Subsequence Invariants

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Klaus Dräger

Saarbrücken, 2010

Tag des Kolloquiums	23.08.2010
Dekan	Prof. Dr. Holger Hermanns

Prüfungsausschuss

Vorsitzender	Prof. Dr. Reinhard Wilhelm
Berichterstattende	Prof. Dr. Bernd Finkbeiner
	Prof. Dr. Ernst-Rüdiger Olderog
	Prof. Dr. Andreas Podelski
Akademischer Mitarbeiter	Dr. Thomas Hillenbrand

Abstract

In this thesis, we introduce *subsequence invariants*, a new class of invariants for the specification and verification of systems. Unlike state invariants, which refer to the state variables of the system, subsequence invariants characterize the behavior of a concurrent system in terms of the occurrences of sequences of synchronization events.

The first type of such invariants, *pure* subsequence invariants, are linear constraints over the possible numbers of such occurrences, where we allow every occurrence of a subsequence to be interleaved arbitrarily with other events. We then describe the more general class of *phased* subsequence invariants, in which additional restrictions can be placed on the events that may occur between those of a given sequence. In either case, subsequence invariants are preserved when a given process is composed with additional processes. subsequence invariants can therefore be computed individually for each process and then be used to reason about the full system.

We present an efficient algorithm for the computation of subsequence invariants of finite-state systems. Our construction can be applied incrementally to obtain a growing set of invariants given a growing set of event sequences. We then address the problem of proving subsequence invariants of infinite-state systems. For this we use an *abstraction refinement* procedure that uses small, incrementally transformed graph-based abstractions. In order to explain the techniques we use, we first introduce a simpler version of this method for state-based properties, and then show how to verify subsequence invariants.

Zusammenfassung

Inhalt dieser Arbeit sind *Subsequenzinvarianten*, eine neue Klasse von Invarianten für die Systemspezifikation und -verifikation. Im Gegensatz zu zustandsbasierten Invarianten, die über den Zustandsvariablen des Systems definiert sind, beschreiben Subsequenzinvarianten das gewünschte Systemverhalten anhand des Auftretens verschiedener Synchronisationsfolgen.

Wir beschreiben zunächst *reine* Subsequenzinvarianten, welche durch lineare Gleichungen auf den möglichen Häufigkeiten solcher Folgen von Events gegeben sind, zwischen denen jeweils beliebige andere Events autreten dürfen. Im Anschluss verallgemeinern wir diese zu Subsequenzinvarianten mit *Phasen*, in denen eine Synchronisationsfolge neben der eigentlichen Folge von Events auch durch Beschränkungen auf den dazwischen auftretenden Events gegeben sein kann. Beide Klassen von Invarianten bleiben gültig, wenn ein Prozess, für den sie gelten, mit beliebigen anderen Prozessen kombiniert wird. Sie können daher für jeden einzelnen Prozess berechnet und dann zur Verifikation des gesamten Systems verwendet werden.

Wir präsentieren einen effizienten Algorithmus für die Berechnung von Subsequenzinvarianten auf Systemen mit endlichen Zustandsräumen. Diese Konstruktion kann auch inkrementell angewandt werden, wenn die Menge der betrachteten Subsequenzen allmählich wächst. Für die Berechnung von Subsequenzinvarianten für Systeme mit unendlichem Zustandsraum führen wir eine Methode ein, die auf dem Prinzip der Abstraktionsverfeinerung basiert. Unsere Version dieses Ansatzes zeichnet sich durch die Verwendung sehr kleiner, graphenbasierter Abstraktionen aus. Wir präsentieren zunächst eine einfachere Variante des Verfahrens für zustandsbasierte Fehlerbedingungen, an der sich die verwendeten Operationen leichter demonstrieren lassen, und beschreiben dann die Anpassungen für die Verifikation von Subsequenzinvarianten.

Acknowledgments

I would like to thank my advisor, Bernd Finkbeiner, for his tireless support and many inspiring discussions. This thesis would not have been possible without his insightful advice.

I also thank Ernst-Rüdiger Olderog and Andreas Podelski for agreeing to review this thesis. The conversations I have had with both over the years have been very enlightening.

The members of the Reactive Systems Group at Saarland University were also very helpful in all kind of ways. Rayna Dimitrova, Rüdiger Ehlers, Michael Gerke, Lars Kuhtz, Andrey Kupriyanov, Hans-Jörg Peter, Markus Rabe, Christa Schäfer, and Sven Schewe: Thanks for too many things to list.

I am grateful to the German Research Foundation (DFG) for supporting this work as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

Contents

Chapter 1

Introduction

The importance of formal verification of reactive systems, i.e. systems which are engaged in an ongoing interaction with an environment, has become more and more obvious as the impact of such systems on our daily lives has become ubiquitous. Such spectacular cases as the Ariane 5 disaster already made the case for verification of mission-critical software, but nowadays, systems whose errors can cause significant damage are everywhere, from systems governing financial transactions to automotive electronics.

The properties we will be treating are *safety* properties, which express that some "bad" behavior can never occur in a given system. The usual approach is to describe the desired system behavior in terms of its states, using a specification language such as temporal logic. The inherent problem of this idea is that many critical systems consist of a large number of components, and the state of such a system is then given by the combination of the component states. The number of such states therefore grows *exponentially* with the size of the system.

This inherent *state explosion* problem has inspired many clever methods which can mitigate its effect, such as modular verification [41], symmetry and partial order reduction [35], and directed model checking [76]. Our approach, on the other hand, introduces an alternative to the state-based approach itself. We present *subsequence invariants*, a language for specifying systems in terms of component *interaction.*

These subsequence invariants are *compositional.* This means that an invariant which holds for some component of the system, automatically also holds for the system as a whole. We can therefore generate the invariants for each process individually and simply combine to obtain a set of system invariants.

We present invariant generation algorithms for finite-state processes, which generate all subsequence invariants involving a given set of subsequences in time linear in the size of the process and cubic in the number of subsequences. This low time-complexity is another major advantage of our approach.

Many important system classes, such as real-time systems, involve data from infinite domains. The most successful way of verifying such systems is *abstraction refinement*, which is based on a sequence of finite-state abstractions of the system. We present a version of the abstraction refinement approach which, using local refinements of a graph-based abstraction, allows the verification of subsequence invariants of infinite-state systems.

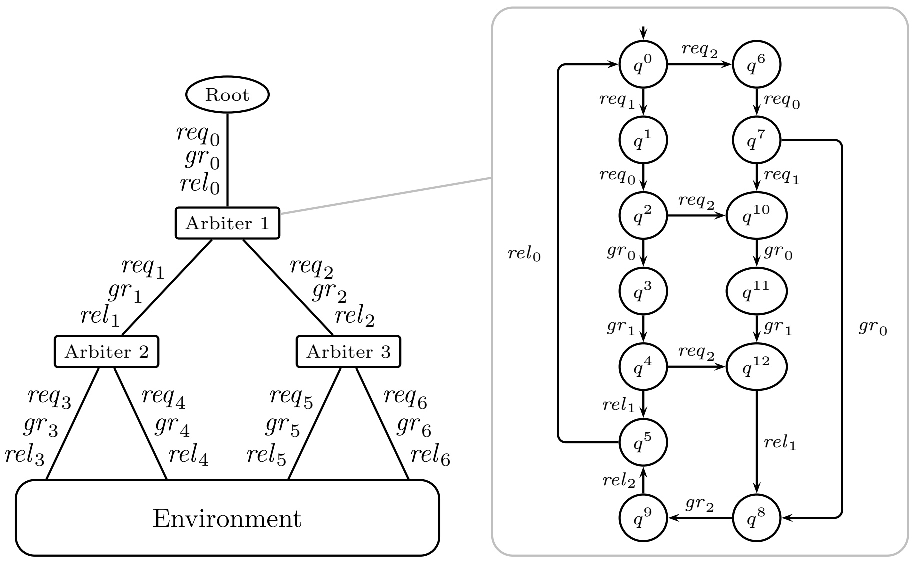

Root
req_0 gr_0 rel_0
Arbiter 1
req_1 gr_1 rel_1
req_2 gr_2 rel_2
Arbiter 2
Arbiter 3
req_3 gr_3 rel_3
req_4 gr_4 rel_4
req_5 gr_5 rel_5
req_6 gr_6 rel_6
Environment
q^0 q^1 q^2 q^3 q^4 q^5 q^6 q^7 q^8 q^9 q^{10} q^{11} q^{12}
req_2 req_1 req_0 req_0 req_2 req_1 gr_0 gr_0 gr_1 gr_1 req_2 rel_1 rel_1 rel_2 gr_2 rel_0 gr_0

Figure 1.1: Arbiter tree: Access to a shared resource is controlled by binary arbiters arranged in a tree, with a central root process.

1.1 Subsequence Invariants

A subsequence invariant is a linear constraint over the possible numbers of occurrences of a finite set of *subsequences* of synchronization events. Each occurrence of a subsequence may be *scattered* over a sequence of synchronization events: for example, the sequence *acacb* contains two occurrences ($\underline{a}cac\underline{b}$ and $ac\underline{a}c\underline{b}$) of the subsequence *ab*. This robustness with respect to arbitrary interleavings with other events ensures that subsequence invariants are preserved when a given process is composed with additional processes. Subsequence invariants can therefore be computed individually for each process and then be used to reason about the full system.

As an example, consider the arbiter tree shown in Figure 1.1. The environment represents the clients of the system, which may request access to a shared resource from one of the leaf nodes of the arbiter tree. The arbiter node then sends a request to its parent in the tree. This request is forwarded up to the central root process, which generates a grant as soon as the resource is available. The grant is propagated down to a requesting client, which then accesses the resource and eventually sends a release signal when it is done. Each arbiter node should satisfy the following subsequence invariants:

(1) Whenever a grant is given to a child, the number of grants given to the other child so far equals the number of releases received from it. For example, for Arbiter 1, each occurrence of gr_2 in an event sequence w is preceded by an equal number of occurrences of gr_1 and rel_1. As we will see in the following, this can be expressed in terms of subsequences as:

$$|w|_{gr_1 gr_2} = |w|_{rel_1 gr_2} \text{ and, symmetrically, } |w|_{gr_2 gr_1} = |w|_{rel_2 gr_1}.$$

(2) Whenever a grant is given to a child, the number of grants received from the parent exceeds the number of releases sent to it by exactly 1. For example, for Arbiter 1, each occurrence of gr_1 or gr_2 is preceded by one more occurrence of gr_0 than of rel_0. This corresponds to the subsequence invariant

$$|w|_{gr_0 gr_i} = |w|_{rel_0 gr_i} + |w|_{gr_i}, \text{ for } i = 1, 2.$$

(3) Whenever a release is sent to, or a grant received from, the parent, the number of releases received from each child equals the number of grants given to that child. For Arbiter 1:

$$|w|_{gr_i gr_0} = |w|_{rel_i gr_0} \text{ and } |w|_{gr_i rel_0} = |w|_{rel_i rel_0}, \text{ for } i = 1, 2.$$

(4) The differences between the corresponding numbers of grants and releases only take values in $\{0, 1\}$. For Arbiter 1, this can be expressed by

$$|w|_{gr_i rel_i} + |w|_{rel_i gr_i} = |w|_{gr_i gr_i} + |w|_{rel_i rel_i} + |w|_{rel_i}, \text{ for } i = 0, 1, 2.$$

Combined, the subsequence invariants (1) - (4) of all arbiter nodes imply that the arbiter tree guarantees mutual exclusion among its clients.

Phased subsequence invariants. The subsequence invariants we have seen so far have been *pure* subsequence invariants, by which we mean that they are defined in terms of sequences of events without any restrictions on what may occur in between. Introducing such restrictions increases the expressive power. In particular, we will present an extension, *phased* subsequence invariants, which takes into account possible sets of events which must not occur in between two successive elements of the sequence. These invariants are thus linear constraints on expressions such as $|w|_{a\{b,c\}d}$: the number of occurrences of a, followed by an occurrence of d, with no b or c in between.

For example, consider again the arbiter tree. Another property that one usually wants to require is the following kind of *liveness* property: After receiving the grant, an arbiter can only give finitely many grants to its children before returning the grant to its parent. One way of guarantee this is to impose a stronger (safety) requirement: Any grant the arbiter receives must be returned after giving at most one grant to each of its children. This can be expressed (for Arbiter 1) using a phased sequence invariant which forbids two occurrences of the event gr_i $(i = 1, 2)$ with no occurrence of rel_0 in between, i.e. that the sequence $gr_i\{rel_0\}gr_i$ never occurs:

$$|w|_{gr_i\{rel_0\}gr_i} = 0.$$

In order to additionally specify that, upon receiving a grant, an outstanding request must be satisfied before the grant is returned, one can use the invariant

$$|w|_{req_i\{gr_i\}gr_0\{gr_i\}rel_0} = 0.$$

Note that specifying this property using pure subsequence invariants is not possible: The closest one can get is something like

$$|w|_{req_i rel_0} = |w|_{gr_i rel_0},$$

requiring that whenever the arbiter returns its grant, the child has been given one grant per request. Unless the clients are restricted from making multiple requests, this does not capture the intended behavior.

1.2 Invariant Generation

As we will see in Chapter 2, the behavior of the numbers of subsequence occurrences follows some simple linear recurrence relations. In particular, for any event a and subsequences $U = \{u^1, \dots, u^n\}$, there is a matrix F_a such that, for any $w \in \Sigma^*$,

$$\begin{pmatrix} |w.a|_{u^1} \\ \vdots \\ |w.a|_{u^n} \end{pmatrix} = F_a \begin{pmatrix} |w|_{u^1} \\ \vdots \\ |w|_{u^n} \end{pmatrix}.$$

For example, if $\Sigma = \{a, b\}$ and $U = \{\epsilon, a, b, ab, ba\}$, then the matrices are

$$F_a = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}, \qquad F_b = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

so that, starting from the values $|\epsilon|_\epsilon = 1$ and $|\epsilon|_u = 0$ for all other u, we can obtain the values $|abaa|_u$:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \overset{F_a}{\to} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \overset{F_b}{\to} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \overset{F_a}{\to} \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 1 \end{pmatrix} \overset{F_a}{\to} \begin{pmatrix} 1 \\ 3 \\ 1 \\ 1 \\ 2 \end{pmatrix}.$$

Since subsequence invariants are linear equalities, they can be computed for a finite-state system by determining, for each state q, the vector space spanned by the vectors associated to those words for which there is a run to q. This can be done in time linear in the size of the process and cubic in the number of subsequences, as we will show in Chapter 3.

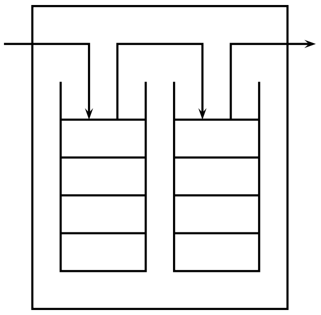
Due to the compositionality of subsequence invariants, any invariants obtained for the processes of a system are automatically also system invariants. There are, however, ways to obtain additional invariants, which we also treat in Chapter 3. In particular, we will make use of the algebraic and combinatorial structure of the sets of subsequence occurrences.

We will also show how the set of subsequence invariants of a process can be extended to accommodate a larger set of subsequences. This allows an *incremental* invariant generation procedure, which starts with a small core of subsequences and then gets gradually refined.

1.3 Infinite-State Systems

Many important systems that one wants to verify involve data from infinite domains, such as the real numbers or a priori unbounded data structures. We will represent such systems as transition systems consisting of

- a finite set of system variables,
- an initial condition, describing the possible starting states,



init	$i = 0, u = 1, v = 1, x = 1, y = 1$
rda	$i < MAX, x = 1, y = 1, u := u + v, i := i + 1$
rdb	$i < MAX, x = 1, y = 1, v := u + v, i := i + 1$
sha	$i > 0, u > v, u := u - v, x := x + y, i := i - 1$
shb	$i > 0, u < v, v := v - u, y := x + y, i := i - 1$
wra	$i = 0, u = 1, v = 1, x > y, x := x - y$
wrb	$i = 0, u = 1, v = 1, x < y, y := y - x$
mty	$i = 0, u = 1, v = 1, x = 1, y = 1$
inv	$\lvert w\rvert_{rda.rda.mty} - \lvert w\rvert_{wra.wra.mty} = 0,$ $\lvert w\rvert_{rda.rdb.mty} - \lvert w\rvert_{wra.wrb.mty} = 0,$ $\lvert w\rvert_{rdb.rda.mty} - \lvert w\rvert_{wrb.wra.mty} = 0,$ $\lvert w\rvert_{rdb.rdb.mty} - \lvert w\rvert_{wrb.wrb.mty} = 0$

Figure 1.2: Message buffer example. The table shows the initial condition *init*, the transition relations, and the subsequence invariants.

- a finite set of labeled transitions, representing the possible state transformations, and
- a set of invariants to be satisfied.

For example, consider the system given in Figure 1.2. It represents a simple message buffer using a two stacks, each encoded into a pair of integer variables. After a number of messages (not exceeding a predefined bound MAX) has been read, the buffer moves the contents of the first stack to the second stack, reversing the order of the messages, and then flushing the second stack to the output, reversed for the second time. The buffer then is ready to accept messages again. The invariants specify that, whenever the buffer is empty, the output stream contains equally many occurrences of aa, ab, ba, and bb as the input stream.

In order to check subsequence invariants on infinite-state systems, we combine the fixed-point iteration introduced for the finite-state case with a special version of the *abstraction refinement* approach [36, 21, 5, 18, 25, 39, 56], which we describe in Chapters 4 and 5. Chapter 4 will concentrate on the refinement procedure as such, which is also suitable for state-based invariants, in order to introduce the underlying concepts. Chapter 5 will then describe the handling of subsequences, in particular the integration of the invariant generation and abstraction refinement loops.

1.4 Related work

Property specification. Formal specifications of system requirements are an essential prerequisite for any verification effort. For reactive systems, the allowed system behavior is usually given in terms of state assertions, using temporal logics such as LTL [62], CTL [17], and PSL [33], or automata-based models such as statecharts [38]. While such specifications are expressive, their verification is inherently state-based and therefore plagued by the state explosion problem.

In order to avoid this problem, it is desirable to not have to explicitly deal with the system state at all. This can be achieved by specifying the desired system behavior in terms of the sequences of interactions between processes.

The treatment of the states and transitions of a process is then only necessary in order to determine the properties satisfied by the interface of the process.

This kind of specification was pioneered by the *stream*- and *trace*-based approaches of [48, 78, 60, 10]. These approaches make use of operators on sequences such as concatenation and projection. A set of sequences can then be specified by logical formulas involving predicates such as the prefix order, and linear expressions over lengths of sequences. In particular, constraints such as (4) in the arbiter example, or more generally linear inequalities on event counters, are expressible using trace logic formulas. Unlike this subclass of trace properties, Subsequence Invariants can only specify a finite range of values for a linear expression, but can involve arbitrary sequences of events. Trace-based specification logics are very general and expressive, for example the trace logic of [78, 60] can express any recursively enumerable set of traces (Theorem 4.2.5 in [60]).

By contrast, subsequence invariants are decidable. They condense whole sets of sequences to a set of linear expressions, thus allowing for fast computation of an abstraction of the set of interaction sequences in which a process can engage. Since the precision of this abstraction can be adapted by expanding the set of subsequences, and the computation can be done incrementally, this provides a set of efficient algorithmic tools for interaction-based system analysis.

Invariants. There is a significant body of work on the generation of invariants over program variables, ranging from heuristics (cf. [34]), to methods based on abstract interpretation (cf. [23, 8, 7, 70]) and constraint solving (cf. [20]). The key difference to our approach is that, while these approaches aim at finding a concise characterization of a complex state space, we aim at finding a concise representation of a complex process interaction. T-invariants, which relate the number of firings of different transitions in a Petri net, have a similar motivation (cf. [59]), but are not applied in the compositional manner of subsequence invariants.

Subsequences. Subsequence occurrences have, to the best of our knowledge, not been used in verification before. However, there has been substantial interest in subsequences in the context of formal languages, in particular in connection with Parikh matrices and their generalizations; see, for example, [65, 53, 66, 24], as well as Parikh's original paper [61], introducing Parikh images.

Subsequences are also used in machine learning, in the context of kernel-based methods for text classification [50]; here the focus is on their use as characteristic values of given pieces of text, not on the characterization of languages or systems by constraints on their possible values.

Weighted automata. The invariant generation algorithms we investigate can be formalized in the setting of weighted automata [30]. The general idea is to associate with each event a a linear transformation over a semiring (in our case, the field of real numbers). One can then characterize sequences of events by the vector that is obtained by applying the corresponding sequence of transformations to some initial vector. The relation to subsequences is given by the fact that the subsequence counters $|w|_u$ evolve according to linear transformations over the real numbers, see Section 2.2.

This very general formalism has received a lot of attention for applications such as natural language processing [58], image processing [47] or enumerative combinatorics [64], and also for the formal specification of systems [30, 22, 54]. In particular, weights over suitable semirings can be used to model probabilistic aspects or costs of actions. Unlike our approach, the focus in the latter works is on the verification of properties of systems which are weighted automata, rather than the use of weighted automata for the specification of properties. Closest to our ideas is the recent work of Baier [2] on recognizability of ω-languages by probabilistic automata.

The general setting of weighted automata is an interesting field for future extensions of our approach. The generality comes at a price, however: Even usually simple concepts such as linear independence become nontrivial in general semirings [1], and even over relatively tame semirings like the integers, properties of weighted automata can quickly become undecidable [37].

Abstraction. There is a rich literature on *predicate abstraction* and the *abstraction refinement loop* [36, 21, 5, 18, 25]. Similar to our approach, the model checker BLAST [40] also uses *Craig interpolation* [55] for predicate discovery. Other verification tools, such as the C-code verifier MAGIC [15], propagate predicates from branch and loop conditions using the weakest precondition operator. Standard approaches to reduce the size of the abstraction are to detect redundant predicates [19] and to approximate the abstraction, for example by *Cartesian approximation* [4, 3, 51].

The key difference between our approach and standard predicate abstraction is that we use new predicates only *locally* in order to split individual nodes, while predicate abstraction interprets every predicate *globally* in every abstract state. Our approach can be seen as a generalization of *lazy abstraction* [39, 56], which incrementally refines the abstraction with new predicates as the control flow graph is searched in a forward manner to find an error path. New predicates in lazy abstraction only affect the subgraph reachable from the current node. Lazy abstraction thus exploits locality in branches of the control flow graph while our approach exploits locality in individual nodes of the abstraction.

Our abstraction process is similar to *deductive model checking* [67], which also refines an explicit abstraction by splitting individual nodes. While we only handle safety conditions, deductive model checking provides rules for full linear-time temporal logic. However, deductive model checking is only partly automated, relying on the user to select the nodes and predicates for splitting.

Slicing and state space reduction. Program slicing, introduced by Weiser [75], is a static analysis technique widely used in debugging, program comprehension, testing, and maintenance. Essentially, slicing extracts the parts of a program which might affect some given slicing criterion (e.g., a variable at some control point). Slicing has become one of the standard reduction techniques in finite-state model checking (for instance in SAL [71], Bandera [31], SPIN [57], and IF [9]). As a preprocessing step, slicing is both cheap and effective (see [31] for an experimental evaluation). More recently, slicing has been used in automated abstraction refinement to simplify abstract error paths (thus analyzing individual paths, not the full abstraction). *Path slicing* [46] removes irrelevant parts of the abstract error path before the path is passed to the theorem prover to verify if the path can be concretized.

Usually, the slice is determined by a dependency analysis on the control flow graph of the program. A more refined technique, taking additional information about the property under interest into account, is *conditioned slicing* [14]. Here, an assumption about the initial (forward conditioning) or final states (backward conditioning) is added in the form of a predicate, and slicing then only keeps the statements which can be executed from an initial state or which lead to a final state satisfying the predicate.

Closest to our work is the backward conditioning approach of [32] (used for program comprehension, not verification). Backward conditioning proceeds by a symbolic execution of the program and the use of a theorem prover to prune the execution paths which do not lead into a desired final state. The analysis is however always carried out on the concrete program, not its abstraction, and the technique will – due to its objective of program comprehension – preserve *all* paths to the given final states. A use of conditioned slicing in verification can be found in [72], where the condition is extracted from a temporal logic formula of the form $G(p \to q)$. A conditioning method operating on an abstraction of the program is presented in [45]. On this abstraction it can be determined under which conditions one statement might affect another (while for verification we need to find out whether some condition might hold at all).

Target enlargement [6, 26, 77] is often used in model checking to help detect error paths. Typically, this is done using a size-bounded backward exploration starting from the error states; the resulting underapproximation of the backwards-reachable states replaces the original error condition in the subsequent forward exploration. The rule presented in Section 4.5.4 represents a single (forward or backward) exploration step, so that the enlargement is incremental and can be interleaved with the refinement and other slicing operations. *Partial order reduction* [44, 35] is a standard method to accelerate the exploration of the state space during model checking, applied in tools like SPIN [57] and Java Pathfinder [73]. To the best of our knowledge, the application to an abstraction during the refinement loop as described in Section 4.5.5 is new.

1.5 Contributions

This thesis introduces subsequence invariants, a specification language for concurrent systems based on their interaction instead of the system state. These invariants are expressive, yet efficiently computable and *compositional*, i.e. the system satisfies all subsequence invariants satisfied by any of its processes.

We show how to compute the set of subsequence invariants over a given set of subsequences for a finite-state process in time *linear* in the size of the process. This algorithm can be further optimized for pure subsequence invariants of strongly connected processes, and can be extended to an *incremental* procedure, capable of dealing with a growing set of subsequences.

We further show how to compute additional system invariants beyond the ones inherited directly from the processes. In particular, we prove the existence of classes of *tautological* subsequence invariants, based on algebraic and combinatorial properties of subsequence occurrences.

We introduce a verification procedure for subsequence invariants of infinite-state systems, which integrates the invariant generation algorithm for the finite-state case with *abstraction refinement*. For this, we use a special refinement loop which uses strictly local refinement steps and slicing operations on the abstraction in order to both keep the abstraction small and allow the fixpoint iteration for the subsequence invariants to proceed incrementally, interleaved with the refinement loop.

1.6 Organization of the thesis

- In the next chapter, we introduce the class of subsequence invariants. We present the necessary preliminaries from automata theory and linear algebra, and present a number of useful properties, such as compositionality and closure under disjunction.
- We then show how to compute these invariants for a finite-state machine, and discuss ways of obtaining additional invariants. In particular, we introduce a number of families of *universally valid* linear constraint satisfied by subsequence counters, which can be used to strengthen the system invariants obtained from processes, and consider the case of an incrementally *growing* set of subsequences.
- Next, we introduce the *Slicing Abstractions* approach to abstraction refinement. In Chapter 4, we first introduce the simpler version of the procedure for state-based invariants, describing the refinement and slicing operations involved in the gradual transformation of an abstraction.
- Finally, in Chapter 5, we present the Slicing Abstractions approach for Subsequence Invariants of infinite-state systems. This procedure integrates the refinement and slicing operations introduced in the previous chapter with the invariant generation approach for Subsequence Invariants.

1.7 Publications and Collaborations

The contents of Chapters 2 to 4 is partially based on the following publications:

[11] I. Brückner, K. Dräger, B. Finkbeiner, H. Wehrheim:
Slicing Abstractions.
Fundamentals of Software Engineering (FSEN), 2007.

[12] I. Brückner, K. Dräger, B. Finkbeiner, H. Wehrheim:
Slicing Abstractions.
Extended version of [11], Fundamenta Informaticae (FI), 2008.

[27] K. Dräger, B. Finkbeiner:
Subsequence Invariants.
International Conference on Concurrency Theory (CONCUR), 2008.

[28] K. Dräger, B. Finkbeiner :
Subsequence Invariants.
Extended version of [27], AVACS Technical Report 42, 2008.

[29] K. Dräger, A. Kupriyanov, B. Finkbeiner, H. Wehrheim:
SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems.
Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2010.

The second version of SLAB has been implemented in collaboration with Andrey Kupriyanov.

Chapter 2

Subsequence Invariants

In this chapter, we introduce the class of subsequence invariants and examine their properties. We start with an overview of the underlying concepts from linear algebra and formal language theory.

2.1 Preliminaries

2.1.1 Linear algebra

For a given finite set U, the *real vector space* $\mathbb{R}^U$ *generated by* U consists of all tuples $\phi = (\phi_u)_{u \in U}$ of real numbers indexed by the elements of U.

For a given set of vectors $\phi^1, \ldots, \phi^k \in \mathbb{R}^U$, the subspace *spanned by* $\phi^1, \ldots, \phi^k$ consists of all linear combinations of the ϕ^i:

$$\mathit{span}(\phi^1, \ldots, \phi^k) = \{\lambda_1 \phi^1 + \ldots \lambda_k \phi^k \mid \lambda_i \in \mathbb{R}\}.$$

Note that we use superscripts for indices in a tuple of vectors or words, and subscripts for indices within a vector or word. We hope this increases the clarity of our notation.

A set $B = \{\phi^1, \ldots, \phi^k\} \subseteq \mathbb{R}^U$ is *linearly independent* if the linear combination $\lambda_1 \phi^1 + \cdots + \lambda_k \phi^k$ is 0 only for $\lambda_1 = \cdots = \lambda_k = 0$. A *basis* for a subspace $H \subseteq \mathbb{R}^U$ is a linearly independent set B with $H = \mathit{span}(B)$.

We assume that the set U is equipped with a total ordering $<$, i.e., $U = \{u^1, \ldots, u^m\}$ with $u^1 < \cdots < u^m$. We write vectors as tuples $\phi = (\phi_{u^1}, \ldots, \phi_{u^m})^T$ according to this order. The *pivot element* $\mathit{pivot}(\phi)$ of a vector $\phi \neq \mathbf{0}$ is the $<$-least element u such that ϕ_u is nonzero.

A set B of nonzero vectors is in *reduced echelon form* if $\phi_{\mathit{pivot}(\psi)} = 0$ for all $\phi, \psi \in B$ with $\phi \neq \psi$. Such a B is obviously linearly independent, since for any linear combination $\psi = \lambda_1 \phi^1 + \cdots + \lambda_k \phi^k$ with $\phi^i \in B$, the coefficient $\psi_{\mathit{pivot}(\phi^i)}$ is just $\lambda_i \cdot \phi^i_{\mathit{pivot}(\phi^i)}$, which is 0 only if $\lambda_i = 0$.

Given a sequence $\phi^1, \phi^2, \ldots$ of vectors, we can incrementally compute bases of $\mathit{span}(\phi^1, \ldots, \phi^m)$ in reduced echelon form using Gauß-Jordan elimination as follows: Let a basis for $\mathit{span}(\phi^1, \ldots \phi^m)$ be $\{\psi^1, \ldots, \psi^k\}$, and let ϕ^{m+1} be the next vector. We then do the following:

1. Compute a vector $\eta = \mathit{reduce}(\phi^{m+1}, \{\psi^1, \ldots, \psi^k\})$ satisfying

- $span(\eta, \psi^1, \ldots, \psi^k) = span(\phi^{m+1}, \psi^1, \ldots, \psi^k)$, and
- $\eta_{u^i} = 0$ for all i, where $u^i = pivot(\phi^i)$,

by subtracting $(\phi^{m+1}_{u^i}/\psi^i_{u^i})\psi^i$ from ϕ^{m+1} for each i.

2. if $\eta = \mathbf{0}$, we are done (ϕ^{m+1} was linearly dependent and therefore redundant).
Otherwise, let v be its pivot element. We reduce each ψ^i such that $\psi^i_v = 0$ by subtracting $(\psi^i_v/\eta_v)\eta$, and add η to the set.

The *reduce* operation generalizes to sets of vectors by

$$reduce(A, B) := \{reduce(\eta, B) | \eta \in A\} \setminus \{\mathbf{0}\};$$

a basis in reduced echelon form for $span(A \cup B)$ is then

$$join(A, B) := reduce(A, B) \cup reduce(B, reduce(A, B)).$$

Example: Consider the vectors $\phi^1 = (1, 0, -1, 2)^T$ and $\phi^2 = (0, 1, 1, 1)^T$ in $\mathbb{R}^{\{1,\ldots,4\}}$, with pivot elements 1 and 2, respectively. A new vector $\eta = (1, 1, 1, 0)^T$ would first be reduced to $(0, 1, 2, -2)^T$ (by subtracting ϕ^1), and then to $(0, 0, 1, -3)^T = \eta'$ (by subtracting ϕ^2), with pivot element 3. Reducing ϕ^1, ϕ^2 with η', we obtain $(1, 0, 0, -1)^T$ and $(0, 1, 0, 4)^T$. We thus get the new basis $\{(1, 0, 0, -1)^T, (0, 1, 0, 4)^T, (0, 0, 1, -3)^T\}$.

The *scalar product* on $\mathbb{R}^U$ is given by $\psi \cdot \phi = \sum_{u \in U} \psi_u \phi_u$. It extends to a subspace $H \subseteq \mathbb{R}^U$ by $\psi \cdot H = \{\psi \cdot \phi \mid \phi \in H\}$. The *orthogonal complement* $H^\perp$ of a subspace $H \subseteq \mathbb{R}^U$ consists of the vectors that are *orthogonal* to those in H, i.e., all vectors ψ for which $\psi \cdot H = \{0\}$. Given a basis B in reduced echelon form for H, a basis for $H^\perp$ is obtained as follows:

Let $N \subseteq U$ be the set of all $u \in U$ which are not the pivot element of any $\phi \in B$. For each $u \in N$, define a vector ψ^u by $\psi^u_u = 1, \psi^u_v = 0$ for all $v \in N \setminus \{u\}$, and for each $\phi \in B$, $\psi^u_{pivot(\phi)} = -\phi_u/\phi_{pivot(\phi)}$. For example, given the basis $B = \{(1, 0, 0, -1)^T, (0, 1, 0, 4)^T, (0, 0, 1, -3)^T\}$, we have that $N = \{4\}$, and therefore obtain the basis vector $\psi^4 = (1, 4, 3, 1)^T$ for $span(B)^\perp$.

2.1.2 Alphabets and sequences

An alphabet is a finite set of symbols. For an alphabet Σ, Σ^* is the set of finite sequences over Σ. The empty sequence is denoted by ϵ, the composition of two sequences $v, w \in \Sigma^*$ by $v.w$, and the length of a sequence w by $|w|$.

For alphabets $\Sigma_1 \subseteq \Sigma_2$, the *projection* $w \downarrow_{\Sigma_1}$ of a sequence $w \in \Sigma_2^*$ onto Σ_1 is defined recursively by

$$\epsilon \downarrow_{\Sigma_1} = \epsilon, \ (w.a) \downarrow_{\Sigma_1} = \begin{cases} (w \downarrow_{\Sigma_1}).a & \text{if } a \in \Sigma_1, \\ w \downarrow_{\Sigma_1} & \text{otherwise.} \end{cases}$$

We assume given some some total order $<$ on Σ, and equip Σ^* with the corresponding length-lexicographical ordering given by $u <_{llex} v$ iff either

- $|u| < |v|$ or
- $|u| = |v|$, and there are $x, y, z \in \Sigma^*, a, b \in \Sigma$ with $a < b, u = xay, v = xbz$.

In particular, elements ϕ of the vector space $\mathbb{R}^U$, generated by a finite subset $U \subset \Sigma^*$, are written according to this order, i.e., $\phi = (\phi_{u^1}, \dots, \phi_{u^n})$ for $U = \{u^1, \dots, u^n\}, u^1 <_{llex} \cdots <_{llex} u^n$.

2.1.3 Communicating automata

The systems that we consider in the first half of this thesis are finite-state concurrent systems that are given as a set of communicating finite automata.

A (nondeterministic) *finite automaton* $P = (Q_P, \Sigma_P, Q^0_P, Q^e_P, T_P)$ consists of

- a finite set Q_P of locations,
- a finite alphabet Σ_P of synchronization events,
- sets $Q^0_P \subseteq Q_P$ and $Q^e_P \subseteq Q_P$ of initial and error locations, and
- a transition relation $T_P \subseteq Q_P \times \Sigma_P \times Q_P$.

When dealing with automata $P_1, \dots, P_n$, we use i as the subscript instead of P_i. We omit subscripts whenever they are clear from the context.

We denote $(q, a, r) \in T_P$ by $q \xrightarrow{a}_P r$. For a sequence $w = w_1 \dots w_n \in \Sigma^*_P$, $q \xrightarrow{w}_P r$ iff $q \xrightarrow{w_1}_P \cdots \xrightarrow{w_n}_P r$. The *language* of a location $q \in Q_P$ is the set $L(q) := \{w \in \Sigma^*_P : q^0 \xrightarrow{w}_P q \text{ for some } q^0 \in Q^0\}$; q is *reachable* iff $L(q) \neq \emptyset$. We assume in the following that our automata only contain reachable locations. For a subset $Q' \subseteq Q_P$, the language of Q', denoted by $L(Q')$, is the union of all languages of the locations in Q'. The language of an automaton P is the language of its locations, $L(P) := L(Q_P)$.

The *system automaton* $P_1 \otimes \cdots \otimes P_n$ of a system $S = (P_1, \dots, P_n)$ of finite automata is given by $(Q, \Sigma, Q^0, Q^e, \rightarrow)$, where $Q = Q_1 \times \cdots \times Q_n$, $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$, $Q^0 = Q^0_1 \times \cdots \times Q^0_n$, $Q^e = Q^e_1 \times \cdots \times Q^e_n$, and $(q_1, \dots, q_n) \xrightarrow{a} (r_1, \dots, r_n)$ iff for all $i \in \{1, \dots, n\}$ either

- $a \in \Sigma_i$ and $q_i \xrightarrow{a}_i r_i$, or
- $a \notin \Sigma_i$ and $q_i = r_i$.

Note that this corresponds to the parallel composition $\|$ of Hoare's CSP [43]. The language $L(S) = L(P_1 \otimes \cdots \otimes P_n)$ of S thus consists of all sequences w over Σ, such that, for each automaton P_i, the projection $w{\downarrow_{\Sigma_i}}$ to the alphabet Σ_i is in the language $L(P_i)$.

2.2 Pure Subsequence Invariants

Throughout this section, we assume given a finite automaton $P = (Q, \Sigma, Q^0, Q^e, T)$ and a finite, prefix-closed set $U = \{u^1, \dots, u^n\} \subset \Sigma^*$, which we call the *set of subsequences*.

2.2.1 Subsequence occurrences

Given two sequences $u = u_1 \dots u_k$ and $w = w_1 \dots w_n \in \Sigma^*$, the *set of occurrences of u as a subsequence* in w is

$$[w]_u := \{(i_1, \dots, i_k) : 1 \leq i_1 < \dots < i_k \leq n, w_{i_l} = u_l \text{ for } 1 \leq l \leq k\}.$$

The set of occurrences is equipped with the *product order*

$$(i_1, \dots, i_k) \leq (j_1, \dots, j_k) \Leftrightarrow i_l \leq j_l \text{ for all } l.$$

Example: For $w = aababb$ and $u = ab$, there are eight occurrences of u as a subsequence in w:

$$\mathbf{a}a\mathbf{b}abb,\ \mathbf{a}aba\mathbf{b}b,\ \mathbf{a}abab\mathbf{b},\ a\mathbf{ab}abb,\ a\mathbf{a}ba\mathbf{b}b,\ a\mathbf{a}bab\mathbf{b},\ aab\mathbf{ab}b,\ aab\mathbf{a}b\mathbf{b},$$

which boils down to

$$[aababb]_{ab} = \{(1,3),\ (1,5),\ (1,6),\ (2,3),\ (2,5),\ (2,6),\ (4,5),\ (4,6)\}.$$

The cardinalities of these sets define the *numbers of occurrences* $|w|_u := card([w]_u)$. These numbers can be computed recursively, using the recurrence [65]

$$|w|_\epsilon = 1, \qquad |\epsilon|_{u.b} = 0, \qquad |w.a|_{u.b} = \begin{cases} |w|_{u.b} + |w|_u & \text{if } a = b, \\ |w|_{u.b} & \text{otherwise,} \end{cases}$$

for all $u, w \in \Sigma^*, a, b \in \Sigma$. The initial case $|w|_\epsilon = 1$ may not be immediately obvious, but can be seen as a consequence of $[w]_u$ becoming the singleton set $\{()\}$ for $u = \epsilon$, independent of w.

The individual functions $w \mapsto |w|_u$ give rise to a mapping $|.|_U$ from Σ^* into $\mathbb{R}^U$ defined by $|w|_U = (|w|_{u^1}, \dots, |w|_{u^n})$. For any subset $Q' \subseteq Q$, the *subsequence hull* of Q' is the subspace $H(Q')$ of $\mathbb{R}^U$ spanned by the subsequence occurrences $\{|w|_U : w \in L(Q')\}$. Subsequence invariants are linear equalities satisfied by $|w|_U$ for all $w \in L(Q')$ or, equivalently, by all $\phi \in H(Q')$:

Definition 2.1 *A* subsequence invariant *for $Q' \subseteq Q$ over U is a vector $\phi \in \mathbb{R}^U$ such that for all $w \in L(Q')$, $\sum_{u \in U} \phi_u |w|_u = 0$.*

Note that although by this definition, subsequence invariants are always homogeneous, we will in the following often encounter inhomogeneous equations. In any such case, the constant term c is actually a shorthand for $c \cdot |w|_\epsilon$.

The subsequence invariants for Q' define a linear subspace $\Gamma(Q') \subseteq \mathbb{R}^U$, which is the orthogonal complement of $H(Q')$ in $\mathbb{R}^U$. Special cases are:

- the *local subsequence invariants* $\Gamma(q) = \Gamma(\{q\})$ at $q \in Q$,
- the *global invariants* of P, $\Gamma(P) = \Gamma(Q)$, and
- the *error conditions* $\Gamma(Q^e)$.

The spaces of the invariants satisfy the relation $\Gamma(Q') = \bigcap_{q \in Q'} \Gamma(q)$.

Requiring invariants to be linear equalities may appear restrictive. In the remainder of this section we illustrate the expressive power of subsequence invariants by translating two useful types of invariants, *event conditions* and *disjunctive* invariants, to equivalent global subsequence invariants.

2.2.2 Event conditions.

Properties (1)–(3) of the arbiter tree discussed in the introduction are examples of situational constraints, stating that a linear equality over the numbers $|w|_u$ should hold whenever some event $a \in \Sigma$ occurs (equivalently, since the invariants are required to hold for all possible traces, whenever a *can* occur). This equality must then be in the space $\Gamma(a) := \Gamma(E_a)$ of *event conditions* for a, where $E_a = \{q \in Q : (q, a, r) \in T \text{ for some } r\}$ is the set of locations in which an outgoing a-transition exists. We can resolve event conditions to obtain a global statement, using subsequences at most one symbol longer than those in U, using the following theorem:

Theorem 2.2 *Let $a \in \Sigma$ and $E_a := \{q \in Q : \exists r, q \xrightarrow{a} r\}$. Then $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in L(E_a)$ if and only if $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ for all $w \in L(P)$.*

Proof:

$\Rightarrow$: Proof by induction over $w \in L(P)$, using the assumption $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in L(E_a)$.

For $w = \epsilon$, the claim is obviously true since $|\epsilon|_{u.a} = 0$ for all u and a. For a word $w = v.b \in L(P)$, first note that $L(P)$ is prefix closed, so that $v \in L(P)$ can be assumed to satisfy the induction hypothesis.

If $b \neq a$, then $|v.b|_{u.a} = |w|_{u.a}$ for all u, and the claim follows immediately from the induction hypothesis.

If $b = a$, we have $|v.a|_{u.a} = |w|_{u.a} + |w|_u$ for all u, and therefore

$\sum_{u \in U} \phi_u |v.a|_{u.a} = \sum_{u \in U} \phi_u |v|_{u.a} + \sum_{u \in U} \phi_u |v|_u$.

The first summand is again zero by the induction hypothesis. By definition of E_a, in order for $v.a$ to be in $L(P)$, v must be in $L(E_a)$, and therefore, by the assumption, the second summand is also zero.

$\Leftarrow$: Assume that $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ for all $w \in L(P)$, and that there is some $w \in L(E_a)$ for which $\sum_{u \in U} \phi_u |w|_u \neq 0$. It follows from $w \in L(E_a)$ that both w and $w.a$ are in $L(P)$. But then, by the second assumption,

$\sum_{u \in U} \phi_u |w.a|_{u.a} = \sum_{u \in U} \phi_u |w|_{u.a} + \sum_{u \in U} \phi_u |w|_u \neq \sum_{u \in U} \phi_u |w|_{u.a}$.

In particular, they cannot both be zero, contradicting the first assumption.

□

Thus, for example, the condition that the left child must have returned all grants it has received whenever a grant is given to the right child, i.e. that $|w|_{gr_1} = |w|_{rel_1}$ for all $w \in L(E_{gr_2})$, is equivalent to requiring $|w|_{gr_1.gr_2} = |w|_{rel_1.gr_2}$ for all $w \in L(P)$.

2.2.3 Resolving disjunctions.

Consider now the fourth statement in the introductory example: The differences between the corresponding numbers of grants and releases only take values in $\{0, 1\}$. Such a *disjunctive condition* can be translated in two steps into an equivalent linear equation: The condition is first transformed into a polynomial equation (Step 1), and then reduced, using algebraic dependencies, to an equivalent linear equation (Step 2).

The first step is standard: The condition

$$(\sum_{u\in U} \phi^1_u |w|_u = 0) \vee (\sum_{u\in U} \phi^2_u |w|_u = 0)$$

is equivalent to

$$(\sum_{u\in U} \phi^1_u |w|_u) \cdot (\sum_{u\in U} \phi^2_u |w|_u) = 0.$$

For the second step, i.e. the transformation of the resulting polynomial equation into a linear equation, we define, as an auxiliary notion, the set of *coverings* of $x \in \Sigma^*$ by u and v to be

$$\begin{aligned}[x]_{u,v} := \{((i_1,\dots,i_k),(j_1,\dots,j_m)) : {} & i_1 < \dots < i_k, j_1 < \dots < j_m, \\ & u = x_{i_1}\dots x_{i_k}, v = x_{j_l}\dots x_{j_m}, \\ & \{i_1,\dots,i_k,j_1,\dots,j_m\} = \{1,\dots,|x|\}\},\end{aligned}$$

i.e., the set of pairs of occurrences of u and v as subsequences of x such that every index in $1,\dots,|x|$ is used in at least one of them.

Example: Let $x = aabaa, u = aaa, v = aba$. The sets of occurrences of u and v in x are $\{(1,2,4),(1,2,5),(1,4,5),(2,4,5)\}$ and $\{(1,3,4),(1,3,5),(2,3,4),(2,3,5)\}$, respectively. Then $[x]_{u,v}$ consists of those pairs from $[x]_u \times [x]_v$ satisfying the covering condition:

$$\begin{aligned}[aabaa]_{aaa,aba} = {} & \{((1,2,4),(1,3,5)),((1,2,4),(2,3,5)),((1,2,5),(1,3,4)),((1,2,5),(2,3,4)), \\ & ((1,4,5),(2,3,4)),((1,4,5),(2,3,5)),((2,4,5),(1,3,4)),((2,4,5),(1,3,5))\}.\end{aligned}$$

Let $|w|_{u,v} = card([w]_{u,v})$ denote the number of coverings, which can be computed recursively as follows:

$$|w|_{u,\epsilon} = |w|_{\epsilon,u} = \begin{cases} 1 & \text{if } u = w, \\ 0 & \text{otherwise,}\end{cases} \qquad |\epsilon|_{u,v} = \begin{cases} 1 & \text{if } u = v = \epsilon, \\ 0 & \text{otherwise,}\end{cases}$$

$$|w.a|_{u.b,v.c} = \begin{cases} |w|_{u,v} + |w|_{u.b,v} + |w|_{u,v.c} & \text{if } b = a = c, \\ |w|_{u,v.c} & \text{if } b = a \neq c, \\ |w|_{u.b,v} & \text{if } b \neq a = c, \\ 0 & \text{if } b \neq a \neq c.\end{cases}$$

It is easy to see that for every $u, v \in \Sigma^*$, the set $C(u,v) := \{x \in \Sigma^* : [x]_{u,v} \neq \emptyset\}$ of sequences coverable by u and v is finite, since it cannot contain sequences longer than $|u| + |v|$.

Theorem 2.3 *(See Theorem 6.3.18 in [65] for an equivalent statement to (2))*

1. *For all $u, v, w \in \Sigma^*$, there is a bijection between $[w]_u \times [w]_v$ and $\biguplus_{x\in C(u,v)}([x]_{u,v} \times [w]_x)$, and therefore,*

2. *For all $u, v, w \in \Sigma^*$, $|w|_u \cdot |w|_v = \sum_{x\in C(u,v)} |x|_{u,v} \cdot |w|_x$.*

Proof:

1. Let $u, v, w \in \Sigma^*$. For two occurrences $i = (i_1, \ldots, i_m) \in [w]_u$ and $j = (j_1, \ldots, j_n) \in [w]_v$, the *shuffle* tuple $s(i,j) = (s_1, \ldots, s_p)$ is the unique ordered tuple containing all elements of the set $\{i_1, \ldots, i_m, j_1, \ldots, j_n\}$. It gives rise to ordered *embedding* tuples $l(i,j) = (l_1, \ldots, l_m)$ and $r(i,j) = (r_1, \ldots, r_n)$ defined uniquely by $i_k = s_{l_k}$ for $k = 1, \ldots, m$ and $j_k = s_{r_k}$ for $k = 1, \ldots, n$.

   For the word $x = w_{s_1} \ldots w_{s_p}$, we obviously have $s \in [w]_x$. Also, the embedding tuples satisfy $(l(i,j), r(i,j)) \in [x]_{u,v}$, because

   - for $k = 1, \ldots, m$, $x_{l_k} = w_{s_{l_k}} = w_{i_k} = u_k$,
   - for $k = 1, \ldots, n$, $x_{r_k} = w_{s_{r_k}} = w_{j_k} = v_k$,
   - $\{l_1, \ldots, l_m, r_1, \ldots, r_n\} = \{1, \ldots, p\}$.

   Together, l, r, s map each pair of occurrences $(i,j) \in [w]_u \times [w]_v$ to an element $(l(i,j), r(i,j), s(i,j)) \in \biguplus_{x \in C(u,v)}([x]_{u,v} \times [w]_x)$. We are going to show that this is a bijection.

   For injectivity, assume that for two pairs of tuples (i^1, j^1) and (i^2, j^2), $(l(i^1,j^1), r(i^1,j^1), s(i^1,j^1)) = (l(i^2,j^2), r(i^2,j^2), s(i^2,j^2))$.
   Then for $k = 1, \ldots, m$,
   $i^1_k = s_{l_k(i^1,j^1)}(i^1,j^1) = s_{l_k(i^2,j^2)}(i^2,j^2) = i^2_k$,
   and for $k = 1, \ldots, n$,
   $j^1_k = s_{r_k(i^1,j^1)}(i^1,j^1) = s_{r_k(i^2,j^2)}(i^2,j^2) = j^2_k$.
   Therefore $(i^1, j^1) = (i^2, j^2)$.

   For surjectivity, let $x \in C(u,v)$, $s' = (s_1, \ldots, s_p) \in [w]_x$, and $(l', r') = ((l'_1, \ldots, l'_m), (r'_1, \ldots, r'_n)) \in [x]_{u,v}$. Defining $i = (i_1, \ldots, i_m) = (s'_{l'_1}, \ldots, s'_{l'_m})$ and $j = (j_1, \ldots, j_n) = (s'_{r'_1}, \ldots, s'_{r'_n})$, it is easy to check that

   - since $(l', r') \in [x]_{u,v}$, we have $\{l'_1, \ldots, l'_m, r'_1, \ldots, r'_n\} = \{1, \ldots, p\}$, and therefore $\{i_1, \ldots, i_m, j_1, \ldots, j_n\} = \{s'_1, \ldots, s'_p\}$, implying $s(i,j) = s'$;
   - for $k = 1, \ldots, m$, $l_k(i,j)$ is defined to be the unique index such that $i_k = s_{l_k(i,j)}(i,j)$; in this case, $i_k = s'_{l'_k}$, for all k, implies $l(i,j) = l'$;
   - analogously, we get $r(i,j) = r'$.

2. Immediately from the first item, we have

$$\begin{aligned} |w|_u |w|_v &= card([w]_u \times [w]_v) \\ &= card(\biguplus_{x \in C(u,v)} ([x]_{u,v} \times [w]_x)) \\ &= \sum_{x \in C(u,v)} |x|_{u,v} |w|_x. \end{aligned}$$

□

Simple examples for Theorem 2.3 are the equalities $|w|_a^2 = 2|w|_{aa} + |w|_a$ and $|w|_a|w|_b = |w|_{ab} + |w|_{ba}$. For $u = ab$ and $v = ba$, we obtain the equality $|w|_{ab}|w|_{ba} = |w|_{aba} + |w|_{bab} + |w|_{abab} + 2|w|_{abba} + 2|w|_{baab} + |w|_{baba}$.

The degree k polynomial equation $p(|w|_{u^1}, \ldots, |w|_{u^n}) = 0$ resulting from Step 1 can then be transformed into a linear equation using the equalities from Theorem 2.3. This linear equation involves subsequences of length up to $k \cdot l$, where l is the maximum length of any $u \in U$.

Example: For property (4) from the introduction, we obtain

$$\begin{aligned} & |w|_{gr_i} - |w|_{rel_i} \in \{0,1\} \\ & \quad \Leftrightarrow \quad (|w|_{gr_i} - |w|_{rel_i})(|w|_{gr_i} - |w|_{rel_i} - 1) = 0 \\ & \quad \Leftrightarrow \quad |w|^2_{gr_i} - 2|w|_{gr_i}|w|_{rel_i} + |w|^2_{rel_i} - |w|_{gr_i} + |w|_{rel_i} = 0 \\ & \quad \Leftrightarrow \quad |w|_{gr_i.gr_i} + |w|_{rel_i.rel_i} + |w|_{rel_i} = |w|_{gr_i.rel_i} + |w|_{rel_i.gr_i}. \end{aligned}$$

This technique can also be applied to more complicated constraints: An alternative characterization of Arbiter 1 is given by the requirement that for all $w \in L(P)$,

$$\begin{pmatrix} |w|_{gr_0} - |w|_{rel_0} \\ |w|_{gr_1} - |w|_{rel_1} \\ |w|_{gr_2} - |w|_{rel_2} \end{pmatrix} \in \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \right\}.$$

Note that the possible values for the linear expressions are mutually dependent. The set of vectors on the right-hand side can be characterized as the set of all $(x, y, z)^T$ for which $x^2 - x, y^2 - y, z^2 - z, xy - y, xz - z$ and yz are all zero. Plugging $x = |w|_{gr_0} - |w|_{rel_0}, y = |w|_{gr_1} - |w|_{rel_1}, z = |w|_{gr_2} - |w|_{rel_2}$ into these polynomials using Theorem 2.3, we can again obtain an equivalent set of linear subsequence constraints. In general, we have:

Theorem 2.4 *Let $card(U) = n$, $M \in \mathbb{R}^{k \times n}$, and $\phi^1, \ldots, \phi^m \in \mathbb{R}^k$. Then the constraint given by $M|w|_U \in \{\phi^1, \ldots, \phi^m\}$ is equivalent to a finite set of linear subsequence constraints involving subsequences of length $\leq l \cdot m$, where l is the maximum length of any $u \in U$.*

Proof: Using standard methods from algebraic geometry, one can find polynomials $p_1, \ldots, p_N$ of degree at most m, such that

$$\{\phi^1, \ldots, \phi^m\} = \{\phi \in \mathbb{R} : p_i(\phi) = 0 \text{ for all } i\}.$$

One straightforward, though usually inefficient, way of finding such a set of polynomials consists of:

- picking for each i a system $l^i_1 = 0, \ldots, l^i_k = 0$ of linear equations whose sole solution is ϕ^i, so that the solutions of $\bigvee_{i=1}^m \bigwedge_{j=1}^k l^i_j = 0$ are $\phi^1, \ldots, \phi^m$,

- transforming this formula into *conjunctive* normal form, and

- obtaining from each conjunct $l^1_{j_1} = 0 \vee \cdots \vee l^m_{j_m} = 0$ the equivalent polynomial equation $l^1_{j_1} \cdot \cdots \cdot l^m_{j_m} = 0$.

In the example, choosing equalities that hold for as many points as possible, this gives:

$$\begin{aligned}
&\{(0,0,0)^T,(1,0,0)^T,(1,0,1)^T,(1,1,0)^T\}\\
&\quad=\{(x,y,z)^T \mid && (x=y+z \wedge y=0 \wedge z=0) \vee (x=1 \wedge y=0 \wedge z=0) \vee\\
&&& (x=y+z \wedge x=1 \wedge y=0) \vee (x=y+z \wedge x=1 \wedge z=0)\}\\
&\quad=\{(x,y,z)^T \mid && (x=y+z \vee x=1) \wedge (x=y+z \vee y=0) \wedge (x=y+z \vee z=0) \wedge\\
&&& (x=1 \vee y=0) \wedge (x=1 \vee z=0) \wedge (y=0 \vee z=0)\}\\
&\quad=\{(x,y,z)^T \mid && (x-y-z)(x-1)=(x-y-z)y=(x-y-z)z=\\
&&& (x-1)y=(x-1)z=yz=0\}\\
&\quad=\{(x,y,z)^T \mid && x^2-x=y^2-y=z^2-z=xy-y=xz-z=yz=0\}
\end{aligned}$$

As before, by Theorem 2.3, there is an equivalent linear subsequence invariant for each of the conditions $p_i(|w|_U)=0$, involving subsequences no longer than $l \cdot \deg(p_i) \le l \cdot m$. □

2.3 Phased Subsequences

In this section, we generalize subsequences to obtain more expressive classes of invariants. We use an extension of subsequences which can additionally restrict the sets of events that may occur between the events of the actual sequence. The resulting *phased* subsequence invariants can, for example, express properties such as bounded overtaking.

Phased subsequence invariants behave almost as nicely as pure subsequence invariants. In particular, they can also be efficiently computed for individual processes, and allow similar handling of event conditions and disjunctions.

2.3.1 Phased subsequence occurrences

Definition 2.5 *A* phased subsequence $u=\sigma_0 e_1 \sigma_1 \ldots e_n \sigma_n$ *over an alphabet* Σ *of events is given by*

- *required events* $e_1,\ldots,e_n \in \Sigma$, *and*
- *sets* $\sigma_0,\ldots,\sigma_n \subseteq \Sigma$ *of forbidden events.*

We define the length $|u|$ *of a phased subsequence to be the number* n *of required events. For a word* $w \in \Sigma^*$, *the set* $[w]_u$ *of* occurrences *of* u *in* w *consists of all index tuples* $i=(i_1,\ldots,i_n)$ *satisfying*

- $1 \le i_1 < \cdots < i_n \le |w|$,
- $w_{i_k}=e_k$ *for* $1 \le k \le n$,
- $w_j \in \Sigma \setminus \sigma_k$ *for* $i_k < j < i_{k+1}, 0 \le k \le n$,

where we extend i *by* $i_0=0, i_{n+1}=|w|+1$. *We again denote the* number *of occurrences by*

$$|w|_u := card([w]_u).$$

Example: The occurrences of the phased subsequence $u = \emptyset\, a\{b\}a\, \emptyset$ in w correspond to those pairs of occurrences of a in w with no intervening b. In particular, for $w = aabacaab$, we get $|w|_u = 4$, where the occurrences are

$$\mathbf{aa}bacaab,\ aab\mathbf{a}c\mathbf{a}ab,\ aab\mathbf{a}ca\mathbf{a}b,\ aabac\mathbf{aa}b,$$

corresponding to the index tuples

$$\{(1,2),(4,6),(4,7),(6,7)\}.$$

Note that phased subsequences with $\sigma_i = \emptyset$ for all i are equivalent to pure subsequences; we will usually omit $\emptyset$, so that for example the above u will simply be written as $a\{b\}a$.

Just as for pure subsequence invariants, there are linear recurrences which can be used to compute $|w|_u$: For an arbitrary word $w \in \Sigma^*$, phased subsequence u, events $a, e \in \Sigma$ and set of events $\sigma \subseteq \Sigma$,

$$|w|_\sigma = \begin{cases} 1 & w \in (\Sigma \setminus \sigma)^* \\ 0 & \text{otherwise} \end{cases}, \qquad |\epsilon|_{u.e.\sigma} = 0,$$

$$|w.a|_{u.e.\sigma} = \begin{cases} |w|_{u.e.\sigma} & a \notin \sigma \\ 0 & \text{otherwise} \end{cases} + \begin{cases} |w|_u & a = e \\ 0 & \text{otherwise.} \end{cases}$$

Subsequence invariants then generalize in a straightforward manner: Let $P = (Q, \Sigma, Q^0, Q^e, T)$ be a finite automaton, and let U be a set of phased subsequences which is prefix-closed, in the sense that for each $u.e.\sigma \in U$, u is also contained in U. We again assume given some order on U. For a word w, we get the *phased subsequence image* $|w|_U = (|w|_{u^1}, \ldots, |w|_{u^n}) \in \mathbb{R}^U$, where $u^1, \ldots, u^n$ are the elements of U ordered according to $<$. A *phased subsequence invariant* of $Q' \subseteq Q$ is then a vector $\phi \in \mathbb{R}^U$ such that for all $w \in L(Q')$, $\sum_{u\in U} \phi_u |w|_u = 0$.

Example: One common requirement for automated teller machines is that after three consecutive failures to enter the correct PIN, the machine should refuse the transaction. One way of specifying this kind of requirement is

$$|w|_{fail\{ok\}fail\{ok\}fail.proceed} = 0,$$

i.e. whenever a *proceed* event occurs, the number of occurrences of $fail\{ok\}fail\{ok\}fail$ must be zero. This is again an example of an event condition, which we can treat just as for pure subsequence invariants:

Theorem 2.6 *Let $a \in \Sigma$ and $E_a := \{q \in Q : q \xrightarrow{a} r$ for some $r \in Q\}$. Then $\sum_{u\in U} \phi_u |w|_u = 0$ for all $w \in L(E_a)$ if and only if $\sum_{u\in U} \phi_u |w|_{u.a} = 0$ for all $w \in L(P)$.*

Proof: We proceed just as in the case of pure subsequence invariants, keeping in mind that $u.a$ is a shorthand for $u.a.\emptyset$:

$\Rightarrow$: Induction over $w \in L(P)$. For $w = \epsilon$, we have $|w|_{u.a} = 0$ by definition. For the step $w \to w.b$, we consider the two cases:

If $b \neq a$, then by the recurrence and the induction hypothesis

$$|w.b|_{u.a} = \begin{cases} |w|_{u.a} & a \notin \emptyset \\ 0 & \text{otherwise} \end{cases} = |w|_{u.a} = 0.$$

If $b = a$, then

$$|w.b|_{u.a} = \begin{cases} |w|_{u.a} & a \notin \emptyset \\ 0 & \text{otherwise} \end{cases} + |w|_u = |w|_{u.a} + |w|_u = |w|_u,$$

which is again 0 since by definition of E_a, w must be in $L(E_a)$.

$\Leftarrow$: Assume that $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ for all $w \in L(P)$, and that there is some $w \in L(E_a)$ for which $\sum_{u \in U} \phi_u |w|_u \neq 0$.

It follows from $w \in L(E_a)$ that both w and $w.a$ are in $L(P)$. But then, by the second assumption, and using $a \notin \emptyset$,

$\sum_{u \in U} \phi_p |w.a|_{u.a} = \sum_{u \in P} \phi_u |w|_{u.a} + \sum_{u \in U} \phi_u |w|_u \neq \sum_{u \in U} \phi_u |w|_{u.a}$.

In particular, they cannot both be zero, contradicting the first assumption.

□

2.3.2 Resolving disjunctions.

Slightly more complicated is the proof that phased subsequence invariants are also able of expressing disjunctive invariants. Just as in Section 2.2.3, the first step is to transform the disjunction

$$(\sum_{u \in U} \phi^1_u |w|_u = 0) \vee (\sum_{u \in U} \phi^2_u |w|_u = 0)$$

into the equivalent polynomial equation

$$(\sum_{u \in U} \phi^1_u |w|_u) \cdot (\sum_{u \in U} \phi^2_u |w|_u) = 0.$$

In order to obtain from this an equivalent linear equation, we extend the notion of covering from Section 2.2.3 to phased subsequences:

Definition 2.7 *Let $u = \sigma^u_0 e^u_1 \sigma^u_1 \dots e^u_l \sigma^u_l$, $v = \sigma^v_0 e^v_1 \sigma^v_1 \dots e^v_m \sigma^v_m$, and $x = \sigma^x_0 e^x_1 \sigma^x_1 \dots e^x_n \sigma^x_n$ be phased subsequences over Σ. A* covering *of x by u and v is a pair of index tuples $((i_1, \dots, i_l), (j_1, \dots, j_m))$ such that*

1. *$((i_1, \dots, i_l), (j_1, \dots, j_m))$ is a covering of the pure subsequence $e^x_1 \dots e^x_n$ by $e^u_1 \dots e^u_l$ and $e^v_1 \dots e^v_m$ in the sense of Section 2.2.3:*
   - *(a) $1 \leq i_1 < \dots < i_l \leq n$, $1 \leq j_1 < \dots < j_m \leq n$,*
   - *(b) $e^x_{i_p} = e^u_p$ for $1 \leq p \leq l$, $e^x_{j_p} = e^v_p$ for $1 \leq p \leq m$,*
   - *(c) $\{i_1, \dots, i_l\} \cup \{j_1, \dots, j_m\} = \{1, \dots, n\}$,*

u	$\emptyset$	b	$\{c\}$	b	$\{c\}$			a	$\emptyset$
x	$\{c\}$	b	$\{c\}$	b	$\{a,c\}$	a	$\{b,c\}$	a	$\{b\}$
v	$\{c\}$			b	$\{a\}$	a	$\{b\}$		

Figure 2.1: A covering of phased subsequences: In addition to the covering conditions for pure subsequences, events in x which are not used by u (resp. v) must be allowed by the relevant set of forbidden events. For example, the first b in x is not used by v, but is allowed by $\sigma_0^v = \{c\}$.

2. *The sets of forbidden events in x are the unions of those in u, v at the corresponding positions, and the interleaving of the e_p^u and e_q^v is consistent with the σ_q^v, σ_p^u.*
To make this precise, set $i_0 = j_0 = 0$ and $i_{l+1} = j_{m+1} = n+1$, and define $b_i(p) = \max\{q \mid i_q \leq p\}$ and $b_j(p) = \max\{q \mid j_q \leq p\}$ for $p \in \{0, \dots, n\}$. Then i, j need to satisfy:

   (a) $\sigma_p^x = \sigma_{b_i(p)}^u \cup \sigma_{b_j(p)}^v$ *for* $0 \leq p \leq n$,

   (b) for $0 \leq q \leq l$ *and* $i_q < p < i_{q+1}$, $e_p^x \notin \sigma_q^u$,

   (c) for $0 \leq q \leq m$ *and* $j_q < p < j_{q+1}$, $e_p^x \notin \sigma_q^v$.

Example: Consider the phased subsequences in Figure 2.1,

$$u = \emptyset\, b\{c\}b\{c\}a\,\emptyset, \qquad v = \{c\}b\{a\}a\{b\}, \qquad x = \{c\}b\{c\}b\{a,c\}a\{b,c\}a\{b\}.$$

The pair of index tuples $i = (1,2,4), j = (2,3)$ is a covering of x by u and v, because:

1. $((1,2,4),(2,3))$ is a (pure subsequence) covering of $bbaa$ by bba and ba,

2. the conditions involving the σ_k are satisfied: The functions b_i, b_j in this case take the values
$b_i : [0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 2, 4 \mapsto 3]$,
$b_j : [0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2]$, and

   (a) $\sigma_0^x = \{c\} = \emptyset \cup \{c\} = \sigma_0^u \cup \sigma_0^v$,
   $\sigma_1^x = \{c\} = \{c\} \cup \{c\} = \sigma_1^u \cup \sigma_0^v$,
   $\sigma_2^x = \{a,c\} = \{c\} \cup \{a\} = \sigma_2^u \cup \sigma_1^v$,
   $\sigma_3^x = \{b,c\} = \{c\} \cup \{b\} = \sigma_2^u \cup \sigma_2^v$,
   $\sigma_4^x = \{b\} = \emptyset \cup \{b\} = \sigma_3^u \cup \sigma_2^v$,

   (b) $e_3^x = a \notin \{c\} = \sigma_2^u$,

   (c) $e_1^x = b \notin \{c\} = \sigma_0^v$, and $e_4^x = a \notin \{b\} = \sigma_2^v$.

We let $[x]_{u,v}$ be the set of all coverings of x by u and v, and $|x|_{u,v} = card([x]_{u,v})$ its cardinality. Obviously, the set $C(u,v) := \{x : [x]_{u,v} \neq \emptyset\}$ must be finite for any u, v. Again, there is a linear recurrence which allows the values $|x|_{u,v}$ to be computed recursively:

$$|\sigma^x|_{u,v} = \begin{cases} 1 & \text{if } u = \sigma^u, v = \sigma^v \text{ with } \sigma = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise,} \end{cases}$$

$$|x.e^x.\sigma^x|_{u.e^u.\sigma^u,\sigma^v} = \begin{cases} |x|_{u,\sigma^v} & \text{if } e^x = e^u \notin \sigma^v, \sigma^x = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise,} \end{cases}$$

$$|x.e^x.\sigma^x|_{\sigma^u,v.e^v.\sigma^v} = \begin{cases} |x|_{\sigma^u,v} & \text{if } e^x = e^v \notin \sigma^u, \sigma^x = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise,} \end{cases}$$

$$\begin{aligned} |x.e^x.\sigma^x|_{u.e^u.\sigma^u,v.e^v.\sigma^v} &= \begin{cases} |x|_{u,v.e^v.\sigma^v} & \text{if } e^x = e^u \notin \sigma^v, \sigma^x = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise} \end{cases} \\ &+ \begin{cases} |x|_{u.e^u.\sigma^u,v} & \text{if } e^x = e^v \notin \sigma^u, \sigma^x = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise} \end{cases} \\ &+ \begin{cases} |x|_{u,v} & \text{if } e^x = e^u = e^v, \sigma^x = \sigma^u \cup \sigma^v \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

We can now state the analogue of Theorem 2.3 for phased subsequences:

Theorem 2.8 *1. For all $w \in \Sigma^*$ and all phased subsequences u, v over Σ, there is a bijection between $[w]_u \times [w]_v$ and $\biguplus_{x \in C(u,v)}([x]_{u,v} \times [w]_x)$, and therefore,*

2. For all $w \in \Sigma^*$ and all phased subsequences u, v over Σ, $|w|_u |w|_v = \sum_{x \in C(u,v)} |x|_{u,v} |w|_x$.

Proof:

1. Let $w \in \Sigma^*$, and let u, v be two phased subsequences over Σ. For two occurrences $i = (i_1, \ldots, i_m) \in [w]_u$ and $j = (j_1, \ldots, j_n) \in [w]_v$, the *shuffle* tuple $s(i,j) = (s_1, \ldots, s_t)$ is the unique ordered tuple containing all elements of the set $\{i_1, \ldots, i_m, j_1, \ldots, j_n\}$. It gives rise to ordered *embedding* tuples $l(i,j) = (l_1, \ldots, l_m)$ and $r(i,j) = (r_1, \ldots, r_n)$ defined uniquely by $i_k = s_{l_k}$ for $k = 1, \ldots, m$ and $j_k = s_{r_k}$ for $k = 1, \ldots, n$. We also use the bottom functions $b_l(p) = \max\{q : l_q \le p\}$ and $b_r(p) = \max\{q : r_q \le p\}$ for $1 \le p \le t$, where again we set $i_0 = j_0 = 0$, such that $b_l(p) = 0$ for $p < l_1$, $b_r(p) = 0$ for $p < r_1$.

   Define a new phased subsequence $x = \sigma_0^x e_1^x \sigma_1^x \ldots e_t^x \sigma_t^x$ by

   - $e_p^x = w_{s_p}$ for $1 \le p \le t$,
   - $\sigma_p^x = \sigma_{b_l(p)}^u \cup \sigma_{b_r(p)}^v$ for $0 \le p \le t$.

   We will show first that s is an occurrence of x in w, and (l, r) is a covering of x by u and v.

   - $s \in [w]_x$:
     The definitions of s and x directly give us
     - $1 \le \min(i_1, j_1) = s_1 < \cdots < s_t = \max(i_m, j_n) \le |w|$, and
     - $w_{s_p} = e_p^x$ for $1 \le p \le t$.

For the intermediate words, let $s_p < q < s_{p+1}$ for some p. Then, by the definition of b_l and b_r,

$$i_{b_l(p)} < q < i_{b_l(p)+1} \text{ and } j_{b_r(p)} < q < j_{b_r(p)+1}.$$

Since i and j are occurrences of u and v, it follows that $w_q \notin \sigma^u_{b_l(p)}$ and $w_q \notin \sigma^v_{b_r(p)}$. Therefore, by the definition of σ^x_p,

$$w_q \in \Sigma \setminus \sigma^x_p.$$

- $(l, r) \in [x]_{u,v}$:
  For the covering conditions, we get directly from the definitions of s, l and r that
  - $1 \leq l_1 < \cdots < l_m \leq t$ and $1 \leq r_1 < \cdots < r_n \leq t$,
  - $x_{l_p} = w_{s_{l_p}} = w_{i_p} = u_p$ for $1 \leq p \leq m$ and $x_{r_p} = w_{s_{r_p}} = w_{j_p} = v_p$ for $1 \leq p \leq n$,
  - For $1 \leq p \leq t$, there is some q such that $s_p = i_q$, and therefore $p = l_q$, or $s_p = j_q$, and therefore $p = r_q$. Since $1 \leq l_q \leq t$ and $1 \leq r_q \leq t$ for all q, $\{l_1, \ldots, l_m, r_1, \ldots, r_n\} = \{1, \ldots t\}$.

  From the definition of σ^x_p,
  - $\sigma^x_p = \sigma^u_{b_l(p)} \cup \sigma^v_{b_r(p)}$.
  - If $l_q < p < l_{q+1}$, then $i_q < s_p < i_{q+1}$, and, since i is an occurrence of u, $e^x_p = w_{s_p} \notin \sigma^u_q$.
  - analogously, if $r_q < p < r_{q+1}$, then $e^x_p \notin \sigma^v_q$.

Together, l, r, s map each pair of occurrences $(i, j) \in [w]_u \times [w]_v$ to an element $(l(i,j), r(i,j), s(i,j)) \in \biguplus_{x \in C(u,v)}([x]_{u,v} \times [w]_x)$.

Bijectivity can then be proved exactly as in the proof of theorem 2.3:

For injectivity, assume that for two pairs of tuples (i^1, j^1) and (i^2, j^2), $(l(i^1, j^1), r(i^1, j^1), s(i^1, j^1)) = (l(i^2, j^2), r(i^2, j^2), s(i^2, j^2))$.

Then for $k = 1, \ldots, m$,

$i^1_k = s_{l_k(i^1,j^1)}(i^1, j^1) = s_{l_k(i^2,j^2)}(i^2, j^2) = i^2_k$,

and for $k = 1, \ldots, n$,

$j^1_k = s_{r_k(i^1,j^1)}(i^1, j^1) = s_{r_k(i^2,j^2)}(i^2, j^2) = j^2_k$.

Therefore $(i^1, j^1) = (i^2, j^2)$.

For surjectivity, let $x \in C(u, v)$, $s' = (s_1, \ldots, s_u) \in [w]_x$, and $(l', r') = ((l'_1, \ldots, l'_m), (r'_1, \ldots, r'_n)) \in [x]_{u,v}$. Defining $i = (i_1, \ldots, i_m) = (s'_{l'_1}, \ldots, s'_{l'_m})$ and $j = (j_1, \ldots, j_n) = (s'_{r'_1}, \ldots, s'_{r'_n})$, it is easy to check that

- since $(l', r') \in [x]_{u,v}$, we have $\{l'_1, \ldots, l'_m, r'_1, \ldots, r'_n\} = \{1, \ldots, p\}$, and therefore $\{i_1, \ldots, i_m, j_1, \ldots, j_n\} = \{s'_1, \ldots, s'_u\}$, implying $s(i, j) = s'$;
- for $k = 1, \ldots, m$, $l_k(i, j)$ is defined to be the unique index such that $i_k = s_{l_k(i,j)}(i, j)$; in this case, $i_k = s'_{l'_k}$, for all k, implies $l(i, j) = l'$;
- analogously, we get $r(i, j) = r'$.

2. Immediately from the first item, we have

$$\begin{aligned}
|w|_u|w|_v &= card([w]_u \times [w]_v) \\
&= card(\biguplus_{x\in C(u,v)} ([x]_{u,v} \times [w]_x)) \\
&= \sum_{x\in C(u,v)} |x|_{u,v}|w|_x.
\end{aligned}$$

$\square$

Chapter 3

Computing Subsequence Invariants

In this chapter, we present algorithms that automatically compute all invariants of an automaton for a given set of subsequences. Since the set of invariants is in general infinite, it is represented algebraically by a finite set of generators. Based on the synthesis algorithms, we propose the following verification technique for subsequence invariants:

To prove a desired system property φ, we first choose, for each process, a set U of relevant subsequences and then synthesize a basis of the subsequence invariants over U. The invariants computed for each individual process translate to invariants of the system. If φ is a linear combination of the system invariants, we know that φ itself is a valid invariant.

The only manual step in this technique is the choice of an appropriate set of subsequences, which depends on the complexity of the interaction between the processes. A practical approach is therefore to begin with a small set of subsequences and then incrementally compute a growing set of invariants based on a growing set of subsequences until φ is proved.

3.1 Computing Process Invariants

In this section, we present two algorithms for computing the subsequence invariants of a given finite automaton $P = (Q, \Sigma, Q^0, Q^e, T)$ with respect to a finite, prefix-closed set U of subsequences. The first algorithm is generally applicable. The second algorithm is a more efficient solution that is applicable if the state graph is strongly connected, and the subsequences are all pure.

The underlying principle is the same for both algorithms: We use a fixpoint iteration to compute, for each location q, the subsequence hull $H(q) = \mathit{span}(|w|_U : w \in L(q))$. The tuple formed by the $H(q)$ is the minimal tuple of the form $(V_q)_{q \in Q}$ such that for each initial q, V_q contains $|\epsilon|_U$, and for all transitions $q \xrightarrow{a} r$, V_r contains the image of V_q under the linear transformation F_a which represents the effect of appending an a, i.e. which satisfies $F_a|w|_U = |w.a|_U$ for all w. The orthogonal complement of the sum of the H_q is the set of subsequence invariants.

```
Data: Automaton $P = (Q, \Sigma, q^0, T)$, finite prefix-closed $U \subset \Sigma^*$
Result: Bases $B_q$ for the subspaces $H(q) = \mathit{span}(|w|_U : w \in L(q))$
// Initialization:
foreach $q \in Q$ do $B_q := \emptyset$;
// $B_{q^0}$ initially contains $\{|\epsilon|_U\}$
$B_{q^0} := \{|\epsilon|_U\}$;
// The open list, containing pairs $(q, \psi)$ to be explored
$O := \{(q^0, |\epsilon|_U)\}$;
// Basis construction:
while $O \neq \emptyset$ do
    take $(q, \psi)$ from $O$;
    foreach $q \xrightarrow{a} r$ do
        $\eta := F_a \psi$;
        begin reduce $\eta$ with $B_r$:
            foreach $\phi \in B_r$ do
                $v := \mathit{pivot}(\phi)$;
                $\eta := \eta - (\eta_v / \phi_v)\phi$;
        end
        if $\eta \neq \mathbf{0}$ then
            $v := \mathit{pivot}(\eta)$;
            foreach $\phi \in B_r$ do
                $\phi := \phi - (\phi_v / \eta_v)\eta$;
            $B_r := B_r \cup \{\eta\}$;
            $O := O \cup \{(r, \eta)\}$;
```

Algorithm 1: Fixpoint iteration computing the subspaces $H(q)$.

3.1.1 The general algorithm

The starting point of the computation of the $H(q)$ is the initial vector $|\epsilon|_U$. From the base case of the recurrence relations for $|w|_u$, we get the definition

$$|\epsilon|_U = (|\epsilon|_u)_{u \in U} : |\epsilon|_u = \begin{cases} 1 & \text{if } |u| = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The fact that for any transition $q \xrightarrow{a} r$ and any word $w \in L(q)$, the word $w.a$ must be in $L(r)$ is then reflected by requiring that for any vector $\phi \in H(q)$, the vector $F_a\phi$ must be in $H(r)$, where F_a is a matrix such that $F_a|w|_U = |w.a|_U$ for any w. This means that F_a consists of the coefficients of the recurrences for $|w.a|_U$ in terms of $|w|_U$, i.e.

$$F_a = (f_{u,v})_{u,v \in U} : f_{u,v} = \begin{cases} 1 & \text{if } u = v.a.\sigma \text{ or } u = v, a \notin \sigma^u_{|u|}, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for $U = \{\epsilon, a\{a\}, a\{b\}, a\{a\}b, a\{b\}a\}$,

$$|\epsilon|_U = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, |w.a|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} |w|_U, |w.b|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} |w|_U.$$

To compute the invariants, we determine, for all $q \in Q$, a basis of the subsequence hull $H(q) = span(\{|w|_U : w \in L(q)\})$, using the fixpoint iteration shown in Figure 1.

Theorem 3.1 *1. The sets B_q computed by the fixpoint iteration shown in Figure 1 are bases for the vector spaces $H(q)$ spanned by $\{|w|_U : w \in L(q)\}$.*

2. When called for an automaton $P = (Q, \Sigma, Q^0, Q^e, T)$ with $card(T) = m$ and $U \subset \Sigma^*$ with $card(U) = n$, the fixpoint iteration terminates in time $O(mn^3)$.

Proof:

1. Note first that the elements of B_q are linearly independent. This is achieved by the reduction step; if a vector ψ is a linear combination of the elements of B_q found so far, it will be reduced to $\mathbf{0}$ and discarded.

   We can prove inductively that all elements of the B_q are linear combinations of the vectors $\{|w|_U : w \in L(q)\}$:

   - Initially, the only elements of any B_q are $|\epsilon|_U \in B_{q^0}$ for all $q^0 \in Q^0$. This fulfills the condition, since $\epsilon \in L(q^0)$.
   - Let $\psi = F_a\phi$ be generated from a vector $\phi \in B_q$, for a transition $q \xrightarrow{a} r$. By the induction hypothesis, $\phi = \lambda_1|w^1|_U + \cdots + \lambda_k|w^k|_U$ for some $\lambda_i \in \mathbb{R}$ and $w^i \in L(q)$. But then $\psi = F_a\phi = \lambda_1 F_a|w^1|_U + \cdots + \lambda_k F_a|w^k|_U = \lambda_1|w^1.a|_U + \cdots + \lambda_k|w^k.a|_U \in H(r)$, since $w^i.a \in L(r)$. The reduction with the existing elements of B_r, which are also in $H(r)$ by the induction hypothesis, does not change this.

   Assume now that there are $r \in Q$ and $w \in L(r)$ such that $|w|_U$ is not in $span(B_r)$. Obviously, $w \neq \epsilon$, since the only r with $\epsilon \in L(r)$ are the $r \in Q^0$, and for them B_r is initialized such that it contains $|\epsilon|_U$.

   So we have $w = v.a$ for some $v \in \Sigma^*$ and $a \in \Sigma$, and there is a $q \in Q$ satisfying $v \in L(q)$ and $q \xrightarrow{a} r$.

   We can assume without loss of generality that r, w were picked such that w is a *minimal* unrepresented word, in the sense that the vector $|v|_U$ associated to its prefix v is in $span(B_p)$ for all $p \in Q$ with $v \in L(p)$. But then we have that $|v|_U = \lambda_1\phi^1 + \cdots + \lambda_k\phi^k$, where $B_q = \{\phi^1, \ldots, \phi^k\}$, and therefore $|w|_U = F_a|v|_U = \lambda_1 F_a\phi^1 + \cdots + \lambda_k F_a\phi^k$. Each of the vectors $F_a\phi^i$ is generated in the course of the algorithm, and only discarded if it is linearly dependent of the existing elements of B_r; in particular, all $F_a\phi^i$, and therefore also $|w|_U$, are in the space spanned by B_r, a contradiction.

2. Obviously, the time used by the initialization is trivial, and the relevant part of the algorithm is the while loop.

   Observe that each transition $q \xrightarrow{a} r$ is used for computation of a vector $\psi = F_a\phi$ exactly once for every basis element $\phi \in B_q$, i.e., $\dim H(q) \leq n$ times. This gives a total of $O(mn)$ iterations, each with

   - a matrix-vector multiplication; the special form of the F_a allows this to be done in time $O(n)$ - basically, for each $u \in U$, $\psi_u = \phi_u$ or $\psi_u = \phi_u + \phi_v$, where $u = v.a$,

```
Data: Automaton $P = (Q, \Sigma, Q^0, Q^e, T)$, finite prefix-closed $U \subset \Sigma^*$
Result: Bases $B_q$ for the subspaces $H(q) = span(|w|_U : w \in L(q))$
// Initialization:
pick $s \in Q$; $M_s := Id$; $B_s := \emptyset$; $O := \{s\}$;
$C := \emptyset$; $D := \emptyset$;
if $s \in Q^0$ then $D := D \cup \{Id\}$;
while $O \neq \emptyset$ do
    take $q$ from $O$;
    foreach $r \xrightarrow{a} q$ do
        $N := M_q F_a$;
        if $M_r$ not yet defined then
            define $M_r := N$;
            $O := O \cup \{r\}$;
            if $r \in Q^0$ then  $D := D \cup \{M_r\}$ ;        // new prefix $r \to s$
        else if $M_r \neq N$ then
            $C := C \cup \{N M_r^{-1}\}$;                      // new cycle through $r$
foreach $M \in D$ do  $O := O \cup \{M(1, 0, \ldots, 0)^T\}$;
while $O \neq \emptyset$ do
    take $\eta$ from $O$;  begin reduce $\eta$ with $B_s$:
        foreach $\phi \in B_s$ do
            $v := pivot(\phi)$;
            $\eta := \eta - (\eta_v / \phi_v)\phi$;
    end
    if $\eta \neq \mathbf{0}$ then
        foreach $M \in C$ do  $O := O \cup \{M\eta\}$;
        begin reduce $B_s$ with $\eta$:
            $v := pivot(\eta)$;
            foreach $\phi \in B_s$ do  $\phi := \phi - (\phi_v / \eta_v)\eta$;
        end
        $B_s := B_s \cup \{\eta\}$;
foreach $q \in Q \setminus \{s\}$ do
    $B_q := \{M_q^{-1}\phi : \phi \in B_s\}$
```

Algorithm 2: Local fixpoint iteration computing the subspaces $H(q)$.

- a reduction of the vector ψ, using the $\leq n$ vectors already in B_r, with a time complexity of $O(n^2)$.

The reduction is the dominant step here, resulting in the overall complexity of $O(mn^3)$.

□

3.1.2 An optimized algorithm for strongly-connected automata

For a set U of pure subsequences, the recurrence matrices are given by

$$F_a = (f_{u,v})_{u,v\in U} : f_{u,v} = \begin{cases} 1 & \text{if } u \in \{v, v.a\}, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for $U = \{\epsilon, a, b, aa, ab, ba, bb\}$,

$$|w.a|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} |w|_U, \; |w.b|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} |w|_U.$$

These matrices are unit lower triangular matrices (recall that U is ordered by $<_{llex}$) and thus have determinant 1; their inverses are

$$F_a^{-1} = (b_{u,v})_{u,v\in U} : b_{u,v} = \begin{cases} (-1)^k & \text{if } u = v.a^k, k \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

For the above example, this gives us

$$|w|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} |w.a|_U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 1 \end{pmatrix} |w.b|_U.$$

If P is also strongly connected, i.e., there is a path from q to r for all locations $q, r \in Q$, we can improve the construction of the invariants. Since the matrices F_a are invertible, for $w = w_1 \dots w_n$ such that $q \overset{w}{\to} r$, the composition $F_w = F_{w_n} \dots F_{w_1}$ is an isomorphism from $H(q)$ to its image $F_w(H(q)) \subseteq H(r)$, implying in particular $\dim(H(q)) \leq \dim(H(r))$. In the strongly connected case, this implies $\dim(H(q)) = \dim(H(r))$ for all q, r, and $H(r) = F_w(H(q))$, i.e., F_w is an isomorphism from $H(q)$ to $H(r)$ when $q \overset{w}{\to} r$.

The local fixpoint iteration shown in Figure 2 exploits this observation by effectively "pulling back" the computation of the subsequence hulls to a single location. It does so by finding, for some chosen location s,

- isomorphisms $M_q : H(q) \to H(s)$ for all $q \in Q$,
- a set C of automorphisms of $H(s)$ corresponding to a cycle basis of the automaton, and
- a set D containing one isomorphism from $H(q)$ to $H(s)$ for each $q \in Q^0$, corresponding to some prefix leading from q to s.

The construction of the basis of $H(s)$ starts with the vectors $M(1, 0, \dots, 0)^T$, for $M \in D$. The full basis then consists of the closure of this set with respect to the matrices in C. For all other $q \in Q$, $H(q)$ is obtained from $H(s)$ via M_q. The main advantage of this algorithm is the lower number of reduction steps if the cycle degree $card(T) - card(Q) + 1$ of P is small compared to $card(T)$:

Theorem 3.2 *1. The set B_s computed by the local fixpoint iteration shown in Figure 2 forms a basis of $H(s)$.*

2. *When called for an automaton $P = (Q, \Sigma, Q^0, Q^e, T)$ with $card(T) = m$ and cycle degree $\gamma := card(T) - card(Q) + 1$, and $U \subset \Sigma^*$ with $card(U) = n$, the local fixpoint iteration terminates in time $O(mn^2 + \gamma n^3)$.*

Proof:

1. As in the general algorithm, reduction of each candidate vector ψ with the existing elements of B_s keeps B_s linearly independent.

   The proof for $B_s \subset H(s)$ relies on the above observation that in the strongly connected case, F_a is an isomorphism from $H(q)$ to $H(r)$ for all $q \xrightarrow{a} r$. The initial elements $M(1, 0, \ldots, 0)^T = M|\epsilon|_U$ for $M \in D$ are again obviously in $H(s)$, since they correspond directly to words $w \in L(s)$.

   Assume $\psi = C_i\phi$ is obtained by the fixpoint iteration, where $\phi \in H(s)$. By construction, $C_i = M_r F_a M_q^{-1}$ for some $q, r \in Q$ with $r \xrightarrow{a} q$. C_i is a composition of isomorphisms and thus an automorphism of $H(s)$, such that $C_i\phi \in H(s)$.

   Now, let $w = w_1 \ldots w_l \in L(s)$, which implies that there is a path $q_0 \xrightarrow{w_1} q_1 \cdots q_{l-1} \xrightarrow{w_l} q_l = s$, with $q_0 \in Q^0$. For each step $q_{i-1} \xrightarrow{w_i} q_i$, define a matrix $N_i := M_{q_i} F_{w_i} M_{q_{i-1}}^{-1}$. Each of these matrices will equal either the identity (if $q_{i-1} \xrightarrow{w_i} q_i$ was used to define M_{q_i} during the exploration) or an element $C_i \in C$.

   Now,

   $$\begin{aligned} |w|_U &= F_{w_l} \cdots F_{w_1} |\epsilon|_U \\ &= M_{q_l} F_{w_l} M_{q_{l-1}}^{-1} M_{q_{l-1}} F_{w_{l-1}} M_{q_{l-2}}^{-1} \cdots M_{q_1}^{-1} M_{q_1} F_{w_1} M_{q_0}^{-1} |\epsilon|_U \\ &\text{(since } M_{q_l} = M_s = Id \text{ and all the } M_{q_i}^{-1} M_{q_i} \text{ cancel)} \\ &= N_l \cdots N_1 M_{q_0} |\epsilon|_U, \end{aligned}$$

   and it suffices to prove that $\psi_i := N_i \cdots N_1 M_{q_0} |\epsilon|_U \in span(B_{q^0})$ for $i = 0, \ldots, l$. But during the exploration, M_{q_0} gets added to D, and B_s therefore contains $M_{q_0}|\epsilon|_U$, and the fixpoint iteration ensures $C_i\phi \in span(B_{q^0})$ for all $\phi \in span(B_{q^0})$ and $C_i \in C$.

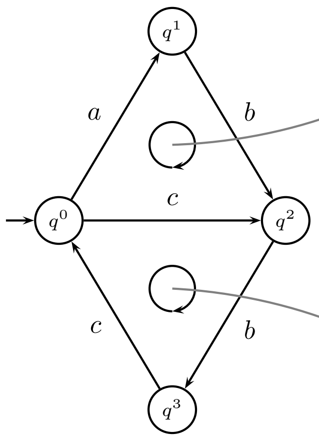
2. The algorithm has two main parts:

   - The computation of the matrices M_q and $C_i \in C$:
     The operations involved are a matrix-matrix multiplication for each $q \xrightarrow{a} r$, which only takes time $O(n^2)$ due to the special form of the F_a, and a matrix inversion and matrix-matrix multiplication (both feasible in time $O(n^3)$[1]) for each cycle.
   - The fixpoint iteration: for each of the $\dim H(s) \leq n$ basis vectors and each of the γ cycle matrices, this requires the computation of a Matrix-vector product and a reduction with the existing elements of B_{q^0}, each feasible in time $O(n^2)$.

   Altogether, this results in the complexity $O(mn^2 + \gamma n^3)$.

□

[1]Algorithms with lower complexity exist, such as the Coppersmith-Winograd algorithm with asymptotic complexity $O(n^{2.376})$. n needs to be rather large for these benefits to manifest, though.



$$C_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$$C_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.1: Example for the local fixpoint construction.

Example: Consider the automaton in Figure 3.1. Using $U = \{\epsilon, a, b, c, aa, ba, ca\}$, we compute B_{q^2} as follows:

1. Initialization: $M_{q^2} = Id, C = \emptyset, D = \emptyset, B = \emptyset$;

2. Exploration:

$$\begin{array}{lcl} q^0 \xrightarrow{c} q^2 & : & M_{q^0} = F_c; \text{add } F_c \text{ to } D; \\ q^1 \xrightarrow{b} q^2 & : & M_{q^1} = F_b; \\ q^0 \xrightarrow{a} q^1 & : & \text{add } C_1 = F_b F_a F_c^{-1} \text{ to } C; \\ q^3 \xrightarrow{c} q^0 & : & M_{q^3} = F_c^2; \\ q^2 \xrightarrow{b} q^3 & : & \text{add } C_2 = F_c^2 F_b \text{ to } C; \end{array}$$

   C_1 and C_2 correspond to the basic undirected cycles $q^2 \xleftarrow{c} q^0 \xrightarrow{a} q^1 \xrightarrow{b} q^2$ and $q^2 \xrightarrow{b} q^3 \xrightarrow{c} q^0 \xrightarrow{c} q^2$.

3. Basis construction: Starting with $\phi^0 = F_c|\epsilon|_U$, and successively extending B by reducing and adding those $C_i\phi^j$ which do not reduce to 0, we obtain basis vectors

$$\begin{array}{lrrrrrrr} \phi^0 & = (1, & 0, & 0, & 1, & 0, & 0, & 0)^T, \\ \phi^1 & = (0, & 1, & 0, & -3, & 0, & 0, & 0)^T, \\ \phi^2 & = (0, & 0, & 1, & 2, & 0, & 0, & 0)^T, \\ \phi^3 & = (0, & 0, & 0, & 0, & 1, & 0, & -3)^T, \\ \phi^4 & = (0, & 0, & 0, & 0, & 0, & 1, & 2)^T, \end{array}$$

4. Local invariant generation: Computing the orthogonal complement, we obtain the following basis for $\Gamma(q^0)$:

$$\begin{array}{lrrrrrrr} \psi^1 & = (1, & -3, & 2, & -1, & 0, & 0, & 0)^T, \\ \psi^2 & = (0, & 0, & 0, & 0, & 3, & -2, & 1)^T, \end{array}$$

   All words $w \in L(q^2)$ thus satisfy the invariants $3|w|_a - 2|w|_b + |w|_c = 1$, and $3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$.

5. Global invariant generation: Collecting the vectors $M_q^{-1}\phi^i$ for all $q \in Q$ and computing the orthogonal complement, we obtain the global invariant $3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$.

3.2 From Process to System Invariants

A key advantage of subsequence invariants is that invariants that have been computed for an individual automaton are immediately inherited by the full system and can therefore by composed by simple conjunction.

Theorem 3.3 *Let S be a system of finite automata communicating over a system alphabet Σ, and let P be one of these automata with local alphabet $\Sigma_P \subseteq \Sigma$. If U is a set of subsequences over Σ_P and $\sum_{u \in U} \phi_u |w|_u = 0$ is a subsequence invariant for P over U, then $\sum_{u \in U} \phi_u |w|_u = 0$ also holds for all $w \in L(S)$.*

Proof: Note that since U is a set of subsequences over Σ_P, any $u \in U$ has the form $\sigma_0 e_1 \sigma_1 \ldots e_n \sigma_n$ with $e_k \in \Sigma_P$, $\sigma_k \subseteq \Sigma_P$ for all k. If $w \in \Sigma^*, a \in \Sigma \setminus \Sigma_P$ and $u \in U$, then obviously $a \neq e_k$ and $a \notin \sigma_k$ for all k, and therefore $|w.a|_u = |w|_u$. Applying this argument inductively, we obtain $|w|_u = |w \downarrow_{\Sigma_P}|_u$ for all $w \in \Sigma^*$. Since $w \downarrow_{\Sigma_P} \in L(P)$ for $w \in L(S)$, this implies $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in L(S)$. $\square$

The system S may satisfy additional invariants, not covered by Theorem 3.3, that refer to interleavings of sequences from Σ_i^* with sequences from a different Σ_j^*. In the following, we present several methods for obtaining such additional invariants.

3.2.1 Invariants obtained by projection

The first approach works similarly to the resolution of conditions in Section 2.2. It uses the fact that given any subsequence invariant for S, we can obtain a new subsequence invariant by appending the same symbol to all involved subsequences:

Theorem 3.4 *Let $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in L(S)$, and $a \in \Sigma$. Then we also have $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ for all $w \in L(S)$.*

Proof: We prove the theorem by induction over w. For $w = \epsilon$, the property holds, because $\epsilon_{u.a} = 0$ for all u, a. Assume that $\sum_{u \in U} \phi_u |w|_{u.a} = 0$ holds for some w, and let $b \in \Sigma$.

If $b \neq a$, then $|w.b|_{u.a} = |w|_{u.a}$ for all u, and the claim follows immediately from the induction hypothesis.

If $b = a$, we have from the recurrence relations $|w.b|_{u.a} = |w|_{u.a} + |w|_u$ for all u, and therefore $\sum_{u \in U} \phi_u |w.b|_{u.a} = \sum_{u \in U} \phi_u |w|_{u.a} + \sum_{u \in U} \phi_u |w|_u$. The sum $\sum_{u \in U} \phi_u |w|_{u.a}$ equals 0 by the induction hypothesis, and $\sum_{u \in U} \phi_u |w|_u$ equals 0 because of the original invariant. $\square$

Example: Consider a system containing the automaton from Figure 3.1. From the invariant

$$3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$$

we obtain the new invariants

$$3|w|_{aad} - 2|w|_{bad} + |w|_{cad} = 0,$$

$$3|w|_{aaad} - 2|w|_{baad} + |w|_{caad} = 0, \text{ and}$$

$$3|w|_{aada} - 2|w|_{bada} + |w|_{cada} = 0$$

by appending d, ad, and da, respectively.

3.2.2 Invariants obtained by algebraic dependencies

The equalities in Theorem 2.3 can be used to derive new invariants from a given set of subsequence invariants:

Let $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in L(S)$, and v be some phased subsequence. Then obviously, $\sum_{u \in U} \phi_u |w|_u |w|_v$ is also zero; Using the equalities $|w|_u |w|_v = \sum_{x \in C(u,v)} |x|_{u,v} |w|_x$, this can be transformed into new linear subsequence invariants $\sum_{u \in U} \sum_{x \in C(u,v)} \phi_u |x|_{u,v} |w|_x = 0$.

Example: Consider a system containing the automaton from Figure 3.1. It contributes the invariant $3|w|_{aa} - 2|w|_{ba} + |w|_{ca} = 0$ for all $w \in L(S)$. For $v = ad$, Theorem 2.3 provides the algebraic dependencies

$$\begin{array}{rcl} |w|_{aa}|w|_{ad} & = & 2|w|_{aad} + |w|_{ada} + 3|w|_{aaad} + 2|w|_{aada} + |w|_{adaa}, \\ |w|_{ba}|w|_{ad} & = & |w|_{bad} + |w|_{abad} + |w|_{abda} + |w|_{adba} + 2|w|_{baad} + |w|_{bada}, \\ |w|_{ca}|w|_{ad} & = & |w|_{cad} + |w|_{acad} + |w|_{acda} + |w|_{adca} + 2|w|_{caad} + |w|_{cada}, \end{array}$$

which can be used to obtain from $(3|w|_{aa} - 2|w|_{ba} + |w|_{ca})|w|_{ad} = 0$ the new subsequence invariant
$6|w|_{aad} + 3|w|_{ada} + 9|w|_{aaad} + 6|w|_{aada} + 3|w|_{adaa} - 2|w|_{bad} - 2|w|_{abad} - 2|w|_{abda} - 2|w|_{adba} - 4|w|_{baad} - 2|w|_{bada} + |w|_{cad} + |w|_{acad} + |w|_{acda} + |w|_{adca} + 2|w|_{caad} + |w|_{cada} = 0$.
Using the invariants from the previous example, the new invariant reduces to
$3|w|_{aad} + 3|w|_{ada} + 3|w|_{aaad} + 3|w|_{aada} + 3|w|_{adaa} - 2|w|_{abad} - 2|w|_{abda} - 2|w|_{adba} + |w|_{acad} + |w|_{acda} + |w|_{adca} = 0$.

3.2.3 Invariants obtained from nonnegativity constraints

One obvious property of subsequence counters $|w|_u$ is that they can only take nonnegative values. This property is not represented in the subsequence invariants as such, but can be used to strengthen existing invariants.

In particular, if the set of system invariants implies an invariant of the form $\phi_1|w|_{u^1} + \cdots + \phi_k|w|_{u^k} = 0$ such that $\phi_i > 0$ for all i, then we can add $|w|_{u^i} = 0$ to the set of invariants, for all i. Such invariants can be found using standard linear programming methods.

For example, the processes in figure 3.2 satisfy the invariants $2|w|_{cc} + |w|_c - |w|_{ac} + |w|_{bc} = 0$ and $2|w|_{cc} + |w|_{ac} - |w|_{bc} = 0$, respectively; adding these gives the system invariant $4|w|_{cc} + |w|_c = 0$, which we can split into $|w|_{cc} = 0$ and $|w|_c = 0$, representing the fact that the c transition can never be taken.

This method can be further improved by taking into account linear *inequalities* on the numbers of occurrences. Obtaining invariants of this form can be

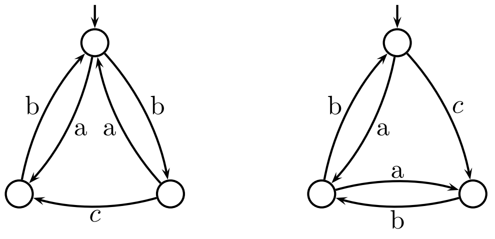


Figure 3.2: Combining invariants from these two processes gives the new invariant $4|w|_{cc} + |w|_c = 0$, which has only nonnegative coefficients and can be split into $|w|_{cc} = 0$ and $|w|_c = 0$.

a challenging problem [23, 20], but for subsequences, there are some inequalities other than $|w|_u \geq 0$ which are always satisfied, such as $|w|_{baaab} + |w|_{bab} - |w|_{baab} \geq 0$[53]. In particular, there are several infinite families of such universally valid inequalities:

Inequalities from nonnegative polynomials. Using Theorem 2.8, we get for any polynomial $p \in \mathbb{R}[x_1, \dots, x_n]$ which is nonnegative for all nonnegative integer arguments, and for any subsequences $v^1, \dots, v^n$, a linear combination $\sum_{u \in U} \phi_u |w|_u$ whose value equals that of $p(|w|_{v^1}, \dots, |w|_{v^n})$, and is therefore nonnegative, for all w.

For example, the polynomial $p(x) = x^2 - 5x + 6$ satisfies $p(n) \geq 0$ for all $n \in \mathbb{N}$. Applying Theorem 2.8 to $p(|w|_{a\{a\}b})$ gives the inequality

$$|w|_{a\{a\}ba\{a\}b} + |w|_{a\{a\}b\{a\}b} - 2|w|_{a\{a\}b} + 3 \geq 0,$$

which holds for all $w \in \Sigma^*$.

Graph-theoretical inequalities. Let $u, w \in \Sigma^*$ with $|u| = n$. In the following, we will assume $|w|_u > 0$, since otherwise the inequality we will derive holds trivially. For any $k \in \{1, \dots, n\}$, we define the *k-equivalence* relation on $[w]_u$ by $i \sim_k j \Leftrightarrow i_l = j_l$ for all $l \neq k$. The *occurrence graph* for u in w is $G = (V, E)$, where $V = [w]_u$ and $E = \{(i, j) \mid i \neq j, i \sim_k j \text{ for some } k\}$ (see figure 3.3).

Let m be the minimal element of $[w]_u$. For any $i \in [w]_u$, we can define occurrences $j^0, \dots, j^n \in [w]_u$ with

$$j^k_l = \begin{cases} i_l & \text{if } l \leq k \\ m_l & \text{otherwise,} \end{cases}$$

such that $j^0 = m, j^n = i$, and for $k = 1, \dots, n$, $j^k \sim_k j^{k-1}$, i.e. either $j^k = j^{k-1}$ or $(j^{k-1}, j^k) \in E$. This means that there is a path from m to i in G for all $i \in V$. Therefore G is connected, and in particular $|E| \geq |V| - 1$.

For $k = 1, \dots, n$, define the k-th *stuttering* subsequence $c^k(u) = a_1 \dots a_k a_k \dots a_n$, i.e. the subsequence obtained by duplicating a_k. Given $i, j \in [w]_u$ such that $i \sim_k j, i < j$, the shuffle tuple $s(i, j) = (s_1, \dots, s_{n+1})$ is given by

$$s_l = \begin{cases} i_l & \text{if } l \leq k \\ j_{l-1} & \text{otherwise.} \end{cases}$$

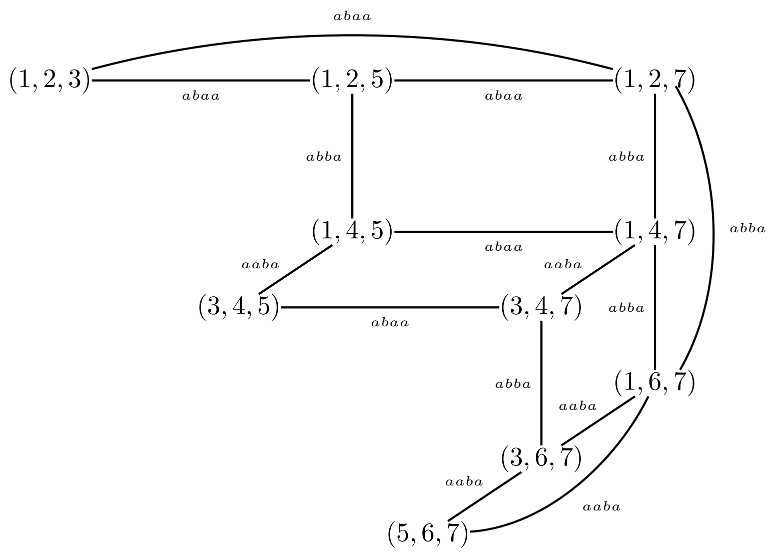


Figure 3.3: The occurrence graph for *aba* in *abababa*. Edges are labeled with the stuttering subsequence corresponding to the shuffle of the incident vertices.

It is easy to check that this is an occurrence of $c^k(u)$, and in fact $(i,j) \mapsto s(i,j)$ is a bijection from $\{(i,j) \mid i \sim_k j, i < j\}$ to $[w]_{c^k(u)}$.

Since from this we get $|E| = |w|_{c^1(u)} + \cdots + |w|_{c^n(u)}$, and $|V| = |w|_u$, we have for every pure subseqence $u \in \Sigma^*$ an inequality

$$\sum_{k=1}^{|u|} |w|_{c^k(u)} - |w|_u + 1 \geq 0$$

which holds for all $w \in \Sigma^*$.

As an example, consider $u = aba$. The stuttering subsequences for u are $aaba, abba$, and $abaa$, and for any $w \in \Sigma^*$, the inequality

$$|w|_{aaba} + |w|_{abba} + |w|_{abaa} - |w|_{aba} + 1 \geq 0$$

is satisfied.

3.2.4 Invariants obtained from nested subsequences

For pure subsequences, transitivity of the order $\preceq$ given by $u \preceq v \Leftrightarrow |v|_u > 0$ implies another obvious way of deriving new invariants: if $u \preceq v$ and $u \not\preceq w$, then $v \not\preceq w$, i.e. we get $|w|_v = 0$ from $|w|_u = 0$.

This can be generalized as follows: We define the order $\preceq$ on *phased* subsequences by $u \preceq v$ if and only if there is an index tuple $(i_0, \ldots, i_{n+1})$, where $n = |u|$, such that

- $0 = i_0 < \cdots < i_{n+1} = |v| + 1$,
- $e^u_p = e^v_{i_p}$ for $1 \leq p \leq n$,

- $\sigma_p^u \subseteq \sigma_q^v$ for $0 \leq p \leq n$ and $i_p \leq q < i_{p+1}$,
- $e_q^v \notin \sigma_p^u$ for $0 \leq p \leq n$ and $i_p < q < i_{p+1}$.

Note that $u \preceq v$ if and only if there is some u' such that u, u' cover v. It is easy to prove that $\preceq$ is indeed a partial order on the set of phased subsequences.

Now if $(j_1, \dots, j_m)$ is an occurrence of v in $w \in \Sigma^*$ and $(i_0, \dots, i_{n+1})$ is a tuple witnessing $u \preceq v$, it is easy to check that $(j_{i_1}, \dots, j_{i_n})$ is an occurrence of u in w, and we again get that for $u \preceq v$, $|w|_u = 0$ implies $|w|_v = 0$.

3.3 Universal Invariants

A subsequence invariant is *universal* iff it holds for all $w \in \Sigma^*$. For pure subsequences, there are no nontrivial universal invariants (note that the equation $|w|_\epsilon = 1$ actually is trivial, since by definition, a subsequence invariant is a homogeneous linear equation, and any constant term c is really just a shorthand for $c|w|_\epsilon$).

Lemma 3.5 *Let $U \subseteq \Sigma^*$ be a set of pure subsequences, and let $\phi \in \mathbb{R}^U$ be such that $\sum_{u \in U} \phi_u |w|_u = 0$ for all $w \in \Sigma^*$. Then $\phi_u = 0$ for all $u \in U$.*

Proof: Assume that ϕ is a nontrivial universal invariant over some set $U \subseteq \Sigma^*$ of pure subsequences. Let u be of minimal length such that $\phi_u \neq 0$. Since for any $v \in \Sigma^*$ such that $|u|_v \neq 0$, we have that either $v = u$ or $|v| < |u|$, there is exactly one $v \in \Sigma^*$ with both $\phi_v \neq 0$ and $|u|_v \neq 0$, namely u itself. But then

$$\sum_{v \in U} \phi_v |u|_v = \phi_u |u|_u = \phi_u \neq 0,$$

which contradicts the assumption.

□

For phased subsequences, on the other hand, this changes. In the following, we present some classes of universal phased subsequence invariants.

3.3.1 Prefix splits

Consider the phased subsequence $\{a\}a$. An occurrence of u in w is essentially just an index i such that $w_i = a$, and there is no $j < i$ with $w_j = a$. This subsequence matches the first occurrence of a, if there is one, and nothing otherwise, so that we have

$$|w|_{\{a\}a} = \begin{cases} 1 & |w|_a > 0 \\ 0 & |w|_a = 0. \end{cases}$$

On the other hand, $|w|_{\{a\}} = 1$ if $|w|_a = 0$, and 0 otherwise, giving the invariant

$$|w|_{\{a\}a} + |w|_{\{a\}} = 1 \text{ for all } w \in \Sigma^*.$$

This result can be generalized:Let $u = a_1 \dots a_n \in \Sigma^*$ be a pure subsequence. The *prefix split* for u is the set of phased subsequences $P_u = \{p^0(u), \dots, p^n(u)\}$, where $p^k(u) = \{a_1\}a_1 \dots \{a_k\}a_k\{a_{k+1}\}$ for $k < n$, and $p^n(u) = \{a_1\}a_1 \dots \{a_n\}a_n$. Then $|w|_{p^k(u)}$ is 1 if the longest prefix of u which occurs as a subsequence in w has length k, and 0 otherwise:

Lemma 3.6 *Let $u = a_1 \dots a_n \in \Sigma^*$.*

1. *For $k < n$,* $|w|_{p^k(u)} = \begin{cases} 1 & \textit{if } |w|_{a_1 \dots a_k} > 0 \textit{ and } |w|_{a_1 \dots a_{k+1}} = 0 \\ 0 & \textit{otherwise,} \end{cases}$

2. $|w|_{p^n(u)} = \begin{cases} 1 & \textit{if } |w|_u > 0 \\ 0 & \textit{if } |w|_u = 0. \end{cases}$

Proof:

1. Let $k < n$, and $w \in \Sigma^*$. Any occurrence $(i_1, \dots, i_k)$ of $p^k(u)$ in w is obviously also an occurrence of $a_1 \dots a_k$, so that $|w|_{p^k(u)} = 0$ if $|w|_{a_1 \dots a_k} = 0$. Assume now that $|w|_{a_1 \dots a_k} > 0$, and let $(j_1, \dots, j_k) \in [w]_{a_1 \dots a_k}$. If $(i_1, \dots, i_k) \in [w]_{p^k(u)}$, then $w_l \neq a_1$ for all $l < i_1$, so that $j_1 \geq i_1$. Since also $w_l \neq a_m$ for $i_m < l < i_{m+1}, m = 1 \dots, k-1$, we get by induction that $j_l \geq i_l$ for all l, i.e. i is minimal in $[w]_{a_1 \dots a_k}$. In particular, this implies $|w|_{p^k(u)} \leq 1$.
   Let now $i = (i_1, \dots, i_k)$ be the minimal element of $[w]_{a_1 \dots a_k}$. Then $|w|_{a_1 \dots a_{k+1}} = 0$ if and only if there is no $l > i_k$ with $w_l = a_{k+1}$,which is equivalent to i being an occurrence of $p^k(u)$.

2. Analogously to the first part, we get that $i = (i_1, \dots, i_n)$ is an occurrence of $p^n(u)$ if and only if it is the minimal occurrence of u. $[w]_{p^n(u)}$ is therefore empty if $[w]_u$ is, and a singleton set otherwise.

□

It immediately follows from this lemma that

$$\sum_{i=0}^{n} |w|_{p^i(u)} = 1,$$

so that for example

$$|w|_{\{a\}} + |w|_{\{a\}a\{b\}} + |w|_{\{a\}a\{b\}b\{a\}} + |w|_{\{a\}a\{b\}b\{a\}a} = 1$$

for all $w \in \Sigma^*$.

One intuitively appealing way of getting further universal invariants from P_u is to use the fact that for any subset $U \subseteq P_u$, $\sum_{v \in U} |w|_v$ is in $\{0, 1\}$, using the standard approach for resolving disjunctions. It turns out, however, that the results are trivial:

Lemma 3.7 *Consider $p^k(u), p^l(u) \in P_u$ for some pure subsequence $u = a_1 \dots a_n \in \Sigma^*$ and $k, l \leq n$. Then the set $C(p^k(u), p^l(u))$ of subsequences coverable by $p^k(u), p^l(u)$ satisfies*

$$C(p^k(u), p^l(u)) = \begin{cases} \{p^k(u)\} & \textit{if } k = l \\ \emptyset & \textit{otherwise,} \end{cases}$$

and therefore for any x

$$[x]_{p^k(u), p^l(u)} = \begin{cases} 1 & \textit{if } x = p^k(u) = p^l(u) \\ 0 & \textit{otherwise.} \end{cases}$$

Proof: Let $((i_1,\ldots,i_k),(j_1,\ldots,j_l))$ be some covering of $x \in C(p^k(u),p^l(u))$. Let us assume that there is some minimal $m \leq k,l$ such that $i_m \neq j_m$. Without loss of generality, we can take i_m to be less that j_m. Then $e^x_{i_m} = a_m \in \sigma^{p^l(u)}_m$ and $j_{m-1} < i_m < j_m$, contradicting the conditions for j to be an occurrence of $p^l(u)$.
We therefore must have $i_m = j_m$ for $m \leq k,l$. Assume now that $k \neq l$; without loss of generality, $k < l$. But then $e^x_{j_{k+1}} = a_{k+1} \in \sigma^{p^k(u)}_{k+1}$, and i cannot be an occurrence of $p^k(u)$.
It follows that $k = l$ and $i = j = (1,\ldots,k)$. From this, we also get $p^k(u)$.

$\square$

Therefore, the linear invariant one obtains from $\sum_{v\in U}|w|_v \in \{0,1\}$ using Theorem 2.8 is trivial: For any $U \subseteq P_u$, the linearization of $(\sum_{v\in U}|w|_v)^2$ is

$$\sum_{v^1,v^2\in U}\ \sum_{x\in C(v^1,v^2)} |x|_{v^1,v^2}|w|_x = \sum_{v\in U}|w|_v,$$

And $(\sum_{v\in U}|w|_v)(\sum_{v\in U}|w|_v - 1) = 0$ reduces to the trivial invariant $0 = 0$.

Symmetrically to P_u, one can define the *suffix split* $S_u = \{s^0(u),\ldots,s^n(u)\}$ for u, where $s^k(u) = \{a_{n-(k+1)}\}a_{n-1}\{a_{n-1}\}\ldots a_n\{a_n\}$ for $k < n$, and $s^n(u) = a_1\{a_1\}\ldots a_n\{a_n\}$. The above lemmata hold mutatis mutandis. There is no straightforward relation between the occurrences of $p^i(u)$ and $s^i(u)$, except for $i = n$, for which we get

$$|w|_{p^n(u)} = |w|_{s^n(u)} \text{ for all } w \in \Sigma^*.$$

3.3.2 Spanning tree invariants

Another way of obtaining universal invariants is based on phased subsequences capturing spanning trees in a generalization of the occurrence graph.

Let $u = a_1\ldots a_n \in \Sigma^*$, and for $k = 1,\ldots,n$, define the k-th *stuttering* subsequence $c^k(u) = \sigma_0 e_1 \sigma_1 \ldots e_{n+1}\sigma_{n+1}$ by

- $e_i = \begin{cases} a_i & \text{if } i \leq k \\ a_{i-1} & \text{otherwise,} \end{cases}$
- $\sigma_i = \begin{cases} \{a_i\} & \text{if } k \leq i \leq n \\ \emptyset & \text{otherwise.} \end{cases}$

For example, if $u = abc$, then

$$c^1(u) = a\{a\}a\{b\}b\{c\}c, \qquad c^2(u) = ab\{b\}b\{c\}c, \qquad c^3(u) = abc\{c\}c.$$

We are going to show that

$$|w|_u - \sum_{k=1}^{n}|w|_{c^k(u)} = \begin{cases} 1 & \text{if } |w|_u > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Let us call an occurrence $i \in [w]_u$ *k-minimal* if it is minimal in its $\sim_k$-class. For any $i \neq \min([w]_u)$, there is at least one k such that i is *not* k-minimal. We call the maximal such k the *rank* $rk(i)$ of i. We also define, for any $i \neq \min([w]_u)$, the *parent* of i to be $\pi(i) = max\{j \in [w]_u \mid j \sim_{rk(i)} i, j < i\}$. Note that, since for all i except $\min([w]_u)$, $\pi(i)$ is defined and satisfies $\pi(i) < i$, it is in fact the parent function of a tree, with $[w]_u$ as its set of nodes and $\min([w]_u)$ as its root.

$$\begin{tikzcd}
(1,2,3) & & (1,2,5) \arrow[ll, "aba\{a\}a"'] & & (1,2,7) \arrow[ll, "aba\{a\}a"'] \\
& & (1,4,5) \arrow[u, "ab\{b\}b\{a\}a"] & & (1,4,7) \arrow[ll, "aba\{a\}a"'] \\
& (3,4,5) \arrow[ur, "a\{a\}a\{b\}b\{a\}a"] & & (3,4,7) \arrow[ll, "aba\{a\}a"'] & \\
& & & & (1,6,7) \arrow[uu, "ab\{b\}b\{a\}a"'] \\
& & & (3,6,7) \arrow[uu, "ab\{b\}b\{a\}a"] & \\
& & (5,6,7) \arrow[ur, "a\{a\}a\{b\}b\{a\}a"] & &
\end{tikzcd}$$

Figure 3.4: The tree of occurrences of *aba* in *abababa*, edges labeled with the corresponding stuttering subsequences.

Example. Figure 3.4 shows the tree of occurrences of $u = aba$ in $w = abababa$. Of the elements of $[w]_u$,

- $(1,2,5), (1,2,7), (1,4,7), (3,4,7)$ are not 3-minimal, and therefore have rank 3,
- $(1,4,5), (1,4,7), (1,6,7), (3,6,7)$ are not 2-minimal, and therefore $(1,4,5), (1,6,7), (3,6,7)$ have rank 2,
- $(3,4,5), (3,4,7), (3,6,7), (5,6,7)$ are not 1-minimal, and therefore $(3,4,5), (5,6,7)$ have rank 1.

This results in the parent function indicated by the arrows.

Lemma 3.8 *1. For any $k \in \{1, \dots, n\}$, there is a bijection between the set of edges $\{(i, \pi(i)) \mid rk(i) = k\}$ and $[w]_{c^k(u)}$.*

2. $|w|_u - \sum_{k=1}^{n} |w|_{c^n(u)} = \begin{cases} 1 & \textit{if } |w|_u > 0 \\ 0 & \textit{if } |w|_u = 0. \end{cases}$

3. $|w|_u - \sum_{k=1}^{n} |w|_{c^n(u)} = |w|_{p^n(u)} = |w|_{s^n(u)}$.

Proof:

1. For any pair of tuples $i, j \in [w]_u$ with $i \sim_k j, i > j$ we can construct the shuffle $s(i,j) = (s_1, \dots, s_{n+1})$ such that $s_l = j_l$ for $l \leq k$ and $s_l = i_{l-1}$ for

$l > k$. Then

$$\begin{aligned}
&i, j \in [w]_u, rk(i) = k \text{ and } j = \pi(i) \\
\Leftrightarrow\ & 1 \le i_1 = j_1 < \ldots < i_{k-1} = j_{k-1} < j_k < i_k < i_{k+1} = j_{k+1} < \ldots < i_n = j_n \le |w|, \\
& w_{i_l} = w_{j_l} = a_l \text{ for all } l, \\
& w_p \neq a_k \text{ for } j_k < p < i_k, \\
& w_p \neq a_l \text{ for } i_l < p < i_{l+1}, l \ge k \\
\Leftrightarrow\ & s(i,j) \in [w]_{c^k(u)}.
\end{aligned}$$

Since (for any given k) i, j can be reconstructed from $s(i,j)$, this is a bijection.

2. From the first item, we get that $\sum_{k=1}^{n} |w|_{c^n(u)}$ is the number of edges in the tree $T = ([w]_u, \{(i, \pi(i)) \mid i \neq \min([w]_u)\})$, and therefore equals 0 if $|w|_u = 0$, and $|w|_u - 1$ otherwise.

3. Immediately from the previous item and Lemma 3.6.

□

Example. Using $u = ab$, we get that for all $w \in \Sigma^*$,

$$|w|_{ab} - |w|_{ab\{b\}b} - |w|_{a\{a\}a\{b\}b} - |w|_{\{a\}a\{b\}b} = 0$$

and

$$|w|_{ab} - |w|_{ab\{b\}b} - |w|_{a\{a\}a\{b\}b} - |w|_{a\{a\}b\{b\}} = 0.$$

3.4 Incremental Invariant Generation

For the invariant generation algorithm of Section 3.1, we considered the set U of subsequences as given and fixed. In practice, however, the set of subsequences depends on the complexity of the interaction between the processes, and is therefore not necessarily known in advance. In this section, we present an incremental method that allows for growing sets of subsequences.

Let $P = (Q_P, \Sigma_P, q_P^0, T_P)$ be an automaton and $U \subset \Sigma^*$ be finite and prefix-closed. Let $V = U \uplus \{v\}$ again be prefix-closed, i.e. $v = u.a$ for some $u \in U, a \in \Sigma$.

Theorem 3.9 *Assume that for $q \in Q_P$ and the set of subsequences U, a basis of the space $H_U(q) = span(|w|_U : w \in L(q))$ has already been computed, consisting of the vectors $\phi^1, \ldots, \phi^k$. Then either*

1. *$H_V(q)$ is spanned by vectors $\psi^1, \ldots, \psi^k$ such that $\psi_u^j = \phi_u^j$ for all $u \in U$, or*

2. *$H_V(q)$ is spanned by the vectors $\psi^1, \ldots, \psi^k, \eta$ given by:*
   - *$\psi_u^j = \phi_u^j$ for all $u \in U$, and $\psi_v^j = 0$;*
   - *$\eta_u = 0$ for all $u \in U$, and $\eta_v = 1$.*

Proof: For any $w \in \Sigma^*$, the orthogonal projection of $|w|_V = (|w|_u)_{u\in V}$ onto $\mathbb{R}^U$ is just $(|w|_u)_{u\in U} = |w|_U$; since any $\psi \in H_V(q)$ is a linear combination of the form $\psi = \mu_1|w^{(1)}|_V + \cdots + \mu_m|w^{(m)}|_V$, this means that the projection of ψ onto $\mathbb{R}^U$ also lies in $H_U(q)$.

An immediate consequence of this is that the elements of $H_V(q)$ satisfy all local invariants over U that hold for $H_U(q)$, and its dimension is either $\dim H_U(q)$ or $\dim H_U(q) + 1$.

If $\dim H_V(q) = \dim H_U(q)$, then for each vector $\phi^{(j)} = \sum_{i=1}^m \lambda_i|w^{(i)}|_U$ of the existing basis for $H_U(q)$, define the corresponding vector $\psi^{(j)}$ to be $\sum_{i=1}^m \lambda_i|w^{(i)}|_V$; then for all $u \in U$, $\psi_u = \phi_u$. These form a basis of $H_V(q)$ and satisfy the conditions in (1).

Note that the vector η given in (2) cannot be in $H_V(q)$ in this case: If it were, it would have to be a linear combination of the $\psi^{(j)}$ in which at least one coefficient is nonzero; but then its projection onto $\mathbb{R}^U$, which is $\mathbf{0}$, would be a linear combination of the $\phi^{(j)}$ with the same coefficients, which is impossible since the $\phi^{(j)}$ are linearly independent.

If $\dim H_V(q) = \dim H_U(q) + 1$, then $H_V(q)$ must be the space of all vectors in $\mathbb{R}^V$ satisfying the invariants in $I_{q,U}$. The vectors given in (2) form a linearly independent $(k+1)$-tuple of such vectors, and thus a basis of $H_V(q)$. □

All invariants obtained for U remain valid; in the first case, we additionally obtain a new invariant $|w|_v - \sum_{i=1}^k(\psi_v^i/\psi_{u^i}^i)|w|_{u^i} = 0$, where $u^i = pivot(\psi^i)$ for all i, while in the second case, the set of invariants is unchanged.

Example: Consider again the automaton in Figure 3.1. Starting with the smaller set of subsequences $U = \{\epsilon, a, b, c\}$, we obtain the basis $\{(1,0,0,0)^T, (0,1,0,-3)^T, (0,0,1,2)^T\}$ for $H_U(q^0)$, along with the single local invariant $3|w|_a - 2|w|_b + |w|_c = 0$ for q^0. When U is extended to $V = U \cup \{aa, ab\}$ by first adding aa and then ab, case (2) of Theorem 3.9 holds each time. $H_V(q^0)$ has the basis $\{(1,0,0,0,0,0)^T, (0,1,0,-3,0,0)^T, (0,0,1,2,0,0)^T, (0,0,0,0,1,0)^T, (0,0,0,0,0,1)^T\}$. Extending V to $W = V \cup \{ac\}$, case (1) holds: $H_W(q^0)$ has the basis $\{(1,0,0,0,0,0,0)^T, (0,1,0,-3,0,0,-3)^T, (0,0,1,2,0,0,0)^T, (0,0,0,0,1,0,-3)^T, (0,0,0,0,0,1,2)^T\}$, and we obtain a new invariant, $3|w|_a + 3|w|_{aa} - 2|w|_{ab} + |w|_{ac} = 0$.

We compute $H_V(q)$ incrementally from $H_U(q)$ as follows:

- for each basis vector ϕ, except for the initial unit vector $|\epsilon|_U$, we remember by which multiplication $F_a\psi$ it was obtained and how it was reduced; these steps are repeated for the new index v.
- we also remember which successors $F_a\psi$ are reduced to zero; when extending U by $v = u.a$, where $u \in U$, we check for all such ψ whether the reductions result in a nonzero vector, indicating that case (2) of Theorem 3.9 holds.

If case (2) holds for some location q, then the new basis vector η of $H(q)$ is invariant under all $F_{a,V}$, because, by choice, v cannot be a prefix of another sequence in V. Therefore, η is also contained in the subspace $H(r)$ for all locations r reachable from q. The check for case (2) therefore only needs to be performed in one location of each strongly connected component.

Chapter 4

Abstraction-Based Verification

4.1 Introduction

We now extend our approach to infinite-state systems. We do this by integrating the fixpoint iteration algorithm we presented in the previous chapter with *abstraction refinement.* This integrated method will work with two data structures:

- a finite-state abstraction of the infinite-state system, which is gradually refined to eliminate spurious errors, and
- a forest, with vertices labeled by subsequence images, which *covers* the abstraction. This forest represents the state of the fixpoint iteration on the abstraction. It is occasionally pruned when the abstraction is refined.

We split the description of this method into two parts: In this chapter, we first introduce the infinite-state systems with which we work, as well as the abstractions we use, and the refinement and slicing operations. We will use simple state-based error conditions as a source of counterexamples to drive the refinement loop.

The topic of the next chapter is then the integration of subsequence invariants. In particular, we introduce subsequence forests and discuss their behaviour in the course of the refinement loop.

Transition systems. We use a general representation of concurrent systems as transition systems, which can be defined using an assertion language based on first-order logic. We use the following underlying concepts, for any set $\mathcal{V}$ of system variables and alphabet Σ of events:

- For each $v \in \mathcal{V}$ we define a primed variable $v' \in \mathcal{V}'$, which indicates the value of v in the next state.
- We call the set $Asrt(\mathcal{V})$ of first-order formulas over the system variables the *state predicates* and the set $Asrt(\mathcal{V} \cup \mathcal{V}')$ of assertions over the system variables and the primed variables the *transition relations.* For a state

predicate φ, let φ' denote the assertion where each variable v is replaced by v'.

- $\mathbb{T}(\mathcal{V}, \Sigma)$ is the set of *labeled transitions* $\tau = (w_\tau, \rho_\tau)$ over $\mathcal{V}$ and Σ. For each τ,
    - $w_\tau \in \Sigma^*$ is the *label* of τ. We call a transition τ *atomic* if $w_\tau \in \Sigma$. We assume that the system specification contains only atomic transitions, but applying the *Bypass Transition* rule to the abstraction may produce non-atomic transitions during the refinement.
    - $\rho_\tau \in \mathit{Asrt}(\mathcal{V} \cup \mathcal{V}')$ is the *transition relation* of τ. It is in turn a conjunction $\rho_\tau(\mathcal{V}, \mathcal{V}') = \bigwedge_i g_i(\mathcal{V}) \ \wedge \ \bigwedge_i t_i(\mathcal{V}, \mathcal{V}')$ of *guards* g_i and *updates* t_i. In the special case where, for a given set W of variables, ρ_τ is of the form $\rho_\tau(\mathcal{V}, \mathcal{V}') = \bigwedge_i g_i(\mathcal{V}) \ \wedge \ \bigwedge_{v \in W}(v' = e_v(\mathcal{V}))$, i.e., each variable in W is assigned a value defined over $\mathcal{V}$, we say that τ is a *guarded W-assignment.*

A *transition system* $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ consists of the following components:

- $\mathcal{V}$: a finite set of system *variables.*
- $\mathit{init}(\mathcal{V}) \in \mathit{Asrt}(\mathcal{V})$: the *initial condition*, a state predicate characterizing all states in which the computation of the system can start.
- $\mathcal{T} \subset \mathbb{T}(\mathcal{V}, \Sigma)$: a finite set of system *transitions.*

In addition to $\mathcal{S}$, in this chapter we assume given $\mathit{error}(\mathcal{V}) \in \mathit{Asrt}(\mathcal{V})$, a state predicate characterizing all error states.

A *state* of $\mathcal{S}$ is a valuation of the system variables $\mathcal{V}$. A *run* is an alternating sequence $s_0, \tau_0, s_1, \tau_1, \ldots \tau_{n-1}, s_n$ of states and transitions such that $\mathit{init}(s_0)$ holds and for all positions $0 \leq i < n$, $\rho_{\tau_i}(s_i, s_{i+1})$ holds.

In addition to $\mathcal{S}$, we assume given a specification for a set of *errors*, which is a subset of the runs of $\mathcal{S}$. In this chapter, this condition will be given by $\mathit{error}(\mathcal{V}) \in \mathit{Asrt}(\mathcal{V})$, a state predicate characterizing a set of error states. A run $s_0, \tau_0, s_1, \ldots, \tau_{n-1}, s_n$ is an error if there is a position $0 \leq i \leq n$ such that $\mathit{error}(s_i)$ holds. In chapter 5, a run will be an error if the sequence formed by the transition labels violates the subsequence invariants. We say $\mathcal{S}$ is *correct* if it has no errors.

Predicate abstraction. The abstractions that are commonly used for abstraction refinement are based on sets of predicates describing a finite partition of the concrete state space:

A *predicate abstraction* $\mathcal{A} = (Q, Q^0, T)$ of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ with respect to a finite set $\Phi \subseteq \mathit{Asrt}(\mathcal{V})$ consists of the following components:

- An *abstract state space* $Q = 2^\Phi$. Each abstract state $a \subseteq \Phi$ represents the set $\gamma(a)$ of all concrete states satisfying exactly those predicates from Φ which are in a, i.e. $\gamma(a) = \{s \mid s \models \psi_a\}$, where $\psi_a = \bigwedge_{\varphi \in a} \varphi \wedge \bigwedge_{\varphi \in \Phi \setminus a} \neg\varphi$.
- A set $Q^0 \subseteq Q$ of abstract initial states: It contains exactly those abstract states for which the corresponding set of concrete states contains an initial state. In other words, $Q^0 = \{q \in Q \mid \psi_q \wedge \mathit{init} \text{ satisfiable}\}$.

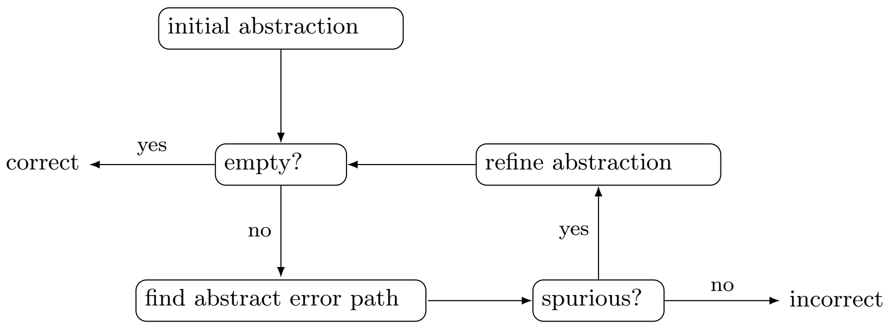


Figure 4.1: Abstraction refinement loop.

- A set $T \subseteq 2^{Q \times Q}$ containing for each transition relation $\tau \in \mathcal{T}$ an *abstract* transition relation $\tau_{\mathcal{A}}$, such that $w_{\tau_{\mathcal{A}}} = w_{\tau}$, and $\rho_{\tau_{\mathcal{A}}}(a_1, a_2)$ if and only if there are $s_1 \in \gamma(a_1), s_2 \in \gamma(a_2)$ with $\rho_{\tau}(s_1, s_2)$.

For the error condition *error*, we get a corresponding set $Q^e = \{q \in Q \mid \psi_q \wedge$ *error* satisfiable$\}$ of abstract error states, which are exactly those abstract states containing an error state.

Since the initial and error states and the transition relation are overapproximated in the abstraction, it can be proved by a simple induction that the same is true for the set of errors. In particular, if the abstraction is correct, then so is the original system. In order to find such an abstraction, one can use abstraction refinement.

Abstraction Refinement. The standard abstraction refinement algorithm follows the loop in figure 4.1: Starting from a suitable initial abstraction, one checks for the existence of an error. If none exists, the system is correct; otherwise, one finds a counterexample which either corresponds to an actual error in the system, or is an artifact of the abstraction. In the latter case, one uses the way in which the counterexample fails to be reproducible in the system to extract a new predicate to add to Φ.

The way in which the new predicates are obtained is an important choice in a refinement-based approach. A very general and successful method is *Craig interpolation.* For a given pair of formulas $\varphi(X)$ and $\psi(Y)$, such that $\varphi \wedge \psi$ is unsatisfiable, a Craig interpolant $\eta(X \cap Y)$ is a formula over the variables common to φ and ψ such that η is implied by φ and $\neg\eta$ is implied by ψ. In particular, one can split a spurious counterexample at some intermediate point and then obtain an interpolant η for the formulas corresponding to the two halves, such that

- any concrete run corresponding to the first half of the counterexample ends in a state satisfying η, and
- any concrete run corresponding to the second half of the counterexample starts in a state satisfying $\neg\eta$.

Adding η to Φ therefore results in an abstraction in which the counterexample no longer exists, see figure 4.2.

$$\begin{tikzcd}
{} \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[r] & a_5 \\
{} \arrow[r] \arrow[dr] & a_1,\eta \arrow[r] \arrow[dr] & a_2,\eta \arrow[r] & a_3,\eta & a_4,\eta \arrow[r] \arrow[dr] & a_5,\eta \\
& a_1,\neg\eta \arrow[r] \arrow[ur] & a_2,\neg\eta \arrow[ur] & a_3,\neg\eta \arrow[r] \arrow[ur] & a_4,\neg\eta \arrow[r] \arrow[ur] & a_5,\neg\eta
\end{tikzcd}$$

Figure 4.2: Refinement with the Craig interpolant η eliminates a spurious counterexample.

Craig interpolants can be automatically generated for a number of theories, including systems of linear inequalities over the reals combined with uninterpreted function symbols [55].

The initial abstraction is usually chosen in such a way that it is exact with respect to the property and the enabledness conditions of the transitions in $\mathcal{T}$. This means that Φ initially contains all predicates that occur in *init*, *error*, and the guards g_i of all $\tau \in \mathcal{T}$.

The advantage of predicate abstraction is its *precision*: when successful, the refinement loop automatically produces a set of predicates that eliminates all spurious counterexamples. On the other hand, the abstract systems generated by predicate abstraction tend to become prohibitively large: the size of the abstract system, and hence the complexity of the verification step of the loop, grows exponentially with the number of predicates.

Additionally, in those cases where the systems consists of a large number of processes, the initial abstraction can already be very large, since Φ then already contains the variables specifying the control state of each process. The abstract state space therefore spans the product of the control skeletons, which is exponentially large in the number of components.

Slicing abstractions. Abstraction and slicing are both techniques for reducing the size of the state space to be inspected during verification. We are going to present a model checking procedure for infinite-state concurrent systems that interleaves automatic abstraction refinement, which splits abstract states according to new predicates obtained by Craig interpolation, with slicing, which removes irrelevant states and transitions from the abstraction. The effects of abstraction and slicing complement each other. As the refinement progresses, the increasing accuracy of the abstract model allows for a more precise slice; the resulting smaller representation gives room for additional predicates in the abstraction. The procedure terminates when an error path in the abstraction can be concretized, which proves that the system is erroneous, or when the slice becomes empty, which proves that the system is correct.

We maintain an explicit graph representation of the abstract model: each node represents a set of concrete states, identified by a set of predicates; each edge represents a set of concrete transitions, identified by their transition rela-

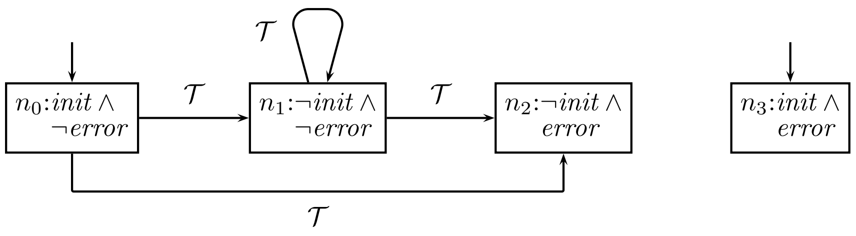


Figure 4.3: Initial abstraction, based on the predicates *init*, characterizing the initial states, and *error*, characterizing the error states.

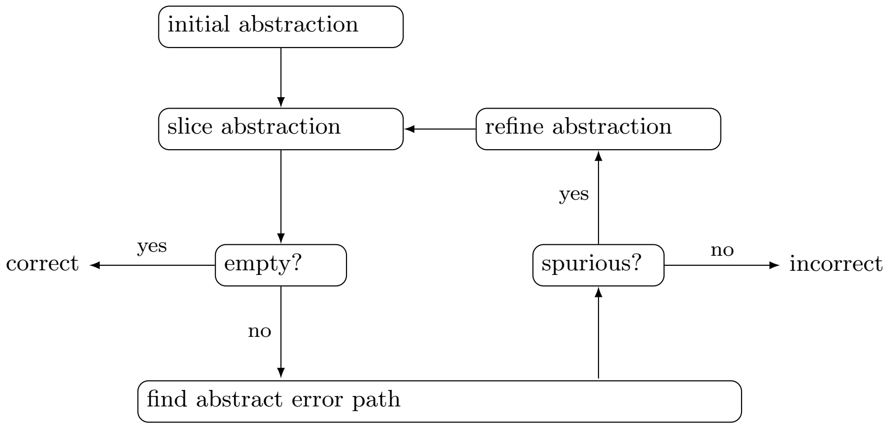


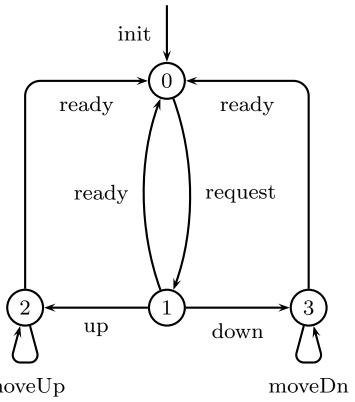
Figure 4.4: Abstraction refinement loop with interleaved refinement and slicing steps.

tions. Since we are interested in small abstract models, we do not insist on the abstract system representing all possible concrete behaviors. Instead, we call an abstraction *sound* if the abstraction has some concretizable error path whenever the system is incorrect.

Consider, for example, the initial abstraction for a state-based error condition. The abstraction initially consists of four nodes over the predicates *init* (characterizing the initial states) and *error* (characterizing error states). The set of edges contains all pairs $(m, n) \in N \times N$, and every edge is labelled with the full set of transitions.

One of the simplification transformations, presented in section 4.5.3, that can be applied during the refinement loop, eliminates incoming edges into nodes labeled by *init* and outgoing edges from nodes labeled with *error*: for any error path that traverses such an edge, there is a shorter error path that visits exactly one node labeled with *init* and one node labeled with *error*. The corresponding transformation for subsequence invariants is given in section 5.6.3. This results in the reduced initial abstraction which is shown in Figure 4.3.

Figure 4.4 shows the abstraction refinement loop. Starting with the initial abstraction, the abstract model is transformed by refinement and slicing steps until the model either becomes empty (which indicates that the system is correct) or a concretizable error path is found (which indicates that the system is incorrect). *Refinement steps* increase the precision of the abstraction by in-



init	$pc{=}0 \wedge cur{\leq}Max \wedge in{\leq}Max$
request	$pc{=}0 \wedge pc'{=}1 \wedge cur'{=}cur \wedge req'{=}in$
ready	$pc{\geq}1 \wedge req{=}cur \wedge pc'{=}0 \wedge cur'{=}cur \wedge req'{=}req \wedge in'{\leq}Max$
up	$pc{=}1 \wedge req{>}cur \wedge pc'{=}2 \wedge cur'{=}cur \wedge req'{=}req$
down	$pc{=}1 \wedge req{<}cur \wedge pc'{=}3 \wedge cur'{=}cur \wedge req'{=}req$
moveUp	$pc{=}2 \wedge req{>}cur \wedge pc'{=}2 \wedge cur'{=}cur+1 \wedge req'{=}req$
moveDn	$pc{=}3 \wedge req{<}cur \wedge pc'{=}3 \wedge cur'{=}cur-1 \wedge req'{=}req$
error	$cur{>}Max$

Figure 4.5: Simple elevator example. The table shows the initial condition *init*, the error condition *error*, and the transition relations.

troducing a new predicate, which is obtained by Craig interpolation from the unsatisfiable formula corresponding to some spurious error path. To minimize the increase in the size of the graph, the new predicate is not applied to the entire graph, but only to a specific node on the error path. This node is split into two copies, such that the label of one copy now additionally contains the predicate and the label of the other copy now additionally contains its negation. The *slicing steps* consist of a collection of reduction rules that maintain soundness in the sense described above. *Elimination rules* drop nodes and edges from the abstraction if they have become unreachable or if their label has become unsatisfiable. *Simplification rules* remove constraints from transition relations that have become irrelevant and simplify the graph structure of the abstraction. Unlike the elimination rules, applicability of simplification rules depends strongly on the kind of error condition, since the irrelevance criteria are quite sensitive to it.

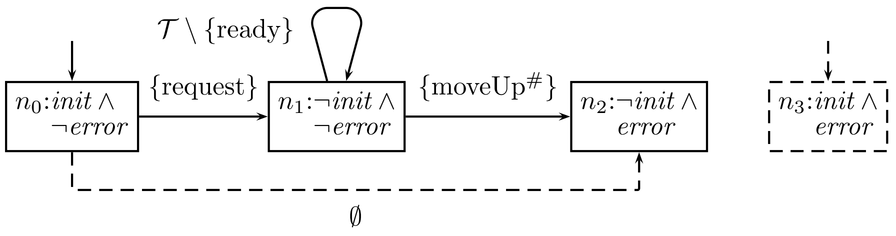
In Section 4.2, we informally explain the various refinement and slicing steps using a motivating example. Formal definitions and proofs for the abstraction mechanism and the refinement and slicing steps follow in Sections 4.3 to 4.5.5.

4.2 A Motivating Example

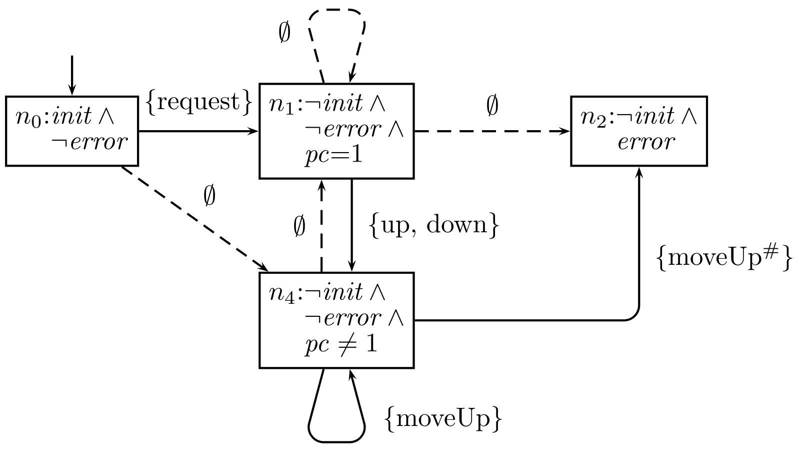
We motivate our approach with a simple elevator example. As shown in Figure 4.5, the elevator accepts a request for a certain floor and then moves up or down accordingly. Once the requested floor is reached, the elevator is ready for a new request. The system variables include the program counter *pc*, the current floor *cur*, the requested floor *req*, and a nondeterministic input variable *in* (*in* is constrained to be in the valid range $in{\leq}Max$ when the elevator is ready to receive its next request). We verify the correctness of the elevator by showing that the error condition $cur > Max$ is never satisfied.

Step 1. The verification starts with the initial abstraction as shown in Figure 4.3. As discussed in the introduction, there are no incoming edges into nodes labeled with *init*, and no outgoing edges from nodes labeled with *error*.

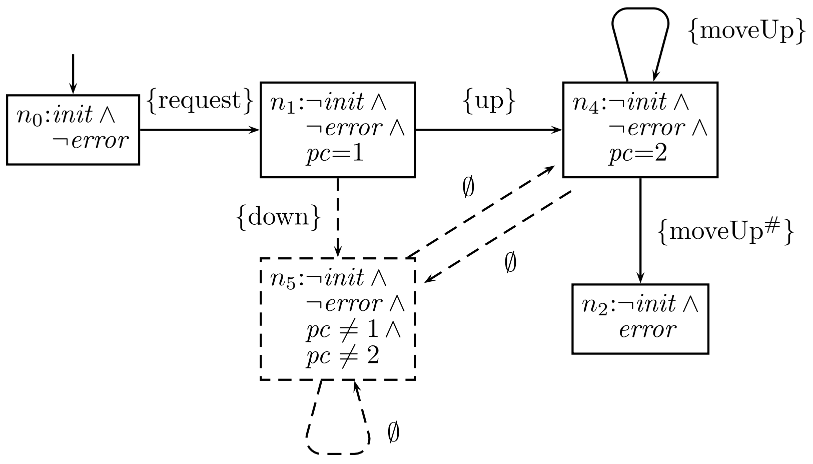
Step 2. Since *init* ∧ *error* is unsatisfiable, node n_3 is eliminated. Similarly, transition relations that are inconsistent with the labels on the nodes are removed from the edges. For example, all transitions on the edge from node n_0 to node n_2 (and hence the entire edge) are eliminated. The ready transition no longer appears in the abstraction. The transition relation for moveUp on edge (n_1, n_2) simplifies to $pc{=}2 \wedge req > cur \wedge cur'{=}cur + 1$, resulting in the new transition moveUp$^{\#}$, because the variables *req* and *pc* are not live in n_2 ($\neg init \wedge error$ simplifies to $cur > Max$, which does not contain *req* or *pc*.)



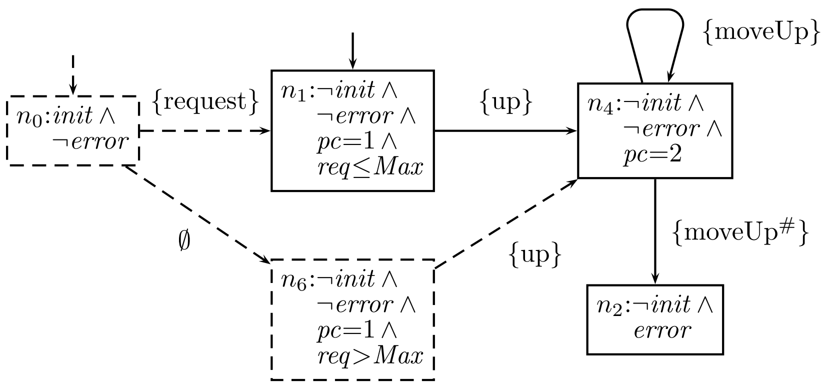
Step 3. The first refinement step splits node n_1 with predicate $pc{=}1$. All edges leading into or out of n_1, including its self loop, are duplicated. After slicing inconsistent parts of the abstraction, transition moveDn on the self loop at n_5 commutes with both outgoing transitions moveUp and moveUp$^{\#}$. It is therefore removed by partial order reduction; since the successor node n_3 is an error node, moveDn is effectively postponed until after the error is reached, and thus removed from consideration.



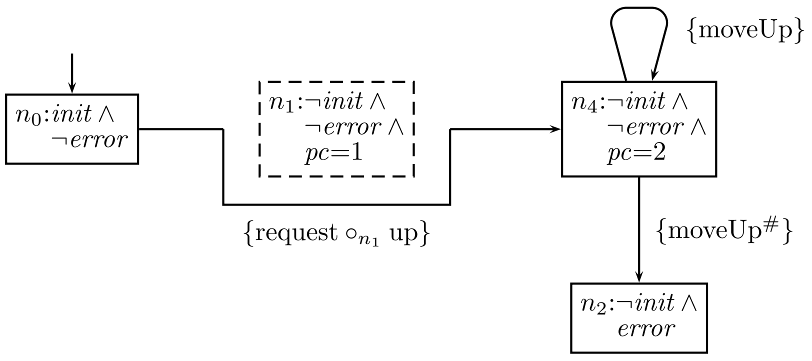
Step 4. After splitting node n_4 with predicate $pc{=}2$, the edge (n_5, n_4) can be removed. As a consequence, there are no more paths from node n_5 to an error state, and it is deleted.


{request}
{up}
{moveUp}
n_0:*init* ∧ ¬*error*
n_1:¬*init* ∧ ¬*error* ∧ *pc*=1
n_4:¬*init* ∧ ¬*error* ∧ *pc*=2
{down}
∅
∅
{moveUp$^{\#}$}
n_5:¬*init* ∧ ¬*error* ∧ *pc* ≠ 1 ∧ *pc* ≠ 2
n_2:¬*init* ∧ *error*
∅

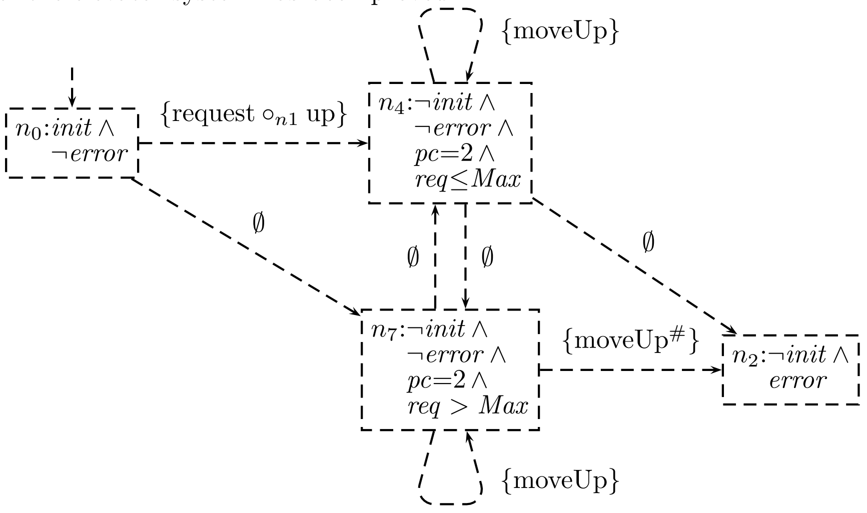
Step 5a: Source enlargement adds new initial nodes: Splitting n_1 with the strongest postcondition of request with respect to the label of n_0 results in a node whose label is only satisfied by reachable concrete states. Node n_1 can therefore be added to the set of initial nodes even though the label does not imply *init*. As n_1 is initial, we can remove the incoming edge (n_0, n_1). The second copy of node n_1, node n_6, is unreachable after slicing inconsistent transitions and empty edges, and is therefore removed. Finally, node n_0 is deleted because it no longer has a path to an error node.


{moveUp}
{request}
{up}
n_0:*init* ∧ ¬*error*
n_1:¬*init* ∧ ¬*error* ∧ *pc*=1 ∧ *req*≤*Max*
n_4:¬*init* ∧ ¬*error* ∧ *pc*=2
∅
{up}
{moveUp$^{\#}$}
n_6:¬*init* ∧ ¬*error* ∧ *pc*=1 ∧ *req*>*Max*
n_2:¬*init* ∧ *error*

Step 5b: As an alternative to source enlargement, node n_1 can be bypassed via transition relation request $\circ_{n_1}$ up, which summarizes the two transition relations and the node label.


{moveUp}
n_0:*init* ∧ ¬*error*
n_1:¬*init* ∧ ¬*error* ∧ *pc*=1
n_4:¬*init* ∧ ¬*error* ∧ *pc*=2
{request $\circ_{n_1}$ up}
{moveUp$^{\#}$}
n_2:¬*init* ∧ *error*

Step 6: We continue with the result from Step 5b by splitting node n_4 with predicate $req \leq Max$. After the elimination of inconsistent transitions, the initial and error nodes are in different connected components of the abstraction. Therefore all nodes are eliminated and the abstraction is empty. The correctness of the elevator system has been proved.


{moveUp}
n_0:*init* ∧ ¬*error*
{request $\circ_{n1}$ up}
n_4:¬*init* ∧ ¬*error* ∧ *pc*=2 ∧ *req*≤*Max*
∅
∅
∅
∅
n_7:¬*init* ∧ ¬*error* ∧ *pc*=2 ∧ *req* > *Max*
{moveUp$^{\#}$}
n_2:¬*init* ∧ *error*
{moveUp}

4.3 Abstraction

Our abstractions are graphs where the nodes are labeled with sets of predicates and the edges are labeled with sets of transition relations. They contain a subset N^0 of *initial* nodes, representing the initial states of the system. For a state-based error condition *error*, we assume additionally a subset N^e of error nodes, representing the error states given by *error*.

Definition 4.1 *An* abstraction $\mathcal{A} = (N, N^0, E, \nu, \eta)$ *of a transition system* $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ *consists of the following components:*

- *a finite set N of nodes,*
- *a subset $N^0 \subseteq N$ of initial nodes,*
- *a subset $N^e \subseteq N$ of error nodes,*

- *a set* $E \subseteq N \times N$ *of edges,*
- *a labeling* $\nu : N \to \mathit{Asrt}(\mathcal{V})$ *of nodes with assertions, and*
- *a labeling* $\eta : E \to 2^{\mathbb{T}(\mathcal{V},\Sigma)}$ *of edges with finite sets of transitions.*

A *path* of an abstraction is a finite alternating sequence of nodes and transitions $n_0, \tau_1, n_1, \tau_1, \ldots, \tau_k, n_k$ such that for all $1 \leq i \leq k, (n_{i-1}, n_i) \in E$ and $\tau_i \in \eta(n_{i-1}, n_i)$.

An *error path* is a path $n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ which starts with an initial node $n_0 \in N^0$ and violates the property that we want to verify. In this chapter, that amounts to n_k belonging to N^e.

A path $n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ is *concretizable* in $\mathcal{S}$ if there exist states $s_0, \ldots, s_k$ such that for every position $0 \leq i \leq k$, $\nu(n_i)(s_i)$ holds and for every position $1 \leq i \leq k$, $\tau_i(s_{i-1}, s_i)$ holds. We call the alternating sequence of system states and transitions $s_0, \tau_1, s_1, \ldots, \tau_k, s_k$ a *concretization* of $n_0, \tau_1, n_1, \ldots, \tau_k, n_k$. An abstract error path that is not concretizable is called *spurious*. An abstraction $\mathcal{A}$ of a transition system $\mathcal{S}$ is *sound* if there exists a concretizable error path in $\mathcal{A}$ if and only if $\mathcal{S}$ is not correct. Our abstraction refinement procedure starts with a sound initial abstraction and then preserves soundness in each transformation. The verification process terminates as soon as the abstraction has a concretizable error path (in which case the system is incorrect) or is empty (in which case the system is correct).

4.3.1 Initial abstraction

We now define the initial abstraction. Since we assume a state-based error condition, which we can represent in the abstraction itself, we obtain four abstract states corresponding to the possible boolean combinations of *init* and *error*.

Definition 4.2 *The* initial abstraction $\mathcal{A}_0 = (N, N^0, N^e, E, \nu, \eta)$ *of a transition system* $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ *with respect to an error condition error* $\in \mathit{Asrt}(\mathcal{V})$ *consists of the following components:*

- $N = \{ie, \bar{i}e, i\bar{e}, \overline{ie}\}$,
- $N^0 = \{ie, i\bar{e}\}$,
- $N^0 = \{ie, \bar{i}e\}$,
- $E = \{(i\bar{e}, \overline{ie}), (i\bar{e}, \bar{i}e), (\overline{ie}, \bar{i}e), (\overline{ie}, \overline{ie})\}$,
- $\nu(n) = \begin{cases} \mathit{init} \land \mathit{error} & \mathit{if}\ n = ie, \\ \neg\mathit{init} \land \mathit{error} & \mathit{if}\ n = \bar{i}e, \\ \mathit{init} \land \neg\mathit{error} & \mathit{if}\ n = i\bar{e}, \\ \neg\mathit{init} \land \neg\mathit{error} & \mathit{if}\ n = \overline{ie} \end{cases},$
- $\eta(e) = \mathcal{T}$ *for all* $e \in E$.

The initial abstraction is shown in Figure 4.3. The edge set E is already reduced according to the *Initiality Subsumption* and *Error Subsumption* rules, as explained in the introduction.

Proposition 4.3 *The initial abstraction $\mathcal{A}_0$ of a transition system $\mathcal{S}$ with respect to error is sound.*

Proof: By definition, the concretization of an error path of $\mathcal{A}_0$ is the prefix of a run of $\mathcal{S}$ that leads to a state that satisfies the error condition. Hence, the existence of a concretizable error path implies that $\mathcal{S}$ is not correct. Suppose, on the other hand, that $\mathcal{S}$ is not correct, i.e., there exists a run $s_0, \tau_1, s_1, \ldots, \tau_m, s_m$ such that $\mathit{error}(s_k)$ holds for some $0 \leq k \leq m$. Let i be the greatest index between 0 and k such that $\mathit{init}(s_i)$ holds, and let j be the smallest index between i and k such that $\mathit{error}(j)$ holds. Then $s_i, \tau_{i+1}, s_{i+1}, \ldots, \tau_j, s_j$ defines a concretizable abstract path $n_i, \tau_{i+1}, n_{i+1}, \ldots, \tau_j, n_j$ as follows: for $l = i, \ldots, j$,

$$n_l = \begin{cases} ie & \text{if } \quad \mathit{init}(s_l) \wedge \quad \mathit{error}(s_l), \\ \bar{i}e & \text{if } \neg \mathit{init}(s_l) \wedge \quad \mathit{error}(s_l), \\ i\bar{e} & \text{if } \quad \mathit{init}(s_l) \wedge \neg \mathit{error}(s_l), \\ \overline{ie} & \text{if } \neg \mathit{init}(s_l) \wedge \neg \mathit{error}(s_l). \end{cases}$$

Since $n_i \in N^0$ and $n_j \in N^e$, $n_i, \tau_{i+1}, n_{i+1}, \ldots, \tau_j, n_j$ is an error path.

□

4.3.2 Abstraction Refinement

We first introduce the refinement step for a given predicate and node, show that it maintains soundness of the abstraction, and then discuss how both can be obtained automatically by analysis of a spurious error path.

Node splitting. Given some new predicate q, we split an abstract node labeled φ into two copies, one labeled $\varphi \wedge q$, the other $\varphi \wedge \neg q$.

Node split Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ with respect to *error*, and let $n \in N$ be some abstract node and $q(\mathcal{V})$ some predicate. Splitting node n with q results in the new abstraction $\mathcal{A}' = (N', N^{0\prime}, N^{e\prime}, E', \nu', \eta')$, where

- $N' = N \cup \{n^-\}$ where $n^- \notin N$ is a fresh node;
- $N^{0\prime} = \begin{cases} N^0 \cup \{n^-\} & \text{if } n \in N^0, \\ N^0 & \text{otherwise,} \end{cases}$
- $N^{e\prime} = \begin{cases} N^e \cup \{n^-\} & \text{if } n \in N^e, \\ N^e & \text{otherwise.} \end{cases}$
- $E' = \bigcup_{e \in E} \mathit{split}(e)$, where
$$\mathit{split}(e) = \begin{cases} \{e, (n, n^-), (n^-, n), (n^-, n^-)\} & \text{if } e = (n, n), \\ \{e, (m, n^-)\} & \text{if } e = (m, n), m \neq n, \\ \{e, (n^-, m)\} & \text{if } e = (n, m), m \neq n, \\ \{e\} & \text{otherwise,} \end{cases}$$
- $\nu'(m) = \begin{cases} \nu(n) \wedge q & \text{if } m = n, \\ \nu(n) \wedge \neg q & \text{if } m = n^-, \\ \nu(m) & \text{otherwise,} \end{cases}$ and
- $\eta'(e') = \eta(e)$ for all $e' \in \mathit{split}(e)$.

The elevator example involves several node splits. For instance, in Step 3, node n_1 is split into the new nodes n_4 and n_5 with the predicate $pc{=}1$.

Proposition 4.4 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ with respect to error $\in Asrt(\mathcal{V})$, and let $n \in N$ be some abstract node and $q(\mathcal{V})$ be a predicate. Then the result of applying* node split *to $\mathcal{A}$ with respect to n and q is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: If $\mathcal{S}$ is not correct, then the sound abstraction $\mathcal{A}$ contains an error path $n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ which has a concretization $s_0, \tau, s_1, \ldots, \tau_k, s_k$. In this case, the node split $\mathcal{A}'$ of $\mathcal{A}$ with respect to n and q contains the concretizable error path $n'_0, \tau_1, n'_1, \ldots, \tau_k, n'_k$ where, for all $0 \leq i \leq k$,

$$n'_i = \begin{cases} n_i & \text{if } n_i \in N \setminus \{n\}, \\ n & \text{if } n_i = n \text{ and } q(s_i), \\ n^- & \text{if } n_i = n \text{ and } \neg q(s_i). \end{cases}$$

Suppose, on the other hand, that $\mathcal{A}'$ contains a concretizable error path $n'_0, \tau'_0, n'_1, \tau'_1, \ldots, \tau'_{k-1}, n'_k$, then $\mathcal{A}$ contains the concretizable error path $n_0, \tau'_0, n_1, \tau'_1, \ldots, \tau'_{k-1}, n_k$ where $n_i = n'_i$ if $n'_i \in N$, and $n_i = n$ if $n'_i = n^-$, implying that $\mathcal{S}$ is not correct. Hence, $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to *error*.

□

Error path analysis. The refinement process is driven by the analysis of spurious error paths. Our technique is based on Craig interpolation. In order to obtain the new predicate, we use a variation of a standard error path *cutting* technique [40] from predicate abstraction, which splits the path into two subsequences such that the new predicate is an interpolant for the first-order formulas corresponding to the first and second parts. To ensure that the new predicate affects as many error paths as possible, we focus on minimal spurious subpaths:

For a spurious error path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$, we call a subpath $n_i, \tau_i, n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$ with $0 \leq i < j \leq k$ *minimal* if the subpath is not concretizable but both $n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$ and $n_i, \tau_{i+1}, \ldots, n_{j-1}$ are concretizable.

We translate paths to first-order formulas in the following way. Let, for each $i \in \mathbb{N}$, $\mathcal{V}_i$ be a set of fresh variables such that for each $v \in \mathcal{V}$, $\mathcal{V}_i$ contains a corresponding fresh variable $v_i \in \mathcal{V}_i$. Given a finite path $p = n_i, \tau_i, n_i, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$ in an abstraction $\mathcal{A}$, we define two first-order formulas

$$\begin{aligned} \Gamma_1(p) &= \nu(n_i)(\mathcal{V}_i) \wedge \rho_{\tau_i}(\mathcal{V}_i, \mathcal{V}_{i+1}) \wedge \nu(n_{i+1})(\mathcal{V}_{i+1}) \wedge \ldots \wedge \nu(n_{j-1})(\mathcal{V}_{j-1}), \\ \Gamma_2(p) &= \rho_{\tau_{j-1}}(\mathcal{V}_{j-1}, \mathcal{V}_j) \wedge \nu(n_j)(\mathcal{V}_j) \end{aligned}$$

and their conjunction $\Gamma(p) = \Gamma_1(p) \wedge \Gamma_2(p)$. The path p is concretizable iff the formula $\Gamma(p)$ is satisfiable. We analyze a given spurious error path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ in two steps:

1. We find a minimal subpath $p = n_i, \tau_i, n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$. This determines the node $n = n_{j-1}$ which will be split.

2. We compute the interpolant of $\Gamma_1(p)$ and $\Gamma_2(p)$. The interpolant $\eta(\mathcal{V}_{j-1})$ defines the new predicate $q = \eta(\mathcal{V})$ on which we split node n.

After Step 2 of the elevator example, we obtain the abstract error path $p = n_0, \text{request}, n_1, \text{moveUp}^{\#}, n_2$, corresponding to the formula

$$\begin{array}{rcl} \Gamma(p) & = & pc_0{=}0 \land cur_0{\leq}Max \land in_0{\leq}Max \\ & \land & pc_0{=}0 \land pc_1{=}1 \land cur_1{=}current_0 \land req_1{=}in_0 \\ & \land & (pc_1 \neq 0 \lor in_1 > Max) \land cur_1{\leq}Max \\ & \land & pc_1{=}2 \land req_1 > cur_1 \land pc_2{=}2 \land cur_2{=}cur_1 + 1 \\ & \land & cur_2 > Max. \end{array}$$

Because of the conjuncts $pc_1{=}1$ (from $\text{request}(\mathcal{V}_0, \mathcal{V}_1)$) and $pc_1{=}2$ (from $\text{moveUp}^{\#}(\mathcal{V}_1, \mathcal{V}_2)$), the formula is unsatisfiable. The error path is minimal, since both $n_1, \text{request}, n_2$ and $n_2, \text{moveUp}^{\#}, n_3$ are concretizable. Hence, n_2 is selected for the split. The interpolant of $\Gamma_1(p)$ and $\Gamma_2(p)$ is the predicate $pc_1 = 1$, which is implied by $\Gamma_1(p)$ and contradicts $\Gamma_2(p)$.

4.4 Elimination Rules

4.4.1 Eliminating transitions

The abstraction may contain transitions that are irrelevant because the predicates on the source and target nodes of the edge contradict the transition relation. Such transitions are eliminated in the slice:

Inconsistent Transition Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains a transition $\tau \in \eta(m, n)$ on some edge $(m, n) \in E$ that is *inconsistent* with the node labels, i.e., the formula $\nu(m) \land \rho_\tau \land \nu(n)'$ is unsatisfiable. We remove τ, resulting in the abstraction $\mathcal{A}' = (N, N^0, N^e, E, \nu, \eta')$, where

$$\eta'(e) = \begin{cases} \eta(e) \smallsetminus \{\tau\} & \text{if } e = (m, n) \\ \eta(e) & \text{otherwise.} \end{cases}$$

Empty Edges Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains an edge $e \in E$ with $\eta(e) = \emptyset$. Any such edge can be removed, resulting in the abstraction $\mathcal{A}' = (N, N^0, N^e, E', \nu, \eta|_{E'})$, where $E' = \{e \in E \mid \eta(e) \neq \emptyset\}$.

In Step 2 of the elevator example, all transition relations on the edge between nodes n_0 and n_2 are contradicted by the predicates on the nodes: there is no transition that leads directly from an initial state to an error state (the only transition enabled in the initial state is request, which does not modify cur; but $init$ requires $cur \leq Max$, and $error$ requires $cur > Max$). As a result, all transitions are removed from the label according to the *Inconsistent Transition* operation, and the empty edge is removed from the abstraction according to the *Empty Edges* operation.

Proposition 4.5 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ with respect to $error \in Asrt(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Inconsistent Transition *or* Empty Edges. *Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to $error$.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. Obviously, every error path in $\mathcal{A}'$ is also an error path in $\mathcal{A}$. Conversely, suppose that $p = n_0, \tau_1, n_1, \ldots, n_k$ is a concretizable error path in $\mathcal{A}$ and that transition τ_i has been eliminated. Then $\nu(n_i) \wedge \rho_{\tau_{i+1}} \wedge \nu(n_{i+1})'$ must be unsatisfiable, contradicting the assumption that p is concretizable.

□

4.4.2 Eliminating nodes

Nodes are removed from the abstraction if they are either labeled with an inconsistent combination of predicates or do not occur on any error paths.

Inconsistent Node Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains a node $n \in N$ such that $\nu(n)$ is unsatisfiable. We remove n, resulting in the abstraction $\mathcal{A}' = (N', N^{0\prime}, N^{e\prime}, E', \nu|_{N'}, \eta|_{E'})$, where $N' = N \smallsetminus \{n\}$, $N^{0\prime} = N^0 \smallsetminus \{n\}$, $N^{e\prime} = N^e \smallsetminus \{n\}$, and $E' = E \cap (N' \times N')$.

Unreachable Node Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains a node $n \in N$ which is unreachable from initial nodes or from which no error node can be reached. We remove n, resulting in $\mathcal{A}' = (N', N^{0\prime}, N^{e\prime}, E', \nu|_{N'}, \eta|_{E'})$, where $N' = N \smallsetminus \{n\}$,$N^{0\prime} = N^0 \smallsetminus \{n\}$, $N^{e\prime} = N^e \smallsetminus \{n\}$, and $E' = E \cap (N' \times N')$.

In Step 2 of the elevator example, the *inconsistent node* n_3 is removed, since the conjunction $\mathit{init} \wedge \mathit{error}$ implies $\mathit{cur} \leq \mathit{Max} \wedge \mathit{cur} > \mathit{Max}$, which is unsatisfiable. *Unreachable nodes* are removed in Step 4 (node n_5), Step 5a (nodes n_0 and n_6), Step 5b (node n_1), and Step 6 (the entire abstraction).

Proposition 4.6 *Let $\mathcal{A} = (N, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ with respect to $\mathit{error} \in \mathit{Asrt}(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Inconsistent Node *or* Unreachable Node*. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: The reduction of the graph does not eliminate any concretizable error paths, because, by definition, each node along such a path has a consistent combination of predicates and occurs on an error path. Hence, $\mathcal{A}$ has a concretizable error path iff the same path is a concretizable error path in $\mathcal{A}'$.

□

4.5 Simplification Rules

4.5.1 Simplifying transition relations

The next slicing mechanism removes constraints from transition relations that are irrelevant for the existence of a concretizable error path. For this purpose we assign to each node $n \in N$ a set of *live* variables $L(n) \subseteq \mathcal{V}$, containing all variables whose value may possibly affect the existence of a concretizable path from n to the error.

Just as in program slicing, the set of live variables is computed by a fixpoint computation. Initially, the live variables of a node $n \in N$ are those appearing in

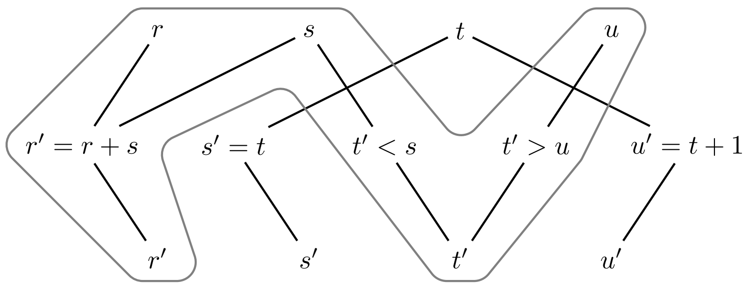


Figure 4.6: Finding the set of dependencies for r'.

its labeling $\nu(n)$ and in the enabling conditions of the transitions on outgoing edges: $L_0(n) = \mathit{vars}(\nu(n)) \cup \bigcup_{(n,m)\in E,\tau\in\eta(n,m)} \mathit{vars}(\mathit{enabled}(\tau))$.

Then, this labeling is updated according to dependencies through transition relations on edges. For a predicate q, we let $\mathit{vars}(q)$ denote the set of its free variables. For a transition τ and a set of variables X, we let $\mathit{depend}_\tau(X)$ denote the set of variables that potentially influence the value of variables in X when τ is taken: for $\rho_\tau = \bigwedge_i g_i(\mathcal{V}) \wedge \bigwedge_i t_i(\mathcal{V}, \mathcal{V}')$, $\mathit{depend}_\tau(X) = W \cap \mathcal{V}$, where $W \subseteq \mathcal{V} \cup \mathcal{V}'$ is the smallest set of variables such that $X' \subseteq W$, for all i, $\mathit{vars}(g_i) \subseteq W$, and for all i with $\mathit{vars}(t_i) \cap W \neq \emptyset$, $\mathit{vars}(t_i) \subseteq W$. The labeling is updated as follows until a fixpoint is reached: $L_{i+1}(n) = L_i(n) \cup \bigcup_{(n,m)\in E,\tau\in\eta(n,m)} \mathit{depend}_\tau(L_i(m))$.

Example: Consider the transition relation

$$\rho_\tau : r' = r + s \wedge s' = t \wedge t' < s \wedge t' > u \wedge u' = t + 1.$$

Starting with $X = \{r\}$, we get from τ:

$$r' \in W \Rightarrow r, s \in W \Rightarrow t' \in W \Rightarrow u \in W.$$

Therefore $\mathit{depend}_\tau(\{r\}) = \mathcal{V} \cap \{r', r, s, t', u\} = \{r, s, u\}$. Figure 4.6 illustrates this iteration: In the graph whose nodes are the variables in $\mathcal{V}$ and $\mathcal{V}'$ and the constraints of ρ_τ, with edges connecting each constraints to the variables it contains, the variables which can influence the value of r' are exactly those occurring in its connected component.

Assume that an abstraction contains an error node n_e with label $\nu(n_e) = r > 2$, and a node n with $(n, n_e) \in E$, $\eta(n, n_e) = \{\tau\}$, and $(n, n') \notin E$ for $n' \neq n_e$. From $\nu(n_e)$ (and the fact that error nodes have no successors) we get $L(n_e) = \{r\}$, and $L(n) = \{r, s, u\}$; therefore the transition τ can be replaced in $\eta(n, n_e)$ by $\tau^\# : r' = r + s \wedge t' < s \wedge t' > u$. Note that while $t \notin L(n_e)$, the two inequalities involving t' combine to give an implicit guard $s < u$, and removing them might therefore introduce errors.

Given the set of live variables for all nodes, we simplify the transition relations by eliminating constraints that do not refer to live variables. We assume that the conjunction of all such constraints is satisfiable (which can be achieved by a prior application of Rule *Inconsistent Transition*).

Simplify Transition Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction and let $L(n) \subseteq \mathcal{V}$ indicate the set of live variables for each node n. The simplification *simplify*(τ, m, n) of a transition τ on an edge $(m, n) \in E$ is obtained by removing from ρ_τ all conjuncts ϕ with *vars*$(\phi) \cap (L(m) \cup L(n)') = \emptyset$. In the special case of a guarded W-assignment τ, the simplification *simplify*(τ, m, n) is obtained by removing from ρ_τ all conjuncts $v' = e_v(\mathcal{V})$ with $v \notin L(n)$.

Simplifying all transitions results in the new abstraction $\mathcal{A}' = (N, N^0, N^e, E, \nu, \eta')$ where $\eta'(m, n) = \{\textit{simplify}(\tau, m, n) \mid \tau \in \eta(m, n)\}$ for all $(m, n) \in E$.

In Step 2 of the elevator example, node n_2 is labeled with the set $\{\textit{cur}, \textit{in}\}$ and nodes n_0 and n_1 are labeled with the full set of variables. As a result of the slicing operation *Simplify Transition*, the transition relation moveUp on the edge from n_1 to n_2 is simplified to moveUp$^{\#}$ by dropping the conjuncts $\textit{req}'{=}\textit{req}$ and $\textit{pc}'{=}2$.

Proposition 4.7 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \textit{init}, \mathcal{T})$ with respect to error $\in \textit{Asrt}(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Simplify Transition*. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. The implication from $\mathcal{A}$ to $\mathcal{A}'$ is straightforward since *SimplifyTransition* eliminates conjuncts from transition relations and thus every concretization of an error path in $\mathcal{A}$ is a concretization of the corresponding modified error path in $\mathcal{A}'$.

For the reverse direction assume $n_0, \tau_1, n_1, \ldots, n_k$ to be a concretizable error path in $\mathcal{A}'$ with concretization $s_0, \ldots, s_k$. Let $\widehat{\tau_i}$ be the corresponding non-simplified version of τ_i in $\mathcal{A}$. We inductively construct a concretization $\widehat{s_0}, \ldots, \widehat{s_k}$ of $n_0, \widehat{\tau_1}, n_1, \ldots, n_k$. For a state s and a set of variables $X \subseteq \mathcal{V}$, we write $s|_X$ to stand for the valuation s restricted to X. We use the operator $\oplus$ to conjoin valuations over disjoint sets of variables. The construction of the concretization starts with $\widehat{s_0} = s_0$. $\widehat{s_0}$ can be written as $s_0|_{L(n_0)} \oplus t_0$ for some valuation t_0 of variables in $\mathcal{V} \setminus L(n_0)$. Then we set $\widehat{s_1}$ to $s_1|_{L(n_1)} \oplus t_1$, where t_1 is a valuation of $\mathcal{V} \setminus L(n_1)$ such that $\phi(t_0, t_1)$ for all removed conjuncts ϕ of $\widehat{\tau_1}$. Such a t_1 exists since we assumed that the conjunction of all ϕ is satisfiable and ϕ furthermore contains no variables from *enabled*$(\widehat{\tau_1})$. Then $\widehat{\tau_1}(\widehat{s_0}, \widehat{s_1})$ since ϕ does not constrain variables in $L(n_1)$ (definition of *depends*) and the enabledness of $\widehat{\tau_1}$ is independent of ϕ $(\textit{vars}(\textit{enabled}(\widehat{\tau_1})) \subseteq L(n_0))$. This construction can analogously be continued for all states. □

4.5.2 Bypass transitions

The following construction allows us to bypass nodes without self loops. For a node n with an incoming transition τ_1 and an outgoing transition τ_2, we define the bypass relation $\rho_{\tau_1} \circ_n \rho\tau_2(\mathcal{V}, \mathcal{V}') = \exists \mathcal{V}'' . \tau_1(\mathcal{V}, \mathcal{V}'') \wedge \nu(n)(\mathcal{V}'') \wedge \tau_2(\mathcal{V}'', \mathcal{V}')$, and the bypass transition $\tau_1 \circ_n \tau_2 = (w_{\tau_1}.w_{\tau_2}, \rho_{\tau_1} \circ_n \rho\tau_2)$.

Bypass Transitions Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction and let $n \in N \setminus (N^0 \cup N^e), (n, n) \notin E$ be a node which is not an initial node, not an error node, and does not have a self loop. Every pair $(\tau_1, \tau_2) \in \eta(m,n) \times \eta(n,n')$ of an incoming transition τ_1 and an outgoing transition τ_2 is modified to bypass node n, resulting in the new abstraction $\mathcal{A}' = (N', N^{0\prime}, N^{e\prime}, E', \nu, \eta')$, where

- $N' = N \setminus \{n\}$,
- $N^{0\prime} = N^0 \setminus \{n\}$,
- $N^{e\prime} = N^e \setminus \{n\}$,
- $E' = E \cap (N' \times N') \cup \{(m,n') \mid (m,n),(n,n') \in E\}$, and
- $\eta'(m,n') = \eta(m,n') \cup \{\tau_1 \circ_n \tau_2 \mid \tau_1 \in \eta(m,n), \tau_2 \in \eta(n,n')\}$ for all $(m,n') \in E'$ (with $\eta(m,n') = \emptyset$ for $(m,n') \notin E$).

In Step 5b of the elevator example, node n_1 is bypassed via request $\circ_{n_1}$ up. As a result, n_1 becomes unreachable and is eliminated. The new transition relation is computed as follows:

$$\begin{aligned}
\rho_{\text{request}} \circ_{n_1} \rho_{\text{up}} &\Leftrightarrow \exists \mathcal{V}'' . \rho_{\text{request}}(\mathcal{V}, \mathcal{V}'') \wedge \nu(n_4)(\mathcal{V}'') \wedge \rho_{\text{up}}(\mathcal{V}'', \mathcal{V}') \\
&\Leftrightarrow \exists \mathcal{V}'' . \mathit{pc}{=}0 \wedge \mathit{pc}''{=}1 \wedge \mathit{cur}''{=}\mathit{cur} \wedge \mathit{req}''{=}\mathit{in} \\
&\qquad \wedge \neg \mathit{init}'' \wedge \neg \mathit{error}'' \wedge \mathit{pc}''{=}1 \\
&\qquad \wedge \mathit{pc}''{=}1 \wedge \mathit{req}''{>}\mathit{cur}'' \wedge \mathit{pc}'{=}2 \wedge \mathit{cur}'{=}\mathit{cur}'' \wedge \mathit{req}'{=}\mathit{req}'' \\
&\Leftrightarrow \mathit{pc}{=}0 \wedge \mathit{cur}{\leq}\mathit{Max} \wedge \mathit{in}{>}\mathit{cur} \wedge \mathit{pc}'{=}2 \wedge \mathit{cur}'{=}\mathit{cur} \wedge \mathit{req}'{=}\mathit{in}.
\end{aligned}$$

Proposition 4.8 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ with respect to $\mathit{error} \in \mathit{Asrt}(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Bypass Transitions. *Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. Assume that $n_0, \tau_1, n_1, \ldots, n_k$ is a concretizable error path in $\mathcal{A}$ and that node n_i is bypassed. Then $\mathcal{A}'$ has a concretizable error path $n_0, \tau_1, n_1, \ldots, n_{i-1}, \tau_{i-1} \circ_{n_i} \tau_i, n_{i+1}, \ldots, n_k$. Conversely, whenever $\mathcal{A}'$ has a concretizable error path containing a bypass transition τ, then there was an intermediate node n and two transitions τ_1, τ_2 such that $\tau = \tau_1 \circ_n \tau_2$ and thus a corresponding concretizable error path of $\mathcal{A}$ can be constructed. □

4.5.3 Initiality and Error Subsumption

It is intuitively clear that, for a state-based error condition, an error path that contains more than one initial or error node has at least one subpath which is also an error path. Any error paths lost by removing edges leading into N^0 or out of N^e are thus expendable.

Initiality Subsumption Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains an edge $e = (m,n) \in E$ such that $n \in N^0$. We remove e, resulting in the new abstraction $\mathcal{A} = (N, N^0, N^e, E \setminus \{e\}, \nu, \eta)$.

Error Subsumption Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction, and let $e = (m,n) \in E$ be an edge with $m \in N^e$. We remove e, resulting in the new abstraction $\mathcal{A} = (N, N^0, N^e, E \setminus \{e\}, \nu, \eta)$.

Proposition 4.9 *Let $\mathcal{A} = (N, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ with respect to $error \in Asrt(\mathcal{V})$. Applying* Initiality Subsumption *or* Error Subsumption *to remove (m, n) from $\mathcal{A}$ results in another sound abstraction $\mathcal{A}'$ of $\mathcal{S}$ with respect to error.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. Obviously, every error path in $\mathcal{A}'$ is also an error path in $\mathcal{A}$. Conversely, suppose that $p = n_0, \tau_1, n_1, \dots, n_k$ is a concretizable error path in $\mathcal{A}$. Let $i \in \{1, \dots, k\}$ be the largest index such that $n_i \in N^0$, and let j be the smallest index in $\{j, \dots, k\}$ with $n_j \in N^e$.

$p' = n_i, \tau_{i+1}, n_{i+1}, \dots, n_j$ is obviously also a concretizable error path, and since it contains no edges (m, n) with $m \in N^e$ or $n \in N^0$, p' still exists in $\mathcal{A}'$.

□

4.5.4 Source and Target Enlargement

The *Source Enlargement* and *Target Enlargement* transformations reduce the length of error paths by identifying parts of the state space that are guaranteed to be forward reachable from the initial states or backward reachable from the error states. Nodes representing such state sets can be included in the set of initial and error nodes, respectively.

In the definition of the transformations we use the standard *preimage* and *strongest postcondition* operators pre and post: $\text{pre}(\rho, \phi)(\mathcal{V}) \Leftrightarrow \exists \mathcal{V}'(\rho(\mathcal{V}, \mathcal{V}') \wedge \phi(\mathcal{V}'))$, $\text{post}(\rho, \phi)(\mathcal{V}') \Leftrightarrow \exists \mathcal{V}(\rho(\mathcal{V}, \mathcal{V}') \wedge \phi(\mathcal{V}))$.

Source Enlargement Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains a node $n \in N$ such that $\nu(n) \Rightarrow \bigvee_{i \in N^0} \bigvee_{\tau \in \eta(i,n)} \text{post}(\rho_\tau, \nu(i))$. We add n to N^0, obtaining $\mathcal{A} = (N, N^0 \cup \{n\}, N^e, E, \nu, \eta)$.

Target Enlargement Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction that contains a node $n \in N$ such that $\nu(n) \Rightarrow \bigvee_{f \in N^e} \bigvee_{\tau \in \eta(n,f)} \text{pre}(\rho_\tau, \nu(f))$. We add n to N^e, obtaining $\mathcal{A} = (N, N^0, N^e \cup \{n\}, E, \nu, \eta)$.

A node that satisfies the conditions of *Source Enlargement* can be obtained by splitting a successor n of an initial node i with the strongest postcondition of a transition connecting i with n.

Example: Consider Step 5a of the elevator verification. Splitting node n_1 with

$$\varphi = \text{post}(\rho_{\text{request}}, init \wedge \neg error) \equiv pc{=}1 \wedge cur \leq Max \wedge req{\leq}Max,$$

we obtain two nodes, n_1 with $\nu(n_1) = \neg init \wedge \neg error \wedge pc{=}1 \wedge req{\leq}Max$ and n_6 with $\nu(n_6) = \neg init \wedge \neg error \wedge pc{=}1 \wedge req{>}Max$. *Source Enlargement* adds n_1 to N^0, and *Initiality Subsumption* then deletes the edge (n_0, n_1). Node n_6 is eliminated by the slicing rules. Node n_0 then has no more outgoing edges and is also removed.

For *Target Enlargement*, a node that satisfies the conditions can analogously be obtained by splitting the predecessor n of an error node f with the preimage of a transition connecting n with f.

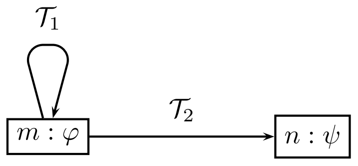


Figure 4.7: Abstraction with self loop in node m. Partial order elimination removes a transition $\tau \in \mathcal{T}_1$ on the self loop if τ is unblocked by every other transition in $\mathcal{T}_1 \cup \mathcal{T}_2$.

Proposition 4.10 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ with respect to $error \in Asrt(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Source Enlargement *or* Target Enlargement *to a node $n \in N$. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: Let $\mathcal{A}'$ be the result of applying *Source Enlargement* to node n in $\mathcal{A}$. We show that $\mathcal{A}$ contains a concretizable error path iff $\mathcal{A}'$ contains a concretizable error path.

Any error path that exists in $\mathcal{A}$ obviously still exists in $\mathcal{A}'$.

For the implication from $\mathcal{A}'$ to $\mathcal{A}$, let $p' = n'_0, \tau'_1, n'_1, \tau'_2, \ldots, \tau'_k, n'_k$ be a concretizable error path in $\mathcal{A}'$ such that $n_0 = n$, and let $s'_0, s'_1, \ldots s'_k$ be a concretization of p'. Since $\nu(n'_0) \Rightarrow \bigvee_{i \in N^0} \bigvee_{\tau \in \eta(n'_0, f)} \text{post}(\rho_\tau, \nu(i))$, there exists a node $i \in N^0$, a state t that satisfies $\nu(i)$, and a transition τ such that $\rho_\tau(t, s)$ holds. We therefore obtain an error path $p = i, \tau, n'_0, \tau'_1, n'_1, \tau'_2, \ldots, \tau'_k, n'_k$ in $\mathcal{A}$ with concretization $t, s'_0, s'_1, \ldots s'_k$.

The proof for *Target Enlargement* is analogous. $\square$

4.5.5 Partial Order Reduction

Partial order reduction (cf. [35]) identifies commuting transitions in order to eliminate redundant interleavings. The central concept of our version of this idea is that of *unblocked* transitions: Consider a system in which, whenever transition τ followed by transition τ' leads from a state s to some state s', τ' followed by τ also leads from s to s'. If a sequence t of transitions can be transformed into another sequence t' by swapping replacing the subsequence τ, τ' with τ', τ, then for any concretization of t there is a corresponding concretization of t', and t is redundant if both sequences are possible in $\mathcal{S}$.

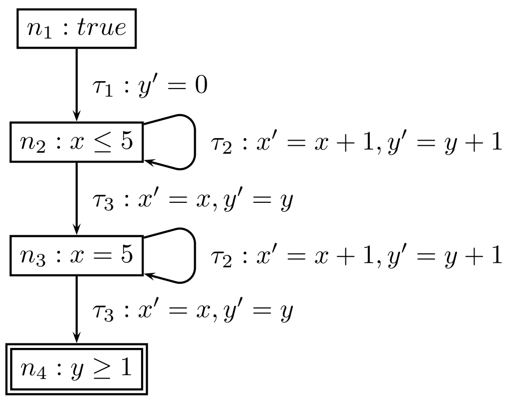
In an abstraction, we can define a specialized partial order reduction rule that is useful to eliminate transitions from self loops in the abstraction. Consider the abstraction in Figure 4.7. We define $\tau \in \mathcal{T}_1$ to be unblocked by $\tau' \in \mathcal{T}_2$ if for any concretization of $p = m, \tau, m, \tau', n$ there is a concretization of $p' = m, \tau', n, \tau, n$ with the same concrete source and target states.

In terms of the node labels and transition relations, this means that τ is unblocked by τ' with respect to m, n if

$$\nu(m) \wedge (\rho_{\tau_1} \circ_m \rho_{\tau_2}) \wedge \nu(n)' \qquad \Rightarrow \qquad \nu(m) \wedge (\rho_{\tau_2} \circ_n \rho_{\tau_1}) \wedge \nu(n)',$$

using the same notation

$$\rho_1 \circ_n \rho_2(\mathcal{V}, \mathcal{V}') = \exists \mathcal{V}'' \, . \, \rho_1(\mathcal{V}, \mathcal{V}'') \wedge \nu(n)(\mathcal{V}'') \wedge \rho_2(\mathcal{V}'', \mathcal{V}')$$



$n_1 : true$
$\tau_1 : y' = 0$
$n_2 : x \le 5$
$\tau_2 : x' = x+1, y' = y+1$
$\tau_3 : x' = x, y' = y$
$n_3 : x = 5$
$\tau_2 : x' = x+1, y' = y+1$
$\tau_3 : x' = x, y' = y$
$n_4 : y \ge 1$

Figure 4.8: Taking only the transition constraints into account would lead to an unsound partial order reduction rule: The identity transition τ_3 commutes trivially with τ_2, but τ_2 cannot be removed, since there are concrete paths corresponding to $n_2 \xrightarrow{\tau_2} n_2 \xrightarrow{\tau_3} n_3$, but none for $n_2 \xrightarrow{\tau_3} n_3 \xrightarrow{\tau_2} n_3$.

as for the *Bypass Transitions* rule.

Any transition $\tau \in \mathcal{T}_1$ on the self loop in node m that is unblocked by every other transition in $\mathcal{T}_1 \cup \mathcal{T}_2$ is redundant: all states reachable through τ (on the self loop) followed by some transition $\tau' \in \mathcal{T}_1 \cup \mathcal{T}_2$ are also reachable through τ' followed by τ. It is therefore sound to eliminate τ from the self loop in node m.

Note that we need to take into account the constraints of the respective intermediate nodes. For example, consider the abstraction given in Figure 4.8. Since transition τ_3 is the identity, it commutes with any other transition, and in particular we have $s_1 \xrightarrow{\tau_3}\xrightarrow{\tau_2} s_2$ whenever $s_1 \xrightarrow{\tau_2}\xrightarrow{\tau_3} s_2$ for states s_1, s_2. Removing τ_2 would, however, remove the concretizable error path $n_1 \xrightarrow{\tau_1} n_2 \xrightarrow{\tau_2} n_2 \xrightarrow{\tau_3} n_3 \xrightarrow{\tau_3} n_4$, and the resulting abstraction would contain no more errors.

Partial Order Reduction Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be an abstraction containing a node $m \in N$ with $(m,m) \in E$ and a transition $\tau_1 \in \eta(m,m)$. If for all nodes $n \in N$ such that $(m,n) \in E$ and transitions $\tau_2 \in \eta(m,n)$, $\nu(m) \wedge (\rho_{\tau_1} \circ_m \rho_{\tau_2}) \wedge \nu(n)' \Rightarrow \nu(m) \wedge (\rho_{\tau_2} \circ_n \rho_{\tau_1}) \wedge \nu(n)'$, then τ_1 is removed from $\eta(m,m)$, resulting in $\mathcal{A}' = (N, N^0, N^e, E, \nu, \eta')$, where $\eta'(m,m) = \eta(m,m) \setminus \{\tau_1\}$ and $\eta'(e) = \eta(e)$ for $e \neq (m,m)$.

In Step 3 of the verification of the elevator, we can remove transition moveDn $\in \eta(n_4, n_4)$:

- moveDn is trivially unblocked by moveUp $\in \eta(n_4, n_4)$, in the sense that $\nu(n_4) \wedge (\rho_{\text{moveDn}} \circ_{n_4} \rho_{\text{moveUp}}) \wedge \nu(n_4)$ is unsatisfiable: moveDn can never be immediately followed by moveUp, since it leads to pc=3, while moveUp requires pc=2.
- moveDn is also trivially unblocked by moveUp$^{\#}$ $\in \eta(n_4, n_2)$, for the same reason.

As another example, consider the abstraction in Figure 4.7, where we assume

- $\phi \equiv y > 0$,
- $\psi \equiv y \leq 0$,
- $\mathcal{T}_1 = \{\tau_1\}$ with $\rho_{\tau_1} \equiv x' \geq x + y \wedge y' = y$, and
- $\mathcal{T}_2 = \{\tau_2\}$ with $\rho_{\tau_2} \equiv x' = x \wedge y' = 0$.

The formulas we need to compare are:

$$\begin{aligned}
&\exists x'', y'' : \phi \wedge \rho_{\tau_1}(\mathcal{V}, \mathcal{V}'') \wedge \phi'' \wedge \rho_{\tau_2}(\mathcal{V}'', \mathcal{V}') \wedge \psi' \\
\Leftrightarrow\ &\exists x'', y'' : y > 0 \wedge x'' \geq x + y \wedge y'' = y \wedge y'' > 0 \wedge x' = x'' \wedge y' = 0 \wedge y' \leq 0 \\
\Leftrightarrow\ &y > 0 \wedge x' \geq x + y \wedge y' = 0
\end{aligned}$$

and

$$\begin{aligned}
&\exists x'', y'' : \phi \wedge \rho_{\tau_2}(\mathcal{V}, \mathcal{V}'') \wedge \psi'' \wedge \rho_{\tau_1}(\mathcal{V}'', \mathcal{V}') \wedge \psi' \\
\Leftrightarrow\ &\exists x'', y'' : y > 0 \wedge x'' = x \wedge y'' = 0 \wedge y'' \leq 0 \wedge x' \geq x'' + y'' \wedge y' = y'' \wedge y' \leq 0 \\
\Leftrightarrow\ &y > 0 \wedge x' \geq x \wedge y' = 0.
\end{aligned}$$

It is easy to check that $(y > 0 \wedge x' \geq x + y \wedge y' = 0) \Rightarrow (y > 0 \wedge x' \geq x \wedge y' = 0)$, so that for any concretization of $m \xrightarrow{\tau_1} m \xrightarrow{\tau_2} n$ there is a concretization of $m \xrightarrow{\tau_2} n \xrightarrow{\tau_1} n$ leading from the same concrete source state to the same concrete target state, and therefore τ_1 can be removed.

Proposition 4.11 *Let $\mathcal{A} = (N, N^0, N^e, E, \nu, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, init, \mathcal{T})$ with respect to $error \in Asrt(\mathcal{V})$, and let $\mathcal{A}'$ be the result of applying* Partial Order Reduction *to $\tau \in \eta(n, n)$. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$ with respect to error.*

Proof: We show that $\mathcal{A}$ as a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. The implication from $\mathcal{A}'$ to $\mathcal{A}$ is straightforward, since every path in $\mathcal{A}'$ also exists in $\mathcal{A}$.

For the reverse direction, we assume there exists a concretizable error path p in $\mathcal{A}$ and construct a concretizable error path p' in $\mathcal{A}'$. Without loss of generality, we can assume p to be minimal, i.e. contain exactly one initial and one error node. Let t be the number of occurrences of n, τ, n in p. We prove the claim by induction on t. If $t = 0$, then $p' = p$. For $t > 0$, we construct a path p'' in $\mathcal{A}$ with $t - 1$ occurrences of n, τ, n. Let $p = n_0, \tau_1, \ldots, \tau_k, n_k$ and let i be the least index such that $n_i = n_{i+1} = n$, $\tau_{i+1} = \tau$.

If $i + 1 = k$, the error path p ends with n, τ, n, and $n \in N^e$. We can therefore use $p'' = n_0, \tau_1, n_1, \ldots, n_i$, which is also a concretizable error path and contains $t - 1$ occurrences of n, τ, n.

Let $i + 1$ be smaller than k. *Partial Order Reduction* guarantees that τ is unblocked by with τ_{i+2}. Let j, $i < j \leq k$, be the largest index such that τ is unblocked by every transition in $\{\tau_{i+2}, \ldots, \tau_j\}$.

If $j = k$, we eliminate the first occurrence of n, τ, n by transforming p into

$$p'' = n_0, \tau_1, \ldots, \tau_i, n_i, \tau_{i+2}, n_{i+2}, \ldots, \tau_k, n_k.$$

The path p'' leads to the error node n_k and is concretizable, because it can be extended to the sequence $n_0, \tau_1, \ldots, \tau_{i-1}, n_i, \tau_{i+1}, n_{i+1}, \ldots, \tau_k, n_k, \tau, n_k$, which is obtained from the concretizable path p by *Partial Order Reduction* and thus also concretizable.

For the case that $j < k$, we show that $\mathcal{A}$ must contain a self loop in n_{j+1} that contains τ. Since p is minimal, n_{j+1} can neither be an initial node nor an error node. Node n_{j+1} has therefore been constructed in a series of node splits from the node $\overline{ie}$ in the initial abstraction, which has a self loop with $\tau \in \eta(\overline{ie}, \overline{ie})$.

If τ is not in $\eta(n_{j+1}, n_{j+1})$, it must have been removed by either *Inconsistent Transition* or by *Partial Order Reduction.* It is impossible that τ was removed by *Inconsistent Transition*, because this transformation requires $\nu(n_{j+1}) \wedge \rho_\tau \wedge \nu(n_{j+1})'$ to be unsatisfiable; since the path $n, \tau_{i+1}, \ldots, \tau_j, n_{j+1}, \tau, n_{j+1}$ is concretizable, however, $\nu(n_{j+1}) \wedge \rho_\tau \wedge \nu(n_{j+1})'$ is satisfiable. Likewise, it is impossible that τ was removed by *Partial Order Reduction*, because τ is not unblocked by τ_{j+1}. We use the self loop in n_{j+1} to transform p into the error path

$$p'' = n_0, \tau_1, \ldots, \tau_{i-1}, n_i, \tau_{i+1}, n_{i+1}, \ldots, \tau_j, n_{j+1}, \tau, n_{j+1}, \ldots, n_{k+1}.$$

Since p'' is obtained from p by *Partial Order Reduction*, p'' is concretizable. Since τ is not unblocked by τ_{j+1}, n_{j+1} must be different from n. The path p thus has one more occurrence of n, τ, n than p''. □

4.6 SLAB

We have implemented our approach as a prototype verification tool named SLAB (for *sl*icing *ab*stractions). The current version, SLAB 2, is written in C++ and uses the MathSAT 4 SMT solver [13] as an underlying mechanism for satisfiability checking and Craig interpolation. In this section, we discuss the main implementation decisions in SLAB and report on experimental results.

Error path selection and analysis. SLAB annotates each node n in the abstraction with the distance $d_i(n)$ to the closest initial node and the distance $d_e(n)$ to the closest error node. Node splits and most slicing steps do not affect the distances; an update is only necessary when a complete edge is removed.

To find a minimal subpath of an error path, SLAB traverses the abstraction backwards from some error node such that $d_i(n)$ decreases in each step. Along the way, the node labels and transition relations are collected and checked for satisfiability. If the entire path to an initial node is concretizable, the system has been proved incorrect; otherwise, the backward traversal stops as soon as the formula corresponding to the suffix has become unsatisfiable, and the suffix is reexplored forwards until the shortest spurious prefix of the suffix is found.

Slicing strategy. SLAB maintains a reduced abstraction: after the initial abstraction and after each node split all applicable slicing steps are performed. Suppose a minimal error subpath $\vec{p} = n_0, \tau_1, n_1, \ldots, n_{j-1}, \tau_{j-1}, n_j$ has been identified. SLAB splits node n_{j-1} with an interpolant ϕ for $\Gamma_1(\vec{p})$ and $\Gamma_2(\vec{p})$. The labels of the new nodes n^+ and n^- are $\nu(n^+) = \nu(n_{j-1}) \wedge \phi$ and $\nu(n^-) = \nu(n_{j-1}) \wedge \neg\phi$, respectively, the labels of all other nodes remain the same. The slicing after the node split can be restricted to the affected parts of the abstraction as follows.

- It is never necessary to apply Rule *Inconsistent Node* after a node split, because labels remain satisfiable: For node n^+, $\Gamma_1(\vec{p})$ implies $\phi(V_{j-1})$.

Therefore, if $\nu(n_{j-1}) \wedge \phi(V_{j-1})$ were unsatisfiable, $\vec{p}$ would have a spurious subpath $n_0, \tau_1, n_1, \ldots, \tau_{j-1}, n_{j-1}$, which cannot be the case because $\vec{p}$ is minimal. For node n^-, the conjunction $\tau_j(V_{j-1}, V_j) \wedge \nu(n_j)(V_j)$ implies $\neg\phi(V_{j-1})$. Hence, if $\nu(n_k) \wedge \neg\phi$ were unsatisfiable, then τ_{k+1} would have been removed previously by *Inconsistent Transition*, because $\nu(n_{j-1}) \wedge \tau_j \wedge \nu(n_{k+1})'$ is inconsistent. Rule *Inconsistent Node* is therefore only applied to the initial abstraction.

- Rule *Inconsistent Transition* is applied to all transitions in $\eta(n, n')$, where n or n' is one of the new nodes n^+ or n^-.

- As a result of the elimination of transitions, edges may become empty. We update the annotation with the distances to the closest error and initial nodes as follows. Let (n, n') be an edge that was removed by Rule *Empty Edge*. If there is no longer a node n'' such that $(n, n'') \in E$ and $d_e(n) = d_e(n'') + 1$, we set $d_e(n) := \infty$, and repeat this update with all predecessors of n. Analogously, if there is no longer a node n'' such that $(n'', n') \in E$ and $d_i(n) = d_i(n'') + 1$, set $d_i(n') := \infty$, and repeat this update with all successors of n'. After this iteration terminates, we recompute the values of $d_i(.), d_e(.)$ using breadth-first search, starting from each node n with $d_i(n) = \infty$, but $d_i(n') < \infty$ for some n' with $(n', n) \in E$, and each node n with $d_e(n) = \infty$, but $d_e(n') < \infty$ for some n' with $(n, n') \in E$, respectively.

- Rule *Unreachable Nodes* eliminates all nodes n with $d_i(n) = \infty$ or $d_e(n) = \infty$.

- The integration of the simplification rules into SLAB 2 is work in progress at the time of writing.

4.6.1 Experiments

Table 4.1 shows the running time of SLAB for a collection of benchmarks. For comparison, we also give the running times of the *Abstraction Refinement Model Checker* ARMC [63] and the *Berkeley Lazy Abstraction Software Verification Tool* BLAST [42] and the *New Symbolic Model Checker* NuSMV [16], where applicable. The benchmarks include a finite-state concurrent systems (Deque and Philosophers), an infinite-state discrete system (Bakery), and a real-time system (Fischer). Because NuSMV is able to verify only finite state systems, the integer variables for its version of the benchmarks are bounded to 100 values.

Deque. The *Deque* benchmark is an abstract version of a cyclic buffer for a double-ended queue. We model the cells of the buffer by n flags, where *true* indicates a currently allocated cell. Initially, all but the first flag are *false*. Adding or deleting an element at either end is represented by toggling a flag under the condition that the values of the two neighboring flags are different: $(\mathit{true}, \mathit{true}, \mathit{false}) \leftrightarrow (\mathit{true}, \mathit{false}, \mathit{false})$ and $(\mathit{false}, \mathit{true}, \mathit{true}) \leftrightarrow (\mathit{false}, \mathit{false}, \mathit{true})$. The error condition is satisfied if there are no unallocated cells left in the buffer.

	SLAB	ARMC	BLAST	NuSMV
Benchmark	time	time	time	time
Deque 5	0.06	1.81	0.55	5.64
Deque 10	0.18	776.33	2.32	13.89
Deque 15	0.40	timeout	6.40	22.71
Deque 20	0.69	timeout	13.41	36.08
Bakery 2	0.43	2.26	21.71	0.72
Bakery 3	1.45	33.44	134.72	6.44
Bakery 4	4.00	753.15	error	293.65
Bakery 5	10.26	timeout	879.71	timeout
Philosophers 3	0.76	125.82	15.02	7.82
Philosophers 4	3.05	timeout	92.04	25.16
Philosophers 5	11.50	timeout	658.80	554.86
Philosophers 6	42.57	timeout	timeout	timeout
Fischer 2	0.65	1.45	N/A	N/A
Fischer 3	9.16	48.68	N/A	N/A
Fischer 4	122.77	1842.85	N/A	N/A

Table 4.1: Experimental results of SLAB, comparing its performance on a range of benchmarks to the tools ARMC, BLAST and NuSMV. Running times are given in seconds, with a timeout of 1 hour. All benchmarks were measured on AMD Opteron 2.6Ghz processors.(BLAST and NuSMV are not applicable to the real-time system Fischer.)

Bakery. The *Bakery* protocol [49] is a mutual exclusion algorithm that uses *tickets* to prevent simultaneous access to a critical resource. Whenever a process wants to access the shared resource, it acquires a new ticket with a value v that is higher than that of all existing tickets. Before the process accesses the critical resource, it waits until every process that is currently requesting a ticket has obtained one, and every process that currently holds a ticket with a lower value than v has finished using the resource. An error occurs if two processes access the critical resource at the same time.

Philosophers. The *Dining Philosophers* problem is a standard example of concurrency. It features a number of philosophers seated around a table, with one chopstick on the table between each pair of neighbors. In order to eat, a philosopher needs to pick up both adjacent chopsticks. The naive solution can easily lead to a deadlocked state, with each philosopher holding one chopstick.

One solution to this problem, modeled in this benchmark, involves giving the philosophers the ability to go to sleep, and requiring at least one of them to be asleep at any time. This suffices to ensure starvation-freedom.

Fischer. Fischer's algorithm, as described in [52], is a real-time mutual exclusion protocol. Access to a resource shared between n processes is controlled through a single integer variable *lock* and real-time constraints involving two fixed bounds C1 < C2. Each process uses an individual (resettable) clock c to keep track of the passing of time between transitions. Each process first checks if the lock is free, then, after waiting for no longer than bound C1, sets *lock* to its (unique) id. It then waits for at least C2, and if the value of the lock is

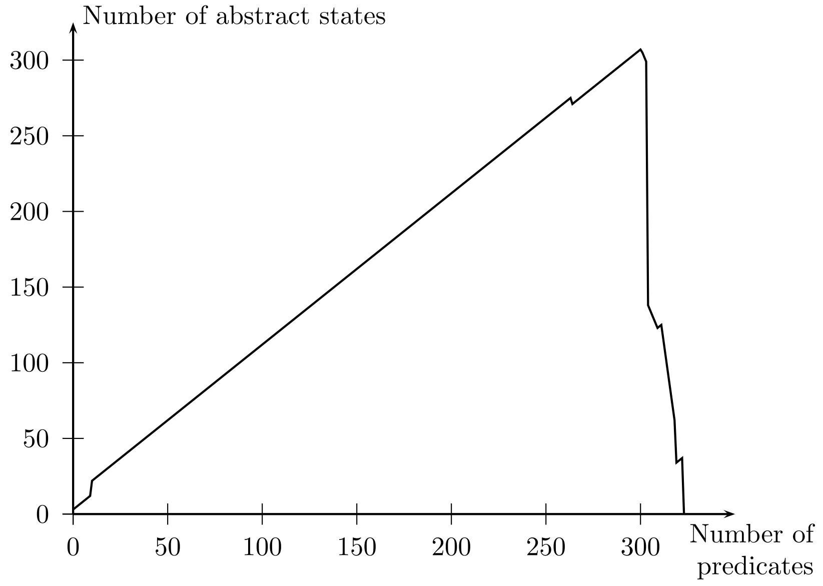


Figure 4.9: Number of abstract states in relation to the number of predicates in intermediate abstractions during the verification of a benchmark example.

unchanged, accesses the critical resource. When leaving, it frees up the lock. As in the previous benchmark, an error occurs if two processes access the critical resource at the same time.

On our benchmarks, SLAB outperforms the other tools, and scales much better to larger systems. The abstract state space constructed by SLAB grows much more slowly in the number of predicates than the (fully exponential) state space considered by standard predicate abstraction: Figure 4.9 depicts the relation between the number of predicates and the number of abstract states in intermediate abstractions from the verification of a benchmark modelling two parallel processes computing a product of two numbers.

Chapter 5

Abstraction Refinement for Subsequence Invariants

5.1 Introduction

We now show how the Slicing Abstractions procedure works for properties given by subsequence invariants over a set of transition labels. Note that state-based error conditions could, if desired, be easily transferred to this setting by introducing a dedicated *error* transition having the error condition as a guard, and requiring the system language to satisfy the invariant $|w|_{error} = 0$.

In principle, it is also possible to introduce new state variables representing the subsequence counters occurring in a set of subsequence invariants. One could then add the appropriate recurrence equations for the $|w|_U$ to all transition relations, and turn the invariants themselves into state assertions. However, this would obfuscate the characteristic features of the subsequence invariants that make their analysis efficient.

As before, we maintain a labeled graph representations of the abstract state space. In addition, we successively construct a forest of subsequence images which *covers* the graph. We add new vectors as in the exploration in Section 3.1, pruning branches which lead to linearly dependent vectors, and performing a refinement step when encountering a vector which violates one of the invariants. In this case, we preserve as much as possible of the existing trees through the refinement and slicing operations, and continue the exploration afterward. If the forest becomes *saturated* in the sense that there are no more unpruned branches, the iteration terminates – the system is correct.

Like in the state-based case, we only require the refinement procedure to preserve the (non-)existence of error paths, allowing a number of simplifications; however, since a search state now consists of both an abstract state and a synchronization history (represented by its subsequence image), the conditions under which these simplifications can be applied are more complex.

Since we have no designated error states, our initial abstraction, which is shown in Figure 5.1, only has one initial and one noninitial node. Unlike the initial abstraction in Chapter 4, we cannot simply eliminate incoming edges into nodes labeled by *init*; like the other simplification rules, *Initiality Subsumption* now has additional conditions to be applicable.

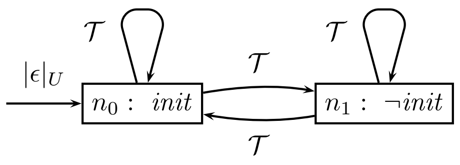


Figure 5.1: Initial abstraction, based on the predicate *init* characterizing the initial states.

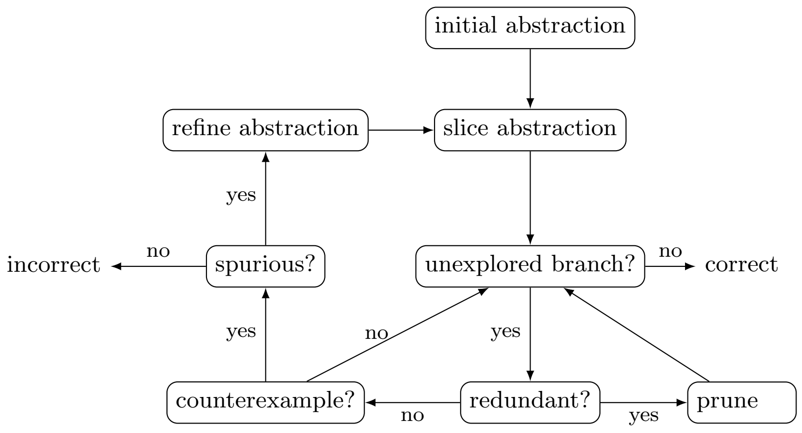


Figure 5.2: Abstraction refinement loop with exploration, refinement and slicing steps.

The label on the arrow into the initial node represents a related issue: When applying *Source Enlargement*, we now need to keep track of the prefix that we omit by starting in the new initial node. We do this by associating to each node n its *initial subspace* $\pi(n)$.

Figure 5.2 shows the abstraction refinement loop. Besides the abstraction itself, we maintain an open list of tree vertices that have not yet been examined. In each step, we take one of these vertices from the list. If the vector on the vertex is linearly dependent, we discard it. If it violates an invariant, we have either found a concretizable counterexample, in which case the system is incorrect, or we can perform a refinement step, followed by a sequence of slicing steps. If neither of these occurs, we generate the children of the selected vertex and add them to the open list.

Refinement steps work much like they did in Chapter 4: From a spurious counterexample, we obtain an interpolant representing the reason for spuriousness, and a node which we split to refine the abstraction. The effect on the forest of subsequences is simply the duplication of all vertices corresponding to the split node.

The *slicing steps* also mirror those in Chapter 4, but have more complex interactions with the subsequence forest. For example, removing a transition in the abstraction not only leads to the removal of all corresponding branches in the forest, but also requires re-examination of branches that used to be redundant. Performing a source enlargement operation makes it necessary to keep track of prefixes by which the new initial node could be reached from the original nodes, and so on.

5.2 A Motivating Example

We first give an example for the subsequence-based refinement procedure.

Consider the system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$, where

- $\mathcal{V} = \{x, y\}$,
- $\mathit{init} \equiv y > 0$,
- $\mathcal{T} = \{\tau_1, \tau_2\}$ with
  - $w_{\tau_1} = a$, $\rho_{\tau_1} \equiv x + y > 0 \wedge x' = x - 3y \wedge y' = 3x + y$,
  - $w_{\tau_2} = b$, $\rho_{\tau_2} \equiv x' > x \wedge y' = y + x' - x$.

We want to verify that this system satisfies the invariant $|w|_{a\{b\}a\{b\}a} = 0$, i.e. that τ_1 cannot be taken three times in a row. The set of subsequences is $U = \{\epsilon, a\{b\}, a\{b\}a\{b\}, a\{b\}a\{b\}a\}$. The transformation matrices of our transitions and the invariant vector are

$$F_{\tau_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, F_{\tau_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \phi = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

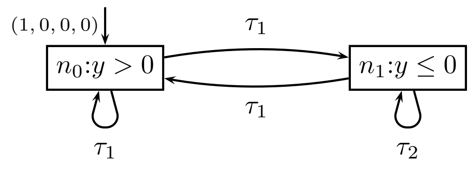
We begin with the initial abstraction:

$$\begin{tikzcd}
{} \arrow[d, "{(1,0,0,0)}"'] & \\
\boxed{n_0{:}y > 0} \arrow[r, bend left, "\mathcal{T}"] \arrow[loop below, "\mathcal{T}"] & \boxed{n_1{:}y \leq 0} \arrow[l, bend left, "\mathcal{T}"] \arrow[loop below, "\mathcal{T}"]
\end{tikzcd}$$

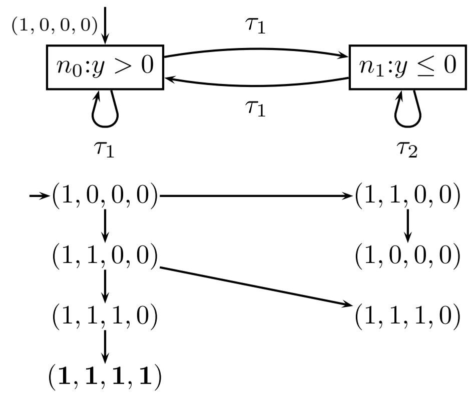
Step 1. We delete the inconsistent transitions $n_0 \overset{\tau_2}{\to} n_1$ and $n_1 \overset{\tau_1}{\to} n_1$.

$$\begin{tikzcd}
{} \arrow[d, "{(1,0,0,0)}"'] & \\
\boxed{n_0{:}y > 0} \arrow[r, bend left, "\tau_1"] \arrow[loop below, "\mathcal{T}"] & \boxed{n_1{:}y \leq 0} \arrow[l, bend left, "\mathcal{T}"] \arrow[loop below, "\tau_2"]
\end{tikzcd}$$

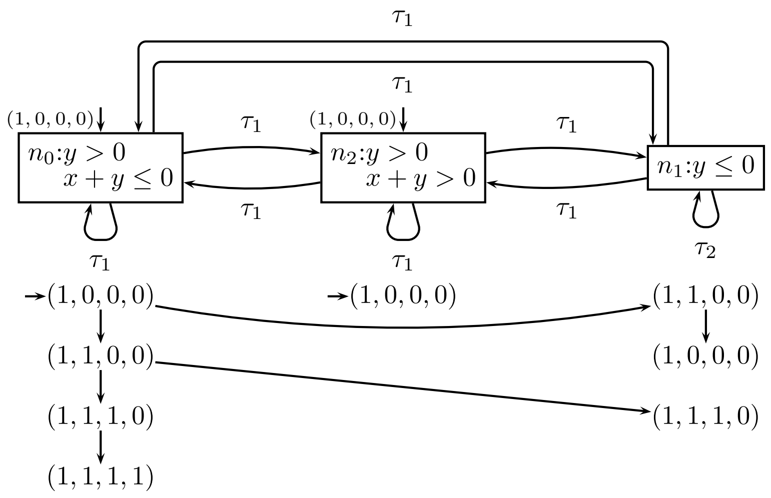
Step 2. We can also apply *Initiality Subsumption* to delete τ_2 on all edges leading to n_0, since F_{τ_2} maps any vector $\phi \in \mathbb{R}^U$ into the subspace spanned $(1, 0, 0, 0)$, which is the initial subspace of n_0.



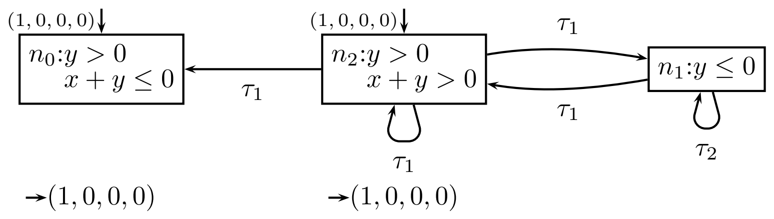
Step 3. We begin to explore the subsequence hulls of the abstraction, stopping when we encounter a violation of the invariant (i.e. a vector whose last component is nonzero). The path to this vertex corresponds to a counterexample in the abstraction, in this case $n_0 \xrightarrow{\tau_1} n_0 \xrightarrow{\tau_1} n_0 \xrightarrow{\tau_1} n_0$.



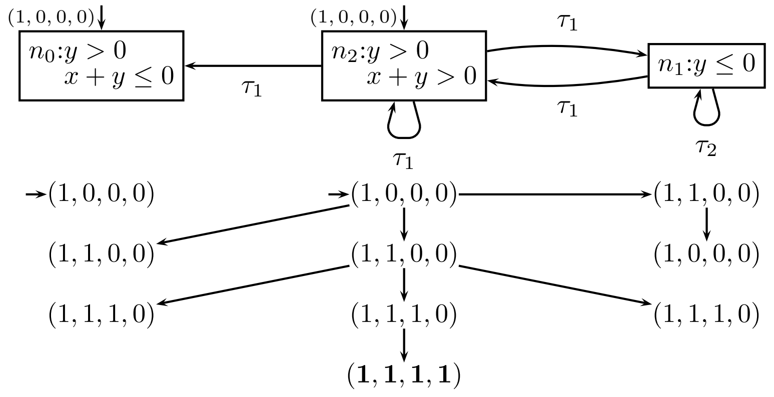
Step 4. From the spurious counterexample, we obtain the interpolant $x+y \le 0$. We use it to split node n_0, getting a new node n_2. The forest of subsequence images gets a new (trivial) tree rooted over n_2.



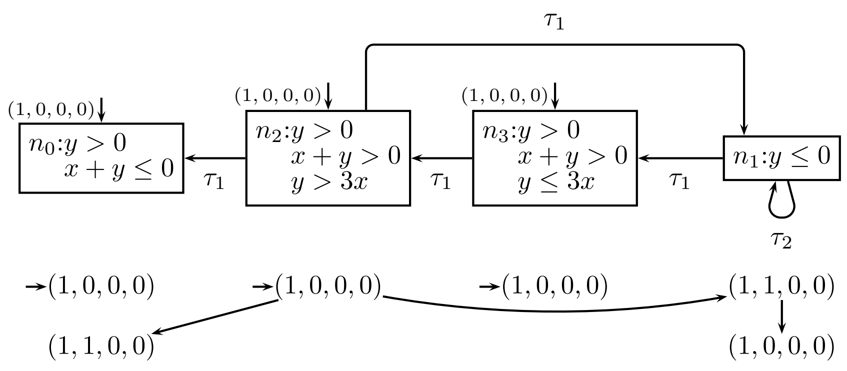
Step 5. Removing inconsistent transitions and empty edges, we also lose all subtrees in the forest which are only reachable through them, which in this case leaves us with only the roots.



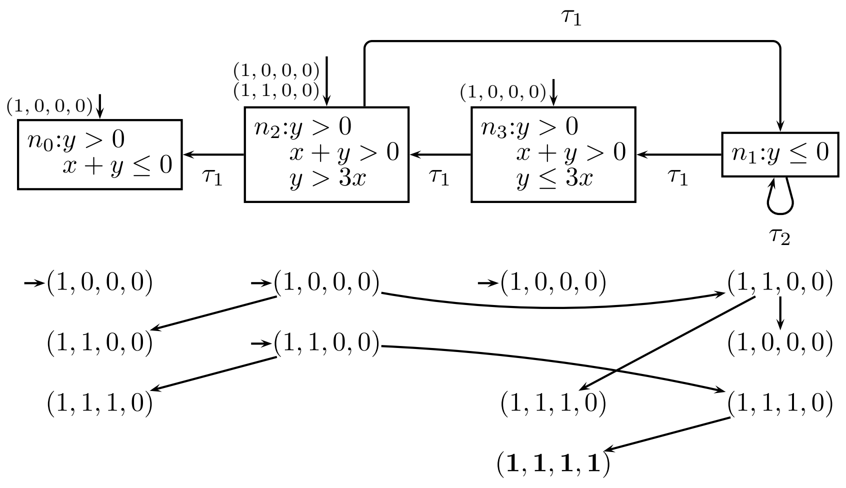
Step 6. We resume exploration of the subsequence forest, finding another counterexample.



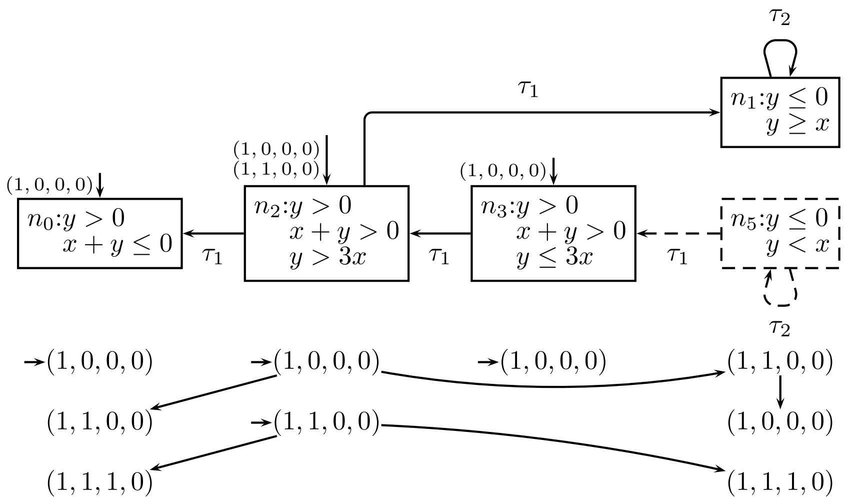
Step 7. After splitting with the new interpolant $y > 3x$ and slicing, we again lose part of the subsequence forest.



Step 8. The node label $\nu(n_2)$ implies the postcondition $\text{post}(\tau_1, \nu(n_3))$, allowing us to apply *source enlargement*. The result is a new basis vector $(1,1,0,0)$ of the initial subspace at n_2. The subsequence forest gets a corresponding root vertex, resulting in a shorter counterexample when the exploration is resumed.



Step 9. Splitting n_1 with the interpolant $y \geq x$ produces a new node n_5 which, after slicing, becomes unreachable. The exploration of the subsequence forest reaches a fixed point without encountering any more errors; the system is correct.



5.3 Preliminaries

As before, we work with a finite set $\mathcal{V}$ of state variables and a finite alphabet of events Σ. We also assume given a fixed finite and prefix-closed set of subsequences U, from which we obtain for each a the transformation matrices $F_a \in \mathbb{R}^{U \times U}$ with $|w.a|_U = F_a|w|_U$ for all w. For any labelled transition $\tau = (w_\tau, \rho_\tau) \in \mathbb{T}(\mathcal{V}, \Sigma)$, we have the associated transformation matrix $F_\tau = F_{a_n} \cdots F_{a_1}$, where $w_\tau = a_1 \dots a_n$.

The specification is given by a labelled transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$ and a set $\Phi = \{\phi^1, \dots, \phi^n\} \subseteq \mathbb{R}^U$ of subsequence invariants over U. We say $\mathcal{S}$ is correct with respect to Φ if the invariants in Φ hold on every run $s_0, \tau_1, s_1, \dots, \tau_n, s_n$

of $\mathcal{S}$, i.e. the word $w := w_{\tau_1} \dots w_{\tau_n}$ satisfies the equations $\phi \cdot |w|_U = 0$ for all $\phi \in \Phi$.

5.4 Abstraction

Our abstractions are labeled graphs as in Chapter 4, except that nodes are now additionally labeled with vectors representing possible event prefixes, and there is no set of error nodes.

Definition 5.1 *An* abstraction $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ *of a system* $\mathcal{S}$ *consists of the following components:*

- *a finite set* N *of nodes,*
- *a subset* $N^0 \subseteq N$ *of initial nodes,*
- *a set* $E \subseteq N \times N$ *of edges,*
- *a labeling* $\nu : N \to Asrt(\mathcal{V})$ *of nodes with assertions,*
- *a labeling* $\pi : N \to 2^{\mathbb{R}^U}$ *of nodes with finite sets of vectors, and*
- *a labeling* $\eta : E \to 2^{\mathbb{T}(\mathcal{V},\Sigma)}$ *of edges with finite sets of transitions.*

The only new component is the label π, which assigns to each node a basis of its *initial subspace*. For noninitial nodes, this is the trivial space $\{\mathbf{0}\}$; for initial n it is the subsequence hull of prefixes that can occur before starting in this node. Initially, the only such prefix is the empty word; however, the *Source Enlargement* transformation may introduce others.

To any path $p = n_0, \tau_1, n_1, \dots, n_k$ in this kind of abstraction we can associate its *image* $H(p) = F_{\tau_k} \cdots F_{\tau_1} span(\pi(n_0))$, the space of vectors reachable from the initial subspace at n_0 via p. The path is an *error path* if, for some invariant $\phi \in \Phi$, $\phi \cdot H(p) \neq \{0\}$.

Definition 5.2 *A* subsequence forest $\Theta = (V, V^0, R, \beta, \psi)$ *over an abstraction* $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ *consists of*

- *A finite set* V *of vertices,*
- *a subset* $V^0 \subseteq V$ *of initial vertices,*
- *a set* $R \subseteq V \times \mathbb{T} \times V$ *of edges,*
- *a function* $\beta : V \to N$ *assigning to each vertex its* base node*, and*
- *a function* $\psi : V \to \mathbb{R}^U$ *assigning to each vertex its* image.

Given a subsequence forest Θ*, we define for each node* $n \in N$ *the* fibre *of* Θ *over* n *to be* $\beta^{-1}(n) = \{v \in V : \beta(v) = n\}$*, and the* hull *at* n *to be* $H_\Theta(n) = span(\psi(\beta^{-1}(n))) = span(\{\psi(v) : \beta(v) = n\})$.

A subsequence forest Θ over $\mathcal{A}$ is *valid* iff

- (V, R) is a directed forest, with V^0 as its set of roots,

- for each $n \in N$, the vectors $\psi(v)$ for $v \in \beta^{-1}(n)$ are linearly independent,
- for each $v \in V^0$, $\beta(v) \in N^0$ and $\psi(v) \in \mathit{span}(\pi(\beta(v)))$, and
- for all $v, v' \in V$, if $v \xrightarrow{\tau} v'$, then $\beta(v) \xrightarrow{\tau} \beta(v')$ and $\psi(v') = F_\tau \psi(v)$.

Θ is *full* iff

- for each $n \in N^0$, $\mathit{span}(\psi(V^0 \cap \beta^{-1}(n))) = \mathit{span}(\pi(n))$, and
- For each $v \in V$ and $n \in N$, if $\beta(v) \xrightarrow{\tau} n$, then $F_\tau \psi(v) \in V_\Theta(n)$.

Validity of Θ guarantees that for all $n \in N$, $H_\Theta(n)$ contains only linear combinations of subsequence images of runs in $\mathcal{A}$. It also implies that there are no redundant vectors in the forest. If Θ is also full, we actually have $H_\Theta(n) = H(n)$, so that the abstraction is safe if Θ contains no vectors violating Φ:

Lemma 5.3 *Let $\Theta = (V, V^0, R, \beta, \psi)$ be a valid and full subsequence forest over an abstraction $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ such that $\phi \cdot \psi(v) = 0$ for all $\phi \in \Phi$ and $v \in V$. Then $\mathcal{A}$ contains no error paths.*

Proof: Let $p = n_0, \tau_1, n_1, \dots, \tau_k, n_k$ be a path in $\mathcal{A}$. We show by induction that for each i, the image $H(p^i)$ of the path $p^i = n_0, \tau_1, n_1, \dots, \tau_i, n_i$ is contained in $H_\Theta(n_i)$:

For $i = 0$, this follows from the first condition for fullness. For any vector $\eta \in H(p^{i+1})$, there is $\eta' \in H(p^i)$ such that $\eta = F_{\tau_{i+1}} \eta'$. By the induction hypothesis, $\eta' \in H_\Theta(n_i)$, i.e. η' can be written as a linear combination

$$\eta' = \sum_{v \in \beta^{-1}(n_i)} \lambda_v \psi(v),$$

and thus

$$\eta = F_{\tau_{i+1}} \eta' = \sum_{v \in \beta^{-1}(n_i)} \lambda_v F_{\tau_{i+1}} \psi(v),$$

which is in $H_\Theta(n_{i+1})$.

But this implies that $H(p) \subseteq H_\Theta(n_k)$, and therefore $\phi \cdot H(p) = 0$ for all $\phi \in \Phi$, so that p is not an error path.

□

5.4.1 Initial abstraction

Definition 5.4 *The* initial abstraction $\mathcal{A}_0 = (N, N^0, E, \nu, \pi, \eta)$ *of a transition system $\mathcal{S} = (V, \mathit{init}, \mathcal{T})$ and set $\Phi = \{\phi^1, \dots, \phi^n\}$ of invariants over the set U of subsequences consists of the following components:*

- $N = \{i, \bar{i}\}$,
- $N^0 = \{i\}$,
- $E = N \times N$,
- $\nu : i \mapsto \mathit{init}, \bar{i} \mapsto \neg \mathit{init}$,
- $\pi : i \mapsto \{|\epsilon|_U\}, \bar{i} \mapsto \emptyset$,

- *$\eta : e \mapsto \mathcal{T}$ for all $e \in E$.*

The initial abstraction is shown in Figure 5.1.

We use a similar notion of soundness as in Chapter 4: For a given set of subsequence invariants Φ over U, we merely require the abstraction to contain some concretizable error path iff the system is incorrect. This again allows us to remove, under the right circumstances, redundant parts of the abstraction.

Proposition 5.5 *The initial abstraction $\mathcal{A}_0$ of $\mathcal{S}$ is sound with respect to Φ.*

Proof: By definition, the concretization of an error path of $\mathcal{A}_0$ is a run $s_0, \tau_1, s_1, \dots, \tau_m, s_m$ of $\mathcal{S}$ such that $\phi \cdot |w_{\tau_1} \dots w_{\tau_m}|_U \neq 0$ for some invariant $\phi \in \Phi$, implying that the system is incorrect.

Suppose, on the other hand, that $\mathcal{S}$ is not correct, i.e., there exists a run $s_0, \tau_1, s_1, \tau_1, \dots, \tau_m, s_m$ of $\mathcal{S}$ violating one of the invariants. Since the initial abstraction has edges labeled with the full set of system transitions between each pair of nodes, it has a corresponding run $n_0, \tau_1, n_1, \dots, \tau_m, n_m$, where $n_j = i$ if $s_j \models \mathit{init}$ and $n_j = \bar{i}$ otherwise. This is an error path, since it features the same sequence of transitions as the run in $\mathcal{S}$. It is concretizable, since the states s_j by assumption form a satisfying assignment for the corresponding set of constraints. $\square$

Definition 5.6 *The* initial subsequence forest *of an abstraction $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ is $\Theta_0 = (V_0, V_0^0, R_0, \beta_0, \psi_0)$, where*

- *$V_0 = V_0^0$ contains one vertex $v(n, \phi)$ for each $n \in N^0$ and $\phi \in \pi(n)$,*
- *$R_0 = \emptyset$,*
- *$\beta_0(v(n, \phi)) = n$ for all n, ϕ, and*
- *$\psi_0(v(n, \phi)) = \phi$ for all n, ϕ.*

Proposition 5.7 *The initial subsequence forest of an abstraction is valid.*

Proof: Since Θ_0 contains exactly one initial vertex for each initial node n of $\mathcal{A}$ and each $\phi \in \pi(n)$, and no edges, we have that

- (V, R) is trivially a directed forest with root set V_0^0,
- $\beta(v) \in N^0$ and $\psi(v) \in \pi(\beta(v))$ for all v by definition,
- The condition that for every edge in Θ there is a corresponding edge in $\mathcal{A}$ holds vacuously, and
- for each n, the vectors $\psi(v)$ for those v with $\beta(v) = n$ are, by definition, exactly the elements of the basis $\pi(n)$ and thus linearly independent.

$\square$

5.4.2 Forest Exploration

The main change to the refinement loop itself is the exploration of the subsequence forest, which is interleaved with the refinement and slicing steps in the abstraction. We now describe the relevant details of these steps.

Throughout the refinement loop, we maintain three lists of vertices:

- the *open list* of vertices that we still have to evaluate,
- the *closed list* of vertices that have already been evaluated and make up the current set V of vertices in the subsequence forest Θ, and
- the *suspend list* of vertices that were pruned because the associated subsequence image was redundant. We need to keep these vertices because some slicing steps may actually remove the redundancy, making it necessary to add some of these vertices back to the open list.

Initially, the closed list contains the initial vertices V_0^0 of the initial subsequence forest, the open list contains all children of the $v \in V_0^0$, and the suspend list is empty. Here we assume that the vectors $\psi(v)$ for $v \in V_0^0$ do not violate any invariants; otherwise, the system is incorrect and we can terminate immediately.

In each iteration of the loop, as long as the open list is not empty, we

1. pick a vertex v from the open list and compute its image $\psi(v) = F_\tau \psi(u)$, based on its parent transition $u \xrightarrow{\tau} v$;
2. if the image is redundant, i.e. $\psi(v) \in H_\Theta(\beta(v))$, add v to the suspend list and go to (1);
3. if the image satisfies the invariants, i.e. $\phi \cdot \psi(v) = 0$ for all $\phi \in \Phi$, then add v to the closed list, and for each transition $\beta(v) \xrightarrow{\tau'} n$ in $\mathcal{A}$,add to the open list a child v' with $\beta(v') = n$ and $v \xrightarrow{\tau'} v'$. Go to (1);
4. otherwise, we have a counterexample. Let $u \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_n} v$ be the unique path to v with $u \in V^0$, check the corresponding run $\beta(u) \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_n} \beta(v)$ for concretizability, and either terminate or refine the abstraction.

The refinement and slicing operations are affected in two ways by the addition of subsequence invariants:

- We have to be more careful about what parts of the abstraction are actually safe to remove. The removal of transitions into initial nodes is one example: While in the original approach, this could only cut unnecessary prefixes off possible error paths, those prefixes may now actually be important, since what we actually need to keep track of are combinations of system states and synchronization histories.
- Transformations of the abstraction will have an impact on the subsequence forest as well. We need to reflect operations such as node split or removal of transitions in the forest. One approach would be to just start over with the exploration, but we try to retain as much as possible of the forest, in order to avoid unnecessary recomputations.

5.4.3 Abstraction Refinement

We first introduce the refinement step for a given predicate and node, and show that it maintains soundness of the abstraction and validity of the subsequence forest.

Given some new predicate q, we again split an abstract node n labeled φ into two new nodes, one labeled $\varphi \wedge q$, the other $\varphi \wedge \neg q$.

Given a subsequence forest Θ over $\mathcal{A}$, we duplicate all vertices v for which $\beta(v) = n$, along with the edges from their parent vertices. Note that we do not duplicate any of the children of these vertices, since their image is identical to that of the original. The copy of the child would therefore have both the same base node and image as the original child, making it redundant.

Node split Let $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ be an abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$, let $\Theta = (V, V^0, R, \beta, \psi)$ be a subsequence forest over $\mathcal{A}$, and let $n \in N$ be some node and $q(\mathcal{V})$ some predicate. Splitting node n with q results in the new abstraction $\mathcal{A}' = (N', N^{0\prime}, E', \nu', \pi', \eta')$ and the new subsequence forest $\Theta' = (V', V^{0\prime}, R', \beta', \psi')$ over $\mathcal{A}'$, where

- $N' = N \cup \{n^-\}$ where $n^- \notin N$ is a fresh node;
- $N^{0\prime} = \begin{cases} N^0 \cup \{n^-\} & \text{if } n \in N^0, \\ N^0 & \text{otherwise,} \end{cases}$
- $E' = \bigcup_{e \in E} \mathit{split}(e)$, where
$$\mathit{split}(e) = \begin{cases} \{e, (n, n^-), (n^-, n), (n^-, n^-)\} & \text{if } e = (n, n), \\ \{e, (m, n^-)\} & \text{if } e = (m, n), m \neq n, \\ \{e, (n^-, m)\} & \text{if } e = (n, m), m \neq n, \\ \{e\} & \text{otherwise,} \end{cases}$$
- $\nu'(m) = \begin{cases} \nu(n) \wedge q & \text{if } m = n \\ \nu(n) \wedge \neg q & \text{if } m = n^-, \text{ and} \\ \nu(m) & \text{otherwise,} \end{cases}$
- $\pi'(m) = \begin{cases} \pi(n) & \text{if } m = n^-, \\ \pi(m) & \text{otherwise,} \end{cases}$
- $\eta'(e') = \eta(e)$ for all $e' \in \mathit{split}(e), e \in E$,
- $V' = V \cup \{s(v) : v \in \beta^{-1}(n)\}$, where $s(v)$ is a fresh node for each $v \in \beta^{-1}(n)$,
- $V^{0\prime} = V^0 \cup \{s(v) : v \in (V^0 \cap \beta^{-1}(n))\}$,
- $R' = R \cup \{(v', \tau, s(v)) : v \in \beta^{-1}(n), (v', \tau, v) \in R)\})$
- $\beta'(v) = \begin{cases} n^- & \text{if } v \in V' \setminus V \\ \beta(v) & \text{otherwise} \end{cases}$
- $\psi'(v) = \begin{cases} \psi(v') & \text{if } v = s(v') \in V' \setminus V \\ \psi(v) & \text{otherwise} \end{cases}$

Proposition 5.8 *Let $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ be a sound abstraction of a transition system $\mathcal{S} = (\mathcal{V}, \mathit{init}, \mathcal{T})$, let $\Theta = (V, V^0, R, \beta, \psi)$ be a valid subsequence tree* *over $\mathcal{A}$, and let $n \in N$ be some abstract node and $q(\mathcal{V})$ be a predicate. Then the application of* node split *to $\mathcal{A}$ and Θ with respect to n and q produces another sound abstraction $\mathcal{A}'$ of $\mathcal{S}$, and a valid subsequence forest Θ' over $\mathcal{A}'$.*

Proof: If $\mathcal{S}$ is not correct, then the sound abstraction $\mathcal{A}$ contains an error path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ which has a concretization $s_0, \tau_0, s_1, \tau_1, \ldots, \tau_{k-1}, s_k$. In this case, the node split $\mathcal{A}'$ of $\mathcal{A}$ with respect to n and q contains the concretizable error path $n'_0, \tau_0, n'_1, \tau_1, \ldots, \tau_{k-1}, n'_k$ where, for all $0 \le i \le k$, $n'_i = n_i$ if $n_i \in N \smallsetminus \{n\}$, $n'_i = n$ if $n_i = n$ and $q(s_i)$, and $n'_i = n^-$ if $n_i = n$ and $\neg q(s_i)$.

Suppose, on the other hand, that $\mathcal{A}'$ contains a concretizable error path $n'_0, \tau'_0, n'_1, \tau'_1, \ldots, \tau'_{k-1}, n'_k$, then $\mathcal{A}$ contains the concretizable error path $n_0, \tau'_0, n_1, \tau'_1, \ldots, \tau'_{k-1}, n_k$ where $n_i = n'_i$ if $n'_i \in N$, and $n_i = n$ if $n'_i = n^-$, implying that $\mathcal{S}$ is not correct. Hence, $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$.

The only changes from Θ to Θ' are the addition of some vertices, each possibly with an edge from a parent vertex. More precisely, the possibilities for any new vertex $v' = s(v) \in V' \setminus V$ are:

1. $v \in V^0$ is an initial vertex. Then v' is also initial in Θ'. By validity of Θ, v has no parent, and by the definition of R', neither does v'. We therefore have added a new tree, consisting only of the root vertex $v' \in V^{0\prime}$.

   Also, since $v \in V^0$, $n = \beta(v) \in N^0$, and therefore $n^- = \beta(v') \in N^{0\prime}$. As $\pi'(n^-) = \pi(n)$, $\psi'(v') = \psi(v) \in span(\pi'(n-))$.

2. If $v \notin V^0$, then again by the validity of Θ, there is a unique edge $u \xrightarrow{\tau} v$ in R, and R' contains a matching new edge $u \xrightarrow{\tau} v'$. We thus have added a new branch to an existing tree.

   Since $u \xrightarrow{\tau} v$ and Θ is valid, $\beta(u) \xrightarrow{\tau} n$ in $\mathcal{A}$, and therefore $\beta(u) \xrightarrow{\tau} n^- = \beta(v')$ in $\mathcal{A}'$.

It only remains to show that the vectors $\psi'(v')$ for $v' \in \beta'^{-1}(n^-)$ are linearly independent. This holds because each such v' equals $s(v)$ for some $v \in \beta^{-1}(n)$, and $\psi'(v') = \psi(v)$ by definition, and Θ is valid.

□

5.5 Elimination Rules

We now show how to extend the elimination rules to the subsequence invariant case. Throughout this and the following section, we assume given a sound abstraction $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ and a valid subsequence tree $\Theta = (V, V^0, R, \beta, \psi)$ over $\mathcal{A}$ such that the vectors $\psi(v)$ satisfy all invariants. We will show that each transformation preserves soundness and validity.

The elimination rules all modify the abstraction by removing some nodes, transitions, or edges, i.e. we get a new abstraction $\mathcal{A}' = (N', N^0 \cap N', E', \nu|_{N'}, \pi|_{N'}, \eta')$ such that

- $N' \subseteq N$,
- $E' \subseteq E \cap (N' \times N')$, and
- $\eta'(e) \subseteq \eta(e)$ for all $e \in E'$.

The transformation of Θ associated to each elimination rule will either be trivial or consist of the removal of some subtrees. As long as this removal satisfies some ocvious conditions, the resulting subsequence forest is again valid for $\mathcal{A}'$:

Lemma 5.9 *Let $\mathcal{A}, \mathcal{A}', \Theta$ be as described above, and let $\Theta' = (V', V^0 \cap V', R \cap (V' \times V'), \beta|_{V'}, \psi|_{V'})$ be a subsequence forest. such that*

- *$V' \subseteq V$ is closed under the parent relation, i.e. if $v \notin V'$ and $(v, \tau, v') \in R$, then $v' \notin V'$,*
- *all vertices over removed nodes are also removed, i.e. for all $v \in V'$, $\beta(v) \in N'$, and*
- *all vertices reached by a transition $v \xrightarrow{\tau} v'$ such that the corresponding transition is removed in $\mathcal{A}'$ are also removed, i.e. if $v \xrightarrow{\tau} v'$ in Θ', then $\beta'(v) \xrightarrow{\tau} \beta'(v')$ in $\mathcal{A}'$.*

Then Θ' is a valid subsequence forest over $\mathcal{A}'$.

Proof:

- A tree in the forest from which a subtree is removed either becomes empty (in which case the root is removed both from V and from V^0), or is again a tree with the same root as before. This takes care of the first condition.
- The values of β and ψ are unchanged for the remaining vertices, implying the second condition.
- Also unchanged are the membership in N^0 and the bases $\pi(n)$ for the nodes $n \in N'$. As all vertices associated with removed nodes or transitions are also removed, this implies the last two conditions.

$\square$

5.5.1 Eliminating transitions

The effect of these rules on the abstraction is unchanged, and the effect on the subsequence forest is straightforward:

Inconsistent Transition If $\mathcal{A}$ contains a transition $\tau \in \eta(m, n)$ on some edge $(m, n) \in E$ that is *inconsistent* with the node labels, i.e., such that the formula $\nu(m) \wedge \rho_\tau \wedge \nu(n)'$ is unsatisfiable, we remove τ, resulting in the abstraction $\mathcal{A}' = (N, N^0, E, \nu, \pi, \eta')$, where $\eta'(m, n) = \eta(m, n) \setminus \{\tau\}$ and $\eta'(e) = \eta(e)$ for $e \neq (m, n)$.
We transform Θ into a subsequence forest over $\mathcal{A}'$ by pruning, for every edge $v \xrightarrow{\tau} v'$ in R with $\beta(v) = m$ and $\beta(v') = n$, the subtree rooted in v'.

Empty Edges Let $\mathcal{A}$ contain an edge $e \in E$ with $\eta(e) = \emptyset$. Any such edge can be removed, resulting in the abstraction $\mathcal{A}' = (N, N^0, E', \nu, \pi, \eta|_{E'})$, where $E' = \{e \in E \mid \eta(e) \neq \emptyset\}$.
The subsequence forest Θ remains unchanged.

Proposition 5.10 *Applying* Inconsistent Transition *or* Empty Edges *to $\mathcal{A}$ and Θ results in another sound abstraction $\mathcal{A}'$ of $\mathcal{S}$, and a valid subsequence forest Θ' over $\mathcal{A}'$.*

Proof: The soundness proof for $\mathcal{A}$ is the same as in chapter 4. Θ' is valid by lemma 5.9.

□

5.5.2 Eliminating nodes

Nodes are removed from the abstraction if they are either labeled with an inconsistent combination of predicates or do not occur on any error paths.

Inconsistent Node Let $\mathcal{A}$ contain a node $n \in N$ such that $\nu(n)$ is unsatisfiable. We remove n, resulting in the abstraction $\mathcal{A}' = (N', {N^0}', E', \nu|_{N'}, \pi|_{N'}, \eta|_{E'})$, where $N' = N \setminus \{n\}$, ${N^0}' = N^0 \setminus \{n\}$, and $E' = E \cap (N' \times N')$.
We transform Θ into $\Theta' = (V', V^0 \cap V', R \cap (V' \times V'), \beta|_{V'}, \psi|_{V'})$ by removing all subtrees rooted in vertices v with $\beta(v) = n$.

Unreachable Node Let $\mathcal{A}$ contain a node $n \in N$ which is unreachable from the initial nodes. We remove n, resulting in $\mathcal{A}' = (N', {N^0}', E', \nu|_{N'}, \pi|_{N'}, \eta|_{E'})$, where $N' = N \setminus \{n\}$,${N^0}' = N^0 \setminus \{n\}$, and $E' = E \cap (N' \times N')$.
We transform Θ into $\Theta' = (V', V^0 \cap V', R \cap (V' \times V'), \beta|_{V'}, \psi|_{V'})$ by removing all subtrees rooted in vertices v with $\beta(v) = n$.

Proposition 5.11 *Let $\mathcal{A}'$ and Θ' be the results of applying* Inconsistent Node *or* Unreachable Node*. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$, and Θ' is a valid subsequence forest over $\mathcal{A}'$.*

Proof:
Again, the soundness proof is the same as in the previous chapter, and Θ' is valid by lemma 5.9.

□

5.6 Simplification Rules

Unlike elimination rules, the simplification rules remove parts of the abstraction in a way that potentially reduces the concrete behaviors represented by the abstraction. As such, the arguments for their soundness are more subtle already for state-based invariants. This issue becomes even more complex for subsequence invariants. Nevertheless, there are corresponding transformations for the simplifications presented in Section 4.5.

In the following, we describe exactly how the simplification rules have to be adapted, and prove that these new versions preserve soundness of the abstraction and validity of the subsequence forest.

5.6.1 Simplifying transition relations

This transformation is actually unchanged from its version in Section 4.5. The reason for this is that what gets removed are not actually nodes or transitions, possibly corresponding to actually reachable concrete states and transitions, but just constraints which have been shown to be irrelevant for concretizability of any abstract runs.

Simplify Transition Let $\mathcal{A} = (N, N^0, E, \nu, \pi, \eta)$ be an abstraction and let $L(n) \subseteq \mathcal{V}$ indicate the set of live variables for each node n, as described in Section 4.5. The simplification $\mathit{simplify}(\tau, m, n)$ of a transition τ on an edge $(m, n) \in E$ is obtained by removing from ρ_τ all conjuncts ϕ with $\mathit{vars}(\phi) \cap (L(m) \cup L(n)') = \emptyset$. In the special case of a guarded W-assignment τ, the simplification $\mathit{simplify}(\tau, m, n)$ is obtained by removing from τ all conjuncts $v' = e_v(\mathcal{V})$ with $v \notin L(n)$.

Simplifying all transitions results in the new abstraction $\mathcal{A}' = (N, N^0, E, \nu, \pi, \eta')$ where $\eta'(m, n) = \{\mathit{simplify}(\tau, m, n) \mid \tau \in \eta(m, n)\}$ for all $(m, n) \in E$.

Θ remains unchanged by *Simplify Transition*.

Proposition 5.12 *Let $\mathcal{A}'$ be the result of applying* Simplify Transition. *Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$, and Θ is still a valid subsequence forest over $\mathcal{A}'$.*

Proof:
The proof of soundness is identical to the one in Chapter 4.3.2. Validity of Θ is obvious, since the only change in the abstraction occurs in the ρ_τ, which do not affect any of the validity conditions. □

5.6.2 Bypass transitions

In order to be able to apply this rule, we need to make sure that any violation that might occur at the node to be bypassed will be preserved. There are several ways in which this can be achieved. One option that is always possible is a version of *Bypass Transitions* that only removes the edges going out of the bypassed node n, not the node itself or its incoming transitions. The other possibilities are showing that either

- no error can occur at n, or
- any error that might occur at n will be preceded by an error at another node.

More formally, these conditions can be captured as follows:

Definition 5.13 *Let $B(n)$ be the set of nodes in N from which n is reachable. We call n* saturated *by Θ if*

- *for all $n' \in N^0 \cap B(n)$, $\pi(n') \subseteq H_\Theta(n')$, and*
- *for all $n' \in B(n)$ and all transitions $m \xrightarrow{\tau} n'$, $F_{w_\tau} H_\Theta(m) \subseteq H_\Theta(n')$.*

Lemma 5.14 *Let $n \in N$ be saturated by Θ. Then $H_\Theta(n') = H(n')$ for all $n' \in B(n)$.*

Proof: $H_\Theta(n') \subseteq H(n')$, because it is spanned by the images $H(p)$ of paths leading to n', and $H(n')$ is spanned by the images of all such paths.

Conversely, we show by induction on k that for any path $p = n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ with $n_k \in B(n)$, the image $H(p)$ is contained in $H_\Theta(n_k)$.

- For $k = 0$, $H(p) = \mathit{span}(\pi(n_0)) \subseteq H_\Theta(n_0)$ by the definition.
- For $k > 0$, let $p' = n_0, \tau_1, n_1, \ldots, \tau_{k-1}, n_{k-1}$; by the induction hypothesis, $H(p') \subseteq H_\Theta(n_{k-1})$. Then $H(p) = F_{\tau_k} H(p') \subseteq F_{\tau_k} H_\Theta(n_{k-1}) \subseteq H_\Theta(n_k)$.

Since $H(n')$ is spanned by the images of the paths leading to n', it follows that $H(n') \subseteq H_\Theta(n')$ for $n' \in B(n)$.

□

In particular, the subsequence forest being full corresponds to all $n \in N$ being saturated by it. A node n being saturated implies that any possible violation of an invariant in $B(n)$ would have been found already.

Definition 5.15 *An event $a \in \Sigma$ is* harmless *with respect to the set of invariants $\Phi \subseteq \mathbb{R}^U$ if the space $\Phi^\perp = \{\psi \in \mathbb{R}^U : \phi \cdot \psi = 0$ for all $\phi \in \Phi\}$ is closed under F_a. This implies that for any $w \in \Sigma$, if w satisfies the invariants, then so does $w.a$.*

Call a node avoidable *if $n \notin N^0$, and all incoming transitions $m \xrightarrow{\tau} n$ satisfy $w_\tau \in h(\Phi)^*$, where $h(\Phi)$ is the set of harmless events with respect to Φ.*

For such an avoidable node n we get that for any concretizable error path $n_0, \tau_1, n_1, \ldots, \tau_k, n_k = n$, the prefix $n_0, \tau_1, n_1, \ldots, \tau_{k-1}, n_{k-1}$ is already an (obviously also concretizable) error path. A node which is saturated or avoidable is *disposable* in the sense that we can actually delete it when it is bypassed, without affecting the existence of concretizable error paths.

For a node n with an incoming transition τ_1 and an outgoing transition τ_2, we define the bypass relation $\tau = \tau_1 \circ_n \tau_2$ by

- $w_\tau = w_{\tau_1} w_{\tau_2}$, and
- $\rho_\tau(\mathcal{V}, \mathcal{V}') = \exists \mathcal{V}'' \,.\, \rho_{\tau_1}(\mathcal{V}, \mathcal{V}'') \wedge \nu(n)(\mathcal{V}'') \wedge \rho_{\tau_2}(\mathcal{V}'', \mathcal{V}')$. If $W = L(n)$ is the set of live variables of n and τ_1 is a guarded W-assignment $\rho_{\tau_1}(\mathcal{V}, \mathcal{V}') = \bigwedge_i g_i(\mathcal{V}) \wedge \bigwedge_{v \in W}(v' = e_v(\mathcal{V}))$, then τ can be simplified to $\rho_\tau(\mathcal{V}, \mathcal{V}') = \bigwedge_i g_i(\mathcal{V}) \wedge \nu(n)[e_v/v](\mathcal{V}) \wedge \rho_{\tau_2}[e_v/v](\mathcal{V}, \mathcal{V}')$.

Bypass Transitions Let n be a node which is not initial, and does not have a self loop. Every pair $(\tau_1, \tau_2) \in \eta(m, n) \times \eta(n, n')$ of an incoming transition τ_1 and an outgoing transition τ_2 is modified to bypass node n, resulting in the new abstraction $\mathcal{A}' = (N', N^0, E', \nu, \pi, \eta')$, where

- $N' = \begin{cases} N \setminus \{n\} & \text{if } n \text{ is disposable} \\ N & \text{otherwise} \end{cases}$,
- $E' = \begin{cases} E \cap (N' \times N') \cup E_b & \text{if } n \text{ is disposable} \\ E \setminus \{n\} \times N \cup E_b & \text{otherwise} \end{cases}$,

  $E_b = \{(m, n') \mid (m, n), (n, n') \in E\}$, and
- $\eta'(m, n') = \eta(m, n') \cup \{\tau_1 \circ_n \tau_2 \mid \tau_1 \in \eta(m, n), \tau_2 \in \eta(n, n')\}$ for all $(m, n') \in E'$ (with $\eta(m, n') = \emptyset$ for $(m, n') \notin E$).

Since n is not initial, we know that any vertex $v \in V$ with $\beta(v) = n$ is not initial either, i.e., there is a parent vertex p and a transition τ_1 such that $p \xrightarrow{\tau_1} v$. For all transitions τ_2, we replace each edge $v \xrightarrow{\tau_2} w$ in R by an edge $p \xrightarrow{\tau} m$, where $\tau = \tau_1 \circ_n \tau_2$.

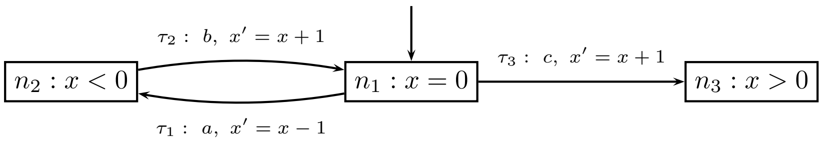

$\tau_2 :\ b,\ x' = x + 1$
$n_2 : x < 0$
$n_1 : x = 0$
$\tau_3 :\ c,\ x' = x + 1$
$n_3 : x > 0$
$\tau_1 :\ a,\ x' = x - 1$

Figure 5.3: Removing transitions into initial nodes may be unsound: The invariant $|w|_{ac} = 0$ does not hold for this abstraction, but would become true after removing τ_2.

Proposition 5.16 *Let $\mathcal{A}'$ and Θ be the results of applying* Bypass Transitions*. Then $\mathcal{A}'$ is a sound abstraction of $\mathcal{S}$, and Θ is a valid subsequence forest over $\mathcal{A}'$.*

Proof: We first show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. Assume that $n_0, \tau_1, n_1, \ldots, n_k$ is a concretizable error path in $\mathcal{A}$ and that node n is bypassed. Let I be the set of all $i \in \mathbb{N}$ such that $n_i = n$.

If $k \in I$, the error is obtained at the bypassed node, which cannot have been saturated. In that case, either n_k still exists, along with all incoming transitions, or n was avoidable, in which case we have another concretizable error path $n_0, \tau_1, n_1, \ldots, n_{k-1}$ in $\mathcal{A}$.

For all $i \in I \setminus \{k\}$, we can replace the steps $n_{i-1}, \tau_i, n_i, \tau_{i+1}, n_{i+1}$ in the error path by $n_{i-1}, \tau_i \circ_{n_i} \tau_{i+1}, n_{i+1}$ to obtain a concretizable error path in $\mathcal{A}'$.

Conversely, whenever $\mathcal{A}'$ has a concretizable error path containing a bypass transition τ, then there was an intermediate node n and two transitions τ_1, τ_2 such that $\tau = \tau_1 \circ_n \tau_2$ and thus a corresponding concretizable error path of $\mathcal{A}$ can be constructed.

The changes to Θ cannot introduce any cycles, since any new edge $p \xrightarrow{\tau} v'$, replaces a previously existing path $p \xrightarrow{\tau_1} v \xrightarrow{\tau_2} v'$, and thus Θ would have already contained a cycle. The set of roots V^0 is unchanged after *Bypass Transitions*, since n cannot have been initial.

The values of β and ψ also stay unchanged for all $v \in V'$, and the label of the bypass transition is the concatenation of those of the original transitions, so that the remaining validity conditions are still satisfied.

5.6.3 Initiality Subsumption

In the original Slicing Abstractions approach, it was sound to remove all transitions into initial states, since any error trace that used such a transition could be pruned, so that it started in the corresponding initial state instead. When the safety property involves subsequence counters, which contain information about the history of the system, matters are more complicated, since the prefix that leads to the initial state in question may be necessary for the trace to lead to an error at all. For example, the abstraction in figure 5.3 does not satisfy the invariant $|w|_{ac} = 0$: $n_1 \xrightarrow{\tau_1} n_2 \xrightarrow{\tau_2} n_1 \xrightarrow{\tau_3} n_3$ is a counterexample. Removing the transition τ_2 would lead to the invariant being satisfied, making the simplified abstraction unsound.

We can, however, construct conditions under which this problem cannot occur. Intuitively, if all subsequence images that can occur after a transition $m \xrightarrow{\tau} n$ are already contained in the initial subspace of n, i.e. $F_\tau \psi \in \mathit{span}(\pi(n))$ for all $\psi \in \mathbb{R}^U$, then it is safe to remove τ. This condition can be relaxed: First of all, it only really needs to hold for those ψ which do not already violate an invariant. Secondly, it is enough for all such $F_\tau \psi$ to be indistinguishable from vectors in $\mathit{span}(\pi(n))$ in the sense that

- For all $\phi \in \Phi$, $\phi \cdot F_\tau \psi$ is contained in $\phi \cdot \mathit{span}(\pi(n))$ and therefore 0 (we can assume that all initial vectors satisfy the invariants, otherwise we can terminate immediately), and

- for all transitions τ' leading out of the initial node, $F_{\tau'} F_\tau \psi$ is contained in $F_{\tau'} \mathit{span}(\pi(n))$.

Initiality Subsumption Let $\psi^1, \ldots, \psi^k$ be a basis of $\Phi^\perp$, the space of all vectors satisfying the invariants in Φ. If $n \in N^0$, and $m \xrightarrow{\tau} n$ such that for each ψ^j,

- for all $\phi \in \Phi$, $\phi \cdot F_\tau \psi^j = 0$, and
- for all $n \xrightarrow{\tau'} n'$, $F_{\tau'} F_\tau \psi^j \in F_{\tau'} \mathit{span}(\pi(n))$,

then we remove τ from $\nu(m, n)$, leading to the new abstraction $\mathcal{A}' = (N, N^0, E, \nu, \pi, \eta')$, where $\eta'(m, n) = \eta(m, n) \setminus \{\tau\}$, and $\eta'(e) = \eta(e)$ for $e \neq (m, n)$.
We transform Θ into a subsequence forest over $\mathcal{A}'$ by pruning, for every edge $v \xrightarrow{\tau} v'$ in R with $\beta(v) = m$ and $\beta(v') = n$, the subtree rooted in v'.

Proposition 5.17 *Applying* Initiality Subsumption *to remove $m \xrightarrow{\tau} n$ from $\mathcal{A}$ and Θ results in another sound abstraction $\mathcal{A}'$ of $\mathcal{S}$, and a valid subsequence forest Θ' over $\mathcal{A}'$.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. Obviously, every error path in $\mathcal{A}'$ is also an error path in $\mathcal{A}$. Conversely, suppose that $p = n_0, \tau_1, n_1, \ldots, n_k$ is a concretizable error path in $\mathcal{A}$. Without loss of generality, we assume that p is minimal, i.e. none of its proper prefixes is an error path. Suppose that transition τ_i has been eliminated, and i is the maximal such index. Then obviously $n_i = n$, and $i < k$ since by the conditions for *Initiality Subsumption*, τ_i cannot be the transition that leads to the error.

The path $n_i, \tau_{i+1}, n_{i+1}, \ldots, n_k$ is again an error path: By the second condition for *Initiality Subsumption*, the image $H(n_0, \tau_1, n_1, \ldots, n_i)$ is subsumed by the initial space at n, i.e. we have $F_{\tau_{i+1}} H(n_0, \tau_1, n_1, \ldots, n_i) \subseteq F_{\tau_{i+1}} \mathit{span}(\pi(n))$. This inclusion is preserved by the remaining transitions, so that $H(p) \subseteq H(n_i, \tau_{i+1}, n_{i+1}, \ldots, n_k)$, and in particular $n_i, \tau_{i+1}, n_{i+1}, \ldots, n_k$ is an error path. Being a subpath of p, it is obviously also concretizable.

Validity of Θ' again follows from lemma 5.9.

□

5.6.4 Source Enlargement

When applying this transformation, we have to keep track of the initial subspace associated to the node n which becomes initial. In order to do this, we need to be a little more restrictive than in Section 4.5.4, requiring all states associated with n to be reachable from *one* initial node m via *one* transition τ. We then add the image of $\pi(m)$ under F_τ to $\pi(n)$. Note that, since there are transitions into initial nodes, n may have already been initial, and $\pi(n)$ nonempty.

We again use the standard *strongest postcondition* operator $\mathrm{post}(\tau, \phi)(V') \Leftrightarrow \exists V(\tau(V, V') \wedge \phi(V))$.

Source Enlargement Let $\mathcal{A}$ contain a node $n \in N$ such that for some $m \in N^0$ and $m \xrightarrow{\tau} n$, $\nu(n) \Rightarrow \mathrm{post}(\tau, \nu(m))$. We add n to N^0, obtaining $\mathcal{A}' = (N, N^0 \cup \{n\}, E, \nu, \pi', \eta)$, where $\pi'(n) = \mathit{join}(F_\tau \pi(m), \pi(n))$, and $\pi'(n') = \pi(n')$ for $n' \neq n$.
From Θ, we obtain a subsequence forest Θ' over $\mathcal{A}'$ by removing all subtrees rooted in vertices v with $\beta(v) = n$, and adding new initial vertices $v(n, \psi) \in V^0$ for each basis element $\psi \in \pi'(n)$.

A node that satisfies the conditions of *Source Enlargement* can again be obtained by splitting a successor n of an initial node m with the strongest postcondition of a transition connecting m with n.

Proposition 5.18 *Let $\mathcal{A}'$ and Θ' be the results of applying* Source Enlargement *to a node $n \in N$, using a transition $m \xrightarrow{\tau} n$ from an initial node m. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$, and Θ' is a valid subsequence forest over $\mathcal{A}'$.*

Proof: We show that $\mathcal{A}$ contains a concretizable error path iff $\mathcal{A}'$ contains a concretizable error path. Note first that the sets of nodes and transitions remain unchanged after *Source Enlargement*, so that any path that exists in $\mathcal{A}$ also exists in $\mathcal{A}'$ and vice versa. In particular, any concretizable error path in $\mathcal{A}$ still exists in $\mathcal{A}$.

For the implication from $\mathcal{A}'$ to $\mathcal{A}$, let $p = n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ be a concretizable error path in $\mathcal{A}'$ such that $n_0 = n$, and let $s_0, s_1, \ldots s_k$ be a concretization of p. The image $H'(p)$ of p in $\mathcal{A}'$ is

$$\begin{aligned} H'(p) =& F_{\tau'_k} \cdots F_{\tau'_1} \mathit{span}(\pi'(n)) \\ =& F_{\tau'_k} \cdots F_{\tau'_1} (\mathit{span}(\pi(n)) \oplus F_\tau \mathit{span}(\pi(m))) \\ =& H(p) \oplus F_{\tau'_k} \cdots F_{\tau'_1} F_\tau \mathit{span}(\pi(m)), \end{aligned}$$

where $H(p)$ is the image of p in $\mathcal{A}$. In particular, since $H'(p)$ contains a vector ψ with $\phi \cdot \psi \neq 0$ for some invariant ϕ, either

- $H(p)$ already contains such a vector, and p is an error path in $\mathcal{A}$, or
- the image $H(p')$ of the extended path $p' = m, \tau, n_0, \tau_1, n_1, \ldots, \tau_k, n_k$ contains such a vector, and p' is an error path in $\mathcal{A}$.

The transformation of Θ only deletes some subtrees and adds new initial vertices $v(n, \psi)$ for all $\psi \in \pi(n)$, which obviously results in a correctly shaped forest.

Since none of the labels except π change, the remaining validity conditions remain true. $\square$

5.6.5 Partial Order Reduction

Applying partial order reduction in a context where the order of events is important for the properties under consideration is obviously problematic. In the case where (the order of) some events is irrelevant, however, one can still apply this transformation. We restrict ourselves to atomic transitions.

The most obvious case where *Partial Order Reduction* can be applied to a transition $\tau \in \eta(m,m)$ is when (besides the usual condition on the transition relations), each subsequence u occurring in an invariant either does not contain the event w_τ, or does not contain w'_τ for any of the other transitions τ' out of m. What one really needs, though, is the condition that for any such τ', the effect of applying τ and τ' on the subsequence counters is independent of the order, i.e. that F_τ and $F_{\tau'}$ commute. We therefore extend the notion of unblocked transitions to the labeled case:

A transition τ is unblocked by a transition τ' if and only if

- ρ_τ is unblocked by $\rho_{\tau'}$ with respect to m, n, and
- $F_\tau F_{\tau'} = F_{\tau'} F_\tau$.

One additional issue is the possibility of an error path p ending at m. In this case, there may be occurrences of $m \xrightarrow{\tau} m$ in p which get postponed to the end of the path, where it is not sound to remove them, because they may contribute to the actual error. We eliminate this possibility by additionally requiring w_τ to be *harmless* (see Section 5.6.2 with respect to Φ.

Partial Order Reduction Let $\mathcal{A}$ contain a node $m \in N$ with $(m,m) \in E$ and an atomic transition $\tau_1 \in \eta(m,m)$. If

- w_{τ_1} is harmless, i.e. for any ψ such that $\phi \cdot \psi = 0$ for all invariants $\phi \in \Phi$, also $\phi \cdot F_{\tau_1}\psi = 0$ for all $\phi \in \Phi$,
- for all nodes $n \in N$ such that $(m,n) \in E$ and transitions $\tau_2 \in \eta(m,n)$,
  - F_{τ_1} commutes with F_{τ_2}, and
  - ρ_{τ_1} is unblocked by ρ_{τ_2}, i.e. $\nu(m) \wedge (\rho_{\tau_1} \circ_m \rho_{\tau_2}) \wedge \nu(n)' \Rightarrow \nu(m) \wedge (\rho_{\tau_2} \circ_n \rho_{\tau_1}) \wedge \nu(n)'$,

then τ_1 is removed from $\eta(m,m)$, resulting in $\mathcal{A}' = (N, N^0, E, \nu, \pi, \eta')$, where $\eta'(m,m) = \eta(m,m) \setminus \{\tau_1\}$ and $\eta'(e) = \eta(e)$ for $e \neq (m,m)$.

Θ is transformed into a subsequence forest over $\mathcal{A}'$ be removing, for each $v \xrightarrow{\tau_1} v'$ with $\beta(v) = \beta(v') = m$, the subtree rooted in v'.

Proposition 5.19 *Let $\mathcal{A}'$ and Θ' be the results of applying* Partial Order Reduction *to $\tau \in \eta(n,n)$. Then $\mathcal{A}'$ is also a sound abstraction of $\mathcal{S}$, and Θ' is a valid subsequence forest over $\mathcal{A}'$.*

Proof: We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has a concretizable error path. The implication from $\mathcal{A}'$ to $\mathcal{A}$ is straightforward, since every path in $\mathcal{A}'$ also exists in $\mathcal{A}$.

For the reverse direction, we assume there exists a concretizable error path p in $\mathcal{A}$ and construct a concretizable error path p' in $\mathcal{A}'$. Let t be the number of occurrences of n, τ, n in p. We prove the claim by induction on t. If $t = 0$, then

$p' = p$. For $t > 0$, we construct a path p'' in $\mathcal{A}$ with $t-1$ occurrences of n, τ, n. Let $p = n_0, \tau_1, \ldots, \tau_k, n_k$ and let i be the least index such that $n_{i-1} = n_i = n$, $\tau_i = \tau$. If $n_0, \tau_1, \ldots, \tau_{i-1}, n_{i-1}$ is already an error path, we can use $p'' = n_0, \tau_1, \ldots, \tau_i, n_i$.

Otherwise, *Partial Order Reduction* guarantees that τ is unblocked by τ_{i+1}. Let j, $i < j \leq k$, be the largest index such that ρ_τ is unblocked by every transition in $\{\tau_{i+1}, \ldots, \tau_j\}$.

If $j = k$, we eliminate the occurrence of n, τ, n by first transforming p into

$$p''' = n_0, \tau_1, \ldots, \tau_{i-1}, n_{i-1}, \tau_{i+1}, n_{i+1}, \ldots, \tau_k, n_k, \tau, n_k.$$

Because F_τ commutes with $F_{\tau_{i+1}}, \ldots, F_{\tau_k}$, the image $H(p''')$ equals that of p, so that p''' is an error path. It is concretizable because τ is unblocked by $\tau_{i+1}, \ldots, \tau_k$ and p is concretizable. Because τ is harmless, however, the path $p'' = n_0, \tau_1, \ldots, \tau_{i-1}, n_{i-1}, \tau_{i+1}, n_{i+1}, \ldots, \tau_k, n_k$ must already be an error path. It is obviously also concretizable.

For the case that $j < k$, we show that $\mathcal{A}$ must contain a self loop in n_j that contains τ. Node n_j has been constructed in a series of node splits from one of the nodes n' in the initial abstraction, each of which has a self loop with $\tau \in \eta(n', n')$.

If τ is not in $\eta(n_j, n_j)$, it must have been removed by either *Inconsistent Transition* or by *Partial Order Reduction*. It is impossible that τ was removed by *Inconsistent Transition*, because this transformation requires $\nu(n_j) \wedge \tau \wedge \nu(n_j)'$ to be unsatisfiable; since the path $n, \tau_{i+1}, \ldots, \tau_j, n_j, \tau, n_j$ is concretizable, however, $\nu(n_j) \wedge \tau \wedge \nu(n_j)'$ is satisfiable. Likewise, it is impossible that τ was removed by *Partial Order Reduction*, because τ does not commute with τ_{j+1}. We use the self loop in n_j to transform p into the error path

$$p'' = n_0, \tau_1, \ldots, \tau_{i-1}, n_{i-1}, \tau_{i+1}, n_{i+1}, \ldots, \tau_j, n_j, \tau, n_j, \tau_{j+1}, \ldots, n_{k+1}.$$

Again p'' is an error path because its image $H(p'')$ coincides with $H(p)$, and is concretizable because τ is unblocked by $\tau_{i+1}, \ldots, \tau_j$. Since τ_{j+1} does not commute with τ, n_j must be different from n. The path p thus has one more occurrence of n, τ, n than p''. □

5.7 Experiments

We implemented our approach in a variant of the current version of SLAB, and compared its performance on a number of examples to both the standard version of SLAB 2 and ARMC, which were given a version of the examples with a state-based error condition, in which the subsequence counters were added to the state variables.

The main focus was therefore on checking whether the specialized treatment of subsequences would indeed lead to the expected boost compared to the translation-based approach.

The results in Table 5.1 indicate that this is the case. The benchmarks are a version of the readers-writers problem (RW) and a message buffer (Buffer).

RW. The *RW* benchmark represents a solution to the readers-writers problem with writers preference. The system keeps track of the current numbers of read and write requests, and uses internal budget and mode variables for representing

	SUBSLAB		SLAB 2		ARMC
specification	iterations	time (s)	iterations	time (s)	time (s)
RW 2	29	0.70	78	4.57	4.36
RW 3	37	0.85	91	5.11	7.16
RW 4	41	0.99	103	6.00	11.46
RW 5	45	1.08	116	6.98	17.30
RW 6	48	1.25	128	8.07	29.32
RW 7	52	1.41	140	9.69	44.10
Buffer 1	17	0.42	127	1.72	9.90
Buffer 2	85	4.28	345	22.38	70.92
Buffer 3	134	10.38	timeout		909.86

Table 5.1: Experimental results for subsequence-aware SLAB (SUBSLAB) vs. standard SLAB 2 and ARMC: number of iterations of the refinement loop and running times in seconds on the benchmarks RW (with bounds $2, \ldots, 7$ on the number of concurrent read accesses) and Buffer (with bound $1, \ldots, 3$ on the height of the internal stacks).

the number of current read and write accesses (bounded by the parameter value and 1, respectively, and mutually exclusive).

The invariants we verify are

- $|w|_{ra.wa} = |w|_{rr.wa}$,
- $|w|_{wa.wa} = |w|_{wr.wa}$,
- $|w|_{wa.ra} = |w|_{wr.ra}$,

where ra, rr, wa, wr are the labels of the read access, read release, write access, and write release transitions, respectively. We thus show that nobody is currently using the resource when write access is granted, and nobody is writing to it when read access is granted.

Buffer. The *Buffer* benchmark represents a message buffer implemented using two stacks, each represented by a pair of integers, such that pushing and popping is encoded into arithmetic operations. A received signal is pushed onto the first stack. At any time, the buffer may flush its contents by suspending input, moving the contents of the first stack to the second stack (in reverse order of receiving it) and then outputting the contents of the second stack. When it is empty, input is again enabled. The parameter is the number of signals the buffer can accept before it needs to flush its contents.

The invariants we verify are:

- $|w|_{ra.ra.ok} = |w|_{oa.oa.ok}$,
- $|w|_{rb.ra.ok} = |w|_{ob.oa.ok}$,
- $|w|_{ra.rb.ok} = |w|_{oa.ob.ok}$,
- $|w|_{rb.rb.ok} = |w|_{ob.ob.ok}$,

where ra, rb, oa, ob, ok are the labels of the transitions receiving an a, receiving a b, outputting an a, outputting a b, and signifying the end of the flushing procedure, respectively. These invariants specify that when the buffer is empty, the output stream contains the same number of aa, ab, ba, and bb subsequences as the input stream.

Chapter 6

Conclusions

6.1 Summary

The main results of this thesis are:

- the introduction of *subsequence invariants*,
- two efficient invariant generation algorithms, which compute the set of subsequence invariants for a given finite-state process and set of subsequences,
- an analysis of combinatorial and algebraic properties of subsequence invariants, which imply several ways of obtaining additional invariants, and
- an abstraction refinement procedure for the verification of subsequence invariants of infinite-state systems.

Subsequence invariants. In chapter 2, we have introduced subsequence invariants, which specify systems in terms of occurrences of sequences of synchronization events. This allows us to completely avoid referring to the system state, and the invariants we obtain are *compositional*, i.e. an invariant derived for one process holds for the entire system as well.

Invariant generation. In chapter 3, we showed how to compute subsequence invariants. The first algorithm we presented is a simple fixpoint iteration using only linear algebra, and determines, for any finite-state process P and set of subsequences U, the set of all subsequence invariants in time linear in the size of P and cubic in the size of U. We also gave an optimized version for the subclass of pure subsequence invariants over strongly connected processes. We further showed that the invariant generation can be done *incrementally*, allowing for a gradual refinement by extending the set of subsequences.

Combinatorics and algebra. We showed that subsequence invariants satisfy very interesting combinatorial and algebraic properties. They allow, for example, us to replace any disjunction of subsequence invariants by a subsequence invariant, and to introduce additional, universally valid invariants beyond those obtained from the processes of a system.

Infinite-state systems. In order to handle infinite-state systems, we have also introduced the *Slicing Abstractions* (SLAB) approach, an abstraction refinement method augmented with slicing operations. This approach uses a graph-based abstraction which is gradually refined using local node splits. Besides producing very small abstractions, this makes SLAB the method of choice for verifying subsequence invariants on infinite-state systems, since it allows the tight integration of the invariant generation procedure into the refinement loop. The practicality of both the basic SLAB approach and the version for subsequence invariants has been shown on a number of benchmarks.

6.2 Future Work

There are several promising directions for future work. We will sketch a number of them.

More general subsequences. The concept of subsequences is related to the language-theoretic concepts of *marked products* and *products with counters* [74]. The Straubing-Thérien hierarchy of star-free languages [68, 69] suggests one way of generalizing phased subsequences, by counting occurrences of patterns such as $a\{b\{a\}c\}a$: two occurrences of a, with no occurrence of $b\{a\}c$ between them.

The questions, then, would be how to effectively count such occurrences, and what properties they satisfy. In particular, it would be interesting if such closure properties as given by Theorem 2.3 still hold for these more general patterns.

Inequalities. Properties given by linear inequalities over subsequence occurrences are a very promising extension. We have already shown in Section 3.2.3 that there are infinitely many such inequalities which hold for all w, even for pure subsequences. This suggests the question [66] whether it is possible to classify these tautological inequalities.

For verification purposes, an interesting approach would be to investigate suitable invariant generation algorithms. The most promising approach would be based on augmenting existing methods for linear transition systems [23, 20] with knowledge about the behavior of subsequence occurrences.

Algebraic topology. The graph-based results in sections 3.2.3 and 3.3 are related to some applications of topological methods in combinatorics: The general theme is to construct some kind of nice topological space (such as a cell complex) from combinatorical structures, and then infer properties of these structures from standard topological invariants of the spaces, such as their cohomology. Using this approach in the context of subsequence invariants seems quite promising.

Compositional abstraction refinement. The refinement procedure in Chapter 5 was based on the assumption that we are given a set of invariants for each individual process, which imply the desired system invariant and can be verified in isolation. There have been very interesting results [41, 51] on compositional approaches for abstraction refinement, in which the abstractions of the processes in a system are refined in parallel in order to prove a global property. It would be interesting to adapt these methods for the verification of subsequence invariants.

Conversely, subsequence invariants are very promising candidates for information to be shared between the process abstractions, since their correctness is independent of the rest of the system, and combining them only requires cheap linear arithmetic.

Bibliography

[1] M. Akian, S. Gaubert, and A. Guterman. Linear independence over tropical semirings and beyond, 2009.

[2] C. Baier, N. Bertrand, and M. Größer. Probabilistic acceptors for languages over infinite words. In *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin / Heidelberg, 2009.

[3] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In K. Jensen and A. Podelski, editors, *Proc. TACAS '04*, volume 2988 of *LNCS*, pages 388–403. Springer-Verlag, 2004.

[4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *Proc. TACAS '01*, pages 268–283. Springer-Verlag, 2001.

[5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. SPIN '01*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[6] J. Baumgartner, A. Kuehlmann, and J. A. Abraham. Property checking via structural analysis. In E. Brinksma and K. G. Larsen, editors, *Proc. CAV '00*, volume 2404 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2002.

[7] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[8] N. S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Comput. Sci.*, 173(1):49–87, Feb. 1997.

[9] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 307–327. Springer, 1999.

[10] M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[11] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In F. Arbab and M. Sirjani, editors, *Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN)*, 2007.

[12] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundamenta Informaticae*, 89(4):369–392, 2008.

[13] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. TheMathSAT 4 SMT solver. In *CAV*, volume 5123 of *LNCS*, pages 299–303. Springer, 2008.

[14] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.

[15] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[16] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 241–268. Springer, 2002.

[17] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, pages 154–169, London, UK, 2000. Springer-Verlag.

[19] E. M. Clarke, O. Grumberg, M. Talupur, and D. Wang. Making predicate abstraction efficient: How to eliminate redundant predicates. In W. A. H. Jr. and F. Somenzi, editors, *Proc. CAV '03*, volume 2725 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2003.

[20] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pages 420–432. Springer-Verlag, 2003.

[21] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A. J. Hu and M. Y. Vardi, editors, *Proc. CAV'98*, volume 1427 of *LNCS*, pages 293–304. Springer-Verlag, July 1998.

[22] C. Cortes and M. Mohri. Context-free recognition with weighted automata. *Grammars*, 3(2-3):133–150, May 2000.

[23] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *Proc. POPL*, pages 84–97, Jan. 1978.

[24] V. N. Şerbănuţă and T. F. Şerbănuţă. Injectivity of the Parikh matrix mappings revisited. *Fundam. Inf.*, 73(1,2):265–283, 2006.

[25] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proc. FMCAD'02*. Springer-Verlag, November 2002.

[26] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Detecting errors before reaching them. In E. A. Emerson and A. P. Sistla, editors, *Proc. CAV '00*, volume 1855 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2000.

[27] K. Dräger and B. Finkbeiner. Subsequence invariants. In F. van Breugel and M. Chechik, editors, *Proceedings of the 19th International Conference on Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*, pages 172–168, Berlin Heidelberg, 2008. Springer-Verlag.

[28] K. Dräger and B. Finkbeiner. Subsequence invariants. Reports of SFB/TR 14 AVACS 42, SFB/TR 14 AVACS, June 2008. ISSN: 1860-9821, http://www.avacs.org.

[29] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[30] M. Droste, W. Kuich, and H. Voglerd, editors. *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2009.

[31] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 73–89. Springer, 2006.

[32] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Backward Conditioning: A New Program Specialisation Technique and Its Application to Program Comprehension. In *IWPC*, pages 89–97. IEEE Computer Society, 2001.

[33] D. Geist. The PSL/Sugar specification language. a language for all seasons. In *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[34] S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. *IEEE transactions on Software Engineering*, 1(1):68–75, Mar. 1975.

[35] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. PhD thesis, Université de Liege, 1994.

[36] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.

[37] V. Halava and T. Harju. Undecidability in integer weighted finite automata. *Fundam. Inf.*, 38(1-2):189–200, 1999.

[38] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[39] T. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Lazy abstraction. In *Proc. POPL'02*, pages 58–70. ACM Press, 2002.

[40] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL '04*, pages 232–244, New York, NY, USA, 2004. ACM Press.

[41] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *In: CAV*, pages 262–274. Springer, 2003.

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proc. SPIN '03*, volume 1427 of *LNCS*, pages 235–239. Springer-Verlag, 2003.

[43] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[44] G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *Proc. FORTE '94*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.

[45] H. Hong, I. Lee, and O. Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM*, pages 25–34. IEEE Computer Society, 2005.

[46] R. Jhala and R. Majumdar. Path slicing. In *Proc. PLDI '05*, pages 38–47, New York, NY, USA, 2005. ACM Press.

[47] Z. Jiang, B. E. Litow, and O. Y. d. Vel. Similarity enrichment in image compression through weighted finite automata. In *COCOON '00: Proceedings of the 6th Annual International Conference on Computing and Combinatorics*, pages 447–456, London, UK, 2000. Springer-Verlag.

[48] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[49] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):435–455, 1974.

[50] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *J. Mach. Learn. Res.*, 2:419–444, 2002.

[51] A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs.* PhD thesis, Universität Freiburg, 2010.

[52] Z. Manna and A. Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, Apr. 1996.

[53] A. Mateescu, A. Salomaa, and S. Yu. Subword histories and Parikh matrices. *J. Comput. Syst. Sci.*, 68(1):1–21, 2004.

[54] C. Mathissen. Weighted logics for nested words and algebraic formal power series. *CoRR*, abs/1001.2175, 2010.

[55] K. L. McMillan. Applications of Craig interpolants in model checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2005.

[56] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[57] L. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *Software Tools for Technology Transfer*, 2(4):343–349, 2000.

[58] M. Mohri, F. Pereira, O. Pereira, and M. Riley. Weighted automata in text and speech processing. In *In ECAI-96 Workshop*, pages 46–50. John Wiley and Sons, 1996.

[59] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[60] E.-R. Olderog. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship.* Cambridge University Press, 1991. Paperback Edition 2005.

[61] R. J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[62] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[63] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. PADL'07*, 2006.

[64] J. J. M. M. Rutten. Coinductive counting with weighted automata. *J. Autom. Lang. Comb.*, 8(2):319–352, 2003.

[65] J. Sakarovitch and I. Simon. Subwords. In M. Lothaire, editor, *Combinatorics on Words*, pages 105–144. Addison-Wesley, 1983.

[66] A. Salomaa and S. Yu. Subword conditions and subword histories. *Inf. Comput.*, 204(12):1741–1755, 2006.

[67] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, July 1999. Preliminary version appeared in *Proc. 8th Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

[68] H. Straubing. A generalization of the schützenberger product of finite monoids. *Theor. Comput. Sci.*, 13:137–150, 1981.

[69] D. Thérien. Classification of finite monoids: The language approach. *Theor. Comput. Sci.*, 14:195–208, 1981.

[70] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *Proc. TACAS*, LNCS 2031, pages 113–127. Springer-Verlag, Apr. 2001.

[71] N. S. V. Ganesh, H. Saidi. Slicing SAL. Technical report, SRI International, http://theory.stanford.edu/, 1999.

[72] S. Vasudevan, E. A. Emerson, and J. A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. *ENTCS*, 128(6):279–294, 2005.

[73] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[74] P. Weil. On varieties of languages closed under products with counter. In *Mathematical Foundations of Computer Science 1989*, volume 379 of *Lecture Notes in Computer Science*, pages 534–544. Springer Berlin / Heidelberg, 1989.

[75] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[76] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Design Automation Conference*, pages 599–604, 1998.

[77] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. DAC '98*, pages 599–604, 1998.

[78] J. Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *Lecture Notes in Computer Science*. Springer, 1989.