Temporal Verification with Transition Invariants

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes
von

Andrey Rybalchenko

Saarbrücken
2004

Tag des Kolloquiums:	1. Juni 2005
Dekan:	Prof. Dr. Jörg Eschmeier
Prüfungsausschuss:	Prof. Dr. Reinhard Wilhelm
	Prof. Dr. Andreas Podelski
	Prof. Dr. Amir Pnueli
	Prof. Dr. Bernd Finkbeiner
	Dr. Patrick Maier

Abstract

Program verification increases the degree of confidence that a program will perform correctly. Manual verification is an error-prone and tedious task. Its automation is highly desirable. The verification methodology reduces the reasoning about temporal properties of program computations to testing the validity of implication between auxiliary first-order assertions. The synthesis of such auxiliary assertions is the main challenge for automated tools. There already exist successful tools for the verification of *safety* properties. These properties require that some "bad" states never appear during program computations. The tools construct invariants, which are auxiliary assertions for safety. Invariants are computed symbolically by applying techniques of abstract interpretation. *Liveness* properties require that some "good" states will eventually appear in every computation. The synthesis of auxiliary assertions for the verification of liveness properties is the next challenge for automated verification tools.

This dissertation argues that *transition invariants* can provide a new basis for the development of automated methods for the verification of liveness properties. We support this thesis as follows. We introduce a new notion of auxiliary assertions called transition invariant. We apply this notion to propose a proof rule for the verification of liveness properties. We provide a viable approach for the automated synthesis of transition invariants by abstract interpretation, which automates the proof rule. For this purpose, we introduce a *transition predicate abstraction*. This abstraction does not have an inherent limitation to preserve only safety properties.

Most liveness properties of concurrent programs only hold under certain assumptions on non-deterministic choices made during program executions. These assumptions are known as fairness requirements. A direct treatment of fairness requirements in a proof rule is desirable. We specialize our proof rule for the direct accounting of two common ways of specifying fairness. Fairness requirements can be imposed either on program *transitions* or on sets of programs *states*. We treat both cases via *abstract-transition programs* and *labeled transition invariants* respectively.

We have developed a basis for the construction of automated tools that can not only prove that a program never does anything bad, but can also prove that the program eventually does something good. Such proofs increase our confidence that the program will perform correctly.

Kurzzusammenfassung

Programmverifikation stärkt unsere Überzeugung darin, dass ein Programm korrekt funktionieren wird. Manuelle Verifikation ist fehleranfällig und mühsam. Deren Automatisierung ist daher sehr erwünscht. Die allgemeine Vorgehensweise bei der Verifikation besteht darin, die temporale Argumentation über die Programmberechnungen auf die Überprüfung der Gültigkeit von Implikation zwischen Hilfsaussagen in Prädikatenlogik zu reduzieren. Die größte Herausforderung in der Automatisierung von Verifikationsmethoden liegt in der automatischen Synthese solcher Hilfsaussagen. Es gibt bereits erfolgreiche Werkzeuge für die automatische Verifikation von Safety-Eigenschaften. Diese Eigenschaften erfordern, dass keine „unerwünschten“ Programmzustände in Berechnungen auftreten. Die Werkzeuge synthetisieren Invarianten, die Hilfsaussagen für die Verifikation von Safety-Eigenschaften darstellen. Invarianten werden symbolisch, mit Hilfe von Techniken der abstrakten Interpretation berechnet. Liveness-Eigenschaften erfordern, dass bestimmte „gute“ Zustände irgendwann in jeder Berechnung vorkommen. Die Synthese von Hilfsaussagen für die Verifikation von Liveness-Eigenschaften ist die nächste Herausforderung für automatische Werkzeuge.

Diese Dissertation vertritt die Auffassung, dass *Transitionsinvarianten* (engl.: transition invariants) eine neu Basis für die Entwicklung automatischer Methoden für die Verifikation von Liveness-Eigenschaften bereitstellen können. Wir unterstützen diese These wie folgt. Wir führen einen neuen Typ von Hilfsaussagen ein, der als Transitionsinvariante bezeichnet wird. Wir benutzen Transitionsinvariante, um eine Beweisregel für die Verifikation von Liveness-Eigenschaften zu entwickeln. Wir stellen einen praktikablen Ansatz für die Synthese von Transitionsinvarianten basierend auf der abstrakten Interpretation vor und automatisieren dadurch die Beweisregel. Zu diesem Zweck führen wir eine *Transitionsprädikaten-Abstraktion* (engl.: transition predicate abstraction) ein. Diese Abstraktion ist nicht darauf beschränkt, nur Safety-Eigenschaften erhalten zu können.

Die meisten Liveness-Eigenschaften nebenläufiger Programme gelten nur unter bestimmten Annahmen bzgl. der nicht-deterministischen Wahl, die bei den Programmberechnungen getroffen wird. Diese Annahmen sind als Fairness-Anforderungen bekannt und deren direkte Berücksichtigung in einer Beweisregel ist wünschenswert. Wir spezialisieren unsere Beweisregel für die direkte Behandlung von zwei verbreiteten Arten von Fairness-Spezifikationen. Zum einem berücksichtigen wir die Fairness-Anforderungen an Programmübergänge durch abstrakte Transitionsprogramme (engl.: abstract-transition programs). Zum anderen werden die durch Zustandsmengen angegebenen Fairness-Anforderungen mit Hilfe von markierten Transitionsinvarianten (engl.: labeled transition invariants) behandelt.

Wir haben eine Basis für die Entwicklung automatischer Werkzeuge bereitgestellt, die beweisen können, dass ein Programm nicht schadet und dass das Programm etwas Gutes bewirkt. Solche Beweise stärken unsere Überzeugung darin, dass das Programm korrekt funktionieren wird.

Contents

List of Figures

Acknowledgements

I would like to gratefully acknowledge the time and attention that Andreas Podelski has been giving to me as a research adviser, collaborator, teacher, and colleague. Most of the results presented here are joint work with him. Andreas turned my work on temporal verification into an exciting experience. He has always been patient and generously supportive to any new ideas I have in mind. Thank you, Andreas!

I would like to thank Harald Ganzinger for giving me an opportunity to work his group. Unfortunately he passed away, but his wisdom, knowledge, and rigor remain to be an incredible source of inspiration.

I thank Neil Jones and Chin Soon Lee for their visit in Saarbrücken in September 2002, and a discussion that has lead to the work on transition invariants. Without that fortunate meeting this work would not be possible.

I would like to thank Amir Pnueli for encouraging comments, and his commitment to become a referee of this dissertation. I thank Patrick and Radhia Cousot for inviting me to give a talk in their group. Visiting the ENS was an unforgettable experience.

I thank Chin Soon Lee and Patrick Maier for valuable comments on a preliminary version of the dissertation.

I have been very lucky to work in a wonderful environment of Max-Planck-Institut für Informatik in Saarbrücken. I thank Werner Backes, Peter Baumgartner, Friedrich Eisenbrand, Bernd Finkbeiner, Brigitta Hansen, Thomas Hillenbrand, Chin Soon Lee, Patrick Maier, Mark Pichora, Virgile Prevosto, Stefan Ratschan, Viorica Sofronie-Stokkermans, Hans de Nivelle, Uwe Waldmann, and every other with whom I have communicated in the institute and outside.

It has been a great pleasure to interact with fellow students Malte Hübner, Carsten Ihlemann, Anne Proetzsch, Dalibor Topić, Ina Schäfer, Nassim Seghir, Silke Wagner, and Thomas Wies.

I thank Konstantin Korovin for interesting conversations during tee breaks and for providing access to his sugar and chocolate repository, Yevgeny Kazakov and Kirill Dmitriev for being wind surfing partners. I wish to thank Rachel for the wonderful years.

During my graduate studies I have been supported by the International Max Planck Research School for Computer Science, and the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38. I am grateful for their support.

Finally, I thank my parents. I dedicate this dissertation to my mother.

Introduction

Program verification increases the degree of confidence that a program will perform correctly. Manual verification is an error-prone and tedious task. Its automation is highly desirable. Transition invariants can provide a new basis for the development of automated methods for the verification of concurrent programs.

The methodology for the verification of temporal properties of concurrent programs is to reduce the reasoning about program computations (sequences of program states) to the first-order reasoning about auxiliary assertions. Invariants and variants are typical auxiliary assertions used for verification. Invariants are properties that hold for every reachable state of the program, *e.g.*, the value of some arithmetic expression over program variables is always positive. Variants indicate the progress that a computation makes towards some particular program states, *e.g.* ranking functions for proving termination. The methodology requires the user to supply auxiliary assertions. The construction of auxiliary assertions demands the user's experience, ingenuity, and understanding of the program. Once the necessary assertions are identified, the rest of the verification effort amounts to testing the validity of implication between assertions. Such tests are accomplished routinely by state-of-the-art tools. The main challenge for the automated verification tools is the synthesis of auxiliary assertions.

There already exist successful tools like SLAM [1], ASTRÉE [3], and BLAST [19] for the automated verification of particular temporal properties, which require the absence of "bad" states in each program computation. These properties are known as safety properties. The typical examples are the absence of division by zero, overflow, and out of bounds array access. The tools automatically synthesize invariants that imply the non-reachability of such "bad" states. This is achieved by symbolically computing an approximation of the set of reachable states, which is formalized in the abstract interpretation framework [10].

Thus, the next challenge for the automated tools is the synthesis of auxiliary assertions for the verification of the remaining temporal properties, which are known as liveness properties. Liveness properties require that some "good" states appear in every computation. A typical liveness property is program termination. For this property, all states that do not admit any further program steps (*i.e.* terminal states) are considered to be "good" ones. Another typical liveness property requires that every request (for some service) eventually succeeds. The verification of liveness properties requires synthesis of variants. A variant is a well-founded measure attached to the program states such that its value decreases after every program step, and is minimal for the "good" states.

Most liveness properties of concurrent programs only hold under certain assumptions on the non-deterministic choices made during program computations, *e.g.* eventual execution of an idling process or eventual, successful transmission over a lossy

communication channel. These assumptions are known as fairness requirements. They are not explicitly shown in the program text. The common way to express fairness requirements is to impose conditions on the occurrence of particular program transitions or states in computations. For example, we may require that every transition must be taken infinitely often during every infinite computation, or that it is not the case that the program stays in a particular location forever. Treatment of fairness requirements complicates verification, since several sets of "good" states that correspond to the fairness requirements must be considered. This requires the synthesis of more involved variants, *e.g.* variants that decrease only at particular states or after particular transitions.

Until this work, there were no similar tools for the automated verification of liveness properties, as we have for the verification of safety properties. In this dissertation, we propose transition invariants — a new kind of auxiliary assertions for the verification of liveness properties. Transition invariants have the potential for automated synthesis. One can apply the techniques of abstract interpretation to synthesize them. These techniques have facilitated the success of the tools for the verification of safety properties. In this dissertation, we show that the verification of liveness properties via transition invariants can be automated by abstract interpretation.

Contributions

This dissertation advances the state-of-the-art by proposing the notion of transition invariants for the automated verification of liveness properties. We summarize the main contributions as follows.

- We develop a new proof rule for the verification of liveness properties. The proof rule is based on transition invariants.
- We introduce two new notions: transition predicate abstraction and abstract-transition programs. We use these notions to propose an automated method for proving termination under fairness requirements.
- We introduce labeled transition invariants, which are an extension of transition invariants, for the direct accounting of fairness requirements imposed on program states, and develop a corresponding proof rule. We automate the proof rule via abstract interpretation.
- We propose an algorithm for the synthesis of linear ranking functions for 'single while' programs over linear arithmetic, which can be applied as a subroutine in our verification methods.

Next, we describe the contributions in more detail.

Transition Invariants A transition invariant is a superset of the transitive closure of the transition relation of the program. A transition invariant is disjunctively well-founded if it is a finite union of well-founded relations. We characterize the validity of liveness properties by the existence of disjunctively well-founded transition invariants. We formulate an inductiveness principle for transition invariants. This principle allows one to identify a given relation as a transition invariant. The disjunctive well-foundedness and the inductiveness principle provide the basis for our proof rule. We formalize a uniform setting by representing the fairness requirements and the temporal property in an abstract way, *i.e.* by sets of infinite sequences of program states.

Transition Predicate Abstraction We explore the automation of transition invariant-based proof rule via transition predicate abstraction. Transition predicates are binary relations over states. We introduce a notion of abstract-transition programs, which are built using transition predicates. Abstract-transition programs overcome the inherent limitation of abstract-state programs to safety properties. An abstract-transition program is a finite directed graph whose nodes are labeled by conjunctions of transition predicates, called abstract transitions, and whose edges are labeled by program transitions. We check whether a program terminates under fairness requirements by computing a corresponding abstract-transition program and considering its components in the following way. We reason about the termination of the subject program by testing the well-foundedness of the abstract transitions. We account for fairness requirements (both weak and strong fairness) that are imposed on program transitions by considering the edge labeling. We provide an algorithm for the automated construction of abstract-transition programs.

Labeled Transition Invariants Another common way to express fairness requirements (together with the transition-based fairness, which we address via abstract-transition programs) is to impose them on sets of states. We propose labeled transition invariants for a direct consideration of such fairness requirements. We extend transition invariants by sets of labels that correspond to the indices of fairness requirements. We account for the satisfaction of fairness requirements by keeping the indices of all possibly satisfied requirements in the labeling sets. We weaken the disjunctive well-foundedness criterion as follows. Let a finite union of relations be a transition invariant. Only those relations in the union need to be well-founded (to verify a liveness property) whose labeling sets contain the indices of all fairness requirements. We propose an inductiveness criterion for labeled transition invariants, and formulate a corresponding proof rule. The direct treatment of the state-based fairness allows us to handle specifications of liveness properties given by Büchi, generalized Büchi, and Streett automata in a uniform way. We automate the construction of labeled transition invariants via abstract interpretation.

Linear Ranking Functions We represent components of (labeled) transition invariants, and abstract transitions by ‘single while’ programs. These programs only contain (possibly non-deterministic) update statements in the loop body. Their termination proofs are required by the proposed verification methods. In the case of concurrent programs with linear arithmetic, we prove the termination of the corresponding ‘single while’ programs automatically. For this purpose, we propose an algorithm for the synthesis of linear ranking functions. We encode a linear ranking function as a solution to a system of linear inequalities derived from the while-condition and the update expressions of a ‘single while’ program.

Proof of Concept

We provide an experimental justification for the potential of automation of (labeled) transition invariants and abstract-transition programs. For this purpose we have built a prototype tool called ARMC-Live. All inductive (labeled) transition invariants and abstract-transition programs that we present in the following chapters have been synthesized by ARMC-Live.

In addition, the application of ARMC-Live ensures that the sets of (labeled) relations and abstract transitions that we present for the example programs actually form inductive (labeled) transition invariants and abstract-transition programs respectively. We also applied ARMC-Live to test the well-foundedness of (labeled) relations and abstract transition.

Outline and Sources

In the first chapter we introduce transition invariants and the corresponding proof rule in an abstract setting. We presented this material at LICS'2004 [38]. The second chapter describes a possible way of automating the introduced proof rule by applying transition predicate abstraction. We present this material at POPL'2005 [39]. In the third chapter we describe labeled transition invariants and the corresponding proof rule, which we presented at TACAS'2005 [35]. The algorithm for the synthesis of linear ranking functions is shown in the fourth chapter. We presented it at VMCAI'2004 [37]. The last two chapters discuss directions for future research and conclude the dissertation.

Chapter 0

Preliminaries

In this chapter, we formalize programs, review definitions for automata on infinite words, and the synchronous parallel composition of programs and automata; these notions are used in the rest of the dissertation.

Program P Following [33], we abstract away from the syntax of a concrete (concurrent) programming language and represent a program P by a *transition system*

$$P \;=\; \langle \Sigma, \Theta, \mathcal{T} \rangle$$

consisting of:

- Σ: a set of *states*,
- Θ: a set of *initial* states such that $\Theta \subseteq \Sigma$,
- $\mathcal{T}$: a finite set of *transitions* such that each transition $\tau \in \mathcal{T}$ is associated with a *transition relation* $\rho_\tau \subseteq \Sigma \times \Sigma$.

A *computation* σ is a maximal sequence of states $s_1, s_2, \ldots$ such that:

- s_1 is a *initial* state, *i.e.* $s_1 \in \Theta$,
- for each $i \geq 1$ there exists a transition $\tau \in \mathcal{T}$ such that s_i goes to s_{i+1} under ρ_τ, *i.e.* $(s_i, s_{i+1}) \in \rho_\tau$.

A finite segment $s_i, s_{i+1}, \ldots, s_j$ of a computation where $i < j$ is called a *computation segment*.

The set Acc of *accessible states* consists of all states that appear in some computations.

We introduce fairness requirements in the following chapters. We use different definitions of fairness requirements in different chapters, as explained in the introduction.

Programming language SPL We write example programs using the Simple Programming Language SPL of [33]. The translation from SPL and other (concurrent) programming languages into transition systems is standard.

We represent the transition relations ρ_τ by assertions over the unprimed and primed program variables. The distinguished variable π ranges over sets of locations of the

program. Each concurrent process has its own set of control locations. The value of π is a state denotes all location in which control currently stays. For each location label ℓ we define a predicate at_ℓ that holds if the current location of control is labeled by ℓ, *i.e.*, the predicate at_ℓ holds if the label ℓ is an element of π.

Automaton $\mathcal{A}$ Temporal properties can be abstractly represented as sets of infinite sequences of program states. Following the automata-theoretic framework for the verification of concurrent programs [51], we use automata on infinite words to represent such sequences. We refer to an automaton that represents the property of interest as *specification automaton*.

We consider an alphabet consisting of the program states Σ. The *automaton*

$$\mathcal{A} \;=\; \langle Q, Q^0, \Delta, F\rangle$$

with the Büchi acceptance condition consists of:

- Q: a (possibly infinite) set of states,
- Q^0: the set of *starting* states, such that $Q^0 \subseteq Q$,
- Δ: the *transition relation*. It is a set of triples $(q, s, q') \in Q \times \Sigma \times Q$.
- F: the set of accepting states, such that $F \subseteq Q$.

A *run* of the automaton $\mathcal{A}$ *on* the word $s_1, s_2, \ldots$ is a sequence of the automaton states $q_1, q_2, \ldots$ such that $q_1 \in Q^0$ and $(q_i, s_i, q_{i+1}) \in \Delta$ for all $i \geq 1$. The automaton *accepts* a word w if it has a run $q_1, q_2, \ldots$ on w such that for infinitely many i's we have $q_i \in F$.

Parallel Composition $P|||\mathcal{A}$ In the automata-theoretic framework, the verification of a temporal property amounts to a proof that there is no program computation that is accepted by the specification automaton (in fact, in the specification automaton we encode the set of all program computations that satisfy the fairness requirements and violate the property). We tie together a program P and a specification automaton $\mathcal{A}$ by taking their synchronous parallel composition $P|||\mathcal{A}$.

The program $P|||\mathcal{A}$, which in fact is equipped with the Büchi acceptance condition, is obtained by the synchronous parallel composition of P and $\mathcal{A}$. The set of states of $P|||\mathcal{A}$ is the Cartesian product

$$\Sigma_Q \;=\; \Sigma \times Q.$$

The set of starting states is $\Theta \times Q^0$. The transition relation of $P|||\mathcal{A}$ consists of pairs $((s, q), (s', q'))$ such that $(s, s') \in R$ and $(q, s, q') \in \Delta$. The set of accepting states is the product

$$\Sigma_F \;=\; \Sigma \times F.$$

A computation $(s_1, q_1), (s_2, q_2), \ldots$ of $P|||\mathcal{A}$ is *fair* if for infinitely many i's we have $(s_i, q_i) \in \Sigma_F$.

Chapter 1

Transition Invariants

1.1 Introduction

Temporal verification of concurrent programs is an active research topic; for entry points to the literature see e.g. [16, 24, 29, 32, 33, 34, 51]. In the unifying automata-theoretic framework of [51], a temporal proof is reduced to the proof of fair termination, which again can be done using deductive proof rules, e.g. [29]. The application of these proof rules requires the construction of auxiliary assertions. This construction is generally considered hard to automate, especially when ranking functions and well-founded (lexicographic) orderings are involved.

We propose a proof rule whose auxiliary assertions are *transition invariants*. We introduce the notion of a transition invariant as a binary relation over program states that contains the transitive closure of the transition relation of the program. We formulate an *inductiveness principle* for transition invariants. This principle allows us to identify a given relation as a transition invariant. We also introduce the notion of *disjunctive well-foundedness* as a property of relations. We characterize the validity of a liveness property by the existence of a disjunctively well-founded transition invariant. This is the basis of the soundness and relative completeness of the proof rule.

Applying our proof rule for verifying termination or another liveness property of the program amounts to the following steps: the automata-theoretic construction of a new program (the parallel composition of the original program and a Büchi automaton as in [51]), the inductive proof of the validity of the transition invariant for the new program, and, finally, the test of its disjunctive well-foundedness.

Using transition invariants, we account for the Büchi acceptance condition (and hence, for fairness) in a direct way, namely, by intersecting the transition invariant with a relation over the Büchi accepting states.

If the transition invariant is well-chosen, the test of disjunctive well-foundedness amounts to testing well-foundedness of transition relations of programs of a very particular form: each program is one while loop whose body is a simultaneous update statement. In the case of concurrent programs with linear-arithmetic expressions we obtain while loops for which efficient termination tests are already known (see [8, 37, 49] and Chapter 4).

The main contribution of our proof rule lies in its potential for automation. It is a starting point for the development of automated verification methods for temporal properties *beyond safety* of [concurrent] programs over infinite state spaces. As de-

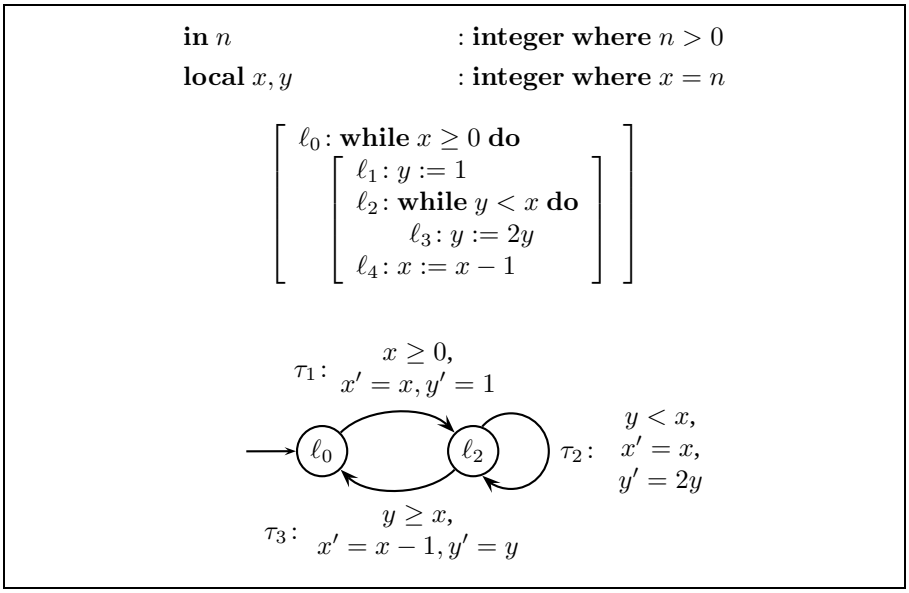

Figure 1.1: Program NESTED-LOOPS.

tailed in Section 1.5, the inductiveness principle allows one to compute the auxiliary assertions of the proof rule. Namely, the transition invariants can be automatically synthesized by computing abstractions of least fixed points of an operator over the domain of relations. Methods to do this correctly and efficiently are studied in the framework of abstract interpretation [10]. Such methods have helped to realize the potential of the inductive proof rules for (state) invariants [33] for the automation of the verification of safety properties [1, 3, 6, 10, 11, 18, 19]. We show a possible way for the realization of the analogous potential for transition invariants in Chapter 2.

Examples To simplify the presentation of the notion "transition invariants", in this chapter we ignore idling transitions for the presented concurrent programs. The depicted control-flow graphs treat each straight-line code segment as a single statement. For each of the example programs, we give a (non-inductive) transition invariant, along with an informal argument, in Sections 1.3 resp. 1.4; the corresponding formal argument is based on a stronger inductive transition invariant, which we present in Section 1.5.

NESTED-LOOPS Usually the termination argument for the program NESTED-LOOPS on Figure 1.1 is based on a lexicographic combination of well-founded orderings.

We observe that there are only two kinds of loops, those that go through ℓ_0 at least once and decrease the non-negative integer x, and those that go only through ℓ_2 (and not through ℓ_0) and decrease the non-negative value $x - y$. Transition invariants allow one to use this observation for a formal proof of termination.

CHOICE For the termination of the program CHOICE on Figure 1.2, we observe that the execution of any fixed sequence of transitions τ_1 or τ_2 decreases either of: x, y or

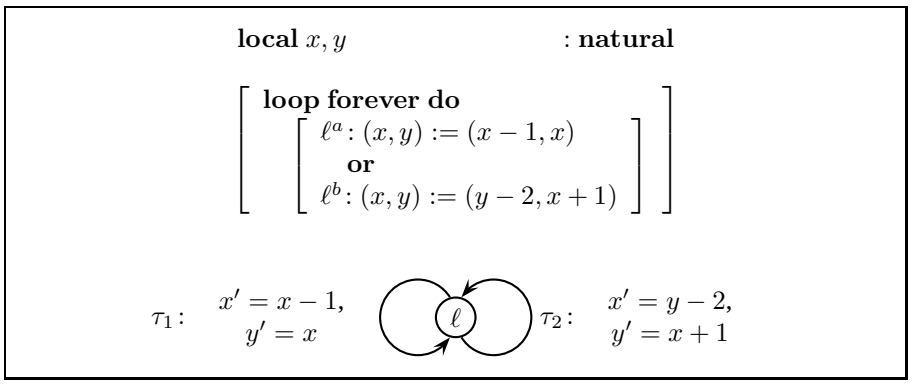

Figure 1.2: Program CHOICE.

$x + y$. Sections 1.2 and 1.3 show that this observation translates to a formal termination argument. Section 1.5 shows how one can formally justify this observation by an inductive proof.

ANY-DOWN The program ANY-DOWN on Figure 1.3 consists of two concurrent processes. Each of the processes can be scheduled to be executed by an external scheduler. The program is not terminating if we consider all possible scheduler behaviors. For example, in the following infinite computation of ANY-DOWN the process P_2 is never executed (a program state is a tuple containing the location of P_1, the location of P_2, the value of x, and the value of y).

$$\langle \ell_0, m_0, 1, 0 \rangle, \quad \langle \ell_1, m_0, 1, 0 \rangle, \quad \langle \ell_0, m_0, 1, 1 \rangle, \quad \ldots$$

This computation is not *fair* because the process P_2 is never executed although it is continually enabled. If we assume that the scheduling for each process is fair (see [29, 33] for a detailed treatment of fairness assumptions), then the program ANY-DOWN is terminating.

In Section 1.4 we show how we incorporate the fairness assumption into a termination proof.

CONC-WHILES A termination proof for the program CONC-WHILES on Figure 1.4 requires a more complicated fairness assumption (each of the processes must be scheduled infinitely often, hence it is not possible that a process waits forever).

Our formal proof in Section 1.4 will follow the intuition that each infinite fair computation decreases the value of x as well as the value of y infinitely often.

1.2 Transition Invariants

This section deals with properties of general binary relations. For concreteness we formulate the properties for the transition relation of a program and its restriction to the set of accessible states.

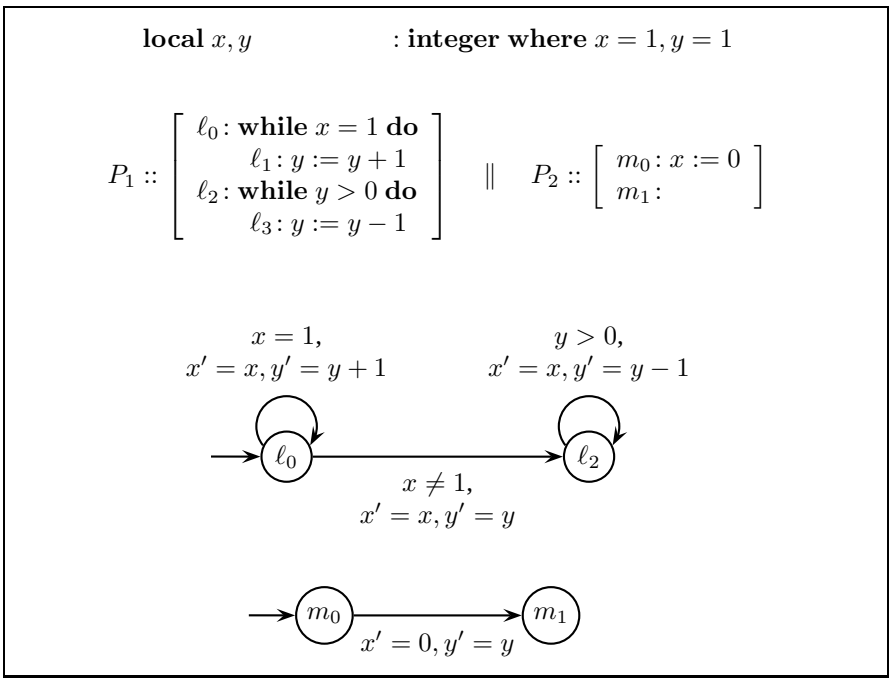

Figure 1.3: Program ANY-DOWN.

We fix a program $P = \langle \Sigma, \Theta, \mathcal{T} \rangle$. We define the transition relation R of the program P to be the union of the transition relations of all program transitions.

$$R \;=\; \bigcup_{\tau \in \mathcal{T}} \rho_\tau$$

Definition 1.1 (Transition Invariant) *A* transition invariant T *is a superset of the transitive closure of the transition relation* R *restricted to the accessible states* Acc*. Formally,*

$$R^+ \cap (Acc \times Acc) \;\subseteq\; T.$$

Thus, a transition invariant of the program is a relation T on the program states such that for every computation segment $s_i, s_{i+1}, \ldots, s_j$ the pair of states (s_i, s_j) is an element of T.

Note that the Cartesian product of the set of states with itself, *i.e.* the relation $\Sigma \times \Sigma$, is a transition invariant of the program. A superset of the transitive closure of the transition relation of the program is a transition invariant of the program; the converse does not hold.

A *state invariant* is a superset of the set of accessible states Acc. Given the transition invariant T and the set of starting states Θ, the set

$$\Theta \cup \{s' \mid s \in \Theta \text{ and } (s, s') \in T\}$$

is a state invariant. Conversely, a transition invariant can be strengthened by restricting it to a given state invariant.

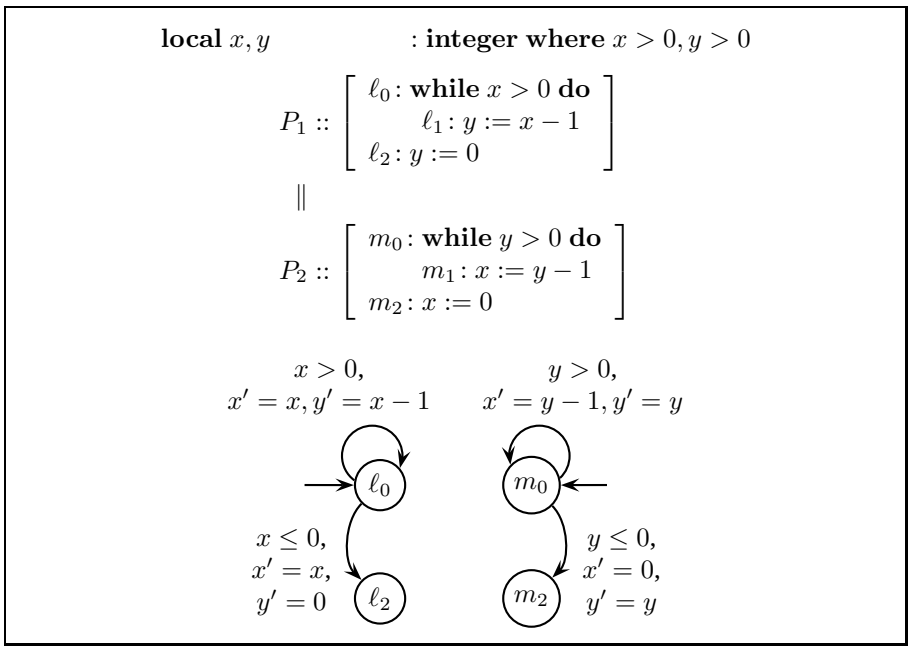

Figure 1.4: Program CONC-WHILES.

A program is *terminating* if it does not have infinite computations. This is equivalent to the fact that its transition relation restricted to the accessible states, *i.e.* $R \cap (Acc \times Acc)$, is well-founded. We investigate the well-foundedness of a transition relation through a weaker property of its transition invariant, introduced next.

Definition 1.2 (Disjunctive Well-foundedness) *A relation T is* disjunctively well-founded *if it is a finite union $T = T_1 \cup \cdots \cup T_n$ of well-founded relations.*

Every well-founded relation is disjunctively well-founded. The converse does not hold in the general case. For example, the relation ACK-REQ defined by

$$\{(ack, req)\} \cup \{(req, ack)\}$$

is disjunctively well-founded, but is not well-founded.

Given a disjunctively well-founded relation T, the implication:

$$R \text{ is well-founded if } R \subseteq T$$

does not hold (for a counterexample, take R and T to be the relation ACK-REQ). However, the implication:

$$R \text{ is well-founded if } R^+ \subseteq T$$

does hold, as we show below.

Theorem 1.1 (Termination) *The program P is terminating if and only if there exists a disjunctively well-founded transition invariant for P.*

Proof. **if**-direction: Assume, for a proof by contraposition, that

$$T \;=\; T_1 \cup \cdots \cup T_n$$

is a disjunctively well-founded transition invariant for the program P, and that P is not terminating. We show that at least one sub-relation T_i of the transition invariant is not well-founded.

By the assumption that P is not terminating, there exists an infinite computation $\sigma = s_1, s_2, \ldots$.

We choose a function f that maps an ordered pair of indices of the states in the computation σ to one of the sub-relations in the transition invariant T as follows.

$$\text{For } k < l,\ f(k,l) \;=\; T_i \text{ such that } (s_k, s_l) \in T_i$$

Such a function f exists because T is a transition invariant, and thus we can arbitrarily choose one relation from the (finite) set $\{T_i \mid (s_k, s_l) \in T_i\}$ as the image of the pair (k,l). Note that the range of the function f is finite.

For the fixed computation σ, the function f induces an equivalence relation $\sim$ on pairs of positive integers (in this proof we always consider pairs whose first element is smaller than the second one).

$$(k,l) \sim (k',l') \;=\; f(k,l) = f(k',l')$$

The equivalence relation $\sim$ has finite index, since the range of f is finite.

By Ramsey's theorem [41], there exists an infinite sequence of positive integers $K = k_1, k_2, \ldots$ such that all pairs of elements in K belong to the same equivalence class, say $[(m,n)]_\sim$ with $m, n \in K$. That is, for all $k, l \in K$ such that $k < l$ we have $(k,l) \sim (m,n)$. We fix m and n.

Let T_{mn} denote the relation $f(m,n)$. Since $(k_i, k_{i+1}) \sim (m,n)$ for all $i \geq 1$, the function f maps every pair (k_i, k_{i+1}) to T_{mn} for all $i \geq 1$. Hence, the infinite sequence $s_{k_1}, s_{k_2}, \ldots$ is induced by T_{mn}, *i.e.*,

$$(s_{k_i}, s_{k_{i+1}}) \in T_{mn},\ \text{for all } i \geq 1.$$

Hence, the sub-relation T_{mn} is not well-founded.

only if-direction: Assume that the program P is terminating. We define the relation T as the restriction of the transition relation to accessible states.

$$T \;=\; R^+ \cap (Acc \times Acc)$$

Clearly, T is a transition invariant. Assume that $\sigma = s^1, s^2, \ldots$ is an infinite sequence of states such that $(s^i, s^{i+1}) \in T$ for all $i \geq 1$. Since the state s^1 is accessible, and for all $i \geq 1$ there is a non-empty computation segment leading from s^i to s^{i+1} (*i.e.* $(s^i, s^{i+1}) \in R^+$), there exists an infinite computation $s_1, \ldots, s^1, \ldots, s^2, \ldots$. This fact is a contradiction to our assumption that P is terminating. Hence, T is (disjunctively) well-founded. □

The relation ACK-REQ shows that we cannot drop the requirement that not just the transition relation of a program, but also its transitive closure must be contained in the disjunctively well-founded relation T.

The next example shows that we cannot drop the finiteness requirement in the definition of disjunctive well-foundedness. The following transition relation

$$R \;=\; \{(i, i+1) \mid i \geq 1\}$$

has a transition invariant $T = T_1 \cup T_2 \cup \ldots$ that is the union of well-founded relations T_i, where

$$T_i \;=\; \{(i, i+j) \mid j \geq 1\}, \qquad \text{for all } i \geq 1.$$

However, the relation R is not well-founded.

1.3 Termination

Theorem 1.1 gives a (complete) characterization of program termination by disjunctively well-founded transition invariants.

We next present disjunctively well-founded transition invariants for the first resp. second program shown in the introduction to this chapter. Here, we only give informal arguments; in Section 1.5 we will show how one can formally prove that the relations are indeed transition invariants, and give the formal argument in the form of (stronger) inductive transition invariants.

NESTED-LOOPS The union of the relations T_1, T_2 and T_{ij} for $i \neq j \in \{0, \ldots, 4\}$ denoted by the following assertions over the unprimed and primed program variables is a transition invariant for the program NESTED-LOOPS.

$$\begin{array}{rcll} T_1 & = & x \geq 0 \wedge x' < x & \\ T_2 & = & x - y > 0 \wedge x' - y' < x - y & \\ T_{ij} & = & at\_\ell_i \wedge at'\_\ell_j & \text{where } i \neq j \in \{0, \ldots, 4\} \end{array}$$

The intuitive argument that the union of the relations above indeed identifies a transition invariant may go as follows. We can distinguish three kinds of computation segments that lead a state s to a state s'. All pairs of states (s, s') in R^+ such that s goes to s' via the location ℓ_0 (and in particular the loops at ℓ_0) are contained in the relation T_1. All pairs of states (s, s') in R^+ such that s goes to s' via the location ℓ_2 and not ℓ_0 (and in particular the loops at ℓ_2) are contained in the relation T_2. Every pair of states in R^+ that has different location labels is contained in one of T_{ij}'s.

Obviously, the relations T_1 and T_2 as well as the relations T_{ij}'s are well-founded.

CHOICE The union the relations below is a transition invariant for the program CHOICE.

$$\begin{array}{rcl} T_1 & = & x' < x \\ T_2 & = & x' + y' < x + y \\ T_3 & = & y' < y \end{array}$$

Again, the relations T_1, T_2, and T_3 are obviously well-founded.

1.4 Liveness

We follow the automata-theoretic framework for the temporal verification of concurrent programs [51]. This framework allows us to assume that the temporal correctness specification, viz. a liveness property Ψ and a fairness assumption Φ, are given by a (possibly infinite-state) automaton $\mathcal{A}_{\Phi,\Psi}$. The intuition is that the automaton $\mathcal{A}_{\Phi,\Psi}$ accepts exactly the infinite Φ-fair sequences of program states that do not satisfy the property Ψ. We assume that the automaton $\mathcal{A}_{\Phi,\Psi}$ is equipped with the Büchi acceptance condition.

The program P satisfies the liveness property Ψ under the fairness assumption Φ if there exists no infinite computation of P that satisfies the fairness condition Φ and falsifies the property Ψ, *i.e.,* all computations of the program P are rejected by the automaton $\mathcal{A}_{\Phi,\Psi}$ (computations are infinite words over the alphabet Σ; finite computations are added an idling transition for the last state). We export the program computations to the automaton by the synchronous parallel composition $P|||\mathcal{A}_{\Phi,\Psi}$ of the program and the automaton.

The program P is correct with respect to the property Ψ under the fairness condition Φ if and only if all (infinite) computations of $P|||\mathcal{A}_{\Phi,\Psi}$ are not fair (see Theorem 4.1 in [51]). The terminology '$P|||\mathcal{A}_{\Phi,\Psi}$ *is fair terminating*' is short for 'all (infinite) computations of $P|||\mathcal{A}_{\Phi,\Psi}$ are not fair'.

The following theorem characterizes the validity of the temporal property Ψ (under the fairness assumption Φ) through the existence of a disjunctively well-founded transition invariant for the program $P|||\mathcal{A}_{\Phi,\Psi}$ (with the set Σ_F of Büchi accepting states).

Theorem 1.2 (Liveness) *The program P satisfies the liveness property Ψ under the fairness assumption Φ if and only if there exists a transition invariant T for $P|||\mathcal{A}_{\Phi,\Psi}$ such that $T \cap (\Sigma_F \times \Sigma_F)$ is disjunctively well-founded.*

Proof. **if**-direction (sketch): Assume, for a proof by contraposition, that the finite union

$$T \;=\; T_1 \cup \cdots \cup T_n,$$

such that $T_i \cap (\Sigma_F \times \Sigma_F)$ is well-founded for all $i \in \{1, \ldots, n\}$, is a transition invariant for the program $P|||\mathcal{A}_{\Phi,\Psi}$. Furthermore, we assume that $P|||\mathcal{A}_{\Phi,\Psi}$ has an (infinite) fair computation (*i.e.,* is not fair terminating). We prove that at least one relation $T_i \cap (\Sigma_F \times \Sigma_F)$ is not well-founded.

By the assumption that $P|||\mathcal{A}_{\Phi,\Psi}$ is not fair terminating, there exists an infinite fair computation $\sigma = s_1, s_2, \ldots$. Let $\xi = s^1, s^2, \ldots$ be an infinite subsequence of σ such that $s^i \in \Sigma_F$ for all $i \geq 1$.

Now we can follow the lines of the **if**-part of the proof of Theorem 1.1. We show that there exists an infinite subsequence of ξ and an index $i \in \{1, \ldots, n\}$ such that each pair of consecutive states in the subsequence is an element of the very same relation $T_i \cap (\Sigma_F \times \Sigma_F)$. Thus we obtain a contradiction to the assumption that $T_i \cap (\Sigma_F \times \Sigma_F)$ is well-founded for all $i \in \{1, \ldots, n\}$.

only if-direction: Assume that the program $P|||\mathcal{A}_{\Phi,\Psi}$ is fair terminating (*i.e.,* has no (infinite) fair computation). Let Acc denote the set of accessible states of $P|||\mathcal{A}_{\Phi,\Psi}$.

We define the following relations on the accessible states of $P|||\mathcal{A}_{\Phi,\Psi}$.

$$\begin{aligned} T_1 &= R^+ \cap (Acc \cap \Sigma_F \times Acc) \\ T_2 &= R^+ \cap (Acc \setminus \Sigma_F \times Acc) \end{aligned}$$

Clearly, the relation

$$T = T_1 \cup T_2$$

is a transition invariant. Assume that $\sigma = s^1, s^2, \ldots$ is an infinite sequence of states such that $(s^i, s^{i+1}) \in T_1$ for all $i \geq 1$. Since the state s^1 is accessible, and for all $i \geq 1$ there is a non-empty computation segment leading from s^i to s^{i+1} (*i.e.* $(s^i, s^{i+1}) \in R^+$) there exists an infinite fair computation $s_1, \ldots, s^1, \ldots, s^2, \ldots$. This fact is a contradiction to our assumption that P is fair terminating. Hence, T_1 is well-founded. Clearly, the intersection $T_2 \cap (\Sigma_F \times \Sigma_F)$ is empty. We conclude that the **only-if** direction holds. □

Examples We give a transition invariant for each of the programs $P|||\mathcal{A}_{\Phi,\Psi}$ obtained by the parallel composition of the program ANY-DOWN resp. CONC-WHILES with the Büchi automaton $\mathcal{A}_{\Phi,\Psi}$ that encodes the appropriate fairness assumption Φ (the liveness property Ψ is termination; the automaton $\mathcal{A}_{\Phi,\Psi}$ accepts exactly the infinite Φ-fair computations). We do not explicitly present $\mathcal{A}_{\Phi,\Psi}$ and $P|||\mathcal{A}_{\Phi,\Psi}$ since they can be easily derived.

ANY-DOWN Here, the Büchi automaton $\mathcal{A}_{\Phi,\Psi}$ encodes the fairness assumption "eventually the process P_2 leaves the location m_0" which is expressed by the temporal logic formula $\Phi = \mathrm{F}(\neg at\_m_0)$. The union of the relations below forms a transition invariant for $P|||\mathcal{A}_{\Phi,\Psi}$. The predicates at_ℓ, at_m, and at_q describe the current location labels of the processes and the Büchi automaton. The predicate at_q_F holds if the Büchi automaton is in its accepting location.

$$\begin{aligned} T_1 &= at\_q_F \wedge y > 0 \wedge y' < y \\ T_2 &= \neg at\_q_F \\ T_3 &= at\_q_0 \wedge at'\_q_F \\ T_4 &= at\_m_0 \wedge at'\_m_1 \\ T_{ij} &= at\_\ell_i \wedge at'\_\ell_j \qquad \text{where } i \neq j \in \{0, \ldots, 3\} \end{aligned}$$

The relation T_1 contains the pairs of states $((s, q), (s', q'))$ from the transitive closure R^+ of the program $P|||\mathcal{A}_{\Phi,\Psi}$ that are the initial and the final states of the loops starting in the Büchi accepting state. These loops are induced by the execution of the **while**-statement at the location ℓ_2. For the **while**-statement at the location ℓ_0 the initial-final state pairs are elements of T_2. The relations T_3, T_4, and T_{ij} where $i \neq j \in \{0, \ldots, 3\}$ contain pairs of states that have different location labels wrt. either the Büchi automaton or one of the processes.

The relations T_1, T_3, T_4, and T_{ij}'s are well-founded. According to the formal argument of this section, the relation T_2 is not considered; the restriction of T_2 to the Büchi accepting states is empty.

CONC-WHILES We encode the fairness assumption that no process can wait forever (except in the final location) by the temporal formula below.

$$\begin{array}{c}\mathrm{GF}(\neg at\_\ell_0) \wedge \mathrm{GF}(\neg at\_\ell_1) \wedge \\ \mathrm{GF}(\neg at\_m_0) \wedge \mathrm{GF}(\neg at\_m_1)\end{array}$$

The corresponding Büchi automaton has the four states $\{q_0, q_1, q_2, q_F\}$, where the state q_F is accepting.

The union of the following relations is a transition invariant for $P|||\mathcal{A}_{\Phi,\Psi}$.

$$\begin{array}{rcll} T_1 & = & at\_q_F \wedge x > 0 \wedge x' < x & \\ T_2 & = & at\_q_F \wedge y > 0 \wedge y' < y & \\ T_3 & = & \neg at\_q_F & \\ T^4_{ij} & = & at\_q_i \wedge at'\_q_j & \text{where } i \neq j \in \{0,1,2\} \\ T^5_{ij} & = & at\_\ell_i \wedge at'\_\ell_j & \text{where } i \neq j \in \{0,1,2\} \\ T^6_{ij} & = & at\_m_i \wedge at'\_m_j & \text{where } i \neq j \in \{0,1,2\} \end{array}$$

The relations T_1 and T_2 capture loops that start in the Büchi accepting state and contain execution steps of both processes P_1 and P_2. The loops that contain the executions of only P_1 or only P_2 are captured by the relation T_3. The relations T^4_{ij}, T^5_{ij}, and T^6_{ij} with $i \neq j \in \{0,1,2\}$ capture computation segments that are not loops wrt. the location labels of either the Büchi automaton or one of the processes.

The well-foundedness of the relations T_1, T_2, T^4_{ij}, T^5_{ij}, and T^6_{ij} for $i \neq j \in \{0,1,2\}$ is sufficient for proving the fair termination property; the restriction of T_3 to the Büchi accepting state is empty.

1.5 Inductiveness

In this section, we formulate a proof rule for verifying liveness properties of concurrent programs. The proof rule is based on inductive transition invariants.

Definition 1.3 (Inductive Relation) *Given a program with the transition relation R, a binary relation T on program states is* inductive *if it contains the transition relation R and it is closed under the relational composition with R. Formally,*

$$R \ \cup \ T \circ R \ \subseteq \ T.$$

As usual, the *composition operator* $\circ$ denotes the relational composition, *i.e.*, for $P, Q \subseteq \Sigma \times \Sigma$ we have

$$P \circ Q \ \ = \ \ \{(s,s') \mid (s,s'') \in P \text{ and } (s'',s') \in Q\}.$$

Replacing the inductiveness criterion above by $R \cup R \circ T \subseteq T$ yields an equivalent criterion. Replacing it by $R \cap (Acc \times Acc) \cup T \circ R \cap (Acc \times Acc) \subseteq T$ yields a slightly weaker criterion. This may be useful in some situations.

Remark 1.1 *An inductive relation for the program P is a transition invariant for P.*

Program P,
liveness property Ψ,
fairness assumption Φ,
Büchi automaton $\mathcal{A}_{\Phi,\Psi}$,
parallel composition of P and $\mathcal{A}_{\Phi,\Psi}$ is program $P|||\mathcal{A}_{\Phi,\Psi}$ with:
transition relation R,
set of states Σ_Q,
set of accepting states Σ_F,
relation $T \subseteq \Sigma_Q \times \Sigma_Q$

P1:	R	$\subseteq$	T
P2:	$T \circ R$	$\subseteq$	T
P3:	$T \cap (\Sigma_F \times \Sigma_F)$	disjunctively well-founded	

P satisfies Ψ under Φ

Figure 1.5: Rule LIVENESS.

Inductive relations are called *inductive transition invariants*.

Note that a transition invariant T, even if it is inductive, is in general not closed under the composition with itself, *i.e.*, in general

$$T \circ T \not\subseteq T.$$

In other words, a transition invariant, even if it is inductive, need not be transitive.

We note in passing a simple but perhaps curious consequence of Theorem 1.1 and Remark 1.1.

Corollary 1.1 (Compositionality) *A finite union of well-founded relations is well-founded if it is closed under the relational composition with itself.*

Proof. Let the relation T be the finite union of the well-founded relations that is closed under the composition with itself, *i.e.* $T \circ T \subseteq T$.

By Remark 1.1, T is an inductive transition invariant for itself. Since T is disjunctively well-founded, we have that T is well-founded by Theorem 1.1. □

Proof Rule Theorem 1.2 and Remark 1.1 give rise to a proof rule for the verification of liveness properties; see Figure 1.5. Again, the formulation uses the automata-based framework for verification of concurrent programs [51]. We obtain a proof rule for termination by taking R as the transition relation of the program P, a relation $T \subseteq \Sigma \times \Sigma$ and replacing $T \cap (\Sigma_F \times \Sigma_F)$ by T in the premise P3.

In our examples we split the reasoning on disjunctive well-foundedness and inductiveness. This can be seen as using an alternative, equivalent formulation of the proof rule: one takes two relations T and T' such that T satisfies the premise P3 and T' is a subset of T that satisfies the premises P1 and P2 (*i.e.*, one identifies T as a transition invariant by strengthening T with the inductive relation T'). The two formulations are equivalent since the disjunctive well-foundedness of a relation is inherited by each of its subsets.

As already mentioned, a transition invariant can be strengthened by restricting it to a given state invariant S. This means that if T is a transition invariant and S is a state invariant, then

$$T' \;=\; T \cap (S \times S)$$

is a (stronger) transition invariant.

Validation of the Premises of the Proof Rule We have assumed that the transition relation R of the program is given by a union of transition relation ρ_τ of transitions τ.

If T is given as the union $T = T_1 \cup \cdots \cup T_n$, then the composition $T \circ R$ is the union of the relations $T_i \circ \rho_\tau$ for $i \in \{1, \ldots, n\}$ and $\tau \in \mathcal{T}$. Each relation $T_i \circ \rho_\tau \in \mathcal{T}$ is represented by an assertion over unprimed and primed program variables. Thus, the premises P1 and P2 can be established by entailment checks between assertions.

The premise P3 can be established using traditional methods for proving termination. In the extreme case, when $n = 1$, *i.e.*, the transition invariant or its partitioning are ill-chosen, the reduction to the disjunctive well-foundedness has not brought any simplification and is as hard as before the reduction. In the other cases, with a well-chosen transition invariant and partitioning, the premise P3 can be established by a number of pairwise independent 'simple' well-foundedness tests.

Note that all relations T_i in the transition invariants of the programs presented in this chapter correspond to 'single while' programs that consist of a single while loop with only update statements in its body.

More generally, the relation $g(\vec{X}) \wedge e(\vec{X}', \vec{X})$ is well-founded if and only if the while loop

$$\left[\begin{array}{c} \textbf{while } g(\vec{X}) \textbf{ do} \\ e(\vec{X}', \vec{X}) \end{array}\right]$$

is terminating.

In the case of concurrent programs with linear-arithmetic expressions, the well-foundedness test in the premise P3 amounts to the termination test of single while programs, for which an efficient test exists; see [37, 49] and Chapter 4.

In the special case of finite-state systems (a case that we do not target), each 'small' termination problem is to check whether a transition is a self-loop.

Inductive Transition Invariants for Examples Each of the relations T shown in Section 1.3 and 1.4 is not inductive (*i.e.*, the composition of one of the relations T_i and one of the transition relations ρ_τ is not a subset of T). We formally identify each T as a transition invariant by presenting an inductive one that strengthens T (*i.e.*, is a subset of T). We thus complete the termination resp. liveness proof according to the proof rule.

NESTED-LOOPS The union of the following relations is an inductive transition invariant for the program NESTED-LOOPS (in the version according to the depicted

control-flow graph).

$$
\begin{aligned}
&at\_\ell_0 \wedge x \geq 0 \wedge x' \leq x \wedge at'\_\ell_2\\
&at\_\ell_2 \wedge x' < x \wedge at'\_\ell_0\\
&at\_\ell_2 \wedge x - y > 0 \wedge x' \leq x \wedge y' > y \wedge at'\_\ell_2\\
&at\_\ell_0 \wedge x \geq 0 \wedge x' < x \wedge at'\_\ell_0\\
&at\_\ell_2 \wedge x \geq 0 \wedge x' < x \wedge at'\_\ell_2
\end{aligned}
$$

The inductiveness can be easily verified. For example, the composition of the relation below (which is the transition for the straight-line code from the location ℓ_2 to ℓ_0; it is obtained by composing the transition between the locations ℓ_2 and ℓ_4 and the transition from ℓ_4 to ℓ_0),

$$at\_\ell_2 \wedge y \geq x \wedge x' = x - 1 \wedge y' = y \wedge at'\_\ell_0$$

with the first of the five relations above yields the relation below, a relation that entails the fourth.

$$at\_\ell_0 \wedge x \geq 0 \wedge x' \leq x - 1 \wedge at'\_\ell_0$$

CHOICE The union of the four relations below is an inductive transition invariant for the program CHOICE.

$$
\begin{aligned}
&x' < x \wedge y' \leq x\\
&x' < y - 1 \wedge y' \leq x + 1\\
&x' < y - 1 \wedge y' < y\\
&x' < x \wedge y' < y
\end{aligned}
$$

ANY-DOWN We next present (the interesting part of) an inductive transition invariant for the parallel composition $P|||\mathcal{A}_{\Phi,\Psi}$ of the program ANY-DOWN with the Büchi automaton $\mathcal{A}_{\Phi,\Psi}$ that accepts exactly the infinite sequences of program states that are fair, *i.e.*, where the second process does not wait forever. We do not present the relations where the values of one of the program counters are different before and after the transition; we only present the relations that are loops in the control flow graph for the program $P|||\mathcal{A}_{\Phi,\Psi}$. We omit the conjunct $\pi' = \pi$ in each of the assertions below.

$$
\begin{aligned}
&at\_q_F \wedge at\_\ell_2 \wedge at\_m_1 \wedge y > 0 \wedge x' = x \wedge y' < y\\
&at\_q_F \wedge at\_\ell_3 \wedge at\_m_1 \wedge y > 0 \wedge x' = x \wedge y' < y\\
\neg&at\_q_F \wedge at\_\ell_0 \wedge at\_m_0 \wedge x' = x\\
\neg&at\_q_F \wedge at\_\ell_1 \wedge at\_m_0 \wedge x' = x\\
\neg&at\_q_F \wedge at\_\ell_2 \wedge at\_m_1 \wedge y > 0 \wedge x' = x \wedge y' < y\\
\neg&at\_q_F \wedge at\_\ell_3 \wedge at\_m_1 \wedge y > 0 \wedge x' = x \wedge y' < y
\end{aligned}
$$

CONC-WHILES The transition invariant for $P|||\mathcal{A}_{\Phi,\Psi}$ contains the following relations. We show only those that are loops wrt. the location labels; again, we omit the

conjunct $\pi' = \pi$ in each assertion below.

$$\begin{aligned} at\_q_F \wedge x > 0 \wedge x' < x \wedge y' < x \\ at\_q_F \wedge y > 0 \wedge x' < y \wedge y' < y \\ \neg at\_q_F \wedge x > 0 \wedge x' \leq x \wedge y' < x \\ \neg at\_q_F \wedge y > 0 \wedge x' \leq y \wedge y' \leq y \end{aligned}$$

Soundness and Completeness The separation of the temporal reasoning from the reasoning about the auxiliary assertions in the 'relative' completeness statement below is common practice; see e.g. [32, 33].

Theorem 1.3 (Proof Rule LIVENESS) *The rule* LIVENESS *is sound, and complete relative to the first-order assertional validity and the well-foundedness validity of the relations that constitute the transition invariant.*

Proof. The soundness of the rule follows directly from Remark 1.1 and Theorem 1.2.

For proving the relative completeness, we observe that the transition invariant constructed in the proof of Theorem 1.2 is in fact inductive. In order to establish the completeness relative to assertional provability, we need to show that this inductive transition invariant is expressible by a first-order assertion.

We need to construct the assertion T over unprimed and primed program variables that denotes a transition invariant satisfying the premises of the rule LIVENESS. We omit the construction, which follows the lines of the method for constructing the assertion Acc that denotes the set of all accessible states [33]. □

Automated Liveness Proofs Given a program with the transition relation R, we are interested in the subclass of its inductive transition invariants.

We define the operator F over relations by

$$F(T) \;=\; T \circ R.$$

We write $F^{\#} \supseteq F$ and say that $F^{\#}$ is an *approximation* of F, if $F^{\#}(S) \supseteq F(S)$ holds for all relations S.

The inductive transition invariants are (exactly the) least fixed points above R of operators $F^{\#}$ such that $F^{\#} \supseteq F$.

There are many techniques based e.g. on widening or predicate abstraction that have been applied with great success to the automated construction of least fixed points of approximation of the *post* operator [1, 3, 6, 10, 11, 18, 19]. Now we can start to carry over the abstract interpretation techniques in order to construct least fixed points of approximations of the operator F. Thus, relations T that satisfy the premises P1 and P2 can be constructed automatically.

As already mentioned, the validation of the premise P3 can be automated for interesting classes of concurrent programs over linear-arithmetic expressions (see [8, 37, 49] and Chapter 4). Automated checks for other classes of programs are an open topic of research.

1.6 Related Work

There is a large body of work on proof rules for liveness properties of concurrent programs, see [16, 29, 32, 34]. They all rely on auxiliary well-founded (lexicographic) or-

derings for the transition relation, and not on independent orderings for sub-relations, as in our approach.

The automata-theoretic approach for verification of concurrent programs [51] reduces the verification problem to proving termination. It leaves open how to prove termination. We indicate one possible way.

A rank predicate [52] (a notion directly related to progress measures [24]) proves fair termination of a program if the rank does not increase in every computation step and decreases in the accepting states. In a disjunctively well-founded transition invariant a rank need not decrease in all sub-relations if an accepting state is visited, *i.e.*, the rank of one sub-relation must decrease and all other ranks may increase.

In [31], an axiomatic approach to prove total correctness (safety property + termination) of sequential programs uses assertions connecting the initial and final values of the program variables. This must not be confused with transition invariants that capture all pairs of intermediate values in computations of arbitrary length, possibly going through loops.

It is interesting to compare our use of Ramsey's theorem in the proofs of Theorems 1.1 and 1.2 with its use in the theory of (finite) Büchi automata (see *e.g.* [46, 48]). The equivalence classes over computation segments in our proofs are related to the state transformers in the *transition monoid* of the Büchi automaton. In both uses of Ramsey's theorem, the sets of transformers are finite and thus induce an equivalence relation of finite index (which is why Ramsey's theorem can be applied). However, our proofs consider *finite* sets of transformers over an *infinite* state space, as opposed to transformers over a finite state space.

The termination analysis for functional programs in [28] has been the starting point of our work. The analysis is based on the comparison of infinite paths in the control flow graph and in 'size-change graphs'; that comparison can be reduced to the inclusion test for Büchi automata. The transitive closure of a (finite) set of size-change graphs can be seen as a graph representation of a special case of a transition invariant.

1.7 Conclusion

We have presented a (sound and relatively complete) proof rule for the temporal verification of concurrent programs. In a well-chosen instantiation, this proof rule allows one to decompose the verification problem into a number of independent smaller verification problems: one for establishing a transition invariant, and the others for establishing the disjunctive well-foundedness. The former is done in a way that is reminiscent of establishing state invariants, using a familiar inductive reasoning. The other ones amount to testing the termination of single while loops.

Our conceptual contribution is the notion of a transition invariant, and its usefulness in temporal proofs. This notion is at the basis of our proof rule. In particular, it allows one to account for Büchi accepting conditions (and hence for fairness) in a direct way, namely by intersecting relations.

Our technical contribution is the characterization of the validity of termination or another liveness property by the existence of a disjunctively well-founded transition invariant. The application of Ramsey's theorem allows us to replace the argument that the transition relation R is contained in the *(transitive)* well-founded relation r_f induced by a ranking function f (*i.e.*, $(s, s') \in r_f$ if $f(s) > f(s')$) by the argument that the transitive closure of R is contained in a union of well-founded relations. This

means that we have

$$R \subseteq r_f \qquad \text{vs.} \qquad R^+ \subseteq T_1 \cup \cdots \cup T_n.$$

As outlined in Section 1.5, our proof rule is a starting point for the development of automated verification methods for liveness properties of concurrent programs. In Chapter 2, we have started one line of research based on predicate abstraction as used in the already existing tools for safety properties [1, 6, 19]; many different other ways are envisageable.

Another line of research are methods to reduce the size of the transition invariants by encoding relevant specific kinds of fairness, such as weak and strong fairness, in a more direct way than encoding them in Büchi automata. We address this question in Chapters 2 and 3.

Chapter 2

Transition Predicate Abstraction

2.1 Introduction

Since 1977, a high amount of research, both theoretical and applied, has been invested in honing the tools for abstract interpretation [10] for verifying safety and invariance properties of programs. This effort has been a success. One promising approach is *predicate abstraction* on which a number of academic and industrial tools are based [1, 6, 18, 19, 53].

What has been left open is how to obtain the same kind of tools for the full set of temporal properties. So far, there was no viable approach to the use of abstract interpretation for analogous tools establishing liveness properties (under fairness assumptions). This chapter presents the first steps towards such an approach. We believe that our work may open the door to a series of activities for liveness, similar to the one mentioned above for safety and invariance.

One basic idea of abstraction is to transform the program to be checked into a more abstract one, one on which the property still holds. When we are interested in termination under fairness assumptions, we need to solve two problems: the abstract program needs to preserve (1) the termination property and (2) the fairness assumptions (checking liveness can be reduced to fair termination, just as safety reduces to reachability). In this chapter, we show how to solve these two problems. We propose a transformation of a program into a node-labeled edge-labeled graph such that the termination property can be retrieved from the node labels and the fairness assumptions from the edge labels. (To avoid the possibility of confusion, note that our method does not check the absence of loops in the graph.) The transformation is based on *transition predicate abstraction*, an extension of predicate abstraction that we propose.

The different steps in our automated method for checking a liveness property under fairness assumptions are:

- the reduction of the liveness property to fair termination (this reduction is standard, see *e.g.* [51]);

- the transition predicate abstraction-based transformation of the program P into a node-labeled edge-labeled graph, the *abstract-transition program* $P^{\#}$;

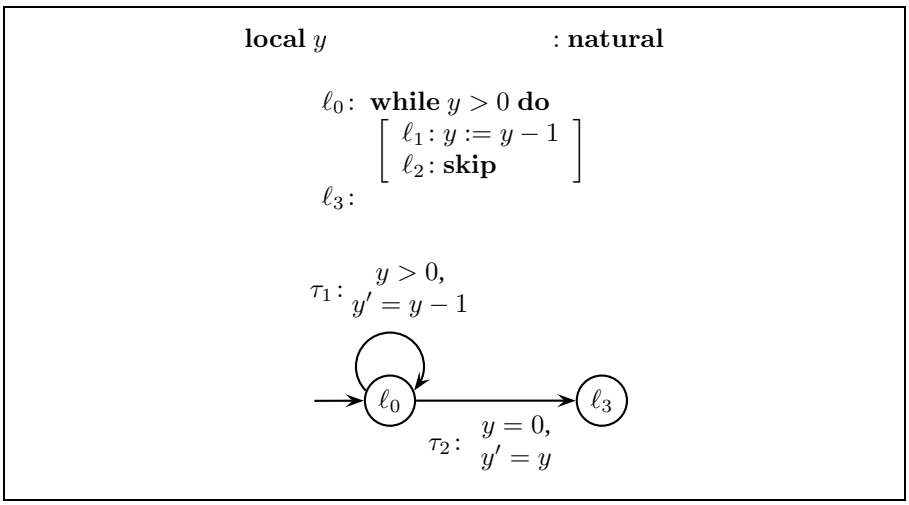

Figure 2.1: Program LOOP.

- a number of termination checks that mark some nodes of $P^{\#}$ as 'terminating';
- an algorithm on the automaton underlying $P^{\#}$ that marks some nodes as 'fair';
- the method returns 'property verified' if each 'fair' node is marked 'terminating'.

Our conceptual contribution lies in the use of transition predicates for automated liveness proofs. Our technical contributions are the algorithm to retrieve fairness in the abstract program $P^{\#}$, and the proof of the correctness of the overall method. We use both relevant kinds of fairness, which are justice and compassion (to model the assumption that a transition is eventually taken if it is continually resp. infinitely often enabled).

2.2 Related Work

Our work is most closely related to the work on predicate abstraction; see *e.g.* [1, 6, 18, 19, 53]. The key idea of predicate abstraction is to partition the state space of the program into a finite set of equivalence classes using predicates over states. The equivalence classes are treated as the *abstract states* forming the nodes of a finite graph. A safety property can then be checked on the abstract system.

Unfortunately, predicate abstraction is inherently limited to safety properties. That is because, every sufficiently long computation of the program (with the length greater then the number of abstract states) results in a computation of the abstract system that contains a loop. I.e., termination (as well as more general liveness properties) cannot be preserved by predicate abstraction.

We illustrate the limitation on a very simple program LOOP [21], shown on Figure 2.1 together with the (slightly simplified) control-flow graph. The predicates $y = 0$ and $y > 0$ split the data domain of the variable y into zero and pos. The corresponding abstraction transforms the program LOOP into the finite-state abstract program shown on Figure 2.2. That program contains a self-loop at the abstract state S_1, *i.e.* is not terminating. The abstract state S_1 corresponds to the conjunction $at\_\ell_0 \wedge y > 0$ denoting

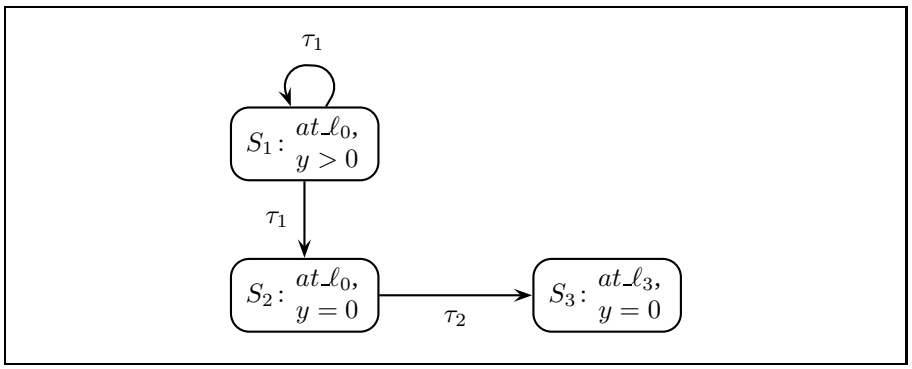

Figure 2.2: Non-terminating abstract-state program for LOOP.

the set of states where the program counter has the value ℓ_0 and y is strictly positive. If we split the abstract state S_1 (by adding more predicates) then at least one of the resulting abstract states will have a self-loop, and so on.

In the *augmented abstraction* framework for proving liveness properties, the finite-state abstraction is annotated by progress monitors or the like [21, 23, 36, 54]. The annotation involves the manual construction of ranking functions or other termination arguments. Until now, this has been the only known way to overcome the inherit limitation of predicate abstraction to safety properties. In contrast, the method that we propose does not require the manual construction of termination arguments.

In [38] we presented a proof rule for termination and liveness based on *transition invariants*. In this chapter, we make the first steps towards realizing its potential for automation.

We note a major difference in the notions of fairness used here and in [38]. In [38], we used an automata-theoretic notion of state-based fairness to formalize a uniform setting. Here we use justice and compassion, two transition-based notions of fairness. These are the two notions of fairness that are relevant with concrete concurrent programs. It is widely accepted that one needs a direct treatment of justice and compassion since the translation to the automata-theoretic notion is prohibitively expensive. As a consequence, the notion of transition invariant in [38] is not applicable as such. For intuition, an abstract-transition program $P^{\#}$ can be imagined as a new notion of transition invariant, one that encodes justice and compassion assumptions in a graph with labeled edges.

The abstract interpretation framework formalizes the conservative approximation of fixed point expressions [10]. For the verification of liveness properties denoted by fixed points expressions, this approximation involves the under-approximation of least fixed points or (equivalently) the over-approximation of greatest fixed points. Although possible in principle, the automation of the corresponding extrapolation seems difficult, and practical techniques (analogous to the extrapolation by intervals, convex hulls, Cartesian products, etc.) are not in sight (cf. [4, 15, 45, 50]).

One source of inspiration for the idea of abstracting relations is the work on higher-order abstract interpretation in [12]. Its instantiation to transition predicate abstraction and its use for liveness with justice and compassion is proper to this paper.

Verification diagrams are graphs that are useful to factorize deductive proofs of temporal properties including liveness [5]. Their nodes denote sets of states (and not

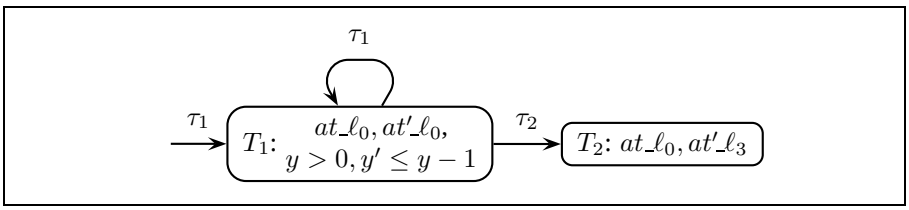

Figure 2.3: Abstract-transition program LOOP$^{\#}$.

pairs of states) and are hence close in spirit to abstract-state programs (and not to the abstract-transition programs). It may be interesting to consider verification diagrams with nodes denoting sets of *pairs* of states, and to come up with according proof rules.

2.3 Abstract-Transition Programs

Informal Description We propose to abstract *relations* instead of *sets of states*, and to use *transition predicate* abstraction instead of *predicate* abstraction. Transition predicates are binary relations over states (given *e.g.* by assertions over unprimed and primed program variables).

Transition predicate abstraction goes beyond the idea of abstracting a program by a finite *abstract-state* program. Instead, we abstract a program by a finite *abstract-transition* program. An abstract transition is a binary relation represented by a conjunction of transition predicates. An abstract-transition program is given by a finite directed graph whose nodes are labeled by abstract transitions, and whose edges are labeled by program transitions.

On Figure 2.3, we see the abstract-transition program LOOP$^{\#}$. One node is labeled by the abstract transition T_1. It corresponds to the conjunction of *transition predicates*

$$at\_\ell_0 \wedge at'\_\ell_0 \wedge y > 0 \wedge y' \leq y - 1$$

denoting the set of all pairs of states (s, s'), both at the program location ℓ_0. The value of y is strictly positive in the state s and changes to a strictly smaller value in s'. The node labeled by T_2 refers to states s and s' at ℓ_0 respectively at ℓ_3 (with unspecified values for y).

The abstract-transition program LOOP$^{\#}$ abstracts the program LOOP. What does this mean?

We first recall the meaning of abstraction of a program by an abstract-state program. If a state s has a transition to s' under the execution of the program transition τ, then there is an edge labeled by τ between two corresponding abstract states S_1 and S_2 (*i.e.* $s \in S_1$ and $s' \in S_2$).

The meaning of abstraction of a program by an abstract-transition program is analogous. If a pair of states (s, s') can be 'extended' to the pair (s, s'') by the execution of the program transition τ (which is: s' goes to s'' under the execution of the transition τ), then there is an edge labeled by τ between two corresponding abstract transition T_1 and T_2 (which is: $(s, s') \in T_1$ and $(s, s'') \in T_2$).

Note that LOOP$^{\#}$ only serves to illustrate the concept of abstract-transition programs. To illustrate how our method works to verify termination and general liveness properties, we will use concurrent programs with nested loops. In fact, the program

LOOP is an example of a *single while loop* program. Our method calls (as a subroutine) a termination check that exists for single while loop programs (see [8, 37, 49] and Chapter 4).

We now start the formal definitions.

Transition Predicates We define the building blocks for abstract-transition programs.

Definition 2.1 (Transition Predicate p) *A* transition predicate p *is a binary relation over states.*

Usually, transition predicates are given by atomic assertions over unprimed and primed program variables. We fix a transition predicate Id for the identity relation.

$$Id \;=\; \{(s,s) \mid s \in \Sigma\}$$

From now on, the formal statements refer to a fixed *finite* set of transition predicates $\mathcal{P}$.

The predicates at_ℓ and at'_ℓ are implicitly contained in $\mathcal{P}$, for all program locations ℓ.

Definition 2.2 (Abstract Transition T) *An* abstract transition T *is a conjunction of transition predicates. We write $\mathcal{T}_{\mathcal{P}}^{\#}$ for the (finite) set of abstract transitions. Formally,*

$$\mathcal{T}_{\mathcal{P}}^{\#} \;=\; \{p_1 \wedge \ldots \wedge p_n \mid n \geq 0 \text{ and } p_1, \ldots, p_n \in \mathcal{P}\}.$$

Alternatively, we may define an abstract transition to be a conjunction in which every transition predicate appears either positively or negated. In this case, abstract transitions can be identified by bit-vectors. The difference is only relevant for implementation issues.

An abstract-transition program uses abstract transitions for its node labels:

Definition 2.3 (Abstract-Transition Program $P^{\#}$) *An* abstract-transition program $P^{\#}$ *is a finite directed rooted node-labeled edge-labeled graph*

$$P^{\#} \;=\; \langle V, E, v_0, L_V, L_E \rangle$$

where:

- V *and* E *are the set of nodes resp. edges,*
- $v_0 \in V$ *is the root node,*
- $L_V : V \to \mathcal{T}_{\mathcal{P}}^{\#}$ *and* $L_V(v_0) = Id$,
  i.e., *every node* v *is labeled by an abstract transition* $L(v)$ *which we also write* T_v, *the root node is labeled* Id,
- $L_E : E \to \mathcal{T}$,
  i.e., *every edge* (u, v) *is labeled by a transition* τ.

We will often use the set V^- of all *non-root* nodes (on figures illustrating examples, we do not show v_0).

$$V^- \;=\; V \setminus \{v_0\}$$

We can now define the meaning of abstraction of a program P by an abstract-transition program $P^{\#}$. Later on, we present an algorithm for the transformation of a program P into an abstract-transition program $P^{\#}$.

Definition 2.4 (Abstraction $P \sqsubseteq P^{\#}$) *An abstract-transition program $P^{\#} = \langle V, E, v_0, L_V, L_E \rangle$ is an* abstraction *of the program $P = \langle \Sigma, \Theta, \mathcal{T} \rangle$ if for all nodes v_1 labeled by, say, the abstract transition T_1, and for all transitions τ of the program P,*

> *if T_1 contains a pair of states (s, s') such that s' goes to some state s'' under the transition τ, then*
>
> - *there exists a non-root node v_2 that is labeled by an abstract transition T_2 containing the pair (s, s''), and*
> - *there exists an edge from v_1 to v_2 labeled by τ.*

Formally:
$v_1 \in V$, $L_V(v_1) = T_1$, $(s, s') \in T_1$, $(s', s'') \in \rho_\tau$ implies the existence of $v_2 \in V^-$ and $(v_1, v_2) \in E$ such that $L_E(v_1, v_2) = \tau$ and, for $L_V(v_2) = T_2$, $(s, s'') \in T_2$.

Note that the target node v_2 in the definition above must be different from the root node v_0. However, there may exist a target node v_2 labeled by Id.

In the rest of the chapter, the notation $P^{\#}$ always refers to an abstract-transition program $P^{\#}$ that is an abstraction the program P, *i.e.* $P \sqsubseteq P^{\#}$.

2.4 Automated Abstraction $P \mapsto P^{\#}$

Given a finite set of transition predicates $\mathcal{P}$, the algorithm shown on Figure 2.4 takes a program P and returns a program $P^{\#}$ abstracting it, *i.e.* $P \sqsubseteq P^{\#}$.

The algorithm constructs the nodes (and edges) of $P^{\#}$ in a breadth-first manner. The set of nodes whose successors have not been yet explored are kept in the queue Q.

The set of transition predicates $\mathcal{P}$ defines a unique 'best-abstraction' function α for the abstract domain $\mathcal{T}^{\#}_{\mathcal{P}}$. It maps a binary relation T over states to the smallest abstract transition containing the relation T.

For example, if the set of transition predicates is

$$\mathcal{P} \;=\; \{x \geq 0, x' \leq x - 1, x' = x, x' \geq x + 1\},$$

the relation

$$T \;=\; x > 0 \wedge x' = x - 1$$

is abstracted to the abstract transition

$$\alpha(T) \;=\; x \geq 0 \wedge x' \leq x - 1.$$

The algorithm implements the abstraction function α using the following equality.

$$\alpha(T) \;=\; \bigwedge \{p \in \mathcal{P} \mid T \subseteq p\}$$

Here, the assertions p and T define binary instead of unary relations over states, and use primed and unprimed variables instead of just unprimed variables. Everything else is as in classical predicate abstraction. That is, a theorem prover is called for each entailment test "$T \subseteq p$". If n is the number of predicates, then for each newly created node and each transition τ we have n calls to the theorem prover. Thus, the theoretical worst-case number of calls to the theorem prover is the same as in classical predicate abstraction.

input
  P: program with finite set of transitions $\mathcal{T}$
  $\mathcal{P}$: finite set of transition predicates
output
  abstract-transition program $P^{\#}$ with:
    V: set of nodes labeled by abstract transitions
    E: set of edges labeled by transitions τ
begin
  $Q :=$ empty queue
  $\alpha := \lambda T.\ \bigwedge\{p \in \mathcal{P} \mid T \subseteq p\}$
  $v_0 :=$ new node labeled by Id
  $V := \{v_0\}$
  enqueue(Q, v_0)
  $E := \emptyset$
  while Q not empty **do**
    $u :=$ dequeue(Q)
    foreach $\tau \in \mathcal{T}$ **do**
      $T := \alpha(T_u \circ \rho_\tau)$
      if $T = \emptyset$ **then** continue with next τ **fi**
      if exists $w \in V^-$ such that $T = T_w$ **then**
        $v := w$
      else
        $v :=$ new node labeled by T
        $V := V \cup \{v\}$
        enqueue(Q, v)
      fi
      $(u, v) :=$ new edge labeled by τ
      $E := E \cup \{(u, v)\}$
    od
  od
end.

Figure 2.4: Transition predicate abstraction $P \mapsto P^{\#}$.

2.5 Overall Method

Our overall method to check a liveness property of a program under fairness assumptions consists of the five steps given in the introduction to this chapter.

We do not further elaborate the first step, which is the reduction of the verification problem for general temporal properties to the one for fair termination. This step is standard (cf. [51]), analogous one for safety and reachability.

We have just presented the second step, the transition predicate abstraction-based transformation of the program P into a node-labeled edge-labeled graph, the *abstract-transition program* $P^{\#}$. We now fix $P^{\#}$.

The third step checks, for each node v of $P^{\#}$, whether its label, the abstract transition T_v, is well-founded (and then marks the node accordingly as 'terminating' or not). In fact, our method can be parameterized by the well-foundedness test we apply. Here, we assume that the transition predicates are linear arithmetic formulas (without dis-

junction). Then we can apply one of the well-foundedness tests described in [8, 37, 49] and Chapter 4. For intuition, the well-foundedness of a relation defined by a conjunctive formula in primed and unprimed variables is the termination of a corresponding program that consists of a single while loop. The loop body only contains a simultaneous (possibly non-deterministic) update statement. For example, $x > 0 \wedge x' = x - 1$ corresponds to $\mathbf{while}\ x > 0\ \mathbf{do}\ x := x - 1$. From our experience, checking well-foundedness of abstract transitions (termination of single while loops) can be done very efficiently. For example, our prototype implementation of [37] handles over 500 single while loops in a couple of milliseconds.

The only missing link is the fourth step of our overall method: an algorithm on the automaton underlying $P^{\#}$ that marks nodes as 'fair' resp. 'unfair'. Before we give the formal definition of each kind of fairness, justice resp. compassion in Section 2.6 resp. Section 2.7, we outline the algorithm.

The first part of the algorithm computes, for each node v, a set $\mathsf{abc}(\mathcal{L}_v)$ of transitions (which we define in the next paragraph), *i.e.* $\mathsf{abc}(\mathcal{L}_v) \subseteq \mathcal{T}$. The second part checks a condition on $\mathsf{abc}(\mathcal{L}_v)$. That condition is specific to the kind of fairness, namely (2.1) in Section 2.6 resp. (2.2) in Section 2.7. The algorithm marks the node v according to the outcome of the check.

In its fifth, final step, our method returns 'property verified' if each 'fair' node is marked 'terminating'. Hence, the correctness of our overall method follows from Theorem 2.1 in Section 2.6 resp. Theorem 2.2 in Section 2.7, depending on the kind of fairness.

Finite Automata We observe that the graph of $P^{\#}$ without the node labels is the transition graph of a deterministic finite automaton over the alphabet $\mathcal{T}$. Each node $v \in V$ defines an automaton $\mathcal{A}_v$ whose initial state is the root node v_0, and whose only final state is the node v.

$$\mathcal{A}_v \ = \ \langle \mathcal{T}, V, \delta, v_0, \{v\} \rangle$$

The transition relation δ is the following.

$$\delta \ = \ \{(u, \tau, v) \mid (u, v) \in E \text{ is an edge labeled by } \tau\}$$

Let $\mathcal{L}_v$ be the language defined by the automaton $\mathcal{A}_v$. We next formalize the fact that the language $\mathcal{L}_v$ covers all relevant compositions of transition relations.

Lemma 2.1
Every word $\tau_1 \ldots \tau_n$ over transitions in $\mathcal{T}$ lies in the language $\mathcal{L}_v$ for a non-root node v, unless the composition of the corresponding transition relations is empty. Formally,

$$\rho_{\tau_1} \circ \ldots \circ \rho_{\tau_n} \neq \emptyset \implies \exists v \in V^{-}.\ \tau_1 \ldots \tau_n \in \mathcal{L}_v.$$

Proof. By induction over n. □

The set $\mathsf{abc}(\mathcal{L}_v)$ consists of all letters appearing in some word in $\mathcal{L}_v$, *i.e.* of all transitions $\tau \in \mathcal{T}$ labeling the edges that constitute a path from the root node v_0 to the node v.

$$\mathsf{abc}(\mathcal{L}_v) \ = \ \bigcap \{M \subseteq \mathcal{T} \mid \mathcal{L}_v \subseteq M^*\}$$

We compute $\mathsf{abc}(\mathcal{L}_v)$ by a standard algorithm for finite automata.

2.6 Justice

Justice is a conditional fairness requirement [33]. It is sensitive to the enabledness of transitions. A transition τ is *enabled* on the state s if the set of states $\{s' \mid (s, s') \in \rho_\tau\}$ is not empty. We write $\mathsf{en}(\tau)$ for the set of states on which the transition τ is enabled.

$$\mathsf{en}(\tau) \;=\; \{s \mid \text{exists } s' \in \Sigma \text{ such that } (s, s') \in \rho_\tau\}$$

The justice requirement is represented by a set $\mathcal{J}$ of *just* transitions, $\mathcal{J} \subseteq \mathcal{T}$. Every just transition that is continually enabled beyond a certain point must be taken infinitely often.

We make the following assumption on the transition relations of the program P.

Assumption 2.1 (Transition Disjointness for $\mathcal{J}$) *Transition relation of each just transition is disjoint from the transition relation of every other transition. Formally,*

$$\forall \tau^j \in \mathcal{J}\ \forall \tau \in \mathcal{T}.\ \tau^j \neq \tau \implies \rho_{\tau^j} \cap \rho_\tau = \emptyset.$$

The assumption is not a proper restriction. In fact, it is automatically fulfilled by the transition relations of SPL programs. For every pair of transitions τ_ℓ and τ_m that belong to different processes we have the following transition relations.

$$\begin{aligned} \rho_{\tau_\ell} &= at\_\ell \wedge at'\_\ell' \wedge at\_m \wedge at'\_m \wedge \ldots \\ \rho_{\tau_m} &= at\_\ell \wedge at'\_\ell \wedge at\_m \wedge at'\_m' \wedge \ldots \end{aligned}$$

Transitions that belong to the same process are marked with different labels, so they enabledness sets are disjoint.

We make the following assumption on the enabledness sets of transition in the program P.

Assumption 2.2 (Enabledness for $\mathcal{J}$) *The enabledness set of each just transition is either* disjoint *or* coincides *with the enabledness set of every other transition. Formally,*

$$\begin{aligned} \forall \tau^j \in \mathcal{J}\ \forall \tau \in \mathcal{T}.\ \tau^j \neq \tau \implies \\ (\mathsf{en}(\tau^j) \cap \mathsf{en}(\tau) = \emptyset \vee \\ \mathsf{en}(\tau^j) = \mathsf{en}(\tau)). \end{aligned}$$

Assumption 2.2 is not a proper restriction either; for completeness, we give the corresponding syntactic transformation in the appendix.

We define an auxiliary predicate $\mathsf{just}(v, \tau^j)$ as follows.

$$\begin{aligned} \mathsf{just}(v, \tau^j) \;=\; & \tau^j \in \mathsf{abc}(\mathcal{L}_v) \vee \\ & \exists \tau \in \mathsf{abc}(\mathcal{L}_v).\ \mathsf{en}(\tau) \cap \mathsf{en}(\tau^j) = \emptyset \end{aligned}$$

Given a non-root node $v \in V^-$ and a transition τ^j, the predicate $\mathsf{just}(v, \tau^j)$ holds if τ^j is either taken or not continually enabled on some path connecting the root v_0 and the node v.

A node $v \in V^-$ is marked (justly) 'fair' if the predicate $\mathsf{just}(v, \tau^j)$ holds for every just transition.

$$\mathsf{fair}_{\mathcal{J}}(v) \;=\; \forall \tau^j \in \mathcal{J}.\ \mathsf{just}(v, \tau^j) \tag{2.1}$$

We say that a program *justly terminates* if it does not have infinite computations that satisfy the justice requirement.

Theorem 2.1 (Just Termination) *The program P justly terminates if every non-root 'fair' marked node v of the abstract-transition program $P^{\#}$ is labeled by a well-founded abstract transition T_v. Formally,*

$$\forall v \in V^{-}.\ \mathsf{fair}_{\mathcal{J}}(v) \implies \mathsf{well\text{-}founded}(T_v).$$

Proof. Assume that the program P does not justly terminate. We show that there exists a non-root node v labeled by a non-well-founded abstract transition T_v, and that for every just transition τ^j the predicate $\mathsf{just}(v, \tau^j)$ holds.

Let $\sigma = s_1, s_2, \ldots$ be an infinite computation induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \ldots$, where $(s_i, s_{i+1}) \in \rho_{\tau_i}$ for all $i \geq 1$, that satisfies the justice requirement.

The computation σ partitions the set of just transitions $\mathcal{J}$ into the sets $\mathcal{J}^{d(isabled)}$ and $\mathcal{J}^{t(aken)}$ as follows. A transition $\tau \in \mathcal{J}$ is in the set $\mathcal{J}^d$ if it is not continually enabled. Otherwise, *i.e.*, if τ is taken infinitely often, we have $\tau \in \mathcal{J}^t$.

Let $L = l_1, l_2, \ldots$ be an infinite ordered set of indices of σ such that for all $i \geq 1$ we have:

- Every transition from $\mathcal{J}^d$ is not enabled on a state lying between the positions l_i and l_{i+1}.
$$\forall \tau \in \mathcal{J}^d\ \forall i \geq 1\ \exists\ l_i < p < l_{i+1}.\ s_p \notin \mathsf{en}(\tau)$$
- Every transition from $\mathcal{J}^t$ is taken on a state lying between the positions l_i and l_{i+1}.
$$\forall \tau \in \mathcal{J}^t\ \forall i \geq 1\ \exists\ l_i < p < l_{i+1}.\ \tau_p = \tau$$

The set L exists since σ satisfies the justice requirement.

For the fixed sequences ξ and L, we choose a function f that maps a pair of indices (k, l), where $k < l$, from L to one of the nodes of the abstract-transition program $P^{\#}$ in the following way. We define $f(k, l)$ to be the node v such that the word $\tau_k \ldots \tau_{l-1}$, which is a segment of ξ, is in the language $\mathcal{L}_v$. The function f exists, by Lemma 2.1.

The function f induces an equivalence relation $\sim$ on pairs of elements of L.

$$(k, l) \sim (k', l') \quad \text{if and only if} \quad f(k, l) = f(k', l')$$

Since the range of f is finite, the equivalence relation $\sim$ has finite index.

By Ramsey's theorem [41], there exists an infinite ordered set of indices $K = k_1, k_2, \ldots$, where $k_i \in L$ for all $i \geq 1$, that satisfies the following property. All pairs of elements in K belong to the same equivalence class. That is, there exists a non-root node v such that for all $k, l \in K$ such that $k < l$ we have $f(k, l) = v$. We fix the node v.

Since $f(k_i, k_{i+1}) = v$ for all $i \geq 1$, the infinite sequence $s_{k_1}, s_{k_2}, \ldots$ is induced by the relation T_v.

$$(s_{k_i}, s_{k_{i+1}}) \in T_v \quad \text{for all } i \geq 1$$

We conclude that the abstract transition T_v is not well-founded.

We show that each transition $\tau^j \in \mathcal{J}^t$ is contained in the set of transitions $\mathsf{abc}(\mathcal{L}_v)$. By the choice of the set L and taking into consideration that the set K is a subset of L, we have

$$\tau^j \in \{\tau_{l_i}, \ldots, \tau_{l_{i+1}-1}\} \subseteq \{\tau_{k_i}, \ldots, \tau_{k_{i+1}-1}\} \text{ for all } i \geq 1.$$

Since the word $\tau_{k_i} \ldots \tau_{k_{i+1}-1}$ is in the language $\mathcal{L}_v$, we conclude $\tau^j \in \mathsf{abc}(\mathcal{L}_v)$.

We show that for every $\tau^d \in \mathcal{J}^d$ there exists a transition $\tau \in \mathsf{abc}(\mathcal{L}_v)$ such that $\mathsf{en}(\tau) \cap \mathsf{en}(\tau^d) = \emptyset$. By the choice of L, there exists a position p in σ between the positions k_i and k_{i+1} such that the transition τ^d is not enabled on the state s_p. Thus, the transition from the state s_p to its successor state is induced by a transition $\tau \neq \tau^d$. We have $\tau \in \mathsf{abc}(\mathcal{L}_v)$. By Assumption 2.2, the sets $\mathsf{en}(\tau^d)$ and $\mathsf{en}(\tau)$ are disjoint. $\square$

We now illustrate an application of Theorem 2.1 for proving just termination of example programs.

ANY-DOWN We show the program ANY-DOWN on Figure 1.3 in Chapter 1. We obtain the control-flow graph shown on Figure 2.5 by taking the asynchronous parallel composition of the processes. Every transition is just.

$$\mathcal{J} \;=\; \{\tau_1, \ldots, \tau_4\}.$$

We compute the abstract-transition program ANY-DOWN$^{\#}$, shown on Figure 2.6, by taking the following set of transition predicates.

$$\mathcal{P} \;=\; \{x = 0,\; x = 1,\; y > 0,\; y' \leq y - 1\}$$

The abstract transition T_1 is the only one that is not well-founded. From the graph of ANY-DOWN$^{\#}$, we obtain the following set $\mathsf{abc}(\mathcal{L}_1)$.

$$\mathsf{abc}(\mathcal{L}_1) \;=\; \{\tau_1\}$$

Since the enabledness condition of the transition τ_1 coincides with the enabledness condition of the transition τ_4, the predicate $\mathsf{just}(1, \tau_4)$ does not hold. Hence, the non-well-foundedness of T_1 is not required for the just termination of ANY-DOWN. Since all other abstract transitions are well-founded, by Theorem 2.1, we conclude the ANY-DOWN justly terminates.

ANY-WHILE We make the program ANY-DOWN more interesting by adding a loop in the second process. The resulting program ANY-WHILE and the control-flow graph for the parallel composition of its processes are shown on Figures 2.7 and resp. 2.8. Every transition is just.

$$\mathcal{J} \;=\; \{\tau_1, \ldots, \tau_6\}.$$

For the set of transition predicates

$$\begin{aligned}\mathcal{P} \;=\; \{&x = 0,\; x = 1,\; x' = x,\; x' = 0,\\ &y > 0,\; y' = y,\; y' \leq y - 1\}\end{aligned}$$

we compute the abstract-transition program ANY-WHILE$^{\#}$, shown on Figure 2.9.

We observe that the abstract transitions T_1, T_5, and T_6 are not well-founded. We read the following sets from the graph of ANY-WHILE$^{\#}$.

$$\begin{aligned}\mathsf{abc}(\mathcal{L}_1) \;&=\; \{\tau_1\}\\ \mathsf{abc}(\mathcal{L}_5) \;&=\; \{\tau_5\}\\ \mathsf{abc}(\mathcal{L}_6) \;&=\; \{\tau_6\}\end{aligned}$$

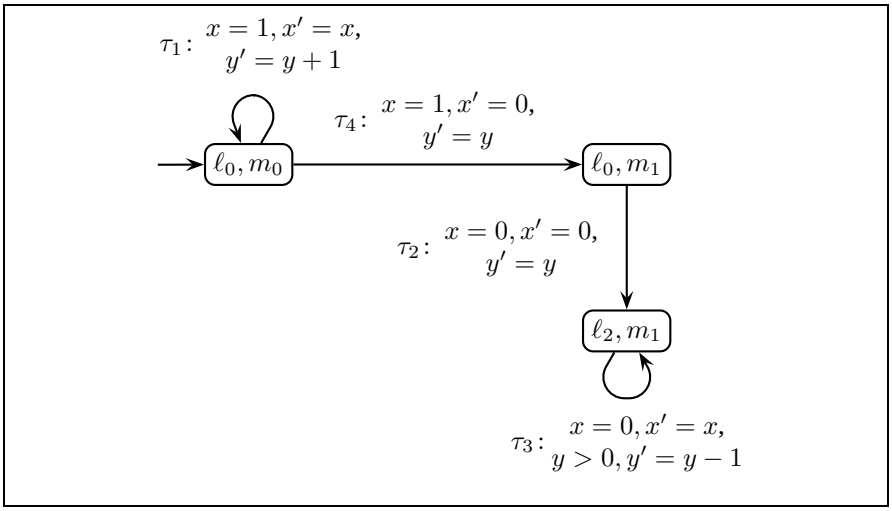

Figure 2.5: Control-flow graph for the parallel composition of processes P_1 and P_2 in ANY-DOWN.

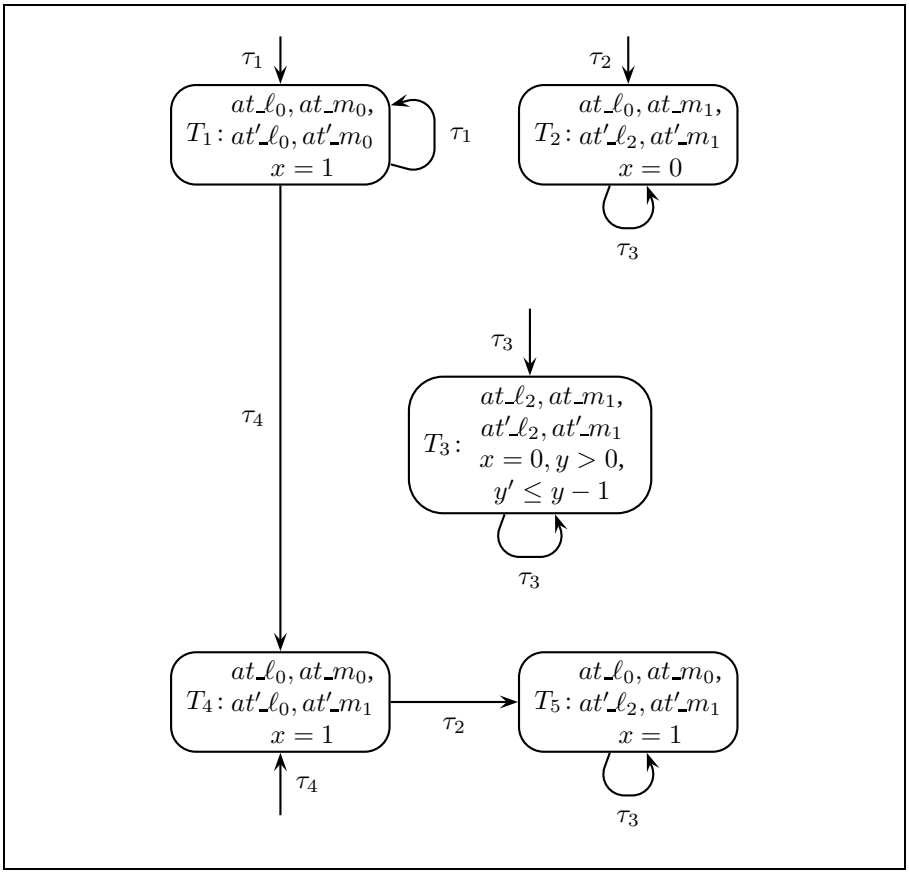

Figure 2.6: Abstract-transition program ANY-DOWN$^{\#}$.

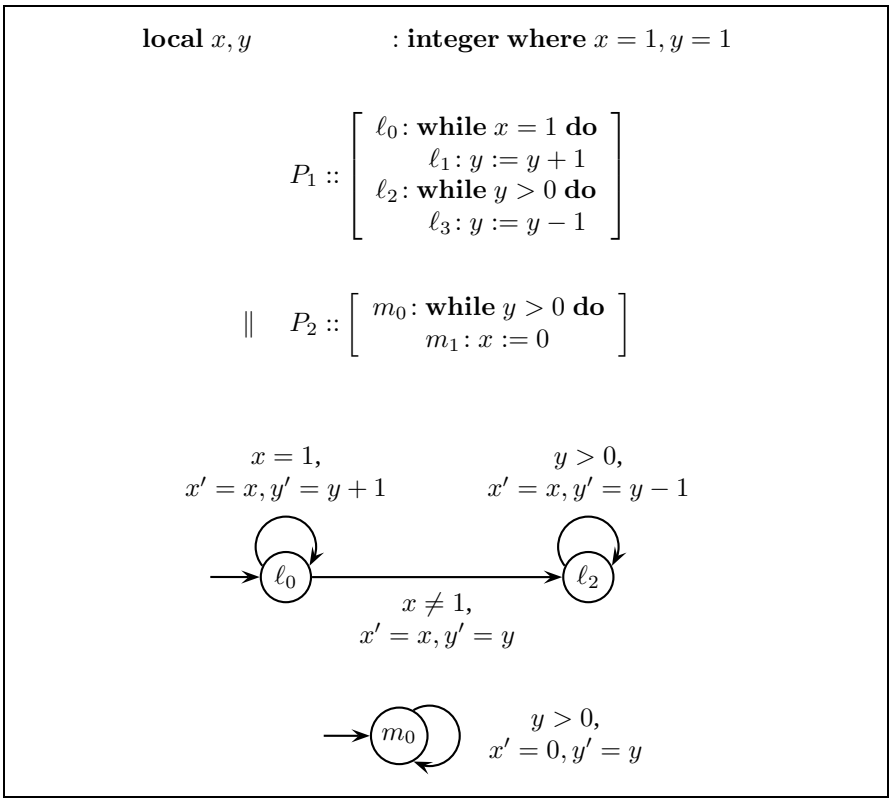

Figure 2.7: Program ANY-WHILE.

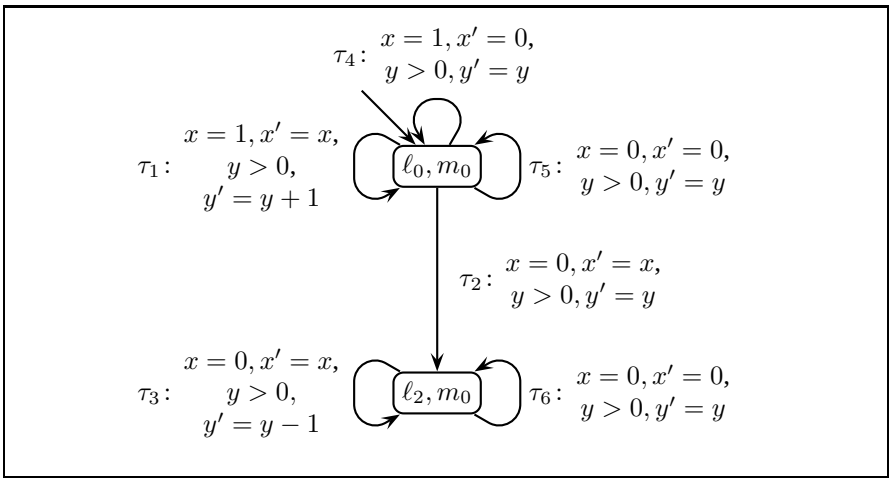

Figure 2.8: Control-flow graph for the parallel composition of the processes P_1 and P_2 in ANY-WHILE.

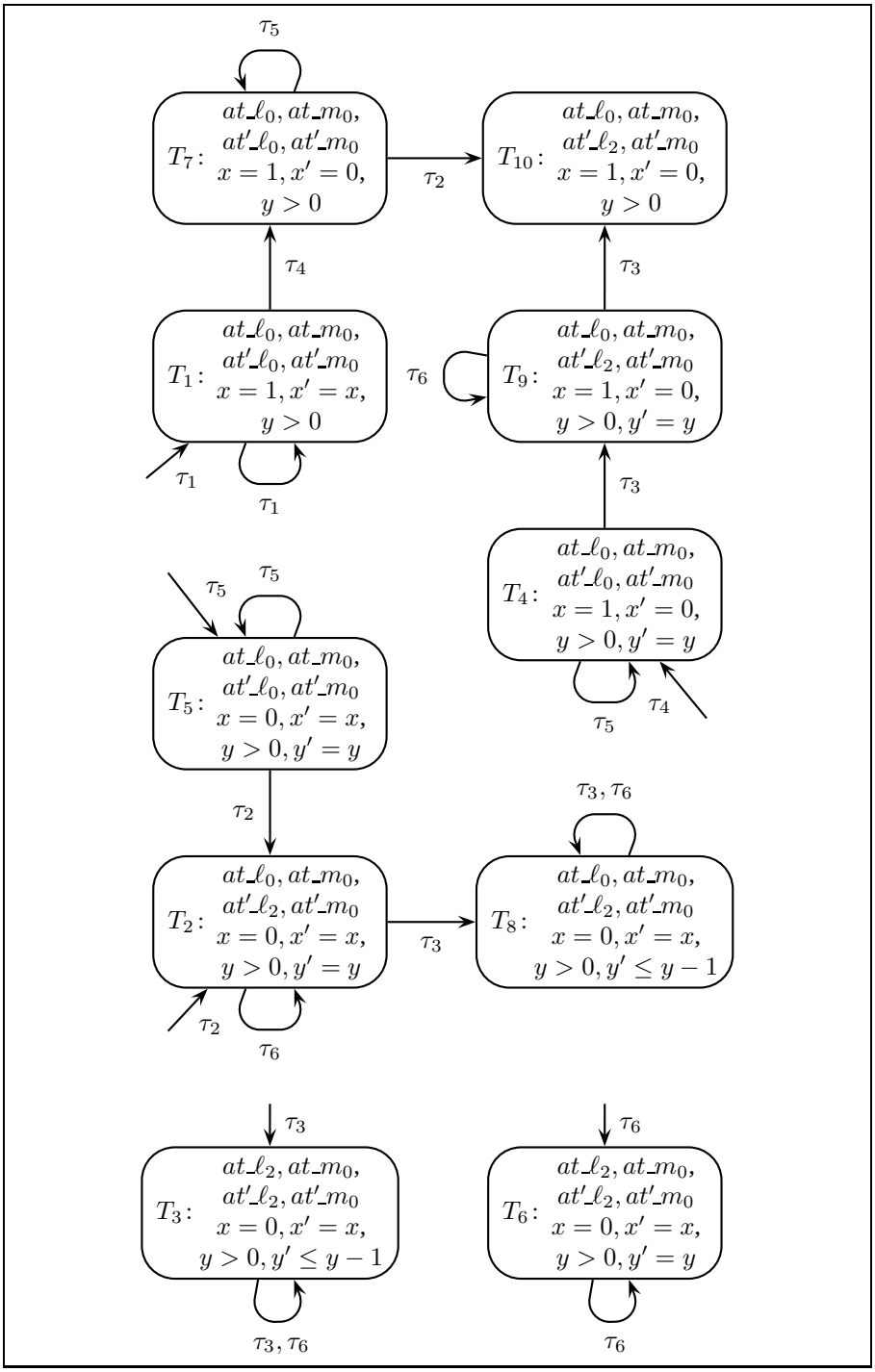

Figure 2.9: Abstract-transition program ANY-WHILE$^{\#}$.

Looking at the control-flow graph on Figure 2.8, we observes the following.

$$\begin{aligned}\mathsf{en}(\tau_1) &= \mathsf{en}(\tau_4)\\ \mathsf{en}(\tau_5) &= \mathsf{en}(\tau_2)\\ \mathsf{en}(\tau_6) &= \mathsf{en}(\tau_3)\end{aligned}$$

This means that the predicates $\mathsf{just}(1,\tau_4)$, $\mathsf{just}(5,\tau_2)$, and $\mathsf{just}(6,\tau_3)$ do not hold. Hence, the well-foundedness of T_1, T_5, and T_6 is not required for the just termination. We conclude that ANY-DOWN justly terminates.

2.7 Compassion

Compassion is another conditional fairness requirement [33]. Compared to justice, it is not sensitive to the interruption of transition enabledness infinitely many times. Compassion requirement is represented by a set $\mathcal{C}$ of *compassionate* transitions, $\mathcal{C} \subseteq \mathcal{T}$. Every compassionate transition that is enabled infinitely often must be taken infinitely often.

We extend Assumption 2.1 to compassionate transitions. We also extend Assumption 2.2 to compassionate transitions.

Assumption 2.3 (Enabledness for $\mathcal{C}$) *The enabledness set of each compassionate transition is either* disjoint *or* coincides *with the enabledness set of every other transition.*

Again, this assumption is not a proper restriction (see the appendix for details).

For dealing with compassion, we are interested in the set of letters (transitions) $\mathsf{abc}(\bigcap \mathcal{L}_v)$ that appear in every word of the language $\mathcal{L}_v$.

$$\mathsf{abc}(\bigcap \mathcal{L}_v) \;=\; \{\tau \mid \mathcal{L}_v \cap (\mathcal{T} \setminus \{\tau\})^* = \emptyset\}$$

We compute the set $\mathsf{abc}(\bigcap \mathcal{L}_v)$ by a standard algorithm.

We define an auxiliary predicate $\mathsf{comp}(v, \tau^c)$ as follows.

$$\begin{aligned}\mathsf{comp}(v,\tau^c) \;= \tau^c \in \mathsf{abc}(\mathcal{L}_v) \;\vee\\ \forall \tau \in \mathsf{abc}(\bigcap \mathcal{L}_v).\; \mathsf{en}(\tau) \cap \mathsf{en}(\tau^c) = \emptyset\end{aligned}$$

Given a non-root node $v \in V^-$ and a transition τ^c, the predicate $\mathsf{comp}(v, \tau^c)$ holds if τ^c is either taken on some path connecting the nodes v_0 and v, or if τ^c is not continually enabled on every path between v_0 and v. If the later case applies, then τ^c may be continually disabled on every path connecting v_0 and v.

A node $v \in V^-$ is marked (compassionately) 'fair' if the predicate $\mathsf{comp}(v, \tau^c)$ holds for every compassionate transition.

$$\mathsf{fair}_{\mathcal{C}}(v) \;=\; \forall \tau^c \in \mathcal{C}.\; \mathsf{comp}(v, \tau^c) \tag{2.2}$$

We say that a program *compassionately terminates* if it does not have infinite computations that satisfy the compassion requirement.

Theorem 2.2 (Compassionate Termination) *The program P compassionately terminates if every non-root 'fair' marked node v of the abstract-transition program $P^{\#}$ is labeled by a well-founded abstract transition T_v. Formally,*

$$\forall v \in V^-.\; \mathsf{fair}_{\mathcal{C}}(c) \implies \mathsf{well\text{-}founded}(T_v).$$

Proof. Assume that the program P does not compassionately terminate. We show that there exists a non-root node v labeled by a non-well-founded abstract transition T_v, and that for every compassionate transition τ^c the predicate $\mathsf{comp}(v, \tau^c)$ holds.

Let $\sigma = s_1, s_2, \ldots$ be an infinite computation induced by the infinite sequence of transitions $\xi = \tau_1, \tau_2, \ldots$, where $(s_i, s_{i+1}) \in \rho_{\tau_i}$ for all $i \geq 1$, that satisfies the compassion requirement.

The computation σ partitions the set of compassionate transitions $\mathcal{C}$ into the sets $\mathcal{C}^{d(isabled)}$ and $\mathcal{C}^{t(aken)}$ as follows. A transition $\tau \in \mathcal{C}$ is in the set $\mathcal{C}^d$ if it is not enabled infinitely often. Otherwise, *i.e.*, if τ is taken infinitely often, we have $\tau \in \mathcal{C}^t$.

Let $L = l_1, l_2, \ldots$ be an infinite ordered set of indices of σ such that:

- Every transition $\tau \in \mathcal{C}^d$ is not enabled on states at positions after l_1.

$$\forall \tau \in \mathcal{C}^d \; \forall p \geq l_1.\; s_p \notin \mathsf{en}(\tau)$$

- Every transition $\tau \in \mathcal{C}^t$ is taken on a state lying between the positions l_i and l_{i+1} for all $i \geq 1$.

$$\forall \tau \in \mathcal{C}^t \; \forall i \geq 1 \; \exists \; l_i < p < l_{i+1}.\; \tau_p = \tau$$

By defining an equivalence relation on pair from the set L and applying Ramsey's theorem along the lines of the proof of Theorem 2.1, we obtain an infinite ordered set $K \subseteq L$ and a non-root node v with the following property. For every pair of elements (k, l) in K we have $f(k, l) = v$. Again, we observe that the abstract transition T_v is not well-founded. Furthermore, since every transition from $\mathcal{C}^t$ is taken on a state between the positions k_i and k_{i+1} for all $i \geq 1$, we conclude that $\mathcal{C}^t$ is contained in the set of transitions $\mathsf{abc}(\mathcal{L}_v)$.

By the choice of L, a transition $\tau^d \in \mathcal{C}^d$ is not enabled on the state s_p for every position p in σ after the position k_1. Since every transition $\tau \in \mathsf{abc}(\bigcap \mathcal{L}_v)$ must appear between the positions k_i and k_{i+1}, we conclude that there exists a state s such that $s \in \mathsf{en}(\tau)$ and $s \notin \mathsf{en}(\tau^d)$. By Assumption 2.3, the sets $\mathsf{en}(\tau^d)$ and $\mathsf{en}(\tau)$ are disjoint. □

SUB-SKIP We illustrate Theorem 2.2 on the program SUB-SKIP, shown on Figure 2.10. The set of compassionate transitions $\mathcal{C}$ is the following.

$$\mathcal{C} \; = \; \{\tau_2, \tau_3\}$$

Every infinite computation σ of SUB-SKIP may take the transition τ_2 only finitely many times, although it is enabled infinitely often, thus, violating the compassion requirement $\mathcal{C}$.

We show the abstract transition program SUB-SKIP$^{\#}$ on Figure 2.11. We compute SUB-SKIP$^{\#}$ by applying the set of transition predicates below.

$$\mathcal{P} \; = \; \{y > 0,\; y' \leq y,\; y' \leq y - 1\}$$

The only non-well-founded abstract transitions are T_5 and T_7. We show that according to Theorem 2.2, the well-foundedness of these two abstract transitions is not needed for proving compassionate termination. We show that the predicates $\mathsf{comp}(5, \tau_2)$ and $\mathsf{comp}(7, \tau_2)$ do not hold.

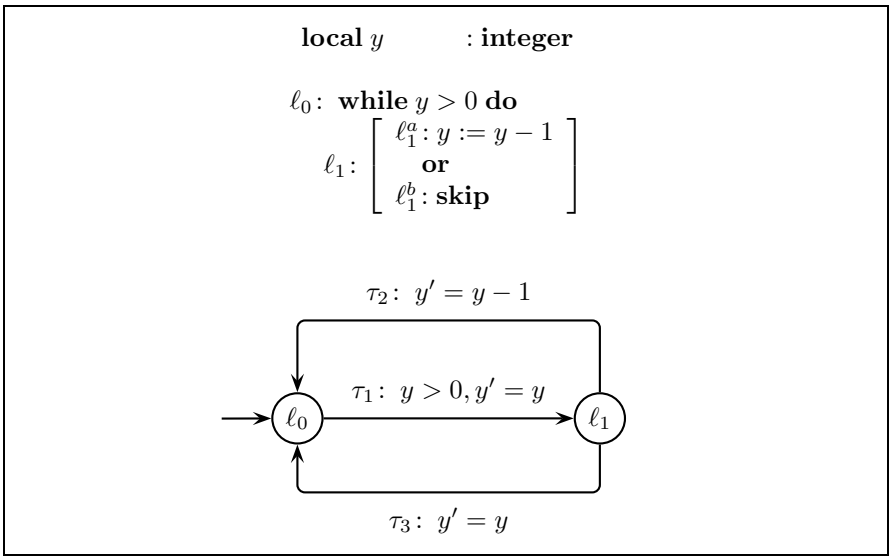

Figure 2.10: Program SUB-SKIP.

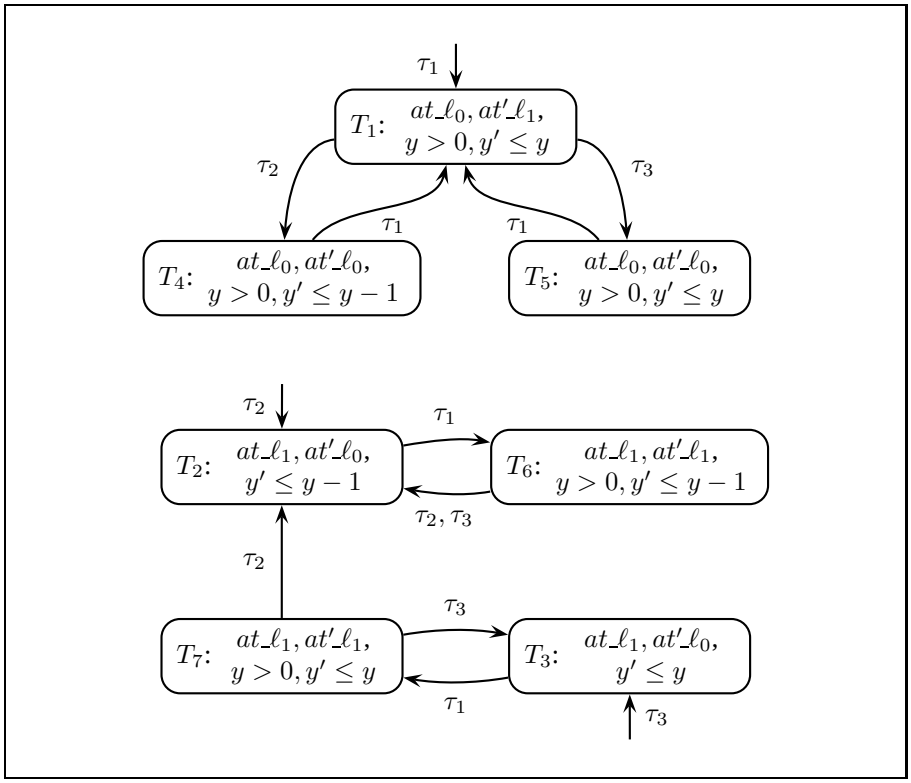

Figure 2.11: Abstract-transition program SUB-SKIP$^{\#}$.

From Figure 2.11, we obtain the following sets of transitions.

$$\mathsf{abc}(\mathcal{L}_5) = \mathsf{abc}(\mathcal{L}_7) = \mathsf{abc}(\bigcap \mathcal{L}_5) = \mathsf{abc}(\bigcap \mathcal{L}_7) = \{\tau_1, \tau_3\}$$

Furthermore, we observe (on Figure 2.10)

$$\mathsf{en}(\tau_2) \;=\; \mathsf{en}(\tau_3).$$

Hence, the predicates $\mathsf{comp}(5, \tau_2)$ and $\mathsf{comp}(7, \tau_2)$ do not hold.

2.8 Enabledness Assumptions

For completeness, we give the syntactic transformation for Assumptions 2.2 and 2.3.

We replace every fair transition $\tau \in \mathcal{J} \cup \mathcal{C}$ by a set of transitions obtained as follows. For each bit-vector over the enabledness sets of transitions $\mathcal{T} \setminus \{\tau\}$ we create a new transition with the transition relation obtained from ρ_τ by intersecting its enabledness set $\mathsf{en}(\tau)$ with the set defined by the bit-vector. The following conditions hold for the transition relations and the enabledness sets obtained by splitting the transition τ into the set of transitions $\{\tau_1, \ldots, \tau_n\}$.

$$\mathsf{en}(\tau) \;=\; \mathsf{en}(\tau_1) \uplus \cdots \uplus \mathsf{en}(\tau_n) \tag{2.3a}$$

$$\rho_\tau \;=\; \rho_{\tau_1} \uplus \cdots \uplus \rho_{\tau_n} \tag{2.3b}$$

The set of just (compassionate) transitions $\mathcal{J}$ ($\mathcal{C}$) of the program is modified by replacing τ by the set $\{\tau_1, \ldots, \tau_n\}$.

We show that the above modification preserves the fair termination property.

Lemma 2.2
The program P with the set of just transitions $\mathcal{J}$ justly terminates if it justly terminates after replacing each just transition by the set of transitions satisfying Equation (2.3)*.*

Proof. Assume that there exists an infinite computation $\sigma = s_1, s_2, \ldots$ of the original program that satisfies the justice requirement $\mathcal{J}$. Since partitioning does not make the transition relation of the program smaller, see Equation (2.3b), σ is a computation of the modified program.

We show that for every $\tau \in \mathcal{J}$ replaced by the set of transitions $\{\tau_1, \ldots, \tau_n\}$, the computation σ satisfies the justice requirement for each τ_i, where $1 \leq i \leq n$.

If τ is disabled infinitely often then each of τ_i, for $1 \leq i \leq n$, is disabled infinitely often. If τ is continually enabled, and, hence, infinitely often taken, we consider the following two cases.

We assume that there exists an enabledness set $\mathsf{en}(\tau_j)$ for some $1 \leq j \leq n$ such that σ eventually does not leave the set $\mathsf{en}(\tau_j)$, formally,

$$\exists 1 \leq j \leq n \; \exists k \geq 1 \; \forall l \geq k.\; s_l \in \mathsf{en}(\tau_j).$$

Every transition τ_i, where $1 \leq i \neq j \leq n$, is not continually enabled, by Assumption 2.2. The transition τ_j is taken infinitely often, by Assumption 2.1.

If the assumption above does not hold, then none of the transitions τ_i, for $1 \leq i \leq n$, is continually enabled. □

Lemma 2.3
The program P with the set of compassionate transitions $\mathcal{C}$ compassionately terminates if it compassionately terminates after replacing each compassionate transition by the set of transitions satisfying Equation (2.3)*.*

Proof. Assume that there exists an infinite computation $\sigma = s_1, s_2, \ldots$ of the original program that satisfies the compassion requirement $\mathcal{C}$. Since partitioning does not make the transition relation of the program smaller, see Equation (2.3b), σ is a computation of the modified program.

We show that for each $\tau \in \mathcal{C}$ replaced by the set of transitions $\{\tau_1, \ldots, \tau_n\}$, the computation σ satisfies the computation requirement for each τ_i, where $1 \leq i \leq n$.

If τ is not enabled infinitely often then each of τ_i, for $1 \leq i \leq n$, is not enabled infinitely often. If τ is enabled often, and, hence, infinitely often taken, we consider the following two cases.

For each $1 \leq j \leq n$ such that the set $\mathsf{en}(\tau_j)$ is visited infinitely often, by Assumptions 2.1 and 2.3, the transition τ_j is taken infinitely often. All other transitions are not enabled infinitely often. □

2.9 Lexicographic Completeness

Our main interest is in fair termination. But let us look also at termination. This allows us to compare the power of transition predicate abstraction with the classical means to construct termination arguments for programs with nested loops, which is the lexicographic combination of ranking functions (see *e.g.* [34]). We show that, if each lexicographic component of a ranking function for the program can be expressed by some conjunction of transition predicates in $\mathcal{P}$, then transition predicate abstraction will construct a termination argument for the program.

The characterization of (plain) termination of a program P (namely, by the well-foundedness of the abstract transitions labeling the nodes of the abstract-transition program $P^{\#}$) is the instance of the characterization of fair termination where the set of fair transitions to be empty.

Termination *The program P terminates if every non-root node in the abstract-transition program $P^{\#}$ is labeled by well-founded abstract transitions. Formally,*

$$\forall v \in V^{-}.\ \mathsf{well\text{-}founded}(T_v).$$

We use the example program NESTED-LOOPS shown on Figure 1.1 in Chapter 1 to illustrate our method for plain termination.

We obtain the abstract-transition program NESTED-LOOPS$^{\#}$, shown on Figure 2.12, by taking the following set of transition predicates.

$$\begin{aligned}\mathcal{P} \ = \ \{&x \geq 0,\ x' \leq x,\ x' \leq x - 1,\\ &y > 0,\ y < x,\ y' \geq 2y\}\end{aligned}$$

The program NESTED-LOOPS terminates, since every non-root node of NESTED-LOOPS$^{\#}$ is labeled by a well-founded abstract transition.

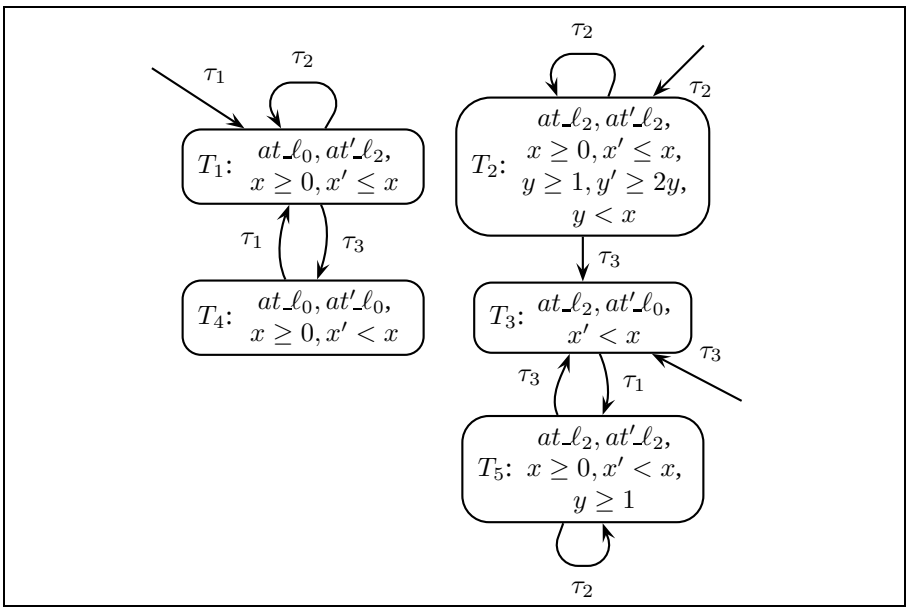

Figure 2.12: Abstract-transition program NESTED-LOOPS$^{\#}$.

Let $(f_1, \ldots, f_n)$ be a tuple of functions from the set of states Σ into the domains $(\mathcal{W}_1, \succ_1)$, ..., $(\mathcal{W}_n, \succ_n)$ such that $\succ_i$ is an ordering relation, *i.e.* transitive and irreflexive, for each $1 \leq i \leq n$.

The tuple $(f_1, \ldots, f_n)$ is a *lexicographic ranking function* for the program P if each ordering $\succ_i$ is well-founded and for every transition τ there exists an index $j \in \{1, \ldots, n\}$ such that the auxiliary predicate $\mathsf{lex}(\rho_\tau, j)$, defined as follows, holds.

$$\begin{aligned} \mathsf{lex}(R, j) \;=\; & \forall (s, s') \in R.\ f_j(s) \succ_j f_j(s') \wedge \\ & \forall 1 \leq i < j.\ f_i(s) \succeq_i f_i(s') \end{aligned}$$

For each function f_i we define a pair $f_i \succ_i f_i'$ and $f_i \succeq_i f_i'$ of transition predicates.

$$\begin{aligned} f_i \succ_i f_i' &= \{(s, s') \mid f_i(s) \succ_i f_i(s')\} \\ f_i \succeq_i f_i' &= \{(s, s') \mid f_i(s) \succeq_i f_i(s')\} \end{aligned}$$

Obviously, the transition predicate $f_i \succ_i f_i'$ is well-founded.

For example, the function $f(x, y) = x + y$, where the variables x and y range over integers, into the set of natural numbers defines the transition predicates $x + y > x' + y'$ and $x + y \geq x' + y'$.

Theorem 2.3 (Lexicographic Completeness) *If the set $\mathcal{T}_{\mathcal{P}}^{\#}$ generated by the set of transition predicates $\mathcal{P}$ contains the relation $f_i \succ_i f_i'$ and the relation $f_i \succeq_i f_i'$ for every component f_i of the lexicographic ranking function $(f_1, \ldots, f_n)$ for the program P, then every non-root node of the abstract program $P^{\#}$ obtained by transition predicate abstraction algorithm is labeled by a well-founded abstract transition.*

Proof. Let the tuple $(f_1, \ldots, f_n)$ be a lexicographic ranking function for the program P such that the transition predicates $f_i \succ_i f_i'$ and $f_i \succeq_i f_i'$ are contained in the set of abstract transitions $\mathcal{T}_{\mathcal{P}}^{\#}$ for each component f_i of the tuple.

We prove for each non-root node v, by induction over the length of a shortest path from the root node v_0 to the node v, that there exists an index $j \in \{1, \ldots, n\}$ such that the predicate $\mathsf{lex}(T_v, j)$ holds. The well-foundedness of T_v follows directly.

For the base case, let τ be the transition that labels the edge from the node v_0 to the node v. Since $\mathsf{lex}(\rho_\tau, j)$ holds for some $j \in \{1, \ldots, n\}$, we have

$$\begin{aligned} \rho_\tau \subseteq f_j \succ_j f'_j \ \in \mathcal{T}^\#_\mathcal{P}, \\ \forall 1 \leq i < j.\ \rho_\tau \subseteq f_i \succeq_i f'_i \ \in \mathcal{T}^\#_\mathcal{P}. \end{aligned}$$

Since α is the 'best-abstraction' function, we have

$$\begin{aligned} \alpha(\rho_\tau) \subseteq f_j \succ_j f'_j, \\ \forall 1 \leq i < j.\ \alpha(\rho_\tau) \subseteq f_i \succeq_i f'_i. \end{aligned}$$

Hence, we conclude $\mathsf{lex}(T_v, j)$ where $T_v = \alpha(\rho_\tau)$.

For the induction step, let u be a predecessor node of a non-root node v such that u is on a shortest path from v_0 to v. Let the predicate $\mathsf{lex}(T_u, j)$ hold for some index $j \in \{1, \ldots, n\}$. For a transition τ that labels the edge (u, v) there exists an index $l \in \{1, \ldots, n\}$ such that $\mathsf{lex}(\rho_\tau, l)$ holds. Let $m = \min(j, l)$. We show that $\mathsf{lex}(\alpha(T_v), m)$ holds.

By the induction hypothesis, we have

$$T_u \subseteq f_j \succ_j f'_j$$

and

$$\forall 1 \leq i < j.\ T_u \subseteq f_i \succeq_i f'_i.$$

From $\mathsf{lex}(\rho_\tau, l)$ we have

$$\rho_\tau \subseteq f_l \succ_l f'_l$$

and

$$\forall 1 \leq k < l.\ \rho_\tau \subseteq f_k \succeq_k f'_k.$$

By the transitivity of $\succ_i$ for $1 \leq i \leq n$, we have

$$\begin{aligned} T_u \circ \rho_\tau \subseteq f_m \succ_m f'_m, \\ \forall 1 \leq i < m.\ T_u \circ \rho_\tau \subseteq f_i \succeq_i f'_i. \end{aligned}$$

Analogously to the base case, we conclude $\mathsf{lex}(T_v, m)$, where $T_v = \alpha(T_u \circ \rho_\tau)$. $\square$

The following example illustrates that transition predicate abstraction may apply to programs whose termination cannot be proven by lexicographic ranking functions whose components are contained in $\mathcal{T}^\#_\mathcal{P}$.

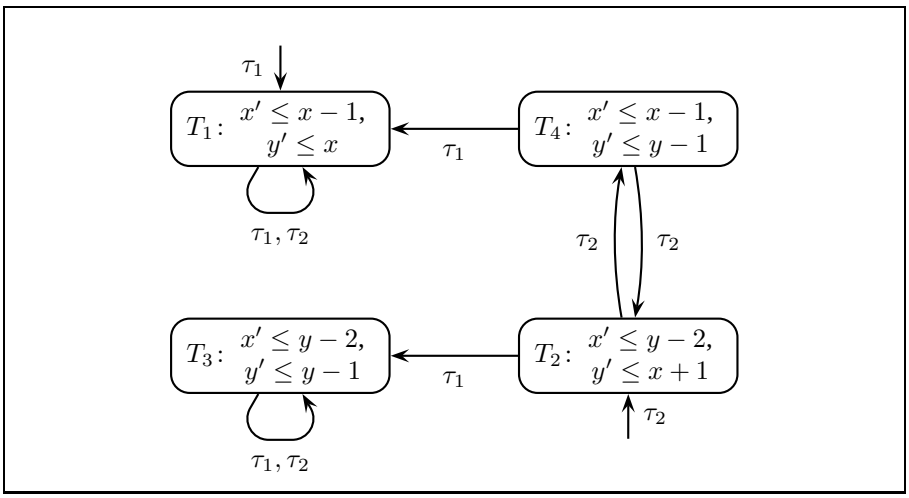

Figure 2.13: Abstract-transition program CHOICE$^{\#}$.

CHOICE We consider the program CHOICE shown on Figure 1.2 in Chapter 1. This program terminates. As one can easily see, no lexicographic combination of the functions

$$f_1(x,y) \;=\; x, \qquad f_2(x,y) \;=\; y, \qquad f_3(x,y) \;=\; x+y$$

is a ranking function for CHOICE. Executing the transition τ_1 may strictly increase the value of x and $x+y$, and executing the transition τ_2 the value of y may increase.

We compute the abstract-transition program CHOICE$^{\#}$, shown on Figure 2.13, by taking the following set of transition predicates.

$$\begin{aligned}\mathcal{P} \;=\; \{&x' \leq x,\ x' \leq x-1,\ x' \leq y-2,\\ &y' \leq y,\ y' \leq y-1,\ y' \leq x+1,\ y' \leq x\}\end{aligned}$$

Note that the set of abstract transition $\mathcal{T}^{\#}_{\mathcal{P}}$ induced by the transition predicates above contains the transition predicates $f_i \succ_i f_i'$ and $f_i \succeq_i f_i'$ for each $i \in \{1,2,3\}$ (and no other ranking functions.)

We observe that every non-root node in CHOICE$^{\#}$ is labeled by a well-founded abstract transition, *i.e.*, the program CHOICE terminates.

2.10 Conclusion

In this chapter, we have proposed the extension of predicate abstraction to transition predicate abstraction as a way to overcome the inherent limitation of predicate abstraction to safety properties. Previously, the only known way to overcome this limitation was to annotate the finite-state abstraction of a program in a process that involved the manual construction of ranking functions. We have gone beyond the idea of abstracting a program to a finite-state program and checking the absence of loops in its finite graph. Instead, we have given the transformation of a program into a finite *abstract-transition* program. We have given algorithms to check fair termination on the abstract-transition

program. The two algorithms together yield an automated method for the verification of liveness properties under full fairness assumptions (justice and compassion). In conclusion, we have exhibited principles that extend the applicability of predicate abstraction-based program verification to the full set of temporal properties.

We believe that our work may trigger a series of activities to develop tools for checking liveness, similar to the series of activities that have lead to the success of tools for safety and invariance properties [1, 6, 18, 19, 53]. Although it is too early for a systematic practical evaluation, we have developed a prototypical tool that implements the method described in this chapter and show its promising practical potential on concrete examples (including the ones in this chapter).

The logical next step is to investigate counterexample-driven abstraction refinement [1, 7, 19]. Our tool extracts transition predicates from guards (which yields the special case of assertions such as $x > 0$, *i.e.* in unprimed variables) and transition predicates of the form $x' \leq e$ and $x' \geq e$ from update statements `x:=e`). Although this was sufficient for our experiments so far, an automated counterexample-driven abstraction refinement will be desirable at some point. A counterexample will here be a relation $\tau_1 \circ \ldots \circ \tau_n$ corresponding to a path in the graph of an abstract-transition program, a path that leads to a 'fair', 'non-terminating' node.

Our algorithm suggests a verification methodology where the input to the algorithm is a liveness property without fairness assumptions. One then takes the computed abstract-transition program and its node labeling ('terminating' or not) to derive what fairness assumptions are required for the liveness property to hold. It should be possible to automate this derivation step.

Chapter 3

Labeled Transition Invariants

3.1 Introduction

Most temporal properties of concurrent programs only hold under certain assumptions concerning treatment of program transitions. We typically need to assume that every program transition is eventually taken if continually enabled. This assumption is known as justice requirement. Furthermore, we require that some transitions must be taken infinitely often if enabled infinitely often. This assumption is called compassion requirement. One possible way to express justice requirements is to demand that the starting location of every transition is infinitely often left during the computation, *i.e.*, the control does not stay in some starting location forever. Compassion requirements can be expressed in a similar way. Thus, we obtain fairness requirements imposed on sets of program states. A translation of these requirements into a specification automaton, as needed by the automata-theoretic framework [51], may produce a very large automaton, since the number of fairness requirements, *e.g.* induced by program transitions, can be large. When we try to prove the fair termination of the product of the synchronous parallel composition of the program and the specification automaton, we may face a product program that is too large to be handled by an automated tool or too incomprehensible for a human applying an interactive tool. Hence, proof methods that handle fairness requirements directly and avoid the blow-up are desirable.

In this chapter, we describe a proof rule for the verification of temporal properties that directly accounts for fairness requirements that are imposed on sets of states. We consider the full fairness, including both *justice* and *compassion*. We apply the automata-theoretic framework for the verification of general temporal properties, but we only encode the temporal property (but not the fairness requirements) into the specification automaton. We translate the acceptance condition of the product of the automata-theoretic construction into additional fairness requirements, which we handle in the same way as the fairness requirements of the program.

Our proof rule is based on an extended notion of transition invariants (see Chapter 1). Assume a program together with a transition invariant given by a finite union of relations. The program is terminating if every relation in the union is well-founded, *i.e.*, if the transition invariant is disjunctively well-founded (see Theorem 1.1). Disjunctive well-foundedness is a too strong condition for proving fair termination, since it does not account for the fairness requirements. We propose to extend each relation in the finite union with a set of *labels* that record the information about the satisfaction of fairness

requirements. Thus, we obtain a set of *labeled relations* that forms a *labeled transition invariant*. Each label corresponds to a fairness requirement, *e.g.*, one label for each program transition that should be handled in a fair way. A label is attached to a relation if all infinite sequences of program states induced by the relation falsify the fairness requirement that corresponds to the label. By a formal argument in this chapter (by Theorem 3.1 below), we can safely ignore the non-well-foundedness of the relations that are not labeled by the full set of labels. This means, we weaken the disjunctive well-foundedness criterion by taking fairness via the labeling into account. Next, we describe the condition when a label must be attached to a relation more precisely.

Assume that a transition invariant of the program, which is equipped with a set of fairness requirements, contains a non-well-founded element in its representation as a finite union of relations. We consider a set of infinite sequences over the program states that is induced by this non-well-founded relation in the following way. Given a pair of states (s, s') from the relation, we choose a computation segment that "connects" the states s and s', *i.e.* whose first and last states are s and s' respectively. We obtain an infinite sequence by concatenating the segment with itself infinitely many times. We consider all such infinite sequences that can be obtained by taking all possible connecting segments for each pair of states in the relation. We check whether these sequences satisfy the fairness requirements. If a fairness requirement is satisfied by some sequences from the set, then the label that corresponds to the fairness requirement is attached to the relation.

The above description does not immediately provide effective means to identify or synthesize labeled transition invariants. Thus, we introduce an *inductiveness principle* for labeled transition invariants. This principle allows one to identify a given set of labeled relations as an *inductive* labeled transition invariant. Testing the inductiveness amounts to subset inclusion tests between binary relations over states, and between sets of labels.

We illustrate the proposed proof rule on interesting examples of concurrent programs. We consider the program CORR-ANY-DOWN whose termination relies on the eventual reliability of a lossy and corrupting communication channel. The eventual reliability is modeled by a fairness requirement. We also consider two examples of mutual exclusion protocols, namely, MUX-BAKERY and MUX-TICKET. For each protocol, we prove the non-starvation property, *i.e.* the accessibility of the critical section, for the first process. Fairness requirements are needed to deal with the process idling.

Contributions In this chapter, we make the following contributions. We propose a sound and relatively complete proof rule for the verification of termination/temporal properties under fairness requirements imposed on sets of states that accounts for the fairness requirements directly. We account for specification automata, which we use to encode general temporal properties, equipped with the Büchi, the generalized Büchi, and the Streett acceptance conditions in a uniform way. Thus, our method allows one to use specification automata with the generalized Büchi and the Streett acceptance condition, which in general have fewer states and a simpler underlying structure than the equivalent Büchi automata.

We propose an automated method for the synthesis of labeled transition invariants (*i.e.* the intermediate assertions in our proof rule) by abstract interpretation, which leads to the automation of the proof rule.

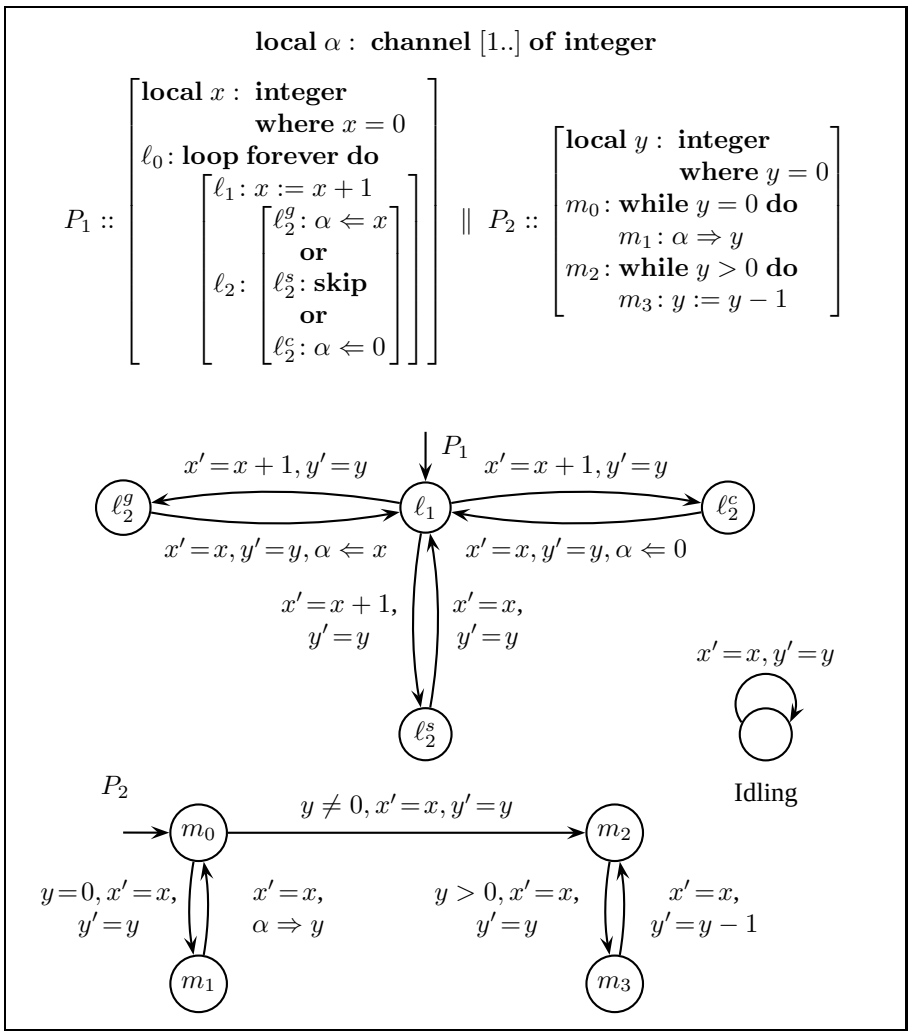

Figure 3.1: Program CORR-ANY-DOWN.

Examples We make the examples more interesting by admitting an idling transition at each program location. We show control-flow graphs for each program. The idling transitions are implicit in the program text, but are explicitly shown on the control-flow graphs. For presentation purposes, we simplify the control-flow graphs by composing straight-line code segments to single transitions. In the rest of the chapter we consider the simplified versions of the programs.

For each program, we show the fairness requirements, and give a (non-inductive) labeled transition invariant with a corresponding informal justification in Sections 3.2 resp. 3.4; the corresponding formal argument is based on a stronger inductive labeled transition invariant, which we present in Section 3.5.

CORR-ANY-DOWN The program shown on Figure 3.1 is a modification of the program ANY-DOWN from Chapter 1. The communication between the processes takes

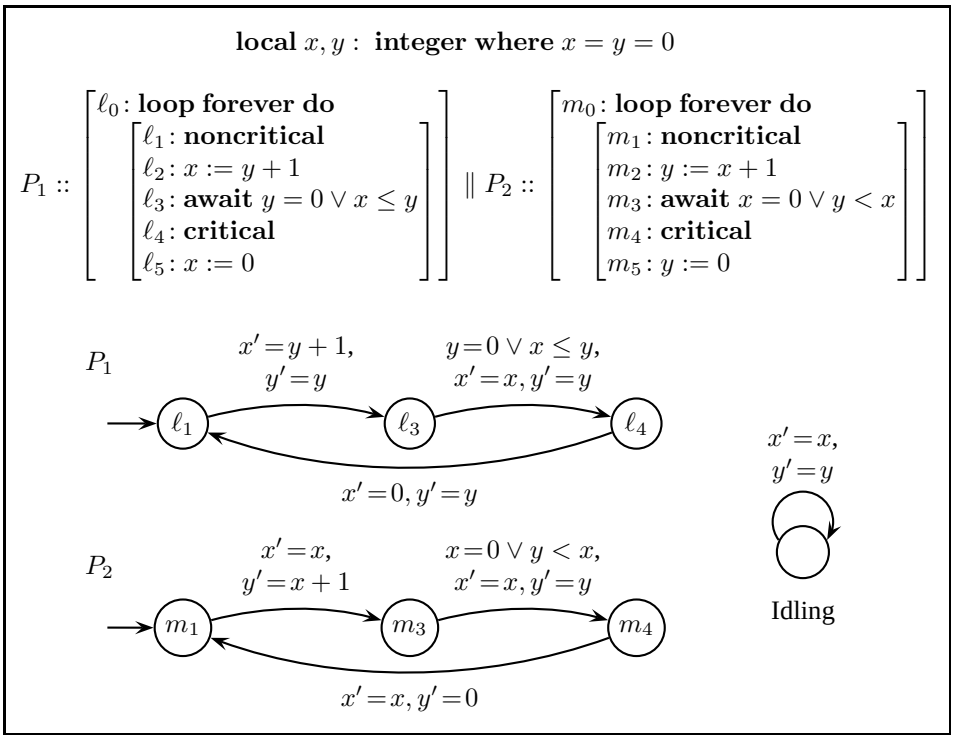

Figure 3.2: Program MUX-BAKERY.

place over an asynchronous channel α. The channel α is unreliable. Messages sent over the channel can be transmitted correctly, get lost or corrupted during the transmission. The transition $\alpha \Leftarrow x$ models a correct transmission, **skip** models the message loss, and $\alpha \Leftarrow 0$ models the message corruption [34]. The temporal property we wish to prove is termination under the assumption that the second process cannot stay forever in the location m_2 when $y \leq 0$.

The program termination relies on the assumption that the value of the variable x is eventually communicated to the variable y, *i.e.*, that the channel α is eventually reliable. We model this assumption by a compassion requirement that ensures a successful transmission if there are infinitely many attempts to send a message.

The eventual reliability of the communication channel is in fact not sufficient for proving termination. We also need to exclude computations in which one of the processes idles forever when one of its transitions can be taken. Hence, we introduce a justice requirement for each transition.

MUX-BAKERY The program MUX-BAKERY on Figure 3.2 is a simplified version [33] of the Bakery mutual exclusion protocol [27]. The temporal property we wish to verify is the starvation freedom for the first process. This means that whenever P_1 leaves the non-critical section, it will eventually reach the critical section. The property relies on justice assumptions that every continuously enabled transition will be eventually taken.

MUX-TICKET The program MUX-TICKET on Figure 3.3 is another mutual exclusion protocol. We verify the starvation freedom property for the first process. It requires the same kind of fairness requirements as the program MUX-BAKERY.

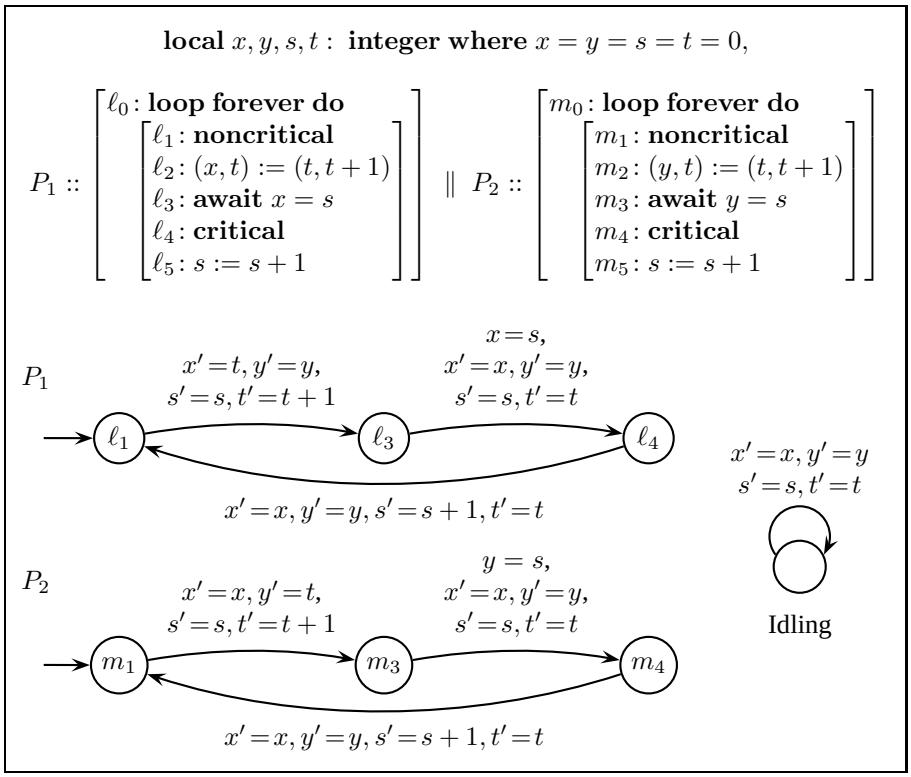

Figure 3.3: Program MUX-TICKET.

3.2 Labeled Transition Invariants

Before introducing labeled transition invariants, we formalize fairness requirements imposed on sets of states.

We fix a program $P = \langle \Sigma, \Theta, \mathcal{T} \rangle$. Let

$$\mathcal{J} \;=\; \{J_1, \ldots, J_k\},$$

such that $J_i \subseteq \Sigma$ for each $i \in \{1, \ldots, k\}$, be a set of *justice* requirements. Let

$$\mathcal{C} \;=\; \{\langle p_1, q_1 \rangle, \ldots, \langle p_m, q_m \rangle\},$$

such that $p_i, q_i \subseteq \Sigma$ for each $i \in \{1, \ldots, m\}$, be a set of *compassion* requirements.

A computation $\sigma = s_1, s_2, \ldots$ satisfies the set of justice requirements $\mathcal{J}$ when for each $J \in \mathcal{J}$ there exist infinitely many positions i in σ such that $s_i \in J$. The computation σ satisfies the set of compassion requirements $\mathcal{C}$ when for each $\langle p, q \rangle \in \mathcal{C}$ either σ contains only finitely many positions i such that $s_i \in p$, or σ contains infinitely many positions j such that $s_j \in q$.

We observe that justice requirements can be translated into compassion requirements as follows. For every justice requirement J we extend the set of compassion requirements by the pair $\langle \Sigma, J \rangle$. We assume that all justice requirements are translated into the compassion requirements, and that the set of compassion requirements $\mathcal{C}$ contains the translated justice requirements. A specialization of the notions presented in this chapter for an explicit treatment of justice requirements is straightforward.

Let $|\mathcal{C}|$ be the set of the indices of all compassion requirements.

$$|\mathcal{C}| \;=\; \{1, \ldots, m\}$$

We define *labeled relations*, which we will use as building blocks for labeled transition invariants.

Definition 3.1 (Labeled Relation) *A* labeled relation (T, M) *consists of a binary relation* $T \subseteq \Sigma \times \Sigma$ *and a set of indices (*labels*)* $M \subseteq |\mathcal{C}|$*. The labeled relation* (T, M) captures *a segment* $s_1, \ldots, s_n$ *if we have:*

- $(s_1, s_n) \in T$*, and*
- *if the infinite sequence* $(s_1, \ldots, s_n)^\omega$*,* i.e. *the concatenation of the segment* $s_1, \ldots, s_n$ *with itself infinitely many times, satisfies a compassion requirement* $\langle p_i, q_i \rangle$*, where* $i \in |\mathcal{C}|$*, then the index* i *is an element of* M*.*

We write $\mathsf{seg}(T, M)$ *for the set of all computation segments that are captured by the labeled relation* (T, M)*.*

We define *labeled transition invariants* that contain an explicit encoding of the satisfaction of compassion requirements.

Definition 3.2 (Labeled Transition Invariant) *A* labeled transition invariant L *is a finite set of labeled relations such that every computation segment is captured by some labeled relation in* L*.*

We will give a characterization of termination under compassion requirements using labeled transition invariants in Section 3.3. Now we show a labeled transition invariant for the first program presented in the introduction to this chapter.

CORR-ANY-DOWN First, we describe how we model the asynchronous communication channel α by an integer array of infinite size. We keep track of the positions in the array at which the read and write operations take place, as well as the position at which the first successfully transmitted value is written.

Let the variable $w(rite)$ ranging over the positive integers denote the position at which the next transmission transition, either correct or corrupting, will put a message into the channel. Let the variable r (for *read*) denote the position from which the next read transition will read a message from the channel. The channel contains unread messages, *i.e.*, the transition $\alpha \Rightarrow y$ can be taken, if $r < w$. Both w and r are initialized by 1. We use the variable $v(alue)$ to store the first value that is successfully sent over α, which is called the "good" value. The variable $p(osition)$ stores the position at which the "good" value is stored in the channel. Initially, both v and p contain the value 0. The resulting translation of the communication transitions into transitions that manipulate the variables r, w, v, and p is shown in Table 3.1

Transition	Translation	Comment
$\alpha \Leftarrow x$	**if** $v = 0$ **then** $(v, p, w) := (x, w, w + 1)$	first transmission
	else $w := w + 1$	other transmissions
$\alpha \Leftarrow 0$	$w := w + 1$	corrupted transmission
$\alpha \Rightarrow y$	**if** $r \geq w$ **then await**	nothing to read
	else if $r = p$ **then** $(y, r) := (v, r + 1)$	read the "good" value
	else $r := r + 1$	read other value

Table 3.1: Modeling of the asynchronous communication channel α.

The following set of justice requirements excludes computations in which one of the processes idles forever when one of its transitions can be taken.

$$\begin{aligned}\mathcal{J} \;=\; \{&\neg at\_\ell_1,\ \neg at\_\ell_2^g,\ \neg at\_\ell_2^s,\ \neg at\_\ell_2^c,\\ &\neg(at\_m_0 \wedge y = 0),\ \neg(at\_m_0 \wedge y \neq 0),\ \neg(at\_m_1 \wedge r < w),\\ &\neg(at\_m_2 \wedge y > 0),\ \neg at\_m_3\}\end{aligned}$$

We extend $\mathcal{J}$ with the justice requirement $\neg(at\_m_2 \wedge y \leq 0)$ that encodes our assumption that the second process cannot stay forever in the location m_2 when $y \leq 0$. We assume that not all of the sent messages are either lost or corrupted, *i.e.*, that the transmitting transition at the location ℓ_s^g is not ignored forever. We model this assumption by the following compassion requirement.

$$\mathcal{C} \;=\; \{\langle at\_\ell_1,\ at\_\ell_2^g\rangle\}$$

After translation of each justice requirement into a compassion requirement, we obtain eleven compassion requirements (including the compassion requirement shown above).

The set of the labeled relations below is a labeled transition invariant for the program CORR-ANY-DOWN.

$$\begin{aligned}
L_1 &= (v = 0 \wedge v' > 0, && \{1, \ldots, 11\})\\
L_2 &= (r = w \wedge r' = r \wedge w' > w, && \{1, \ldots, 11\})\\
L_3 &= (r \leq p \wedge r' > r \wedge p' = p, && \{1, \ldots, 11\})\\
L_4 &= (y > 0 \wedge y' < y, && \{1, \ldots, 11\})\\
L_i^5 &= (\mathrm{T}, && \{1, \ldots, 11\} \setminus \{i\}) \text{ for } i \in \{1, \ldots, 11\}\\
L_{ij}^6 &= (at_\ell_i \wedge at'_\ell_j, && \{1, \ldots, 11\}) \text{ for } i \neq j \in \{1, \tfrac{g}{2}, \tfrac{s}{2}, \tfrac{c}{2}\}\\
L_{ij}^7 &= (at_m_i \wedge at'_m_j, && \{1, \ldots, 11\}) \text{ for } i \neq j \in \{0, \ldots, 3\}
\end{aligned}$$

All computation segments that are loops wrt. location labels and whose infinite concatenations may satisfy all compassion requirements are captured by the labeled relations L_1, L_2, L_3, and L_4. The first three labeled relations capture the computation segments that start at the locations m_0 or m_1. The labeled relation L_1 captures the segments that contain the first successful transmission. L_2 captures the segments in which unread messages appear in the channel. L_3 contains the segments on which the second process reads corrupted messages from the channel until it reaches the "good" one.

The labeled relation L_4 captures the segments that start at the locations m_2 or m_3. The value of the variable y decreases until it reaches 0. The labeled relations L_i^5, where

$i \in \{1, \ldots, 11\}$, capture all segments whose infinite concatenations does not satisfy all compassion requirements.

All other computation segments are captured by the labeled relations L^6_{ij}, where $i \neq j \in \{1, \frac{g}{2}, \frac{s}{2}, \frac{c}{2}\}$, and L^7_{ij}, where $i \neq j \in \{0, \ldots, 3\}$.

3.3 Termination under Compassion

We give a *direct* characterization of termination under compassion requirements via labeled transition invariants, *i.e.*, a translation of the compassion requirements into a Büchi automaton and an application of the automata-theoretic framework of [51] is not needed.

Theorem 3.1 (Termination under Compassion) *The program P terminates under the set of compassion requirements $\mathcal{C}$ if and only if there exists a labeled transition invariant L such that for every labeled relation (T, M) in L, either $|\mathcal{C}| \neq M$ or the relation T is well-founded.*

Proof. **if**-direction: For a proof by contraposition, assume that L is a labeled transition invariant such that for each $(T, M) \in L$ holds that either $|\mathcal{C}| \neq M$ or the relation T is well-founded, and that P does not terminate under the compassion requirements $\mathcal{C}$. We will show that there exists a labeled relation (T, M) in L such that the relation T is not well-founded and $|\mathcal{C}| = M$.

By the assumption that P does not terminate under $\mathcal{C}$, there exists an infinite computation $\sigma = s_1, s_2, \ldots$ that satisfies all compassion requirements.

We partition the set $|\mathcal{C}|$ of indices of compassion requirements into two subsets $|\mathcal{C}|^p$ and $|\mathcal{C}|^q$ as follows. An index j (of the compassion requirement $\langle p_j, q_j \rangle$) is an element of the subset $|\mathcal{C}|^p$ if there exist only finitely many positions i in σ such that $s_i \in p_j$; otherwise, j is an element of the subset $|\mathcal{C}|^q$. There exists a position r such that for each $i \geq r$ and for each $j \in |\mathcal{C}|^p$ we have $s_i \notin p_j$.

Let $H = h_1, h_2, \ldots$ be an infinite ordered set of positions in σ such that $h_1 = r$ and for each $i \geq 1$ and for each $j \in |\mathcal{C}|^q$ there exist a position h between the positions h_i and h_{i+1} with $s_h \in q_j$. Since σ satisfies all compassion requirements such a set H exists.

For the fixed σ and the fixed H, we choose a function f that maps an ordered pair (k, l), where $k < l$, of indices in H to one of the labeled relations in the labeled transition invariants L as follows.

$$f(k, l) \quad = \quad (T, M) \in L \quad \text{such that } (s_k, \ldots, s_l) \in \mathsf{seg}(T, M)$$

Such a function f exists since L is a labeled transition invariant.

The function f induces an equivalence relation $\sim$ on ordered pairs of elements from H.

$$(k, l) \sim (k', l') \quad = \quad f(k, l) = f(k', l')$$

The equivalence relation $\sim$ has finite index since the range of f is finite.

By Ramsey's theorem [41], there exists an infinite ordered set of positions $K = k_1, k_2, \ldots$, where $k_i \in H$ for all $i \geq 1$, with the following property. All pairs of elements in K belong to the same equivalence class, say $[(m, n)]_\sim$ with $m, n \in K$. That is, for all $k, l \in K$ such that $k < l$ we have $(k, l) \sim (m, n)$. We fix m and n. Let (T_{mn}, M_{mn}) denote the labeled relation $f(m, n)$.

Since $(k_i, k_{i+1}) \sim (m, n)$ for all $i \geq 1$, the function f maps the pair (k_i, k_{i+1}) to (T_{mn}, M_{mn}) for all $i \geq 1$. Hence, the infinite sequence $s_{k_1}, s_{k_2}, \ldots$ is induced by the relation T_{mn}, *i.e.*,

$$(s_{k_i}, s_{k_{i+1}}) \in T_{mn}, \text{ for all } i \geq 1.$$

Hence, the relation T_{mn} is not well-founded.

By the choice of elements in H the following claims hold. For every $i \geq k_1$ and for every $j \in |\mathcal{C}|^p$ the state s_i is not an element of p_j. For every $i \geq 1$ and for every $j \in |\mathcal{C}|^q$ there exists a position k between the positions k_i and k_{i+1} such that $s_k \in q_j$. Hence, for every $i \geq 1$ the infinite sequence

$$(s_{k_i}, \ldots, s_{k_{i+1}})^\omega$$

satisfies all compassion requirements. We conclude $M_{mn} = |\mathcal{C}|$.

only if-direction: Assume that the program P terminates under the compassion requirements $\mathcal{C}$. Let L be a set of labeled relations defined as follows. For each subset M of $|\mathcal{C}|$ let (T, M) be a labeled relation in L such that a pair of states (s, s') is an element of the relation T if there exists a computation segment $s_1, \ldots, s_n$ such that $s_1 = s$, $s_n = s'$, and the following equality holds.

$$M = \{j \in |\mathcal{C}| \mid (s_1 \notin p_j \text{ and } \ldots \text{ and } s_n \notin p_j) \text{ or } s_1 \in q_j \text{ or } \ldots \text{ or } s_n \in q_j\}$$

Thus, for every computation segment $s_1, \ldots, s_n$ there exists a labeled relation $(T, M) \in L$ such that $(s_1, \ldots, s_n) \in \mathsf{seg}(T, M)$. Hence, L is a labeled transition invariant. Note that L contains only one relation that is labeled by the set of indices of all compassion requirements $\mathcal{C}$.

We show, by contraposition, that for the labeled relation $(T, |\mathcal{C}|)$ in L we have that the relation T is well-founded.

Assume that there exists an infinite sequence of states $s^1, s^2, \ldots$ such that (s^i, s^{i+1}) is an element of T for all $i \geq 1$, *i.e.*, the relation T is not well-founded. Since $s^1, \ldots, s^2$ is a computation segment, the state s^1 is accessible from some initial state $s_1 \in \Theta$. Furthermore, for all $i \geq 1$ there exists a computation segment $(s^i, \ldots, s^{i+1}) \in \mathsf{seg}(T, |\mathcal{C}|)$ connecting the states s^i and s^{i+1}. For connecting the states s^i and s^{i+1} we choose a computation segment in the set $\mathsf{seg}(T, |\mathcal{C}|)$ whose infinite concatenation satisfies all compassion requirements in $|\mathcal{C}|$. Such a segment exists by construction of $(T, |\mathcal{C}|)$. We conclude that there exists an infinite computation $\sigma = s_1, \ldots, s^1, \ldots, s^2, \ldots$. Next, we prove that σ satisfies all compassion requirements.

For each $j \in |\mathcal{C}|$, by the condition for the pair (s^i, s^{i+1}) to be an element of T, the following holds. Either there exists an index $r \geq 1$ such that for each $i \geq r$ the computation segment $s^i, \ldots, s^{i+1}$ does not have p_j-states, or there are infinitely many computation segments in which a p_j-state appears, and a q_j-state appears as well. Hence, σ satisfies all compassion requirements.

There is a contradiction to our assumption that P terminates under the compassion requirements $\mathcal{C}$. □

CORR-ANY-DOWN The labeled transition invariant shown in Section 3.2 satisfies the condition of Theorem 3.1. The labeled relations L_1, L_2, L_3, L_4, L^6_{ij}, and L^7_{ij} are well-founded. None of the labeled relations L^5_i needs to have a well-founded T-relation,

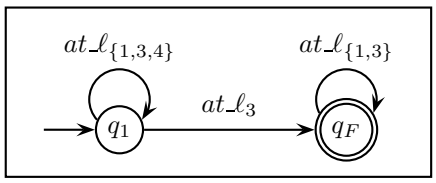

Figure 3.4: Büchi automaton for $\neg\mathrm{G}(at\_\ell_3 \rightarrow \mathrm{F}(at\_\ell_4))$.

since their labels does not contain the indices of all compassion requirements. Hence, the program CORR-ANY-DOWN is terminating under the assumptions that the communication channel is eventually reliable, and that enabled transitions are eventually taken.

3.4 Temporal Properties under Compassion

Given the program P, we verify a temporal property Ψ under the compassion requirements $\mathcal{C}$ by applying the automata-theoretic framework [51]. We assume that the property is given by a (possibly infinite-state) specification automaton $\mathcal{A}_\Psi$ that accepts exactly the infinite sequences of program states that violate the property Ψ. We do not encode the compassion requirements into the automaton.

Next, we give characterizations of the validity of the temporal property Ψ given by a specification automaton $\mathcal{A}_\Psi$ for the cases when $\mathcal{A}_\Psi$ is a Büchi automaton, a generalized Büchi automaton, and a Streett automaton.

Automaton $\mathcal{A}_\Psi$ with Büchi Acceptance Condition Let $\mathcal{A}_\Psi$ be a Büchi automaton with the set of states Q and the acceptance condition $F \subseteq Q$. Let the program $P|||\mathcal{A}_\Psi$ be the product of the synchronous parallel composition of P and $\mathcal{A}_\Psi$.

Remark 3.1 *The program P with the compassion requirements $\mathcal{C}$ satisfies the property Ψ given by the Büchi automaton $\mathcal{A}_\Psi$ if and only if the program $P|||\mathcal{A}_\Psi$ terminates under the compassion requirements $\mathcal{C}_{|||}$ shown below.*

$$\mathcal{C}_{|||} \;=\; \{\langle p \times Q, q \times Q\rangle \mid \langle p, q\rangle \in \mathcal{C}\} \cup \{\langle \Sigma \times Q, \Sigma \times F\rangle\} \tag{3.1}$$

We show labeled transition invariants for the programs $P|||\mathcal{A}_\Psi$, where P is the second resp. third program from the introduction, and the property Ψ is given by a Büchi automaton $\mathcal{A}_\Psi$.

MUX-BAKERY We encode the starvation freedom property for the first process by the temporal formula $\mathrm{G}(at\_\ell_3 \rightarrow \mathrm{F}(at\_\ell_4))$. A corresponding Büchi automaton for its negation is shown on Figure 3.4. The automaton has the set of states $\{q_1, q_F\}$. The state q_F is accepting.

First, we show a transition invariant T for the parallel composition of the program

with the automaton. The transition invariant T is the union of the relations below.

$$
\begin{aligned}
T_1 &= at\_q_1 \\
T_2 &= at\_q_F \wedge at\_\ell_3 \wedge y = 0 \wedge x' = x \wedge y' = y \\
T_3 &= at\_q_F \wedge at\_\ell_3 \wedge x \leq y \wedge x' = x \wedge y' = y \\
T_4 &= at\_q_F \wedge at\_m_3 \wedge y < x \wedge x' = x \wedge y' = y \\
T_5 &= at\_q_F \wedge at\_m_4 \wedge x' = x \wedge y' = y \\
T_6 &= at\_q_F \wedge y < x \wedge x' = x \wedge y' > x \\
T^7_{ij} &= at\_q_i \wedge at'\_q_j && \text{for } i \neq j \in \{1, F\} \\
T^8_{ij} &= at\_\ell_i \wedge at'\_\ell_j && \text{for } i \neq j \in \{1, 3, 4\} \\
T^9_{ij} &= at\_m_i \wedge at'\_m_j && \text{for } i \neq j \in \{1, 3, 4\}
\end{aligned}
$$

All computation segments that are loops wrt. the location labels and do not visit the Büchi accepting state are captured by the relation T_1. All other loops, which visit the Büchi accepting state, are captured by the relations T_2, T_3, T_4, T_5, and T_6 as follows. Loops that are induced by idling when one of the transitions is enabled are captured by the relations T_2, T_3, T_4, and T_5. The location ℓ_3 cannot be left while staying in the Büchi accepting state because the only way to leave ℓ_3 is via the location ℓ_4, which is excluded by the transition relation of the Büchi automaton. Hence, the idling in the locations ℓ_1 and ℓ_4 is not possible in the Büchi accepting state. Loops that are induced by idling at the location m_1 are captured by the relation T_2, since in this case, when staying in the Büchi accepting state, the first process is in the location ℓ_3 and the value of the variable y is 0. The relation T_6 captures the loops where the first process becomes enabled for entering the critical section.

The relations T^7_{ij} where $i \neq j \in \{1, F\}$, T^8_{ij} and T^9_{ij} where $i \neq j \in \{1, 3, 4\}$ capture computation segments that are not loops wrt. the location labels of either the Büchi automaton or one of the processes.

We observe that the relations $T_1, \ldots, T_5$ are not well-founded. Hence, we cannot prove that the product program terminates by applying Theorem 1.1. We show that these relations capture computation segments whose infinite concatenations violate some fairness requirements, which we describe next, and, hence, their non-well-foundedness can be safely ignored.

The following set of justice requirements excludes computation of the program MUX-BAKERY in which one of the processes idles forever in any but the non-critical location when one of its transitions can be taken.

$$
\begin{aligned}
\mathcal{J} \;=\; & \{\neg(at\_\ell_3 \wedge (y = 0 \vee x \leq y)),\ \neg at\_\ell_4 \\
& \neg(at\_m_3 \wedge (x = 0 \vee y < x)),\ \neg at\_m_4\}
\end{aligned}
$$

We translate $\mathcal{J}$ into a set of compassion requirements $\mathcal{C}$. Both $\mathcal{C}$ and the Büchi acceptance condition translate to the set of compassion requirements $\mathcal{C}_{|||}$ (as described by Equation (3.1)) that contains five requirements.

We observe that the relation T_1 captures all computation segments that do not visit the Büchi accepting state, thus violating the compassion requirement in $\mathcal{C}_{|||}$ that is induced by the Büchi acceptance condition, whose index is 5. Infinite concatenations of computation segments captured by the labeled relation T_2 and T_3 violate the compassion requirement in $\mathcal{C}_{|||}$ that is induced by the first justice requirement (indexed by 1). Analogous observations hold for the relations T_4 and T_5 together with the third and the

fourth justice requirement respectively. We show below a labeled transition invariant L for the parallel composition of the program and the automaton.

$$\begin{array}{rcll} L_1 & = & (T_1, & \{1,2,3,4\}) \\ L_2 & = & (T_2, & \{2,3,4,5\}) \\ L_3 & = & (T_3, & \{2,3,4,5\}) \\ L_4 & = & (T_4, & \{1,2,4,5\}) \\ L_5 & = & (T_5, & \{1,2,3,5\}) \\ L_6 & = & (T_6, & \{1,\ldots,5\}) \\ L^7_{ij} & = & (T^7_{ij}, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,F\} \\ L^8_{ij} & = & (T^8_{ij}, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,3,4\} \\ L^9_{ij} & = & (T^9_{ij}, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,3,4\} \end{array}$$

By Theorem 3.1 and Remark 3.1, the program MUX-BAKERY satisfies the non-starvation property for the first process, since the relations T_6, T^7_{ij}, T^8_{ij}, and T^9_{ij}, which are labeled by the set of indices of all compassion requirement, are well-founded.

MUX-TICKET We prove that the first process in MUX-TICKET satisfies the non-starvation property $\mathrm{G}(at\_\ell_3 \rightarrow \mathrm{F}(at\_\ell_4))$ (see Figure 3.4 for the corresponding Büchi automaton) under the following set of justice requirements.

$$\begin{array}{rcl} \mathcal{J} & = & \{\neg(at\_\ell_3 \wedge x = s),\ \neg at\_\ell_4, \\ & & \ \ \neg(at\_m_3 \wedge y = s),\ \neg at\_m_4\} \end{array}$$

The set of the labeled relations below is a labeled transition invariant for the parallel composition MUX-TICKET with the Büchi automaton.

$$\begin{array}{rcll} L_1 & = & (at\_q_1, & \{1,2,3,4\}) \\ L_2 & = & (at\_\ell_3 \wedge x = s \wedge x' = x \wedge s' = s, & \{2,3,4,5\}) \\ L_3 & = & (at\_m_3 \wedge y = s \wedge y' = y \wedge s' = s, & \{1,2,4,5\}) \\ L_4 & = & (at\_m_4 \wedge x' = x \wedge y' = y \wedge s' = s, & \{1,2,3,5\}) \\ L_5 & = & (s < x \wedge x' = x \wedge s' > s, & \{1,\ldots,5\}) \\ L^6_{ij} & = & (at\_q_i \wedge at'\_q_j, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,F\} \\ L^7_{ij} & = & (at\_\ell_i \wedge at'\_\ell_j, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,3,4\} \\ L^8_{ij} & = & (at\_m_i \wedge at'\_m_j, & \{1,\ldots,5\}) \text{ for } i \neq j \in \{1,3,4\} \end{array}$$

The labeled relations L_1, L_2, L_3, L_4, and L_5 capture computation segments that are loops wrt. the location labels; the justification is similar to the example MUX-BAKERY. All computation segments that are not loops are captured by the labeled relations L^6_{ij} where $i \neq j \in \{1,F\}$, L^7_{ij} and L^8_{ij} where $i \neq j \in \{1,3,4\}$.

Automaton $\mathcal{A}_\Psi$ with Generalized Büchi Acceptance Condition Generalized Büchi automata are automata on infinite words equipped with multiple sets of accepting states. We account for the generalized Büchi acceptance condition directly, by translating it into a set of justice requirements. Since we do not translate the automaton into a degeneralized one, we avoid the corresponding increase of the automaton

size. The characterization of the validity of a temporal property given by a generalized Büchi automaton follows the lines of the previous paragraph (dealing with "plain" Büchi automata).

Automaton $\mathcal{A}_\Psi$ with Streett Acceptance Condition The Streett acceptance condition is a finite collection of pairs $\{(L_i, U_i) \mid i \in I\}$ indexed by I such that $L_i, U_i \subseteq Q$ for all $i \in I$. The automaton accepts a word σ if it has a run $q_1, q_2, \ldots$ on σ such that for every $i \in I$, if there are infinitely many j's such that $q_j \in L_i$ then there are infinitely many j's such that $q_j \in U_i$. We note a direct relationship between Streett acceptance conditions and compassion requirements.

A characterization of the validity of a temporal property given by a Streett automaton $\mathcal{A}_\Psi$ is similar to the case when $\mathcal{A}_\Psi$ is a Büchi automaton. The translation of the Streett acceptance condition into a set of compassion requirements for the synchronous parallel composition of the program P with the Streett automaton $\mathcal{A}_\Psi$ is straightforward.

3.5 Proof Rule

In this section, we formulate a proof rule for the verification of temporal properties of concurrent programs under compassion requirements. The proof rule is based of inductive labeled transition invariants, and accounts for the compassion requirements in an explicit way.

First, we define the following auxiliary functions that map sets of program states into sets of indices of compassion requirements. For a set of states $S \subseteq \Sigma$ we have

$$\begin{aligned} \mathsf{None}(S) &= \{j \in |\mathcal{C}| \mid S \cap p_j = \emptyset\}, \\ \mathsf{Some}(S) &= \{j \in |\mathcal{C}| \mid S \cap q_j \neq \emptyset\}. \end{aligned}$$

We refine the notion of labeled relation for a more precise accounting of compassion requirements.

Definition 3.3 (Labeled Relation (Refined)) *A* labeled relation (T, P, Q) *consists of a binary relation $T \subseteq \Sigma \times \Sigma$ and two sets of indices (*labels*) $P, Q \subseteq |\mathcal{C}|$. The labeled relation* (T, P, Q) captures *a computation segment $s_1, \ldots, s_n$ if $(s_1, s_n) \in T$ and*

$$\begin{aligned} \mathsf{None}(\{s_1, \ldots, s_n\}) &\subseteq P, \\ \mathsf{Some}(\{s_1, \ldots, s_n\}) &\subseteq Q. \end{aligned}$$

We write $\mathsf{seg}(T, P, Q)$ *for the set of all computation segments that are captured by the labeled relation* (T, P, Q).

From now on, we use the refined version of labeled relations.

We define the ordering $\trianglelefteq$ on labeled relations. We have

$$(T_1, P_1, Q_1) \trianglelefteq (T_2, P_2, Q_2)$$

if the following three conditions hold.

$$T_1 \subseteq T_2, \qquad P_1 \subseteq P_2, \qquad Q_1 \subseteq Q_2$$

We introduce the ordering $\trianglelefteq$ for a practical reason. Testing whether $(T_1, P_1, Q_1) \trianglelefteq (T_2, P_2, Q_2)$ holds amounts to entailment tests between relations and sets of indices

vs. entailment tests between implicitly denoted sets of computation segments that are needed for checking $\mathsf{seg}(T_1, P_1, Q_1) \subseteq \mathsf{seg}(T_2, P_2, Q_2)$. Note that the ordering $\trianglelefteq$ approximates the subset inclusion ordering between the sets of computation segments captured by labeled relations, as we formalize in Remark 3.2.

Remark 3.2 *The relation $\trianglelefteq$ is an approximation of the entailment relation between the sets of computation segments that are captured by two labeled relations. Formally,*

$$(T_1, P_1, Q_1) \trianglelefteq (T_2, P_2, Q_2) \implies \mathsf{seg}(T_1, P_1, Q_1) \subseteq \mathsf{seg}(T_2, P_2, Q_2).$$

We canonically extend the ordering $\trianglelefteq$ to sets of labeled relations, *i.e.*, we have

$$\{(T_i, P_i, Q_i) \mid i \in I\} \trianglelefteq \{(T_j, P_j, Q_j) \mid j \in J\}$$

if the following condition holds.

$$\forall i \in I\ \exists j \in J.\ (T_i, P_i, Q_i) \trianglelefteq (T_j, P_j, Q_j)$$

We canonically extend the functions None and Some to binary relations. Given a relation $T \subseteq \Sigma \times \Sigma$, the extension yields the following.

$$\begin{aligned} \mathsf{None}(T) &= \bigcup_{(s_1,s_2)\in T} \mathsf{None}(\{s_1, s_2\}) \\ \mathsf{Some}(T) &= \bigcup_{(s_1,s_2)\in T} \mathsf{Some}(\{s_1, s_2\}) \end{aligned}$$

We define a *labeled* composition operator $\boxdot$ that composes labeled relations (T, P, Q) with transition relations ρ_τ. The product of the composition is a labeled relation. The symbol $\circ$ denotes the relational composition operator.

$$(T, P, Q) \boxdot \rho_\tau = (T \circ \rho_\tau, P \cap \mathsf{None}(T \circ \rho_\tau), Q \cup \mathsf{Some}(T \circ \rho_\tau))$$

The following lemma indicates that the labeled composition is 'compatible' with the relational composition operator.

Lemma 3.1 *Every extension of a computation segment that is captured by a labeled relation (T, P, Q) by a segment consisting of a pair of states in a transition relation ρ_τ is captured by the labeled composition of (T, P, Q) and ρ_τ. Formally,*

$$\begin{aligned} (s_1, \ldots, s_n) \in \mathsf{seg}(T, P, Q) \text{ and } (s_n, s_{n+1}) \in \rho_\tau \implies \\ (s_1, \ldots, s_n, s_{n+1}) \in \mathsf{seg}((T, P, Q) \boxdot \rho_\tau). \end{aligned}$$

Proof. Let $s_1, \ldots, s_n$ be a computation segment that is captured by the labeled relation (T, P, Q), and let (s_n, s_{n+1}) be an element of the transition relation ρ_τ. By the definition of labeled relations, for the set of indices of compassion requirements

$$P_n = \mathsf{None}(\{s_1, \ldots, s_n\})$$

we have $P_n \subseteq P$. Furthermore, for the set of indices

$$P_{n+1} = \mathsf{None}(\{s_1, \ldots, s_n, s_{n+1}\})$$

holds $P_{n+1} \subseteq \mathsf{None}(\{s_1, s_{n+1}\}) \subseteq \mathsf{None}(T \circ \rho_\tau)$ and $P_{n+1} \subseteq P_n$. Hence, we have $P_{n+1} \subseteq P$ and $P_{n+1} \subseteq \mathsf{None}(T \circ \rho_\tau)$. We conclude $P_{n+1} \subseteq P \cap \mathsf{None}(T \circ \rho_\tau)$. Analogously, we have

$$\mathsf{Some}(\{s_1, \ldots, s_n\}) \subseteq Q,$$

and, hence, for the set of indices

$$Q_{n+1} \;=\; \mathsf{Some}(\{s_1, \ldots, s_n, s_{n+1}\})$$

holds $Q_{n+1} \subseteq Q \cup \mathsf{Some}(T \circ \rho_\tau)$. The pair of states (s_1, s_{n+1}) is an element of the relational composition $T \circ \rho_\tau$, since (s_1, s_n) is an element of the relation T. We conclude that $(s_1, \ldots, s_n, s_{n+1})$ is captured by $(T, P, Q) \circledcirc \rho_\tau$. □

Definition 3.4 (Inductive Labeled Relations) *A set of labeled relations L is* inductive *for the program P with the set of transitions $\mathcal{T}$ and the set of compassion requirements $\mathcal{C}$ if the following two conditions hold.*

$$\{(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau)) \mid \tau \in \mathcal{T}\} \;\trianglelefteq\; L$$
$$\{(T, P, Q) \circledcirc \rho_\tau \mid (T, P, Q) \in L \text{ and } \tau \in \mathcal{T}\} \;\trianglelefteq\; L$$

Remark 3.3 *We obtain a weaker definition of inductive labeled relations by restricting the transition relations ρ_τ in the first condition of Definition 3.4 to the accessible states Acc.*

Next, we prove that an inductive set of labeled relations is a labeled transition invariant. We will call such labeled transition invariants *inductive*.

Theorem 3.2 *An inductive set of labeled relations L for the program P is a labeled transition invariant for P.*

Proof. Given an inductive set of labeled relations L, we show that every computation segment $s_1, \ldots, s_n$ is captured by some labeled relation in L by induction over the segment length.

Let s_1, s_2 such that $(s_1, s_2) \in \rho_\tau$, where τ is a program transition, be a computation segment. From the inclusions $\mathsf{None}(\{s_1, s_2\}) \subseteq \mathsf{None}(\rho_\tau)$ and $\mathsf{Some}(\{s_1, s_2\}) \subseteq \mathsf{Some}(\rho_\tau)$ follows directly that the segment s_1, s_2 is captured by the labeled relation $(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau))$. By Remark 3.2, we have that the segment s_1, s_2 is captured by some labeled relation in L, which is $\trianglelefteq$-greater than $(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau))$.

The induction assumption is that the computation segment $s_1, \ldots, s_n$ is captured by a labeled relation (T, P, Q) from L. Let (s_n, s_{n+1}) be an element of ρ_τ. By Lemma 3.1, we have $(s_1, \ldots, s_n, s_{n+1}) \in \mathsf{seg}((T, P, Q) \circledcirc \rho_\tau)$. By Remark 3.2, the segment $s_1, \ldots, s_{n+1}$ is captured by some labeled relation in L, which is $\trianglelefteq$-greater than $(T, P, Q) \circledcirc \rho_\tau$. □

For legibility, we split the proof rule for the verification of temporal properties into two (specific) ones. The first proof rule deals with termination, the second one deals with (general) temporal properties.

Program P with:
set of states Σ,
set of transitions $\mathcal{T}$,
set of compassion requirements $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots, \langle p_m, q_m\rangle\}$,

Set $|\mathcal{C}| = \{1, \ldots, m\}$,

Set of labeled relations $L = \{(T_1, P_1, Q_1), \ldots, (T_n, P_n, Q_n)\}$ with:
$T_i \subseteq \Sigma \times \Sigma$ and $P_i, Q_i \subseteq |\mathcal{C}|$ for all $i \in \{1, \ldots, n\}$

P1: $\{(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau)) \mid \tau \in \mathcal{T}\} \trianglelefteq L$
P2: $\{(T, P, Q) \boxdot \rho_\tau \mid (T, P, Q) \in L \text{ and } \tau \in \mathcal{T}\} \trianglelefteq L$
P3: T_i well-founded or $|\mathcal{C}| \neq P_i \cup Q_i$ for each $i \in \{1, \ldots, n\}$

P terminates under compassion requirements $\mathcal{C}$

Figure 3.5: Rule COMP-TERM: termination under compassion requirements.

Proof Rule COMP-TERM Theorems 3.1 and 3.2 give rise to a proof rule COMP-TERM for termination under compassion requirements, shown on Figure 3.5.

Theorem 3.3 *The rule* COMP-TERM *is sound and semantically complete.*

Proof. First, we prove the soundness. Let L be a set of labeled relations that satisfies all premises of the proof rule COMP-TERM. Let $L^{u(nrefined)}$ be a set of unrefined labeled relations (recall Definition 3.1) defined as follows.

$$L^u \;=\; \{(T, P \cup Q) \mid (T, P, Q) \in L\}$$

We observe that each computation segment $s_1, \ldots, s_n$ that is captured by (T, P, Q) is also captured by $(T, P \cup Q)$, since the set of compassion requirements that are satisfied by the infinite concatenation $(s_1, \ldots, s_n)^\omega$ is a subset of $P \cup Q$. Since L satisfies the premises P1 and P2, by Theorem 3.2, we have that L is an inductive transition invariants. Hence, the set $L^{u(nrefined)}$ captures all computation segments as well, *i.e.*, it is a unrefined labeled transition invariants (recall Definition 3.2). By premise P3, for every unrefined labeled relation $(T, P \cup Q)$ in L^u such that $|\mathcal{C}| = P \cup Q$ we have that the relation T is well-founded. By Theorem 3.1, the program P terminates under the compassion requirements $\mathcal{C}$.

Now we prove the semantic completeness. We assume that the program terminates under the compassion requirements $\mathcal{C}$. We construct a set L of labeled relations that satisfies all premises of the proof rule COMP-TERM. Let L be a set of labeled relations defined as follows. For each pair of sets of indices $P \subseteq |\mathcal{C}|$ and $Q \subseteq |\mathcal{C}|$ let (T, P, Q) be a labeled relation in L such that a pair of states (s, s') is an element of the relation T if there exists a computation segment $s_1, \ldots, s_n$ such that $s_1 = s$, $s_n = s'$, $P = \mathsf{None}(\{s_1, \ldots, s_n\})$, and $Q = \mathsf{Some}(\{s_1, \ldots, s_n\})$.

We observe that for every computation segment $s_1, \ldots, s_n$ there exists a labeled relation $(T, P, Q) \in L$ such that $(s_1, \ldots, s_n) \in \mathsf{seg}(T, P, Q)$. Hence, L is a labeled transition invariant.

The proof that for each labeled relation (T, P, Q) in L either $|\mathcal{C}| \neq P \cup Q$ or the relation T is well-founded follows the lines of the corresponding part of the proof of Theorem 3.1.

We prove that the labeled transition invariant L is inductive. We make the following assumptions on the transition relations ρ_τ, where $\tau \in \mathcal{T}$.

Assumption 3.1 *For every pair (s_1, s_2) of states in the transition relation ρ_τ, where $\tau \in \mathcal{T}$, the sequence s_1, s_2 is a computation segment.*

This assumption is not a proper restriction. Theoretically, we can always restrict the transition relation to the accessible states. Alternatively, we may use a weaker version of the proof rule whose first premise restricts the transition relations to the accessible states (see Remark 3.3).

Assumption 3.2 *For each transition $\tau \in \mathcal{T}$ there exists two sets of indices P and Q of compassion requirements such that for every pair (s_1, s_2) of states in ρ_τ we have $P = \mathsf{None}(\{s_1, s_2\})$ and $Q = \mathsf{Some}(\{s_1, s_2\})$.*

This assumption can be established by splitting every transition relation according to the sets that appear in the fairness requirements, analogously to the procedure described in Section 2.8.

First, we show that for every program transition $\tau \in \mathcal{T}$ the condition $(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau)) \trianglelefteq (T, P, Q)$ holds for the labeled relation $(T, P, Q) \in L$ such that $P = \mathsf{None}(\rho_\tau)$ and $Q = \mathsf{Some}(\rho_\tau)$. We need to prove $\rho_\tau \subseteq T$. For every pair of states (s, s') in ρ_τ the sequence s, s' is a computation segment, by Assumption 3.1. Furthermore, we have $\mathsf{None}(\{s, s'\}) = P$ and $\mathsf{Some}(\{s, s'\}) = Q$, by Assumption 3.2. Hence, by construction of the labeled relation (T, P, Q), the pair (s, s') is an element of the relation T.

Next, we show that for every labeled relation $(T_1, P_1, Q_1) \in L$ and for every transition $\tau \in \mathcal{T}$ holds

$$(T_1, P_1, Q_1) \circledcirc \rho_\tau \trianglelefteq (T_2, P_2, Q_2),$$

where (T_2, P_2, Q_2) is the labeled relation in L such that $P_2 = P_1 \cap \mathsf{None}(T_1 \circ \rho_\tau)$ and $Q_2 = Q_1 \cup \mathsf{Some}(T_1 \circ \rho_\tau)$. Again, we need to prove $T_1 \circ \rho_\tau \subseteq T_2$.

We note the following auxiliary statement. For every pair (s, s') of states in T_1 we have

$$\begin{array}{ll} P_1 \subseteq \mathsf{None}(\{s\}) & \mathsf{Some}(\{s\}) \subseteq Q_1 \\ P_1 \subseteq \mathsf{None}(\{s'\}) & \mathsf{Some}(\{s'\}) \subseteq Q_1. \end{array}$$

To justify the statement above for the pair $(s, s') \in T_1$, we consider a computation segment $s, \ldots, s'$ that is captured by (T_1, P_1, Q_1) such that $\mathsf{None}(\{s, \ldots, s'\}) = P_1$ and $\mathsf{Some}(\{s, \ldots, s'\}) = Q_1$, which exists by construction of (T_1, P_1, Q_1). From the definitions of None and Some, our auxiliary statement follows directly.

Now we are ready to prove $T_1 \circ \rho_\tau \subseteq T_2$. For a pair of states $(s_1, s_n) \in T_1$ there exists a computation segment $s_1, \ldots, s_n$ that is captured by the labeled relation (T_1, P_1, Q_1) such that $\mathsf{None}(\{s_1, \ldots, s_n\}) = P_1$ and $\mathsf{Some}(\{s_1, \ldots, s_n\}) = Q_1$, by construction of (T_1, P_1, Q_1). By Lemma 3.1, for a pair of states $s_n, s_{n+1} \in \rho_\tau$ the computation segment $s_1, \ldots, s_n, s_{n+1}$ is captured by the labeled relation $(T_1, P_1, Q_1) \circledcirc \rho_\tau$. Next, we prove the equalities

$$\begin{aligned} \mathsf{None}(\{s_1, \ldots, s_n, s_{n+1}\}) &= P_1 \cap \mathsf{None}(T_1 \circ \rho_\tau) \\ \mathsf{Some}(\{s_1, \ldots, s_n, s_{n+1}\}) &= Q_1 \cup \mathsf{Some}(T_1 \circ \rho_\tau), \end{aligned}$$

from which $(s_1, s_{n+1}) \in T_2$ follows directly, by construction of (T_2, P_2, Q_2). We follow the chain of observations below.

$$
\begin{aligned}
&\mathsf{None}(\{s_1, \ldots, s_n, s_{n+1}\}) \\
&= P_1 \cap \mathsf{None}(\{s_n, s_{n+1}\}) \\
&= P_1 \cap \mathsf{None}(\{s_n, s_{n+1}\}) \cap \bigcup_{\substack{(s,s')\in T_1 \\ (s',s'')\in \rho_\tau}} \mathsf{None}(\{s\}) && \text{since } P_1 \subseteq \mathsf{None}(\{s\}) \\
&= P_1 \cap \bigcup_{\substack{(s,s')\in T_1 \\ (s',s'')\in \rho_\tau}} (\mathsf{None}(\{s\}) \cap \mathsf{None}(\{s_n, s_{n+1}\})) \\
&= P_1 \cap \bigcup_{\substack{(s,s')\in T_1 \\ (s',s'')\in \rho_\tau}} (\mathsf{None}(\{s\}) \cap \mathsf{None}(\{s', s''\})) && \text{by Assumption 3.2} \\
&= \bigcup_{\substack{(s,s')\in T_1 \\ (s',s'')\in \rho_\tau}} (\mathsf{None}(\{s, s''\}) \cap \mathsf{None}(\{s'\}) \cap P_1) \\
&= \bigcup_{\substack{(s,s')\in T_1 \\ (s',s'')\in \rho_\tau}} (\mathsf{None}(\{s, s''\}) \cap P_1) && \text{since } P_1 \subseteq \mathsf{None}(\{s'\}) \\
&= P_1 \cap \mathsf{None}(T_1 \circ \rho_\tau)
\end{aligned}
$$

The proof of $\mathsf{Some}(\{s_1, \ldots, s_n, s_{n+1}\}) = Q_1 \cup \mathsf{Some}(T_1 \circ \rho_\tau)$ is analogous. □

Proof Rule COMP-LIVENESS We show a proof rule COMP-LIVENESS for the verification of programs with compassion requirements wrt. general temporal properties given by Büchi automata on Figure 3.6. The proof rule is a modification of the proof rule COMP-TERM; we account for the temporal property by following Remark 3.1. A proof rule for the case when the property is given by a generalized Büchi automaton or a Streett automaton can by obtained from the rule COMP-LIVENESS in a straightforward way.

We look again at our examples.

CORR-ANY-DOWN We have computed an inductive labeled transition invariant that satisfies all premises of the proof rule COMP-TERM by applying our prototype implementation of the method that we will present in Section 3.6. The computed inductive labeled transition invariant is too large to be shown here. It contains refined versions of some (unrefined) labeled relations from the (non-inductive) labeled transition invariant for CORR-ANY-DOWN that we presented in Section 3.2. Furthermore, it contains additional labeled relations that are required to establish the inductiveness, *i.e.*, the premises P1 and P2 of the proof rule COMP-TERM.

MUX-BAKERY An inductive labeled transition invariant for the product program consists of the labeled relations below. We show only those labeled relations that are loops wrt. the location labels of the processes and the Büchi automaton. We omit the conjunct $\pi' = \pi$, which denotes loops wrt. location labels, in each assertion below.

Program P with:
set of states Σ,
set of compassion requirements $\mathcal{C}$,

Property Ψ,

Büchi automaton $\mathcal{A}_\Psi$ with:
set of states Q,
set of accepting states F,

Parallel composition of P and $\mathcal{A}_\Psi$ is program $P|||\mathcal{A}_\Psi$ with:
set of states $\Sigma_{|||} = \Sigma \times Q$,
set of transitions $\mathcal{T}$,
set of compassion requirements
$$\mathcal{C}_{|||} = \{\langle p \times Q, q \times Q\rangle \mid \langle p, q\rangle \in \mathcal{C}\} \cup \{\langle \Sigma \times Q, \Sigma \times F\rangle\},$$

Set of labeled relations $L = \{(T_1, P_1, Q_1), \ldots, (T_n, P_n, Q_n)\}$ with:
$T_i \subseteq \Sigma_{|||} \times \Sigma_{|||}$ and $P_i, Q_i \subseteq |\mathcal{C}_{|||}|$ for all $i \in \{1, \ldots, n\}$

P1: $\{(\rho_\tau, \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau)) \mid \tau \in \mathcal{T}\} \trianglelefteq L$
P2: $\{(T, P, Q) \boxdot \rho_\tau \mid (T, P, Q) \in L \text{ and } \tau \in \mathcal{T}\} \trianglelefteq L$
P3: T_i well-founded or $|\mathcal{C}_{|||}| \neq P_i \cup Q_i$ for each $i \in \{1, \ldots, n\}$

P satisfy Ψ under compassion requirements $\mathcal{C}$

Figure 3.6: Rule COMP-LIVENESS: temporal property under compassion requirements.

$$
\begin{array}{lll}
(at\_q_1, & \emptyset, & \{1,2,3,4\}) \\
(at\_\ell_3 \wedge at\_q_F \wedge x \leq y \wedge x' = x \wedge y' = y, & \emptyset, & \{2,3,4,5\}) \\
(at\_\ell_3 \wedge at\_m_1 \wedge at\_q_F \wedge y = 0 \wedge y < x \wedge x' = x \wedge y' = y, & \emptyset, & \{2,3,4,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge y < x \wedge x' = x \wedge y' = y, & \emptyset, & \{1,2,4,5\}) \\
(at\_\ell_3 \wedge at\_m_4 \wedge at\_q_F \wedge y < x \wedge x' = x \wedge y' = y, & \emptyset, & \{1,2,3,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge y < x \wedge x' = x \wedge y' > x \wedge y' \geq y, & \emptyset, & \{1,\ldots,5\}) \\
(at\_\ell_3 \wedge at\_m_4 \wedge at\_q_F \wedge y < x \wedge x' = x \wedge y' > x \wedge y' \geq y, & \emptyset, & \{1,\ldots,5\})
\end{array}
$$

Each relation that is labeled by the set $\{1, \ldots, 5\}$, which contains the indices of all compassion requirements, is well-founded. By the proof rule COMP-LIVENESS, the program MUX-BAKERY satisfies the non-starvation property.

MUX-TICKET Again, we show only the labeled relation of the inductive labeled transition invariant that are loops wrt. the location labels, and we omit the conjunct $\pi' = \pi$ in each assertion below.

$$
\begin{array}{lll}
(at\_q_1, & \emptyset, & \{1,2,3,4\}) \\
(at\_\ell_3 \wedge at\_m_1 \wedge at\_q_F \wedge x = s \wedge x' = x \wedge y' = y \wedge s' = s, & \emptyset, & \{2,3,4,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge x = s \wedge x' = x \wedge y' = y \wedge s' = s, & \emptyset, & \{2,3,4,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge y = s \wedge x' = x \wedge y' = y \wedge s' = s, & \emptyset, & \{1,2,4,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge y = s \wedge x' = x \wedge y' = y \wedge s' = s, & \emptyset, & \{1,2,4,5\}) \\
(at\_\ell_3 \wedge at\_m_4 \wedge at\_q_F \wedge x' = x \wedge y' = y \wedge s' = s, & \emptyset, & \{1,2,3,5\}) \\
(at\_\ell_3 \wedge at\_m_1 \wedge at\_q_F \wedge s < x \wedge x' = x \wedge s' > s, & \emptyset, & \{1,\ldots,5\}) \\
(at\_\ell_3 \wedge at\_m_3 \wedge at\_q_F \wedge s < x \wedge x' = x \wedge s' > s, & \emptyset, & \{1,\ldots,5\}) \\
(at\_\ell_3 \wedge at\_m_4 \wedge at\_q_F \wedge s < x \wedge x' = x \wedge s' > s, & \emptyset, & \{1,\ldots,5\})
\end{array}
$$

It is easy to see that every relation labeled by the set $\{1,\ldots,5\}$ containing the indices of all compassion requirements is well-founded. Hence, the non-starvation property is satisfied by the program MUX-TICKET.

3.6 Automated Synthesis

We apply the Galois connection approach for abstract interpretation [10] to propose a method for the automated synthesis of labeled transition invariants. We define operators on the domain of labeled relations whose least fixed points are labeled transition invariants. By applying the idea, proposed in Chapter 2, of abstracting binary relations over the program states we obtain an abstract interpretation based method for the automated synthesis of labeled transition invariants.

Fixed Point Operator For the given program P with the set of transitions $\mathcal{T}$ we define an operator $F_{\boxdot}$ on the domain of labeled relations as follows.

$$F_{\boxdot}(T,P,Q) \;=\; \{(T,P,Q) \boxdot \rho_\tau \mid \tau \in \mathcal{T}\}$$

Lemma 3.2 *The operator $F_{\boxdot}$ is monotonic wrt. the ordering $\trianglelefteq$ on labeled relations. Formally,*

$$(T_1,P_1,Q_1) \trianglelefteq (T_2,P_2,Q_2) \implies F_{\boxdot}(T_1,P_1,Q_1) \trianglelefteq F_{\boxdot}(T_2,P_2,Q_2).$$

Proof. Let (T_1,P_1,Q_1) and (T_2,P_2,Q_2) be a pair of labeled relations such that $(T_1,P_1,Q_1) \trianglelefteq (T_2,P_2,Q_2)$. Since $T_1 \subseteq T_2$, for each $\tau \in \mathcal{T}$ we have

$$\bigcup_{(s,s')\in T_1\circ\rho_\tau} \mathsf{None}(\{s,s'\}) \subseteq \bigcup_{(s,s')\in T_2\circ\rho_\tau} \mathsf{None}(\{s,s'\}),$$

i.e., we have $\mathsf{None}(T_1 \circ \rho_\tau) \subseteq \mathsf{None}(T_2 \circ \rho_\tau)$. Analogously, for each $\tau \in \mathcal{T}$ holds $\mathsf{Some}(T_1 \circ \rho_\tau) \subseteq \mathsf{Some}(T_2 \circ \rho_\tau)$. For each $\tau \in \mathcal{T}$ we conclude $(T_1,P_1,Q_1) \boxdot \rho_\tau \trianglelefteq (T_2,P_2,Q_2) \boxdot \rho_\tau$. □

Abstraction Given a concrete and an abstract domains $(D,\subseteq)$ resp. $(D^\#,\sqsubseteq)$ for binary relations over the program states, we define the concrete and abstract domains

$D_{\boxdot}$ resp. $D^{\#}_{\boxdot}$ for labeled relations (where $|\mathcal{C}|$ is the set of indices of the compassion requirements).

$$\begin{aligned} D_{\boxdot} &= D \times 2^{|\mathcal{C}|} \times 2^{|\mathcal{C}|} \\ D^{\#}_{\boxdot} &= D^{\#} \times 2^{|\mathcal{C}|} \times 2^{|\mathcal{C}|} \end{aligned}$$

The domains $D_{\boxdot}$ is ordered by the relation $\trianglelefteq$. We define the ordering $\trianglelefteq^{\#}$ on the abstract domain $D^{\#}_{\boxdot}$ as follows. We have

$$(T^{\#}_1, P_1, Q_1) \trianglelefteq^{\#} (T^{\#}_2, P_2, Q_2)$$

if the following three conditions hold.

$$T^{\#}_1 \sqsubseteq T^{\#}_2 \qquad\qquad P_1 \subseteq P_2 \qquad\qquad Q_1 \subseteq Q_2$$

Given an abstraction function α and the concretization function γ for binary relations over the program states that form a Galois connection, we define an abstraction function $\alpha_{\boxdot}$ for labeled relations.

$$\alpha_{\boxdot}(T, P, Q) \;=\; (\alpha(T), P, Q)$$

We only abstract the part of a labeled relation that ranges over the possibly infinite domain (of pairs of program states). The concretization function $\gamma_{\boxdot}$ is defined by

$$\gamma_{\boxdot}(T^{\#}, P, Q) \;=\; (\gamma(T^{\#}), P, Q).$$

Lemma 3.3 *The pair of functions ($\alpha_{\boxdot}$, $\gamma_{\boxdot}$) is a Galois connection.*

Proof. From the monotonicity of γ and α follows that $\alpha_{\boxdot}$ and $\gamma_{\boxdot}$ are monotonic. We carry out the following transformations.

$$\begin{aligned} \alpha_{\boxdot}(\gamma_{\boxdot}(T^{\#}, P, Q)) &= \alpha_{\boxdot}(\gamma(T^{\#}), P, Q) \\ &= (\alpha(\gamma(T^{\#})), P, Q) \end{aligned}$$

Since γ and α is a Galois connection, we have that $\alpha(\gamma(T^{\#})) \sqsubseteq T^{\#}$ and hence $\alpha_{\boxdot}(\gamma_{\boxdot}(T^{\#}, P, Q)) \trianglelefteq (T^{\#}, P, Q)$. Similarly, we obtain $(T, P, Q) \trianglelefteq \gamma_{\boxdot}(\alpha_{\boxdot}(T, P, Q))$. By Theorem 5.3.0.4 in [11], we conclude that $\alpha_{\boxdot}$ and $\gamma_{\boxdot}$ form a Galois connection. □

We canonically extend $\alpha_{\boxdot}$ to sets L of labeled relations. Formally,

$$\alpha_{\boxdot}(L) \;=\; \{\alpha_{\boxdot}(T, P, Q) \mid (T, P, Q) \in L\}.$$

The abstraction function $\alpha_{\boxdot}$ for extended command formulas defines the best abstraction of the operator $F_{\boxdot}$.

$$F^{\#}_{\boxdot}(T^{\#}, P, Q) \;=\; \alpha_{\boxdot}(F_{\boxdot}(\gamma_{\boxdot}(T^{\#}, P, Q)))$$

Abstract Fixed Points The monotonicity of the fixed point operator $F^{\#}_{\boxdot}$ is a direct consequence of Lemma 3.2 and the monotonicity of the abstraction/concretization functions. By Tarski's fixed point theorem, the least fixed point of $F^{\#}_{\boxdot}$ exists. We denote the least fixed point of $F^{\#}_{\boxdot}$ above $\{(\alpha(\rho_\tau), \mathsf{None}(\rho_\tau), \mathsf{Some}(\rho_\tau)) \mid \tau \in \mathcal{T}\}$ by $\mathsf{lfp}(F^{\#}_{\boxdot}, \mathcal{T})$. We compute $\mathsf{lfp}(F^{\#}_{\boxdot}, \mathcal{T})$ in the usual fashion. If the range of the abstraction function α does not allow infinite ascending chains then the fixed point computation always terminates after finitely many iterations.

input
program P, Büchi automaton $\mathcal{A}_\Psi$,
composition P and $\mathcal{A}_\Psi$ is $P|||\mathcal{A}_\Psi$ with:
set of transitions $\mathcal{T}$,
set of compassion requirements $\mathcal{C}_{|||} = \{\langle p_1, q_1\rangle, \ldots, \langle p_m, q_m\rangle\}$,
abstraction/concretization function α/γ
on/to binary relations over states of $P|||\mathcal{A}_\Psi$
begin
$F^{\#}_{\circledcirc} = \lambda(T^{\#}, P, Q).\ \{(\alpha(\rho_\tau \circ \gamma(T^{\#})),$
$P \cap \mathsf{None}(\rho_\tau \circ \gamma(T^{\#})),$
$Q \cup \mathsf{Some}(\rho_\tau \circ \gamma(T^{\#}))) \mid \tau \in \mathcal{T}\}$
$L^{\#} = \mathsf{lfp}(F^{\#}_{\circledcirc}, \mathcal{T})$
if foreach $(T^{\#}, P, Q)$ in $L^{\#}$ such that $\{1, \ldots, m\} = P \cup Q$
$\mathsf{well\text{-}founded}(\gamma(T^{\#}))$
then
return("Property Ψ holds under $\mathcal{C}$")
else
return("Don't know")
end.

Figure 3.7: Algorithm COMP-TRANS-PREDS: Verification of temporal property Ψ under compassion requirements $\mathcal{C}$ for the program P via abstract interpretation.

Algorithm The proof rule COMP-LIVENESS together with the above method for the synthesis of labeled transition invariants give rise to the algorithm for the verification of temporal properties under compassion requirements, shown in Figure 3.7. For each labeled relation $(T^{\#}, P, Q)$, the relation $\gamma(T^{\#})$ is represented by a 'simple' program that consists of a single while loop with only update statements in the loop body. There exist efficient well-foundedness tests for the class of simple while programs built using linear arithmetic expressions [37, 49].

We assumed that the property is given by the automaton $\mathcal{A}_\Psi$ equipped with the Büchi accepting condition. We obtain an algorithm for the case that $\mathcal{A}_\Psi$ is a generalized Büchi, or a Streett automaton in a straightforward way (see Section 3.4).

3.7 Related Work

This chapter continues the research on transition invariants started in Chapter 1, in which we account for the fairness requirements by applying the encoding into a Büchi automaton. The use of labeling allows us to account for the fairness requirements, both justice and compassion, directly, without resorting to automata.

There exists verification methods for the finite-state systems that account for the fairness requirements on the algorithmic level, *e.g.* [22, 30]. Experimental evaluations has confirmed the advantage of the direct treatment of fairness.

For dealing with infinite-state systems, there exists proof rules for the verification of termination [29] and general temporal properties [32] under justice and compassion requirements that account for the fairness requirements without applying the automata-

theoretic encoding. The proof rules rely on well-founded orderings, which must be supplied by the user. Justice requirements are handled directly by the proof rules; verification under compassion requirements is done by recursive application of the proof rule to a transformed program. Our proof rule treats justice and compassion in a uniform way.

The stack assertions based method of [24] for proving fair termination accounts for justice and compassion requirements directly. The method requires identification of tuples of well-founded mappings (stacks assertions), one element for each fairness requirement, which must by supplied by the user. The method keeps track on the satisfaction of the fairness through the tuple structure. In our proof rule, we use labeling for this purpose.

The automata-theoretic framework of [51] is the basis of our proof rule for the verification of general temporal properties. For infinite-state concurrent programs, the Büchi and the Streett acceptance conditions are translated to the Wolper (*i.e.* all states are accepting) acceptance condition. Thus, a proof of fair termination is reduced to a proof of termination of a program obtained from the original one by a transformation that encodes the fairness requirements into the state space. This approach is converse to ours.

3.8 Conclusion

We have presented a proof rule for the verification of temporal properties of concurrent programs under the fairness requirements of justice and compassion. We deal with the fairness requirements directly, *i.e.*, their encoding into automata is not needed. The direct accounting for the fairness requirements allows one to reduce the size of the specification automaton.

The proof rule relies on labeled transition invariants, which are finite sets of binary relation over program states extended with labels that keep track on the satisfaction of the fairness requirements. We treat temporal specifications given by an automaton with the Büchi, the generalized Büchi and the Streett acceptance condition in a uniform way. We have proposed a method for the automated construction of labeled transition invariants via abstract interpretation.

Chapter 4

Linear Ranking Functions

4.1 Introduction

In Chapters 1, 2, and 3 we observed that the components of (labeled) transition invariants, and abstract transitions can be represented by programs of a particular form. These programs, called *single while programs*, consist of a single while statement that only contains (possibly) non-deterministic update expressions. The verification via (labeled) transition invariants and abstract-transition programs requires termination checks for the corresponding single while programs. In this chapter, we describe an algorithm for proving termination of single while programs via linear ranking functions.

We propose the following method. Given a single while program for which we want to find a linear ranking function, we construct a corresponding system of linear inequalities over reals. This system encodes a test for the existence of linear ranking functions. A linear ranking function can be computed from a solution of the system. If the system is infeasible (has no solutions) then no linear ranking function exists. One can use the existing highly-optimized tools for linear programming to compute linear ranking functions efficiently.

4.2 Single While Programs

We formalize the notion of single while programs by a class of programs that are built using a single "while" statement and satisfy the following conditions:

- the loop condition is a conjunction of atomic propositions,
- the loop body may only contain update statements,
- all update statements are executed simultaneously.

We call this class *single while programs*. Pseudo-code notation for the programs of this class is given below.

while $(Cond_1$ **and** $\ldots$ **and** $Cond_m)$ **do**
    $Simultaneous\ Updates$
od

We are particularly interested in the subclass of single while programs built using linear arithmetic expressions over program variables.

Definition 4.1 (LASW Programs) *A linear arithmetic single while (LASW) program over the tuple of program variables $x = (x_1, \ldots, x_n)$ is a single while program such that:*

- *program variables have the domain of integers, rationals or reals,*
- *every atomic proposition in the loop condition is a linear inequality over (unprimed) program variables:*

$$c_1 x_1 + \cdots + c_n x_n \leq c_0,$$

- *every update statement is a linear inequality over unprimed and primed program variables*

$$a'_1 x'_1 + \cdots + a'_n x'_n \leq a_1 x_1 + \cdots + a_n x_n + a_0.$$

Note that we allow the left-hand side of an update statement to be a linear expression over program variables, and that an update can be nondeterministic, e.g., $x' + y' \leq x + 2y - 1$. This is a due to the fact that we use single while programs, and LASW programs in particular, to represent sub-relations of transition invariants (see Chapter 1) and abstract transitions (see Chapter 2).

We define a *program state* to be a valuation of program variables. The set of all program states is called the *program domain*. The *transition relation* denoted by the loop body of an LASW program is the set of all pairs of program states (s, s') such that the state s satisfies the loop condition, and (s, s') satisfies each update inequality. A *trace* is a sequence of states such that each pair of consecutive states belongs to the transition relation of the loop body.

We observe that the transition relation of a LASW program can be expressed by a system of linear inequalities over unprimed and primed program variables. The translation procedure is straightforward. For the rest of the chapter, we assume that an LASW program over the tuple of program variables $x = (x_1, \ldots, x_n)$ (treated as a column vector) can be represented by the system

$$(AA')\begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

of linear inequalities. We identify an LASW program with the corresponding system.

Next, we give an example of an LASW program.

Example 4.1 The following program loop with nondeterministic updates

while $(i - j \geq 1)$ **do**
$\quad (i, j) := (i - Nat, j + Pos)$
od

is represented by the following system of inequalities.

$$\begin{aligned} -i + j &\leq -1 \\ -i + i' &\leq 0 \\ j - j' &\leq -1 \end{aligned}$$

input
program $(AA')\binom{x}{x'} \leq b$
begin
if exists λ_1 **and** λ_2 **such that**
$\lambda_1, \lambda_2 \geq 0$
$\lambda_1 A' = 0$
$(\lambda_1 - \lambda_2)A = 0$
$\lambda_2(A + A') = 0$
$\lambda_2 b < 0$
then
return("Program Terminates")
else
return("Don't Know")
end.

Given λ_1 and λ_2, solutions of the system above, define $r = \lambda_2 A'$, $\delta_0 = -\lambda_1 b$, and $\delta = -\lambda_2 b$. A linear ranking function ρ is defined by

$$\rho(x) \quad = \quad \begin{cases} rx & \text{if exists } x' \text{ such that } (AA')\binom{x}{x'} \leq b, \\ \delta_0 - \delta & \text{otherwise.} \end{cases}$$

Figure 4.1: Termination Test and Synthesis of Linear Ranking Functions.

Note that the relations between program variables denoted the nondeterministic update statements $i := i - Nat$ and $j := j + Pos$, where Nat and Pos stand for any nonnegative and positive integer number respectively, can be expressed by the inequalities $i' \leq i$ and $j' \geq j + 1$.

4.3 Synthesis of Linear Ranking Functions

We say that a single while program is *terminating* if the program domain is well-founded by the transition relation of the loop body of the program, *i.e.*, if there is no infinite sequence $\{s_i\}_{i=1}^{\infty}$ of program states such that each pair (s_i, s_{i+1}), where $i \geq 1$, is an element of the transition relation.

The following theorem allows us to use linear programming over rationals (or reals) to test existence of a linear ranking function, and thus to test a sufficient condition for termination of LASW programs. The corresponding algorithm is shown on Figure 4.1.

Theorem 4.1 *A linear arithmetic single while program given by the system* $(AA')\binom{x}{x'} \leq b$ *is terminating if there exist two nonnegative vectors over rationals (or reals)* λ_1 *and* λ_2 *such that the following system is satisfiable.*

$$\lambda_1 A' = 0 \tag{4.1a}$$
$$(\lambda_1 - \lambda_2)A = 0 \tag{4.1b}$$
$$\lambda_2(A + A') = 0 \tag{4.1c}$$
$$\lambda_2 b < 0 \tag{4.1d}$$

Proof. Let the pair of nonnegative (row) vectors λ_1 and λ_2 be a solution of the system (4.1a)–(4.1d). For every x and x' such that $(AA')\binom{x}{x'} \leq b$, by assumption that $\lambda_1 \geq 0$, we have $\lambda_1(AA')\binom{x}{x'} \leq \lambda_1 b$. We carry out the following sequence of transformations.

$$\begin{aligned}
\lambda_1(Ax + A'x') &\leq \lambda_1 b \\
\lambda_1 Ax + \lambda_1 A'x' &\leq \lambda_1 b \\
\lambda_1 Ax &\leq \lambda_1 b && \text{by (4.1a)} \\
\lambda_2 Ax &\leq \lambda_1 b && \text{by (4.1b)} \\
-\lambda_2 A'x &\leq \lambda_1 b && \text{by (4.1c)}
\end{aligned}$$

From the assumption $\lambda_2 \geq 0$ follows $\lambda_2(AA')\binom{x}{x'} \leq \lambda_2 b$. Then, we continue with

$$\begin{aligned}
\lambda_2(Ax + A'x') &\leq \lambda_2 b \\
\lambda_2 Ax + \lambda_2 A'x' &\leq \lambda_2 b \\
-\lambda_2 A'x + \lambda_2 A'x' &\leq \lambda_2 b && \text{by (4.1c)}
\end{aligned}$$

We define $r = \lambda_2 A'$, $\delta_0 = -\lambda_1 b$, and $\delta = -\lambda_2 b$. Then, we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$ for all x and x' such that $(AA')\binom{x}{x'} \leq b$. Due to (4.1d) we have $\delta > 0$.

We define a function ρ as follows.

$$\rho(x) \;=\; \begin{cases} rx & \text{if exists } \; x' \; \text{ such that } \; (AA')\binom{x}{x'} \leq b, \\ \delta_0 - \delta & \text{otherwise.} \end{cases}$$

Any program trace induces a strictly descending sequence of values under ρ that is bounded from below, and the difference between two consecutive values is at least δ. Since no such infinite sequence exists, the program is terminating. □

The theorem above states a sufficient condition for termination. We observe that if the condition applies then a linear ranking function, *i.e.*, a linear arithmetic expression over program variables which maps program states into a well-founded domain, exists. The following theorem states that our termination test is complete for programs with linear ranking functions if the program variables range over rationals or reals.

Theorem 4.2 *If there exists a linear ranking function for the linear arithmetic single while program over rationals or reals with nonempty transition relation then the termination condition of Theorem 4.1 applies.*

Proof. Let the vector r together with the constants δ_0 and $\delta > 0$ define a linear ranking function. Then, for all pairs x and x' such that $(AA')\binom{x}{x'} \leq b$ we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$.

By the non-emptiness of the transition relation, the system $(AA')\binom{x}{x'} \leq b$ has at least one solution. Hence, we can apply the 'affine' form of Farkas' lemma (in [43]), from which follows that there exists δ_0' and δ' such that $\delta_0' \geq \delta_0$, $\delta' \geq \delta$, and each of the inequalities $-rx \leq -\delta_0'$ and $-rx + rx' \leq -\delta'$ is a nonnegative linear combination of the inequalities of the system $(AA')\binom{x}{x'} \leq b$. This means that there exist nonnegative real-valued vectors λ_1 and λ_2 such that

$$\begin{aligned}
\lambda_1(AA')\tbinom{x}{x'} &= -rx \\
\lambda_1 b &= -\delta_0'
\end{aligned}$$

and

$$\lambda_2(AA')\binom{x}{x'} = -rx + rx'$$
$$\lambda_2 b = -\delta'.$$

After multiplication and simplification we obtain

$$\lambda_1 A = -r \qquad \lambda_1 A' = 0$$
$$\lambda_2 A = -r \qquad \lambda_2 A' = r,$$

from which equations (4.1a)–(4.1c) follow directly. Since $\delta' \geq \delta > 0$, we have $\lambda_2 b < 0$, *i.e.*, the equation (4.1d) holds. □

The following corollary is an immediate consequence of Theorems 4.1 and 4.2.

Corollary 4.1 *Existence of linear ranking functions for linear arithmetic single while programs over rationals or reals with nonempty transition relation is decidable in polynomial time.*

Not every LASW program has a linear ranking function (see the following example).

Example 4.2 Consider the following LASW program over integers.

while $(x \geq 0)$ **do**
$x := -2x + 10$
od

The program is terminating, but it does not have a linear ranking function. For termination proof consider the following ranking function into the domain $\{0, \ldots, 3\}$ well-founded by the less-than relation $<$.

$$\rho(x) = \begin{cases} 1 & \text{if } x \in \{0,1,2\}, \\ 2 & \text{if } x \in \{4,5\}, \\ 3 & \text{if } x = 3, \\ 0 & \text{otherwise.} \end{cases}$$

It can be easily tested that the system (4.1a)–(4.1d) is not satisfiable for the LASW program

$$\begin{pmatrix} -1 & 0 \\ 2 & 1 \\ -2 & -1 \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix} \leq \begin{pmatrix} 0 \\ 10 \\ -10 \end{pmatrix}.$$

The following example illustrates an application of the algorithm based on Theorem 4.1.

Example 4.3 We prove termination of the LASW program from Example 4.1. The program translates to the system $(AA')\binom{x}{x'} \leq b$, where:

$$A = \begin{pmatrix} -1 & 1 \\ -1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad A' = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & -1 \end{pmatrix},$$
$$x = \begin{pmatrix} i \\ j \end{pmatrix}, \qquad b = \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}.$$

Let $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ and $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$. The system (4.1a)–(4.1d) is feasible, it has the following solutions:

$$\begin{aligned} \lambda'_2 = \lambda'_3 = \lambda''_1 = 0, \\ \lambda'_1 = \lambda''_2 = \lambda''_3, \\ \lambda'_1, \lambda''_2, \lambda''_3 > 0. \end{aligned}$$

Since the system is feasible the program is terminating. We construct a linear ranking function following the algorithm on Figure 4.1. We define $r = \lambda_2 A'$, $\delta_0 = -\lambda_1 b$, and $\delta = -\lambda_2 b$, and obtain $r = (\lambda'_1 \ \ -\lambda'_1)$, $\delta_0 = \delta = \lambda'_1$. Taking $\lambda'_1 = 1$ we obtain the following ranking function.

$$\rho(i,j) = \begin{cases} i - j & \text{if } \ i - j \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

4.4 Example: Singular Value Decomposition Program

We considered an algorithm for constructing the singular value decomposition (SVD) of a matrix. SVD is a set of techniques for dealing with sets of equations or matrices that are either singular or numerically very close to singular [40]. A matrix A is singular if it does not have a matrix inverse A^{-1} such that $AA^{-1} = I$, where I is the identity matrix.

Singular value decomposition of the matrix A whose number of rows m is greater or equal to its number of columns n is of the form

$$A = UWV^T,$$

where U is an $m \times n$ column-orthogonal matrix, W is an $n \times n$ diagonal matrix with positive or zero elements (called singular values), and the transpose matrix of an $n \times n$ orthogonal matrix V. Orthogonality of the matrices U and V means that their columns are orthogonal, *i.e.*,

$$U^T U = VV^T = I.$$

The SVD decomposition always exists, and is unique up to permutation of the columns of U, elements of W and columns of V, or taking linear combinations of any columns of U and V whose corresponding elements of W are exactly equal.

SVD can be used in numerically difficult cases for solving sets of equations, constructing an orthogonal basis of a vector space, or for matrix approximation [40].

We proved termination of a program implementing the SVD algorithm based on a routine described in [17]. The program was taken from [40]. It is written in C and contains 163 lines of code with 42 loops in the control-flow graph, nested up to 4 levels.

We used our transition invariant generator to compute a transition invariant for the SVD program. Proving the disjunctive well-foundedness of the computed transition invariant required testing termination of 219 LASW programs.

We applied our implementation of the algorithm on Figure 4.1, which was done in SICStus Prolog [26] using the built-in constraint solver for linear arithmetic [20]. Proving termination required 800 ms on a 2.6 GHz Xeon computer running Linux, which is in average 3.6 ms per each LASW program.

4.5 Related Work

A heuristic-based approach for discovery of ranking functions is described in [13]. It inspects the program source code for ranking function candidates. This method works for programs where the ranking function appears in the source code, which is often not the case.

In [8], an algorithm for generation of linear ranking functions for unnested program loops is proposed. It extracts a linear ranking function by manipulating polyhedral cones representing the transition relation of the loop and the loop invariant. The loop invariant is expected to be a system of linear inequalities produced by an invariant generator. The algorithm is not complete, since the loop invariant may not be linear. The algorithm uses the double description method to manipulate cones, which requires the worst-case exponential space to store cone representation.

The approach described in [9] is a generalization of the algorithm for unnested loops for programs with complex control structures. It uses the polyhedral cones method presented in [8] to detect linear ranking functions for strongly connected components in the control-flow graph of the program.

A decision procedure for the termination of single while programs with deterministic updates is proposed in [49]. The termination argument of the procedure relies on the eigenvalues of the update matrix. No ranking functions are constructed.

4.6 Conclusion

We presented an algorithm for generation of linear ranking functions for unnested program loops, which we are single while programs built using linear arithmetic expressions (LASW programs). Proving termination of such programs is required for verification of liveness properties of infinite-state systems via transition invariants [38], and abstract-transition programs [39].

Our method exploits the characteristic feature of LASW programs. They consist of a singe while loop without nested loops and branching statements within the loop body. Termination of an LASW program is implied by the feasibility of the system of linear inequalities derived from the program. The method is guaranteed to find a linear ranking function, and therefore to prove termination, if a linear ranking function exists.

The proposed algorithm can be efficiently implemented using a solver for linear programming over rationals. We used our prototypical implementation to prove termination of a singular value decomposition program, which required termination proofs for two hundred LASW programs.

Considering future work, we would like to find a characterization of a LASW programs which have linear ranking functions, *i.e.*, for which our algorithm decides termination. Another direction of work is to handle single while programs built using expressions other than linear arithmetic.

Chapter 5

Future Work

We have proposed the notion of transition invariant for the verification of liveness properties, and have shown a possible way of the automaton of transition invariant-based verification methods via abstract interpretation. Substantial work remains towards an automated tool for the verification of liveness properties of concurrent programs, based on transition invariants. We describe several directions for future work below.

Transition Abstraction Refinement We turn transition predicate abstraction into a full-fledged verification method by identifying a means for the automated abstraction refinement. This requires a notion of counterexample for liveness properties (of infinite-state systems). Its spuriousness must be effectively verifiable. Such a counterexample must also provide information that facilitates the discovery of new transition predicates. It is interesting to study the (relative) completeness of such a refinement procedure [2].

Transition Summaries Program blocks, *e.g.* loops or procedures, can be summarized by the corresponding transition invariants, thus generalizing the functional approach to program analysis of [44]. Such summaries are not inherently limited to the verification of safety properties, and can be refined on demand.

Parameterized Systems We may combine the counter abstraction technique, *e.g.* [14, 36], and the notion of abstract-transition programs to obtain abstractions of parameterized systems that preserve liveness properties, and, hence, do not require construction of additional fairness requirements for proving liveness.

Pointer Analysis Verification methods for programs with dynamically allocated memory ("program heap") must account for the temporal violations of heap invariants that occur during destructive updates. Such violations can be summarized by transition summaries, and safely ignored if the effect of the summary (re-)establishes the desired invariant. Such a technique can be useful in the context of shape analysis, see *e.g.* [42].

Program Analysis like "modifies x" We obtain an analysis that checks if a program variable x is not modified within a program block, *e.g.* [25], by proving that the relation $x' = x$ is a transition invariant for the block.

Chapter 6

Conclusion

We started by introducing the notion transition invariant. We identified the disjunctive well-foundedness as a property of relations that provides the characterization of the validity of liveness properties via transition invariants. The introduced inductiveness principle allows one to identify a given relation as a transition invariant. Consequently, we proposed a proof rule for the verification of liveness properties, based on (inductive) transition invariants. We claimed that our proof rule had a potential for automaton via abstract interpretation.

Next, we described a possible way to realize such a potential via transition predicate abstraction. Transition predicate abstraction and the corresponding notion of abstract-transition programs served as a basis for an automated method for proving termination under compassion requirements via abstract interpretation. This method accounts for fairness requirements imposed on program transitions in a direct way, which is generally considered desirable.

We introduced labeled transition invariants for the direct treatment of fairness requirements imposed on sets of program states, which is another common way to specify fairness. We attached sets of indices of fairness requirements to the components of transition invariants, thus accounting for fairness. We proposed a characterization of the validity of liveness properties via labeled transition invariants. The corresponding inductiveness principle together with the characterization of liveness resulted in a proof rule. We advised a method for the automation of the proof rule via abstract interpretation.

When dealing with concurrent systems with linear arithmetic expressions, the components of (labeled) transition invariants and abstract transition can be represented by single while programs. Their termination proofs are required by the proposed verification methods. We developed an algorithm for proving termination of single while programs via linear ranking functions.

We implemented the proposed methods in a prototype tool ARMC-Live. We applied ARMC-Live to synthesize the (labeled) transition invariants and abstract-transition programs that we presented for the example programs, and to perform the necessary well-foundedness checks. Thus, we obtain an experimental evidence for the claimed potential for automation of the proposed methods.

This dissertation demonstrates that transition invariants can provide a basis for the development of automated methods for the verification of liveness properties of concurrent programs. Thus, we hope that our work on transition invariants might lead to a series of activities for liveness, analogous to the activities leading to successful tools for safety.

Zusammenfassung

Programmverifikation stärkt unsere Überzeugung darin, dass ein Programm korrekt funktionieren wird. Manuelle Verifikation ist fehleranfällig und mühsam. Deren Automatisierung ist daher sehr erwünscht. Transitionsinvarianten (engl.: transition invariants) können eine neue Grundlage für die Entwicklung von automatischen Methoden zur Verifikation von nebenläufigen Programmen bereitstellen.

Die allgemeine Vorgehensweise zur Verifikation von temporalen Eigenschaften nebenläufiger Programme besteht darin, die Argumentation über die Programmberechnungen (Sequenzen von Programmzuständen) auf die Argumentation über Hilfsaussagen in Prädikatenlogik, wie z.B. Schleifeninvarianten und Rankingfunktionen, zu reduzieren. Solche Hilfsaussagen werden zuerst von dem Benutzer vorgeschlagen und danach durch ein automatisches Werkzeug überprüft. Die größte Herausforderung in der Automatisierung der Verifikationmethoden liegt in der automatischen Synthese dieser Hilfsaussagen.

Es gibt bereits erfolgreiche Werkzeuge, wie z.B. SLAM [1], ASTRÉE [3] und BLAST [19], zur automatischen Verifikation einer Teilklasse von temporalen Eigenschaften, die als Safety-Eigenschaften bezeichnet werden. Diese Eigenschaften setzen die Abwesenheit von Fehlern, wie Division durch Null, Überlauf und Zugriff auf einen Array außerhalb der Array-Grenzen, in allen Programmberechnungen voraus. Die genannten Werkzeuge können die dafür notwendigen Hilfsaussagen, die die Unerreichbarkeit der Fehlerzustände implizieren, automatisch synthetisieren. Somit verbleibt die automatische Synthese der Hilfsaussagen zur Verifikation von Liveness-Eigenschaften als die zentrale Herausforderung. Liveness-Eigenschaften verlangen, dass in jeder Berechnung bestimmte Programmzustände irgendwann auftreten. Die typischen Liveness-Eigenschaften sind Programmterminierung, d.h. das Auftreten von Zuständen, die keinen Nachfolger haben, und die garantierte Abarbeitung jeder gestellten Anfrage. Die Verifikation von Liveness-Eigenschaften erfordert die Synthese von Rankingfunktionen, die den Fortschritt in Richtung bestimmter Programmzustände nachweisen.

Die meisten Liveness-Eigenschaften nebenläufiger Programme gelten nur unter bestimmten Fairness-Anforderungen, wie z.B. die Anforderungen, dass jeder Prozess irgendwann ausgeführt wird oder ein Kommunikationskanal irgendwann erfolgreich eine Nachricht übermittelt. Fairness-Anforderungen werden in der Regel als Bedingungen an das Vorkommen von Programmübergängen oder -zuständen in Programmberechnungen spezifiziert. Es wird verlangt, dass z.B. in jeder unendlichen Berechnung jeder Programmübergang unendlich oft genommen wird oder dass keine Berechnung eine bestimmte Zustandsmenge nie verlässt. Das Einbeziehen von Fairness-Anforderungen erschwert die Verifikation, da das Auftreten von unterschiedlichen Mengen bestimmter Programmzustände berücksichtigt werden muss. Dies führt zu komplizierteren Ran-

kingfunktionen, die synthetisiert werden müssen.

Vor dieser Arbeit gab es keine Werkzeuge zur automatischen Verifikation von Liveness-Eigenschaften, die ähnlich zu denen sind, die wir zur Verifikation von Safety-Eigenschaften bereits besitzen. In dieser Dissertation schlagen wir Transitionsinvarianten vor, die einen neuen Typ von Hilfsaussagen zur Verifikation von Liveness-Eigenschaften darstellen. Transitionsinvarianten besitzen das Potential zur automatischen Synthese. Wir können die Techniken der abstrakten Interpretation zur automatischen Synthese von Transitionsinvarianten einsetzen. Diese Techniken haben bereits zum Erfolg der Werkzeuge zur Verifikation von Safety-Eigenschaften beigetragen. Wir beschreiben einen Weg zur Automatisierung der Synthese von Transitionsinvarianten, der die Verifikation von Liveness-Eigenschaften mit Hilfe der abstrakten Interpretation ermöglicht.

Diese Dissertation treibt den neusten Stand der Forschung voran, indem sie Transitionsinvarianten für die automatische Verifikation von Liveness-Eigenschaften vorschlägt. Wir fassen die Hauptbeiträge wie folgt zusammen.

Wir entwickeln eine neue Beweisregel für die Verifikation von Liveness-Eigenschaften, der Transitionsinvarianten zu Grunde liegen. Eine Transitionsinvariante ist eine Übermenge des transitiven Abschlusses der Übergangsrelation eines Programms. Eine Transitionsinvariante heißt disjunktiv wohl-fundiert, falls sie durch eine endliche Vereinigung von wohl-fundierten Relationen darstellbar ist. Wir charakterisieren die Gültigkeit einer Liveness-Eigenschaft durch die Existenz einer disjunktiv wohl-fundierten Transitionsinvariante. Wir führen ein Induktionsprinzip ein, das es uns erlaubt, eine gegebene Relation als eine Transitionsinvariante zu identifizieren. Die disjunktive Wohlfundiertheit und das Induktionsprinzip stellen die Basis unserer Beweisregel dar.

Wir beschreiben einen Weg, um diese Beweisregel zu automatisieren. Dafür führen wir zwei neuen Begriffe von ein: Transitionsprädikaten-Abstraktion (engl.: transition predicate abstraction) und abstraktes Transitionsprogramm (engl.: abstract-transition program). Wir benutzen diese Begriffe, um eine automatische Methode für den Beweis der Terminierung unter Fairness-Anforderungen zu entwickeln. Transition Predicates sind binäre Relationen über Programmzustände. Abstrakte Transitionsprogramme sind endliche gerichtete Graphen, deren Knoten durch Transitionsprädikate und deren Kanten durch Programmübergänge markiert sind. Wir geben einen Algorithmus zur automatischen Synthese eines abstrakten Transitionsprogramms für ein gegebenes Programm an. Wir argumentieren über die Terminierung anhand der Knotenmarkierung. Fairness-Anforderungen werden mit Hilfe der Kantenmarkierung berücksichtigt.

Um eine direkte Berücksichtigung der den Programmzuständen auferlegten Fairness-Anforderungen zu ermöglichen, führen wir markierte Transitionsinvarianten (engl.: labeled transition invariants) ein, die eine Erweiterung von Transitionsinvarianten darstellt. Die Mengen von Markierungen, die an die einzelnen Teilrelationen einer Transitionsinvariante angehängt werden, beinhalten die Indices der erfüllten Fairness-Anforderungen. Wir schwächen das Kriterium der disjunktiven Wohlfundiertheit ab, indem wir die Wohlfundiertheit nur für diejenigen Relationen einer endlichen Vereinigung voraussetzen, deren Mengen von Markierungen die Indices aller Fairness-Anforderungen enthalten. Wir entwickeln eine entsprechende Beweisregel und automatisieren diese mit Hilfe der abstrakten Interpretation.

Wir stellen Teilrelationen einer (markierten) Transitionsinvariante und abstrakte Transitionen, die bei der Verifikation von nebenläufigen, aus linearen arithmetischen Ausdrücken bestehenden Programmen entstehen, mit Hilfe von linearen ‘single while’ Programmen dar. Diese Programme bestehen aus einer While-Schleife, die nur

(möglicherweise nichtdeterministische) Update-Befehle enthält. Wir entwickeln einen Algorithmus zur Synthese linearer Rankingfunktionen für lineare ‘single while’ Programme und automatisieren somit die Wohlfundiertheitsbeweise, die bei der Anwendung der oben erwähnten Methoden auftreten.

Diese Dissertation demonstriert, dass Transitionsinvariante eine Basis für die Entwicklung von automatischen Methoden zur Verifikation von Liveness-Eigenschaften nebenläufiger Programmen bereitstellen können. Wir hoffen, dass unsere Arbeit an Transitionsinvarianten möglicherweise zu einer ähnlichen Reihe von Aktivitäten führen wird, die zur Entstehung erfolgreicher Safety-Werkzeuge beitragen.

Bibliography

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of PLDI'2001: Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[2] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. of TACAS'2002: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172. Springer, 2002.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of PLDI'2003: Programming Language Design and Implementation*, pages 196–207. ACM Press, June 7–14 2003.

[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. of PLDI'1993: Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.

[5] I. Browne, Z. Manna, and H. Sipma. Generalized verification diagrams. In *Proc. of FSTTCS'1995: Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer, 1995.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE'2003: Int. Conf. on Software Engineering*, pages 385–395, 2003.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'2000: Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[8] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *Proc. of TACAS'2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer, 2001.

[9] M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. of CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'1977: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL'1979: Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[12] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proc. of ICCL'1994: Int. Conf. on Computer Languages*, pages 95–112. IEEE, 1994.

[13] D. Dams, R. Gerth, and O. Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification (WAVe'00)*, pages 1–8, 2000.

[14] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[15] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[16] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. In Steffen and Levi [47], pages 223–238.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Univ Press, 3rd edition, 1996.

[18] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV'1997: Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[19] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL'2002: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[20] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[21] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation, a special issue on Compositionality*, 163, 2000.

[22] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. of ICALP'1998: Int. Colloq. on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 1–16. Springer, 1998.

[23] Y. Kesten, A. Pnueli, and M. Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences*, 62(4):668–690, 2001.

[24] N. Klarlund. Progress measures and stack assertions for fair termination. In *Proc. of PODC'1992: Principles of Distributed Computing*, pages 229–240. ACM Press, 1992.

[25] V. Kuncak and R. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[26] T. I. S. Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.

[27] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[28] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. of POPL'2001: Principles of Programming Languages*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

[29] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. of ICALP'1981: Int. Colloq. on Automata, Languages and Programming*, volume 115 of *LNCS*, pages 264–277. Springer, 1981.

[30] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of POPL'1985: Principles of Programming Languages*, pages 97–107. ACM Press, 1985.

[31] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, (3):243–263, 1974.

[32] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):91–130, 1991.

[33] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[34] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Progress. Draft, 1996.

[35] A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *Proc. of TACAS'2005: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 124–139. Springer, 2005.

[36] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Proc. of CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.

[37] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In Steffen and Levi [47], pages 239–251.

[38] A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. of LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.

[39] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Proc. of POPL'2005: Principles of Programming Languages*, pages 132–144. ACM Press, 2005.

[40] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, 1992.

[41] F. P. Ramsey. On a problem of formal logic. In *Proc. London Math. Soc.*, volume 30, pages 264–285, 1930.

[42] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[43] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.

[44] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[45] H. Sipma, T. Uribe, and Z. Manna. Deductive model checking. In *Proc. of CAV'1996: Computer Aided Verification*, volume 1102 of *LNCS*, pages 208–219. Springer, 1996.

[46] P. A. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1987.

[47] B. Steffen and G. Levi, editors. *Proc. of VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*. Springer, 2004.

[48] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 133–192. Elsevier and MIT Press, 1990.

[49] A. Tiwari. Termination of linear programs. In *Proc. of CAV'2004: Computer Aided Verification*, volume 3114 of *LNCS*, pages 70–82. Springer, 2004.

[50] T. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1999.

[51] M. Y. Vardi. Verification of concurrent programs — the automata-theoretic framework. *Annals of Pure and Applied Logic*, 51:79–98, 1991.

[52] M. Y. Vardi. Rank predicates vs. progress measures in concurrent-program verification. *Chicago Journal of Theoretical Computer Science*, 1996.

[53] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of POPL'2001: Principles of Programming Languages*, pages 27–40. ACM Press, 2001.

[54] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *Proc. of ESOP'2003: European Symp. on Programming*, volume 2618 of *LNCS*, pages 204–222. Springer, 2003.