(Pr_2, Pr_3) : (0,11) \rightarrow R(Pr_1, Pr_3) : (0,12)$

If the third process starts immediately or disjointly after the second, then the relation between Pr_1 and Pr_3 is disjoint before.

- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (2, 9) \rightarrow$
  $R(Pr_1, Pr_3) : (0, 10) \vee (5, 7) \vee (8, 4)$

If the third process starts between the second process starting and finishing, and finishes either at the same time or after the second, then the relation between Pr_1 and Pr_2 can be disjoint before, immediate before, joint before, f-included before or included after.

- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (4, 8) \rightarrow R(Pr_1, Pr_3) : (0, 0)$
- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (5, 6) \rightarrow R(Pr_1, Pr_3) : (8, 4)$
- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (7, 5) \rightarrow R(Pr_1, Pr_3) : (8, 0)$
- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (8, 3) \rightarrow R(Pr_1, Pr_3) : (8, 4)$
- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (10, 2) \rightarrow R(Pr_1, Pr_3) : (8, 0)$

If the third process is likewise included within the second, then the relation between Pr_1 and Pr_3 is completely unknown. If the start times of Pr_2 and Pr_3 coincide and Pr_3 finishes either at the same time as Pr_2 or after it, then it also includes Pr_1 within itself, giving the relation included after. If, however, Pr_3 finishes before Pr_2 in this setting, then know only that the relation between Pr_1 and Pr_3 is after.

Again if Pr_3 finishes at the same time as Pr_2 and starts before it, then it also includes Pr_1 within itself, and if Pr_3 is jointly before Pr_2 then it is known to be before Pr_1.

- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (11, 1) \rightarrow R(Pr_1, Pr_3) : (12, 0)$
- $R(Pr_1, Pr_2) : (8, 4) \wedge R(Pr_2, Pr_3) : (12, 0) \rightarrow R(Pr_1, Pr_3) : (12, 0)$

If Pr_3 finishes before or as Pr_2 starts, then the relation between Pr_1 and Pr_3 is disjoint before.

4.1.10 S-included after - sa (9,3)

The first process starts after the second process and they finish at the same time.

- $R(Pr_1, Pr_2) : (9, 3) \wedge R(Pr_2, Pr_3) : (0, 12) \rightarrow R(Pr_1, Pr_3) : (0, 12)$
- $R(Pr_1, Pr_2) : (9, 3) \wedge R(Pr_2, Pr_3) : (1, 11) \rightarrow R(Pr_1, Pr_3) : (1, 11)$

When the third process is immediately or disjointly after the second, the same relation exists between Pr_3 and Pr_1.

- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (2,10) \rightarrow$
  $R(Pr_1, Pr_3) : (2,10) \vee (5,7) \vee (8,4)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (3,9) \rightarrow$
  $R(Pr_1, Pr_3) : (3,9) \vee (6,6) \vee (9,3)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (4,8) \rightarrow$
  $R(Pr_1, Pr_3) : (4,8) \vee (7,5) \vee (10,0)$

If Pr_2 is jointly before Pr_3, meaning that the finish time of Pr_3 is after the finish of and the start time is included within Pr_2, then Pr_1 is either joint before, f-included before or included after Pr_3. If all three processes share an end time and Pr_1 and Pr_3 start after Pr_2, then their relation is one of s-included after/before, or paraconsistent before-after. If Pr_3 is included within Pr_2 then the resulting relation can be either likewise included before, f-included after (Pr_1 and Pr_3 start at the same time, Pr_1 finishes after), or any of joint, immediate or disjoint after.

- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (5,7) \rightarrow R(Pr_1, Pr_3) : (8,4)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (6,6) \rightarrow R(Pr_1, Pr_3) : (9,3)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (7,5) \rightarrow R(Pr_1, Pr_3) : (10,0)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (8,4) \rightarrow R(Pr_1, Pr_3) : (8,4)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (9,3) \rightarrow R(Pr_1, Pr_3) : (9,3)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (10,2) \rightarrow R(Pr_1, Pr_3) : (10,0)$

If Pr_2 and Pr_3 start at the same time and Pr_3 finishes after, then Pr_1 is included after Pr_3. If, however, Pr_3 finishes before, then Pr_1 can be joint after, immediate after or disjoint after Pr_3. If Pr_3 includes Pr_2 then it necessarily also includes Pr_1. Similarly if Pr_3 is s-included before Pr_2 then it is likewise s-included before Pr_1, and if it is included before Pr_2 then it is joint before, immediate before or disjoint before Pr_1.

- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (11,1) \rightarrow R(Pr_1, Pr_3) : (12,0)$
- $R(Pr_1, Pr_2) : (9,3) \wedge R(Pr_2, Pr_3) : (12,0) \rightarrow R(Pr_1, Pr_3) : (12,0)$

Finally if Pr_3 finishes before or as Pr_2 starts, then Pr_1 is disjoint after Pr_3.

4.1.11 Joint after - ja (10,2)

The first process starts after the second starts but before it finishes, and finishes only after the second has finished.

- $R(Pr_1, Pr_2) : (10,2) \wedge R(Pr_2, Pr_3) : (0,12) \rightarrow R(Pr_1, Pr_3) : (0,8)$
- $R(Pr_1, Pr_2) : (10,2) \wedge R(Pr_2, Pr_3) : (1,11) \rightarrow R(Pr_1, Pr_3) : (2,8)$

- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (2, 10) \rightarrow R(Pr_1, Pr_3) : (2, 2)$

If Pr_3 starts after Pr_2 has already finished then Pr_1 is before Pr_3 but nothing else can be known about their relation. If Pr_3 starts as Pr_2 finishes, then Pr_1 is joint, s-included or included before Pr_3, and if both Pr_1 and Pr_3 are joint after Pr_2 then the relation between the two of them can be anything except disjoint or immediate before / after.

- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (3, 9) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 2)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (4, 8) \rightarrow$
  $R(Pr_1, Pr_3) : (4, 8) \vee (7, 5) \vee (10, 0)$

If Pr_3 starts after and finishes at the same time as Pr_2, then it is possible that it is either completely included within Pr_1, starts at the same time as Pr_1 and finishes before it, or starts before Pr_1 and finishes while Pr_1 is running. If Pr_3 finishes before Pr_2, then it is also possible that Pr_3 finishes right at the time Pr_1 starts or before that time. Thus the possibilities are included before, f-included after, joint after, and for $R(Pr_2, Pr_3) : (4, 8)$, immediate after, and disjoint after.

- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (5, 7) \rightarrow R(Pr_1, Pr_3) : (8, 2)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (6, 6) \rightarrow R(Pr_1, Pr_3) : (10, 2)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (7, 5) \rightarrow R(Pr_1, Pr_3) : (10, 0)$

If Pr_3 starts at the same time as and finishes after Pr_2, then the relation between Pr_1 and Pr_3 can either be joint after, s-included after or included after. If Pr_2 finishes after Pr_3, then the relation becomes either joint after, immediate after, or disjoint after.

- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (8, 4) \rightarrow R(Pr_1, Pr_3) : (8, 2)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (9, 3) \rightarrow R(Pr_1, Pr_3) : (10, 2)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (10, 2) \rightarrow R(Pr_1, Pr_3) : (10, 0)$
- $R(Pr_1, Pr_2) : (10, 2) \wedge R(Pr_2, Pr_3) : (11, 0) \rightarrow R(Pr_1, Pr_3) : (12, 0)$

If Pr_2 is included within Pr_3, then Pr_1 is either included after, s-included after, or included after Pr_3. If Pr_2 and Pr_3 finish at the same time and Pr_3 starts before Pr_2 then the relation between Pr_1 and Pr_3 is then necessarily joint after. If instead Pr_3 finishes after Pr_2 starts and before it finishes, then the relation is either disjoint, immediate or joint after. Finally if Pr_3 finishes as or before Pr_2 starts, then the relation between Pr_1 and Pr_3 becomes necessarily disjoint after.

4.1.12 Immediate after - ma (11,1)

The first process starts right as the second process finishes.

- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (0, 12) \rightarrow R(Pr_1, Pr_3) : (0, 8)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (1, 11) \rightarrow R(Pr_1, Pr_3) : (5, 5)$

If the third process starts after the second finishes, then the relation between the first and third is before and nothing more specific can be stated. If the third process starts as the second finishes, then the first and third processes start at the same time and the relation becomes f-included before or after, or paraconsistent before-after.

- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (2, 10) \rightarrow R(Pr_1, Pr_3) : (8, 2)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (3, 9) \rightarrow R(Pr_1, Pr_3) : (11, 1)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (4, 8) \rightarrow R(Pr_1, Pr_3) : (12, 0)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (5, 7) \rightarrow R(Pr_1, Pr_3) : (8, 2)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (6, 6) \rightarrow R(Pr_1, Pr_3) : (11, 1)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (7, 5) \rightarrow R(Pr_1, Pr_3) : (12, 0)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (8, 4) \rightarrow R(Pr_1, Pr_3) : (8, 2)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (9, 3) \rightarrow R(Pr_1, Pr_3) : (11, 1)$
- $R(Pr_1, Pr_2) : (11, 1) \wedge R(Pr_2, Pr_3) : (10, 0) \rightarrow R(Pr_1, Pr_3) : (12, 0)$

The rest of the relations are simple: If the finish time of Pr_3 is only bound to be after the finish time of Pr_2 (*jb*, *fb*, *ia*), then the relation is one of joint after, s-included after and included after. If the finish time is the same as the finish time of Pr_2 (*sb*, *pba*, *sa*), then the relation is immediate after. Finally if the finish time of Pr_3 is bound to be before the finish time of Pr_2 (*ib*, *fa*, *ja*, *ma*, *da*) then the relation between Pr_1 and Pr_2 is disjoint after.

4.1.13 Disjoint after - da (12,0)

The second process finishes before the first process starts.

- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (0, 12) \rightarrow R(Pr_1, Pr_3) : (0, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (1, 10) \rightarrow R(Pr_1, Pr_3) : (8, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (3, 8) \rightarrow R(Pr_1, Pr_3) : (12, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (5, 7) \rightarrow R(Pr_1, Pr_3) : (8, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (6, 5) \rightarrow R(Pr_1, Pr_3) : (12, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (8, 4) \rightarrow R(Pr_1, Pr_3) : (8, 0)$
- $R(Pr_1, Pr_2) : (12, 0) \wedge R(Pr_2, Pr_3) : (9, 0) \rightarrow R(Pr_1, Pr_3) : (12, 0)$

If the second process is before both the first and third processes, then the first and third process may have any relation whatsoever. Otherwise, as far as the finish time of the third process is after the second process's finish time then the relation between the first and third processes is after, and otherwise the relation is disjoint after.

4.2 Model to Automata States

Imagine that we have a stratifiable bf-EVALPSN program P. Now when we want to apply the immediate consequence operator T_P on the program to find a least model, we run into a problem. Atoms that, for instance, are true on such time points t that some process is active can only be said to be just that, true on those time points. But unless we know these time points beforehand, we can say nothing about the actual time. To know this, we would need to deduce the timings of the system. Luckily for us, it is possible to do this using our previously outlined deductive bf-rules. For every pair of bf-relations in the program with one process shared between them, we use one of the deduction rules to gather bf-relation data of the other processes. Of note here is that although our rules were written from the presumption that the shared process between the two bf-relations in the rules occur once as the second and once as the first process in a relation, it is easy to avoid this requirement as bf-relations are asymmetric relations. That is, the following holds:

$$\text{If } R(Pr_1, Pr_2, t) : [(a, b), \delta] \text{ then } R(Pr_2, Pr_1, t) : [(b, a), \delta]. \tag{4.2}$$

This means we can always flip either one of the relations to fit it into one of the deduction rules. With these deduction rules we work through the program's bf-relations with the goal of establishing a definite bf-relation between each pair of processes. Our aim is to determine all the possible states that the system can have with regards to the processes present in the system. Once these are known we can create a complete model for each of these states of the program.

Before we do this, however, we may as well use the immediate consequence operator T_P to the program before we start dividing it into states. Even though the resulting model from this action will not tell us much with regards to the temporal relations, it will still be a a complete model of the program in the logic programming sense. It satisfies every clause in the program, as guaranteed by the stratification and the upward iteration on the program. This model does serve to give us some information on the system, showing which bf-relations are fulfilled and a basic clue on how the system may run. However, it is not easy to process. The information comes, at its core, as a list of annotated atoms. It does not outline any clear view of how the program runs. To improve the visibility into the system's run time, we then begin our work using the transitive bf-relation deduction rules.

This step is best done as a simple width-first application of the rules for each triple of processes. If no rule can be applied for a given triple, it is marked for later processing. Once all triples are processed once, the marked triples are reprocessed with the new information gained from the previous round of applications. The same cycle of processing and marking is repeated until no new information is gained on two sequential rounds.

If after this our system has only definite bf-relations, we can rest easy as our system is completely synchronous and building an automaton representation for the program can proceed without any further steps. If, however, there are indefinite bf-relations remaining in the program it means that the program has asynchronous parts to it, having multiple possible routes that it may take during the run time, and as such to anticipate all possible states of the system we must begin splitting the model into alternate models for each possible bf-relation.

To split the model, choose one indefinite bf-relation in the system, giving preference to

1. hints given by the program defintion, such as a group of processes from which the earliest process is chosen from,
2. those relations known to occur earlier in the program (eg. two indefinite *sba* relations exist, but one is known to be disjointly before Pr_x while the other is known to be disjointly after it),
3. those relations with the least number of possible definite bf-relations (eg. *sba* (3,3) has three, *sb* (3,9), *pba* (6,6), and *sa* (9,3), while *jba* has 9, from *jb* (2,10) to *ja* (10,2)),
4. preferring lower 'after' values and higher 'before' values of the bf-relations (referring to the (a, b) lattice of the bf-relations with a here being called 'after' value and b being called 'before' value, thus eg. *sba* (3,3) takes precedence over *fba* (5,5) while both have the same number, 3, of possible definite bf-relations).

Having chosen an indefinite bf-relation, replace the previous model with a set of models where for each definite bf-relation possible for the chosen relation a new model exists with indefinite bf-relation replaced with the definite relation. We mark this model with the definite bf-relation that distinguishes it. Then for each new model, use the immediate consequence operator to iterate upwards until a new least model is found. If one or more of these new models still contains indefinite bf-relations, we continue the splitting and these models are marked with a concatenation of the parent model's label and the definite bf-relation that replaces the chosen indefinite one. An example label would then be such as

$$R(Pr_1, Pr_2, T) : [(11, 1), \alpha], R(Pr_2, Pr_3, T) : [(4, 8), \alpha]. \tag{4.3}$$

This way we should end up with a set of models containing only definite bf-relations, each labeled with a list of bf-relations chosen during the creation of the model. Each of these models can then be converted into an automata-like state and transition tree with only one branch from beginning to end. The following algorithm is done for each alternative model in turn.

Theorem 5 (Translation of definite bf-EVALPSN Program into Formal Automaton)**.** Let Pr_n, $n \in \mathcal{N}$ be a finite set of processes, R be a set of bf-EVALPSN rules on Pr_n all relatable to each other, and P be a set of EVALPSN formulas related to the processes. Presume that the bf-EVALPSN rules are in order from the earliest to the latest in terms of bf-relations and no indefinite relations remain in R. Now for each rule R in the set, in order of earliest to latest, the following steps are taken:

1. The starting and finish times of the two processes in the rule are nominated as t_i with i being the next unused integer in the set of all time points.
2. List the times in order along with a note on what happens at that point of time, marked with $start(Pr_n)$ for a process starting, or $finish(Pr_n)$ for a process finishing.
3. For each new interval between two set time points assign a state name S_K with K being the concatenation of processes state changes preceding the state. (eg. $start(Pr_1)$, $start(Pr_2)$, $finish(Pr_2)$)
4. For each acceptable state, add to a ledger of models, the zero state being $S_\emptyset = T_P$ and other states being f.ex. $S_{start(Pr_1)} = T_{P \cup \{start(Pr_1)\}}$ according to which processes have started and finished preceding the state.

From the bf-rule application previously we know the sets of started processes and finished processes to be unchanging for the whole duration of each state, as if a process started or finished during the state then the state should have been split into two separate states earlier. Thus we can drop out the time variable t from the bf-relations, and $start$ and $finish$ function literals in the state. As the creation of the alternate models is somewhat an implementation specific system, it is allowed for the program to set certain acceptability categories for the resultant states. A fairly normal category would be, for instance, to accept only such states which do not contain such active processes that are marked as forbidden from being active during the state using a deontic literal. A more strict version of this would be to only allow such states for which the state transition event (eg. $start(Pr_2)$) has explicit permission set in the system using a deontic literal.

It is possible that some time variables in the system still cannot be unified into our system of states but we will leave those complications to a later discussion. Thus we should have a state program with only definite bf-relations with no dependency to the exact time of the system, a set of started and finished processes, and normal EVALPSN formulas. This combination should still be stratifiable according to our premise, given that we have not added any strongly negated literals, and thus upwards iteration will find us a least model, or show that the current state is already a least model.

Once all models are translated to states, take all states from all models and combine them into one automaton by taking all coinciding states across the trees, and for every state collect all incoming and outgoing state transitions from the different trees based on the timings marked down in step 2 above. When selecting coinciding trees we should be selecting based on the contents of the state but ignore the bf-EVALPSN $start$ and $finish$ functional literals. This is both to try and limit the number of states in the resultant automaton and to make sure that we do not end up with two states that are functionally equivalent but differ in what kind of a route the system has taken to reach the state. While the routing is an important feature, it is already contained in the state transitions and thus is not needed in the states themselves.

Once the states are combined and the state transitions are marked down, we should have a formal automaton described by the set of states, with the state transitions describing the transition function, the localised $start$ and $finish$ functional literals forming the alphabet of the automaton, and the zero state $S_\emptyset$ being both the start state and the only accept state.

4.3 Analysing the Algorithm

The outline for the algorithm for translating a bf-EVALPSN program into an automaton was given in the previous section. The premise of the algorithm was said to be a stratifiable bf-EVALPSN program and this premise certainly needs to be fulfilled for the algorithm to be at all usable, given the requirement of upwards iteration of the program. A further requirement that was hinted at but not outright mentioned is the forbiddance of complex use of the time parameter in annotated atoms of the program P. To give an example, the following clause

$$\exists t \, \forall t^{'} > t : \; A(t^{'}) : [(1,0),\alpha] \leftarrow B(t) : [(1,0),\alpha] \tag{4.4}$$

cannot really be unified with the states generated from the algorithm. The state division works based on bf-relations, which do not contain knowledge of the length of a time periods. The clause 4.4 seems to suggest a bf-relation connection between A and B where B at least starts before A starts. But where the temporal relation of the two may be expressible using bf-literals, there is no way to express the knowledge that A should become true on the next time point after B. Effectively this sort of explicit use of the time parameter may force the system into new states that the algorithm cannot divine.

Furthermore, it is possible that during the run of the algorithm more bf-relations are uncovered. Take for instance the following program:

$$\begin{aligned}
&R(Pr_1, Pr_2, t) : [(1,11),\alpha] \leftarrow \\
&R(Pr_1, Pr_3, t) : [(1,11),\alpha] \leftarrow \\
&R(Pr_3, Pr_4, t) : [(0,12),\alpha] \leftarrow R(Pr_2, Pr_3, t) : [(7,5),\alpha] \\
&R(Pr_3, Pr_4, t) : [(1,11),\alpha] \leftarrow R(Pr_2, Pr_3, t) : [(6,6),\alpha] \\
&R(Pr_3, Pr_4, t) : [(1,11),\alpha] \leftarrow R(Pr_2, Pr_3, t) : [(5,7),\alpha]
\end{aligned} \tag{4.5}$$

This program describes system where first a process Pr_1 starts, followed immediately by Pr_2 and Pr_3. The end times of these two are not specifically defined (a classic asynchronous parallel process) and can thus their bf-relation can be either f-included before, paraconsistent before-after, or f-included after. Now the program states that if process Pr_2 finishes at the same time or before Pr_3, then a fourth process Pr_4 starts immediately as Pr_3 finishes. If, however, Pr_2 finishes after Pr_3, then Pr_4 starts disjointly after Pr_3.

In this program our initial model can give us no information on the relation between Pr_4 and any other process. It might even be argued on that basis that Pr_4 is completely disjoint from the rest of the program although this is obviously not the case. Only through the application of the algorithm will we find the two different possible bf-relations between Pr_3 and Pr_4, which also incidentally opens up three possible relations between Pr_2 and Pr_4 (disjoint before, immediate before and included before). This obviously means that after using the algorithm we end up with new states that have not been added to the ledger of states yet as well as new indefinite bf-relations that have not yet been split into definite alternate models. Through this it is clear that the outlined algorithm should actually be considered rather a part of the upwards iteration of the program P instead of a separate algorithm that is done in addition to it.

The upwards iteration of a bf-EVALPSN program could be described as a two step process where first a given model is upwards iterated using T_P until a least model is found, then if needed the model is split into (hopefully) definite models from the earliest known indefinite bf-relation and the process is began anew from each definite model thus created.

It becomes quite clear that in a general case the algorithm is not in any way tractable and can cause us to recurse into a tree of nearly infinite splitting models, the number of which could be cubic to the size of the input ruleset. Yet this sort of input would only happen in the case where all of the processes can start and end at nearly any time and with little or no regard to each other. This sort of a system would essentially be a chaotic random system and the correct tool to describe such a system is something completely other than logic programming. Looking at the table of deduction rules given in appendix A, most of the indeterminate rules only define a set of 3 to 5 possible definite bf-relations and a large majority of the rules give definite relations. Thus it is clear that most of our programs should still be relatively easy to translate with the algorithm.

5. APPLYING THE SEMANTICS

In the previous chapter we created an algorithm for creating a perfect model, or more accurately models in the form of an automata, from a given bf-EVALPSN program P. In this chapter the algorithm will be used to model a four-way traffic intersection. We aim to show that the algorithm works and generates a readable, reasonable automaton from the program that can be used to easily verify the system's veracity and the completeness of the process control.

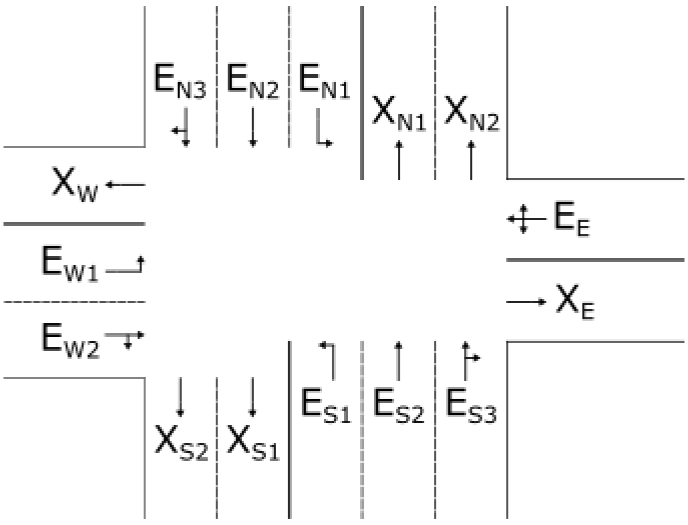
A traffic intersection is chosen as an example due to its general simplicity and ease of understanding without any deep context knowledge.

5.1 Four-Way Traffic Intersection

Let us imagine a four-way traffic intersection of two roads going North-South and East-West. Let the road towards north and south be five lanes, west be three lanes, and east be just two lanes wide. Inbound from the north are three lanes with the eastern lane used solely for turning east, the middle lane a forwards heading one and the western lane both for heading south and west. Inbound from the west are two lanes, the northern one turning north while the southern lane allows both driving forwards due east and turning due south. From the south there are three inbound lanes, one for turning right towards east and heading north, one lane only heading north, and one turning due west. Finally from the east there is only one inbound lane heading to all directions. The lanes left unmentioned are outgoing lanes with differing inbound lanes using them. These relationships will be covered later.

As can be seen from the figure 5.1, the intersection is quite complicated. With nine incoming lanes, six outgoing lanes and 14 different headings the intersection becomes hard to control by any finer methods than a strict round robin of entry-exit directions with extra turns for crosswise turning traffic. This is the usual way an intersection such as this is organised with North-South and East-West directions taking turns in using the intersection while in between crosswise traffic, for example coming from south and turning west, is given a turn.

Figure 5.1 *Four-way intersection*



Four-way traffic signal intersections have been the topic of extensive research due to their vital importance in urban traffic as well as their heavy effect on urban congestion and delay. Many alternative control systems have been put forth. Our attempt here is to firstly model the intersection. Second to that comes the aim of designing a new control logic for the traffic flow.

5.1.1 bf-EVALPSN Traffic Rules

We begin by modeling the basics of the intersection. First thing is to assign an atom to each lane. The lanes shall be assigned names by their direction from the centre of the intersection and a running index counted from the middle of the road. Additionally lanes are named differently with inbound or entering lanes denoted by E and outbound or exit lanes denoted by X.

Thus the inbound lanes are

1. E_{S1}, the west heading southern inbound lane,
2. E_{S2}, the north heading southern inbound lane,
3. E_{S3}, the north and east heading southern inbound lane,
4. E_E, the south, west and north heading eastern inbound lane,
5. E_{N1}, the east heading northern inbound lane,
6. E_{N2}, the south heading northern inbound lane,
7. E_{N3}, the south and west heading northern inbound lane,
8. E_{W1}, the north heading western inbound lane, and
9. E_{W2}, the east and south heading western inbound lane,

and the outbound lanes are similarly called

1. X_{S1}, the east-side southern outbound lane,
2. X_{S2}, the west-side southern outbound lane,
3. X_E, the eastern outbound lane,
4. X_{N1}, the west-side northern outbound lane,
5. X_{N2}, the east-side northern outbound lane, and
6. X_W, the south-side western outbound lane.

We shall define a literal $active(x, t)$ which, when true, means that at a time t a particular inbound or outbound lane x is being used and thus has active traffic flow. The literal is defined for the entry lanes E using the bf-EVALPSN *start* and $finish$ literals as follows:

$$\forall e \in E, n:\ active(x,t) : [(n,0),\alpha] \leftarrow start(x,t) : [(n,0),\alpha] \wedge finish(x,t) : [(0,n),\alpha] \tag{5.1}$$

For outbound lanes we leave the active status unbound from the *start* and $finish$ literals, as we will not be defining bf-relations for the exit lanes. Instead their active status will be defined by dependencies from inbound lanes using them.

5.1.2 Basic Lane Operation

For each lane we define a basic operational structure through dependencies. Outgoing lanes are active only when there is an incoming lane using that particular outgoing lane. Thus these lanes depend on incoming lanes.

Incoming lanes on the other hand are tied together so that lanes that move traffic in the same direction must be active at the same time. For example if there is one lane moving traffic south to north it stands to reason that at the same time the other south to north heading lane should be active.

Now beginning with a simple example we take the E_{S2} entry lane coming from the south, heading due north. This lane routes to the west-side northern exit lane X_{N1}. Thus our first dependency statement is

$$active(X_{N1}, t) : [(1,0),\alpha] \leftarrow active(E_{S2}, t) : [(1,0),\alpha]. \tag{5.2}$$

This is read as the outbound lane being used if the inbound lane is used. Here we

Table 5.1 *Basic lane operation rules*

Lane	**Rules**
E_{S1}, South to West	$active(X_{W2}, t) : [(1,0), \alpha] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
E_{S2}, South to North	$active(X_{N1}, t) : [(1,0), \alpha] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
E_{S3}, South to North and East	$active(X_{N2}, t) : [(1,0), \alpha] \leftarrow active(E_{S3}, t) : [(1,0), \alpha]$ $active(X_{E}, t) : [(1,0), \alpha] \leftarrow active(E_{S3}, t) : [(1,0), \alpha]$
E_{E}, East to All	$active(X_{S1}, t) : [(1,0), \alpha] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$ $active(X_{W1}, t) : [(1,0), \alpha] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$ $active(X_{N2}, t) : [(1,0), \alpha] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
E_{N1}, North to East	$active(X_{E}, t) : [(1,0), \alpha] \leftarrow active(E_{N1}, t) : [(1,0), \alpha]$
E_{N2} North to South	$active(X_{S1}, t) : [(1,0), \alpha] \leftarrow active(E_{N2}, t) : [(1,0), \alpha]$
E_{N3} North to South and West	$active(X_{S2}, t) : [(1,0), \alpha] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$ $active(X_{W2}, t) : [(1,0), \alpha] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$
E_{W1}, West to North	$active(X_{N1}, t) : [(1,0), \alpha] \leftarrow active(E_{W1}, t) : [(1,0), \alpha]$
E_{W2}, West to East and South	$active(X_{E}, t) : [(1,0), \alpha] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$ $active(X_{S2}, t) : [(1,0), \alpha] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$

are

Additionally, this mid-southern entry lane is must be bound together with the east-side southern entry lane. This particular lane heads towards north and east. Since both of these lanes head north they must run at the same time as we decided above. This relation will not, however, be coded into our basic lane operation rules but will instead be codified using bf-relations later.

With these basic rules defined we look into forming some forbiddances. Lanes that clearly cross paths should not be run at the same time. There are exceptions to this, for instance in this particular intersection we might allow the east entry lane to run at the same time with the western entry lanes, with cars simply waiting for their turn to turn crosswise over the intersection.

These crossing points are of vital importance in determining which lanes are hazardous to each other.

We shall write out the crossing points into a table for readability in table 5.2. In the table we mark with a light grey colour those lanes that despite crossing each other do not pose a threat to traffic safety. These mainly include lanes that merely converge from different directions to the same exit lane. These will definitely cause

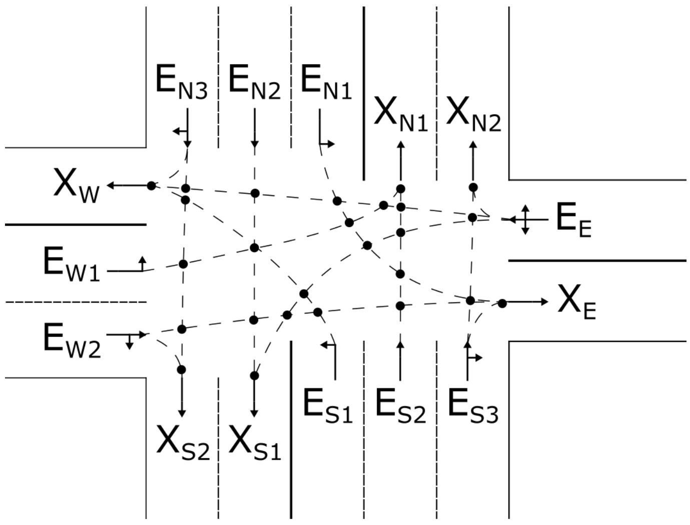


Figure 5.2 *Crossing points of the intersection*

Table 5.2 *Crossing points table*

	E_E	E_{N1}	E_{N2}	E_{N3}	E_{W1}	E_{W2}	E_{S1}	E_{S2}	E_{S3}
E_E	■	X	X	X	X	X	X	X	X
E_{N1}	X	■			X	X		X	X
E_{N2}	X		■		X	X	X		
E_{N3}	X			■	X	X	X		
E_{W1}	X	X	X	X	■		X	X	
E_{W2}	X	X	X	X		■	X	X	X
E_{S1}	X		X	X	X	X	■		
E_{S2}	X	X			X	X		■	
E_{S3}	X	X				X			■

some congestion on the exit lane but the convergence itself is not hazardous. We readily see that the only eastern entry lane crosses with every single other lane in the intersection. This is a clear warning sign of congestion for that lane.

With this table we are ready to write out our forbiddance rules. We will only forbid outright crossing of lanes now but omit the eastern entry lane at first due to need of special handling.

The eastern entry lane is problematic as a naïve set of forbiddance rules would forbid the lane from ever running with any other lane. This is definitely a possible choice for the intersection to be built upon and would then keep the eastern lane usage mostly congestion free and most importantly safe under all circumstances. However, this would come at a cost to the throughput of other lanes in the intersection. Here we choose to accept some risk and congestion on the eastern lane and allow the lane to run alongside some other lane that it crosses with in a less dangerous fashion.

Table 5.3 *Lane operation forbiddance rules*

Lane	Forbiddance Rules
$\boldsymbol{E_{N1}}$	$active(E_{W1}, t) : [(0,1), \beta] \leftarrow active(E_{N1}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{N1}, t) : [(1,0), \alpha]$
	$active(E_{S2}, t) : [(0,1), \beta] \leftarrow active(E_{N1}, t) : [(1,0), \alpha]$
	$active(E_{S3}, t) : [(0,1), \beta] \leftarrow active(E_{N1}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{N2}}$	$active(E_{W1}, t) : [(0,1), \beta] \leftarrow active(E_{N2}, t) : [(1,0), \alpha]$
	$active(E_{W2}, t) : [(0,1), \beta] \leftarrow active(E_{N2}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{N2}, t) : [(1,0), \alpha]$
	$active(E_{S1}, t) : [(0,1), \beta] \leftarrow active(E_{N2}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{N3}}$	$active(E_{W1}, t) : [(0,1), \beta] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$
	$active(E_{W2}, t) : [(0,1), \beta] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$
	$active(E_{S1}, t) : [(0,1), \beta] \leftarrow active(E_{N3}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{W1}}$	$active(E_{N1}, t) : [(0,1), \beta] \leftarrow active(E_{W1}, t) : [(1,0), \alpha]$
	$active(E_{N2}, t) : [(0,1), \beta] \leftarrow active(E_{W1}, t) : [(1,0), \alpha]$
	$active(E_{N3}, t) : [(0,1), \beta] \leftarrow active(E_{W1}, t) : [(1,0), \alpha]$
	$active(E_{S1}, t) : [(0,1), \beta] \leftarrow active(E_{W1}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{W2}}$	$active(E_{N2}, t) : [(0,1), \beta] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$
	$active(E_{N3}, t) : [(0,1), \beta] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$
	$active(E_{S1}, t) : [(0,1), \beta] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$
	$active(E_{S2}, t) : [(0,1), \beta] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$
	$active(E_{S3}, t) : [(0,1), \beta] \leftarrow active(E_{W2}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{S1}}$	$active(E_{N2}, t) : [(0,1), \beta] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
	$active(E_{N3}, t) : [(0,1), \beta] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
	$active(E_{W1}, t) : [(0,1), \beta] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
	$active(E_{W2}, t) : [(0,1), \beta] \leftarrow active(E_{S1}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{S2}}$	$active(E_{N1}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
	$active(E_{W2}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{S3}}$	$active(E_{N1}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
	$active(E_{E}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
	$active(E_{W2}, t) : [(0,1), \beta] \leftarrow active(E_{S2}, t) : [(1,0), \alpha]$
$\boldsymbol{E_{E}}$	$active(E_{N1}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
	$active(E_{N2}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
	$active(E_{N3}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
	$active(E_{S1}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
	$active(E_{S2}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$
	$active(E_{S3}, t) : [(0,1), \beta] \leftarrow active(E_{E}, t) : [(1,0), \alpha]$

This is the more commonly adopted approach with the completely crossing entry lane usually sharing the intersection with the opposite side entry lanes. This forces cars to wait in the intersection for crossing traffic, hindering the flow but in exchange doing away with a the east-reserved turn from the intersection usage.

5.1.3 Lane Operation Order

We move into the temporal order of lane operation. Before we've already set down rules to ensure that certain lanes cannot be used at the same time and that others will always run with each other. The next thing to consider is the order the lanes take in operating.

Given that our system should not, initially, make any hard choices on when eg. crosswise turning lanes should operate and instead leave that decision to be made using the forbiddance rules, it makes sense to define only that the bf-relation of turning lanes to their neighbouring lanes should be between paraconsistent before-after (lanes are active at the same time) and immediate after (turning lanes operate immediately after their neighbours). Thus we set

$$\begin{aligned} &R(E_{S2}, E_{S1}, T) : [(6,1), \alpha] \leftarrow \\ &R(E_{N2}, E_{N1}, T) : [(6,1), \alpha] \leftarrow \\ &R(E_{W2}, E_{W1}, T) : [(6,1), \alpha] \leftarrow \end{aligned} \tag{5.3}$$

where T is again the "terminal" time of the system, meaning that these bf-relations will eventually be fulfilled. Additionally we set the conjoined lanes' bf-relations:

$$\begin{aligned} &R(E_{S2}, E_{S3}, T) : [(6,6), \alpha] \leftarrow \\ &R(E_{N2}, E_{N3}, T) : [(6,6), \alpha] \leftarrow \end{aligned} \tag{5.4}$$

With regards to the other lanes we do not want to decide what order they will operate in. The program should allow all operation orders to be possible and only set some constraints on how the lanes operate with regards to each other. In our case we want to set the constraint that inbound directions operate immediately after each other or at the same time. To show this constraint in bf-EVALPSN we shall define two sets, one of inbound direction's activity finishing lanes G_f and one of their activity starting lanes G_s.

$$G_f := \{E_{S1}, E_E, E_{N1}, E_{W1}\} \tag{5.5}$$

$$G_s := \{E_{S2}, E_E, E_{N2}, E_{W2}\} \tag{5.6}$$

These sets are chosen based on the order of individual lanes operating order within the inbound direction. For example, E_{S2} is known to start (as is E_{S3}) at the very least at the same time or earlier than E_{S1} while the latter is known to finish as the last of three lanes coming from the south.

Now for each tuple (G_f, G_s) we want to set that if the bf-relation between the two lanes is not already known to be disjointly or immediately after, or paraconsistent before-after (the starting lane started and finished earlier or starts and finishes at the same time as the finishing lane) then their bf-relation should be either immediate or disjoint before.

$$\begin{aligned} \forall(E_X, E_Y) \in (G_f, G_s) :\ & R(E_X, E_Y, T) : [(0, 11), \alpha] \leftarrow \\ & \neg^* R(E_X, E_Y, T) : [(6, 6), \alpha] \wedge \neg^* R(E_X, E_Y, T) : [(11, 0), \alpha] \end{aligned} \tag{5.7}$$

With these rules we have now successfully joined each of our lane literals to one another through some bf-clause or another. The conjoining bf-clauses in (5.4) bind the ”secondary” (from the viewpoint of our rules, not from the intersections’) lanes together with the primary lane of the inbound direction, while the turning lanes are bound to the ”primary” lanes with the bf-clauses in (5.3), and finally the primary lanes are bound to one-another using the bf-relations from (5.7), possibly with a turning lane acting as an intermediary.

We are finally only missing a hint to start our bf-relation splitting. For this we use an existentially quantified rule we impose on the system. This rule acts to give the start of our program some orientation by defining some singular inbound direction’s activity starting lane that is, for all other activity starting lanes, either at most f-included paraconsistent before-after them:

$$\exists(E_X \in G_s)\forall(E_Y \in G_s) :\ (R(E_X, E_Y, T) : [(0, 5), \alpha]) \tag{5.8}$$

Now as G_s only contains four lanes, we have a much simpler start phase to our model

Table 5.4 *Initial events of the system*

Initial events
$start(E_{S2}, t_1), start(E_{S3}, t_1)$
$start(E_E, t_1)$
$start(E_{N2}, t_1), start(E_{N3}, t_1)$
$start(E_{W2}, t_1)$

splitting. Before this rule it might have seemed like we would need to try every single inbound lane as a potential ”starter” in our initial split. Now can begin building the automaton that models our four-way traffic intersection with a tractable amount of starting models.

5.1.4 Building the Automaton

As we now begin to build our automaton according to the section 4.2 our first step is to search for a least model using the immediate consequence operator T_P. Fortunately or unfortunately, this is very simple as the only rules we can gain come from the conjoined lanes and their turning lanes:

$$\begin{aligned} R(E_{S3}, E_{S1}, T) &: [(6,1), \alpha] \leftarrow \\ R(E_{N3}, E_{N1}, T) &: [(6,1), \alpha] \leftarrow \end{aligned} \tag{5.9}$$

All of our information is, essentially, behind indefinite bf-relations and as such we need to begin splitting the indefinite model into more definite models. To do this we begin with the hint given to us in (5.8). It gives us four alternative initial models, each choosing one inbound lane to be the earliest lane in the system to active. This then gives us the means to take the first step in the splitting.

Now for each starting lane it is possible that some other lane or lanes start at the same time. We will informally walk through these next.

For the southern entry lane it is possible that the west-turning southern lane, or the northern lanes (not including the turning lane), not both, start together with it. The western and eastern lanes are forbidden from being active according to the forbiddance rules in Table 5.3. The reverse works for the northern entry lane.

The eastern lane can start together with the western entry lane. The western entry lane can start together with either the eastern lane or the north turning western lane

Table 5.5 *Expanded initial events of the system*

Initial events
$start(E_{S2}, t_1), start(E_{S3}, t_1)$
$start(E_{S1}, t_1), start(E_{S2}, t_1), start(E_{S3}, t_1)$
$start(E_{S2}, t_1), start(E_{S3}, t_1), start(E_{N2}, t_1), start(E_{N3}, t_1)$
$start(E_E, t_1)$
$start(E_E, t_1), start(E_{W2}, t_1)$
$start(E_{N2}, t_1), start(E_{N3}, t_1)$
$start(E_{N1}, t_1), start(E_{N2}, t_1), start(E_{N3}, t_1)$
$start(E_{W2}, t_1)$
$start(E_{W1}), start(E_{W2}, t_1)$

but not both. These further expands our possible of initial events in the system.

Now we need to begin finishing the activity of our initially active lanes. The northern and southern inbound lanes, if started at the same time, likewise need to finish at the same time.

If a turning lane started with the northern or southern inbound lanes, those may finish before the neighbouring lane finishes, as it finishes or after it finishes. If the turning lane did not start at the same time, it may still start while the neighbouring lane is running (except for the forbiddances outlined above) and then may similarly finish before, at the same time, or after the neighbouring lane does.

The eastern and western lanes if started at the same time may only finish at the same time, both being inbound direction starting lanes that were bound to be either active at the exact same time, or immediately or disjointly before / after one-another. The western lane also has the turning lane that may start with it, or until immediately as it finishes. The ending time of that is similarly quite free with possibilities being before the western lane finishes, as it finishes or after it finishes.

Finally the northern and southern turning lanes, if they have not started yet, may start before the neighbouring lanes finish or as they finish. Thus our next first veritable set of actual definite bf-relations divined from the system become as shown in Tables 5.6, 5.7, and 5.8 in abbreviated format.

With this data we can proceed to sketch out the first states of our process. For these states we shall use the previously defined *active* literal to notate our states. We note that from the tables' data that a few basic state types crop up in the different possible paths that the system may take. These are:

1. A forward heading lane alone with a conjoined lane if one exists,

Table 5.6 *South starting, definite bf-relations*

Events	**Before-After Relations of Interest**
$t_1: \; st(E_{S2}), st(E_{S3})$ $t_2: \; fi(E_{S2}), fi(E_{S3}), st(E_{S1})$ $t_3: \; fi(E_{S1})$	$R(E_{S2}, E_{S1}) : (1, 11)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 12)$
$t_1: \; st(E_{S2}), st(E_{S3})$ $t_2: \; st(E_{S1})$ $t_3: \; fi(E_{S2}), fi(E_{S3})$ $t_4: \; fi(E_{S1})$	$R(E_{S2}, E_{S1}) : (2, 10)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 12)$
$t_1: \; st(E_{S2}), st(E_{S3})$ $t_2: \; st(E_{S1})$ $t_3: \; fi(E_{S2}), fi(E_{S3}), fi(E_{S1})$	$R(E_{S2}, E_{S1}) : (3, 9)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 11)$
$t_1: \; st(E_{S2}), st(E_{S3})$ $t_2: \; st(E_{S1})$ $t_3: \; fi(E_{S1})$ $t_4: \; fi(E_{S2}), fi(E_{S3})$	$R(E_{S2}, E_{S1}) : (4, 8)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 11)$
$t_1: \; st(E_{S1}), st(E_{S2}), st(E_{S3})$ $t_2: \; fi(E_{S1}), fi(E_{S2})$ $t_3: \; fi(E_{S3})$	$R(E_{S2}, E_{S1}) : (5, 7)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 12)$
$t_1: \; st(E_{S1}), st(E_{S2}), st(E_{S3})$ $t_2: \; fi(E_{S1}), fi(E_{S2}), fi(E_{S3})$	$R(E_{S2}, E_{S1}) : (6, 6)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 11)$
$t_1: \; st(E_{S1}), st(E_{S2}), st(E_{S3})$ $t_2: \; fi(E_{S3})$ $t_3: \; fi(E_{S1}), fi(E_{S2}),$	$R(E_{S2}, E_{S1}) : (7, 5)$ $E_X \in G_s: \; R(E_{S2}, E_X) : (0, 11)$
$t_1: \; st(E_{S2}), st(E_{S3}),$ $st(E_{N2}), st(E_{N3})$ $t_2: \; fi(E_{S2}), fi(E_{S3}), fi(E_{N2}),$ $fi(E_{N3}), st(E_{S1}), st(E_{N1})$ $t_3: \; st(E_{S1}), fi(E_{N1})$	$R(E_{S2}, E_{S1}) : (1, 11)$ $R(E_{S2}, E_{N2}) : (6, 6)$ $R(E_{S1}, E_{N1}) : (6, 6)$ $E_X \in G_s - \{E_{N2}\}: \; R(E_{S2}, E_X) : (0, 12)$ $E_X \in G_s - \{E_{S2}\}: \; R(E_{N2}, E_X) : (0, 12)$

2. A forward heading lane with a neighbouring turning lane,
3. A forward heading lane with the opposite forward heading lane(s),
4. A turning lane alone,
5. A turning lane with the opposite turning lane,

As an example the second item in table 5.6 can be divided into three states: $S_{active(E_{S2}),active(E_{S3})}$ which is a forward heading lane with a conjoined lane, then $S_{active(E_{S1}),active(E_{S2}),active(E_{S3})}$ which is a forward heading lane with a conjoined lane and neighbouring turning lane, then finally $S_{active(E_{S1})}$ which is a turning lane alone.

If we regard our current knowledge of the system as ”final”, then we may write out the general transitions allowed between the aforementioned general states under some circumstances. Of these general states, number 1 can transition to number 2 or 4, or may be a ”final” state. Number 2 may transition into number 1 or 4, or

Table 5.7 *North starting, definite bf-relations*

Events	**Before-After Relations of Interest**
$t_1: st(E_{N2}), st(E_{N3})$ $t_2: fi(E_{N2}), fi(E_{N3}), st(E_{N1})$ $t_3: fi(E_{N1})$	$R(E_{N2}, E_{N1}): (1, 11)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 12)$
$t_1: st(E_{N2}), st(E_{N3})$ $t_2: st(E_{N1})$ $t_3: fi(E_{N2}), fi(E_{N3})$ $t_4: fi(E_{N1})$	$R(E_{N2}, E_{N1}): (2, 10)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 12)$
$t_1: st(E_{N2}), st(E_{N3})$ $t_2: st(E_{N1})$ $t_3: fi(E_{N1}), fi(E_{N2}), fi(E_{N3})$	$R(E_{N2}, E_{N1}): (3, 9)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 11)$
$t_1: st(E_{N2}), st(E_{N3})$ $t_2: st(E_{N1})$ $t_3: fi(E_{N1})$ $t_4: fi(E_{N2}), fi(E_{N3})$	$R(E_{N2}, E_{N1}): (4, 8)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 11)$
$t_1: st(E_{N1}), st(E_{N2}), st(E_{N3})$ $t_2: fi(E_{N2}), fi(E_{N3})$ $t_3: fi(E_{N1})$	$R(E_{N2}, E_{N1}): (5, 7)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 12)$
$t_1: st(E_{N1}), st(E_{N2}), st(E_{N3})$ $t_2: fi(E_{N1}), fi(E_{N2}), fi(E_{N3})$	$R(E_{N2}, E_{N1}): (6, 6)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 11)$
$t_1: st(E_{N1}), st(E_{N2}), st(E_{N3})$ $t_2: fi(E_{N1})$ $t_3: fi(E_{N2}), fi(E_{N3})$	$R(E_{N2}, E_{N1}): (7, 5)$ $E_X \in G_s: R(E_{N2}, E_X): (0, 11)$

be may be a ”final” state. Number 3 may only transition into number 5, or also number 4 in a sense (east-west combined traffic transitions to only western turning lane). Number 4 is only ever a ”final” state, as is number 5.

We shall next outline the different types of state transitions the system holds. Each type of state transition is given a natural language explanation of what the transition corresponds with in the intersection. In the following listing F shall stand for the currently understood ”final” state of the system.:

$1 \to 2:$ The forward heading traffic is joined by the neighbouring turning lane. The turning lane must not have ran already.

$1 \to 4:$ The forward heading traffic ceases in favour of the neighbouring turning lane. The turning lane must not have ran already.

$1 \to F:$ The forward heading traffic ceases. The turning lane must have already ran, and thus the system reaches a final position.

$2 \to 1:$ The turning lane traffic ceases, while forward heading traffic keeps going. The turning lane has thus ran.

$2 \to 4:$ The forward heading traffic ceases, while turning traffic keeps going. After the

Table 5.8 *East, West starting, definite bf-relations*

Events	**Before-After Relations of Interest**
$t_1: st(E_E)$ $t_2: fi(E_E)$	$E_X \in G_s: R(E_E, E_X): (0, 11)$
$t_1: st(E_E), st(E_{W2})$ $t_2: fi(E_E), fi(E_{W2}), st(E_{W1})$ $t_3: fi(E_{W1})$	$R(E_E, E_{W2}): (6, 6), R(E_{W2}, E_{W1}): (1, 11)$ $E_X \in G_s: R(E_E, E_X): (0, 12)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 12)$
$t_1: st(E_{W2})$ $t_2: fi(E_{W2}), st(E_{W1})$ $t_3: fi(E_{W1})$	$R(E_{W2}, E_{W1}): (1, 11)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 12)$
$t_1: st(E_{W2})$ $t_2: st(E_{W1})$ $t_3: fi(E_{W2})$ $t_4: fi(E_{W1})$	$R(E_{W2}, E_{W1}): (2, 10)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 12)$
$t_1: st(E_{W2})$ $t_2: st(E_{W1})$ $t_3: fi(E_{W1}), fi(E_{W2})$	$R(E_{W2}, E_{W1}): (3, 9)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 11)$
$t_1: st(E_{W2})$ $t_2: st(E_{W1})$ $t_3: fi(E_{W1})$ $t_4: fi(E_{W2})$	$R(E_{W2}, E_{W1}): (4, 8)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 11)$
$t_1: st(E_{W1}), st(E_{W2})$ $t_2: fi(E_{W2})$ $t_3: fi(E_{W1})$	$R(E_{W2}, E_{W1}): (5, 7)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 12)$
$t_1: st(E_{W1}), st(E_{W2})$ $t_2: fi(E_{W1}), fi(E_{W2})$	$R(E_{W2}, E_{W1}): (6, 6)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 11)$
$t_1: st(E_{W1}), st(E_{W2})$ $t_2: fi(E_{W1})$ $t_3: fi(E_{W2})$	$R(E_{W2}, E_{W1}): (7, 5)$ $E_X \in G_s: R(E_{W2}, E_X): (0, 12)$

turning lane ceases the system will reach a final position.

$2 \to F$: Both the forward heading and turning traffic ceases. The system has reached a final position.

$3 \to 5$: The forward heading traffic on both sides ceases, giving way to turning lanes on both sides. If only one side has a turning lane, this transition is effectively a $3 \to 4$ transition.

$4 \to F$: The turning lane traffic ceases. The forward heading lane must have already ran, and thus the system reaches a final position.

$5 \to F$: The turning lane traffic on both sides ceases. The forward heading lanes on both sides must have already ran, and thus the system reaches a final position.

From this we can quite easily write out a state graph for the possible states of the system thus far. This graph is shown in 5.3 with the state transition types shown.

On the graph the diamonds show that the joined state is one of the "final" states that we have so far

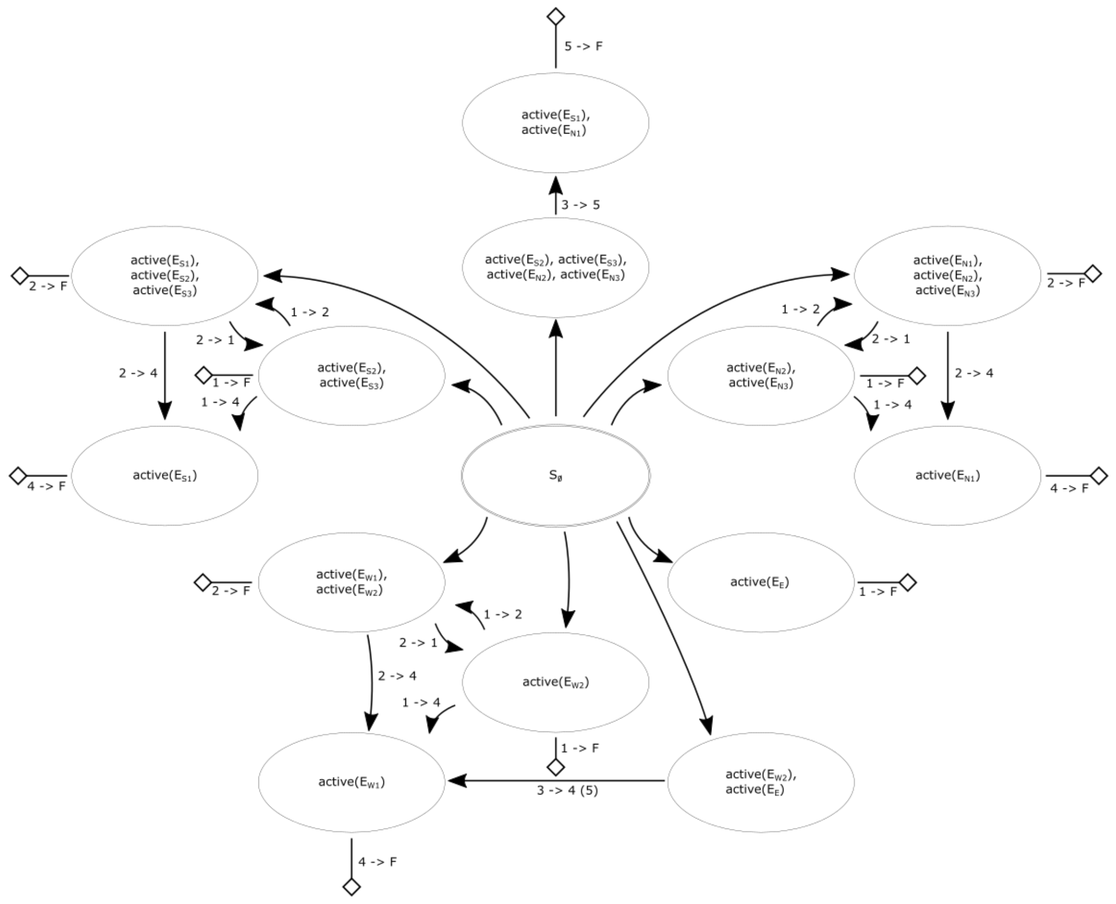

5 -> F
active(E_S1), active(E_N1)
3 -> 5
active(E_S2), active(E_S3), active(E_N2), active(E_N3)
active(E_S1), active(E_S2), active(E_S3)
2 -> F
1 -> 2
2 -> 1
2 -> 4
1 -> F
1 -> 4
active(E_S2), active(E_S3)
active(E_S1)
4 -> F
active(E_N1), active(E_N2), active(E_N3)
active(E_N2), active(E_N3)
active(E_N1)
S_∅
active(E_W1), active(E_W2)
active(E_W2)
active(E_E)
active(E_W2), active(E_E)
active(E_W1)
3 -> 4 (5)

Figure 5.3 *State graph of the four-way intersection*

Although one would think that there is still a lot of work to do in decrypting the bf-EVALPSN program into all the possible states of the automaton, it turns out we have actually already calculated all the states. A critical look at the program shows that although the order of the states does have dependencies in the states that the program has already encountered, the actual contents of the states do not change. Effectively the "final" states mentioned above can be understood as allowing a re-transition from the zero state $S_\emptyset$ to such a state for which none of the inbound lanes active in that state have yet ran during the course of the program. Or if we wish to rid of the requirement to run all lanes ones during one "round" of the program, we could rewrite some of the bf-EVALPSN clauses in the program to truly finish once a "final" state is reached. In this case it would be useful to also bind the choice of the first state transition (from the initial state) to some sensory input with eg. the aim of choosing the most congested lane.

With the state graph and the tables of state data it is now possible to do upwards iteration to resolve various questions regarding the various states of the program. As an example, it would be possible to resolve which exit lanes are used in each of the different states using the previously defined *active* relation of exit lanes to certain input lanes. Or if we defined a new *incoming* functional literal to denote which inbound lanes use a given outbound lane, we could use the states to quickly and easily determine for each state if some outbound lane is being used by dangerously many inbound lanes at the same time.

We leave the actual upwards iteration of individual states out of this thesis as the work is mainly mechanical applying of the transitive bf-relation deduction rules, program clauses and basic transitive bf-relation rules onto the various states. As earlier mentioned, upwards iteration to create a least model is not a new or unsolved problem. Rather the focus is and has been to take bf-relations and map the information contained therein into a more readable format, formal automaton states to be exact.

6. CONCLUSIONS

The aim of this thesis has been to show bf-EVALPSN to be a powerful annotated logic program for use in process control modelling. The system has previously been used to do active process control and verification successfully but the modelling of these control systems has been outside the scope of the program.

In the previous chapters we have shown the theoretic frame of reference within which bf-EVALPSN works, outlining the focal results of using annotated logics to do logic programming. Further the complete definition of bf-EVALPSN with its rather complex and vast vector annotation system has been shown in a level of detail that has not been seen in likely any published paper on the system. This has been thanks to direct conversations with professor Kazumi Nakamatsu, the creator of bf-EVALPSN.

Arguably bf-EVALPSN has been shown here to be a capable, albeit somewhat heavy or even unwieldy tool for logic programming model building. At the same time some problems in the system have come up. One problem with model building using bf-EVALPSN was the question of annotation representation. With online process verification using bf-EVALPSN one question has largely gone unasked. That is, how to represent the value of A when the following two bf-clauses exist in the system:

$$\begin{aligned} A &: [(1,0),\alpha] \leftarrow \\ A &: [(0,1),\beta] \Leftarrow \end{aligned} \tag{6.1}$$

That is A is true but is obligated to be false. The literal representation format would suggest the interpretation of A in this case to be

$$I(A) = [(1,1),*_1] \tag{6.2}$$

but this naïve combination of the two values would cause all sorts of weird effects.

For instance now the query

$$\leftarrow A : [(0, 1), \alpha] \tag{6.3}$$

would be answered in the affirmative. The correct answer for the complete representation of annotations in bf-EVALPSN was the 8-tuple representation shown in (3.2). Naturally as annotated logic programming only deals with well-formed annotations, these complete annotations are not needed at all normally. However, when modelling is the aim the actual representation of data as annotations becomes very important. At the very least if the theoretical basis of the system is to be outlined then it is needed. This information has likely not been published in any article on bf-EVALPSN before and as such finding the information required direct communications with the author.

Another problem with the system is of course the problem that this thesis took to resolve. That is the question of mapping integer time points into some more easily readable format. This has not been a relevant question posed to bf-EVALPSN until now. As such the creation of even a passable imitation of an algorithm for translating the bf-relations into distinct states needed to be done completely from scratch. The result is definitely not pretty and the algorithm itself is one only in the broadest of definitions. A proper mathematical algorithm would require much more rigorous research into the basis of bf-EVALPSN as well as a proper, formal definition of all the steps and eg. hints given to the algorithm by the program.

Nevertheless, as the author of this algorithm I believe it to be a good first step to a formal definition of bf-EVALPSN as a modellable logic program. The intuitionist logic that builds the algorithm are at least well-founded in the theory of stratifiable logic programs, perfect model semantics, automata theory, and the transitive bf-relation deduction rules. Which brings up the next problem point that this thesis solves in bf-EVALPSN.

The bf-relation inference rules given in section 3.3 are only written from the viewpoint of a machine looking at a bf-EVALPSN program as it is running. No deduction rules for bf-relations have previously been laid out as far as the author knows. Calculating out all the possible transitive deduction rules then given in section 4.1 was then a relatively arduous task that needed to be done on the road to model building using bf-EVALPSN.

At the end of all this, the system for deduction and model building in bf-EVALPSN created in this thesis is one more tool in the already massive system that is the paraconsistent (or more accurately non-alethic), deontic, temporal, annotated logic program named Before-After Extended Vector Annotated Logic Program with Strong Negation, or bf-EVALPSN for short. The shortcomings of this new tool are many and the result of no other than the author, yet it would be hasty to reject it altogether on that basis. Rather, this new tool should be welcomed as an opportunity to work to make bf-EVALPSN into a more robust, rigorously defined logic program. The program is already a forerunner into the complex world of multi-paradigm logic programs. Any and all progress made towards the hardening and the expansion of the program's capabilities should be met with enthusiasm.

BIBLIOGRAPHY

[1] J. M. Abe, S. Akama, and K. Nakamatsu, *Introduction to Annotated Logics: Foundations for Paracomplete and Paraconsistent Reasoning (Intelligent Systems Reference Library).* Springer, 2015. [Online]. Available: https://www.amazon.com/Introduction-Annotated-Logics-Paracomplete-Paraconsistent/dp/331917911X?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=331917911X

[2] N. da Costa, V. Subrahmanian, and C. Vago, "The paraconsistent logic $p\tau$," *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, vol. 37, pp. 139–148, 1991.

[3] C. Hewitt, "Planner: A language for proving theorems in robots," in *International Joint Conference on Artificial Intelligence*, 1969.

[4] R. Kowalski, "Predicate logic as programming language," in *IFIP Congr.*, vol. 74, 01 1974, pp. 569–574.

[5] K. Nakamatsu, "Application of paraconsistent annotated logic program evalpsn to intelligent control/safety verification," in *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 09 2017, pp. 113–113.

[6] K. Nakamatsu, J. Abe, and S. Akama, *Paraconsistent Annotated Logic Program EVALPSN and Its Applications.* Springer, 06 2015, vol. 94, pp. 39–85.

[7] ——, "Paraconsistent annotated logic program evalpsn and its application to intelligent control," in *New Approaches in Intelligent Control*, 06 2016, vol. 107, pp. 337–401.

[8] K. Nakamatsu, J. Abe, and A. Suzuki, "Annotated semantics for defeasible deontic reasoning," in *Rough Sets and Current Trends in Computing, Second International Conference (RSCTC)*, 01 2000, pp. 470–478.

APPENDIX A. TABLE OF TRANSITIVE BEFORE AFTER RELATION DEDUCTION RULES

R(P2, P3)	R(P1, P2) db (0,12)	mb (1,11)	jb (2,10)	sb (3,9)	ib (4,8)	fb (5,7)	pba (6,6)	fa (7,5)	ia (8,4)	sa (9,3)	ja (10,2)	ma (11,1)	da (12,0)
db (0,12)	db (0,12)	db (0,12)	db (0,12)	db (0,12)	be (0,8)	db (0,12)	db (0,12)	be (0,8)	db (0,12)	db (0,12)	be (0,8)	be (0,8)	⊥ (0,0)
mb (1,11)	db (0,12)	db (0,12)	db (0,12)	mb (1,11)	be- (2,8)	db (0,12)	mb (1,11)	be- (2,8)	db (0,12)	mb (1,11)	be- (2,8)	fba (5,5)	af (8,0)
jb (2,10)	db (0,12)	db (0,12)	be* (0,10)	jb (2,10)	be- (2,8)	be* (0,10)	jb (2,10)	be- (2,8)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jba (2,2)	af- (8,2)	af (8,0)
sb (3,9)	db (0,12)	db (0,12)	be* (0,10)	sb (3,9)	ib (4,8)	be* (0,10)	sb (3,9)	ib (4,8)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	sba (3,9) ∨ (6,6) ∨ (9,3)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ma (11,1)	da (12,0)
ib (4,8)	db (0,12)	db (0,12)	be (0,8)	ib (4,8)	ib (4,8)	be (0,8)	ib (4,8)	ib (4,8)	⊥ (0,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	da (12,0)	da (12,0)
fb (5,7)	db (0,12)	mb (1,11)	jb (2,10)	jb (2,10)	be- (2,8)	fb (5,7)	fb (5,7)	fba (5,5)	ia (8,4)	ia (8,4)	af- (8,2)	af- (8,2)	af (8,0)
pba (6,6)	db (0,12)	mb (1,11)	jb (2,10)	sb (3,9)	ib (4,8)	fb (5,7)	pba (6,6)	fa (7,5) (7,5)	ia (8,4)	sa (9,3)	ja (10,2)	ma (11,1)	da (12,0)
fa (7,5)	db (0,12)	mb (1,11)	be- (2,8)	ib (4,8)	ib (4,8)	fba (5,5)	fa (7,5) (7,5)	fa (7,5) (7,5)	af (8,0)	af* (10,0)	af* (10,0)	da (12,0)	da (12,0)
ia (8,4)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jba (2,2)	ia (8,4)	ia (8,4)	af- (8,2)	ia (8,4)	ia (8,4)	af- (8,2)	af- (8,2)	af (8,0)
sa (9,3)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	sba (3,9) ∨ (6,6) ∨ (9,3)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ia (8,4)	sa (9,3)	ja (10,2)	ia (8,4)	sa (9,3)	ja (10,2)	ma (11,1)	da (12,0)
ja (10,2)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	jb ∨ fb ∨ ia (2,10) ∨ (5,7) ∨ (8,4)	jba (2,2)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	af- (8,2)	ja (10,2)	ja (10,2)	af (8,0)	af* (10,0)	af* (10,0)	da (12,0)	da (12,0)
ma (11,1)	be* ∨ fb ∨ ia (0,10) ∨ (5,7) ∨ (8,4)	sba (3,9) ∨ (6,6) ∨ (9,3)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ib ∨ fa ∨ ja (4,8) ∨ (7,5) ∨ (10,2)	ma (11,1)	ma (11,1)	ma (11,1)	da (12,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)
da (12,0)	⊥ (0,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	ib ∨ fa ∨ af* (4,8) ∨ (7,5) ∨ (10,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)	da (12,0)

Legend

be*	(0,10)
be-	(2,8)
af*	(10,0)
be-	(8,2)
jba	(2,2)
sba	(3,9) ∨ (6,6) ∨ (9,3)
fba	(5,5)