TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Lemma Generation for Induction-based Proof Rules

Master-Thesis

of

Felix Hasselwander

November 15, 2019

Technische Universität Kaiserslautern,
Department of Computer Science,
67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Schneider
M.Sc. Martin Köhler

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Lemma Generation for Induction-based Proof Rules" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 15.11.2019

_____

Felix Hasselwander

Abstract

The formal verification of temporal logic properties of reactive systems is an important part of modern development processes. Since the introduction of the *property directed reachability* (PDR) method in 2011, this induction-based proof method significantly increased the performance of software and hardware safety model checking. Earlier this year, inference proof rules have been published, which provide an abstraction from the PDR method, and suggest that induction-based proof methods are not limited to safety properties. Those inference rules rely on *intermediate lemmas* and allow inferring a proof from a provided intermediate lemma, given that it satisfies a set of assertions. Those intermediate lemmas have to exist if the desired property holds since the proof rules are *complete*. Furthermore, the intermediate lemmas are sufficient as proof as those rules are *correct*. In this thesis, I present properties as well as the structure of those intermediate lemmas. Most notably, I discovered and proved that the set of intermediate lemmas for greatest fixpoint proofs has a lattice structure. In addition, it is discussed why intermediate lemmas of least fixpoints do not have a lattice structure and thus why this method is not applicable to least fixpoints. Furthermore, a method for generating all intermediate lemmas for the greatest fixpoint proof is presented, which is a consequence of the lattice property. Within this thesis, the generation of intermediate lemmas is implemented, tested, documented and depicted in F#.

Zusammenfassung

Die formale Verifikation von temporalen Eigenschaften von reaktiven Systemen ist ein wichtiger Schritt in modernen Entwicklungsprozessen. Seit der Veröffentlichung der *property directed reachability* (PDR) Methode in 2011, hat diese, auf dem Konzept der Induktion beruhende Beweismethode, deutlich die Effizienz und Möglichkeiten zur Verifikation von *safety* Eigenschaften in Hardware- und Software-Systemen gesteigert. Anfang dieses Jahres wurden weitere Beweisregeln veröffentlicht, welche eine Abstraktion von der PDR Methode erlauben und diese legen nahe, dass induktions-basierte Beweismethoden nicht auf *safety* Eigenschaften eingeschränkt sind. Diese Beweisregeln ermöglichen einen Beweis mittels Mengen von Zuständen. Diese Mengen müssen gewisse Zusicherungen erfüllen und werden *Zwischen-Lemma* genannt. Dieser Beweis zeigt dann eine gewünschte Eigenschaft. Aufgrund der gezeigten Korrektheit und Vollständigkeit der Beweisregeln, müssen diese Zwischen-Lemma existieren. In dieser Arbeit werden verschiedene neu entdeckte Eigenschaften und die Struktur der Zwischen-Lemmata vorgestellt und diskutiert. Besonders hervorzuheben sind hierbei die *lattice* Eigenschaft der Menge von Zwischen-Lemmata für Beweise von größten Fixpunkten, aber auch warum die Zwischen-Lemma von kleinsten Fixpunkten keine solche Struktur haben. Des Weiteren wird eine daraus folgende Methode zum Bestimmen aller Zwischen-Lemma für Beweise von größten Fixpunkten vorgestellt. Diese Methode wurde implementiert, getestet, dokumentiert und dargestellt in F#.

Contents

List of Figures

List of Lemmas, Theorems and Definitions

1. Introduction

In 1993 Intel released the Pentium P5 as well as the Pentium P54C processor for private use. One and a half years later, in November 1994, Prof. Thomas Ray Nicely from the Lynchburg College published that this processor was flawed. He found that given two special floating-point numbers, the result of the division was imprecise at the thirteenth bit. The cause of the error was a faulty SRT-division unit using a lookup table with five wrong entries. When this error was investigated, Intel published that this error occurs for 3 10^{37} out of $2,28$ 10^{47} potential pairs of dividend and divisor, meaning it is very seldom. However, Intel was forced to react to this error and offered a refund for all affected processors, leading to a financial loss of 475 million dollars as well as a waste of resources.[Pen19]

This event had a massive impact on the design process of hardware systems. Note that only for a very small sample-size of dividend and divisor pairs this error occurred, and thus, it is very doubtful to find such an error by simple testing. However, it is of high importance to avoid flaws like this in processors. This is why processors nowadays have to be verified for functionality against some specifications. By that, it is not only tested that the piece of hardware works, but it is actually proven that it will behave for every input as specified and thereby avoiding errors.

Another important field where verification is necessary is in the automotive industry. This is because autonomous driving is rising in popularity, and the development in that area is progressing rapidly in recent times. However, it is not sufficient that a car is tested for correct function since a unique traffic situation could arise in the real world, for which no testing was done. Any car should behave correctly despite that. Note that a failure in an unpredicted and thus untested situation could lead to fatal consequences. To this end, important elements, as well as the behaviour of the car, have to be formally verified for the correct function.

Currently, there exist two different approaches on how to verify autonomous behaviour systems - the *o ine* as well as the *online* approach. The offline approach is described, for example, in [FDW13]. For this approach, the system is analyzed before deployment by constructing a model for the autonomous agent, as well as model its environment. This model is then verified by the means of formal verification and model checking to guarantee correct behaviour. However, this yields the problem that there still could be impacts from the environment which have not been predicted when generating the models, and furthermore, each model is only a simplification of the real world and thus an imprecise image of the real world.

Online verification was introduced by [GA18] in 2018 to overcome problems

arising due to these particular simplifications. Instead of constructing and deriving models beforehand, the autonomous vehicle is verified during operation, thus online. This way, even the behaviour in unique traffic situations can be verified on the fly and thus guarantee the safety of the vehicle. To this end, the current environment is modelled on the fly with all current nearby traffic participants, and the behaviour in the current situation is formally verified. While the initial model for verification is only a coarse abstraction of the current situation to guarantee real-time functionality, this model is then incrementally refined.[1]

Both approaches have one thing in common. In the end, some property is formally verified to hold against a certain model. These models are typically finite state automatons consisting of a finite amount of states and transitions connecting the states with each other (for example, in [GA18] and [FDW13]). A typical property that is verified is a so-called *safety property*. Thus some condition has to hold in all states which are potentially reachable, for example, that a car should always be safe or opposite that the car should never harm someone.

In 2011 a method named *property directed reachability* (PDR) was proposed for model checking such safety properties. Thus given a finite state automaton this method provided a way to prove that all states reachable in this automaton satisfy the desired property. While this proposed method was highly efficient in practice and beat its competitors in model checking competitions such as HWMCC'10, this proposed method was limited to verifying safety properties.

The principle of this PDR method is the following: A set of states is iteratively refined by removing reachable states which violate the desired property until a fixpoint is reached. As soon as this is the case, this set is inductive, thus closed under the transition relation and an overapproximation of the reachable states. By that, the desired property is proven to hold on all reachable states, since it already holds on an overapproximation of the reachable states. The reason why this method is so efficient is that computationally expensive calculation of reachable states is avoided.

This method can be abstracted to a proof rule to verify that a given state set is indeed a proof for the desired property. This is the case if the given state set satisfies a set of assertions, in which case it will be considered an *intermediate lemma*. The first assertion is, that this set has to be closed under the transition relation. Thus any transition from a state inside this set is not allowed to lead to a state outside of this set. Furthermore, it has to contain all initial states of the structure against which the property is verified and all states of this set have to satisfy the desired property. If those three assertions hold, the given set is indeed a proof and is called a *lemma*[2]. By the PDR method, such a lemma is provided. However there are potentially many different lemmas to prove a single safety property on a specific structure, and other methods of

[1] More details are found in [GA18].

[2] Definition according to [Hig98]: "A *lemma* is an auxiliary result-a stepping stone towards a theorem." Definition according to [Lem19]: "A *helping theorem* or lemma (plural lemmas or lemmata) is a proven proposition which is used as a stepping stone to a larger result rather than as a statement of interest by itself."

deriving lemmas may exist.

Furthermore, the PDR method is limited to proving safety properties solely. However, there are further temporal properties that are relevant for verification, for example, a *liveness property*. A liveness property has again some desired property. Contrarily to a safety property, it has not to hold at any time. However, from any initial state reachability of a state fulfilling this property is required within a finite amount of time.

In 2019, a publication[3] was released in which proof rules for all temporal properties where given. The goal of this publication was to pave the way for methods similar to PDR, but for proving the remaining temporal properties.

1.1. Problem

This leads to the foundation of this thesis. All discovered proof rules have in common that they verify a given set of states as a proof for some property. However, it is not trivial to find those sets of states, and so far, only the PDR method is known to do this efficiently for a special set of states. Thus it is desired to find and prove properties of the intermediate lemmas of the given proof rules, in order to reason about the generation of those lemmas. Furthermore, by knowing additional properties of these lemmas, this might close the gap between different fields of computer science, as well as mathematics and thus, provide further insight into the generation of those lemmas.

Thus the core question of this thesis is: What are the properties of the intermediate lemmas of proof rules, and how are they generated?

1.2. Results

To answer this question, I discovered that the set of intermediate lemmas for safety property proofs forms a lattice structure. This insight does not only hold in the context of safety properties but greatest fixpoint properties in general. However, I also found out that this is not applicable in the case of least fixpoints. This allows concluding that there exists a weakest and a strongest intermediate lemma. Furthermore, in the case of safety properties, the strongest intermediate lemma corresponds to the reachable states, which in turn is a least fixpoint property. In addition, this lattice property allows to derive a method for generating all intermediate lemmas of greatest fixpoint CTL properties.

1.3. Outline

This thesis is structured as follows: In the second chapter, prerequisites are given, and information necessary for the understanding of the rest of this thesis are provided. In the third chapter, the current state of research is described, and the most important publications for this thesis are discussed. The fourth

[3] Further details are found in [KS19].

chapter contains the findings I have acquired while working on this thesis, especially in regards to the aforementioned core question. I have implemented some of those findings and this implementation is described in the fifth chapter, while in the sixth chapter a conclusion and outlook are given.

2. Preliminaries

The contents of this chapter provide fundamentals as well as crucial definitions which are important for this thesis. The comprised information includes background knowledge for some lesser experienced reader. On the other hand, this chapter presents terms, patterns and definitions used in the following chapters. The terms of propositional logic and predicate logic are addressed in sections 2.1 and 2.2. In section 2.3 a brief introduction to state transition systems is presented and in sections 2.4 and 2.6 the mathematical foundations of lattice theory and fixpoint calculation are introduced. In addition, the sections 2.5 and 2.7 provide insight into temporal logic. The definitions in this chapter are based on [Sch19b] and [Sch19a], if not stated otherwise, while some could also be found in [Sch13] for further reading.

2.1. Propositional Logic

When talking of propositional logic, one typically refers to formulas. The most fundamental building block of such a formula in propositional logic (often also referred to as boolean logic or two-valued logic) are *variables*, which could be assigned one of two truth values, namely true or false. Often in computer science, false is abbreviated by 0, while true is abbreviated by 1. Additional to these variables, there is a set of operators that is used to connect variables to formulas. A special operator of those is the negation ($:$), which flips the truth value from true to false or from false to true.

Connecting variables with operators yields different formulas with specific names. By definition any kind of variable that is either negated or not is called a *literal*. Literals solely connected by the *disjunction* ($\_$) are *Clauses*, whilst literals only connected with *conjunctions* ($\wedge$) are *Co-Clauses* or *cubes*. A crucial part of understanding propositional logic is the use of assignments φ (or valuations) that assign truth values to different variables. The *evaluation* of the different operators under a certain assignment is best shown in a truth table, which lists the different interpretations of the operators for all assignments.

Note that an *implication* ($!$) only evaluates to false if the left-hand side of the implication is true and the right-hand side is false. Thus if the left-hand side holds, the right-hand side has to hold as well. Furthermore to make formulas more readable and reducing the amount of necessary brackets, a *precedence order* is defined. This order defines how strong an operator binds. This precedence order is ($:$ $\wedge$ $\_$ $!$,). Thus a formula $a \_ :b \wedge c$ is an abbreviation of $(a \_ ((:b) \wedge c))$.

Formulas build like this could evaluate to different values based on the given

a	b	$a \wedge b$	$a \vee b$	$a \rightarrow b$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

Table 2.1.: *A truth table for the most common propositional logic operators. The first two columns show the assigned value for the variable a and b for each specific row. The third column shows the conjunction ^ , the fourth the disjunction _ and the fifth the implication ! .*

valuation, but for some formulas, the valuation does not make a difference. A formula which will always evaluate to true is called *valid* or a *tautology*. Usually, we use such formulas as *Axioms*, which are *Hypotheses* that could be proven to be correct in all *environments*. The opposite is a formula which could never be *satisfied*, which means that there does not exist a valuation under which the formula will evaluate to true. An example for a *contradiction* would be:

$$x \wedge \neg x \tag{2.1}$$

while an example for a tautology would be:

$$x \wedge \neg x \rightarrow y \tag{2.2}$$

Let's consider both examples. In 2.1 we try to satisfy a variable as well as its negation, which is not possible, because x could either be 1 or 0 but neither something in between nor both. Therefore no matter what is chosen for x the formula 2.1 can not be satisfied, is thus a contradiction and could be substituted with 0. In the second formula we use the previously checked conjunction on the left-hand side of the implication and as shown in table 2.1 if the left-hand side of the negation is false, the right-hand side of the implication does not make a difference. Therefore formula 2.2 will always evaluate to true. Additional formula 2.2 is an often used axiom namely the *Inconsistency Theorem*, stating that if something is inconsistent it will imply all properties since the formula could be proven to be satisfied independent of the valuation.

As a further remark, it should be noted that formulas partition the set of valuations into different equivalence partitions.

The most common task in propositional logic is to check satisfiability, which means to prove that there exists at least one assignment under which the formula evaluates to true. While the *SAT-problem* is *NP-complete*, recent implementations still allow to process formulas with up to hundreds of thousands of variables.[1]. Typically a SAT-solver has one of two results: Either it provides a *witness* that the formula is satisfiable, while the witness is an assignment

[1] This is especially due to recent improvements by *clause learning* and especially by *non-chronological backtracking*. [BHM09]

which satisfies the formula or it provides an *UNSAT-core*, which is a part of the formula which is contradicting and thus makes the formula not satisfiable.

A set of operators is called a *complete operator basis*, if for all possible formulas a *logically equivalent* ( ) formula[2] could be expressed only using those operators of the set. Such a complete operator basis are $f\wedge, :g$ or $f\_, :g$. It is possible to express all possible propositional operators as a macro of those given operators.

Furthermore, a propositional logic formula is called *monotone* if it could be expressed using only the operator basis $f\wedge, \_g$. Given a formula ψ is monotone and in addition, evaluates to true under a given valuation φ. Assume a variable x which appears in ψ is assigned to be false under φ. Given a new assignment φ', which is the same as φ, except that x is assigned to be true then the formula ψ has to evaluate to true under φ' as well since ψ is monotone.

Propositional logic is sometimes extended to *quanti ed propositional logic* by allowing the additional *exists* operator (9) and the *universal* operator (8) which bind stronger then any other operator. Later in this thesis, such formulas will be used. Note that introducing only the universal operators and checking satisfiability leads to a *tautology problem* which is *co-NP-complete*, since its negation is the SAT problem which is NP-complete. Introducing the exists operator as well will lead to a *quanti ed Boolean formula problem* (QBF) which is shown to be *PSPACE-complete*[3]. While it is not shown yet that this is the case, "it is believed that PSPACE-complete problems are strictly harder than NP-complete problems" [Boo19]. This is important to keep in mind for the rest of this thesis since it provides the expected complexity of most algorithms discussed later.

[2] A formula F is *logically equivalent* ( ) to G, if F and G evaluate to the same value for any valuation.

[3] For further details see [Boo19].

2.2. Predicate Logic

In propositional logic only variables and propositional logic operators are allowed. An extension of quantified propositional logic is the *first-order logic* (FO) where the definition[4] has to be split in syntax (i.e. how is a correct formula build) and semantics (i.e. how should this formula be interpreted). First consider the syntax:

A formula in FO is build using a set of *variables* $Var = x, y, z, \ldots$, as well as a *vocabulary* which consists of:

1. *Constant symbols*: $a, b, c, \ldots$
2. *Function symbols*: $f, g, h, \ldots$
3. *Predicate symbols*: $P, Q, R, \ldots$

Note that function and predicate symbols are only *symbols* in a formula. How they should be interpreted will be discussed in the *semantic*. Predicate symbols are often referred to as *relation symbols*. Each function and predicate symbol has an *arity* $k > 0$. This arity states how many arguments the function or the predicate symbol takes. A constant is a function without any arguments, thus 0-ary. Constant, function and predicate symbols, as well as their corresponding arity are given in a *signature* σ. A σ-*term* is defined by the following recursive rules:

Each variable x *2* Var is a term.

Each constant symbol a *2* σ is a term.

If $t_1, \ldots, t_k$ are terms and f is a k-ary function symbol, then $f(t_1, \ldots, t_k)$ is a term.

The set of first-order formulas $\mathrm{FO}(\sigma)$ is recursively defined as[5]:

If $t_1, \ldots, t_k$ are terms and P is a k-ary predicate symbol, then $P(t_1, \ldots, t_k)$ is a formula.

If t_1, t_2 are terms then $t_1 = t_2$ is a formula.

If F is a formula, so is $: F$.

If F, G are formulas, so is $F \quad G$ with being a binary operator of propositional logic.

If F is a formula and x is a variable then $9x.F$ and $8x.F$ are formulas.

[4] This definition is according to the lecture *logic* given in 2019 at the *TU Kaiserslautern*.
[5] Note that each formula has to be defined relative to a signature σ.

While the *syntax* defined how a FO formula should look like, the meaning of this function is defined by the *semantics*:

Given a signature σ where a set of symbols is defined, the structure S over σ is defined as a tuple $\mathsf{S} = (D, I)$, where D is called the *domain* (or the *universe*) and I is called the *interpretation*. The domain D is a nonempty set and describes all values a variable could have. Given for example $D = \{\mathsf{true}, \mathsf{false}\}$ this would resemble propositional logic, since in propositional logic each variable could be either true or false. However this is more flexible in FO. The interpretation I assigns each symbol a meaning. Thus in the case of constants, each constant $a \in \sigma$ is assigned to one element in D. Correspondingly, each function $f \in \sigma$ with its corresponding arity k under the interpretation I is assigned to $I(f) : D^k \to D$. Similarly each predicate $P \in \sigma$ with its corresponding arity k under the interpretation I is assigned to a k-ary relation $I(P) \subseteq D^k$.

Given for example a domain $D = \{A, B, C\}$ and a predicate $\mathsf{Knows}/2 \in \sigma$ (note that $/2$ denotes that Knows is 2-ary). Then a potential interpretation could be $I(\mathsf{Knows}) = \{(A, B), (B, C), (C, A)\}$. Thus $\mathsf{Knows}(A, B)$ would evaluate to true, since $(A, B) \in I(\mathsf{Knows})$, however $\mathsf{Knows}(B, A)$ would evaluate to false, since $(A, B) \notin I(\mathsf{Knows})$. Another example could be the predicate $\mathsf{Beard}/1$, with $I(\mathsf{Beard}) = \{A, C\}$. Thus the predicate Beard is true for the elements $A \in D$ and $C \in D$. Furthermore, the symbol which is used is independent of the interpretation of this symbol. Thus instead of choosing the symbol Beard, also the symbol X could be chosen. While Beard already suggests an interpretation, this is not the case if the symbol X is chosen. Note again that predicates are also called relations, as mentioned above. This is because they allow to define relations between different elements of the domain. This can not be expressed in propositional logic.

First-order logic is used frequently in computer science as for example in most publications which are discussed in the related work chapter and thus it is crucial to understand.

Given first-order logic as defined above, considering the name there should be a similar second-order logic. This is defined by extending the set of symbols allowed in first-order logic by additional symbols for relation-variables and thus allowing quantifications over relations. Lets consider the Peano-Axioms given a signature $S = \{0, \sigma\}$, where 0 denotes a constant, called zero-element and σ denoting the one-ary function of successor of an element [Prä19]:

$$\begin{aligned} &\forall x : \sigma x \neq 0 \\ &\forall x \forall y (\sigma x = \sigma y \rightarrow x = y) \\ &\forall X ((X0 \wedge \forall x (Xx \rightarrow X\sigma x)) \rightarrow \forall x Xx) \end{aligned} \tag{2.3}$$

This set of axioms describes the set of natural numbers and has some more additional interesting properties. This could easily be verified, given that the first axiom states that the *zero element* or starting element is not a successor of any other element. The second axiom states injectivity of the successor function σ, therefore each element has a successor, while if two elements have the same successor they need to be logically equivalent. The third axiom is

the so-called induction-axiom and states that if a one-ary relation X holds for the zero element 0 and for all elements x for which X holds it also holds for the corresponding predecessor σx, then this relation X will also hold for all elements x.

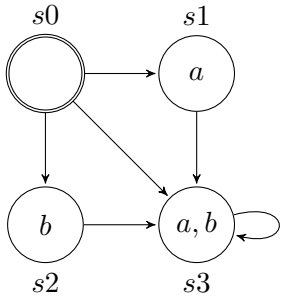

Figure 2.1.: *An example automaton. The initial state is marked with an additional circle around the state. The symbolic definition is given in equation 2.4.*

2.3. State Transition Systems

A *state transition system* (or *Kripke structure*)[6] $K = (V, \Psi_I, \Psi_R)$ is represented by the set of boolean variables V and two propositional logic formulas: the initial states Ψ_I and the transition relation Ψ_R. Each state $s \subseteq V$ of K is a subset of variables of V, such that all variables s hold in that state and all variables in $V \setminus s$ do not hold in s. Therefore each state s of the transition system corresponds to an assignment to all the variables in V and the set of all the states S are all possible assignments to V.[7]

Since all states correspond to assignments, we can use a propositional logic formula φ over the variables V to represent a set of states $[\![\varphi]\!]_K \subseteq 2^V$, which are all the states, where the corresponding assignment, which this state represents, satisfy the formula. As a further remark and as the notation suggests, this is similar to an equivalence partitioning. Note that in any formula corresponds to a set of states. However, if the structure of consideration is obvious from the context the formula, as well as the corresponding set of states, are used interchangeably and the additional brackets are omitted to reduce clutter.
The set of initial states is therefore described by the propositional logic formula Ψ_I, which then results in the set of initial states $[\![\Psi_I]\!]_K \subseteq 2^V$.
The transition relation Ψ_R is a propositional logic formula over the set of variables V as well as the set of primed variables V', while a primed variable $x' \in V'$ denotes a variable which holds in the next step and a variable $x \in V$ denotes a variable which holds in the current step. Therefore the set of transitions of K corresponds to all satisfying assignments of Ψ_R.

Consider the explicit[8] definition of the automaton given in figure (2.1) and

[6]

[7] If there should be more then one state labeled with the same variable assignment, we extend the set of variables V by *non-observable variables*, which are variables that are not of interest for the further evaluation. This way the sets of states still correspond to an assignment to all variables in V.

[8] If a structure is defined by a state-chart, this is called the *explicit* definition of a structure. In contrast if a structure is defined by formulas and sets of variables, this is called a

its symbolic definition:

$$\begin{aligned} K &= (\{a, b\}, \\ &\quad : a \wedge : b, \\ &\quad : (a \_ b) \wedge (a^{\prime} \quad b^{\prime}) \_ a^{\prime} \wedge b^{\prime}) \end{aligned} \tag{2.4}$$

An advantage of symbolic definitions is the scalability, therefore it is possible to describe automaton with millions of states with a symbolic representation, while it is quite difficult to handle such large explicit automaton. It should be noted that the most complex part of this symbolic definition is usually the transition relation Ψ_R. Depending on if this should be more human-readable, one could choose a normal form like the *disjunctive normal form* (DNF) which simplifies the transition relation by unrolling and therefore basically listing all the transitions, but since both expressions are logically equivalent they could be used interchangeably:

$$\begin{aligned} \Psi_R &= : (a \_ b) \wedge (a^{\prime} \quad b^{\prime}) \_ a^{\prime} \wedge b^{\prime} \\ &= \begin{pmatrix} : a \wedge : b \wedge a^{\prime} \wedge : b^{\prime} \_ \\ : a \wedge : b \wedge : a^{\prime} \wedge b^{\prime} \_ \\ : a \wedge : b \wedge a^{\prime} \wedge b^{\prime} \_ \\ a \wedge : b \wedge a^{\prime} \wedge b^{\prime} \_ \\ : a \wedge b \wedge a^{\prime} \wedge b^{\prime} \_ \\ a \wedge b \wedge a^{\prime} \wedge b^{\prime} \end{pmatrix} \end{aligned} \tag{2.5}$$

Given the formula $\varphi := a$, this would encode the state set $[\![\varphi]\!]_K = \{s1, s3\}$, where $s1$ represents the state in which a holds, while b does not hold and $s3$ represents the state where a and b holds.
In the following chapters I will make use of the symbolic definitions of state transition systems for efficient handling of systems of arbitrary size, while using the explicit representation for visualization.

Sometimes those systems are defined by a *finite state machine* (FSM) $\mathsf{A} = (\Sigma_{in}, \Sigma_{out}, I, S, R)$. Those consist of a set of states S, an input alphabet Σ_{in}, an output alphabet Σ_{out}, a set of initial states I and a transition relation which connects states by transitions R, where each transition has a unique input and output. However, most often only the possible transitions of this machine are of interest for verification purposes and the causality (why the transition is taken) does not matter. Thus we consider the corresponding Kripke structure. Note that each FSM could be transformed into a Kripke structure and each Kripke structure could be interpreted as an FSM with only one-letter input and output alphabets.

As already shown for the transition relations Ψ_R in chapter 2.3 there exist temporal relationships between properties, in this case we used the state set V as well as state-set $V^{\prime}$ to denote properties which hold in the current state

symbolic definition.

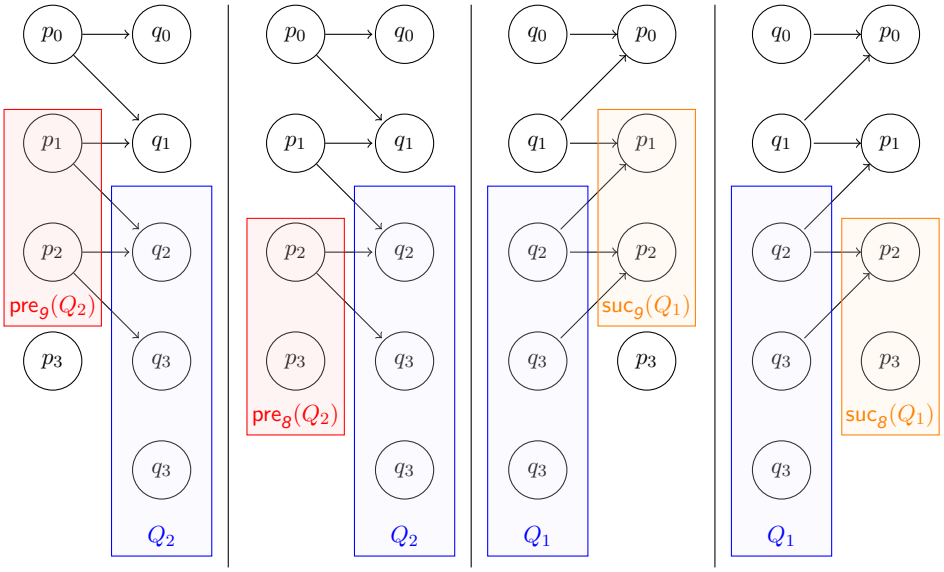

Figure 2.2.: *A visualization of predecessors and successors on some Kripke structures. This figure is based on [Sch19b].*

as well as in the next state. To further formalize this we define the set of *predecessors* and the corresponding *successor*, where we distinguish between the *existential* and the *universal* predecessor and successor:

$$\mathsf{pre}_\exists^R(Q_2) = \{s_1 \in S_1 \mid \exists s_2.(s_1, s_2) \in R \wedge s_2 \in Q_2\} \tag{2.6}$$

$$\mathsf{pre}_\forall^R(Q_2) = \{s_1 \in S_1 \mid \forall s_2.(s_1, s_2) \in R \rightarrow s_2 \in Q_2\} \tag{2.7}$$

$$\mathsf{suc}_\exists^R(Q_1) = \{s_2 \in S_2 \mid \exists s_1.(s_1, s_2) \in R \wedge s_1 \in Q_1\} \tag{2.8}$$

$$\mathsf{suc}_\forall^R(Q_1) = \{s_2 \in S_2 \mid \forall s_1.(s_1, s_2) \in R \rightarrow s_1 \in Q_1\} \tag{2.9}$$

As an intuitive approach the $\mathsf{pre}_\exists^R(Q)$ are all states of K , which have at least one transition to a state in Q. Therefore a *deadend state* (a state without any outgoing transitions) could not be an element of $\mathsf{pre}_\exists^R(Q)$, since it obviously has no outgoing transitions and thus trivially no transition in any set Q. Similar the set of deadend states of the Kripke structure K is equal to $\mathsf{pre}_\forall^R(\{\})$. This is because $\mathsf{pre}_\forall^R(\{\})$ contains all the states for which all outgoing transitions lead into the empty set. Obviously this can not be the case if the state has an outgoing transition, therefore

$$\mathsf{pre}_\forall^R(\{\}) = \{s \in S \mid s \text{ is deadend state}\}. \tag{2.10}$$

2.4. Lattice Theory

Most of the calculations done in this thesis are based on the lattice theory since this forms the foundation of fixpoint calculations. To introduce a lattice, one has to start with binary relations on a set D.

Definition 2.4.1 (Partial Order)**.** *A binary relation $\sqsubseteq$ on a set D is called a partial order if*

$$\textit{reflexivity} : \forall x \in D.x \sqsubseteq x \tag{2.11}$$
$$\textit{antisymmetry} : \forall x, y \in D.x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \tag{2.12}$$
$$\textit{transitivity} : \forall x, y, z \in D.x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq y \tag{2.13}$$

This definition states, that a reflexive, antisymmetric and transitive relation is a partial order. Let us consider as an example the equivalence relation $(=)$ on the natural numbers $\mathbb{N}$: The relation is reflexive because each element is in relation to itself since when comparing the element with itself the equality holds. The relation is antisymmetric since if two elements $x, y \in \mathbb{N}$ are in relation to each other they have to be the same element and the same argument holds for the transitivity.

Definition 2.4.2 (Total Order)**.** *A binary relation $\sqsubseteq$ on a set D is called a total order if*

$$\textit{reflexivity} : \forall x \in D.x \sqsubseteq x \tag{2.14}$$
$$\textit{antisymmetry} : \forall x, y \in D.x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \tag{2.15}$$
$$\textit{transitivity} : \forall x, y, z \in D.x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq y \tag{2.16}$$
$$\textit{totality} : \forall x, y \in D.x \sqsubseteq y \sqcup y \sqsubseteq x \tag{2.17}$$

But the equivalence relation on the natural numbers $\mathbb{N}$ is not a total order. This is because two different elements $x, y \in \mathbb{N}$ are not in relation to each other, because neither $x = y$ nor $y = x$ and thus not all elements of $\mathbb{N}$ are in relation to each other.
A more practical example would be the *subset relation* $A \subseteq B$, which states that A is a subset of or equal to B. This could be visualized for the set $2^{\{1,2,3\}} = \{1, 2, 3\}$ by a *Hasse diagram* shown in figure 2.3. A Hasse diagram is a common way to visualize sets and corresponding partial order, by showing the elements as well as their hierarchical relation. To reduce visual clutter in Hasse diagrams, usually the reflexive and transitive closure is stated, and therefore each node is in relation to itself, as well as any other node to or from which it has a path.

Let us use a more intuitive approach to reason about relations. Given some relation on a set, a partial order lets you compare some elements of the set with each other, for example using the equality relation this would only be the same element. A total order, however, imposes some order on all the elements and it is, therefore, possible to construct a chain of all elements in that order. This is the reason why a totally ordered set is also called a *chain*. When using the

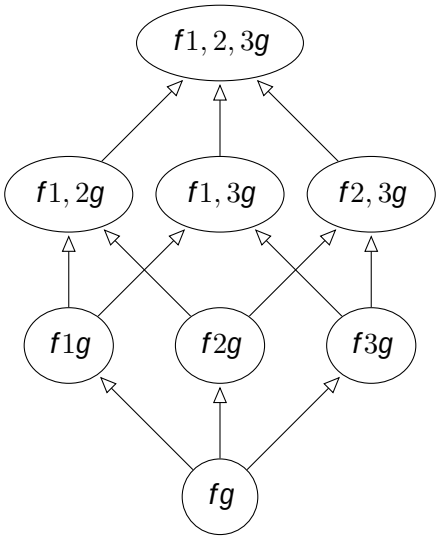

Figure 2.3.: *The Hasse diagramm of* $(2^{f1,2,3g}, \quad)$*, therefore, the set of all subsets of* $f1, 2, 3g$ *and the subset relation* $\quad$*. Assume the reflexive and transitive closure.*

relation shown in figure 2.3 it is well visible, that $f1, 2g$ and $f1, 3g$ are not in a relation to each other, which makes sense, because trivially none is a subset of the other, since they have different elements. However it would be possible to generate a chain by only using $2^{f1,2,3g^{\emptyset}} = ffg, f1g, f1, 2g, f1, 2, 3gg$ and the subset relation, since in this case the subset relation would be a total order. This is an important concept because it shows that totality is depending on the set as well as on the relation which is used. Note that a partial order always consists of a set as well as a relation, but usually the relations considered are the subset-relation ( ) or less-than-relation ( ) and in this case, we only mention the set and abbreviate the relation.

Definition 2.4.3 (Least Upper and Greatest Lower Bound)**.** m *2* D *is an upper bound of* $M \quad D$ *if*

$$8x \ 2 \ M.x \ \mathsf{v} \ m$$

m *2* D *is the least upper bound or supremum* $\sup(M)$ *of* $M \quad D$ *if*

$$m = \sup(M) : , \quad \begin{pmatrix} (8x \ 2 \ M.x \ \mathsf{v} \ m)\mathsf{\wedge} \\ (8y \ 2 \ D.(8x \ 2 \ M.x \ \mathsf{v} \ y) \ ! \ m \ \mathsf{v} \ y) \end{pmatrix}$$

and analogously n *2* D *is the greatest lower bound or* $\inf(M)$ *of* $M \quad D$ *if*

$$n = \inf(M) : , \quad \begin{pmatrix} (8x \ 2 \ M.n \ \mathsf{v} \ x)\mathsf{\wedge} \\ (8y \ 2 \ D.(8x \ 2 \ M.y \ \mathsf{v} \ x) \ ! \ y \ \mathsf{v} \ n) \end{pmatrix}$$

If $m = \sup(M)$ *2* M*, then* m *it is the maximum of* M *and if* $n = \inf(M)$ *2* M *then* n *is called the minimum of* M*.*

Let us consider least upper bounds and greatest lower bounds for the relation shown in figure 2.3. Obviously the set $2^{\{1,2,3\}}$ has a $\inf\left(2^{\{1,2,3\}}\right) = \{\}$ and a $\sup\left(2^{\{1,2,3\}}\right) = \{1,2,3\}$, which could be seen easily in the Hasse diagram because of the hierarchical order and all nodes are lower as the upper bound and higher as the lower bound. Additionally since $\inf\left(2^{\{1,2,3\}}\right) = \{\} \in 2^{\{1,2,3\}}$ and $\sup\left(2^{\{1,2,3\}}\right) = \{1,2,3\} \in 2^{\{1,2,3\}}$ the greatest lower and the least upper bounds are the minimum and maximum respectively. If however we determine the least upper bound for $(2^{\{1,2,3\}} \cap \{1,2,3\}) \subseteq 2^{\{1,2,3\}}$, this would still be the element $\{1,2,3\} \notin (2^{\{1,2,3\}} \cap \{1,2,3\})$ which is therefore no maximum, since it is not in the set. It should be noted that the supremum and the infimum if it exists, is always *unique*.

Definition 2.4.4 (Lattice)**.** *Given a set D with a partial order $\preceq$.*

$(M \subseteq D, \preceq)$ is a bidirected set if all $\{x, y\} \subseteq M$ have lower and upper bounds in M

$(D, \preceq)$ is a lattice if all $\{x, y\} \subseteq D$ have $\sup(x, y), \inf(x, y) \in D$

$(D, \preceq)$ is a complete lattice if all $M \subseteq D$ have $\sup(M), \inf(M) \in D$

If it is possible to show that all two-element subsets of a given set under a partial order have a supremum and an infimum, this forms a lattice. In addition, if $|D|$ is finite and $(D, \preceq)$ is a lattice, then $(D, \preceq)$ is a complete lattice. As an example consider figure 2.3 with $2^{\{1,2,3\}}$ and the subset-relation $\subseteq$. As can be verified each two-element set has a supremum, e.g. if both elements are on the same level, it is the element connected to both previously mentioned element on the next higher level and this element is unique. Thus the order shown in figure 2.3 forms a lattice and since all lattices on finite sets are also complete, it even forms a complete lattice. Also note, that given a total order this by default implies a lattice since all elements could be compared to each other and therefore for any two-element subset those two elements could be compared with one being in relation to the other and thus having even a maximum and a minimum.
As an example that not all infinite lattices are complete, let us consider $(\mathbb{N}, \leq)$. Obviously it is a lattice since all two-element subsets have a least and a largest element, which is derived by the totality of $\leq$ on natural numbers. To be a complete lattice, each subset M of $\mathbb{N}$ also needs to have a least and a largest element, not only two-element subsets, however if $M = \mathbb{N}$ there is no $\sup(M)$ since there is no largest number in $\mathbb{N}$ and therefore $(\mathbb{N}, \leq)$ is not a complete lattice.

Further properties which could be derived from the lattice property are evaluated in chapter 2.6.

2.5. μ-calculus

When reasoning about Kripke structures, this usually leads to a point where some temporal property should be checked. The most common temporal logic property is the so-called *safety* property, which checks if some condition holds on all reachable states or in other words if some property holds in all initial states, as well as on all states on all paths starting in these initial states. To formally describe such properties we use μ-calculus formulas.

Definition 2.5.1 (μ-Calculus Syntax)**.** *The set of μ-calculus formulas L_μ over variables V are the least set of formulas satisfying the following properties provided that φ, ψ 2 L_μ and x 2 V*

variables: V L_μ

propositional operators: $: \varphi, \varphi \wedge \psi, \varphi \_ \psi$ *2* L_μ

modal operators: $\varphi, \quad \varphi, \quad \varphi, \quad \varphi$ *2* L_μ

xpoint operators: $\mu x.\varphi$ *2* L_μ *and* $\nu x.\varphi$ *2* L_μ

It is moreover required that all occurrences of the bound variable x in the xpoint formulas $\mu x.\varphi$ and $\nu x.\varphi$ are positive (i.e., covered by an even number of negations).

As can be seen, the set of propositional logic formulas is extended by some additional operators, namely the modal operators encoding the predecessors as well as the successors and the fixpoint operators, denoting the least and the greatest fixpoint:

Definition 2.5.2 (μ-Calculus Semantics)**.** *Given a state transition system $K = (V, \Psi_I, \Psi_R)$ and a μ-calculus formula Φ 2 L_μ, the following recursively de ned function determines a set of states $\llbracket \Phi \rrbracket_K$ of K as the semantics of Φ:*

$\llbracket x \rrbracket_K := \{ s \in S \mid x \in L(s) \}$

$\llbracket : \varphi \rrbracket_K := S \setminus \llbracket \varphi \rrbracket_K$

$\llbracket \varphi \wedge \psi \rrbracket_K := \llbracket \varphi \rrbracket_K \cap \llbracket \psi \rrbracket_K$

$\llbracket \varphi \_ \psi \rrbracket_K := \llbracket \varphi \rrbracket_K \cup \llbracket \psi \rrbracket_K$

$\llbracket \quad \varphi \rrbracket_K := \mathrm{pre}_\exists^R(\llbracket \varphi \rrbracket_K)$

$\llbracket \quad \varphi \rrbracket_K := \mathrm{pre}_\forall^R(\llbracket \varphi \rrbracket_K)$

$\llbracket \quad \varphi \rrbracket_K := \mathrm{suc}_\exists^R(\llbracket \varphi \rrbracket_K)$

$\llbracket \quad \varphi \rrbracket_K := \mathrm{suc}_\forall^R(\llbracket \varphi \rrbracket_K)$

$\llbracket \mu x.\varphi \rrbracket_K$ *is the least xpoint of* $f(Q) := \llbracket \varphi \rrbracket_{K_x^Q}$

$\mathsf{J}\nu x.\varphi\mathsf{K}_K$ *is the greatest xpoint of* $f(Q) := \mathsf{J}\varphi\mathsf{K}_{K_x^Q}$

The function $f(Q) := \mathsf{J}\varphi\mathsf{K}_{K_x^Q}$ *is called the state transformer of* φ *and* x*. For its de nition, we use the modi ed Kripke structure* K_x^Q *that changes the variable assignment of variable* x *such that exactly the states* Q *satisfy variable* x*.*

Lets consider an example. Given a formula

$$\mathsf{J}\nu x.\ \ x\mathsf{K}_K \tag{2.18}$$

Since it starts with a ν the result will be the greatest fixpoint of the expression x, while encodes the existential predecessor. When calculating this fixpoint, one will start with the set of all states $Q_0 = S$ of K and calculates the set of predecessors of S by building a conjunction with $S^{\mathbb{0}}$ and the transition relation Ψ_R:

$$Q_0 := S \tag{2.19}$$
$$Q_1 := \ \ Q_0 = \mathsf{pre}_\exists^R(Q_0) = \exists x^{\mathbb{0}}.\Psi_R \wedge Q_0^{\mathbb{0}} \tag{2.20}$$
$$Q_2 := \ \ Q_1 = \mathsf{pre}_\exists^R(Q_1) = \mathsf{pre}_\exists^R(\mathsf{pre}_\exists^R(Q_0)) \tag{2.21}$$
$$Q_3 := \ \ Q_2 = \mathsf{pre}_\exists^R(Q_2) = \mathsf{pre}_\exists^R(\mathsf{pre}_\exists^R(\mathsf{pre}_\exists^R(Q_0))) \tag{2.22}$$

Therefore the set Q_n denotes the set of states which have at least one outgoing path with n transitions. When doing this calculation until one reaches a fixpoint, which is the case as soon as $Q_n = Q_{n+1}$ for an $n \in \mathbb{N}$, there are 2 cases: Q_{fixpoint} is the empty set or it contains some states. The case where Q_{fixpoint} is the empty set states that the property $\mathsf{J}\nu x.\ x\mathsf{K}_K$ does not hold in any state given the transition relation Ψ_R. If the set Q_{fixpoint} however is not empty, it contains all the states which have an infinite path. Consider why this is the case. For example, if the paths of some states s are only finite, then there has to be a longest path from this state with length n. In this case, s would be removed in some Q_i, i n. Note that for the calculations of the sets Q_n we make use of the fact that a state set could be represented by a propositional logic formula.
Lets consider another example formula

$$\mathsf{J}\nu x.\varphi \wedge \ \ x\mathsf{K}_K \tag{2.23}$$

This formula is quite similar to the previously shown formula 2.18 since only the φ is added with a conjunction. This formula describes the previously mentioned *safety property* and is of great interest for the rest of this thesis, as well as current research since it is one of the most used formulas. As can be seen this formula ensures two assertions: (1) Each state satisfying this formula is also satisfying φ and (2) each state is a predecessor or a state which also satisfies this property, or corollary that each state has at least one successor on which the property φ holds and which in turn has a successor which satisfies this assertion. This results in a set of all states which have an outgoing infinite

path on which in each step φ holds. This is a very powerful tool since if for example a program of any language is translated to a Kripke structure, one can verify that some property ϕ will always hold in the given program.

Given a Kripke structure $K = (I, S, R, L)$ and a μ-calculus formula $\varphi \in L_\mu$, we call K a *model* of φ if all initial states I of K are contained in the set of states defined by $[\![\varphi]\!]_K$:

$$K \models \varphi :\Leftrightarrow I \subseteq [\![\varphi]\!]_K \tag{2.24}$$

Therefore we call the process of verifying whether a given Kripke structure K is a model of φ *model checking* and the process of checking if there is a Kripke structure K, such that K is a model of φ *satisfiability checking*. Usually when referring to *model checking* there are two options to do so: *local model checking* and *global model checking*.

local model checking: In this case the paths of each of the initial states $s_i \in I$ is considered. Therefore each state s_i is checked regarding property φ and if all initial states satisfy that property, depending on φ the predecessors or the successors are checked until a fixpoint is reached or the check failed.

global model checking: In this case instead of starting with the initial states, we compute the set of states satisfying the property φ according to the definitions above and check if all initial states $s_i \in I$ are contained in this set.

2.6. Fixpoints

Definition 2.6.1 (Monotonic and Continuous Functions)**.** *Given complete lattices* $(D, \sqsubseteq_D)$ *and* $(E, \sqsubseteq_E)$ *and a function* $f : D \to E$

f is monotonic if $x \sqsubseteq_D y \Rightarrow f(x) \sqsubseteq_E f(y)$

f is continuous if $f(\sup(M)) = \sup(f(M))$ *and* $f(\inf(M)) = \inf(f(M))$ *holds for every directed set* $M \subseteq D, M \neq \emptyset$;

The definition 2.6.1 states, that given two complete lattices and a function which maps from one lattice to the other, it can be determined if the given function is monotonic and continuous. Furthermore if it can be proven that the function is continuous, this already implies that said function is monotonic as well.

Proof. Assume that $x \sqsubseteq y$ which states that x and y are comparable and thus $y = \sup(\{x, y\})$. If $f : D \to E$ is a continuous function on the lattice $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, this results in:

$$f(y) = f(\sup(\{x, y\})) = \sup(\{f(x), f(y)\}) \Rightarrow f(x) \sqsubseteq f(y) \tag{2.25}$$

□

In addition if a given lattice $(D, \sqsubseteq)$ is finite and a function $f : D \to D$ on this lattice is monotonic, this means the function is also continuous.

Proof. This proof is based on [Sch19b]: We try to proof that $f(\sup(M)) = \sup(f(M))$ holds for all directed, non empty subsets $M \subseteq D$. First we proof that $\sup(f(M)) \sqsubseteq f(\sup(M))$. Given some $x \in M$, we have $x \sqsubseteq \sup(M)$, according to the definition of the supremum. Furthermore we can conclude that $f(x) \sqsubseteq f(\sup(M))$ by the fact that f is monotone and therefore $f(\sup(M))$ is an upper bound of $f(M)$, by which $\sup(f(M)) \sqsubseteq f(\sup(M))$.

Second consider the induction base $\sup(\{e\}) = e \in \{e\}$ and the induction step $\sup(\{e\} \cup M) = \sup(\{e, \sup(M)\}) \in M$, since M is directed and finite. By that we conclude that $\sup(M) \in M$.

Therefore $f(\sup(M)) \in f(M)$, since we apply the function f to both sides of $\sup(M) \in M$. By the definition 2.4.3 of suprema this yields $f(\sup(M)) \sqsubseteq \sup(f(M))$ and if we combine this with the already shown fact that $\sup(f(M)) \sqsubseteq f(\sup(M))$ and that the relation $(\sqsubseteq)$ is antisymmetric, this results in $\sup(f(M)) = f(\sup(M))$. Therefore it is shown that in finite lattices every monotonic function is continuous. □

Since in the rest of the thesis we will only consider finite sets, this will be quite relevant later on and also entails that we only have to consider complete lattices, since as previously stated all lattices on finite sets are complete by default.

What we have shown by this proofs, is that any monotonic[9] function on lattices which are defined on finite sets, are continuous.

We distinguish between the following types of fixpoints:

Definition 2.6.2 (Fixpoints, Pre-Fixpoints and Post-Fixpoints)**.** *Given a function $f : D \to D$ and an element $x \in D$:*

x is xpoint of $f : D \to D$ if $f(x) = x$ holds.

x is pre- xpoint of $f : D \to D$ if $f(x) \sqsubseteq x$ holds.

x is post- xpoint of $f : D \to D$ if $x \sqsubseteq f(x)$ holds.

Consider the following intuitive approach to that definition. If a function is defined on a lattice and is monotone as well as continuous, this means that by applying the function this will result in a one of three cases.The value could either *increase*, thus $x \sqsubseteq f(x)$, in which case x would be considered a post-fixpoint, *decrease* ($f(x) \sqsubseteq x$) in which case x would be a pre-fixpoint or stay the same ($x = f(x)$) and therefore x would be a fixpoint. This results directly from the properties that $f : D \to D$ is continuous and monotone, as well as D being finite.

Furthermore, in a complete lattice, any monotone function has a fixpoint and dual "the existence of a fixpoint for every increasing function is a necessary condition for completeness of a lattice"[Tar+55]. This does not impose a restriction for the purpose of this thesis, since all domains of interest, despite maybe being large, are always finite and therefore if they form a lattice are always a complete lattice.
Given that there is more than one fixpoint this will result in a unique least and a unique greatest fixpoint.

Proof. The proof goal is to show that the greatest fixpoint of the monotone function $f : D \to D$ on a complete lattice $(D, \sqsubseteq)$ is unique. (Analogous for the least fixpoint)
Given a set of greatest fixpoints G. Consider two elements $x, y \in G$.
Trivially, if $x \sqsubseteq y$ or $y \sqsubseteq x$ not both could be greatest fixpoints. Therefore it remains to consider G' with $\forall x, y \in G', x \not\sqsubseteq y \wedge y \not\sqsubseteq x$. Given any $x' \in G'$ and by the definition of suprema and D being a lattice, this results in $x' \sqsubseteq \sup(G')$. Applying f to both sides of this equation will yield $f(x') \sqsubseteq f(\sup(G'))$ and since f is monotone and continuous and since x' is a greatest fixpoint this results in $x' \sqsubseteq f(\sup(G'))$. This states, that f could be applied to the right part of the equation until the right part reaches a fixpoint without changing the relation. Let $f\ (\sup(G'))$ denote the repeated appliance of f and therefore the fixpoint of $\sup(G')$. This would result in $x' \sqsubseteq f\ (\sup(G'))$ and since x' is a greatest fixpoint in $f\ (\sup(G')) \sqsubseteq x'$. Since $(\sqsubseteq)$ is antisymmetric and f is a

[9] From this point on, when monotone functions are required, it is always assumed, that the function is monotone *growing*. Yet still, despite a function f being monotone growing, an application of f to a value x could still result in a smaller value. E.g. $f(x) = \frac{1}{2}x$ is a monotone growing function, but each application of the function reduces the value.

function and therefore deterministic, it follows that $x^{0} = f\ (\sup(G^{0}))$ for any $x^{0}\ 2\ G^{0}$ and thus G^{0} could only contain one element. □

We use $\mu x.f(x)$ to denote the least fixpoint and $\nu x.f(x)$ to denote the greatest fixpoint of f.

Definition 2.6.3 (Least and Greatest Fixpoint)**.** *Given that $f : D\ !\ D$ is a monotonic function and D is a complete lattice, then:*

$$\mu x.f(x) = \inf(fx\ 2\ Dj f(x)\ v\ xg)$$

$$\nu x.f(x) = \sup(fx\ 2\ Djx\ v\ f(x)g)$$

This de nition is according to [Sch19b].

So far it is only stated that there is a least and a greatest fixpoint, but the *Tarski-Knaster Theorem* states that there actually exists a procedure to calculate those fixpoints. Note that this theorem assumes complete lattices because in a complete lattice (D, v) the existence of a minimal element $\inf(D) = ?\ 2\ D$ and a maximal element $\sup(D) = >\ 2\ D$ is guaranteed.

Theorem 1 (Tarski-Knaster Theorem [Tar+55])**.** *Given a complete lattice D and a continuous function $f : D\ !\ D$:*

The sequence $p_{i+1} := f(p_i)$ with $p_0 := ?$ converges to $\mu x.f(x)$

The sequence $p_{i+1} := f(p_i)$ with $p_0 := >$ converges to $\nu x.f(x)$

Proof. Assume that $p_0 = ?$. Since f is monotone we have $p_i\ v\ p_{i+1}$. Let $C = fp_0, p_1, \ldots g$ be the chain $C = ff^{n}(?)jn\ 2\ Ng$. Then we have

$$\sup(C) = \sup\left(ff^{n+1}(?)jn\ 2\ Ng\right)$$

and therefore

$$f(\sup(C)) = \sup(f(C)) = \sup\left(ff^{n+1}(?)jn\ 2\ Ng\right) = \sup(C\ n\ f?g) = \sup(C)$$

and thus $\sup(C)$ is a fixpoint of f. Next we prove that $8n.f^{n}(?)\ v\ x_f$ for every fixpoint x_f:

The induction base is trivial since $f^{0}(?) = ?\ v\ x_f$. For the induction step consider that $f(x_f) = x_f$ since x_f is a fixpoint. By monotonicity of f we can conclude that $f^{n+1}(?)\ v\ f(x_f) = x_f$.

Thus it follows that $\sup(C)\ v\ x_f$ for every fixpoint x_f and thus $\sup(C) = \mu x.f(x)$.[10] □

As alternative definition to the supremum and infimum (2.4.3) as prerequisite of a lattice, we could also define the infimum as *meet* (^) and the

[10] This proof is based on [Sch19b].

supremum as *join* (_) with _ and ^ being binary operators in *lattice theory*, as it is done and proven in [Gra09]. [11] This would result in:

$$a \wedge b = \inf(\{a, b\}) \tag{2.26}$$
$$a \vee b = \sup(\{a, b\}) \tag{2.27}$$

While the operators _ and ^ satisfy a set of properties, namely:

$$\text{idempotency: } a \wedge a = a, a \vee a = a \tag{2.28}$$
$$\text{commutativity: } a \wedge b = b \wedge a, a \vee b = b \vee a \tag{2.29}$$
$$\text{associativity: } (a \wedge b) \wedge c = a \wedge (b \wedge c), \quad (a \vee b) \vee c = a \vee (b \vee c) \tag{2.30}$$

and both of them are *binary operations* and could be applied to any pair of elements of a lattice. This is interesting, because we could also specify the relation $\sqsubseteq$ by only using _ and ^, since if $a \sqsubseteq b$, then $\inf(\{a, b\}) = a$ and thus $a \wedge b = a$ and conversely with the supremum. By that we could also conclude:

$$a \sqsubseteq b \text{ iff } a \wedge b = a \tag{2.31}$$
$$a \sqsubseteq b \text{ iff } a \vee b = b \tag{2.32}$$

and by that get a fourth property of _ and ^:

$$\text{absorption identities: } a \wedge (a \vee b) = a, \quad a \vee (a \wedge b) = a \tag{2.33}$$

This definition of meet and join allows to conclude two very important insights, which are an important foundation of the rest of the thesis: (1) It characterises a lattice as an algebra and (2) it provides a theoretic foundation between lattices on state-sets of automata and the propositional logic representation of those state sets.

Duality of lattices and algebras

As proven in [Gra09], a lattice $(D, \sqsubseteq)$ could also be defined as $(D : \wedge, \vee)$ and the dual also holds that an algebra $(D : \wedge, \vee)$ is called a lattice, if: D is a nonvoid set, _ and ^ are idempotent, commutative, associative and satisfy the absorption identities.
Since a lattice could be expressed by an algebra and vice versa, "if we treat lattices as algebras, then all the concepts and methods of universal algebra will become applicable" [Gra09]. Thus all algebraic structures which are known and proven could also be used in the context of lattices which provides a link to a field of mathematics and increases the usability of the lattice theory.

[11] At this point should be noted, that the symbol is identical with the one being used in propositional logic, described in chapter 2.1, but is used independently. However it is easy to see that the definition of conjunction and disjunction previously stated does satisfies all of the properties.

Duality of state set union and intersection and propositional logic conjunction and disjunction

It is no coincidence, that the *disjunction* and *conjunction* of propositional logic (as defined in chapter 2.1) as well as the *join* and *meet* share the same symbols _ and ^. In fact, the disjunction as well as the conjunction share *idempotency*, *commutativity*, *associativity* as well as *absorption identities* (see (2.28)-(2.30), (2.33)). By that we can conclude that given a structure K, the supremum and infimum of two sets of states $\llbracket\Psi_1\rrbracket_K$ and $\llbracket\Psi_1\rrbracket_K$ could be described by $\llbracket\Psi_1 \wedge \Psi_2\rrbracket_K$ and $\llbracket\Psi_1 \vee \Psi_2\rrbracket_K$ respectively. Furthermore if we consider a set of $\llbracket\Psi\rrbracket_K$, thus a set of sets of states[12] Ψ , if for any two $\llbracket\Psi_1\rrbracket_K \in \Psi$ and $\llbracket\Psi_2\rrbracket_K \in \Psi$ it is proven that $\llbracket(\Psi_1 \vee \Psi_2)\rrbracket_K \in \Psi$ and $\llbracket(\Psi_1 \wedge \Psi_2)\rrbracket_K \in \Psi$, then we can conclude that Ψ is a lattice. This insight will be used in chapter 4. Furthermore note that by that we can also conclude that the following equivalences hold:

$$\llbracket(\Psi_1 \wedge \Psi_2)\rrbracket_K = \llbracket\Psi_1\rrbracket_K \setminus \llbracket\Psi_2\rrbracket_K \tag{2.34}$$

$$\llbracket(\Psi_1 \vee \Psi_2)\rrbracket_K = \llbracket\Psi_1\rrbracket_K \sqcup \llbracket\Psi_2\rrbracket_K \tag{2.35}$$

Again, this insight will be used later in this thesis.

[12] Note that this can not be expressed by a single propositional logic formula.

2.7. Temporal Logic

While μ-calculus has been proven to be very expressive (see [Sch13]), it still introduces an inherent problem. Namely that the formulas are usually difficult to read, especially for more complex properties and nested fixpoints. To handle this problem the *computational tree logic* (CTL) was introduced. Essentially this language provides *macros* for certain μ-calculus formulas, which make formulas much easier to read. Note that this only applies to a human reader. This logic is defined as following:

Definition 2.7.1 (Temporal Logic CTL)**.** *The set of* CTL *temporal logic formulas is defined in terms of μ-calculus formulas as follows:*

$$
\begin{array}{ll}
\mathsf{EX}\varphi = \quad \varphi & \mathsf{AX}\varphi = \quad \varphi \\
\mathsf{EG}\varphi = \nu x.\ \varphi \ \wedge \quad x & \mathsf{AG}\varphi = \nu x.\ \varphi \ \wedge \quad x \\
\mathsf{EF}\varphi = \mu x.\ \varphi \ \vee \quad x & \mathsf{AF}\varphi = \mu x.\ \varphi \ \vee \quad x \\
\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi] = \mu x.\ \psi \ \vee\ \varphi \ \wedge \quad x & \mathsf{A}[\varphi\ \underline{\mathsf{U}}\ \psi] = \mu x.\ \psi \ \vee\ \varphi \ \wedge \quad x \\
\mathsf{E}[\varphi\ \mathsf{U}\ \psi] = \nu x.\ \psi \ \vee\ \varphi \ \wedge \quad x & \mathsf{A}[\varphi\ \mathsf{U}\ \psi] = \nu x.\ \psi \ \vee\ \varphi \ \wedge \quad x \\
\mathsf{E}[\varphi\ \underline{\mathsf{B}}\ \psi] = \mu x.\ :\psi \ \wedge\ (\varphi \ \vee \quad x) & \mathsf{A}[\varphi\ \underline{\mathsf{B}}\ \psi] = \mu x.\ :\psi \ \wedge\ (\varphi \ \vee \quad x) \\
\mathsf{E}[\varphi\ \mathsf{B}\ \psi] = \nu x.\ :\psi \ \wedge\ (\varphi \ \vee \quad x) & \mathsf{A}[\varphi\ \mathsf{B}\ \psi] = \nu x.\ :\psi \ \wedge\ (\varphi \ \vee \quad x)
\end{array}
$$

Note that this language provides *path quantifiers* A and E as well as temporal logic operators X, G, F, $[\,\mathsf{U}\,]$, $[\,\underline{\mathsf{U}}\,]$, $[\,\mathsf{B}\,]$, and $[\,\underline{\mathsf{B}}\,]$. Each temporal operator has to be guarded by a path quantifier in CTL.

First, consider the path quantifiers A and E. In temporal logic, all infinite paths are considered which start in an initial state. Thus A corresponds to the $\forall$ quantifier over all paths known from predicate logic and E corresponds to the $\exists$ operator over all paths. Thus $\mathsf{AX}\alpha$ is satisfied if all infinite paths starting in all initial states satisfy $\mathsf{X}\alpha$. Analogous, to satisfy $\mathsf{EX}\alpha$ there has to be at least one infinite path starting in each initial state which satisfies $\mathsf{X}\alpha$. Thus the path quantifiers provide a set of paths starting in initial states.

On the other hand, the temporal logic operators work on paths. Thus they have to be provided a set of paths by the path quantifiers and thus have to be guarded by a path quantifier in CTL. Consider those temporal logic operators. The $\mathsf{X}\varphi$ operator states that the next state on the path has to satisfy some property φ. The $\mathsf{G}\varphi$ operator asserts that on every state on the path the property φ holds, while $\mathsf{F}\varphi$ states that after a finite amount of time some property φ has to hold. For the until and the before operator, there is a weak as well as a strong version. The weak until $[\varphi\ \mathsf{U}\ \psi]$ ensures that some property φ holds until ψ holds for the first time on the path, which could also take forever. For the strong until $[\varphi\ \underline{\mathsf{U}}\ \psi]$ this is quite similar, however, it is enforced in addition that ψ holds eventually after a finite amount of time. Similar the weak before $[\varphi\ \mathsf{B}\ \psi]$ asserts that the φ holds at least once on the path before ψ holds. For the strong before however, it has to be ensured that ψ holds after finitely many steps on the path.

There exist a more powerful logic CTL* where any combination of quantifiers and operators is allowed. Note that a translation from CTL* to μ-calculus

is possible, however not as easy and for this thesis not relevant. The interpretation of each operator and quantifier is the same. Consider for example the CTL formula $\mathsf{AFG}\alpha$. The A operator states that any infinite path starting in an initial state has to satisfy the property $\mathsf{FG}\alpha$. Thus for each path, it is checked whether there exists a point in time after finitely many steps, from which on at any point in the future the property α holds. This formula will be important later on.

3. Related Work

In this chapter some publications are presented which have been essential for this thesis: In section 3.1 the publication [Par70] is discussed, in which Park presented the concept of *xpoint induction*. In the second section 3.2 the *property directed reachability* (PDR) method will be discussed and recent improvements will be taken into account. Thus this presents the current state-of-the-art. In section 3.3 the publication [KS19] will be presented which was the starting point, as well as the foundation of the implementation done for this thesis. Lastly in section 3.4 a method for verifying fairness properties is presented based on [Bra+11], which shows the power as well as the weakness of safety model checking based on PDR.

3.1. Park's fixpoint rule

In 1970 Park published a paper called *Fixpoint Induction and Proofs of Program Properties* [Par70]. This paper is in so far important in the sense that it is not only the foundation of the method *Fixpoint Induction* but in this paper Park tries to close the gap between programs and computer science, as well as mathematical concepts. To this end, the program could be rewritten in an equation schema to formally verify certain properties.

By using equation schemes, it is also possible "to model computer programs involving (possibly recursive) subroutine calls, which program schemas could not" [Par70].

In general, Park only used "the fixpoint results on complete boolean algebras, which is the algebra of all subsets of a set" [Par70], being similar to the examples already shown in chapter 2.6, except for some constants. Then a resulting complete boolean algebra $\Lambda =< L, ^{0}, [, \setminus, 0, 1 >$ comprises a finite set L, the notation for *next states* ^{0}, the union of sets $[$, and the intersection of sets $\setminus$, as well as the two constants 0 and 1, and is therefore called *boolean* algebra. As can be seen, this is similar to the definitions given previously, and hence there exists a dual lattice theoretical approach to this boolean algebra.

Furthermore, Park already proved that given a boolean algebra D as well as an algebra $\Lambda_n(D)$ over the set D with all n-ary relations over D, then any predicate calculus formula C which involves only $8, 9, \_, \wedge, :$ and all occurrences of X_i in C are positive (covered by an even number of negations), $0 \quad i < \mathsf{N}$, then C is monotone in each X_i, $0 \quad i < \mathsf{N}$. This proof is also used to show the monotonicity and continuity of μ-calculus, since there the same rules apply, and thus μ-calculus is monotone. This, as well as the fact that all sets of states which are of interest are finite, allows the conclusion that all μ-calculus formulas indeed have a fixpoint. The crucial detail, in this

case, is that all the variables X_i, $0 \quad i < \mathsf{N}$ are positive, which is ensured in μ-calculus, since all variables need to be covered with an even number of negations to form a syntactically correct formula.

The paper [Par70] was published under the premise of machine intelligence, as stated by Park himself in the acknowledgement, and therefore focused on the use of first-order predicate logic. Nevertheless, his results are applicable in many other areas as will be seen in the rest of this thesis. His goal was to express different properties of programs in a first-order logic formula, like for example the *\strong" equivalence property*.

The *\strong" equivalence property* between program schemas states, that both of these schemas compute the same partial functions on all interpretations. Despite being seemingly simple, *strong equivalence* as well as *strong non-equivalence* is only partially decidable as shown in [LPP67] and this, in turn, states, "that valid formulas of such a formal system are not recursively enumerable"[Par70]. To express this, Park makes use of the *convergence* ($Conv$) which is informally the existence of fixpoints.

Given that C is a quantified boolean logic formula, thus only consists of $\mathsf{8}, \mathsf{9}, \_, \mathsf{\wedge}, :$ and all occurrences of variables X_i in C are positive, $0 \quad i < \mathsf{N}$, and is therefore monotone, as seen above, then $Conv$ is defined as shown in equation 3.1:

$$Conv(\mathsf{C})(x) \quad (\mathsf{8}X)[(\mathsf{8}y)[\mathsf{C}(X)(y) \ \mathsf{\$} \ X(y)] \ \mathsf{!} \ X(x)] \tag{3.1}$$

This is the second-order logic definition of $Conv$, while it is also possible to define $Conv$ in a lattice-theoretic approach, in which case c is a monotone map on a lattice:

$$Conv(c) =< Conv_0(c), Conv_1(c), \ldots Conv_{N \ 1}(c) > \tag{3.2}$$

In addition $Conv_i$ is defined as following:

$$Conv_i(c) = \bigcap \mathsf{f} A_i \mathsf{j} c(< A >) \quad < A > \text{ for some } \mathsf{f} A_k \mathsf{j} k \ \mathsf{6}= i \mathsf{gg} \tag{3.3}$$

Which could be described in turn in a second-order logic approach:

$$Conv_i(\mathsf{C})(x) \quad (\mathsf{8}X)[\bigwedge_{j=0}^{N \ 1} (\mathsf{8}y)[\mathsf{C}_j(X)(y) \ \mathsf{!} \ X_j(y)] \ \mathsf{!} \ X_i(x)] \tag{3.4}$$

Consider these formulas and the conclusions which could be reached by this.

In formula 3.3 the application of function c reduces the size of the set and therefore $c(< A >) \quad < A >$. This states that $< A >$ is a pre-fixpoint under the function c. By providing the least fixpoint for any subset of A this will result in the least fixpoint $Conv$ and thus $Conv$ is equivalent to the operator μ used in the previous section. One may also ask why we need second-order logic to express these properties and thus why first-order logic is not sufficient. The difference of first-order and second-order logic is the addition of relation-variables and therefore the possibility to quantify over different relations. Usually the relation-variables are further restricted by using monadic

second-order logic, which reduces the set of allowed relation variables, as the name *monadic* already suggest, to unary relations. This basically allows to quantify over sets, since unary relations describe certain properties of entities[1] and could therefore be used to describe sets. Yet still, second-order logic is strictly more expressive then first-order logic but with a rather important downside, namely that second-order logic is in general undecidable. Usual problems which are described in second-order logic, but can not be expressed in first-order logic are for example:

Reachability: It is not possible to give a single formula in first-order logic, which states that there exists a path to a given state for any model. The proof makes use of the *compactness theorem* and states that this formula need to have infinite length, since there are infinitely many different possible models, which leads to a contradiction.

Transitive closure: Similar to *reachability* it is not possible to state a first-order logic formula which describes the transitive closure of each given model. This again is because the transitive closure of infinite models (models with infinitely many elements in their corresponding *domain*) can not be expressed in a finite first-order logic formula.

Well-ordering [Boo75]: The well-ordering would result in a set like: f"R is a well-ordering", "$a_0 R a_1$","$a_1 R a_2$","$a_2 R a_3$",...g, which could not be expressed in first-order logic, which can again be easily seen by using the *compactness theorem*.

This list is not exhaustive and there are many other similar problems. All these problems however have different things in common, for example, all these could be calculated as well by fixpoint procedures (e.g. reachability could be reduced to a *safety property*, shown in formula 2.23). Furthermore as Boolos states: "Notions of infinity and countability can be characterized (or "expressed") by second-order sentences, though not by first-order sentences (as compactness and Skolem-Löwenheim theorems show)."[Boo75] Since the convergence of Park however allows the calculation of fixpoints it is possible to solve, for example, reachability. Therefore, the expressiveness of second-order logic is necessary, while first-order logic is not sufficient. This lengthy discourse is necessary, because this shows the complexity of the described methods and algebras as well as the expressiveness. In addition, this allows the connection to other fields of mathematics:

Set Theory: Around 1970 there was a discussion if set theory and second-order logic are actually the same. For example Quine (who is known for the *Quine{McCluskey algorithm* for boolean function minimization) expressed that "[second-order logic is] Set Theory in Sheep's Clothing"[Qui86], while Boolos [Boo75] proofed that there is still a difference

[1] Consider as example for unary relations in the domain of natural numbers the relation O_1 which states that an element is odd. Therefore $O(x)$ is true iff $x = 2k \quad 1$ for some k *2* N.

in assumptions necessary for set theory and second-order logic. Meanwhile, both concepts are used more or less synonym and problems could be reduced from one field into the other and vice versa. Yet still, the expression of convergence in second-order logic relates to the concept of convergence in set theory. This allows to project theorems and findings of set theory to fixpoint calculations and in turn allows to solve problems of set theory by the use of fixpoint calculations. This direction is not examined further in this thesis but might be an interesting research topic in the future.

Topologies: A further field of mathematics are topologies (the study of neighbourhood and relation). Convergence in metric spaces, therefore in spaces which provide a distance measurement, is known by ϵ-convergence or the limit function

$$\lim_{n \to 1} f(n) \tag{3.5}$$

which returns the value a function f converges to. Given spaces without a distance metric, this makes it more difficult to define convergence. Given such a topological space $(X, \mathcal{O})$ without a distance metric, a filter-function $\mathcal{F}$ in X and let $U(x)$ be an environment filter of $x \in X$, therefore providing the set of all environments of x. Then this filter F converges to x, if $U(x) \supseteq F$. If

$$\forall U. \forall F'. ((U \in U(x)) \wedge (F' \in F) \wedge (F \cap U \neq \emptyset)) \tag{3.6}$$

then x is called a touching point (*Berührpunkt*). Therefore the set of all these points B is given as

$$B = \bigcap_{F' \in F} \overline{F'} \tag{3.7}$$

with $\overline{F'}$ being the closure of F'. This is based on the translation of [Fil19].

This definition of filters on topologies is similar to the definition of convergence provided by Park, which further shows the similarities and also relates fixpoint calculations and topologies.

Given such a convergence criteria, Park was now able to define the *strong equivalence problem* mentioned earlier in a second-order logic sentence, given two equation schemas $\mathcal{C}$ and $\mathcal{D}$:

$$(\forall x)[Conv_i(\mathcal{C})(x) \Leftrightarrow Conv_j(\mathcal{D}(x))] \tag{3.8}$$

And therefore the strong equivalence problem is reduced to the validity of a second-order logic formula. Furthermore, by defining convergence in the previously mentioned way, Park concluded the weak fixpoint induction rule:

Definition 3.1.1 (Parks Weak Fixpoint Induction Rule [Par70])**.** *Given any formulas $\mathcal{F}$, $\mathcal{C}$, whenever*

$$(\forall x)[\mathcal{C}(\mathcal{F})(x) \rightarrow \mathcal{F}(x)] \tag{3.9}$$

is satis ed, then so is

$$(\forall x)[Conv(C)(x) \;!\; F(x)] \tag{3.10}$$

Consider this definition provided by Park. In less formal words this formula states, that given a function C, which is applied to the result of another function F and makes the result *smaller* (e.g. given a set of states F, then C will provide a subset of those), we can conclude that the convergence value $Conv$ of C will also stay inside this set. This fixpoint induction rule thus allows to deduce certain properties of least fixpoints, with the advantage of not having to do the fixpoint computation as a whole.

This deduction rule resembles the third Peano axiom[2], except some minor differences. One of this differences being, that there is no induction base and the more important difference is, that given Park's induction rule, this would also work on finite sets since it only requires a lattice structure as prerequisite. On the other hand, the Peano axioms define the set of natural numbers or a homomorphic set to the natural numbers, therefore a set which is necessarily infinite. By further consideration, this leads to a set of problems with the term induction in relation to finite sets.

Given a finite set, an induction proof will prove the inductiveness of a certain property, while inductiveness states that if the property holds in any arbitrary element it also needs to hold on all successor elements, with successor being defined by some order. Provided that there is an induction base it is quite obvious why the property will, therefore, hold on all elements. Since the set is finite, this could lead to one out of two problems. Either there is an element without any successor, then if this element is picked for the induction step, this will fail since there is no suitable successor. This problem however could be resolved if the structure forms a ring, which in turn will lead to the problem that there maybe is no *zero* or starting element, since there is not necessarily an element which is not successor of any other state (corresponding to the first Peano axiom, equation 2.3). Despite that, however, we will still call it fixpoint induction, even if the set of states in a Kripke structure would be finite.

[2]The *Peano axioms* provide the theoretical foundation of induction. See chapter 2.2 and equation 2.3 fur more details.

3.2. Property Directed Reachability (PDR)

Property Directed Reachability (PDR) is a recent model checking method with far reaching popularity and is the essential modern application of induction based model verification. This section will be structured in different subsections, the first giving a short overview to explain the concept, while the second section will focus more on the theoretical approach to this topic. The third section will explain the most essential extensions.

Overview

The most intuitive way of explaining PDR is the following: Given a Kripke Structure K as well as a safety property $\mathsf{AG}\alpha$. The model checking task is to prove that all states reachable in K, therefore all states which are on an arbitrary path starting in an initial state, do satisfy some property α. An inductive approach would be, to check if all initial states satisfy α, because if this is not the case this would already yield a counterexample. In this case, the proof would have concluded at that point, since obviously not all reachable states do satisfy α. The second step is to check whether the property itself is inductive, therefore to check if all the successors of the states where α does hold, also satisfy α. In this case since all initial states satisfy α and all successor states of states which satisfy α also do satisfy α, obviously, α will hold on all reachable states. In practice however there could be states where α is not inductive relative to those states, but those states may not be reachable. In this case, the property α is not inductive, but $\mathsf{AG}\alpha$ could still hold anyway and the states in which α holds are maybe an overapproximation of the reachable states. Therefore the *bad* states which break the inductiveness are checked for reachability and if unreachable are excluded from the inductiveness check. This procedure will, therefore, conclude either in a counterexample, or yield an inductive property which implies the property α (therefore describes a subset of α-states) and is maybe an overapproximation of the reachable states. The benefit of this procedure is, that the reachable states do not have to be calculated, instead, an over-approximation is sufficient. Furthermore, when states are excluded from checking inductiveness which are not reachable, this could be further improved by some generalizations and thus PDR has the potential to be very efficient.

Theoretical description

Given a Kripke structure $K = \{V, \Psi_I, \Psi_R\}$, which consists of a set of (state-) variables V, the set of initial states Ψ_I and the transition relation Ψ_R. In general all Ψ denote propositional logic formulas, which result in a set of states of a corresponding Kripke structure. As previously mentioned a state s of K is given as a *cube*, therefore as a conjunction of literals (a variable or its negation), which is usually the label of the state and represents an assignment to all variables. Thus if s is an assignment to all variables of a formula F, s either satisfies F which is denoted as $s \models F$, or the assignment falsifies F

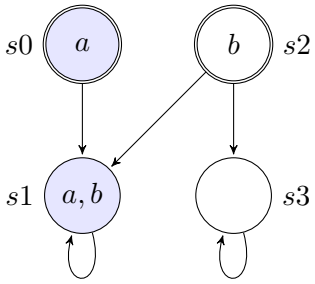

Figure 3.1.: *A structure to visualize the inductiveness of property a. All accepting states are marked in color for better distinction.*

which is in turn denoted as $s \not\models F$. If $s \models F$, s is also called a model of F, if s is interpreted as a state, it is an F-state and if s is interpreted as a formula and models F, the formula $s \rightarrow F$ is valid. Furthermore, $\neg s$ could be represented as a clause (a disjunction of literals) by applying De Morgan's rule (negate all literals, change all $\wedge$ to $\vee$). These clauses could now be represented as a set c and d is called a *subclause* if $d \subseteq c$. It is important to note that dropping a variable in a cube increases the number of states this cube represents while dropping a variable in a clause will decrease the number of states. Contrary, adding a formula G to F with a conjunction, this will decrease the set of states and adding a formula H to F with a disjunction this will increase the set of states. A *trace* $s_0, s_1, s_2, \ldots$, is a sequence of states, such that $s_0 \models \Psi_I$, and for all (s_i, s_{i+1}) in the sequence $s_i, s_{i+1} \models T$, thus a trace is a path of states starting in an initial state and all states on all traces are called reachable.

These concepts are fairly important, therefore consider the example shown in figure 3.1. The formula a would for example describe the two states $s0$ and $s1$, appending as additional constraint another literal $\neg b$ with a conjungtion to the formula would result in only the state $s0$, therefore a propositional logic formula in the context of Kripke structures allows to describe a set of states. Additionally $s0, s1 \models a$ and $s0$, $s1$ are called a-states. The cube $a \wedge \neg b$ describes the single state $s0$ and correspondingly $\neg s0$ could be constructed by De Morgan as $\neg a \vee b$, which is a clause. Dropping one variable of the clause, e.g. the $\neg a$, will result in a subset of those states, namely $\{s1, s2\}$ $\{s1, s2, s3\} = \neg s0$.

A property α is inductive if $\alpha \rightarrow \alpha$ is valid, therefore if α is considered to be a set of states, then all successors of α need to be a subset of α. Consider the inductiveness of b in figure 3.1. Since $s2$ is an b-state, but its successor $s3$ is a $\neg b$-state, thus $b \rightarrow b$ is not valid and $s2$ is the corresponding counterexample. Given $a = \{s0, s1\}$, it is inductive since $a = \{s1\}$ and thus $a \rightarrow a$. Furthermore F is inductive relative to G, if

$$F \wedge G \rightarrow F \tag{3.11}$$

$$F \wedge G \wedge T \rightarrow F' \tag{3.12}$$

and F' denotes that all variables of F are primed and thus mark next states.

This is sometimes also called *inductive strengthening* [Bra11]. Given for example figure 3.1, the property a is even inductive relative to $:b$, since this would reduce to a holding in all successor states of $s0$. Furthermore the reachable states Ψ_{reach} are inductive by definition [LS16], and could be defined with inductive means. This leads to the inference rule [Sch19b]:

$$\frac{\Psi_I \;!\; \Phi \quad \Psi_{reach} \wedge \Phi \;!\; \Phi}{\Psi_{reach} \;!\; \Phi} \tag{3.13}$$

This rule states that given the upper part, the lower part could be deduced. In this case especially that given a property Φ which contains the initial states Ψ_{reach} as well as Φ being inductive relative to Ψ_{reach}, leads to the fact that all reachable states Ψ_{reach} also satisfy property Φ. This rule closely resembles Park's induction rule (see chapter 3) and the proofs of correctness could be found in [Sch19b]. It should be noted that this is equivalent to $\mathsf{AG}\Phi$, which is a *safety property*. Furthermore, this rule is essentially an induction since we start with an induction base $\Psi_I \;!\; \Phi$ and an induction step $\Psi_{reach} \wedge \Phi \;!\; \Phi$. Assuming the induction base would not hold, there would be an initial state violating the property and given that the induction step would not hold, then there needs to be a reachable successor state violating given property [Sch19b].

This rule describes the core of the *Property Directed Reachability* (PDR) algorithm, initially implemented under the name *Incremental Construction of Inductive Clauses for Indubitable Correctness* (IC3) in 2011 by A. Bradley [Bra11]. The intention of PDR was twofold [Bra11]:

> Construct an algorithm which resembles the human approach to model verification.
>
> Make use of easy to solve SAT-instances and therefore avoid unrolling of the transition relation.

Given for example equation 3.13, which allows to deduce $\mathsf{AG}\Phi$. This formula requires to still calculate the set of reachable states, which is actually hard to do because determining the reachable states requires a fixpoint calculations, often also describes as *unrolling* of the transition relation. The most common approach to do this is by starting with $\Psi_I \wedge T$, which does result in a set of next states, which is joined with the initial states. This procedure is repeated until a fixpoint is reached.

Given for example an automaton as shown in figure 3.2. This automaton has an infinite path, visiting each state exactly once, until looping in some state s_n. With a fixpoint calculation of the reachable states, this would add exactly one state for each iteration and the calculation of this next state could be fairly expensive. To avoid this Bradley tried to avoid unrolling of the transition relation and therefore also avoid the explicit calculation of the reachable states. Additionally, Bradley suggested a human-like approach to model checking, which could be stated as follows: Try to find a superset Ψ of the reachable states, which is also inductive, thus $\Psi \;!\; \Psi$ and which implies the property

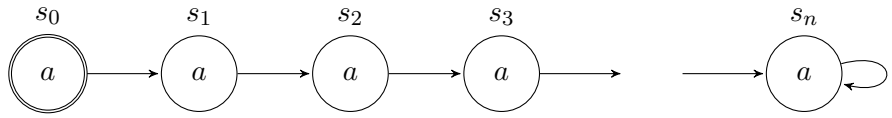

Figure 3.2.: *A worst case example for reachability calculation. Each unrolling of the transition relation adds exactly one state, therefore the transition has to be unrolled $n + 1$ times until a fixpoint is reached. Note that each state has additional variables to make each label unique, but those are not important for further consideration and are therefore omitted.*

Φ. This Ψ is called an inductive assertion and results in the following deduction rule [Sch19b]:

$$\frac{\Psi_I \implies \Psi \quad \Psi \implies \Psi \quad \Psi \implies \Phi}{\Psi_{reach} \implies \Phi} \tag{3.14}$$

The problem in this case is the calculation of such a Ψ, despite there being some procedures to calculate them (e.g. *interpolation based model checking*, making use of *craig-interpolants* [McM03]). However, Bradley significantly improved this by using a human-like approach to the calculations of the inductive assertion, which is not only significantly faster then previous methods [EMB11] but allows to use additional information to further increase efficiency [LS16]. The human-like approach to calculate such an inductive assertion would be an incremental refinement, by checking the inductiveness of a set. Given the set is already inductive, no refinement is needed, however, if the check fails, this will result in one of either case:

> Either the counterexample is reachable and thus the property checked does not hold
>
> or the counterexample is not reachable, is therefore called a *counterexample to induction* (CTI) and the inductive assertion is refined by removing the CTI from the inductive assertion.

Bradley states, that this also resembles the way professional manually checked models in the past [Bra11]. Furthermore, there is a huge potential for optimization when some assumptions over the labels of a model are met. For example, when transforming some software or hardware system to a Kripke structure, it is reasonable to assume that unreachable states are somewhat similar. As an example consider a piece of software where some parts of the software are not reachable, then this will be most likely because of some constraint on variables (e.g. some combinations of variables block the execution of an if-statement). The resulting states will, therefore, most likely share a decent amount of variables. Bradley proposed to abuse this fact, by generalizing CTIs before removal and thus remove potentially many states at once. This concept results in an incremental induction rule [Sch19b]:

$$\frac{\Psi_I \implies \Psi_0 \quad \bigwedge_{i=0}^{k-1} \Psi_i \implies \Psi_{i+1} \wedge \quad \Psi_{i+1} \quad \Psi_k \implies \Phi \quad \bigvee_{i=0}^{k-1} \Psi_i \quad \Psi_{i+1}}{\Psi_{reach} \implies \Phi} \tag{3.15}$$

$$\begin{array}{ccccccccccc} [\![\Psi_{\mathcal{I}}]\!]_{\mathcal{K}} & = & \mathcal{X}_0 & \subseteq & \mathcal{X}_1 & \subseteq & \ldots & \subseteq & \mathcal{X}_k & \subseteq & \mathcal{S}_{\mathsf{reach}} \\ \| & & \| & & \cap & & & & \cap & & \\ [\![\Psi_{\mathcal{I}}]\!]_{\mathcal{K}} & = & [\![\Psi_0]\!]_{\mathcal{K}} & \subseteq & [\![\Psi_1]\!]_{\mathcal{K}} & \subseteq & \ldots & \subseteq & [\![\Psi_k]\!]_{\mathcal{K}} & \subseteq & [\![\Phi]\!]_{\mathcal{K}} \end{array}$$

Figure 3.3.: *A visualization of the overapproximation of the sets X_i by the sets $\mathtt{J}\Psi_i\mathtt{K}_K$, which are in turn a subset of $\mathtt{J}\Phi\mathtt{K}_K$. Depending on construction, the sets X_i could be the set of states reachable in i steps and $\mathtt{J}\Psi_i\mathtt{K}_K$ the corresponding overapproximation. Figure from [LS16].*

Consider the formula 3.15. The first part $\Psi_I \;!\; \Psi_0$ states that the initial states should be part of Ψ_0, while the third part $\Psi_k \;!\; \Phi$ states that Ψ_k has to satisfy Φ. Those two assertions closely resemble formula 3.14 and are essentially the induction base and hypothesis. The most notable difference is the second part $\bigwedge_{i=0}^{k\ 1} \Psi_i \;!\; \Psi_{i+1} \wedge \;\Psi_{i+1}$, but it should be noted that $a \;!\; b \wedge c$ could be rewritten as $a \;!\; b \wedge a \;!\; c$, therefore this part makes two assertions: (1) $\Psi_i \;!\; \Psi_{i+1}$, therefore Ψ_i is a subset of Ψ_{i+1} and (2) $\Psi_i \;!\; \;\Psi_{i+1}$, which states that Ψ_i is also a subset of all successors of Ψ_{i+1}. By that a sequence of sets of states $\mathtt{J}\Psi_i\mathtt{K}_K$, $i = 0 \ldots k$, is generated where each $\mathtt{J}\Psi_i\mathtt{K}_K$ is an overapproximation of the states X_i, which is also shown in figure 3.3. Given the forth assumption $\bigvee_{i=0}^{k\ 1} \Psi_i \;\; \Psi_{i+1}$, which essentially states that we reached a fixpoint, since there is a $\Psi_j \;\; \Psi_{j+1}$, then this ensures in conjunction with the second part that Ψ_j is inductive, since

$$\Psi_{j+1} \;!\; \Psi_{j+1} \wedge \;\Psi_{j+1} \;\; \Psi_{j+1} \;!\; \;\Psi_{j+1} \tag{3.16}$$

More informal this means, that we try to find a sequence of sets of states which over-approximate the corresponding set of states reachable in this many steps, but we only refine the sets further if it is not yet inductive. By default the sets X_i are only restricted to $X_i \;\; \mathtt{J}\Psi_{reach}\mathtt{K}_K$ by the proof rule but could be interpreted as the set of states reachable in i many steps, depending on the construction of Ψ. The proof is stopped if one Ψ_j is inductive, because this already results in a proof, since $(\Psi_j \;!\; \Phi) \wedge (\Psi_j \;!\; \;\Psi_j) \wedge (\Psi_I \;!\; \Psi_j)$. This resembles the assertions of the basic induction formula 3.14. Therefore this formula provides in inductive and constructive proof method of a fixpoint calculation.

In 2011 Bradley published an initial version of PDR which implements those formulas ([Bra11]) and a reduced version of this implementation is shown in figure 3.4. Next, this implementation will be discussed in detail.

First of all, it is important to note that the system S, as well as the property P, are in scope everywhere and thus accessible at any point of execution. Furthermore this program is started by calling the top-level function `prove()`. After the function `prove()` is called, two initial checks are performed and at which point a SAT-solver is called. While Bradley used ZChaff for SAT-solving because "it offers efficient incremental functionality, since clauses can be pushed and popped, which is necessary for finding an inductive subclause" [Bra11], Een refined the implementation to be independent of the choice of

used SAT-solver [EMB11].

The two initial equations which are checked for satisfiability are:

$$I \wedge \neg P \tag{3.17}$$

$$I \wedge T \wedge \neg P^{\emptyset} \tag{3.18}$$

Whereas $P^{\emptyset}$ states, that all occurrences of variables in P are primed and thus P has to hold in all successor states. To get this expression in line with the notation used earlier and given that $\Psi_I := I$, $\Phi := P$ and $K := S$, these checks could also be expressed as ([Sch19b]):

$$[\![\Psi_I]\!] \implies [\![\Phi]\!]_K \stackrel{?}{=} S \tag{3.19}$$

$$[\![\Psi_I]\!] \implies [\![\Phi]\!]_K \stackrel{?}{=} S \tag{3.20}$$

Which could be read as "For each state s in S, if s is an initial state, s and all its successors have to satisfy property Φ".[3] Therefore this check makes sure that no counterexample could be reached in none or only one step and in turn means, that if either of these formulas is satisfiable, then there is an initial state or a successor of an initial state which already harms the desired property. Furthermore, there are two more initial checks proposed in [Sch19b]:

$$[\![\Psi_I]\!] \implies [\![\Psi_I]\!]_K \stackrel{?}{=} S \tag{3.21}$$

$$[\![\Phi]\!] \implies [\![\Phi]\!]_K \stackrel{?}{=} S \tag{3.22}$$

which check whether the property itself or the initial states are already inductive, since if either of these holds, the proof is trivial. This is because, combined with the first check that all initial states have to satisfy the property Φ, given that Φ or Ψ_I are inductive, this is sufficient to prove that all reachable states satisfy Φ.

If all initial checks are completed, a trace of F_i is instantiated, where F_0 denotes the initial states and all other F_i are interpreted as $P \wedge \bigwedge clauses(F_i)$, thus as a conjunction of clauses with the property P. At this point, it is important to note that Bradley used a "syntactic approach" and uses sets of clauses. Therefore equivalence checking is reduced to syntactic equivalence checking, which is far easier then semantic equivalence and the generalization of counterexamples gets simplified because the generalization is essentially archived by "dropping" literals from a clause and checking reachability.[4]

If the algorithm so far is compared to formula 3.15, a few similarities are prominent. First of all by setting up the trace with $\Psi_0 := \Psi_I$ the first assertion $\Psi_I \implies \Psi_0$ is trivially satisfied. Furthermore, the F_i resemble to the Ψ_i and

[3] As reminder: The double square brackets ($[\![]\!]$) with an subscribed structure K state, that this expression returns the set of states which satisfy the property inside the brackets

[4] For completeness it should be noted that different approaches might be possible, for example using BDDs or similar structures instead of sets of clauses for implementation, as for example done in the implementation of some teaching tools, accessible under https://es.cs.uni-kl.de/tools/teaching/. The main concept of PDR will, however, stay the same.

```
1   // The system S : (V, I, T) and property P are in scope everywhere
2   bool prove():
3       if sat(I ^ : P) or sat(I ^ T ^ : P'):
4           return false
5       if not sat(I ^ T ^ : I') or not sat(P ^ T ^ : P'):
6           return true
7       F_0 := I, clauses(F_0):= ;
8       F_i := P, clauses(F_i):= ; for all i>0
9       for k := 1 to ...:
10          if not strengthen(k):
11              return false
12          propagateClauses(k)
13          if clauses(F_i)=clauses(F_{i+1}) for some 1   i   k:
14              return true

16  bool strengthen(k:level)
17      try:
18          while sat(F_k ^ T ^ : P'):
19              s :=  the predecessor extracted from the witness
20              n := inductivelyGeneralize(s,k-2,k)
21              pushGeneralization({(n+1,s)},k)
22          return true
23      except Counterexample:
24          return false

26  void propagateClauses(k:level):
27      for i:=1 to k:
28          for each c 2 clauses(F_i):
29              if not sat(F_i ^ T ^ c'):
30                  clauses(F_{i+1}):=clauses(F_{i+1}) [ {c}

32  level inductivelyGeneralize(s:state, min:level, k:level)
33      if min<0 and sat(F_0 ^ T ^ : s ^ s'):
34          raise Counterexample
35      for i:= max(1,min+1) to k:
36          if sat(F_i ^ T ^ : s ^ s'):
37              generateClause(s, i-1,k)
38              return i-1
39      generateClause(s,k,k)
40      return k

42  void generateClause(s:state, i:level, k:level)
43      c :=  subclause of : s that is inductive relative to F_i
44      for j:=1 to i+1:
45          clauses(F_j) := clauses(F_j) [ {c}
46
```

Figure 3.4.: *A reduced version of the initial pseudo-code of PDR as presented by Bradley. A more detailed version with additional assertions, as well as pre- and post-conditions, could be found in [Bra11]. Lines 5-6 are proposed in [Sch19b] and not originally by Bradley. Furthermore Een introduced a more efficient implementation of PDR and published the corresponding pseudo-code in [EMB11], but his version is less suited for understanding the underlying concept.*

```
1   void pushGeneralization(states:(level,state) set, k:level):
2       while true:
3           if n>k: return
4           if sat(F_n ^ T ^ s'):
5               p := the predecessor extracted from the witness
6               m :=inductivelyGeneralize(p, n-2, k)
7               states := states [ f(m+1,p)g
8           else:
9               m := inductivelyGeneralize(s,n,k)
10              states := states n f(n,s)g [ f(m+1,s)g
11
```

Figure 3.5.: *The* `pushGeneralization()` *function published by Bradley in [Bra11]. This function maintains a set of level and state tuples and provides a way to push clauses to higher* F_i.

since each F_i is set up by using the desired property as well as a conjunction of further clauses, the third assertion $\Psi_k \;\; ! \;\; \Phi$ is also trivially met. In line 11 is stated, that the algorithm is completed as soon as two F_i and F_{i+1} are identical, which satisfies the fourth assertion $\bigvee_{i=0}^{k \;\; 1} \Psi_i \quad \Psi_{i+1}$. To satisfy the second assertion $\bigwedge_{i=0}^{k \;\; 1} \Psi_i \;\; ! \;\; \Psi_{i+1} \; ^ \quad \Psi_{i+1}$ two more functions are called, the first being `strengthen()` and the second being `propagateClauses()`. At this point all F_i are already initialized and therefore always in scope, especially when calling `strengthen()` and `propagateClauses()`.

Consider the second function `propagateClauses()` (Figure 3.4, lines 26-30) first, given a set of F_i, while each F_i is a set of clauses. The goal is to check for any F_i, $0 \quad i \quad k$, if there is a clause c (note that this describes a set of c-states $\mathtt{J}c\mathtt{K}_K$) which could be added to F_{i+1}. Thus for each clause c of F_i the property

$$F_i \; ^ \; T \; ^ \; c' \tag{3.23}$$

is checked for unsatisfiability. This property could also be stated as

$$\mathtt{J}F_i \;\; ! \quad c\mathtt{K}_K \stackrel{?}{=} S \tag{3.24}$$

and thus we check if c holds for all successors of $\mathtt{J}F_i\mathtt{K}_K$. For any (F_i, c) where this check holds, c will be propagated to F_{i+1}.

Consider the main function `prove()` as discussed so far. We set up the proof by doing some initial checks, while the first two checks (line 3-4) would yield an immediate counterexample, while the two additional checks (line 5-6) yield an immediate proof. After that a set of F_i is initialized, with F_0 containing only the initial states and all other F_i are initialized with the property P. After that a refinement iteration is started, where first `strengthen()` and then `propagateClauses()` is called, and the algorithm aborts as soon as some F_i and F_{i+1} are identical. Also `propagateClauses()` tries to push each clause of each F_i to as many other F_j as possible and therefore ensures that an F_i is identical to F_{i+1} as soon as F_i is inductive. Consider why this is the case: `propagateClauses()` pushes each clause c to the next F_i, if all successor states of $\mathtt{J}F_i\mathtt{K}_K$ are c-states. Thus as soon as an F_i is inductive, each successor state

of F_i is also an F_i-state and thus also a c-state for any clause c of F_i. From that directly follows that as soon as an F_i is inductive, F_i is equal to F_{i+1}.

The most complex function of this algorithm is therefore the `strengthen()` function. The goal of this function is to determine for a certain level k, which is an overapproximation of states reachable in k many steps, if there is a counterexample reachable from this set. If a counterexample is found, it is determined if this counterexample is a real counterexample and thus the proof fails, or if this counterexample is a counterexample to induction (CTI), thus a state which is not reachable. Given it is a CTI, a more generalized set of similar states is calculated and excluded from F_k by adding an additional clause to as many F_i (and especially to F_k) as possible. First of all, it is checked if there is a counterexample reachable from F_k and therefore the satisfiability of $(F_k \wedge T \wedge :P^0)$ is checked. If satisfiable, this would mean that there is a successor state of F_k which is a $:P$-state. First, consider both cases. In the first iteration, this check has to be satisfied since F_1 is instantiated with P, but $P \implies P$ does not hold and thus P is not inductive, as already checked earlier in line 5. Therefore there has to be a counterexample in the first iteration of the loop. As a remark, the result of a SAT-solver is called a witness, because it is an assignment which satisfies a certain formula and thus proofs the satisfiability.[5] It should be noted that this state s, which is a counterexample, is a predecessor of a $:P$-state, but it needs to be determined if s is either a real counterexample or a CTI and thus `inductivelyGeneralize()` is called.

In `inductivelyGeneralize()` there are three cases which could occur, since in either case the function is aborted: (1) the given state s is indeed a counterexample, since it is reachable from an initial state where s does not hold or (2) s is reachable from any previous F_i, $min+1 \quad i \quad k$, up to F_k or (3) s is not reachable from any F_i, $min+1 < i <= k$.[6]

Thus the result of `inductivelyGeneralize()` in case (1), is that s is indeed a real counterexample, which raises a `counterexample`-exception if s is reachable from an initial state and $min < 0$, while $min \quad 0$ denotes that $:s$ is inductive relative to F_{min} and thus $\mathtt{J}F_{min} \wedge :s \implies :s\mathtt{K}_K = S$. In case (2), thus if s is reachable from an F_j, $min+1 \quad j \quad k$ a subclause c of $:s$ is added to all F_i with $1 \quad i \quad j$, where c has to be inductive relative to $F_{j\ 1}$. Case (3) is almost the same as (2) except the difference that c has to be inductive relative to F_k and is added to all F_i with $1 \quad i \quad k+1$.

While this is quite complicated at first glance, this could also be expressed more informally: Given a state s which is a counterexample, as well as a min level and the current level k of consideration, `inductivelyGeneralize()` constructs a subclause $c \quad :s$, which is inductive relative to some F_i, $min \quad i \quad k$ and adds this clause to all F_j, $1 \quad j \quad i+1$, thus generating a clause c which describes a subset of $:s$-states, where as many states as possible, which are not reachable, are excluded. This results in an as strong as possible con-

[5] Assignments trivially correspond to states as discussed earlier in 2.3.

[6] Note that min is instantiated with $k \quad 2$ in `strengthen()` or with n or $n \quad 2$ in `pushGeneralization()`, $1 \quad n \quad k$, and is incremented by one in line 35 in figure 3.4.

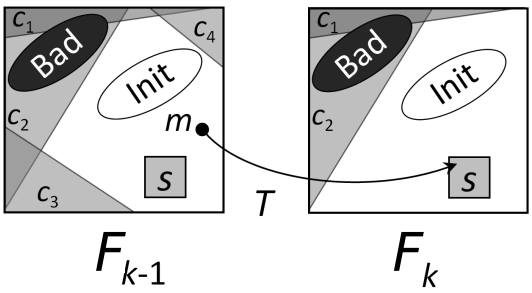

Figure 3.6.: *Given $F_{k\ 1}$ and F_k, s is inductive relative to $F_{k\ 1}$, if there is a state m in $R_{k\ 1}$ from which any state in s could be reached with one step via the transition T. This figure is based on [EMB11].*

straint c on F_i and thus reduces the amount of F_i-states, whereas only those states are dropped from F_i which are not reachable by F_i anyway and are thus not reachable in i many steps. Therefore all F_i are at any point an over-approximation of states reachable in i many steps. The problem introduced by `inductivelyGeneralize()` however, is that this clause c which is added could be only inductive to a single F_i with $i < k$. In this case the function `pushGeneralization()` shown in figure 3.5 takes over.

Consider figure 3.6, which visualizes the inductiveness of some clause $:s$. It should be noted that the set of clauses constraining $F_{k\ 1}$ as well as F_k are visualized, as well as the fact that clauses restricting F_k are a subset of clauses restricting $F_{k\ 1}$. Furthermore the white region satisfies $F_{k\ 1} \wedge :s$, thus inductiveness of s could be checked with a SAT-solver by calling

$$SAT?[F_{k\ 1} \wedge :s \wedge T \wedge s^{\prime}] \tag{3.25}$$

which could be read as: "Is there a state in $F_{k\ 1}$, which is not an s-state, but could reach an s-state with one transition T." Given that this query is UNSAT, this results in a proof of the following statement:

$$(F_{k\ 1} \wedge :s \wedge T) \ \Rightarrow \ :s^{\prime} \tag{3.26}$$

and thus $:s$ is inductive relative to $F_{k\ 1}$.

The result of the function `pushGeneralization()` is to exclude the counterexample not only with a clause inductive relative to an F_i with $1 < i < k$, but with a clause inductive relative to F_k by calling `inductivelyGeneralize()` with different *min* values, until the generated clauses of the different iterations of `inductivelyGeneralize()` is inductive relative to F_k. The only query which has to be considered in this function is $(F_n \wedge T \wedge s^{\prime})$, thus if s is reachable from F_n. If this is the case, also this predecessor p of s is removed, again by calling `inductivelyGeneralize()` with p. It should be noted that this function takes a set of level and state tupels to keep track of the potential counterexamples and to which F_i they are already inductive. This is necessary, since then `pushGeneralization()` is enforced to terminate, despite potential cycles in the given structure.[7]

[7] The proof of termination and correctness could be found in [Bra11], as well as pre- and post-conditions of all functions.

While the procedures seem complicated it should be noted that all of the queries of the SAT-solver are quite easy, since they involve formulas of limited size, furthermore the use of indices might be confusing, but it should be considered that these queries are potentially executed hundreds of thousands of times, thus even small improvements as checking $min < 0$, which is only a bit comparison, instead of $min = \quad 1$ could increase the performance, depending on the used processor, to a significant amount.

A few things are remarkable following this discussion. First of all, in each iteration, a counterexample is removed with a recursive iteration over the predecessors of this state, while each generated F_i describes an overapproximation of the states reachable in i steps and by construction $F_i \; ! \; F_{i+1}$ holds since the clauses are always added to all previous F_i. Furthermore since all predecessors of counterexamples are also removed (see `pushGeneralization()`), this implies $F_i \; ! \quad F_{i+1}$ for all i. These two findings may be familiar, since they resemble the second assumption of formula 3.15, namely

$$\bigwedge_{i=0}^{k \;\; 1} \Psi_i \; ! \; \Psi_{i+1} \; \wedge \quad \Psi_{i+1} \tag{3.27}$$

This closes the gap between the code proposed by Bradley in [Bra11] as well as the inference rules previously shown which are inspired by [Sch19b].

There should be a few takeaways from this: First of all it should be noted, that the formulas initially derived introduce some uncertainties on how to actually calculate certain elements, while the code shows that an actual implementation of these formulas only depends on limited size SAT queries and the used queries are quite simple.

Important improvements and limitations of PDR

While this is not an exhaustive list of published improvements to PDR, there are two extensions which are especially relevant. The first was proposed by Een et al. in 2011 (see [EMB11]) and the second published by Li et al. in 2016 ([LS16]).

Efficient Implementation of Property Directed Reachability

Shortly after the initial publication of PDR (at that point the implementation was called *Incremental Construction of Inductive Clauses for Indubitable Correctness* (IC3)) Een proposed a set of improvements to (IC3) and actually proposed the name *Property Directed Reachability* (PDR) for the method itself in [EMB11]. While the details of implementation are not important for this thesis a few minor things should still be considered.

The most impactful improvement proposed in [EMB11] was the use of ternary logic instead of boolean logic. This theory is known from electrical engineering and resembles the behaviour of an analogue circuit. Instead of each variable having two values ("True" and "False") there is a third value ? introduced, which states that this variable is not yet known to be true or false

A	: A
?	?
0	1
1	0

A ^ B		B		
		?	0	1
A	?	?	0	?
	0	0	0	0
	1	?	0	1

A _ B		B		
		?	0	1
A	?	?	?	1
	0	?	0	1
	1	1	1	1

Table 3.1.: *An interpretation of ternary logic by Kleene and Priest ([Thr19]). Note that ? is similar to an "unknown" value.*

and is called "bottom". There are many different interpretations of ternary logic, but a well-known form is Kleene and Priest logics shown in table 3.1.[8] The benefit of using ternary logic is an efficient decision on which variable in a formula has an impact on the satisfiability of this given formula. Een used this property of ternary logic to significantly improve the efficiency of the generalization of clauses and of the proof of reachability in one step(see [EMB11]). Another advantage of this method was, that this allowed to be independent of the used SAT-solver and thus Een managed to provide a clean SAT-solver interface. It can, therefore, be assumed that PDR is completely independent of the used SAT-solver.

Control-flow Guided Property Directed Reachability for Imperative Synchronous Programs

Another improvement to PDR was proposed by Li in 2016 (see [LS16]). What is notable in this case especially is that the concept is more important than the actual implementation and this concept is actually applicable to further different but similar cases. In the initial implementation of PDR, namely IC3, Bradley already made some considerations which made PDR as effective as it is, while a large part of said efficiency is the generalization of counterexamples. This generalization allows to get rid of potentially large areas of states similar to the found counterexample and thus decreases the number of necessary iterations potentially to a significant amount. This is because Bradley assumed that the given structures are related to either software or hardware circuits. By this, it is reasonable to conclude that most likely unreachable states share some properties and are thus *similar* to each other.

This is because the transition relation of these structures often only depends on a subset of variables. This potentially results in *areas* which share most properties and differ in the free variables of the transition relation. Depending on the transition relation, those areas could then either be reachable or unreachable. This can lead to large unreachable areas which are similar in most variables and this is the reason for the generalization of counterexamples in PDR. Consider for example a structure derived from an implementation. It is reasonable to assume that some conditions assert configurations of variables, however, this will lead to states for which these assertions do not hold and which are therefore unreachable. Those states all share that the assertions

[8] Note that negation with conjunction and disjunction is a complete operator basis.

made in the implementation do not hold.

In her publication, Li proposed to not only exclude counterexamples by generalization but derive further information of reachability in a pre-processing step. By default, this may seem counterintuitive, because this defeats the purpose of not having to explicitly calculate reachability, which is one of the benefits of PDR. However, one has to consider that programs are usually coded with a certain intention and for verification purposes, hardware circuits are usually translated to a suitable synchronous programming language (see [LS16]). These programming languages (e.g. Quartz) provide a way to efficiently derive control-flow information while compilation of the program and these control-flow information could then be used to restrict the set of reachable states. More detailed information could be found in [LS16], especially regarding implementation as well as some experimental results. The key point to take away from this, is however, that there might be further information regarding control flow or invariants which are already provided by the means of implementation or potentially in some other ways by the construction of the system which should be verified. The use of this information could then greatly increase the efficiency because a large state space does not have to be considered for reachability. Furthermore, it should be noted that it is very easy to embed this information in the PDR approach since the sequence of Ψ_i just has to be initialised with additional clauses representing these restrictions. Optimizations like these could, therefore, increase the efficiency by a considerable amount, but are depending on the actual use-case as well as the availability of information restricting the reachable states of the system.

3.3. Further Induction Rules for Temporal Logic

Despite its popularity and being considered the state of the art approach for model checking, PDR, as introduced previously, was only able to prove safety properties (see [KS19]). Therefore there existed a method which is known to be efficient and correct, scales well, is well understood and, as shown previously, already improved by extensions to a significant amount. However, that did not change the fact that only safety properties could be proven with this method. As a reminder, a safety property is a temporal logic property, whereas this property has to hold in each reachable state or informally at any point in time no matter what.

This restriction is tied to the fundamentals of the PDR method itself and all of its development so far. Despite being a weakness of the proof procedure, the restriction to safety properties is not as significant as it may seem to be, because in practice still a lot of interesting properties could be expressed as a safety property. For example the unreachability of harmful states x could be checked with the contrary proof goal of $\mathsf{AG}\,x$.

Furthermore, implementations of PDR are essentially an implementation of the incremental induction rule shown in formula 3.15, whereas the benefits of the speed up and the efficiency are based on the generalization of counterexamples and the fact that an actual calculation of the reachable states could be avoided, if there is an inductive set of states which is sufficient to show the safety property.

However, there where some induction-based proof rules introduced by Köhler in 2019 [KS19] which are of a similar style to the incremental induction rule (formula 3.15) and thus to the foundation of PDR but those new rules allow to construct proofs for all CTL properties (see chapter 2.7).

First consider the incremental induction rule again (formula 3.15):

$$\frac{\Psi_I \;\Rightarrow\; \Psi_0 \quad \bigwedge_{i=0}^{k-1} \Psi_i \;\Rightarrow\; \Psi_{i+1} \wedge \quad \Psi_{i+1} \quad \Psi_k \;\Rightarrow\; \Phi \quad \bigvee_{i=0}^{k-1} \Psi_i \quad \Psi_{i+1}}{\Psi_{reach} \;\Rightarrow\; \Phi}$$

This is an inference rule and, since it is correct, it states that the lower part holds if the upper part holds. Furthermore, since it is complete, if the lower part holds, there is a solution for the upper part.[9] So, in this case, this means that given the structure satisfies some Φ in all states on all infinite paths, then there is a sequence of Ψ_i, which is growing, and each Ψ_{i+1} only consists of states from Ψ_i, as well as states which are a successor to Ψ_i. In addition, Ψ_0 has to contain the initial states, all states in the last Ψ_k have to satisfy Φ and there has to be a point at which we reach an inductive Ψ_i. While this rule has already been discussed earlier and the proof could be found in [Sch19b], some more properties of the formula shall be discussed at this point.

As shown in figure 3.7, by enforcing $\Psi_k \;\Rightarrow\; \Phi$, all other Ψ_i also have to satisfy Φ. This is because Ψ_{i+1} is only allowed to contain states of Ψ_i as well

[9] While correctness of the rule ensures that the conclusion could be reached by satisfying this set of assertions, completeness of the rule ensures, that given the conclusion holds, there exists a solution for the upper part. Therefore it is possible to provide different inference rules for the same conclusion which could be complete and correct.

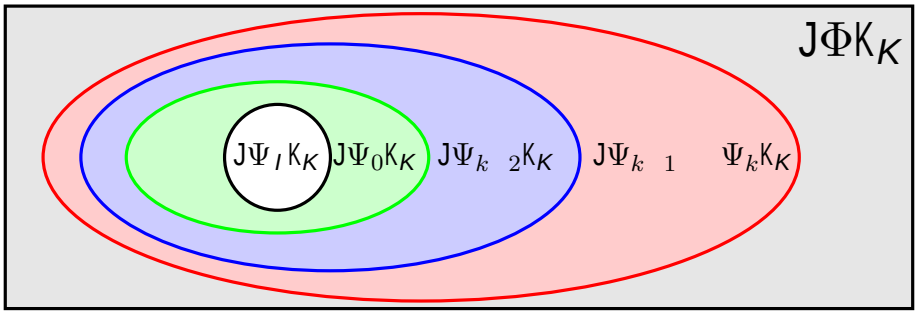

Figure 3.7.: *A visualization of the sequence of sets generated by formula 3.15 or 3.28.*

as successors of Ψ_i. In addition, if $\Psi_{j\ 1}\ !\ \Psi_j$, $j < k$, then Ψ_j is already inductive and $\Psi_j\ !\ \Phi$ does hold, because all previous Ψ_i have to satisfy Φ. This allows the assumption that j could be chosen as k since all further sets do not provide additional information. By doing so the formula could be reduced to:

$$\frac{\Psi_I\ !\ \Psi_0 \quad \bigwedge_{i=0}^{k\ 1}(\Psi_i\ !\ \Psi_{i+1} \wedge \ \Psi_{i+1}) \quad \Psi_k\ !\ \Phi \wedge \Psi_{k\ 1}}{\Psi_{reach}\ !\ \Phi} \tag{3.28}$$

Note that in formula 3.28 the second part enforces $(\Psi_{k\ 1}\ !\ \Psi_k)$ and the third part enforces $(\Psi_k\ !\ \Psi_{k\ 1})$ and thus $\Psi_{k\ 1}$ has to be inductive, which means that there is no transition from any state of $\Psi_{k\ 1}$ to any state which is not in $\Psi_{k\ 1}$. This formula could also be tested given the set of reachable states Ψ_{reach} is known. By definition Ψ_{reach} is already inductive and contains all the initial states and thus Ψ_k as well as Ψ_0 could be chosen as Ψ_{reach}. This results in:

$$\frac{\dfrac{\Psi_I\ !\ \Psi_{reach} \quad \Psi_{reach}\ !\ \Psi_{reach} \wedge \ \Psi_{reach} \quad \Psi_{reach}\ !\ \Phi \wedge \Psi_{reach}}{\Psi_I\ !\ \Psi_{reach} \qquad \Psi_{reach}\ !\ \Psi_{reach} \qquad \Psi_{reach}\ !\ \Phi}}{\Psi_{reach}\ !\ \Phi} \tag{3.29}$$

This resembles formula 3.14 discussed earlier and also illustrates further interesting properties of formula 3.15. While the formula is a correct and complete inference rule and describes the thought process of Bradley's PDR implementation, it is still essentially decoupled from PDR. Actually, PDR as discussed earlier describes only one possible implementation of this formula, by choosing all Ψ_i to be an overapproximation of the reachable states in i steps. However, the inference rule shown in formula 3.28 does not impose this restriction on the Ψ_i but is more general. This is a very important insight for the rest of the thesis as well as in understanding the capabilities and restrictions on PDR. Furthermore, this allows to *guess* Ψ_i and maybe there is another, so far unknown way to calculate a valid sequence of Ψ_i more efficiently. However,

this leads to the question if there are similar inference rules for other CTL properties as well, or if this approach is limited to safety-properties.

To this end consider the origin of the inference rule, which lies in Park's fixpoint induction rule (definition 3.1.1). If we express this original rule in the current notation, we get the following inference rule for least fixpoints:

$$\frac{[\varphi]_x^{\psi} \;\; ! \;\; \psi}{(\mu x.\varphi) \;\; ! \;\; \psi} \tag{3.30}$$

The notation $[\varphi]_x^{\psi}$ is a propositional logic substitution, thus each appearance of the variable x in formula φ is substituted by the expression ψ. This is, therefore, a notation to state that φ is some function which is applied to formula ψ. Note that φ is a formula used in a fixpoint calculation and thus has to be monotonic. While this inference rule for least fixpoints so far is of less importance, a second similar rule for greatest fixpoint could be derived from Park, despite not being stated originally:

$$\frac{\psi \;\; ! \;\; [\varphi]_x^{\psi}}{\psi \;\; ! \;\; (\nu x.\varphi)} \tag{3.31}$$

This formula is complete and correct since it is essentially a conclusion from pre-fixpoints and post-fixpoints, whereas $[\varphi]_x^{\psi} \;\; ! \;\; \psi$ denotes a pre-fixpoint and $\psi \;\; ! \;\; [\varphi]_x^{\psi}$ denotes a post-fixpoint. Furthermore, the least fixpoint is contained in any pre-fixpoint and the greatest fixpoint contains any post-fixpoint by definition. By this, both rules follow trivially. However, the second rule 3.31 provides an important foundation for verification purposes, as it states that given a certain set of states ψ, which is a post-fixpoint, then this state set will be a subset of the greatest fixpoint. Therefore it is sufficient to show that when applying a function φ to a state-set ψ, only states are added to ψ and any ψ which satisfies this rule is also inside the greatest fixpoint of φ.

So far we considered only greatest fixpoints, as for example, the safety property is one of those greatest fixpoints.[10] Furthermore it is easy to deduce an inference rule for safety properties by using modus ponens, since we only have to make sure that ψ also contains the initial states:

$$\frac{\Psi_I \;\; ! \;\; \psi \quad \psi \;\; ! \;\; [\varphi]_x^{\psi}}{\Psi_I \;\; ! \;\; (\nu x.\varphi)} \tag{3.32}$$

The proof of correctness for this rule is trivial, since formula 3.31 already states that from $\psi \;\; ! \;\; [\varphi]_x^{\psi}$ follows $\psi \;\; ! \;\; (\nu x.\varphi)$ and if in addition $\psi_I \;\; ! \;\; \psi$ holds, this leads to $\psi_I \;\; ! \;\; (\nu x.\varphi)$ by transitivity of the implication.

However, while it is easy to derive an inference rule for greatest fixpoints, this is not as easily done for the least fixpoint rule, as the direction of the implication in the final conclusion is inverted in Park's fixpoint induction rule. This is also the reason why there are no induction like rules for least fixpoint

[10] See for example in chapter 2.5 where the formula 2.23 is derived:$(\nu x.\varphi \;\wedge\; x)$. This makes it easy to see that the safety property AG and EG are both greatest fixpoints because the μ-calculus operator is ν, and the A and E are deferred by using either or respectively.

properties like *liveness*. However in [KS19] earlier this year, Köhler introduced an incremental fixpoint induction rule which is more related to the incremental fixpoint induction rule for greatest fixpoints, then it is to the least fixpoint rule of Park, because of the direction of the concluded implication:

$$\frac{\psi_0 \;!\; [\varphi]_x^{\mathsf{false}} \quad \bigwedge_{i=0}^{n\;1} (\psi_i \;!\; \psi_{i+1}) \quad \bigwedge_{i=0}^{n\;1} \left(\psi_{i+1} \;!\; [\varphi]_x^{\psi_i}\right) \quad \Psi_I \;!\; \psi_n}{\Psi_I \;!\; \mu x.\varphi} \tag{3.33}$$

or with inverted (decreasing) set sequence:

$$\frac{\Psi_I \;!\; \psi_0 \quad \bigwedge_{i=0}^{n\;1} (\psi_{i+1} \;!\; \psi_i) \quad \bigwedge_{i=0}^{n\;1} \left(\psi_i \;!\; [\varphi]_x^{\psi_{i+1}}\right) \quad \psi_n \;!\; [\varphi]_x^{\mathsf{false}}}{\Psi_I \;!\; \mu x.\varphi} \tag{3.34}$$

This formula is quite complicated at first glance, therefore consider this formula step by step. First of all, it is important to understand the substitution $[\varphi]_x^{\psi_i}$ as well as the consequences of this substitution. To this regard, it is helpful to imagine the least fixpoint of consideration would be for example a liveness property. In this case the substitution $[\varphi]_x^{\psi_i}$ and especially the *function* φ denotes $\alpha\_\, x\_ \quad x$ where x is substituted by ψ_i. Thus the function φ provides a set of states containing the set x, joined with all predecessors of x as well as all states which satisfy α. Note that states are only added to this set and no state is removed, thus a function φ like this would be monotonic and it has to be monotonic since it is part of a μ-calculus formula in the conclusion of this inference rule.

Given this $\varphi := \alpha\_\, x\_ \quad x$ and consider $\mu x.\varphi$. By global model checking, $x_0 = \mathsf{f}\alpha\mathsf{g}$, $x_1 = \mathsf{f}\alpha\_ \quad \alpha\mathsf{g}$, $x_2 = \mathsf{f}\alpha\_ \quad \alpha\_ \quad (\quad \alpha)\mathsf{g}$, …, thus:

$$x_n = \mathsf{f}sj \text{ All state } s \text{ which have a path to } \alpha \text{ of } \mathit{length} \quad n\mathsf{g} \tag{3.35}$$

If $x_n = x_{n+1}$ this means that x_n is inductive and we reached a fixpoint, which denotes that there is no state which is not in x_n and still has a finite path to an α-state. Therefore the least fixpoint of φ is the set of all states which have a finite path to an α-state and thus $(\mu x.\varphi) = (\mathsf{EF}\alpha)$.

If we substitute φ by $\alpha\_\, x\_ \quad x$ in formula 3.33 we get:

$$\frac{\psi_0 \;!\; \alpha \quad \bigwedge_{i=0}^{n\;1} (\psi_i \;!\; \psi_{i+1}) \quad \bigwedge_{i=0}^{n\;1} (\psi_{i+1} \;!\; \alpha\_\, \psi_i\_ \quad \psi_i) \quad \Psi_I \;!\; \psi_n}{\Psi_I \;!\; \mathsf{EF}\alpha} \tag{3.36}$$

In the third assertion $\bigwedge_{i=0}^{n\;1}(\psi_{i+1} \;!\; \alpha\_\, \psi_i\_ \quad \psi_i)$ there is a disjunction with ψ_i and combined with the second assertion this states that two subsequent

ψ_i could also be identical. While this disjunction is included, such that $\varphi := \alpha\ \_\ x\ \_$ x could intuitively be proofed to be monotone, the second identical set does not provide additional information.[11] Thus if there is is a sequence of ψ_i, 0 i n, which satisfies all assertions of formula 3.36 and there are some $\psi_x = \psi_{x+1}$, 0 x n 1, then there is also a similar sequence of ψ_j, 0 j $m < n$ 1, where only those ψ_i remain which are pairwise different and this will also satisfy all four assertions of formula 3.36. Therefore this rule is still complete and correct if this disjunction in the third assertion is omitted, which results in:[12]

$$\frac{\psi_0\ \ !\ \ \alpha\ \ \bigwedge_{i=0}^{n\ \ 1}(\psi_i\ \ !\ \ \psi_{i+1})\ \ \bigwedge_{i=0}^{n\ \ 1}(\psi_{i+1}\ \ !\ \ \alpha\ \_\ \ \psi_i)\ \ \Psi_I\ \ !\ \ \psi_n}{\Psi_I\ \ !\ \ \mathsf{EF}\alpha} \tag{3.37}$$

In a similar fashion it is possible to derive a formula for safety properties from formula 3.32, which resembles the already discussed formula 3.14:

$$\frac{\Psi_I\ \ !\ \ \psi\ \ \ \psi\ \ !\ \ \alpha\ \wedge\ \ \psi}{\Psi_I\ \ !\ \ \mathsf{AG}\alpha} \tag{3.38}$$

while the resemblance to formula 3.14 is more obvious if the second assertion is split up:

$$\frac{\Psi_I\ \ !\ \ \psi\ \ \ \psi\ \ !\ \ \ \ \psi\ \ \ \psi\ \ !\ \ \alpha}{\Psi_I\ \ !\ \ \mathsf{AG}\alpha} \tag{3.39}$$

There are some key conceptual differences between those two formulas 3.37 and 3.39, as well as some remarkable similarities, which should be discussed. The most obvious difference is the increased complexity of 3.37 and why this is necessary.

As discussed in chapter 2.5 a safety property could be stated as $\nu x.\alpha \wedge\ \ x$ whereas a liveness property would be $\mu x.\alpha\ \_\ \ x$. This coincidence was also considered in [KS19] and imposes the question if

$$\frac{\Psi_I\ \ !\ \ \psi\ \ \ \psi\ \ !\ \ \alpha\ \_\ \ \psi}{\Psi_I\ \ !\ ^{??}\ \mathsf{EF}\alpha} \tag{3.40}$$

would also be sufficient, since only the safety property in 3.38 is substituted by a similar liveness property. In the case of the safety property the inductiveness of ψ was used to show $\mathsf{AG}\alpha$, because inductiveness states that whatever transition is chosen from a state in ψ, it is not possible to reach a

[11] A formula $\varphi := \alpha\ \_\ \ x$ is also monotone, since the relation used is the subset-relation discussed in detail in the preliminaries section. Proof of monotonicity ($x \vee y\ \ !\ \ f(x) \vee f(y)$): Given any state-set x and a set of predecessor states x, when adding a state s to x, also its predecessors s have to be added to x, if they are not already in x. However by adding states to x there is never a state removed from s and thus $\varphi := \alpha\ \_\ \ x$ is monotone.

[12] This formula is also the result of [KS19], while the derivation and the proof are from myself and differ from the previously published proof in [KS19].

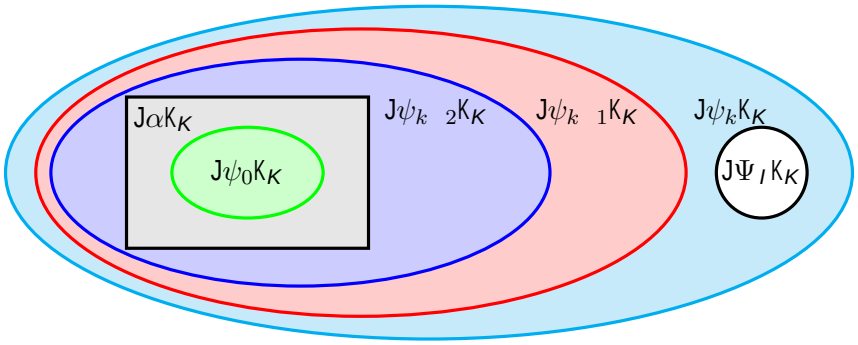

Figure 3.8.: *A visualization of the sequence of state-sets of formula 3.37. From any ψ_i to ψ_{i+1} only predecessors of ψ_i as well as ψ_i-states are added, until all initial states are included in ψ_k.*

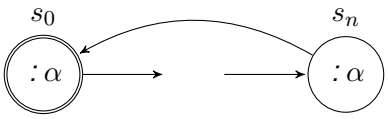

Figure 3.9.: *The counterexample to formula 3.40 given in [KS19]. All states satisfy $:\alpha$ and the only path is a cycle from s_0 to s_n and back to s_0. Since an α-state is never reached, this structure does not satisfy $\mathsf{EF}\alpha$.*

$:\psi$-state. Given the structure shown in figure 3.9 for example, the state-set $\psi = \{s_0, \ldots, s_n\}$ is inductive, since there are only transitions to other states inside this set and no transition will leave this set. However, it is obvious that it is not possible to reach an α-state either and thus on the only path $\mathsf{G}\alpha$ holds. Therefore this structure does not satisfy $\mathsf{EF}\alpha$ since not all initial states (in this special case only s_0) are elements of $\mathsf{JEF}\alpha\mathsf{K}_K$. Despite that ψ does satisfy both assertions of formula 3.40 and thus those assertions are not sufficient to infer $\mathsf{EF}\alpha$, but a weaker conclusion $\mathsf{EF}\alpha \_ \mathsf{EG}\psi$ as proven in [KS19], which leads to the following two inference rules:

$$\frac{\Psi_l \;!\; \psi \quad \psi \;!\; \alpha \_ \quad \psi}{\Psi_l \;!\; \mathsf{EF}\alpha \_ \mathsf{EG}\psi} \qquad \frac{\Psi_l \;!\; \psi \quad \psi \;!\; \alpha \_ \quad \psi}{\Psi_l \;!\; \mathsf{A}(\mathsf{F}\alpha \_ \mathsf{G}\psi)} \tag{3.41}$$

This shows that inductiveness alone is not sufficient, because it does not necessarily lead to the finiteness of a path, which is an essential part of $\mathsf{EF}\alpha$. To enforce finiteness, some kind of "progress measure, i.e., a Noetherian ordering ξ" ([KS19]) is necessary, which is also used in termination proofs. This ordering ξ is defined in a way "that whenever a ψ-state s does not satisfy $[\alpha]$, then $\xi(s)$ w $\xi(s')$ for at least one or all (depending on the rule) successor state s' of s." ([KS19]) This means that ξ could be interpreted as the distance of an s-state to the closest α-state, thus if s does not satisfy α, its successor needs to be closer to an α-state or is an α-state itself. This is implemented by choosing a sequence of ψ_i, $0 \quad i \quad n \; 2 \; \mathsf{N}$, which implicitly have some kind of order or distance measure regarding the α-states. If such a distance measure is added to formula 3.40, this results in another inference rule for liveness properties

$$\frac{\Psi_{\mathcal{I}} \to \psi \quad \psi \to \alpha \wedge \Diamond\psi}{\Psi_{\mathcal{I}} \to \mathsf{EG}\alpha} \qquad \frac{\Psi_{\mathcal{I}} \to \psi_0 \quad \bigwedge_{i=0}^{n-1}(\psi_i \to \psi_{i+1} \wedge \Diamond\psi_{i+1}) \quad \psi_n \to \psi_{n-1} \wedge \alpha}{\Psi_{\mathcal{I}} \to \mathsf{EG}\alpha}$$

$$\frac{\psi_0 \to \beta \quad \bigwedge_{i=0}^{n-1}(\psi_i \to \psi_{i+1}) \quad \bigwedge_{i=0}^{n-1}(\psi_{i+1} \to \beta \vee \Diamond\psi_i) \quad \Psi_{\mathcal{I}} \to \psi_n}{\Psi_{\mathcal{I}} \to \mathsf{EF}\beta} \qquad \frac{\Psi_{\mathcal{I}} \to \psi_0 \quad \bigwedge_{i=0}^{n-1}(\psi_i \to \beta \vee \Diamond\psi_{i+1}) \quad \psi_n \to \beta}{\Psi_{\mathcal{I}} \to \mathsf{EF}\beta}$$

$$\frac{\Psi_{\mathcal{I}} \to \psi \quad \psi \to \beta \vee \alpha \wedge \Diamond\psi}{\Psi_{\mathcal{I}} \to \mathsf{E}[\alpha\ \mathsf{U}\ \beta]} \qquad \frac{\psi_0 \to \beta \quad \bigwedge_{i=0}^{n-1}(\psi_i \to \psi_{i+1}) \quad \bigwedge_{i=0}^{n-1}(\psi_{i+1} \to \beta \vee \alpha \wedge \Diamond\psi_i) \quad \Psi_{\mathcal{I}} \to \psi_n}{\Psi_{\mathcal{I}} \to \mathsf{E}[\alpha\ \underline{\mathsf{U}}\ \beta]}$$

$$\frac{\Psi_{\mathcal{I}} \to \psi \quad \psi \to \neg\beta \wedge (\alpha \vee \Diamond\psi)}{\Psi_{\mathcal{I}} \to \mathsf{E}[\alpha\ \mathsf{B}\ \beta]} \qquad \frac{\psi_0 \to \alpha \wedge \neg\beta \quad \bigwedge_{i=0}^{n-1}(\psi_i \to \psi_{i+1}) \quad \bigwedge_{i=0}^{n-1}(\psi_{i+1} \to \neg\beta \wedge (\alpha \vee \Diamond\psi_i)) \quad \Psi_{\mathcal{I}} \to \psi_n}{\Psi_{\mathcal{I}} \to \mathsf{E}[\alpha\ \underline{\mathsf{B}}\ \beta]}$$

Figure 3.10.: *Induction Rules for Temporal Logic Formulas: In addition to the rules shown above, the same rules are correct and complete when* E *and are replaced with* A *and , respectively. Source: [KS19]*

(see [KS19]):

$$\frac{\Psi_{\mathit{I}}\ \mathit{!}\ \psi_0 \quad \bigwedge_{i=0}^{n\ \ 1}(\psi_i\ \mathit{!}\ \alpha\_\quad \psi_{i+1}) \quad \psi_n\ \mathit{!}\ \alpha}{\Psi_{\mathit{I}}\ \mathit{!}\ \mathsf{AF}\alpha} \qquad (3.42)$$

$$\frac{\Psi_{\mathit{I}}\ \mathit{!}\ \psi_0 \quad \bigwedge_{i=0}^{n\ \ 1}(\psi_i\ \mathit{!}\ \alpha\_\quad \psi_{i+1}) \quad \psi_n\ \mathit{!}\ \alpha}{\Psi_{\mathit{I}}\ \mathit{!}\ \mathsf{EF}\alpha}$$

Note that this rule 3.42 does not use inductiveness of an individual ψ_i, rather than showing close resemblance to formula 3.28 and the use of post-fixpoints. This resemblance is remarkable since rule 3.42 is used to prove a property of a least fixpoint, however, in the case of the safety properties, inductiveness is necessary, while in the case of liveness it is not.

In figure 3.10 correct and complete induction rules for further operators are given. Note that the first four have been discussed in great detail, while the last four are easily constructed by using formula 3.32 and formula 3.33 and choosing φ correspondingly. [13] E.g. when considering the least fixpoint of $\varphi := \beta\_\alpha\ \mathsf{\wedge}\quad x$, then this will result in $x_0 = \mathsf{f}\mathsf{g}$, $x_1 = \mathsf{f}\beta\mathsf{g}$, $x_2 = \mathsf{f}\beta\_\alpha\ \mathsf{\wedge}\quad \beta\mathsf{g}$, ...[14]. Thus for any n this results in a state-set x_n which contains all the states, which are either a β-state or a state which has a path of length n, where on each state on this path α holds. If $x_n = x_{n+1}$ and thus φ reaches a least fixpoint $\mu x.\varphi$, this means that x_n contains all states where β is satisfied or which have a path to a β-state on which in each state α holds. Furthermore since it is a fixpoint, there are no further $:x_n$-states, which have an α-path to a β-state and thus $\mu x.\varphi = \mathsf{E}[\beta\ \underline{\mathsf{U}}\ \alpha]$. Therefore defining $\varphi = \beta\_\alpha\ \mathsf{\wedge}\quad x$ in

[13] Furthermore E and A only differ in the predecessor operator and respectively.

[14] Provided that there are no deadend states (states without any outgoing transition). However we assume that any Kripke structure of relevance has no deadend states.

formula 3.33 leads to the sixth formula shown in figure 3.10. All other rules are constructed analogous.

In conclusion, new inference proof rules have been presented in the publication [KS19]. Those proof rules abstract from the method of PDR and are applicable to not only all CTL properties, but all least and greatest fixpoints. This makes those rules a powerful tool. Since they allow an abstraction from PDR, this allows to study the structure and properties of the proofs themselves. Thus those new proof rules formed the starting point of this thesis and the new insight gained in the scope of this thesis is strongly correlated to those rules.

3.4. An Incremental Approach to Model Checking Progress Properties

After publishing IC3, which has been discussed in chapter 3.2 in great detail and which was an early implementation of the PDR method, Bradley published another paper in 2011 which was entitled *An Incremental Approach to Model Checking Progress Properties* [Bra+11]. In this publication, Bradley introduced a method for verifying a *Büchi fairness condition*, by utilizing and manipulating his previously published implementation of PDR.

First of all a some definitions[15] are necessary, most of which are explained in greater detail in the chapter 2 of this thesis. As a foundation Bradley used a *finite-state system*, defined as a tuple $S : (\bar{i}, \bar{x}, I(\bar{x}), T(\bar{i}, \bar{x}, \bar{x}'))$, where $\bar{i}$ denotes an array of individual input variables i, $\bar{x}$ denotes an array of state variables, $I(\bar{x})$ is a propositional formula describing the initial states and $T(\bar{i}, \bar{x}, \bar{x}')$ is a propositional formula describing the transition relation. A primed variable denotes a variable in the successor state.

Note that such a structure S could be translated to a Kripke structure as used previously in this thesis, whereas the corresponding Kripke structure has more states than S but no input variables $\bar{i}$. Thus a Kripke structure could be used at this point, but in this case, input and state variables can not be distinguished anymore. While this may seem irrelevant at first glance, it should be noted that the input variables could be ignored[16] most often. By doing so, this will reduce the size of SAT-queries to a potentially significant amount.

A *state* s of S is an assignment of Boolean values to all variables $\bar{x}$ and is described by a *cube*, a conjunction of *literals*, over all variables $\bar{x}$. If s satisfies some formula F, $s \models F$, then s is an F-state and a formula F implies another formula G, $F \rightarrow G$, if every satisfying assignment of F satisfies G, thus the set of F-states is a subset of G-states. A *run* of S, $(s_0, s_1, s_2, \dots)$, is a sequence of states, such that (if not stated otherwise) $s_0 \models I$, thus s_0 is an initial state and for each (s_i, s_{i+1}) in this sequence, $\exists \bar{i}.(\bar{i}, s_i, s'_{i+1}) \models T$. Therefore in a run there is an input for the transition formula to get from s_i to s_{i+1} in one step. Each state which is on some run of the system is *reachable*. An *path*, which is an infinite run, $(s_0, s_1, s_2, \dots)s$, is a *computation* of S if infinitely many s_i satisfy B, $s_i \models B$. B is called a *Büchi fairness condition*. This path constraint could be expressed by the CTL formula $\mathsf{FG}B$. An assertion A is *inductive relative to* another assertion B, iff $B(\bar{x}, \bar{x}') \wedge A(\bar{x}) \wedge T(\bar{i}, \bar{x}, \bar{x}') \rightarrow A(\bar{x}')$, thus if B is a restriction which constraints states, as well as next states.

As in PDR, inductiveness is a quite interesting property, since it states that a certain state set is closed or could not be left under the transition relation, furthermore, it is well suited for incremental refinement and thus "storing information" ([Bra+11]). This fact was used for PDR but Bradley showed

[15] Defined in [Bra+11]

[16] Input variables are often ignored for verification purposes, since it is not relevant why a transition it taken. Usually, the verification goal is that all transitions as a whole satisfy a desired property.

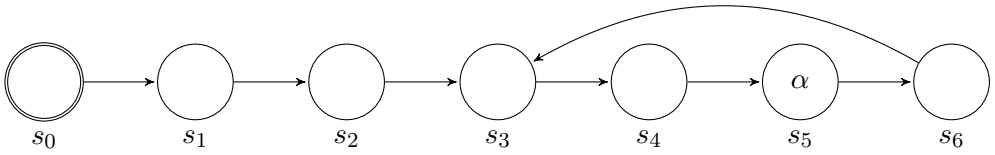

Figure 3.11.: *A structure which consists of one path, which satisfies* $\mathsf{GF}\alpha$ *since from any state on this path it is possible to reach a state which satisfies* α*. Note that the path is a sequence of* s_i *and has to be infinite, thus there are infinitely many* s_i *wich satisfy* α *on this path.*

that this is not limited to *safety properties* $\mathsf{G}\alpha$, but could also be used when proving *fairness properties* $\mathsf{GF}\alpha$.

To this end consider a structure which satisfies some fairness constraints. Since this structure only consists of finitely many states, but the path needs to be of infinite length, there has to be a cycle somewhere in this structure, since otherwise, the longest possible path would visit each state exactly once and hence would be finite. Furthermore, $\mathsf{GF}\alpha$ states that an α-state has to be visited infinitely often, while α could be a set of states and is not limited to only one state. Thus there have to be some (at least one) α-states on this path. However the initial state does not have to be on this cycle, but there has to be a finite run to such a cycle. This cycle is then called a *reachable fair cycle*.

This substructure is often referred to as a *lasso*, consisting of a finite run called *stem* from an initial state to some intermediate state s_i and the corresponding *loop*, which is a finite run from this s_i back to itself. An example is shown in figure 3.11, where the stem is the run (s_0, s_1, s_2, s_3) and the loop is the run $(s_3, s_4, s_5, s_6, s_3)$. However it is also valid to choose as a stem $(s_0, s_1, s_2, s_3, s_4, s_5)$ and as the loop $(s_5, s_6, s_3, s_4, s_5)$, since it is possible to form a stem to any state on the cycle, and by the fact that it is a cycle, the "start" and "end" points are arbitrary.

This idea forms the foundation of [Bra+11] since Bradley proposes a method to incrementally refine information regarding the stem and the loop until it is either proofed that there does not exist such a lasso or a *skeleton* is found. A *skeleton* is a set of states which satisfies all given fairness constraints. Given for example $\mathsf{GF}\alpha$ it consists of one arbitrary α-state, or given two constraints $\mathsf{GF}\alpha \wedge \mathsf{GF}\beta$ it consists of a single arbitrary α as well as a single arbitrary β-state. Therefore the task is to determine for each skeleton if it is on a fair cycle and if this cycle, in turn, is reachable from an initial state. To get a skeleton, Bradley introduced the *skeleton query*:

$$SAT? \bigwedge_{B2B} \left[\begin{array}{l} B(\overline{x}_{j(B)}) \wedge \bigwedge\limits_{R2R} R(\overline{x}_{j(B)}) \\ \wedge \bigwedge\limits_{W2W} (c_W \;!\; W(\overline{x}_{j(B)})) \wedge (\neg c_W \;!\; \neg W(\overline{x}_{j(B)})) \end{array} \right] \tag{3.43}$$

This query is checked each iteration while getting incrementally refined and

given it is unsatisfiable the nonexistence of a fair cycle in the given structure is proved, while each witness[17] of this formula is a skeleton. This skeleton is then proved to be on a fair cycle and reachable, or new information is gained, which further constrains the skeleton query in the next iteration. Note that there is a conjunction over all fairness constraints B and for each $B \mathbin{2} \mathsf{B}$ there could be another state in the skeleton, thus the size of the skeleton is less than or equal to jBj. However, a witness of a SAT-solver is just one assignment to all variables and thus corresponds only to one single state. The skeleton, on the other hand, could consist of up to n states, therefore some projection has to be made and there need to be up to n copies of all state variables. Note that it is important that only the state variables and not the input variables need to be duplicated and thus it makes a significant difference if the corresponding Kripke structure is chosen or the finite-state machine as stated initially. Furthermore choosing the Kripke structure could increase the number of states which makes the SAT-query itself more complex. This projection of variables is done by introducing the index j which depends on the chosen fairness condition B. Since multiple fairness constraints could be solved by the same state and to achieve a potentially smaller skeleton, $j(B)$ could map multiple fairness conditions to the same index. The notation $B(\overline{x}_{j(B)})$, therefore, denotes that all state variables in the formula B have the index $j(B)$ and analogous for similar formulas.

The formula 3.43 itself consists of three parts, while the first one is the fairness constraint B, the second part consists of "reachability information" R and the third part defines "walls" W in the structure which no fair cycle can pass. Note that the transition relation itself does not have to be considered when checking satisfiability of this formula.

First, we consider the fairness constraint B itself. Each fairness condition $\mathsf{GF}B$ describes a set of B-states which has to be visited infinitely often. The following query would yield a basic skeleton if it is satisfiable, namely a satisfying witness.

$$SAT? \bigwedge_{B \mathsf{2} \mathsf{B}} \left[B(\overline{x}_{j(B)})\right] \tag{3.44}$$

Thus this skeleton is a set of states, where each state satisfies at least one B and the skeleton as a whole contains for each given fairness condition $B_{j(B)}$ a state $s_{j(B)}$ which satisfies $B_{j(B)}$, $s_{j(B)} \models B_{j(B)}$. However, it is not yet known if these skeleton states are reachable and if they are on a cycle. To prove this more information are necessary, namely the reachability information (derived from considerations of the stem), as well as walls which split cycles (derived from considerations of the loop).

Next, we consider the reachability information. Given that there is a cycle in the provided structure, which connects all skeleton states to a loop, then it remains to prove that this cycle is reachable from an initial state, thus if there is a stem to any state on this cycle. To check this, Bradley introduced

[17] The witness is a (random) satisfying assignment provided by a SAT-solver to prove satisfiability of a formula.

the *stem query*:

$$reach\left(S, \bigwedge_{R2R} R(\overline{x}), I, s_0\right) \tag{3.45}$$

To this end, we first have to consider the origin of the $reach(S, C, F, G)$ function: Given for example any PDR implementation, this will check whether a certain condition α holds on all reachable states. If this is the case it will return an inductive proof P, thus a set of states which satisfies the following properties: (1) P contains all initial states, $I \;!\; P$, (2) all P states satisfy α, $P \;!\; \alpha$, and (3) P is inductive relative to some constraint C, $C \wedge P \wedge T \;!\; P'$.[18] Since this P is an inductive proof it is also a superset of the reachable states, and not necessarily identical to the reachable states. However, it is correct to restrict the state set of the given structure S to the states of P, since all $\lnot P$-states of S are not reachable anyway. Thus this proof P provides crucial reachability information, given that α is a valid safety property. If α is not a safety property, then PDR will result in a sequence of state sets, which implement some kind of ranking and prove that there is at least one $\lnot\alpha$-state which is reachable in finitely many steps.[19]

Therefore PDR could also be used to prove reachability of a state s since it only has to be proved that all reachable states are $\lnot s$-states. If all reachable states are indeed $\lnot s$-states and thus s is unreachable, then PDR will return a proof P which restricts the set of states to a superset of the reachable states, which is inductive, contains the initial states and excludes the unreachable state s, as well as potentially many more unreachable states by generalisation of counterexamples.[20] However if s is reachable, then PDR will return a finite run to s, thus proving reachability.

This is encapsulated by calling the $reach(S, C, F, G)$ function, which is essentially a PDR interface. This function proves if in a given structure S under a certain set of constraints C (which could restrict the transition relation as well as the set of states of consideration) there is a state in F from which there is a finite run to at least one state in G. This function is also used for evaluating the stem-query and verifies the reachability of one (arbitrary) s_0-state of the skeleton. Therefore it either provides a finite run from an initial state to the s_0-state or it provides a reachability proof R which is an inductive state set including the initial states and more crucially excluding the checked s_0 state. This reachability proof is used in following iterations of the algorithm to reduce the state set and thus restrict the search. Furthermore, since s_0 is excluded, it will not appear in any other skeleton in the future, thus restricting the set of Büchi fairness states B in following iterations.

[18] Note that T is the transition relation of the system S. Furthermore, the constraint is not necessary and with $C := \mathsf{true}$ can be ignored. However, the constraint might increase the efficiency since it reduces the search space and is will be used later.

[19] A precise description of this sequence of sets is found in section 3.2 of this thesis. At this point it is only relevant that a finite run to a $\lnot\alpha$-state could be derived from this sequence.

[20] More information regarding the generalization of counterexamples could be found in section 3.2 of this thesis.

Next, we consider why it is sufficient to check the reachability of this s_0-state and not the reachability of all skeleton states on the possible cycle. In order to be a fair cycle, all of the skeleton states have to lie on the loop in order to be "visited" infinitely often on an infinite path. Therefore, given that we already know that all of them lie on a loop, checking the reachability of a single skeleton state would be sufficient. If they form a lasso the choice of s_i for checking reachability only varies the length of the stem. However, one could also impose n queries for each s_i-state and run them in parallel, depending on the available resources and thus potentially gain information on multiple skeleton states at once, where each unsatisfiable query results in new reachability information R. Since all n queries are started with the same information at the point of initiation, the reachability information gained by the other queries are not present and thus each individual query is less restricted compared to sequential execution of the same queries. This is an interesting trade-off and each choice could potentially increase the performance, however, Bradley chose to run only one stem query in [Bra+11].

Furthermore, the reachability of a disjunction of all skeleton states $\bigvee_{i=0}^{n\ 1} s_i$ could be checked for reachability. Given the query is satisfiable (thus a skeleton state could be reached), then it would be potentially more complicated compared to just checking reachability of a single state while providing only a reduced information regarding individual states. However, given that it is unsatisfiable the generated proof would exclude multiple states and thus be more powerful. Again this is a trade-off, while the actual performance depends on properties of the chosen structure.

From this we can conclude that the stem query, which is essentially a macro for a PDR call, provides a way to decide if a state of the skeleton, and thus a state of a potential loop, is reachable and if this is not the case the query provides information in which direction the given structure should be considered further.

So far we discussed the skeleton query 3.43, as well as the stem query 3.45, which resulted in a skeleton (a set of states which contains a state to satisfy each proposed fairness property), as well as a proof that at least one of the skeleton states is reachable, as well as a restriction of the potential skeleton states. This corresponds to the upper part of formula 3.43. However, it remains to show that the skeleton states do actually connect to a loop, to which end Bradley introduced the *naive cycle query*:

$$reach\,(S, true, s_i, s_{i\ \ n1}) \tag{3.46}$$

This is essentially the same as the stem query, except that reachability is not checked from a set of initial states, but from a skeleton state to another skeleton state. Note that the expression $i\ \ _n\ 1$ states that i is incremented by one modulo n. Thus executing this query n times for each i, $0 \quad i \quad n \quad 1$, will check if every skeleton state s_i lies on a loop connecting all skeleton states, which starts and ends in state s_0. The order of the s_i is in this case not relevant, it only matters that this order is not changed while executing the cycle query. Again there are two potential results:

Given the query is satisfied, the `reach` function will provide a run from each skeleton state to the next skeleton state and therefore proves that the skeleton states all lie on a cycle. If the s_0 state is also reachable, this is a proof that there is a lasso in the structure. The case in which the cycle query is not satisfiable is more complicated. Given that a cycle query is not satisfied and there is no path from any s_i to the next $s_{i\ n1}$, then `reach` will again return an inductive proof W. We call a proof W generated by a cycle query a ***wall***.

Consider an arbitrary structure S where some transitions connect various states of S. A region of this graph is called a ***strongly connected component*** (SCC) if any state in this region can reach any other state in the same region, while each connecting run only visits states from this region. Furthermore, a region is SCC-***closed***, if every SCC is either entirely contained or entirely disjoint from this region. Furthermore, SCC-closed regions could be represented by a sequence of inductive assertions, where any inductive assertion corresponds to an SCC of the structure [Bra+11]. This is because they essentially form a "reachability bubble" around certain states, where everything inside this bubble could be reached and everything outside this bubble could not be reached from this state. The hull of this bubble is called a ***one-way wall*** since it can only be crossed in one direction (from outside to inside). This knowledge is now used to find a useful application of the inductive proofs W generated by the cycle query.

Given that the cycle query fails for some s_i, this means that there is no finite run connecting s_i to the next $s_{i\ n1}$. The inductive proof W generated by the `reach` function is an overapproximation of the states which can be reached from s_i and this set of states does not contain $s_{i\ n1}$, as well as excludes potentially many more states. Therefore s_i is a W-state and $s_{i\ n1}$ is a $:W$-state. Furthermore since the set of W-states can not be left, since it is inductive and thus closed under the transition relation, a fair cycle could either contain s_i as well as other W states, or $s_{i\ n1}$ and some additional $:W$-states and thus has to lie on one side of this wall W, since in order for a run to be a cycle and for this run to have a state on each side of W, it would have to pass the wall twice. But this is not possible since W is a one-way wall at least. Furthermore, the regions which are defined by these walls are called ***arenas***. Thus we try to find a potential skeleton which is on one side of a wall.

To this end consider again the skeleton query 3.43, and especially the lower part of this formula:

$$\bigwedge_{W 2 W} (c_W \ ! \ W(\overline{x}_{j(B)})) \wedge (:c_W \ ! \ :W(\overline{x}_{j(B)})) \tag{3.47}$$

This part is a restriction to the set of states from which a potential skeleton is chosen. A new set of ***choice variables*** c_W are introduced, which are independent of the set of the state variables. Each choice variable c_W states that all skeleton states are W-states of the corresponding wall W, if c_W is chosen to be true or are $:W$-states if c_W is set to false, while the value of those choice variables could be chosen by the SAT-solver.

It now remains to use this gained wall information in the cycle queries. This

can not be done trivially since it is not known on which side of the wall the skeleton lies and thus could not be as easily added as the reachability information gathered earlier by the stem query. At this point a more sophisticated approach is necessary:

Instead of letting the SAT-solver choose the value of the choice variable for the new wall W like in the regular skeleton query shown in formula 3.43, the skeleton query is executed twice, once with the corresponding c_W set to true and once set to false. This way it could be determined if there even is a potential skeleton on either side of W. Note that at least one of those queries has to be done anyway implicitly by the SAT call of the skeleton query. Executing this query with a known c_W provides crucial information since if there is no skeleton on one side of the wall, this wall could be used to restrict the cycle queries. This is done by adding a constraint C to the *constraint list* C.

The knowledge so far collected consists of the fact that there is an s_i from which a next $s_{i\ n1}$ can not be reached and there is a corresponding proof W which is an inductive set of the reachable states from s_i, which does not contain $s_{i\ n1}$. If there is no possible skeleton found in W, then the skeleton has to be in $:W$, thus consists of $:W$-states. But there is a further known restriction: Since W is inductive and we try to find a cycle, all states of a potential cycle have to be $:W$ states. If one state on this cycle would be a W-state, then since W is inductive there is no way to reach a $:W$ state from this state. This insight provides a way to restrict the transition relation in the cycle query by choosing the corresponding constraints C on the cycle query. If there is no W skeleton, the constraint C is $:W \wedge :W'$, thus only the transitions from $:W$-state to other $:W$ remain relevant. Vice versa if there is no $:W$ skeleton, then the constraint is $W \wedge W'$ since potential cycles have to consist of W states. If there is however a skeleton provided by the executed skeleton queries on each side of W, we can not restrict the transition relation as much, however, the knowledge remains. Thus the cycle has to be on one side of the wall, and thus the transition relation is restricted by adding $W \$ W'$. This constraint list is used in the cycle query to reduce the search space:

$$reach\left(S, \bigwedge_{R \in R} R(\overline{x}) \wedge \bigwedge_{C \in C} C(\overline{x}), s_i, s_{i\ n1}\right) \tag{3.48}$$

However there is one problem left: If the skeleton consists of only one state, then the cycle query gets trivial, since a run which starts and stops in s_0 and only has to visit s_0 does not have to leave the s_0 state, since (s_0) would be a satisfying run. Thus it has to be proven that there is a nontrivial run from s_0 back to itself, which uses at least one transition. This is done by verifying that a successor state of s_0 can reach s_0. If this is the case, then either s_0 has a transition to itself, or there is a run starting in s_0 over some other states, which

ends again in s_0. This is checked by the *single-state skeleton cycle query*[21]:

$$reach\left(S, \bigwedge_{R2R} R(\overline{x}) \wedge \bigwedge_{C2C} C(\overline{x}), \quad s_0, s_0\right) \tag{3.49}$$

Given this query is satisfied, there is a run from a successor state of s_0 back to s_0, which could even be of length zero. Thus even looping from s_0 back to itself would satisfy this query. However, if there is no such run and the query results in a proof P, this needs to be handled explicitly.

First, consider why this is the case. If query 3.49 is not satisfied, this means that there is no successor state of s which has a run to s and thus s does not lie on a cycle. Therefore s needs to be excluded as potential skeleton state in further iterations. However, this is not easily done, resulting from the way the constraining sets W, R and C are constructed. The single-state skeleton query will return a proof P which is an inductive set, which separates s from its successors. This is because P has to globally satisfy $\neg s$ since s is not reachable. Therefore s is a $\neg P$-state. However by adding P as a wall, in the next iteration, a SAT-solver could choose a skeleton in $\neg P$ and thus again the same state s. In this situation, this algorithm would not terminate and therefore this case needs to be handled explicitly.

Bradley proposed a way to handle this: instead of choosing the wall to be $W = P$ and $\neg W = \neg P$, the walls should be set to $W = P$ and $\neg W = \neg P \wedge \neg s$. This is because P already excludes s, however, this needs to be specified explicitly in the case of $\neg P$. This way, in either case, it is not possible to choose s as a skeleton state, no matter which side of the wall is chosen. Consider why this is also possible: Given the information that s does not lie on any cycle, even when s is reachable it will never be on a fair cycle itself. Thus s could be excluded from any potential arena since whenever s is chosen it is not possible to form a loop anymore and could, therefore, be excluded from further consideration. However, this introduces the need to save the list of walls W as well as their negations $\neg W$ explicitly, since it is not sufficient anymore to simply negate W.

This algorithm is called *fair* and was published in [Bra+11], as well as the pseudo-code shown in figure 3.12. Note that in the beginning a bijection ι is introduced in line 5, which is then weakened in a way, that multiple fairness constraints could map to a single index j. Line 12 provides the outer loop, stating that there has to be at least one potential skeleton, otherwise the algorithm terminates and returns false. This skeleton consisting of n states is then tested for reachability by `stemQ()`, which resembles formula 3.45. Furthermore if $n = 1$ the function `singleCycleQ()` (formula 3.49) is called to check whether the single skeleton state s lies on a cycle and otherwise `cycleQ()` (formula 3.48) is executed to check whether all skeleton states lie on a cycle[22].

[21] Note that s_0 does not have to be computed explicitly, rather it is provided by the transition relation as well as a constraint and is thus is "only" a formula. It is therefore not necessary to run this query for each state in s_0 individually.

[22] Note that verifying if multiple states are on a cycle is the same as checking whether they are in the same SCC region.

```
1  bool fair(S:system):
2      // reachability information R, walls W
3      // transition constraints C
4      R := ;, W := ;, C := ;
5      ι := bijection between B and {1,...,|B|}

7      // ι is a bijection, while for minimization purposes of the
8      // skeleton multiple B can map to one j
9      while skelQ(R, W, ι) is sat:
10         j :=map(R, W)

12         while skelQ(R, W, j) is sat:
13             result:=
14                 s_0, ..., s_{n-1} :=skelQ(R, W, j)
15                 in parallel do until (all are reachable) or
16                 (one returns inductive proof P as counterexample):
17                     stemQ(R, s_0)
18                     if n = 1:
19                         singleCycleQ(R, C, s_0, P)
20                     elif for i 2 f0, ..., m  1g:
21                         cycleQ(R, C, s_i, s_{i+1})

23             if result is (all reachable):
24                 // there is a lasso in the structure
25                 return true

27             elif result is (a proof P from stemQ):
28                 // new reachability information
29                 R := R [ fPg

31             elif result is (a proof P from cycleQ):
32                 // P is a wall: P, :P are SCC-closed
33                 if n = 1:
34                     // if the skeleton consists of only one state,
35                     // this state needs to be excluded from P and :P
36                     :P := :P ^ :s
37                 W := W [ fPg
38                 if skelQ(R, W, ι) ^c_W is unsat:
39                     C := :P ^ :P'
40                 elif skelQ(R, W, ι) ^:c_W is unsat:
41                     C := P ^ P'
42                 else:
43                     C := P $ P'
44                 C := C [ fCg

46     return false
```

Figure 3.12.: *A simplified version of Bradley's implementation of the query calls presented in this chapter, published in [Bra+11]. Bradley called this algorithm* `fair` *and it proves that there exists a fair cycle in a given structure S. The simplifications done are renaming of a few variables to get in line with the formulas, as well as remove heuristics which may improve performance.*

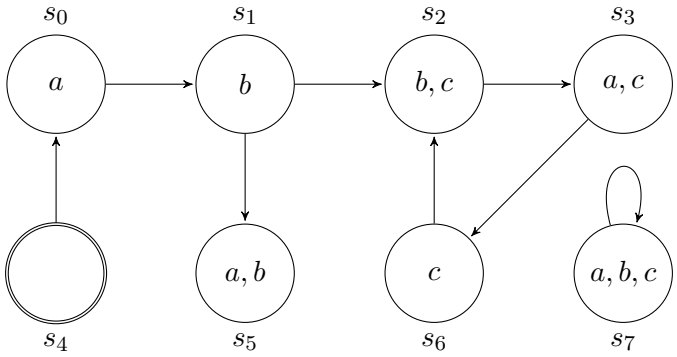

Figure 3.13.: *An example structure to demonstrate the* `fair` *algorithm. Note that there is a cycle on which infinitely often a as well as b holds, which is reachable.*

The stem and cycle queries could be run in parallel if the hardware resources are sufficient to do so. Given that all stem and cycle queries are successful, there is a loop in the given structure, however, if this is not the case a proof P is returned. Depending on the source of P it is added to the corresponding constraint list C, W and/or R, which reduces the set of potential skeleton states, by removing the skeleton states which either harm the stem or the cycle query from further iterations. A more detailed explanation of the algorithm, as well as the proof of termination and correctness, could be found in [Bra+11].

As example consider the structure given in figure 3.13. The two provided Büchi fairness constraints are $B_1 := a$ and $B_2 := b$ and thus the goal is to find a fair cycle, which is reachable and on which there is at least one a-state as well as one b-state.

Since at the beginning of the execution of the `fair` algorithm no constraints are given, a random set of states $skeleton_0$ is selected in a way that

$$\mathsf{8}B_i.(\mathsf{9}s_j \ \mathsf{2}\ \text{skeleton}_0.s_i \not\models B_i) \tag{3.50}$$

is valid. Given we use some heuristics to find a minimal skeleton or by chance, the initial skeleton could be $skeleton_0 = \mathsf{f}s_7\mathsf{g}$. Note that $s_7 \not\models B_1$ and $s_7 \not\models B_2$ and thus s_7 is a witness of the skeleton query (formula 3.43). Furthermore s_7 satisfies the single-state skeleton cycle query 3.49, since s_7 has a transition to itself and thus there is a successor state of s_7 which has a run (of length zero) to s_7. However s_7 is not reachable, since there is no run from s_4, the only initial state, to s_7. Thus it does not satisfy the stem query 3.45, which returns the proof $P = :s_7$, which is inductive and excludes s_7. This information P that only $:s_7 = :a\_ :b\_ :c$ states are reachable is now added to the constraint set R and thus s_7 will not be considered anymore. Note that $:s_7 = :a\_ :b\_ :c$ is easy to construct, and the set is an overapproximation of the reachable states[23]. Up until now, the reachability information is $\mathsf{R} = \mathsf{f}:s_7\mathsf{g}$, the list of

[23] While in the given case $:s_7 = :a\_ :b\_ :c$ are the reachable states, this is not guaranteed. In general this will result in an overapproximation.

walls is $\mathsf{W} = ;$ and the list of cycle constraints is $\mathsf{C} = ;$.

Since the information gained by considering this skeleton is processed, the next iteration is started and the next skeleton is calculated by executing the skeleton query 3.43 with the updated constraint sets. Note that since s_7 is not reachable, it is not possible to get another skeleton which contains s_7. Assume the new provided skeleton which has to be considered is $skeleton_1 = \{s_5\}$. Note that $s_5 \not\models B_1$ as well as $s_5 \not\models B_2$ and $s_5 \not\models R$ and thus is a potential witness of the skeleton query 3.43. Furthermore s_5 is reachable and thus satisfies the stem query. However, it does not satisfy the single-state skeleton cycle query, since there is no successor state of s_5 and thus no successor state of s_5 can reach s_5. The single-state skeleton cycle query 3.49 will thus return an inductive proof P of all states reachable from s_5 which is the empty set. Since $skeleton_1 = \{s_5\}$ is a single-state skeleton, this proof P is not simply added to the list of walls, instead $P = \mathsf{false}$ as well as $: P := : s_5$ are both added to W [24]. Next both sides of this new wall W_0 are tested if there is a potential skeleton on either side. Since obviously there is no possible skeleton in the empty set, but there is at least one possible skeleton (e.g. $\{s_0, s_1\}$) in $: s_5$, a new constraint $C = : s_5 \wedge : s_5^{\emptyset}$ is added to C. This constraint will restrict the transition relation since for future cycle queries only transitions starting in $: s_5$-states are relevant, which lead to other $: s_5$-states. Up until now, the reachability information is $\mathsf{R} = \{: s_7\}$, the list of walls is $\mathsf{W} = \{W_1 = false, : W_1 = : s_5\}$ and the list of cycle constraints is $\mathsf{C} = \{: s_5 \wedge : s_5^{\emptyset}\}$.

Again the skeleton query 3.43 is executed with the updated constraints, which could result in a new $skeleton_2 = \{s_0, s_2\}$. This could be a potential skeleton, since $s_0 \not\models B_1$, $s_2 \not\models B_2$ and

$$\forall s_i \in \text{skeleton}_2.[(s_i \not\models R) \wedge (s_i \not\models : W_1)] \tag{3.51}$$

thus all skeleton states are reachable and are $: W_1$-states, thus on the same side of the wall W_1. Therefore $skeleton_2 = \{s_0, s_2\}$ is a potential witness of the satisfiability of the skeleton query 3.43 with the constraints gathered so far. Next the stem query for s_0, which returns that s_0 is indeed reachable, as well as two cycle queries

$$reach\left(S, \bigwedge_{R \in \mathsf{R}} R(\overline{x}) \wedge \bigwedge_{C \in \mathsf{C}} C(\overline{x}), s_0, s_2\right) \tag{3.52}$$

and

$$reach\left(S, \bigwedge_{R \in \mathsf{R}} R(\overline{x}) \wedge \bigwedge_{C \in \mathsf{C}} C(\overline{x}), s_2, s_0\right) \tag{3.53}$$

are called in parallel. In this case and with the given constraints the first query 3.52 is satisfied, since there is a run from s_0 to s_2. However the second

[24] The negation of each $W_i \in \mathsf{W}$, has to be given explicitly, since the current skeleton state has to be removed from the set of potential skeleton states in the next iteration. Note that $P = \mathsf{false}$ does not contain s_5 anyway. Thus $: W_i$ is not achieved by only negating W_i.

query 3.53 is unsatisfiable and will return a proof $P = c$. Note that $c \;!\; c$ holds and c is therefore inductive as well as an overapproximation[25] of the states reachable from s_2. However s_0 is not a c-state and thus P is indeed an inductive proof separating s_2 from s_0. Since $skeleton_2 = \{s_0, s_2\}$ is not a single-state skeleton and P is the result of a cycle query, P as well as $\neg P$ are added to the set of walls $\mathcal{W}$. Since there is a skeleton on either side of this new wall, namely $\{s_0, s_1\}$, which are both $\neg c$-states, and $\{s_2, s_3\}$, which are both c-states, a new constraint $C = c \;\$\; c^{\emptyset}$ is added to the list of cycle constraints $\mathcal{C}$. Thus so far, the reachability information is $\mathcal{R} = \{\neg s_7\}$, the list of walls is $\mathcal{W} = \{W_1 = false, \neg W_1 = \neg s_5, W_2 = c, \neg W_2 = \neg c\}$ and the list of cycle constraints is $\mathcal{C} = \{\neg s_5 \wedge \neg s_5^{\emptyset}, c \;\$\; c^{\emptyset}\}$.

With the given constraints there are only two choices left for potential skeletons. Consider why this is the case. By the reachability information s_7 is not reachable. Furthermore, the skeleton has to lie one side of W_1 or $\neg W_1$, however, s_5 is on neither side of this wall. Therefore the only remaining states which satisfy B_1 are s_0 and s_3, and the only remaining states which satisfy B_2 are s_1 and s_2. However those are separated by the wall W_2 thus s_0 and s_3 as well as s_1 and s_2 lie in different arenas respectively. Thus if choosing s_0 which is a $\neg W_2$-state, one also has to choose s_1 and respectively when choosing s_3 this could only form a skeleton with s_2.

Given the next skeleton returned by the skeleton query 3.43 would be $skeleton_3 = \{s_0, s_1\}$, then s_0 would not be reachable by s_1. Therefore the cycle query 3.48 would return the proof $P = b \_ c$. Note that P is inductive, contains s_1 and does not contain s_0. This will result in the new wall $W_3 = b \_ c$ and $\neg W_3 = \neg b \wedge \neg c$. Next the skeleton query 3.43 is executed to check whether on both sides of the wall W_3 there is a potential skeleton. Note that this skeleton query is executed with the current set of constraints and thus with the whole set of walls and only the corresponding decision variable c_{W_3} is set once to true and once to false. If $\neg c_{W_3}$ holds, the skeleton query is unsatisfiable, however there is a skeleton if c_{W_3} holds. Thus the new constraint $C = (b \_ c) \wedge (b^{\emptyset} \_ c^{\emptyset})$ is added to the constraint list $\mathcal{C}$. The constraints so far are: The reachability information is $\mathcal{R} = \{\neg s_7\}$, the list of walls is $\mathcal{W} = \{W_1 = false, \neg W_1 = \neg s_5, W_2 = c, \neg W_2 = \neg c, W_3 = b \_ c, \neg W_3 = \neg b \wedge \neg c\}$ and the list of cycle constraints is $\mathcal{C} = \{\neg s_5 \wedge \neg s_5^{\emptyset}, c \;\$\; c^{\emptyset}, (b \_ c) \wedge (b^{\emptyset} \_ c^{\emptyset})\}$.

At this point there is only one possible skeleton which satisfies the skeleton query 3.43, namely $skeleton_4 = \{s_2, s_3\}$. Note that

$$\forall s_i \in \{s_2, s_3\}.[(s_i \models R) \wedge (s_i \models \neg W_1) \wedge (s_i \models W_2) \wedge (s_i \models W_3)] \tag{3.54}$$

is valid. Thus for each given wall W_i, both states lie on the same side of W_i and are reachable. Furthermore s_2 is reachable, thus the stem query 3.45 is satisfied, as well as both cycle queries 3.48. Since the stem query, as well as both cycle queries provide a corresponding run, s_2 as well as s_3 is proven to lie on a reachable fair cycle.

[25] This is because s_7 is a c-state, but s_7 is also not reachable from s_2.

There are two especially important things to note for this algorithm: (1) The witness returned by the skeleton SAT-query is an arbitrary set of states satisfying the skeleton query. Thus there is no order in which skeletons are discovered, rather it depends on the implementation of the SAT-solver which states are returned. However, the algorithm will terminate, since a potential witness will not be considered twice and there are no new potential skeleton states introduced. This introduces the disadvantage that the order of skeleton states can not be predicted and thus can not be used for potential improvement. However the structure of the provided system is also unknown, and therefore the *random* order of skeleton discoveries may increase the performance, compared to a predictable *unfortunate* selection of skeleton states.

(2) The inductive proof returned by the `reach` function is an overapproximation of the reachable states starting from the given state-set, at least excluding the state-set which should be reached. However, it is not guaranteed that more states than necessary are removed. However, Bradley proposed in his paper [Bra+11] to refine the proofs and thus potentially remove many more states from the inductive set of states.

So far in this section, we have considered an algorithm provided by Bradley in [Bra+11], which enables us to determine if there is a fair cycle in a given structure S. This fair cycle connects a set of states which we called a skeleton, in a way that at each given fairness constraint is satisfied at least once. Furthermore, it is shown that this cycle is reachable and thus there exists a way to infinitely often reach a set of properties. Therefore this method could be used to prove the existence or non-existence of a counterexample to the formula $\mathsf{AFG}\alpha$. This is satisfied, if for each initial state the following property holds: On each infinite path after finitely many steps a certain property has to hold in each state after this point in time. A counterexample would be a fair cycle on which $:a$ holds, thus if there is a lasso starting in an arbitrary initial state and on the loop of this lasso $:a$ holds. This way it is proven that there is at least one path on which $\mathsf{FG}\alpha$ does not hold since there is infinitely often a $:a$ on this path. However the non-existence of such a counterexample does prove $\mathsf{AFG}\alpha$ and thus `fair` is a proof engine to prove $\mathsf{AFG}\alpha$, since if there is a counterexample it is guaranteed to be found.

This is an important insight for the rest of the thesis, since the topic of this thesis is to find restrictions as well as capabilities of the previously mentioned PDR method[26], however it is shown in this section that a PDR implementation could be used to prove $\mathsf{AFG}\alpha$ by proving certain safety properties.

[26] A detailed description could be found in section 3.2

4. Findings

With PDR there is a powerful procedure to prove safety properties, which is widely used in verification of hardware and software circuits. However, despite the multitude of improvements in the efficiency of the algorithm, its use is still limited to proving safety properties.

The principle of the PDR method has been described as a inference rule:

$$\frac{\Psi_I \;!\; \Psi_0 \quad \bigwedge_{i=0}^{k\;1} \Psi_i \;!\; \Psi_{i+1} \;\wedge\; \Psi_{i+1} \quad \Psi_k \;!\; \Phi \quad \bigvee_{i=0}^{k\;1} \Psi_i \quad \Psi_{i+1}}{\Psi_{reach} \;!\; \Phi} \tag{4.1}$$

And earlier this year a paper was published at the Technical University of Kaiserslautern in which a set of similar inference rules for different CTL properties has been proposed. All of those inference rules share an important property since all of them rely on the generation of intermediate lemmas to achieve a proof. This generation of intermediate lemmas, however, is not trivial and except the PDR method, there is so far no method known to generate those intermediate lemmas except by guessing. However the results of PDR in terms of efficiency are outstanding and it seems promising to further consider inference rules for the discovery of different, yet similar proof methods for further CTL-operators. However, as a foundation, it is important to find and consider properties of those inference rules as well as of those intermediate lemmas to be able to close the gap to other disciples like mathematics and especially topologies and set-theory. Therefore, it seems promising to evaluate those inference rules, which might lead to discovering knowledge from different fields of research which is applicable to this use-case. Therefore the goal of this thesis is to consider the properties of such intermediate lemmas and those properties will be presented and discussed in this chapter.

4.1. Intermediate Lemmas vs. Invariants

When considering *property directed reachability* (PDR), as well as inference rules, often the term *invariant* is used. In computer science and especially in programming, this term is widely used to describe program properties. Thus an invariant is a set of assertions which does not change by executing a certain piece of code, while usually, this piece of code is a loop. Thus the invariant describes some assertions which do hold in each iteration of this loop.

First, consider the consequences of inductive sets of states. Given for example an inductive set Ψ of states in a Kripke structure K. Since this set is inductive, it will satisfy the property $\Psi \; ! \;$ Ψ. There are two possible interpretations of that. First, it assures that each state of Ψ is a state from that set of states which are predecessors of Ψ. Since there is a -operator, those predecessor states are only allowed to have transitions to Ψ-states. However, the more intuitive approach to describe an inductive set of states is to describe them by $\Psi \; ! \; \Psi$, which is the inverse of $\Psi \; ! \;$ Ψ. Instead of asserting that each state is a universal predecessor of Ψ, we could also assert that each successor state of each Ψ-state is also a Ψ-state. This is because an inductive set of states is a set of states which can not be left and is thus closed under the transition relation. Since Ψ is closed and each transition taken from a Ψ-state will, in turn, lead to a Ψ-state, an inductive set of states is also called an invariant.

However, this name is not suitable in general when considering all fixpoint inference rules and especially the general fixpoint inference rule for greatest fixpoints:

$$\frac{\Psi_I \; ! \; \psi \quad \psi \; ! \; [\varphi]_x^{\psi}}{\Psi_I \; ! \; (\nu x.\varphi)} \tag{4.2}$$

as well as the rule for least fixpoints:

$$\frac{\psi_0 \; ! \; [\varphi]_x^{\mathsf{false}} \quad \bigwedge_{i=0}^{n\ 1} (\psi_i \; ! \; \psi_{i+1}) \quad \bigwedge_{i=0}^{n\ 1} \left(\psi_{i+1} \; ! \; [\varphi]_x^{\psi_i}\right) \quad \Psi_I \; ! \; \psi_n}{\Psi_I \; ! \; \mu x.\varphi} \tag{4.3}$$

Both of those rules enforce $[\varphi]_x^{\psi}$ to be a post fixpoint. Note that x is a post-fixpoint, if $x \; \vee \; f(x)$ is satisfied. This closely resembles the assertions $\psi \; ! \; [\varphi]_x^{\psi}$ as well as $\bigwedge_{i=0}^{n\ 1} \left(\psi_{i+1} \; ! \; [\varphi]_x^{\psi_i}\right)$, since in both cases the application of φ only adds states to the current set of states. Thus ψ is a post-fixpoint of φ.

Consider this in the context of greatest fixpoints and especially a safety property α. In this case φ is substituted by the monotonic[1] function $x \wedge \alpha$.

[1] This function has to be monotonic, as it is guarded by a fixpoint operator in the conclusion of the inference rule.

As discussed in 3.3 the greatest fixpoint $\nu x.\ \ x \wedge \alpha$ consists of all states which satisfy $\mathsf{AG}\alpha$. Note that in this case the inference rule has the form:

$$\frac{\Psi_I \ ! \ \psi \quad \psi \ ! \ \alpha \wedge \quad \psi}{\Psi_I \ ! \ \mathsf{AG}\alpha} \tag{4.4}$$

and since $a \ ! \ b \wedge c$ is equivalent to $(a \ ! \ b) \wedge (a \ ! \ c)$ this inference rule could also be stated as:

$$\frac{\Psi_I \ ! \ \psi \quad \psi \ ! \quad \psi \quad \psi \ ! \ \alpha}{\Psi_I \ ! \ \mathsf{AG}\alpha} \tag{4.5}$$

While the details of these formulas are discussed already in section 3.3, there is still some information gained at this point with regard to the term *invariant*. The second assertion of this inference rule states that ψ has to be inductive since $\psi \ ! \ \psi$ has to hold. Thus ψ is an assertion which still holds after applying the transition relation and will even hold if the transition relation is applied infinitely often since ψ can not be left. Therefore ψ could be called an invariant since it is an assertion which will hold when considering the successor set of ψ.

However, we considered the inference rule which describes the principle of PDR and forms a sequence of inductive assertions:

$$\frac{\Psi_I \ ! \ \psi_0 \quad \bigwedge_{i=0}^{k \ 1} \psi_i \ ! \ \psi_{i+1} \wedge \quad \psi_{i+1} \quad \psi_k \ ! \ \alpha \wedge \psi_k \ _1}{\Psi_I \ ! \ \mathsf{AG}\alpha} \tag{4.6}$$

A visualization of the sets of states generated by this formula is shown in figure 3.7 and the details of this formula as well as its origin are shown in section 3.3. Note that despite allowing the same inference, this rule does not enforce inductiveness of each ψ. Only ψ_k has to be inductive. Since the ψ_i are not inductive but rather form a growing set of states, the term *invariant* should not be used in this case. This is because the assertion ψ_i will most likely not hold when considering the set of successor states of ψ_i and thus each ψ_i is not a property which does not change.

Next, consider the context of the least fixpoint, like for example a liveness property. Recent publications (see [KS19], [Bra+11]) have shown that this case is more complicated than the greatest fixpoint. Given for example a property $\mathsf{EF}\beta$. This property does hold for all states which have a finite path to a β-state. To prove this property to hold for each initial state, the finiteness of each path has to be proven as well. This, in turn, leads to the problem that some form of ranking function[2] has to be implemented. Given some sets of states, however, this has to be implemented by a sequence of assertions since a single set does not provide sufficient ranking information. This fact can also

[2]Like for example the Noetherian ordering ξ which has been described in section 3.3.

be seen in formula 4.3 and gets more apparent when φ is substituted by $\beta\_ \quad x$, in which case the formula reduces to:

$$\frac{\Psi_I \ ! \ \psi_0 \quad \bigwedge_{i=0}^{n \ 1} (\psi_i \ ! \ \beta\_ \quad \psi_{i+1}) \quad \psi_n \ ! \ \alpha}{\Psi_I \ ! \ \mathsf{EF}\beta} \tag{4.7}$$

This formula has already been discussed in section 3.3 and the sets of states are visualized in figure 3.8. Note that the second assertion $\bigwedge_{i=0}^{n \ 1} (\psi_i \ ! \ \psi_{i+1})$ of the inference rule 4.3 is not necessary, since it does not provide additional information and enforces that each ψ_i is a post-fixpoint. However, this could already be derived from the other three assertions which do remain. The important fact is that all ψ_i as a whole form a ranking, since each ψ_i encodes the maximum distance of each ψ_i-state to a β-state. While it was possible to formulate an inference rule for greatest-fixpoints with a sequence of inductive assertions, this is now necessary in the case of least fixpoints. Furthermore, in the case of least fixpoints, this sequence does not need to end in an inductive set and thus each ψ_i might differ from $\psi_{i \ 1}$.

Since this is the case, the term *invariant* is not suitable when referring to the intermediate sets necessary for inference rules. For this reason, I suggest the name *intermediate lemmas* for those intermediate sets of states and this term will be used for the rest of this thesis.

4.2. Homomorphisms of Monotonic Functions

Given the inference rules for least and for greatest fixpoints, as shown in rule 4.3 and 4.2. In both formulas, a substitution $[\varphi]_x^a$ is used, which could be interpreted as a function. First, consider why the substitution could be considered as a function:

This is the case because any arbitrary propositional logic function evaluates to a boolean value under a certain assignment to all its variables and each variable has two potential values, namely true and false. Thus a variable x in a formula φ could be substituted by a formula ψ which results in a new formula $[\varphi]_x^\psi$, which does not depend on x any more but only on the variables of ψ and the remaining variables of φ.

As an example, given an arbitrary formula φ which contains the variable x and a corresponding variable ordering for all variables which occur in φ. Assume that this formula is evaluated by constructing the corresponding BDD[3]. In this BDD each occurrence of the variable x imposes a decision where either a high or a low subtree gets chosen if x is either 1 or 0 respectively. If this variable x in this BDD is substituted by a BDD of formula ψ in a way, that each time ψ evaluates to true, this leaf is substituted by the high subtree and each time it evaluates to false the low subtree is chosen. This new BDD corresponds to formula $[\varphi]_x^a$.

Consider the following example: Given the formula $\varphi = \quad x \wedge \alpha$ and a structure K. Then φ is satisfied by all states of K which are a predecessor state of x and which satisfies α. In this formula, x is a variable and could be substituted by the formula ψ which describes a set of states of K. By substituting x by ψ, $[\varphi]_x^\psi$ will be satisfied by all states which are a predecessor of a ψ-state and thus φ could be considered a function. This understanding is crucial when considering inference rules, since this substitution operator is the foundation of the general inference rules for least an greatest fixpoints (see formula 4.2 and 4.3).

Note that given φ is a function in a μ-calculus formula, this function needs to be monotonic. This is important because otherwise the existence of a fixpoint is not proven. The function φ is monotonic if the variables guarded by fixpoint operators occur only positive, thus covered by an even number of negations.[4]

Since the substitution $[\varphi]_x^\psi$ is an important element of the inference rules for least and greatest fixpoint, it seems promising to further evaluate those substitution. The goal is to find properties which might lead to insight with regard to the intermediate lemmas of the inference rules. To this end, some new and remarkable properties will be presented in the rest of this chapter.

Lemma 1 (Conjunctions of Substitutions)**.** *For all formulas α, β and φ and*

[3] *Binary Decision Diagrams* (BDDs) are explained in detail in [Sch19b]. Essentially this diagram depicts all evaluations of a formula and is thus an alternative definition of a formula.

[4] This is proven in [Sch19b], however, this insight is quite intuitive. Note that a function $f : D \to E$ is monotonic iff $x \sqsubseteq_D y \to f(x) \sqsubseteq_E f(y)$ as discussed in chapter 2.6.

any variable x, the following formula is valid:

$$[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \;!\; [\varphi]_x^{\alpha \wedge \beta} \tag{4.8}$$

Proof. Consider the valuation of φ under an assignment to α and β. The possible evaluations of the formula α and β lead to the following cases:

α	β	$[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \;!\; [\varphi]_x^{\alpha \wedge \beta}$
0	0	$[\varphi]_x^0 \wedge [\varphi]_x^0 \;!\; [\varphi]_x^0$
0	1	$[\varphi]_x^0 \wedge [\varphi]_x^1 \;!\; [\varphi]_x^0$
1	0	$[\varphi]_x^1 \wedge [\varphi]_x^0 \;!\; [\varphi]_x^0$
1	1	$[\varphi]_x^1 \wedge [\varphi]_x^1 \;!\; [\varphi]_x^1$

Note that $a \wedge b \;!\; a$ and $a \wedge b \;!\; b$ are both valid, since if both a and b hold on the left side of the implication, they also have to hold on the right side of the implication. Since all four cases are valid, the formula $[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \;!\; [\varphi]_x^{\alpha \wedge \beta}$ is valid. □

Lemma 2 (Disjunctions of Substitutions)**.** *For all formulas α, β and φ and any variable x, the following formula is valid:*

$$[\varphi]_x^{\alpha \_ \beta} \;!\; [\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta} \tag{4.9}$$

Proof. Consider the valuation of φ under an assignment to α and β. The possible evaluations of the formula α and β lead to the following cases:

α	β	$[\varphi]_x^{\alpha \_ \beta} \;!\; [\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta}$
0	0	$[\varphi]_x^0 \;!\; [\varphi]_x^0 \_ [\varphi]_x^0$
0	1	$[\varphi]_x^1 \;!\; [\varphi]_x^0 \_ [\varphi]_x^1$
1	0	$[\varphi]_x^1 \;!\; [\varphi]_x^1 \_ [\varphi]_x^0$
1	0	$[\varphi]_x^1 \;!\; [\varphi]_x^1 \_ [\varphi]_x^1$

Note that each case results in a valid formula, since $a \;!\; a \_ b$ is valid for any formula a and b. Since all four cases result in a valid formula, $[\varphi]_x^{\alpha \_ \beta} \;!\; [\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta}$ is also valid. □

Both rules shown in lemma 1 and in lemma 2 are valid for any arbitrary formula φ, however, in the application of the substitution which is considered in this thesis the formula φ is constrained to be monotonic. This allows to further strengthen both lemmas.

Lemma 3 (Conjunctions of Substitutions for Monotonic Functions)**.** *For all formulas α, β and φ and any variable x that has only positive occurrences in φ, the following formula is valid:*

$$[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \;\$\; [\varphi]_x^{\alpha \wedge \beta} \tag{4.10}$$

Proof. If x has only positive occurrences in φ, then the function defined by φ is monotonic. By monotonicity and the fact that $\alpha \wedge \beta \sqsubseteq \alpha$ and $\alpha \wedge \beta \sqsubseteq \beta$ are valid, we can conclude that $[\varphi]_x^{\alpha \wedge \beta} \sqsubseteq [\varphi]_x^{\alpha}$ and $[\varphi]_x^{\alpha \wedge \beta} \sqsubseteq [\varphi]_x^{\beta}$ are both valid. Thus the formula $[\varphi]_x^{\alpha \wedge \beta} \sqsubseteq [\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta}$ is also valid. Combined with the previous lemma 1, which proves that $[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \sqsubseteq [\varphi]_x^{\alpha \wedge \beta}$ is valid, this proves that $[\varphi]_x^{\alpha} \wedge [\varphi]_x^{\beta} \mathbin{\$} [\varphi]_x^{\alpha \wedge \beta}$ is a valid formula as well, given that there are only positive occurrences of x in φ. □

Analogous this could be stated for the disjunction as well with a similar proof:

Lemma 4 (Disjunctions of Substitutions for Monotonic Functions)**.** *For all formulas α, β and φ and any variable x that has only positive occurrences in φ, the following formula is valid:*

$$[\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta} \mathbin{\$} [\varphi]_x^{\alpha \_ \beta} \tag{4.11}$$

Proof. If x has only positive occurrences in φ, then the function defined by φ is monotonic. By monotonicity and the fact that $\alpha \sqsubseteq \alpha \_ \beta$ and $\beta \sqsubseteq \alpha \_ \beta$ are valid, we can conclude that $[\varphi]_x^{\alpha} \sqsubseteq [\varphi]_x^{\alpha \_ \beta}$ and $[\varphi]_x^{\beta} \sqsubseteq [\varphi]_x^{\alpha \_ \beta}$ are both valid. Thus the formula $[\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta} \sqsubseteq [\varphi]_x^{\alpha \_ \beta}$ is also valid. Combined with the previous lemma 2, which proves that $[\varphi]_x^{\alpha \_ \beta} \sqsubseteq [\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta}$ is valid, this proves that $[\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta} \mathbin{\$} [\varphi]_x^{\alpha \_ \beta}$ is a valid formula as well, given that there are only positive occurrences of x in φ. □

Those lemmas 1-4 have shown to be helpful when considering intermediate lemmas of least and greatest fixpoints and will be used in the following chapters. However, it should be noted that lemma 1 and lemma 2 hold in general and are thus widely applicable and are strengthened if φ only has positive occurrences of x by lemma 3 and 4. This leads to two additional interesting properties:

Lemma 5 (Closure of pre-fixpoints)**.** *Let A, B and φ be formulas and x an arbitrary variable. Given that $([\varphi]_x^{A} \sqsubseteq A) \wedge ([\varphi]_x^{B} \sqsubseteq B)$ holds, the following formulas are valid:*

$$[\varphi]_x^{A \wedge B} \sqsubseteq (A \wedge B) \tag{4.12}$$

$$[\varphi]_x^{A \_ B} \sqsubseteq (A \_ B) \tag{4.13}$$

and respectively given $([\varphi]_x^{A} \sqsubseteq A) \_ ([\varphi]_x^{B} \sqsubseteq B)$ the following formula is valid:

$$[\varphi]_x^{A \_ B} \sqsubseteq (A \_ B) \tag{4.14}$$

Proof. In order to proof $[\varphi]_x^{A \wedge B} \sqsubseteq (A \wedge B)$ it is sufficient to prove that if $(A \wedge B)$ evaluates to false, then $[\varphi]_x^{A \wedge B}$ has to evaluate to false as well.

Consider a valuation ϕ with $\phi(A \wedge B) = \mathsf{false}$. Then either $\phi(A)$ will evaluate to false or $\phi(B)$ will evaluate to false. By the hypothesis follows, if $\phi(A)$

evaluates to false (respectively $\phi(B)$), then $\phi([\varphi]_x^A)$ also evaluates to false (respectively $\phi([\varphi]_x^B)$) and thus also $\phi([\varphi]_x^{A \wedge B}) = \mathsf{false}$.

Similarly consider a valuation ϕ with $\phi(A \_ B) = \mathsf{false}$. Then $\phi(A)$ will evaluate to false as well as $\phi(B)$. By the hypothesis follows, that if $\phi(A)$ evaluates to false and $\phi(B)$ evaluate to false, then either $\phi([\varphi]_x^A)$ evaluates to false or $\phi([\varphi]_x^A)$ evaluates to false and thus also $\phi([\varphi]_x^{A \_ B})$ (which is equivalent to $[\varphi]_x^{\mathsf{false}}$ under this valuation ϕ).
By that we can conclude, that given the hypothesis $([\varphi]_x^A \ ! \ A) \wedge ([\varphi]_x^B \ ! \ B)$ the following will hold:

$$[\varphi]_x^{A \wedge B} \ ! \ (A \wedge B)$$

and given $([\varphi]_x^A \ ! \ A) \_ ([\varphi]_x^B \ ! \ B)$ the following will hold:

$$[\varphi]_x^{A \_ B} \ ! \ (A \_ B)$$

Since $([\varphi]_x^A \ ! \ A) \_ ([\varphi]_x^B \ ! \ B)$ is sufficient such that $[\varphi]_x^{A \_ B} \ ! \ (A \_ B)$ holds, we can conclude that given $([\varphi]_x^A \ ! \ A) \wedge ([\varphi]_x^B \ ! \ B)$ the formula $[\varphi]_x^{A \_ B} \ ! \ (A \_ B)$ is valid as well. □

Furthermore this also holds for the inverse direction of the implication, which is stated in the following lemma:

Lemma 6 (Closure of post-fixpoints)**.** *Let A, B and φ be formulas and x an arbitrary variable. Given that $(A \ ! \ [\varphi]_x^A) \wedge (B \ ! \ [\varphi]_x^B)$ holds, the following formulas are valid:*

$$(A \wedge B) \ ! \ [\varphi]_x^{A \wedge B} \tag{4.15}$$

$$(A \_ B) \ ! \ [\varphi]_x^{A \_ B} \tag{4.16}$$

Proof. Assume that $(A \ ! \ [\varphi]_x^A) \wedge (B \ ! \ [\varphi]_x^B)$ holds. Thus φ has to contain only positive occurrences of x. Furthermore by the means of propositional logic, $(a \ ! \ b) \wedge (c \ ! \ d)$ implies $(a \wedge c) \ ! \ (b \wedge d)$ as well as $(a \_ c) \ ! \ (b \_ d)$. Thus we can conclude that by the assumption $(A \wedge B) \ ! \ ([\varphi]_x^A \wedge [\varphi]_x^B)$ and $(A \_ B) \ ! \ ([\varphi]_x^A \_ [\varphi]_x^B)$ holds. Furthermore since φ describes a monotonic function we can apply lemma 3 and 4, which states that $[\varphi]_x^\alpha \_ [\varphi]_x^\beta \ \$ \ [\varphi]_x^{\alpha \_ \beta}$ and $[\varphi]_x^\alpha \wedge [\varphi]_x^\beta \ \$ \ [\varphi]_x^{\alpha \wedge \beta}$ are valid. Thus we can conclude that $(A \wedge B) \ ! \ [\varphi]_x^{A \wedge B}$ and $(A \_ B) \ ! \ [\varphi]_x^{A \_ B}$ are valid as well. □

Consider and recap those formulas presented in this chapter. Given any formulas α, β and φ as well as a variable x in φ, then the following formulas have been proven to be valid:

$$[\varphi]_x^\alpha \wedge [\varphi]_x^\beta \ ! \ [\varphi]_x^{\alpha \wedge \beta} \tag{4.17}$$

$$[\varphi]_x^{\alpha \_ \beta} \ ! \ [\varphi]_x^\alpha \_ [\varphi]_x^\beta \tag{4.18}$$

Provided that $([\varphi]_x^\alpha \ ! \ \alpha) \wedge ([\varphi]_x^\beta \ ! \ \beta)$ holds, the following formulas are valid:

$$[\varphi]_x^{\alpha \wedge \beta} \ ! \ (\alpha \wedge \beta) \tag{4.19}$$

$$[\varphi]_x^{\alpha \_ \beta} \ ! \ (\alpha \_ \beta) \tag{4.20}$$

Similarly, given that $([\varphi]_x^\alpha \ ! \ \alpha) \_ ([\varphi]_x^\beta \ ! \ \beta)$ holds, we can conclude that this formula is valid:

$$[\varphi]_x^{\alpha \_ \beta} \ ! \ (\alpha \_ \beta) \tag{4.21}$$

Provided that $(\alpha \ ! \ [\varphi]_x^\alpha) \wedge (\beta \ ! \ [\varphi]_x^\beta)$ holds, the following formulas are valid:

$$(\alpha \wedge \beta) \ ! \ [\varphi]_x^{\alpha \wedge \beta} \tag{4.22}$$

$$(\alpha \_ \beta) \ ! \ [\varphi]_x^{\alpha \_ \beta} \tag{4.23}$$

And furthermore if it is given that all occurrences of x in φ are positive, thus guarded by an even number of negations, then the following two formulas are valid:

$$[\varphi]_x^\alpha \wedge [\varphi]_x^\beta \ \$ \ [\varphi]_x^{\alpha \wedge \beta} \tag{4.24}$$

$$[\varphi]_x^\alpha \_ [\varphi]_x^\beta \ \$ \ [\varphi]_x^{\alpha \_ \beta} \tag{4.25}$$

It is important to note that the application of those presented rules is not limited to the context of inference rules, but they are applicable in general when handling propositional logic substitution.

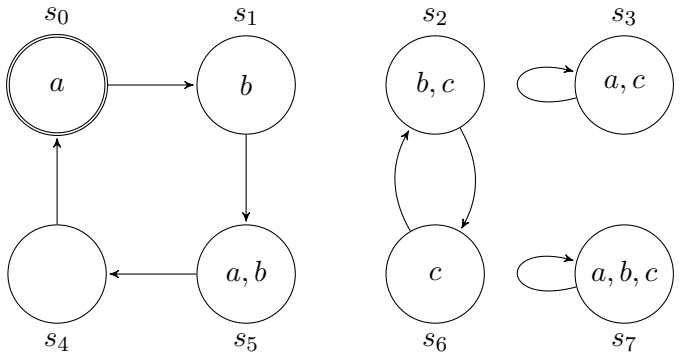

Figure 4.1.: *An example structure K. This structure satisfies $\mathsf{AG}(:a\_ :b\_ :c)$, since the only outgoing infinite path of the initial state only consists of $(:a\_ :b\_ :c)$-states.*

4.3. Lattice properties of intermediate lemmas

The goal of this thesis was to find and prove properties of the intermediate lemmas used in inference rule based fixpoint calculations, to find a way to generate those intermediate lemmas. Note that given a sequence of intermediate lemmas, the inference rules presented previously allow to decide if this given sequence is sufficient as a proof for a certain property. Thus they allow to infer a property if those intermediate lemmas satisfy some assertions. Since those inference rules are correct and complete, the existence of those lemmas is guaranteed. Thus provided a property holds for which an inference rule is given, then it is proven that corresponding intermediate lemmas exist. However, only the existence of at least one lemma is given and it is not stated how many sufficient lemmas do exist.

Consider for example the structure K shown in figure 4.1 and the property $\mathsf{AG}:s_7 = \mathsf{AG}(:a\_ :b\_ :c)$. Note that $K \not\models \mathsf{AG}:s_7$, since the only initial state s_0 has only one infinite path starting in s_0 which consists of only $:s_7$ states. This is because s_7 is not reachable from any other state. Furthermore note that all states except s_7 are $:s_7$-states and all infinite paths which are started in any other state then s_7 satisfy $\mathsf{G}:s_7$, because s_7 is not reachable from any other state.

Consider the inductive sets of states of the structure K. An inductive subset is a set ψ which satisfies the following property: $\psi \; ! \; \psi$. Thus it is not possible to reach a $:\psi$-state by choosing any transition from any state in ψ. Note that there are multiple minimal inductive sets in K:

$$\mathsf{J}\psi 1\mathsf{K}_K := \mathsf{J}:c\mathsf{K}_K = \mathsf{f}s_0, s_1, s_4, s_5\mathsf{g} \tag{4.26}$$
$$\mathsf{J}\psi 2\mathsf{K}_K := \mathsf{J}:a \wedge c\mathsf{K}_K = \mathsf{f}s_2, s_6\mathsf{g} \tag{4.27}$$
$$\mathsf{J}\psi 3\mathsf{K}_K := \mathsf{J}a \wedge :b \wedge c\mathsf{K}_K = \mathsf{f}s_3\mathsf{g} \tag{4.28}$$
$$\mathsf{J}\psi 4\mathsf{K}_K := \mathsf{J}a \wedge b \wedge c\mathsf{K}_K = \mathsf{f}s_4\mathsf{g} \tag{4.29}$$

Those sets are generated by choosing an arbitrary state α in K and adding

successor states of α to ψ until the set is inductive.

Lemma 7 (Union of Inductive Sets)**.** *If* $\psi_1 \;!\; \psi_1$ *and* $\psi_2 \;!\; \psi_2$*, then the following formula is valid*[5]*:*

$$\psi_1 \_ \psi_2 \;!\; (\psi_1 \_ \psi_2) \tag{4.30}$$

Proof. If $(a \;!\; b) \wedge (c \;!\; d)$ holds, we can conclude that $(a \_ c) \;!\; (b \_ d)$ is valid and thus $(\psi_1 \;!\; \psi_1) \wedge (\psi_2 \;!\; \psi_2)$ implies formula $(\psi_1 \_ \psi_2) \;!\; (\psi_1 \_ \psi_2)$. It remains to show that $(\psi_1 \_ \psi_2) \;!\; (\psi_1 \_ \psi_2)$. Note that $\psi_1 = [\varphi]_x^{\psi_1}$ with $\varphi := x$. Thus φ only contains positive occurrences of x. By lemma 4 we can conclude that $[\varphi]_x^{\alpha} \_ [\varphi]_x^{\beta} \;\$\; [\varphi]_x^{\alpha \_ \beta}$ holds and thus $[\varphi]_x^{\psi_1} \_ [\varphi]_x^{\psi_2} \;\$\; [\varphi]_x^{\psi_1 \_ \psi_2}$. Therefore it is proven that $(\psi_1 \_ \psi_2) \;!\; (\psi_1 \_ \psi_2)$ is valid and thus $\psi_1 \_ \psi_2 \;!\; (\psi_1 \_ \psi_2)$ is also valid. □

The previous lemma 7 provides, that given two sets are inductive, the union of those sets is inductive as well. By formula 4.26-4.29 four inductive sets are given and each of those sets is minimal. Being minimal asserts, that if one or more states are dropped from any of those sets they will not remain to be inductive and still contain at least one state. By lemma 7 each union of those inductive sets is an inductive set as well. Which results in:

$$\mathsf{J}\psi_1\mathsf{K}_K := \mathsf{J}: c\mathsf{K}_K = \mathsf{f}s_0, s_1, s_4, s_5\mathsf{g} \tag{4.31}$$
$$\mathsf{J}\psi_2\mathsf{K}_K := \mathsf{J}: a \wedge c\mathsf{K}_K = \mathsf{f}s_2, s_6\mathsf{g} \tag{4.32}$$
$$\mathsf{J}\psi_3\mathsf{K}_K := \mathsf{J}a \wedge : b \wedge c\mathsf{K}_K = \mathsf{f}s_3\mathsf{g} \tag{4.33}$$
$$\mathsf{J}\psi_4\mathsf{K}_K := \mathsf{J}a \wedge b \wedge c\mathsf{K}_K = \mathsf{f}s_7\mathsf{g} \tag{4.34}$$
$$\mathsf{J}\psi_{1,2}\mathsf{K}_K := \mathsf{J}: c \_ (: a \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_2, s_4, s_5, s_6\mathsf{g} \tag{4.35}$$
$$\mathsf{J}\psi_{1,3}\mathsf{K}_K := \mathsf{J}: c \_ (a \wedge : b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_3, s_4, s_5\mathsf{g} \tag{4.36}$$
$$\mathsf{J}\psi_{1,4}\mathsf{K}_K := \mathsf{J}: c \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_4, s_5, s_7\mathsf{g} \tag{4.37}$$
$$\mathsf{J}\psi_{2,3}\mathsf{K}_K := \mathsf{J}(: a \wedge c) \_ (a \wedge : b \wedge c)\mathsf{K}_K = \mathsf{f}s_2, s_3, s_6\mathsf{g} \tag{4.38}$$
$$\mathsf{J}\psi_{2,4}\mathsf{K}_K := \mathsf{J}(: a \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_2, s_6, s_7\mathsf{g} \tag{4.39}$$
$$\mathsf{J}\psi_{3,4}\mathsf{K}_K := \mathsf{J}(: a \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_3, s_7\mathsf{g} \tag{4.40}$$
$$\mathsf{J}\psi_{1,2,3}\mathsf{K}_K := \mathsf{J}: c \_ (: a \wedge c) \_ (a \wedge : b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_2, s_3, s_4, s_5, s_6\mathsf{g} \tag{4.41}$$
$$\mathsf{J}\psi_{1,2,4}\mathsf{K}_K := \mathsf{J}: c \_ (: a \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_2, s_4, s_5, s_6, s_7\mathsf{g} \tag{4.42}$$
$$\mathsf{J}\psi_{1,3,4}\mathsf{K}_K := \mathsf{J}: c \_ (a \wedge : b \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_3, s_4, s_5, s_7\mathsf{g} \tag{4.43}$$
$$\mathsf{J}\psi_{2,3,4}\mathsf{K}_K := \mathsf{J}(: a \wedge c) \_ (a \wedge : b \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_2, s_3, s_6, s_7\mathsf{g} \tag{4.44}$$
$$\mathsf{J}\psi_{1,2,3,4}\mathsf{K}_K := \mathsf{J}c \_ (: a \wedge c) \_ (a \wedge : b \wedge c) \_ (a \wedge b \wedge c)\mathsf{K}_K = \mathsf{f}s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\mathsf{g} \tag{4.45}$$

[5] Note that the union of sets of states could be expressed by a disjunction of the formulas which describe those sets of states. This has been discussed in detail in chapter 2.

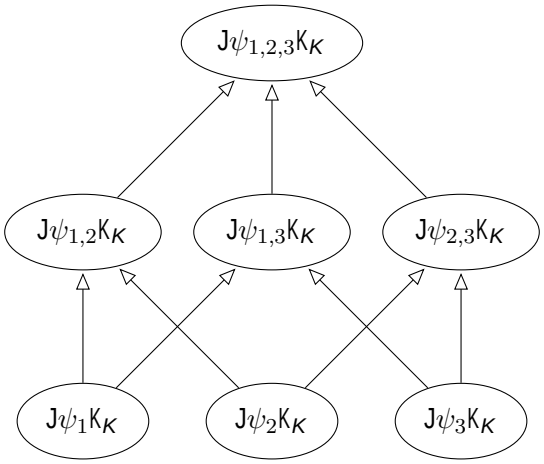

Figure 4.2.: *The Hasse diagram of all inductive sets defined in equation 4.47-4.53 of the structure defined in figure 4.1. Note that all sets $\mathsf{J}\psi_i\mathsf{K}_K$ satisfy $\psi_i \;!\; \psi_i$ and $\psi_i \;!\; :s_7$. The relation shown is the subset relation between sets.*

Those sets are all inductive sets which could be derived from structure K shown in figure 4.1. Consider the inference rule for the safety property $\mathsf{AG}\,{:}\,s_7$ again:

$$\frac{\Psi_I \;!\; \psi \quad \psi \;!\; \psi \quad \psi \;!\; :s_7}{\Psi_I \;!\; \mathsf{AG}\,{:}\,s_7} \tag{4.46}$$

Each of those inductive sets ψ_i in 4.31-4.45 satisfies the second assertion $\psi \;!\; \psi$. However, some of them do not satisfy the third assertion $\psi \;!\; :s_7$, since they contain the state s_7. Those inductive sets which do not satisfy $\psi \;!\; :s_7$ are all the sets which are constructed by using ψ_4, since ψ_1, ψ_2 and ψ_3 do not contain s_7 (and thus the union of those sets will not contain s_7). Therefore the sets which satisfy $\psi \;!\; \psi$ and $\psi \;!\; :s_7$ are all the sets which are constructed by joining ψ_1, ψ_2 and ψ_3:

$$\mathsf{J}\psi_1\mathsf{K}_K := \mathsf{J}\,{:}\,c\mathsf{K}_K = \{s_0, s_1, s_4, s_5\} \tag{4.47}$$
$$\mathsf{J}\psi_2\mathsf{K}_K := \mathsf{J}\,{:}\,a \wedge c\mathsf{K}_K = \{s_2, s_6\} \tag{4.48}$$
$$\mathsf{J}\psi_3\mathsf{K}_K := \mathsf{J}a \wedge {:}\,b \wedge c\mathsf{K}_K = \{s_3\} \tag{4.49}$$
$$\mathsf{J}\psi_{1,2}\mathsf{K}_K := \mathsf{J}\,{:}\,c \_ ({:}\,a \wedge c)\mathsf{K}_K = \{s_0, s_1, s_2, s_4, s_5, s_6\} \tag{4.50}$$
$$\mathsf{J}\psi_{1,3}\mathsf{K}_K := \mathsf{J}\,{:}\,c \_ (a \wedge {:}\,b \wedge c)\mathsf{K}_K = \{s_0, s_1, s_3, s_4, s_5\} \tag{4.51}$$
$$\mathsf{J}\psi_{2,3}\mathsf{K}_K := \mathsf{J}({:}\,a \wedge c) \_ (a \wedge {:}\,b \wedge c)\mathsf{K}_K = \{s_2, s_3, s_6\} \tag{4.52}$$
$$\begin{aligned}\mathsf{J}\psi_{1,2,3}\mathsf{K}_K :=& \mathsf{J}\,{:}\,c \_ ({:}\,a \wedge c) \_ (a \wedge {:}\,b \wedge c)\mathsf{K}_K \\ =& \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}\end{aligned} \tag{4.53}$$

Note that the sets listed above are all possible sets of states ψ in K which satisfy $\psi \;!\; \psi$ and $\psi \;!\; :s_7$. In figure 4.2 the subset relation of those ψ is visualized. Note that there is a maximum element. This maximum element of

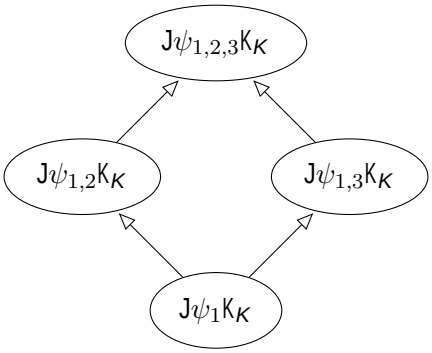

Figure 4.3.: *The Hasse diagram of all intermediate lemmas ψ_i shown in equation 4.54-4.57. Each of those ψ is a proof of $\mathsf{AG}\,{:}\,s_7$, since each ψ_i contains all initial state, is inductive and does only consist of ${:}\,s_7$-states. Note that they form a lattice.*

all ψ with respect to the subset relation is the set of states which contains all minimal inductive sets. Therefore this ***greatest*** inductive set corresponds to the greatest fixpoint since it contains all states of the structure K except the ${:}\,s_7$-state itself as well as excluding all states which have a path to a ${:}\,s_7$-state.

Given the inference rule for $\mathsf{AG}\,{:}\,s_7$, shown in formula 4.46. So far we considered the second as well as the third assertion of the inference rule, however, we need to make sure that the first assertion holds as well. If this is the case and all three assertions hold for any given ψ, then each of those sets ψ is sufficient to prove $\mathsf{AG}\,{:}\,s_7$, since the inference rule is correct. To ensure that the first assertion holds, only sets are allowed which contain all initial states. For the given structure the only initial state is s_0 and thus each valid ψ has to contain at least ψ_1, since only ψ_1 contains s_0 and ψ_1, ψ_2 and ψ_3 are all possible minimal inductive sets which do not contain s_7. By this restriction, the only remaining ψ_i which satisfy all three assertions are:

$$\mathsf{J}\psi_1\mathsf{K}_K := \mathsf{J}\,{:}\,c\mathsf{K}_K = \{s_0, s_1, s_4, s_5\} \tag{4.54}$$

$$\mathsf{J}\psi_{1,2}\mathsf{K}_K := \mathsf{J}\,{:}\,c \_ ({:}\,a \wedge c)\mathsf{K}_K = \{s_0, s_1, s_2, s_4, s_5, s_6\} \tag{4.55}$$

$$\mathsf{J}\psi_{1,3}\mathsf{K}_K := \mathsf{J}\,{:}\,c \_ (a \wedge {:}\,b \wedge c)\mathsf{K}_K = \{s_0, s_1, s_3, s_4, s_5\} \tag{4.56}$$

$$\begin{aligned}\mathsf{J}\psi_{1,2,3}\mathsf{K}_K :=& \mathsf{J}\,{:}\,c \_ ({:}\,a \wedge c) \_ (a \wedge {:}\,b \wedge c)\mathsf{K}_K \\ &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}\end{aligned} \tag{4.57}$$

First, consider those minimal inductive sets of states. As mentioned before, they are constructed by choosing a set of states α and then adding successor states until this set is inductive. Thus those minimal inductive sets are the set of all states reachable from an α-state. Note that they are only minimal in size with regard to the α-states. An example is shown in figure 4.4. In this given case the states reachable from state s_1 are $s_{1reach} = \{s_1, s_2, s_3\}$, which is a subset of $s_{0reach} = \{s0, s_1, s_2, s_3\}$.

Furthermore, the construction of those minimal inductive sets is a fixpoint

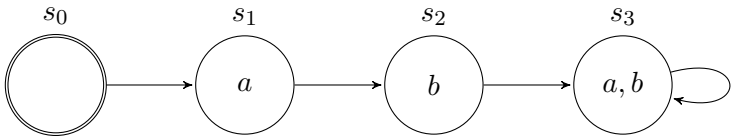

Figure 4.4.: *In this structure, the minimal inductive set of s_3 is a subset of the minimal inductive set of s_2, which in turn is a subset of the minimal inductive set of s_1, which is as well a subset of the reachable states from s_0.*

calculation and could thus be expressed by a μ-calculus formula:

$$\alpha_{reach} = \mu x. \quad x \_ \alpha \tag{4.58}$$

This fixpoint calculation is initialized with $x = \mathsf{false}$ since x is guarded by μ. In each iteration the successor set of x as well as all states of α are added. Thus $x_0 = \mathsf{false}$, $x_1 = \alpha$, $x_2 = \alpha \_ \quad \alpha$, $x_3 = \alpha \_ \quad \alpha \_ \quad (\quad \alpha)$,.... If x_n is a fixpoint, x_n contains all states reachable from α and furthermore x_n is inductive. Given that it would not be inductive, x_n is not a fixpoint, since there is a transition from an x_n-state to a $: x_n$-state s and this state s should have been added.

Next, consider the insights provided by this example. There are four different ψ_i (see formula 4.54-4.57), which satisfy all three assertions of the safety inference rule. Thus each of those ψ_i is a proof that $K \models \mathsf{AG} : s_7$, since the inference rule is proven to be correct. However, with regard to the subset relation, those four intermediate lemmas form a lattice. This is because for any two ψ_i there is a greatest minimal and a least maximal element. Furthermore, this example provided some insight into the construction of multiple ψ_i by calculating the minimal inductive set of all initial states Ψ_{reach}. Furthermore Ψ_{reach} could be joined with any set ψ_{bonus}. A ψ_{bonus} is constructed by calculating the set of reachable states from an arbitrary state of the structure, provided that all states reachable by this state satisfy the desired safety property. Each combination of Ψ_{reach} with any amount of ψ_{bonus} provides an intermediate lemma, because each set constructed this way is inductive, contains all initial states and given that Ψ_{reach} does not contain states which harm the property (in which case it would be a counterexample), all states of this lemma are safe.

That the set of intermediate lemmas forms a lattice in this example might seem coincidental. However, I have proven that this is not the case and that the set of valid intermediate lemmas of greatest fixpoints, thus of intermediate lemmas which satisfy all assertions of the corresponding inference rule, form a lattice. This can be concluded from the following theorem:

Theorem 2 (Lattice property of inductive assertions)**.** *Assume Ψ_1 and Ψ_2 are inductive assertions to prove the greatest xpoint $\nu x.\Phi$, i.e., that the following proof rule can be used to prove the validity of $\Psi_l \; ! \; \nu x.\Phi$ for i 2 f1, 2g:*

$$\frac{\Psi_l \; ! \; \Psi_i \quad \Psi_i \; ! \; [\Phi]_x^{\Psi_i}}{\Psi_l \; ! \; \nu x.\Phi}$$

Then, also the conjunction $\Psi_1 \wedge \Psi_2$, as well as the disjunction $\Psi_1 \vee \Psi_2$, are inductive assertions that can be used for proving the validity of $\Psi_I \Rightarrow \nu x.\Phi$ by the above proof rule.

Proof. Assume Ψ_1 and Ψ_2 are inductive assertions to prove the greatest fixpoint $\nu x.\Phi$, thus this results in:

$$\begin{array}{ll} (1)\ \Psi_I \Rightarrow \Psi_1 & (2)\ \Psi_I \Rightarrow \Psi_2 \\ (3)\ \Psi_1 \Rightarrow [\Phi]_x^{\Psi_1} & (4)\ \Psi_2 \Rightarrow [\Phi]_x^{\Psi_2} \end{array}$$

Assumptions (3) and (4) mean that Ψ_1 and Ψ_2 are post-fixpoints of the state transformer of Φ w.r.t. x. By lemma 5 and lemma 6, we then conclude that also the conjunction $\Psi_1 \wedge \Psi_2$ as well as the disjunction $\Psi_1 \vee \Psi_2$ are post-fixpoints of the state transformer of Φ w.r.t. x so that for both formulas, the second assertion of the proof rule is valid.

Moreover, we also clearly conclude from (1) and (2) that $\Psi_I \Rightarrow \Psi_1 \wedge \Psi_2$ and therefore also that $\Psi_I \Rightarrow \Psi_1 \vee \Psi_2$ holds. Hence, also the first subgoals must be valid for the conjunction $\Psi_1 \wedge \Psi_2$ as well as for the disjunction $\Psi_1 \vee \Psi_2$. □

Note that this theorem is almost a direct consequence of the previously stated lemma 5 and lemma 6 which have been shown in the last chapter, which also proves the power those lemmas provide.

The fact that the inductive assertions for proving greatest fixpoints form a lattice is a core insight which this thesis provides. This allows to further reason about the construction of inductive assertions and furthermore shows dependencies between different inductive assertions for the same proof goal. Thus the fact that the inductive assertions form a lattice is a newly discovered, yet interesting property of the set of inductive assertions.

4.4. Non-lattice Assertions

So far we have discussed that the set of inductive assertions, which are the intermediate lemmas for greatest fixpoint proofs, form a lattice. This is because each pair of inductive assertions (Ψ_1, Ψ_2) has a least upper bound and a greatest lower bound that are inductive assertions as well. Thus there is an inductive assertion which contains precisely the meet of Ψ_1 and Ψ_2 as well as an inductive assertion which is the join of Ψ_1 and Ψ_2. Thus the set of states contained in both Ψ_1 and Ψ_2 is an inductive set which contains the initial states. Furthermore all states of Ψ_1 and Ψ_2 combined form an inductive set, which also contains all initial states.

However, this insight is not applicable in the case of least fixpoints. First, consider both the inference rules for greatest fixpoints:

$$\frac{\Psi_I \ ! \ \psi \quad \psi \ ! \ [\varphi]_x^{\psi}}{\Psi_I \ ! \ (\nu x.\varphi)} \tag{4.59}$$

and respectively for least fixpoints:

$$\frac{\psi_0 \ ! \ [\varphi]_x^{\mathsf{false}} \quad \bigwedge_{i=0}^{n-1} (\psi_i \ ! \ \psi_{i+1}) \quad \bigwedge_{i=0}^{n-1} \left(\psi_{i+1} \ ! \ [\varphi]_x^{\psi_i}\right) \quad \Psi_I \ ! \ \psi_n}{\Psi_I \ ! \ \mu x.\varphi} \tag{4.60}$$

Note that formula 4.59 asserts $\psi \ ! \ [\varphi]_x^{\psi}$ and thus asserts that ψ is a post-fixpoint. By lemma 6 we know that the post-fixpoints are closed under meet and join and thus they form a lattice. By that, we concluded that also the set of inductive assertions form a lattice structure. However, in the case of least fixpoints, neither the property that ψ is a post-fixpoint, nor the property that ψ is inductive are provided. However, in this case, a *sequence* of ψ_i is necessary. For example in the case of $\mathsf{EF}\beta$ this sequence could be an underapproximation of the states which reach a β-state in i steps. Note that this corresponds to the correct and complete inference rule for proving $\mathsf{EF}\beta$ shown in chapter 3.3:

$$\frac{\psi_0 \ ! \ \beta \quad \bigwedge_{i=0}^{n-1} (\psi_i \ ! \ \psi_{i+1}) \quad \bigwedge_{i=0}^{n-1} (\psi_{i+1} \ ! \ \beta \_ \quad \psi_i) \quad \Psi_I \ ! \ \psi_n}{\Psi_I \ ! \ \mathsf{EF}\beta} \tag{4.61}$$

By this rule, the sequence of ψ_i is a growing set of states, where each set is an underapproximation of the states which reach a β-state in i steps, as mentioned before. Furthermore by defining $\varphi := \alpha \_ \quad x$ in formula 4.60 this results in formula 4.61. Note that there has been a formula presented for proving a greatest fixpoint property which also used a sequence of inductive assertion, namely:

$$\frac{\Psi_I \ ! \ \psi_0 \quad \bigwedge_{i=0}^{k-1} \psi_i \ ! \ \psi_{i+1} \wedge \quad \psi_{i+1} \quad \psi_k \ ! \ \alpha \wedge \psi_k \ _1}{\Psi_I \ ! \ \mathsf{AG}\alpha} \tag{4.62}$$

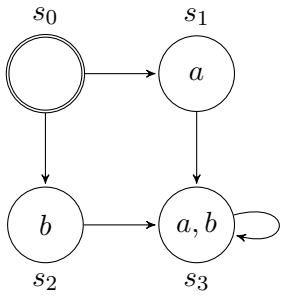

Figure 4.5.: *The structure K shown in this figure provides a counterexample as to why the intermediate lemmas of least fixpoint inference rules do not form a lattice structure in general.*

This correct and complete rule is the foundation of the proof procedure implemented in PDR. However, note that in this rule it is still enforced that any ψ_i of this sequence of state-sets is inductive. Furthermore, this single inductive ψ_i is sufficient and reduces the length of this sequence to exactly one ψ. If this is the case, inference rule 4.62 collapses to the already mentioned rule:

$$\frac{\Psi_I \ ! \ \psi \quad \psi \ ! \quad \psi \quad \psi \ ! \ \alpha}{\Psi_I \ ! \ \mathsf{AG}\alpha} \tag{4.63}$$

Thus a sequence of ψ_i is not necessary for proving a greatest fixpoint property, however, inductiveness is necessary. This is not the case when reasoning about least fixpoints since the sequence of assertions is necessary to provide some kind of ranking.

Consequently, it is fitting to consider intermediate lemmas of greatest fixpoint inference rules as single state-sets and thus proving that those state-sets form a lattice. However, this is not as trivial when considering least fixpoints since in this case, a sequence of sets or some other ranking mechanism is necessary.

Consider for example the inference rule 4.61 as well as the structure shown in figure 4.5. There are three possible sequences of assertions ψ_i to poof that $K \not\models \mathsf{EF}\beta$ with $\beta := (a \wedge b)$:

$$\mathtt{J}\psi_{1\ 0}\mathtt{K}_K := \mathtt{J}a \wedge b\mathtt{K}_K = \{s_3\} \tag{4.64}$$
$$\mathtt{J}\psi_{1\ 1}\mathtt{K}_K := \mathtt{J}a\mathtt{K}_K = \{s_1, s_3\} \tag{4.65}$$
$$\mathtt{J}\psi_{1\ 2}\mathtt{K}_K := \mathtt{J}a\ \_\ : b\mathtt{K}_K = \{s_0, s_1, s_3\} \tag{4.66}$$

$$\mathtt{J}\psi_{2\ 0}\mathtt{K}_K := \mathtt{J}a \wedge b\mathtt{K}_K = \{s_3\} \tag{4.67}$$
$$\mathtt{J}\psi_{2\ 1}\mathtt{K}_K := \mathtt{J}b\mathtt{K}_K = \{s_2, s_3\} \tag{4.68}$$
$$\mathtt{J}\psi_{2\ 2}\mathtt{K}_K := \mathtt{J}b\ \_\ : a\mathtt{K}_K = \{s_0, s_2, s_3\} \tag{4.69}$$

$$\mathtt{J}\psi_{3\ 0}\mathtt{K}_K := \mathtt{J}a \wedge b\mathtt{K}_K = \{s_3\} \tag{4.70}$$
$$\mathtt{J}\psi_{3\ 1}\mathtt{K}_K := \mathtt{J}a\ \_\ b\mathtt{K}_K = \{s_1, s_2, s_3\} \tag{4.71}$$
$$\mathtt{J}\psi_{3\ 2}\mathtt{K}_K := \mathtt{J}\mathtt{true}\mathtt{K}_K = \{s_0, s_1, s_2, s_3\} \tag{4.72}$$

First consider the sequence $\psi_{1\ i}$, $0 \quad i \quad n = 2$. Trivially $a \wedge b \ ! \ a \wedge b$ and thus $\psi_{1\ 0} \ ! \ \beta$ holds. Furthermore note that $\mathtt{J}\psi_{1\ 0}\mathtt{K}_K \quad \mathtt{J}\psi_{1\ 2}\mathtt{K}_K \quad \mathtt{J}\psi_{1\ 3}\mathtt{K}_K$ and thus $\bigwedge_{i=0}^{n\ 1}(\psi_i \ ! \ \psi_{i+1})$ also holds. In addition the only initial state s_0 is an element of $\mathtt{J}\psi_{1\ 2}\mathtt{K}_K$. It remains to make sure that $\bigwedge_{i=0}^{n\ 1}(\psi_{i+1} \ ! \ \beta\ \_ \quad \psi_i)$ holds. To this end, consider the predecessor sets:

$$\mathtt{J}\ (a \wedge b)\mathtt{K}_K = \{s_1, s_2, s_3\} \tag{4.73}$$
$$\mathtt{J}\ (a \wedge : b)\mathtt{K}_K = \{s_0\} \tag{4.74}$$
$$\mathtt{J}\ (: a \wedge b)\mathtt{K}_K = \{s_0\} \tag{4.75}$$
$$\mathtt{J}\ (: a \wedge : b)\mathtt{K}_K = ; \tag{4.76}$$

By that, we can conclude that $\psi_{1\ 1}$ contains only predecessor states of $\psi_{1\ 0}$ and $\psi_{1\ 2}$ contains only predecessor states of $\psi_{1\ 1}$ and thus $\psi_{1\ 0}$, $\psi_{1\ 1}$ and $\psi_{1\ 2}$ satisfy $\bigwedge_{i=0}^{n\ 1}\left(\psi_{1\ (i+1)} \ ! \ \beta\ \_ \quad \psi_{1\ i}\right)$. Therefore it is shown that the sequence $\psi_{1\ i}$, $0 \quad i \quad n = 2$, is indeed a proof that $K \models \mathsf{EF}(a \wedge b)$ by the inference rule 4.61. Both other sequences are analogue.

Given sequence $\psi_{1\ i}$ and $\psi_{2\ i}$, $0 \quad i \quad n = 2$, note that if each individual $\psi_{1\ j}$ is joined with $\psi_{2\ j}$ this results in $\psi_{3\ j}$. Thus the sequence $\psi_{3\ i}$ could be considered as the least upper bound of $\psi_{1\ i}$ and $\psi_{2\ i}$. However, there is no way to form a sequence which could be considered as the greatest lower bound. If for example each $\psi_{1\ j}$ is intersected with $\psi_{2\ j}$ this will result in:

$$\mathtt{J}\psi_{4\ 0}\mathtt{K}_K := \mathtt{J}a \wedge b\mathtt{K}_K = \{s_3\} \tag{4.77}$$
$$\mathtt{J}\psi_{4\ 1}\mathtt{K}_K := \mathtt{J}a \wedge b\mathtt{K}_K = \{s_3\} \tag{4.78}$$
$$\mathtt{J}\psi_{4\ 2}\mathtt{K}_K := \mathtt{J}a\ \$\ b\mathtt{K}_K = \{s_0, s_3\} \tag{4.79}$$

However, the sequence $\psi_{4\ i}$, $0 \quad i \quad n = 2$, is no valid proof for $K \models \mathsf{EF}(a \wedge b)$ by the inference rule 4.61, since s_0 is not a predecessor of s_3 and thus $\psi_{4\ 2} \ ! \ \beta\ \_ \quad \psi_{4\ 1}$ does not hold.

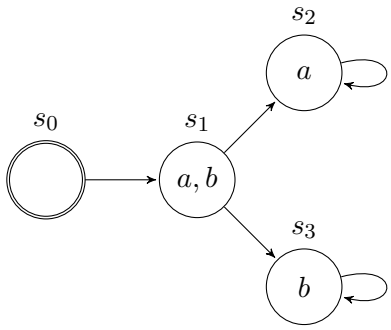

Figure 4.6.: *The structure K shown in this figure provides a second counterexample as to why the intermediate lemmas of least fixpoint inference rules do not form a lattice structure in general. The property which is considered is $K \not\models \mathsf{EF}a \quad b$. Note that it is sufficient to prove that there is a path from s_0 to either s_2 or s_3 in order to prove $K \not\models \mathsf{EF}a \quad b$.*

Another counterexample is shown in figure 4.6. Again there are three possible sequences of assertions which prove that $K \not\models \mathsf{EF}a \quad b$ holds:

$$\mathsf{J}\psi_1\ {}_0\mathsf{K}_K := \mathsf{J}a \wedge : b\mathsf{K}_K = \{s_2\} \tag{4.80}$$
$$\mathsf{J}\psi_1\ {}_1\mathsf{K}_K := \mathsf{J}a\mathsf{K}_K = \{s_1, s_2\} \tag{4.81}$$
$$\mathsf{J}\psi_1\ {}_2\mathsf{K}_K := \mathsf{J}a\ \_\ : b\mathsf{K}_K = \{s_0, s_1, s_2\} \tag{4.82}$$

$$\mathsf{J}\psi_2\ {}_0\mathsf{K}_K := \mathsf{J}b \wedge : a\mathsf{K}_K = \{s_3\} \tag{4.83}$$
$$\mathsf{J}\psi_2\ {}_1\mathsf{K}_K := \mathsf{J}b\mathsf{K}_K = \{s_1, s_3\} \tag{4.84}$$
$$\mathsf{J}\psi_2\ {}_2\mathsf{K}_K := \mathsf{J}b\ \_\ : a\mathsf{K}_K = \{s_0, s_1, s_3\} \tag{4.85}$$

$$\mathsf{J}\psi_3\ {}_0\mathsf{K}_K := \mathsf{J}a \quad b\mathsf{K}_K = \{s_2, s_3\} \tag{4.86}$$
$$\mathsf{J}\psi_3\ {}_1\mathsf{K}_K := \mathsf{J}a\ \_\ b\mathsf{K}_K = \{s_1, s_2, s_3\} \tag{4.87}$$
$$\mathsf{J}\psi_3\ {}_2\mathsf{K}_K := \mathsf{J}\mathsf{true}\mathsf{K}_K = \{s_0, s_1, s_2, s_3\} \tag{4.88}$$

What should be seen by this example, is that the states of the initial assertions $\mathsf{J}\psi_j\ {}_0\mathsf{K}_K$ are a subset of $\mathsf{J}a \quad b\mathsf{K}_K$. This is because it is sufficient to show that at least one state of $\mathsf{J}a \quad b\mathsf{K}_K$ can be reached from each initial state. However, there is no way of finding a *unique* minimal set of β-states, where each initial state has a path to one of those states. This is also shown by this example, since s_2 and s_3 are sufficient to prove $K \not\models \mathsf{EF}a \quad b$.[6]

Thus by those counterexamples, it is proven that the intermediate lemmas of least fixpoint inference rules do not necessarily form a lattice. Again this is an interesting property and further proves the differences between least

[6] At this point it should be noted that the inference rule for least fixpoints could be strengthened in a way such that there is only one sequence possible which satisfies this assertion rule. In this case the single possible intermediate lemma would also form a lattice, however, there is not much insight gained from that.

and greatest fixpoints. This raises the question, which properties are at least necessary to prove a greatest and a least fixpoint. While so far inductiveness was used to prove a greatest fixpoint, this is not applicable to least fixpoints and while least fixpoint proves needed some kind of ranking function this is not necessary for proving the greatest fixpoint. This insight again might help in finding a similar procedure to PDR for proving the properties of a least fixpoint in general.

4.5. Automatic Lemma Generation

The fact that the set of inductive assertions is a lattice while the set of intermediate lemmas for least fixpoints is no lattice allows to make quite significant conclusions. In this section, we will discuss one of those conclusions and concentrate on the greatest fixpoints and their corresponding inductive assertions.

As discussed so far we try to prove some greatest fixpoint property, like for example a safety property on a given Kripke structure. Note that any Kripke structure K which is considered consists of a finite set of states and thus is defined on a finite set of variables $\overline{x} = x_0, x_1, \ldots, x_{n\ 1}$. As discussed previously each state of the Kripke structure corresponds to an assignment to all variables $\overline{x}$, which is usually expressed by a cube[7] over $\overline{x}$. Note that each of those cubes corresponds to a single state and usually this cube is also the label of this state. We call these cubes over all variables $\mathsf{minterm}_i(\overline{x})$ and each $\mathsf{minterm}_i(\overline{x})$ corresponds to precisely one state of the structure. Note that there are n variables defined for the Kripke structure, thus this corresponds to 2^n states and correspondingly 2^n many $\mathsf{minterm}_i(\overline{x})$, $0 \quad i \quad n \quad 1$. Given 2^n new variables $\overline{m} = m_0, \ldots, m_{2^n\ 1}$ this allows to construct the following formula:

$$\Psi_{\mathsf{gen}}(\overline{m}, \overline{x}) = \bigvee_{i=0}^{2^n\ 1} m_i \wedge \mathsf{minterm}_i(\overline{x}) \tag{4.89}$$

Consider this formula 4.89. Note that there are 2^n states and each state is described by a $\mathsf{minterm}_i(\overline{x})$, $0 \quad i \quad n \quad 1$. Each of those $\mathsf{minterm}_i(\overline{x})$ has a corresponding m_i. Thus Ψ_{gen} describes a *template formula* and we call Ψ_{gen} a *generic intermediate lemma*. Note that each possible intermediate lemma $\Psi(\overline{x})$ corresponds to Ψ_{gen} with a certain assignment to all m_i. This is because each variable in $\overline{m}$ corresponds to a state, which is an assignment to *all* variables in $\overline{x}$. Furthermore, each assignment to $\overline{m}$ restricts which assignments to $\overline{x}$, thus which states, satisfy $\Psi_{gen}(\overline{m}, \overline{x})$. Therefore, we can conclude that a quantification over all formulas $\Psi(\overline{x})$ for example $\exists\Psi(\overline{x})$ is reduced to a quantification over all variables $\overline{m}$, thus $\exists\overline{m}.\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$. While this might not seem relevant, note that $\exists\overline{m}.\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$ corresponds to the SAT-problem of $\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$. Thus a second order quantification over propositional logic formulas is reduced to a SAT-problem by the introduction of $\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$.

Furthermore remember the inference rule for greatest fixpoints presented in formula 3.14:

$$\frac{\Psi_I \ ! \ \Psi \quad \Psi \ ! \quad \Psi \quad \Psi \ ! \ \Phi}{\Psi_{reach} \ ! \ \Phi}$$

and note that this inference rule is complete and correct. Thus if $\Psi_{reach} \ ! \ \Phi$ holds, since this rule is complete, we can conclude that there exists a corresponding Ψ which satisfies the three assertions $\Psi_I \ ! \ \Psi$, $\Psi \ ! \quad \Psi$ and $\Psi \ ! \ \Phi$.

[7] A cube is a conjunction of literals.

Thus since this inference rule is complete we can conclude that given the following formula is valid for a given safety property Φ:

$$\forall \overline{x}.\,(\Psi_{reach}(\overline{x}) \;\rightarrow\; \Phi(\overline{x})) \tag{4.90}$$

thus if all reachable states are Φ-states, then this also implies the following formula:

$$\exists \Psi(\overline{x}).\left(\begin{array}{l} \forall \overline{x}.\,(\Psi_I(\overline{x}) \;\rightarrow\; \Psi(\overline{x})) \wedge \\ \forall \overline{x}.\,(\Psi(\overline{x}) \;\rightarrow\; \Phi(\overline{x})) \wedge \\ \forall \overline{x}.\,(\Psi(\overline{x}) \;\rightarrow\; \forall \overline{x}'.\,(\Psi_R(\overline{x},\overline{x}') \;\rightarrow\; \Psi(\overline{x}'))) \end{array}\right) \tag{4.91}$$

Furthermore, the inverse also holds, since the inference rule is correct. Thus correctness and completeness of the Park induction rule can be equivalently expressed as the equivalence of those two quantified formulas. By that, we can conclude that both formulas 4.90 and 4.91 are equivalent.

Consider this formula 4.91 in more detail. This formula existentially quantifies over all possible $\Psi(\overline{x})$ and thus is a tautology if at least one $\Psi(\overline{x})$ exists which satisfies all three assertions. Each of the three assertions is universally quantified. Thus for all states which are initial states, they also have to be in the set described by $\Psi(\overline{x})$. For all other states it is not required to be in $\Psi(\overline{x})$ and thus the implication $(\forall \overline{x}.\Psi_I(\overline{x}) \;\rightarrow\; \Psi(\overline{x}))$. Next, all states in $\Psi(\overline{x})$ have to be $\Phi(\overline{x})$-states, thus states in which the desired property holds. The last assertion provides that from each state of $\Psi(\overline{x})$, each transition $\Psi_R(\overline{x},\overline{x}')$ leads to a state in $\Psi(\overline{x})$ again. Thus the last assertion provides that $\Psi(\overline{x})$ has to be inductive, $\Psi(\overline{x}) \;\rightarrow\; \Psi(\overline{x})$. Note that all variables which occur in this formula are quantified and thus this formula is either true or false for any given structure described by $\Psi_I(\overline{x})$ and $\Psi_R(\overline{x},\overline{x}')$. Thus this formula states whether all initial states $\Psi_I(\overline{x})$ satisfy $\mathsf{AG}\Phi(\overline{x})$ and thus for the corresponding structure K, if $K \models \mathsf{AG}\Phi(\overline{x})$.

Note that in formula 4.91 a quantification over all propositional logic formulas $\Psi(\overline{x})$ is necessary. However, as discussed recently, this could by reduced to a quantification over the variables $\overline{m}$. Thus we substitute $\Psi(\overline{x})$ by the new generic intermediate lemma $\Psi_{\mathsf{gen}}(\overline{m},\overline{x})$. This results in the following equivalent formula:

$$\exists \overline{m}.\left(\begin{array}{l} \forall \overline{x}.\,(\Psi_I(\overline{x}) \;\rightarrow\; \Psi_{\mathsf{gen}}(\overline{m},\overline{x})) \wedge \\ \forall \overline{x}.\,(\Psi_{\mathsf{gen}}(\overline{m},\overline{x}) \;\rightarrow\; \Phi(\overline{x})) \wedge \\ \forall \overline{x}.\,(\Psi_{\mathsf{gen}}(\overline{m},\overline{x}) \;\rightarrow\; \forall \overline{x}'.\,(\Psi_R(\overline{x},\overline{x}') \;\rightarrow\; \Psi_{\mathsf{gen}}(\overline{m},\overline{x}'))) \end{array}\right) \tag{4.92}$$

which is trivially equivalent by the means of propositional logic to:

$$\exists \overline{m}.\forall \overline{x}.\left(\begin{array}{l} (\Psi_I(\overline{x}) \;\rightarrow\; \Psi_{\mathsf{gen}}(\overline{m},\overline{x})) \wedge \\ (\Psi_{\mathsf{gen}}(\overline{m},\overline{x}) \;\rightarrow\; \Phi(\overline{x})) \wedge \\ (\Psi_{\mathsf{gen}}(\overline{m},\overline{x}) \;\rightarrow\; \forall \overline{x}'.\,(\Psi_R(\overline{x},\overline{x}') \;\rightarrow\; \Psi_{\mathsf{gen}}(\overline{m},\overline{x}'))) \end{array}\right) \tag{4.93}$$

By that, we eliminated the quantification over formulas and substituted it by the SAT-problem, whether there is an assignment to all $\overline{m}$ which satisfies

the formula. Note that again all variables which occur in this formula are quantified and correspondingly this formula has no free variables. Thus this formula again evaluates to true or to false for any given structure K. Consider this formula without the existential quantification over all $\overline{m}$:

Definition 4.5.1 (Formula of Inductive Assertions)**.** *For any property Φ and any transition system represented by Ψ_I and Ψ_R over variables $\overline{x}$, we construct the following formula $\Upsilon(\overline{m})$ of inductive assertions:*

$$\Upsilon(\overline{m}) := \forall \overline{x}. \left(\begin{array}{l} (\Psi_I(\overline{x}) \;\rightarrow\; \Psi_{gen}(\overline{m}, \overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;\rightarrow\; \Phi(\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;\rightarrow\; \forall \overline{x}^0. (\Psi_R(\overline{x}, \overline{x}^0) \;\rightarrow\; \Psi_{gen}(\overline{m}, \overline{x}^0))) \end{array} \right) \tag{4.94}$$

Consider this formula with the knowledge gained so far. First of all, note that $\Upsilon(\overline{m})$ is a formula over $\overline{m}$ since all other variables are quantified. Any assignment to all variables $\overline{m}$ yields that $\Psi_{gen}(\overline{m}, \overline{x})$ corresponds to a particular $\Psi(\overline{x})$. Furthermore, we know that the set of intermediate lemmas forms a lattice. Thus that there is a minimal state set, as well as a maximal state set since the lattice is finite and furthermore, each pair of state sets (Ψ_1, Ψ_2) has a $\mathsf{inf}(\Psi_1, \Psi_2)$ and a $\mathsf{sup}(\Psi_1, \Psi_2)$. And each m_i corresponds to a unique state s_i defined by $\mathsf{minterm}_i(\overline{x})$. Given this knowledge we can make some assumptions on $\Upsilon(\overline{m})$.

First we note that each satisfying assignment $\varphi(\overline{m})$ of $\Upsilon(\overline{m})$ implies the existence of an inductive invariant, namely $\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$ under this assignment $\varphi(\overline{m})$. Note that, by construction, this Ψ will contain all initial states, implies the property and is inductive. Furthermore, we know by the lattice property of inductive assertions, as well as the fact that each m_i corresponds to a state $s_i = \mathsf{minterm}_i(\overline{x})$, that there has to exist an assignment where as least as possible m_i are set to true. Furthermore, for any two satisfying assignments φ_1, φ_2 of $\Upsilon(\overline{m})$ also the infimum as well as the supremum have to be a satisfying assignment. This is because the set of inductive assertions is a lattice and each m_i corresponds to a state.

Definition 4.5.2 (Supremum and Infimum of Satisfying Assignments of the Formula of Inductive Assertions)**.** *Given that φ_1 and φ_2 are two satisfying assignments to $\Upsilon(\overline{m})$ for a given structure represented by Ψ_I and Ψ_R. The $\mathsf{inf}(\varphi_1, \varphi_2)$ exists and is defined by:*

$$\varphi_{\mathsf{inf}(\varphi_1,\varphi_2)} = (\varphi_1(m_i) \wedge \varphi_2(m_i)) \tag{4.95}$$

Thus $\varphi_{\mathsf{inf}(\varphi_1,\varphi_2)}$ assigns all variables m_i to true *which are* true *under φ_1 as well as φ_2. Correspondingly the $\mathsf{sup}(\varphi_1, \varphi_2)$ also exists and is defined by:*

$$\varphi_{\mathsf{sup}(\varphi_1,\varphi_2)} = (\varphi_1(m_i) \vee \varphi_2(m_i)) \tag{4.96}$$

Thus $\varphi_{\mathsf{sup}(\varphi_1,\varphi_2)}$ assigns all variables m_i to true which are true under either φ_1 or φ_2 or both.

This definition is a direct conclusion from the fact that the set of inductive assertions is a lattice. Furthermore, note that the strongest invariant

($\mathsf{StrongestInvar}(\overline{x})$) is the invariant generated by choosing as least as possible m_i to hold. We can conclude by the lattice properties, that all states which are described by the $\mathsf{StrongestInvar}(\overline{x})$ have to be contained in all other inductive assertions as well, since the strongest invariant is the least element of the lattice of the inductive assertions. Furthermore $\mathsf{StrongestInvar}(\overline{x})$ corresponds to the satisfying assignment of $\Upsilon(\overline{m})$ where the least amount of m_i is set to true. Accordingly the weakest invariant $\mathsf{WeakestInvar}(\overline{x})$ corresponds to the invariant where most m_i are set to true. Note that the weakest and the strongest invariant could be easily read from the BDD of $\Upsilon(\overline{m})$.

By this method I have shown two things: (1) This provides a method to calculate all possible invariants and especially the weakest and the strongest invariant. (2) By the substitution of Ψ with Ψ_{gen}, the existential quantifier is substituted by a SAT-problem. Note that this method is not efficient and thus not suitable for application, since Ψ_{gen} introduces a new variable for each state in the given structure. However, by this proposed method not only one arbitrary invariant is generated, but all possible invariants are calculated and thus this allows to further analyze proof procedures which are based on fixpoint induction.

5. Implementation

In chapter 4.5 a method has been proposed with which it is possible to generate all possible inductive assertions for any desired safety proof goal. This method was based on the correct and complete rule for inductive assertions:

$$\frac{\Psi_I \;!\; \psi \quad \psi \;!\; \quad \psi \quad \psi \;!\; \varphi}{\Psi_I \;!\; \mathsf{AG}\varphi} \tag{5.1}$$

This allowed to generate a formula Υ for any given structure defined by the set of variables, the initial states as well as the transition relation.

$$\Upsilon(\overline{m}) := \forall \overline{x}. \left(\begin{array}{l} (\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m}, \overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;!\; \varphi(\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;!\; \forall \overline{x}^{\prime}. (\Psi_R(\overline{x}, \overline{x}^{\prime}) \;!\; \Psi_{gen}(\overline{m}, \overline{x}^{\prime}))) \end{array} \right) \tag{5.2}$$

Any satisfying assignment of this formula Υ describes a potential inductive assertion and thus the set of all satisfying assignments corresponds to all inductive assertions. A more detailed explanation of why this is the case could be found in chapter 4.5.

A central task of this thesis was the implementation of this method in order to generate all inductive assertions for a given Kripke structure. To this end the averest framework[1] was provided. This framework has been implemented in F# and was developed by the Embedded Systems Group at the Technical University of Kaiserslautern. Furthermore, this framework provides a set of functions for specification, verification and implementation of reactive systems. Since the averest framework was implemented in F#, all implementations for this thesis have been done in F# as well.[2] Furthermore, only F# script files (`.fsx`-files) are used, which could be run in any F# interactive shell line by line.

The exact implementation can be found in the appendix of this thesis.

[1] More information could be found at http://www.averest.org/. Note that a compiled version could be acquired free of charge.

[2] For a gentle introduction to F# and its advantages visit https://fsharpforfunandprofit.com/. Note that F# is a functional programming language.

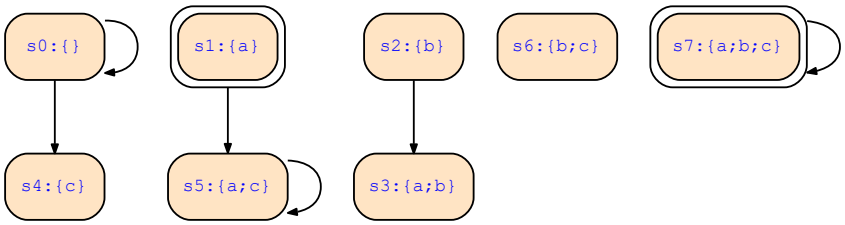

Figure 5.1.: *An example structure for which the implementation was tested.*

5.1. Lemma Generation for Existential Safety

The implementation is explained on an example structure K shown in figure 5.1. Note that this visualization is generated by the code as well. Furthermore, this structure is defined by:

$$\Psi_I = (a \wedge :b \wedge :c) \_ (a \wedge b \wedge c) \tag{5.3}$$

$$\Psi_R = \begin{pmatrix} :a \wedge b \wedge :c \wedge a^0 \wedge b^0 \wedge :c^0 \_ \\ a \wedge b \wedge c \wedge a^0 \wedge b^0 \wedge c^0 \_ \\ a \wedge :b \wedge :c \wedge a^0 \wedge :b^0 \wedge c^0 \_ \\ a \wedge :b \wedge c \wedge a^0 \wedge :b^0 \wedge c^0 \_ \\ :a \wedge :b \wedge :c \wedge :a^0 \wedge :b^0 \wedge :c^0 \_ \\ :a \wedge :b \wedge :c \wedge :a^0 \wedge :b^0 \wedge c^0 \end{pmatrix} \tag{5.4}$$

Consider the code of `LemmaGenerationEG.fsx` shown in listing A.1 in the appendix of this thesis. Note that line 1-12 contain includes for all provided libraries, as well as an include of `OutLib.fsx`. Furthermore the directory for the output is defined. The output of this function are `.pdf` files which show the constructed BDDs.

The included library `Averest.Core` provides the data structures of BDDs and propositional logic expressions, while `TeachingTools.Parser` allows to translate a given string into a propositional logic expression. The `Expressions` library provides functionality for handling those propositional logic expressions, like for example all boolean operators, while the `BDD` library provides functions for BDDs. Note that for the implementation of this method no SAT-solver is necessary and only BDD algorithms are used instead. While this may be a compromise in efficiency, note that efficiency was no goal for this implementation. Furthermore, BDDs are usually better readable by human readers, especially due to the fixed ordering of variables. The `OutLib` library has been implemented in the scope of this thesis as well and provides functions to format the output. The corresponding code is also attached in listing A.3.[3]

[3]The function for the generation of Kripke structures has been provided and was modified

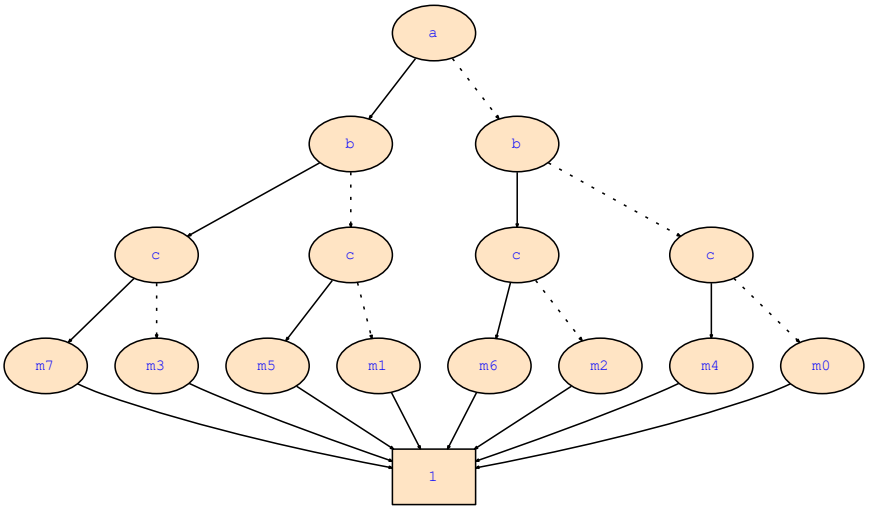

Figure 5.2.: *The template formula $\Psi_{\mathsf{gen}}(\overline{m}, \overline{x})$ with $\overline{x} := (a, b, c)$ as a BDD constructed by the function* `MkMintermFormula` *in line 66 of the code shown in listing A.1.*

In addition, regular expressions have been used to manipulate strings and thus formulas. This allows to implement a function which states that any occurring variable should hold in the next state, denoted as $[F]^{x_1^0, x_2^0, \ldots}_{x_1, x_2, \ldots}$. Note that there exist multiple ways to express this and there is no standardised notation for denoting properties which hold in the next step. While in this thesis for example $F^0 = a^0 \wedge b^0 \wedge c^0$ has been used, also notations like $F^0 = aX \wedge bX \wedge cX$ or $F^0 = next(a) \wedge next(b) \wedge next(c)$ express the same and are used in different implementations. However, in `OutLib` functions are provided to translate from one notation to another. Note that this can be a potential source of errors when using different libraries.

In lines 17-27 the example structure is provided by a string of the formula describing the initial states, as well as the transition relation. Furthermore, the set of defined variables is given and the desired safety property $\varphi := :(:a \wedge :b \wedge c) = :s_4$ is stated. Note that any propositional logic formula could be given, thus this formula does not have to be in DNF. Furthermore it could be implemented to read the structure from a text file or something similar, however, for the sake of this thesis, it was sufficient to provide the structure in the script-file itself.

In lines 32-41 all necessary variables are initialized. Thus all strings are translated to propositional logic expressions as well as a list `minL` of $\overline{m}$ variables is introduced. Note that there are 2^n 1 many new variables introduced with n being the length of the list of defined variables `varL`.

slightly.

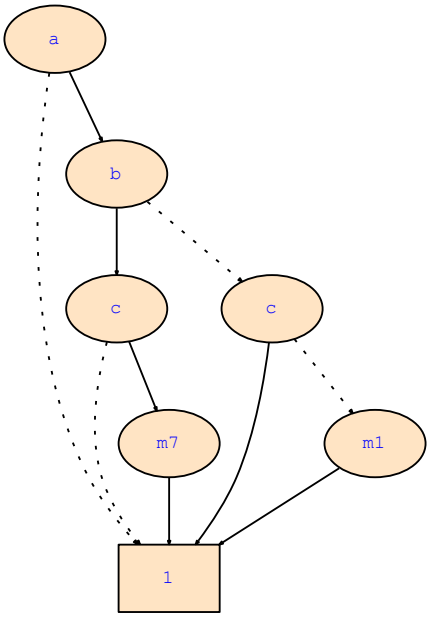

Figure 5.3.: *The corresponding BDD of the first assertion* `g1` $:=$ $\Psi_I(\overline{x})$! $\Psi_{gen}(\overline{m}, \overline{x})$. *Note that the only two initial states of the structure are* $s_1 = a \wedge \neg b \wedge \neg c$ *and* $s_7 = a \wedge b \wedge c$. *Those two assignments force* m_1 *and* m_7 *to hold as well.*

In lines 43-59 the function `MkMintermFormula` is shown. For a given list of variables `varList`, this function will generate the corresponding generic intermediate lemma $\Psi_{\mathsf{gen}}(\overline{m}, \texttt{varList})$. This function is called in line 66 and line 68 and the result of `let psi = MkMintermFormula (List.rev(varL))` is shown in figure 5.2. To remove clutter, all BDDs are zero suppressed, thus each missing edge in this BDD leads to the 0 leaf. Note that the result of `MkMintermFormula` corresponds to the definition of Ψ_{gen} in chapter 4.5, given $\overline{x} := (a, b, c)$:

$$\Psi_{\mathsf{gen}}(\overline{m}, \overline{x}) = \bigvee_{i=0}^{2^n \ 1} m_i \wedge \mathsf{minterm}_i(\overline{x}) \tag{5.5}$$

Next the sets $\overline{m}$, $\overline{x}$ and $\overline{x}^{\emptyset}$ are defined by lines 71-73 and those sets are necessary for quantification later on.

Note that the following equivalence holds:

$$\frac{\Psi_I \ ! \ \psi \quad \psi \ ! \quad \psi \quad \psi \ ! \ \varphi}{\Psi_I \ ! \ \mathsf{EG}\varphi} = \frac{\Psi_I \ ! \ \psi \quad \psi \ ! \quad \psi \wedge \varphi}{\Psi_I \ ! \ \mathsf{EG}\varphi} \tag{5.6}$$

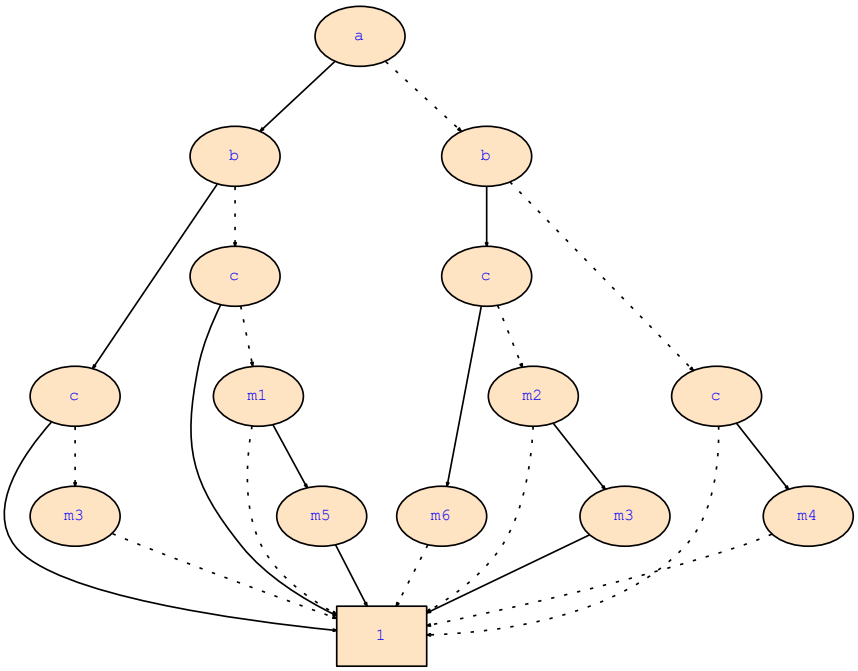

Figure 5.4.: *The corresponding BDD of the second assertion* `g2`.

And thus $\Upsilon(\overline{m})$ (defined in formula 5.2) could be equally defined as:

$$\forall \overline{x}. \left(\begin{array}{l} (\Psi_I(\overline{x}) \;\; ! \;\; \Psi_{gen}(\overline{m}, \overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;\; ! \;\; (\varphi(\overline{x}) \wedge \exists \overline{x}'. (\Psi_R(\overline{x}, \overline{x}') \wedge \Psi_{gen}(\overline{m}, \overline{x}')))) \end{array} \right) \tag{5.7}$$

This leaves two assertions for $\Psi_{gen}(\overline{m}, \overline{x})$, namely the upper and the lower part of formula 5.7. Those two assertions are called `g1` and `g2`:

$$\texttt{g1} = \Psi_I(\overline{x}) \;\; ! \;\; \Psi_{gen}(\overline{m}, \overline{x}) \tag{5.8}$$

$$\texttt{g2} = \Psi_{gen}(\overline{m}, \overline{x}) \;\; ! \;\; \left(\varphi(\overline{x}) \wedge \forall \overline{x}'. \left(\Psi_R(\overline{x}, \overline{x}') \wedge \Psi_{gen}(\overline{m}, \overline{x}')\right)\right) \tag{5.9}$$

Note that `g1` is defined in line 76 and the result is shown in figure 5.3. This formula `g1` asserts, that given an assignment to all variables corresponds to an initial state s_i, then m_i has to hold as well, in order for the formula to be satisfied.

On the other hand, `g2` is split up in multiple subgoals. First of all Ψ_{gen} is calculated in line 80. It is sufficient to substitute the existential quantifier in line 80 by a universal quantifier and change the conjunction to an implication, in order for this algorithm to calculate the invariants of $\mathsf{AG}\varphi$ instead of $\mathsf{EG}\varphi$.

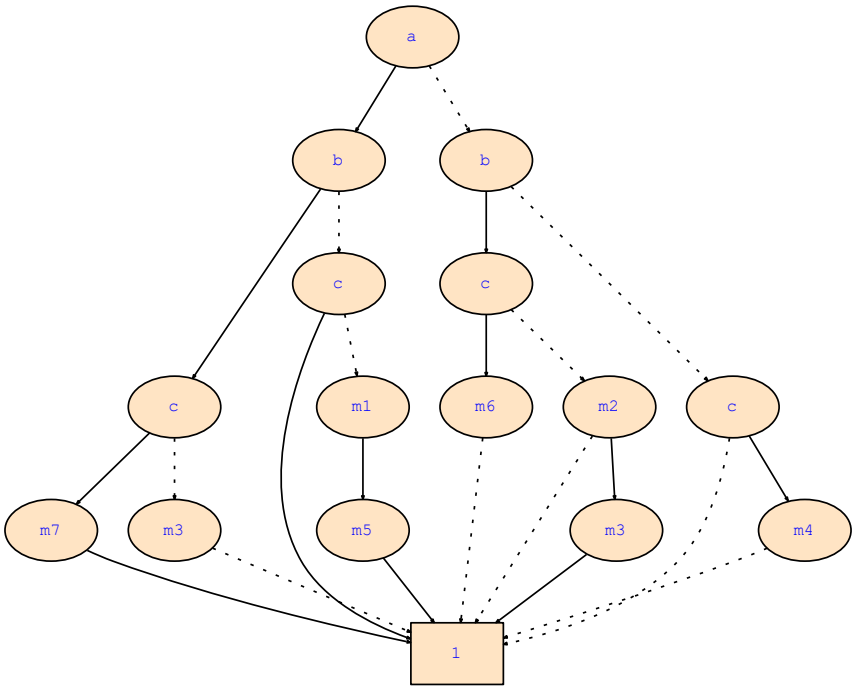

Figure 5.5.: *The corresponding BDD of the conjunction of* `g1` *and* `g2`*.*

This is the only change which has to be done and is also reflected in the proof rules:

$$\frac{\Psi_I \;!\; \psi \quad \psi \;!\; \psi \wedge \varphi}{\Psi_I \;!\; \mathsf{EG}\varphi} \qquad \frac{\Psi_I \;!\; \psi \quad \psi \;!\; \psi \wedge \varphi}{\Psi_I \;!\; \mathsf{AG}\varphi} \tag{5.10}$$

In the end `g2` is defined in line 88 and the corresponding BDD is shown in figure 5.4. Consider what this formula does: First of all $\Psi_{gen}(\overline{m}, \overline{x})$ is a template formula. Thus for an assignment to all variables $\overline{m}$, this formula corresponds to formula which describes a set of states. The goal is to find all assignments to $\overline{m}$ for which the resolving formula describes a set of states which is inductive, as well as a subset of the φ-states.

The conjunction of `g1` and `g2` results in a formula where the corresponding BDD is shown in figure 5.5. This formula is called `MintermAssertions` and is defined in line 91. There are some interesting properties which could be read from this BDD already. For example m_1 and m_5 depend on each other. This could also be seen in the structure (figure 5.1). The property which should be proven is $\mathsf{EG}\,\mathit{:}\,s_4$. Thus that each initial state of the structure has an infinite path starting in this state and on this path at any point $\mathit{:}\,s_4$ should hold. However in order for s_1 to have an infinite path, the state s_5 is also necessary. Therefore if a state set should be inductive and should contain s_1 (thus $\Psi_{gen}(\overline{m}, \overline{x})$ under the assignment that m_1 is set to true), then also s_5

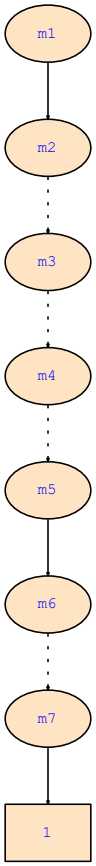

Figure 5.6.: *The corresponding BDD of the first assertion $\Upsilon(\overline{m})$ for the given structure K shown in figure 5.1. Note that m_0 does not appear in this BDD and thus could be chosen arbitrarily.*

needs to be contained (thus $\Psi_{gen}(\overline{m}, \overline{x})$ under the assignment that m_1 as well as m_5 is set to true). The formula `MintermAssertions` is thus so far defined as:

$$\left(\begin{array}{l} (\Psi_I(\overline{x}) \;\; ! \;\; \Psi_{gen}(\overline{m}, \overline{x})) \wedge \\ (\Psi_{gen}(\overline{m}, \overline{x}) \;\; ! \;\; (\varphi(\overline{x}) \wedge \exists \overline{x}^{0}.\, (\Psi_R(\overline{x}, \overline{x}^{0}) \wedge \Psi_{gen}(\overline{m}, \overline{x}^{0})))) \end{array} \right) \tag{5.11}$$

Therefore, this results in:

$$\Upsilon(\overline{m}) := \forall \overline{x}.\texttt{MintermAssertions} \tag{5.12}$$

Note that this is defined in line 94. Thus the `upsilon` defined by the code corresponds to $\Upsilon(\overline{m})$ defined in formula 5.2. Therefore, any assignment to all $\overline{m}$ for which this BDD (shown in figure 5.6) evaluates to true correspond to a set of states in the structure K. Each of these state sets contains all initial

state, is inductive and each state satisfies the desired safety property. Thus each satisfying assignment to all $\overline{m}$ corresponds to an intermediate lemma of a proof for a safety property, by the previously derived proof rules. Furthermore this BDD allows to read the weakest invariant $\mathsf{WeakestInvar}(\overline{x})$, as well as the strongest invariant $\mathsf{StrongestInvar}(\overline{x})$. The strongest invariant is precisely the reachable states and this could be read from the BDD by assigning as least as possible $\overline{m}$ to true. Note that the only variable where an assignment to false and true is possible and the BDD evaluates to true under each assignment is the variable m_0. Thus the strongest invariant corresponds to the assignment (given as a cube):

$$: m_0 \wedge m_1 \wedge : m_2 \wedge : m_3 \wedge : m_4 \wedge m_5 \wedge : m_6 \wedge m_7 \tag{5.13}$$

which corresponds to the set:

$$\mathsf{JStrongestInvar K}_K = \{s_1, s_5, s_7\} \tag{5.14}$$

Correspondingly the weakest invariant is described by the assignment:

$$m_0 \wedge m_1 \wedge : m_2 \wedge : m_3 \wedge : m_4 \wedge m_5 \wedge : m_6 \wedge m_7 \tag{5.15}$$

which in turn corresponds to the set:

$$\mathsf{JWeakestInvar K}_K = \{s_0, s_1, s_5, s_7\} \tag{5.16}$$

Note that both these sets $\mathsf{JStrongestInvar K}_K$ and $\mathsf{JWeakestInvar K}_K$ contain both initial states s_1 and s_7, trivially all states are $: s_4$-states and both sets are inductive. Furthermore s_0,s_1,s_5 and s_7 each have an infinite path starting in this state on which always $: s_4$ holds.

5.2. Extension to CTL

As discussed in section 3.3, there are additional correct and complete proof rules known and proven for further CTL operators. Furthermore, it is proven in [KS19], that given a proof rule for an E property, only the operator has to be substituted by an operator in order to achieve the proof rule of the corresponding A property. Note that these rules are shown in figure 3.10.[4]

Given the following proof rule discussed in section 3.3:

$$\frac{\Psi_I \;!\; \psi \quad \psi \;!\; \alpha \wedge \quad \psi}{\Psi_I \;!\; \mathsf{EG}\varphi} \tag{5.17}$$

As shown in the previous section 5.1, from this proof rule we can conclude the corresponding formula of inductive assertions $\Upsilon_{\mathsf{EG}}(\overline{m})$ defined as:

$$\Upsilon_{\mathsf{EG}}(\overline{m}) := \forall\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (\alpha \wedge \exists\overline{x}'.(\Psi_R(\overline{x},\overline{x}') \wedge \Psi_{gen}(\overline{m},\overline{x}'))))\end{array}\right) \tag{5.18}$$

Correspondingly, this can also be done with the remaining proof rules for greatest fixpoints. Thus there are proof rules known for $\mathsf{EG}\alpha$, $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$ and $\mathsf{E}[\alpha\ \mathsf{B}\ \beta]$. This is limited to the proof rules for greatest fixpoints, since it is required that the resulting set of inductive assertions forms a lattice.[5] While the formula of inductive assertions for $\mathsf{EG}\alpha$ has been shown in formula 5.18, both other formulas of inductive assertions are:

$$\Upsilon_{\mathsf{EU}}(\overline{m}) := \forall\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (\beta \_ \alpha \wedge \exists\overline{x}'.(\Psi_R(\overline{x},\overline{x}') \wedge \Psi_{gen}(\overline{m},\overline{x}'))))\end{array}\right) \tag{5.19}$$

$$\Upsilon_{\mathsf{EB}}(\overline{m}) := \forall\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (:\beta \wedge (\alpha \_ \exists\overline{x}'.(\Psi_R(\overline{x},\overline{x}') \wedge \Psi_{gen}(\overline{m},\overline{x}'))))))\end{array}\right) \tag{5.20}$$

Both those formulas are a direct consequence from the rules shown in [KS19]:

$$\frac{\Psi_I \;!\; \psi \quad \psi \;!\; \beta \_ \alpha \wedge \quad \psi}{\Psi_I \;!\; \mathsf{E}[\alpha\ \mathsf{U}\ \beta]} \qquad \frac{\Psi_I \;!\; \psi \quad \psi \;!\; :\beta \wedge (\alpha \_ \quad \psi)}{\Psi_I \;!\; \mathsf{E}[\alpha\ \mathsf{B}\ \beta]} \tag{5.21}$$

I have implemented both those rules as well and the code is shown in the listing A.2 in the appendix.[6] Note that only minor changes had to be done in the definition of the structure as well as the initialization.

[4] The corresponding proofs are shown in [KS19].

[5] This is discussed in detail in chapter 4.4 and 4.5.

[6] Disclaimer: The code shown in listing A.2 can not be executed as a whole. Either the computation of $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$ or the computation of $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$ has to be excluded. This is possible by defining the corresponding not desired computation as a comment or by executing only lines 1-67 as well as either lines 73-101 for a property $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$ or lines 107-140 for a property $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$ manually. Note that this could be wrapped in a function, however, this introduces new complexity which was not necessary for this thesis.

Furthermore, note that the following equations hold by the definition of the μ-calculus operators and :

$$\varphi = \mathcal{8}\overline{x}^{\emptyset}.\,(\Psi_R(\overline{x},\overline{x}^{\emptyset}) \;!\; \varphi^{\emptyset}) \tag{5.22}$$
$$\varphi = \mathcal{9}\overline{x}^{\emptyset}.\,(\Psi_R(\overline{x},\overline{x}^{\emptyset}) \wedge \varphi^{\emptyset}) \tag{5.23}$$

By this the following formulas for inductive assertions could be concluded:

$$\Upsilon_{\mathsf{AG}}(\overline{m}) := \mathcal{8}\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (\alpha \wedge \mathcal{8}\overline{x}^{\emptyset}.\,(\Psi_R(\overline{x},\overline{x}^{\emptyset}) \;!\; \Psi_{gen}(\overline{m},\overline{x}^{\emptyset}))))\end{array}\right) \tag{5.24}$$

$$\Upsilon_{\mathsf{AU}}(\overline{m}) := \mathcal{8}\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (\beta \_ \alpha \wedge \mathcal{8}\overline{x}^{\emptyset}.\,(\Psi_R(\overline{x},\overline{x}^{\emptyset}) \;!\; \Psi_{gen}(\overline{m},\overline{x}^{\emptyset}))))\end{array}\right) \tag{5.25}$$

$$\Upsilon_{\mathsf{AB}}(\overline{m}) := \mathcal{8}\overline{x}.\left(\begin{array}{l}(\Psi_I(\overline{x}) \;!\; \Psi_{gen}(\overline{m},\overline{x})) \wedge \\ (\Psi_{gen}(\overline{m},\overline{x}) \;!\; (:\beta \wedge (\alpha \_ \mathcal{8}\overline{x}^{\emptyset}.\,(\Psi_R(\overline{x},\overline{x}^{\emptyset}) \;!\; \Psi_{gen}(\overline{m},\overline{x}^{\emptyset})))))\end{array}\right) \tag{5.26}$$

To implement those formulas only `diaPsi` (line 80 in listing A.1 and lines 83 and 123 in listing A.2) has to be changed accordingly in each implementation.

Thus the provided implementation generates the formula of inductive assertions for $\mathsf{AG}\alpha$, $\mathsf{EG}\alpha$, $\mathsf{A}[\alpha\ \mathsf{U}\ \beta]$, $\mathsf{E}[\alpha\ \mathsf{U}\ \beta]$, $\mathsf{A}[\alpha\ \mathsf{B}\ \beta]$ and $\mathsf{E}[\alpha\ \mathsf{B}\ \beta]$.

6. Conclusion and Outlook

Since the introduction of PDR in 2011, this method was a game-changer. Even at the point of introducing this method already outperformed well-optimized model checkers in competitions like HWMCC'10 as stated in [Bra11].

However, this method is restricted to safety properties by construction (as discussed in chapter 3.2). But this is not the only restriction. The whole process of generalizing counterexamples is based on the assumption that unreachable states are split into regions so to say, thus share some properties. This specific assumption makes a lot of sense when structures are considered which are generated from code or modelled to represent some behaviour. Given for example a transition system derived from code. It can be assumed that states are unreachable because in some configuration the input set is limited and all unreachable states will share this configuration setting in and thus from some kind of region.

This leads to an interesting problem in verification. The efficiency of a method strongly depends on the used structures. Furthermore, it can be assumed that most structures which are theoretically possible do not find application at the moment and thus the considered structures are implicitly constrained. However, these constraints may vary depending on the field of application or even depend on the actual use-case. This impacts the efficiency of a procedure for the given set of structures to a significant amount. For example, the efficiency of PDR is substantially worse in the average case for randomly unconstrained structures. Then the assumptions ensuring PDR's efficiency are not necessarily met anymore. The process of verifying properties could be significantly improved by finding additional assumptions on the given structures, as aforementioned that unreachable states share certain properties by construction. Thus it seems promising to evaluate the structure and the properties of proofs in formal verification in order to find and reason about potential improvements of proof methods.

As already stated in chapter 3.2, PDR is a method for proving safety properties. These properties are usually defined in temporal logic by $\mathsf{AG}\alpha$[1]. However, CTL is only a subset of μ-calculus formulas. Thus μ-calculus includes temporal properties that cannot be expressed in CTL. Furthermore, the properties which can be expressed in CTL can not be negated easily. For example, note that $\neg\mathsf{AG}\alpha \not\equiv \mathsf{EF}\neg\alpha$. This is the case because $\mathsf{AG}\alpha$ holds if all infinite paths starting in all initial states satisfy α at any time. Thus $\mathsf{AG}\alpha$ is not satisfied as soon as one initial state has a path on which $\neg\alpha$ holds after a finite amount of time. However, $\mathsf{EF}\neg\alpha$ is satisfied if all initial states have a path on which $\neg\alpha$ holds at some point in time. Therefore the dual property of $\mathsf{AG}\alpha$ is reach-

[1] An introduction to μ-calculus and CTL is given in chapters 2.5 and 2.7.

ability and not $\mathsf{EF}:\alpha$. This was used by Bradley in the method presented in [Bra+11] to prove $\mathsf{AGF}\alpha$. To my knowledge currently no method provides a way to proof $\mathsf{EF}\alpha$ with an efficiency that resembles PDR.

To abstract from PDR and to reason about all CTL properties, a set of inference proof rules has been presented in early 2019 in [KS19]. By those rules, a set of states (or a sequence of state sets) has to satisfy some assertions in order to be a proof of some property.

In this thesis, I presented new exciting properties of those sets of states, which might lead to the discovery of further induction like proof procedures for further CTL properties. The most important property which has been presented in this thesis is the fact, that the set of intermediate lemma for greatest fixpoint inference proofs forms a lattice structure. This proof allows for making powerful assumptions. For example, it is proven that a lattice structure resembles an algebra (see [Gra09]). Thus everything that is known and proven for algebras can also be applied to the set of inductive assertions of greatest fixpoint induction proof procedures. Therefore, this information closes the gap to other different fields of research. In addition, this proves that there is a strongest and a weakest inductive assertion, while the weakest inductive assertion for safety properties corresponds to precisely the reachable states. This property is remarkable because it embeds the least fixpoint calculation of the reachable states into the greatest fixpoint calculation of the safety property.

In addition, I have shown that the lattice property does not hold for fixpoint induction proofs of least fixpoints. This provides further insight into the differences of least and greatest fixpoints. While in the case of greatest fixpoint, intermediate lemmas have to satisfy post-fixpoint assertions, the findings of this thesis show that this is embedded in the inductiveness of inductive assertions. However, there is no dual property known corresponding pre-fixpoints.

Furthermore, finding the lattice property allowed to introduce a method for calculating all inductive assertions. This has also been implemented in the scope of this thesis. A result of this implementation is a BDD where each satisfying assignment of this BDD corresponds to an inductive assertion. Thus if this BDD only consists of the zero leaf, it proves that the given Kripke structure does not satisfy the desired property. Therefore, this method not only provides all inductive assertions but indicates whether the desired property is not met.

The intention of this thesis was to provide a foundation for further research of proof methods in the field of verification of software and hardware. The newfound information connects the field of system verification to algebraic structures, topology and communication systems. This provides the potential, to apply the insights and methods studied in different fields to the search of intermediate lemmas. Furthermore, it is shown that the set of intermediate lemmas forms an algebra. This knowledge may allow to make further assumptions on the structure of intermediate lemmas and thus potentially increasing the efficiency of verification methods. Furthermore, the presented method of finding inductive assertions can be improved by providing additional methods for finding the weakest and the strongest inductive assertion in this BDD. This

is due to the fact that this BDD corresponds to a lattice structure and thus has to have certain properties. Those properties have not been researched so far in the context of system verification and may lead to an improvement or adaptation of SAT-solvers in regard to those special formulas. In addition, this thesis shows that the safety property is connected to reachability. It is even shown that the strongest invariant of safety properties corresponds to the reachable states of a structure, which is remarkable. However, this leads to the following question: What are the least necessary conditions for proving least fixpoints and greatest fixpoints and how are those two fixpoints connected? Answering this question might lead to the discovery of the successor of PDR.

Bibliography

[BHM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satis ability*. Vol. 185. IOS press, 2009.

[Boo19] Boolean Satisfiability Problem. *Boolean Satis ability Problem | Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2019]. 2019. URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[Boo75] George S Boolos. "On second-order logic". In: *The Journal of Philosophy* 72.16 (1975), pp. 509–527.

[Bra+11] Aaron R Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. "An incremental approach to model checking progress properties". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2011, pp. 144–153.

[Bra11] Aaron R Bradley. "SAT-based model checking without unrolling". In: *International Workshop on Veri cation, Model Checking, and Abstract Interpretation*. Springer. 2011, pp. 70–87.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2011, pp. 125–134.

[FDW13] Michael Fisher, Louise A Dennis, and Matthew P Webster. "Verifying autonomous systems." In: *Commun. ACM* 56.9 (2013), pp. 84–93.

[Fil19] Filterkonvergenz. *Filterkonvergenz | Wikipedia, The Free Encyclopedia*. [Online; accessed 12-September-2019]. 2019. URL: https://de.wikipedia.org/wiki/Filterkonvergenz.

[GA18] Felix Gruber and Matthias Althoff. "Anytime Safety Verification of Autonomous Vehicles". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2018, pp. 1708–1714.

[Gra09] George Gratzer. *Lattice theory: First concepts and distributive lattices*. Courier Corporation, 2009.

[Hig98] Nicholas J Higham. *Handbook of writing for the mathematical sciences*. Vol. 63. Siam, 1998.

[KS19] Martin Koehler and Klaus Schneider. "Inductive Proof Rules Beyond Safety Properties". In: *MBMV 2019; 22nd Workshop-Methods and Description Languages for Modelling and Veri cation of Circuits and Systems*. VDE. 2019, pp. 1–9.

[Lem19] Lemma (mathematics). *Lemma (mathematics) | Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2019]. 2019. URL: https://en.wikipedia.org/wiki/Lemma_(mathematics).

[LPP67] DC Luckham, DMR Park, and MS Paterson. *On Formalized Computer Programs, Preliminary draft, Programming Research Group*. 1967.

[LS16] Xian Li and Klaus Schneider. "Control-flow guided property directed reachability for imperative synchronous programs". In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2016, pp. 23–33.

[McM03] Kenneth L McMillan. "Interpolation and SAT-based model checking". In: *International Conference on Computer Aided Veri cation*. Springer. 2003, pp. 1–13.

[Par70] D. Park. "Fixpoint Induction and Proof of Program Semantics". In: *Machine Intelligence*. Ed. by B. Melzer and D. Michie. Vol. 5. Edinburgh University Press, 1970, pp. 59–78.

[Pen19] Pentium-FDIV-Bug. *Pentium-FDIV-Bug | Wikipedia, The Free Encyclopedia*. [Online; accessed 31-October-2019]. 2019. URL: https://de.wikipedia.org/wiki/Pentium-FDIV-Bug.

[Prä19] Prädikatenlogik zweiter Stufe. *Prädikatenlogik zweiter Stufe | Wikipedia, The Free Encyclopedia*. [Online; accessed 10-September-2019]. 2019. URL: https://de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik_zweiter_Stufe.

[Qui86] Willard V Quine. *Philosophy of logic*. Harvard University Press, 1986, pp. 63–68.

[Sch13] Klaus Schneider. *Veri cation of Reactive Systems: Formal Methods and Algorithms*. Springer Science & Business Media, 2013.

[Sch19a] Prof. Klaus Schneider. *Lecture notes in Model Based Design of Embedded Systems*. [Version of 15-August-2019]. 2019.

[Sch19b] Prof. Klaus Schneider. *Lecture notes in Veri cation of Reactive Systems*. [Version of 15-August-2019]. 2019.

[Tar+55] Alfred Tarski et al. "A lattice-theoretical fixpoint theorem and its applications." In: *Paci c journal of Mathematics* 5.2 (1955), pp. 285–309.

[Thr19] Three-valued logic. *Three-valued logic | Wikipedia, The Free Encyclopedia*. [Online; accessed 10-October-2019]. 2019. URL: https://en.wikipedia.org/wiki/Three-valued_logic.

A. My Code

Listing A.1: *LemmaGenerationEG.fsx*

```
#I __SOURCE_DIRECTORY__
#load "OutLib.fsx"

open Averest.Core
open TeachingTools.Parser
open Expressions
open BDD
open OutLib
open System.Text.RegularExpressions

let outDir = System.IO.Path.Combine (__SOURCE_DIRECTORY__, "output"
    + __SOURCE_FILE__.TrimEnd('f','s','x').TrimEnd('.'))

// ------------------------------------------------
// Structure
// ------------------------------------------------

let phiString = "!(!a&!b&c)"
let initCondString = "(a&!b&!c) | (a&b&c)"
let transRelString = "
    !a & b & !c & aX & bX & !cX |
    a & b & c & aX & bX & cX |
    a & !b & !c & aX & !bX & cX |
    a & !b & c & aX & !bX & cX |
    !a & !b & !c & !aX & !bX & !cX |
    !a & !b & !c & !aX & !bX & cX
 "
let vnL = ["a";"b";"c"]

// ------------------------------------------------
// Initialization
// ------------------------------------------------
let n = vnL.Length
let varL = List.map ParsePropLogicExpr vnL
let initCond = ParsePropLogicExpr initCondString
let transRel = ParsePropLogicExpr transRelString
let phi = ParsePropLogicExpr phiString
let phiX = ParsePropLogicExpr (Regex.Replace(phiString, (vnL |>
    String.concat "|"), (fun (m:Match) -> m.Value + "X")))
let nxtL = List.map (fun v-> ParsePropLogicExpr(v+"X")) vnL
let minL = List.map (fun i-> ParsePropLogicExpr("m"+(i.ToString())))
     [0..(pown 2 n)-1]
```

```
41 Initialize (List.rev(varL@nxtL@minL))

43 let MkMintermFormula (varList:BoolExpr list) =
44     let varL = List.rev(varList)
45     let n = varL.Length
46     let pn = pown 2 n
47     let rec mkMinterm varL i =
48         match varL with
49         | [] -> BoolConst true
50         | v::varL ->
51             let mt = mkMinterm varL (i/2)
52             if (i%2)=0 then MkBoolConj(BoolNeg(v),mt)
53             else MkBoolConj(v,mt)
54     let mkMt i =
55         let mv = ParsePropLogicExpr ("m"+(string i))
56         let mt = mkMinterm varL i
57         MkBoolConj(mv,mt)
58     let mtL = [ for i=0 to pn-1 do yield mkMt i ]
59     Averest.Core.Expressions.MkListDisj mtL

61 // ------------------------------------------------
62 // EG \phi
63 // ------------------------------------------------

65 // produce a generic induction lemma
66 let psi  = MkMintermFormula (List.rev(varL))
67 ShowBdd false outDir "1psi" (BoolExpr2Bdd psi)
68 let psiX = MkMintermFormula (List.rev(nxtL))

70 // variables in BDDs for quantification
71 let mVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(minL))
72 let qVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(
       varL@nxtL))
73 let qXVars = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj nxtL)

75 // g1 = I -> psi
76 let g1 = (BoolExpr2Bdd (BoolImpl(initCond,psi)))
77 ShowBdd false outDir "2g1" g1

79 // diaPsi = (<>psi)
80 let diaPsi = Exists qXVars (BoolExpr2Bdd (BoolConj(transRel, psiX)))
81 ShowBdd false outDir "3diaPsi"  diaPsi

83 // PhiAndDiaPsi = (phi & <>psi)
84 let PhiAndDiaPsi = (MkListConj [BoolExpr2Bdd phi ;diaPsi])
85 ShowBdd false outDir "4PhiAndDiaPsi"  PhiAndDiaPsi

87 // g2 = psi -> (phi & <>psi)
88 let g2 = MkListDisj [Negate(BoolExpr2Bdd psi);PhiAndDiaPsi]
89 ShowBdd false outDir "5g2"  g2
```

```
91 let MintermAssertions = (MkListConj [g1;g2])
92 ShowBdd false outDir "6MintermAssertions" MintermAssertions

94 let upsilon = Forall qVars MintermAssertions
95 ShowBdd false outDir "7Upsilon" upsilon

97 let ks = generateKS (Set (List.rev(vnL)): Set<string>) (
       transRelString: string) (initCond: BoolExpr) (BoolConst(true))
98 ShowKripke outDir "0KripkeStructure" ks
```

Listing A.2: *LemmaGenerationEUEB.fsx*

```
#I __SOURCE_DIRECTORY__
#load "OutLib.fsx"

open Averest.Core
open TeachingTools.Parser
open Expressions
open BDD
open OutLib
open System.Text.RegularExpressions

let outDir = System.IO.Path.Combine (__SOURCE_DIRECTORY__, "output"
    + __SOURCE_FILE__.TrimEnd('f','s','x').TrimEnd('.'))

// ------------------------------------------------
// Structure
// ------------------------------------------------

let alphaString = "b"
let betaString = "a"
let initCondString = "(a&!b&!c) | (a&b&c)"
let transRelString = "
    !a & b & !c & aX & bX & !cX |
    a & b & c & aX & bX & cX |
    a & !b & !c & aX & !bX & cX |
    a & !b & c & aX & !bX & cX |
    !a & !b & !c & !aX & !bX & !cX |
    !a & !b & !c & !aX & !bX & cX
 "
let vnL = ["a";"b";"c"]

// ------------------------------------------------
// Initialization
// ------------------------------------------------
let n = vnL.Length
let varL = List.map ParsePropLogicExpr vnL
let initCond = ParsePropLogicExpr initCondString
let transRel = ParsePropLogicExpr transRelString
let alpha = ParsePropLogicExpr alphaString
let alphaX = ParsePropLogicExpr (Regex.Replace(alphaString, (vnL |>
    String.concat "|"), (fun (m:Match) -> m.Value + "X")))
let beta = ParsePropLogicExpr betaString
let betaX = ParsePropLogicExpr (Regex.Replace(betaString, (vnL |>
    String.concat "|"), (fun (m:Match) -> m.Value + "X")))
let nxtL = List.map (fun v-> ParsePropLogicExpr(v+"X")) vnL
let minL = List.map (fun i-> ParsePropLogicExpr("m"+(i.ToString())))
     [0..(pown 2 n)-1]

Initialize (List.rev(varL@nxtL@minL));;

let MkMintermFormula (varList:BoolExpr list) =
    let varL = List.rev(varList)
```

```
48      let n = varL.Length
49      let pn = pown 2 n
50      let rec mkMinterm varL i =
51          match varL with
52          | [] -> BoolConst true
53          | v::varL ->
54              let mt = mkMinterm varL (i/2)
55              if (i%2)=0 then MkBoolConj(BoolNeg(v),mt)
56              else MkBoolConj(v,mt)
57      let mkMt i =
58          let mv = ParsePropLogicExpr ("m"+(string i))
59          let mt = mkMinterm varL i
60          MkBoolConj(mv,mt)
61      let mtL = [ for i=0 to pn-1 do yield mkMt i ]
62      Averest.Core.Expressions.MkListDisj mtL

64  // variables in BDDs for quantification
65  let mVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(minL))
66  let qVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(
        varL@nxtL))
67  let qXVars = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj nxtL)

69  // ------------------------------------------------
70  // E (alpha U beta)
71  // ------------------------------------------------

73  // produce a generic induction lemma
74  let psi  = MkMintermFormula (List.rev(varL))
75  ShowBdd false outDir "1psi" (BoolExpr2Bdd psi)
76  let psiX = MkMintermFormula (List.rev(nxtL))

78  // g1 = I -> psi
79  let g1 = (BoolExpr2Bdd (BoolImpl(initCond,psi)))
80  ShowBdd false outDir "2g1" g1

82  // diaPsi = (<>psi)
83  let diaPsi = Exists qXVars (BoolExpr2Bdd (BoolConj(transRel, psiX)))
84  ShowBdd false outDir "3diaPsi"  diaPsi

86  // AlphaAndDiaPsi = (alpha & <>psi)
87  let AlphaAndDiaPsi = (MkListConj [BoolExpr2Bdd alpha ;diaPsi])
88  ShowBdd false outDir "4AlphaAndDiaPsi"  AlphaAndDiaPsi

90  // g2 = psi -> (beta | alpha & <>psi) = !psi | beta | alpha & <>psi
91  let g2 = MkListDisj [Negate(BoolExpr2Bdd psi);AlphaAndDiaPsi ;
        BoolExpr2Bdd beta]
92  ShowBdd false outDir "5g2"  g2

94  let MintermAssertions = (MkListConj [g1;g2])
95  ShowBdd false outDir "6MintermAssertions" MintermAssertions

97  let upsilon = Forall qVars MintermAssertions
```

```
98  ShowBdd false outDir "7Upsilon" upsilon

100 let ks = generateKS (Set (List.rev(vnL)): Set<string>) (
        transRelString: string) (initCond: BoolExpr) (BoolConst(true))
101 ShowKripke outDir "0KripkeStructure" ks

103 // ------------------------------------------------
104 // E (alpha B beta)
105 // ------------------------------------------------

107 // produce a generic induction lemma
108 let psi  = MkMintermFormula (List.rev(varL))
109 ShowBdd false outDir "1psi" (BoolExpr2Bdd psi)
110 let psiX = MkMintermFormula (List.rev(nxtL))


113 // variables in BDDs for quantification
114 let mVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(minL))
115 let qVars  = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj(
        varL@nxtL))
116 let qXVars = BoolExpr2Bdd(Averest.Core.Expressions.MkListConj nxtL)

118 // g1 = I -> psi
119 let g1 = (BoolExpr2Bdd (BoolImpl(initCond,psi)))
120 ShowBdd false outDir "2g1" g1

122 // diaPsi = (<>psi)
123 let diaPsi = Exists qXVars (BoolExpr2Bdd (BoolConj(transRel, psiX)))
124 ShowBdd false outDir "3diaPsi"  diaPsi

126 // AlphaOrDiaPsi = (alpha | <>psi)
127 let AlphaOrDiaPsi = (MkListDisj [BoolExpr2Bdd alpha ;diaPsi])
128 ShowBdd false outDir "4AlphaOrDiaPsi"  AlphaOrDiaPsi

130 // g2 = psi -> (!beta & (alpha | <>psi)) = !psi | (!beta & (alpha |
        <>psi))
131 let g2 = MkListDisj [Negate(BoolExpr2Bdd psi); MkListConj [
        AlphaOrDiaPsi ;Negate(BoolExpr2Bdd beta)]]
132 ShowBdd false outDir "5g2"  g2

134 let MintermAssertions = (MkListConj [g1;g2])
135 ShowBdd false outDir "6MintermAssertions" MintermAssertions

137 let upsilon = Forall qVars MintermAssertions
138 ShowBdd false outDir "7Upsilon" upsilon

140 let ks = generateKS (Set (List.rev(vnL)): Set<string>) (
        transRelString: string) (initCond: BoolExpr) (BoolConst(true))
141 ShowKripke outDir "0KripkeStructure" ks
```

Listing A.3: *OutLib.fsx*

```
2  #I "averest/bin"
3  #r "Averest.Compilation.dll"
4  #r "Averest.Core.dll"
5  #r "TeachingTools.dll"
6  #r "OnExSy.Core.dll"
7  #r "OnExSy.Library.dll"


10 open System.Text.RegularExpressions
11 open OnExSy
12 open System.IO
13 open Averest.Core
14 open BDD
15 open TeachingTools.Automata
16 open Averest.Core.Expressions
17 open Averest.Core.Printer
18 open TeachingTools.KripkeStructures
19 open Averest.Core.Names
20 open TeachingTools.Global
21 open TeachingTools.Parser

23 let transRel2ToolFormat input = Regex.Replace(input, "([a-z]X)", fun
       (m:Match) -> "next(" + m.Value.TrimEnd('X') + ")")
24 let prefix1 = "s"
25 let outDir = ""

27 let execdot2pdfProcess (fileName:string) (directory:string) =
28    let pStartInfo = System.Diagnostics.ProcessStartInfo()
29    pStartInfo.CreateNoWindow <- true
30    pStartInfo.UseShellExecute <- false
31    let dot2pdfProcess = new System.Diagnostics.Process(StartInfo =
       pStartInfo)
32    dot2pdfProcess.StartInfo.FileName <- "dot"
33    dot2pdfProcess.StartInfo.Arguments <- (System.IO.Path.Combine(
       directory, fileName) + ".dot -Tpdf -o "+ System.IO.Path.Combine(
       directory, fileName) + ".pdf")
34    let started = dot2pdfProcess.Start()
35    dot2pdfProcess.WaitForExit()
36    System.IO.File.Delete (System.IO.Path.Combine(directory, fileName
       ) + ".dot")

38 let ShowBdd showLF dir fileName b =
39     System.IO.Directory.CreateDirectory dir |> ignore
40     let filename = System.IO.Path.Combine(dir, fileName) + ".dot"
41     let ostr = new StreamWriter(filename)
42     WriteBddList2DotFile showLF false [b] ostr
43     ostr.Close()
44     execdot2pdfProcess fileName dir

46 let ShowKripke dir fileName b =
```

```
47      System.IO.Directory.CreateDirectory dir |> ignore
48      let filename = System.IO.Path.Combine(dir, fileName) + ".dot"
49      let ostr = new StreamWriter(filename)
50      Kripke2Dot ostr "s" b
51      ostr.Close()
52      execdot2pdfProcess fileName dir

54  let ShowAll showLF dir fileName showNxt =
55      System.IO.Directory.CreateDirectory dir |> ignore
56      let filename = System.IO.Path.Combine(dir, fileName) + ".dot"
57      let ostr = new StreamWriter(filename)
58      WriteGlobalDotFile showLF showNxt ostr
59      ostr.Close()
60      execdot2pdfProcess fileName dir


63  let ShowAutonaton dir fileName b =
64      System.IO.Directory.CreateDirectory dir |> ignore
65      let filename = System.IO.Path.Combine(dir, fileName) + ".dot"
66      let ostr = new StreamWriter(filename)
67      Automaton2Dot ostr "s" string b
68      ostr.Close()
69      execdot2pdfProcess fileName dir

71  let ShowBddL showLF dir fileName bL =
72      System.IO.Directory.CreateDirectory dir |> ignore
73      let filename = System.IO.Path.Combine(dir, fileName) + ".dot"
74      let ostr = new StreamWriter(filename)
75      WriteBddList2DotFile showLF false bL ostr
76      ostr.Close()
77      execdot2pdfProcess fileName dir


80  // ------------------------------------------------
81  // Generate a Kripke Structure
82  // ------------------------------------------------

84  /// <summary>All occurrences of variables x are replaced with next(x
        ).</summary>
85  let rec old2new (e :BoolExpr) :BoolExpr =
86      match e with
87      | BoolConst _ -> e
88      | BoolVar c -> BoolNext e
89      | BoolNeg e1 -> BoolNeg (old2new e1)
90      | BoolConj (e1,e2) -> BoolConj ((old2new e1),(old2new e2))
91      | BoolDisj (e1,e2) -> BoolDisj ((old2new e1),(old2new e2))
92      | BoolImpl (e1,e2) -> BoolImpl ((old2new e1),(old2new e2))
93      | BoolEqu (e1,e2) -> BoolEqu ((old2new e1),(old2new e2))
94      | BoolIte (e1,e2,e3) -> BoolIte ((old2new e1),(old2new e2),(
        old2new e3))
95      | BoolNext c -> e
96      | _ -> raise (System.NotSupportedException("PropositionalLogic.
```

```
    boolOperatorCount : unexpected expression"))

/// <summary>Generate a Boolean formula for quantified variables.</
    summary>
let rec varConj (vars: QName List): BoolExpr =
   match vars with
   | a::aList ->
       BoolConj((BoolVar a), (varConj aList))
   | [] ->
       BoolConst true

// Check the implication of two BoolExpr
let checkBoolExpr (f1: BoolExpr) (f2: BoolExpr) :bool =
   let f1Implf2 = BoolImpl(f1,(BoolEqu(f2,BoolConst(true))))
   not(PropositionalLogic.isSatisfiable_AverestBDD (BoolNeg(f1Implf2
    )))

// Check whether the transition can take place or not
let checkTran (transBoolExpr: BoolExpr) (tran: BoolExpr) (node1: int
    ) (node2: int) : bool =
   let f = BoolImpl(tran,BoolEqu(transBoolExpr,BoolConst(true)))
   not(PropositionalLogic.isSatisfiable_AverestBDD (BoolNeg(f)))

// Generate the set of states: the powerset of the variables
let generateLable (varsSet: Set<string>) : Set<string> [] =
   let rec generateLabel (labelSet: string List) (returnSet: Set<
    string> List) =
      if labelSet.IsEmpty then returnSet
      else
         let addList = List.map(fun x -> Set.add labelSet.[0] x)
    returnSet
         generateLabel (labelSet.Tail) (List.append returnSet
    addList)
   let a : Set<string> = Set.ofList []
   List.toArray (generateLabel (Set.toList varsSet) [a])

// Encode the node as a BoolExpr
let mkBoolExpr (s1 : Set<string>) (st: bool) (varsSet : Set<string>)
    : BoolExpr =
   let s2 = Set.difference varsSet s1
   let mkStr1 (strL: string List) : string =
      if st then
         String.concat "&" strL
      else
         let s' = List.map(fun x -> "next(" + x + ")") strL
         String.concat "&" s'

   let mkStr2 (strL: string List) : string =
      if st then
         let s' = List.map(fun x -> "!" + x) strL
         String.concat "&" s'
      else
```

```
141             let s' = List.map(fun x -> "!next(" + x + ")") strL
142             String.concat "&" s'
143         let s1Str: string = mkStr1 (Set.toList s1)
144         let s2Str: string = mkStr2 (Set.toList s2)
145         let str =
146             if s1.IsEmpty then
147                 if s2.IsEmpty then "true"
148                 else
149                     s2Str
150             else
151                 if s2.IsEmpty then s1Str
152                 else
153                     s1Str + "&" + s2Str
154         ParsePropLogicExpr  str

156     // Generate the Kripke structure for the FSM
157     let generateKS (varsSet : Set<string>) (transBoolExprX: string) (
            initBoolExpr: BoolExpr) (stateBoolExpr: BoolExpr):
            KripkeStructure =
158         // Generate the labels
159         let label' : Set<string> [] = generateLable varsSet
160         let transBoolExpr = transRel2ToolFormat transBoolExprX |>
            ParsePropLogicExpr
161         // Computete a set of states according to the stateBoolExpr, e.g
            ., only the reachable states, or only the states that are not
            deadends.
162         let generateState : Set<int> =
163             let filterState s1 =
164                 let s2 = (mkBoolExpr s1 true varsSet)
165                 checkBoolExpr s2 stateBoolExpr
166             Set.ofArray (Array.filter(fun x -> (filterState label'.[x]))
            [|0..label'.Length-1|])
167         // Generate Initial states
168         let generateInit : Set<int> =
169             let filterInit s1 =
170                 let s2 = (mkBoolExpr s1 true varsSet)
171                 checkBoolExpr s2 initBoolExpr
172             Set.ofArray (Array.filter(fun x -> (generateState.Contains x)
            &&(filterInit label'.[x])) [|0..label'.Length-1|])
173         // Add transitions
174         let generateTrans : (int*int) list =
175             let tran (i:int) (j:int) : bool =
176                 let mkTranBoolExpr : BoolExpr =
177                     let source = mkBoolExpr label'.[i] true varsSet
178                     let target = mkBoolExpr label'.[j] false varsSet
179                     BoolConj(source,target)
180                 checkTran transBoolExpr mkTranBoolExpr i j
181             let checkTranBool (i:int) : (int*int) List =
182                 let targetList = List.filter(fun j -> (generateState.
            Contains j)&&(tran i j)) [0..label'.Length-1]
183                 List.map(fun x -> (i,x)) targetList
184             let trans (l: int list): (int*int) List =
```

```
185             List.collect (fun x -> (if (generateState.Contains x) then
        (checkTranBool x) else List.empty)) l
186           trans [0..label'.Length-1]
187       /// including those transitions leading to deadends
188       let ks : KripkeStructure=
189           {
190               vars = varsSet;
191               init = generateInit;
192               label = label';
193               trans = generateTrans;
194           }
195       ks
196   ;;
```