Analysis of Dynamic Message Passing Programs

(A framework for the analysis of depth-bounded systems)

PhD Thesis

Author:
Damien Zufferey

Supervisor:
Thomas A. Henzinger, IST Austria, Klosterneuburg, Austria

Thesis Committee:
Viktor Kuncak, EPFL, Lausanne, Switzerland.
Rupak Majumdar, MPI-SWS, Kaiserslautern, Germany.

Program Chair:
Herbert Edelsbrunner, IST Austria, Klosterneuburg, Austria

A Thesis presented to the faculty of the Graduate School of the Institute of Science and Technology Austria, Klosterneuburg, Austria, in partial fulfillment of the requirements for the degree Doctor of Philosophy (PhD).

Institute of Science and Technology

September 5, 2013

I hereby declare that this dissertation is my own work along with stated collaborators, and it does not contain other peoples' work without this being so stated; and that the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

Author:

Damien Zufferey

Supervisor:

Thomas A. Henzinger

Thesis Committee:

Viktor Kuncak

Rupak Majumdar

Program Chair:

Herbert Edelsbrunner

September 5, 2013

Abstract

Motivated by the analysis of highly dynamic message-passing systems, i.e. unbounded thread creation, mobility, etc. We present a framework for the analysis of depth-bounded systems. Depth-bounded systems are one of the most expressive known fragment of the π-calculus for which interesting verification problems are still decidable. Even though they are infinite state systems depth-bounded systems are well-structured, thus can be analyzed algorithmically. We give an interpretation of depth-bounded systems as graph-rewriting systems. This gives more flexibility and ease of use to apply depth-bounded systems to other type of systems like shared memory concurrency.

First, we develop an adequate domain of limits for depth-bounded systems, a prerequisite for the effective representation of downward-closed sets. Downward-closed sets are needed by forward saturation-based algorithms to represent potentially infinite sets of states. Then, we present an abstract interpretation framework to compute the covering set of well-structured transition systems Because, in general, the covering set is not computable, our abstraction over-approximates the actual covering set. Our abstraction captures the essence of acceleration based-algorithms while giving up enough precision to ensure convergence. We have implemented the analysis in the Picasso tool and show that it is accurate in practice. Finally, we build some further analyses like termination using the covering set as starting point.

Acknowledgments

Chapter 2, 3, and 4 are joint work with Thomas A. Henzinger and Thomas Wies. Chapter 2 was published in FoSSaCS 2010 as "Forward Analysis of Depth-Bounded Processes" [112]. Chapter 3 was published in VMCAI 2012 as "Ideal Abstractions for Well-Structured Transition Systems" [114]. Chapter 5.1 is joint work with Kshitij Bansal, Eric Koskinen, and Thomas Wies. It was published in TACAS 2013 as "Structural Counter Abstraction" [13]. The author's contribution in this part is mostly related to the implementation. The theory required to understand the method and its implementation is quickly recalled to make the thesis self-contained, but should not be considered as a contribution. For the details of the methods, we refer the reader to the original publication [13] and the corresponding technical report [14]. Chapter 5.2 is ongoing work with Shahram Esmaeilsabzali, Rupak Majumdar, and Thomas Wies.

I also would like to thank the people who supported over the past 4 years. My advisor Thomas A. Henzinger who gave me a lot of freedom to work on projects I was interested in. My collaborators, especially Thomas Wies with whom I worked since the beginning. The members of my thesis committee, Viktor Kuncak and Rupak Majumdar, who also agreed to advise me. Simon Aeschbacher, Pavol Cerny, Cezara Dragoi, Arjun Radhakrishna, my family, friends and colleagues who created an enjoyable environment.

This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

Contents

List of Figures

List of Tables

Chapter 1

Introduction, motivation, and related work

1.1 Motivation

Following the evolution of processors toward many-cores architectures and ubiquity of connected devices, applications become more and more parallel and distributed. In that context we looked at the programming paradigms which promised to ease the task of programming such applications.

When we started exploring this field we were still at EPFL and the SCALA language was one of our sources of inspiration. The development of the SCALA actor library and its adoption by companies[1] caught our interest. The actor model [65, 28, 6] can be seen as a generalization of the object-oriented programming where the objects have their own threads and executes in parallel. The synchronous method calls (and returns) become messages (and replies) exchanged asynchronously. The actor model implementations come as part of the language, e.g. ERLANG, or as libraries, e.g. SCALA. The actor paradigm has been both used to exploit the parallelism available on a single machine and to create distributed application. In fact, the only difference between a distributed and single node application is the start-up phase where distributed nodes are connected and actors spawned on the different machines. This makes the actor model particularly interesting since it tackles both challenges of scaling *up* and *out* uniformly.

When translating actor programs into a formal model we had our first contact with the π-calculus. The π-calculus provides a great flexibility and uniformity to model the system we were looking at. Also as a calculus, it is minimal and helps separate distinct concepts. In particular the notion of names is central and gives insight into the relation between mobility and process creation. How the two concepts interact and what makes those systems hard to analyze. Unfortunately, the π-calculus is a Turing-complete model of computation. Thus, we started to look for fragments with better decidability properties, i.e. at least simple reachability questions should be decidable. In the author's master thesis [115], we looked at systems without process creation. In that specific case,

[1] A list of companies using actor programs can be found at http://akka.io/

Source code:

```
class Ping(count: Int, pong: Actor)
    extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    loop {
      react {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            println("Ping: stop")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

π-calculus:

$$
\begin{aligned}
pi_0 &= \overline{\text{pong}_{Ping}}\langle \text{ping}_{Pong}\rangle | pi_1 \\
pi_1 &= \text{ping}_{Pong}().pi_2 \\
pi_2 &= pi_{2a} \oplus pi_{2b} \\
pi_{2a} &= \overline{\text{pong}_{Ping}}\langle \text{ping}_{Pong}\rangle | pi_1 \\
pi_{2b} &= \overline{\text{pong}_{Stop}}\langle\rangle | pi_3 \\
pi_3 &= 0
\end{aligned}
$$

Control-flow automaton:

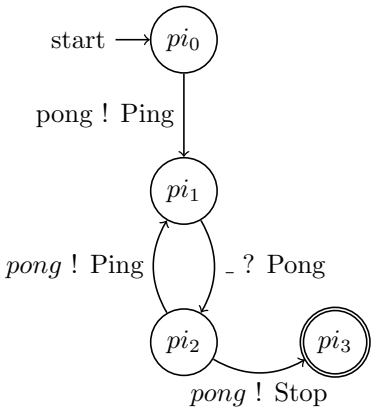


Figure 1.1: The `Ping` actor

we could use the results from [9] to reduce the actor programs to Petri nets and then check properties such as deadlock-freedom.

The *hello world* equivalent in the world of actors is the ping-pong example. In that example, two actors `Ping` and `Pong` exchange messages back and forth. In Figure 1.1 and 1.2, we show the source code of a ping-pong implementation in Scala[2] along which an encoding in the π-calculus and a control-flow automaton of each actor. The automata carry sends and receives on the edges with a CSP-like notation, i.e. `!` to send a message and `?` to receive one. Those two actors can be encoded into a Petri net as shown in Figure 1.3. The net is composed of two 1-bounded parts that corresponds to the control-flow automaton of each actor and of *mailbox* places which stores the messages between the send and receives. The red part of the net corresponds to the `Ping` actor and the blue part to the `Pong` actor. More details about this example can be found in [115].

The next step, which started this body of work, was to lift the restriction on process creation and look for more expressive fragment that would accept some (limited) form of dynamic process creation.

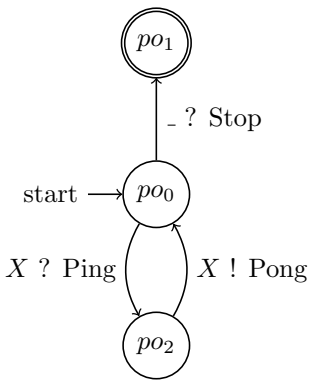
As source of examples, we looked at the lift web framework [78]. On the server side, actors are used to encapsulate the client's sessions. A simple be

[2]original source code available at http://www.scala-lang.org/node/54, retrieved on 23rd of April 2013.

Source code:

```
class Pong extends Actor {
  def act() {
    var pongCount = 0
    loop {
      react {
        case Ping =>
          if (pongCount % 1000 == 0)
            println("Pong: ping "+pongCount)
          sender ! Pong
          pongCount += 1
        case Stop =>
          println("Pong: stop")
          exit()
      }
    }
  }
}
```

Control-flow automaton:



π-calculus:

$$
\begin{aligned}
po_0 &= \text{pong}_{Stop}().po_1 \\
&+ \text{pong}_{Ping}(X).po_2 \\
po_1 &= 0 \\
po_2 &= \overline{X}\langle\rangle|po_0
\end{aligned}
$$

Figure 1.2: The `Pong` actor

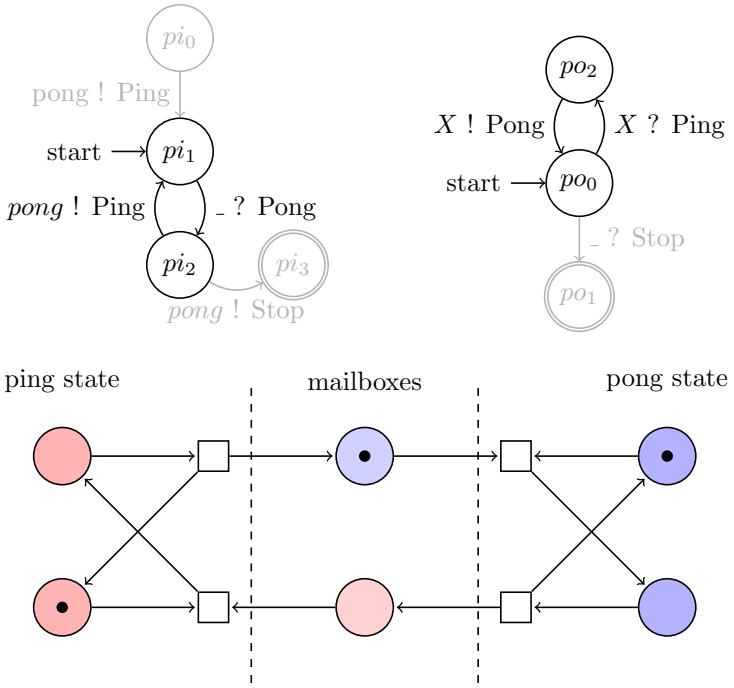


Figure 1.3: Petri net corresponding to a fragment of the ping-pong example

reasonably realistic example that draw our attention was the chat example[3]. In this example exhibit a very specific communication topology which looks like a star, i.e. all the communication goes through a central actor (the chat room) to which all the clients are connected. Our intuition was that those systems have similar monotonicity properties as the fragment without process creation but requires more expressive modeling than what Petri nets can accommodate. After some work we found an formal argument of decidability of such system, again in [115]. What we did not knew at the time is that this fragment had already been discovered one year earlier by Roland Meyer and was known as depth-bounded systems [80]. Yet there was still work to do in order to bring this theoretical result to complete analysis that could be implemented. This thesis will present (1) how we can finitely represent downward-closed sets for depth-bounded systems, (2) how to compute an over-approximation of the covering sets, (3) an implementation of the method in the Picasso tool, and (4) two further analyses that are built on top of the covering set: weak fair termination of depth-bounded systems and the computation of state-machine-like interface for group of interacting objects.

1.2 Preliminaries

We will recall some of the concepts used throughout this documents. Further general notions, along with some context, are presented in Section 1.3. Finally, more specific ideas are presented only in the parts that require them.

1.2.1 Posets, lattices, wqos, and bqos

Posets and lattices. A *quasi-ordering* $\leq$ is a reflexive and transitive relation $\leq$ on a set X. In the following $X(\leq)$ is a quasi-ordered set. The *upward closure* $\uparrow Y$ of a set $Y \subseteq X$ is $\uparrow Y = \{\, x \in X \mid \exists y \in Y.\, x \geq y \,\}$. The *downward closure* $\downarrow Y$ of Y is $\downarrow Y = \{\, x \in X \mid \exists y \in Y.\, x \leq y \,\}$. A set $Y \subseteq X$ is *upward-closed* if $Y = \uparrow Y$ and *downward-closed* if $Y = \downarrow Y$. An *upper bound* $x \in X$ of a set $Y \subseteq X$ is such that for all $y \in Y$, $y \leq x$. The notion of *lower bound* is defined dually. A nonempty set $D \subseteq X$ is called *directed* if any two elements in D have a common upper bound in D. A set $I \subseteq X$ is an *ideal* of X if I is downward-closed and directed. We denote by $Idl(X)$ the set of all ideals of X and call $Idl(X)$ the *ideal completion* of X.

If a quasi-ordering $\leq$ on a set X is antisymmetric it is called a *partial ordering* and $X(\leq)$ a *poset*. A poset $L(\leq)$ is called a *complete lattice* if every subset $X \subseteq L$ has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$ in L. In particular, L has a least element $\bot = \sqcap L$ and a greatest element $\top = \sqcup L$. This lattice will be denoted by $L(\leq, \top, \bot, \sqcup, \sqcap)$. For a function $f : X \to Y$ and $X' \subseteq X$ we denote by $f(X')$ the set $\{\, f(x) \mid x \in X' \,\}$. A monotone function $f : L \to L$ on a complete lattice $L(\leq, \top, \bot, \sqcup, \sqcap)$ is called *continuous* if for every directed subset D of L, $\sqcup f(D) = f(\sqcup D)$. Recall Kleene's fixed point theorem which states that if $f : L \to L$ is continuous then its least fixed point $\mathrm{lfp}^{\leq}(f) \in L$ exists and is given by $\sqcup \{\, f^i(\bot) \mid i \in \mathbb{N} \,\}$.

Let $L_1(\leq_1)$ and $L_2(\leq_2)$ be posets. A *Galois connection* between $L_1(\leq_1)$ and $L_2(\leq_2)$ is a pair of functions $\alpha : L_1 \to L_2$ and $\gamma : L_2 \to L_1$ that satisfy for all

[3] http://demo.liftweb.net/chat

$x \in L_1, y \in L_2$, $\alpha(x) \leq_2 y$ iff $x \leq_1 \gamma(y)$. If γ is also injective then (α, γ) is called *Galois insertion*.

Well-quasi-orderings and better-quasi-orderings.

Definition 1 (Well-quasi-order) *A pair $(X, \leq)$ of a set X and a binary relation $\leq$ on X is called* well-quasi-ordered set *(wqo) if and only if (1) $\leq$ is a quasi-ordering (i.e., reflexive and transitive) and (2) any infinite sequence $x_0, x_1, x_2, \ldots$ of elements from X contains an increasing pair $x_i \leq x_j$ with $i < j$.*

We extend the ordering $\leq$ to an ordering $\leq^+$ on subsets of X as expected: for $Y_1, Y_2 \subseteq X$, we have $Y_1 \leq^+ Y_2$ iff for all $y_1 \in Y_1$ there exists $y_2 \in Y_2$ if $y_1 \leq y_2$. For $Y \subseteq X$ we call $Y' \subseteq X$ *large* in Y iff $Y \leq^+ Y'$. Conversely, we call Y' *small* in Y if $Y' \leq^+ Y$. A subset $Y \subseteq X$ of X is called *irreducible* if for any $Y_1, Y_2 \subseteq X$, $Y \leq^+ Y_1 \cup Y_2$ implies $Y \leq^+ Y_1$ or $Y \leq^+ Y_2$. Unless specified otherwise, lifting ordering on the powerset of set uses $\leq^+$. When the context is clear we will write $\leq$ instead of $\leq^+$.

Let $\leq$ be a quasi-ordering on a set X then define the quasi-ordering $\leq_1$ on subsets of X as follows: for $Y_1, Y_2 \subseteq X$, we have $Y_1 \leq_1 Y_2$ iff there exists an injection $\phi : Y_1 \to Y_2$ such that for all $y_1 \in Y_1$, $y_1 \leq \phi(y_1)$. We are interested in wqo sets $(X, \leq)$ whose powerset is again a wqo with respect to $\leq_1$. For this purpose we consider Nash-William's *better-quasi-orderings* [92]. Better-quasi-ordered (bqo) sets are particular well-behaved wqo sets. Unlike general well-quasi-orderings, bqo sets are closed under the powerset construction. The formal definition of better-quasi-orderings is rather technical and not required for understanding this thesis. We therefore refer to [92] for the actual definition. We only state the properties of bqo sets that we will use in our proofs.

Proposition 1 *Let $(X, \leq)$ be a bqo then*

1. *$(X, \leq)$ is a wqo,*
2. *$(2^X, \leq_1)$ is a bqo,*
3. *$(2^X, \leq^+)$ is a bqo,*
4. *every $Y \subseteq X$ is a bqo with respect to the restriction of $\leq$ to Y.*

Properties 1 and 2 are proved in [92]. Properties 3 follows from the fact that $\leq^+$ contains $\leq_1$. Property 4 immediately follows from the definition of bqo sets.

1.2.2 Transition systems

Definition 2 (Labeled transition system) *A* labeled transition system *is a tuple $\mathcal{T} = (S, s_0, Act, \longrightarrow)$ where S is a set of states, $s_0 \in S$ an initial state, Act a set of action labels, and $\longrightarrow \subseteq S \times Act \times S$ is a transition relation.*

Given a labeled transition system $\mathcal{T} = (S, s_0, Act, \longrightarrow)$ We define $s \xrightarrow{a} s'$ iff $(s, a, s') \in \longrightarrow$. For $A \subseteq Act$, we define $s \xrightarrow{A} s'$ iff $s \xrightarrow{a} s'$ for some $a \in A$. We further define the function $Pred_a$ that maps a set of states $X \subseteq S$ to the

set of its direct predecessors, and the function $Post_a$ that maps C to its direct successors for some $a \in A$:

$$Pred_a(X) \stackrel{\text{def}}{=} \left\{ s \in S \mid \exists s' \in X.\, s \xrightarrow{a} s' \right\}$$
$$Post_a(X) \stackrel{\text{def}}{=} \left\{ s' \in S \mid \exists s \in X.\, s \xrightarrow{a} s' \right\} \quad .$$

Definition 3 (Transition system) *A* transition system *is a tuple* $\mathcal{T} = (S, s_0, \longrightarrow)$ *which corresponds to the special case of a labeled transition system with an unique action:* $(S, s_0, \{unit\}, \longrightarrow)$.

For the sake of clarity, we omit the *unit* action when we work with simple transition systems.

Definition 4 (Reachability set) *The* reachability set *of a transition system* $\mathcal{T}$*, denoted* $\text{Reach}(\mathcal{T})$*, is defined by* $\text{Reach}(\mathcal{T}) = \text{lfp}^{\subseteq}(\lambda X.\{s_0\} \cup Post(X))$.

Definition 5 (Inductive invariant) *A set* $X \subseteq S$ *is called an* invariant *of* $\mathcal{T}$ *if* $\text{Reach}(\mathcal{T}) \subseteq X$*, and* X *is called* inductive *if* $Post(X) \subseteq X$.

1.2.3 Abstract interpretation

Abstract interpretation [32] is a framework for the sound approximation of the semantics of complex systems. The concrete states of the program are mapped by an abstraction function α $(S \mapsto A)$ to elements in A the set of abstract states. A concretization function γ $(A \mapsto 2^S)$ that maps an element of the abstract domain to elements of the concrete domain. We extends α and γ to set of inputs as expected, i.e. take the union or the join of the results. (α, γ) are required to form a Galois connection between the sets 2^S and A, i.e. for all $s \in S$ and $a \in A$, $\alpha(s) \leq a$ iff $s \in \gamma(a)$.

Now that we know how to go from concrete to abstract element we need to lift the transition function to $\rightarrow^{\#}$ $(A \mapsto A)$. Similarly, $Post$ is lifted to $Post^{\#}(\ldots)$. The correctness of the abstraction is captured by the following condition: $Post(\gamma(a)) \subseteq \gamma(Post^{\#}(a))$. We define the most precise $Post^{\#}$ as $\mathsf{post}^{\#}.S = \alpha \circ \mathsf{post} \circ \gamma$.

The set A of abstract states needs to be a complete lattice, i.e. for every a_1, a_2 the least upper bound $(\sqcup)$ and the greatest lower bound $(\sqcap)$ are defined. This property of A and the monotonicity of the function g will ensure the existence of a least fixed point for g (Knaster-Tarski theorem). The goal of abstract interpretation is to compute the least fixed point of $Post^{\#}$ that contains s_0. Another formulation is to say that we want to compute $g^* \stackrel{\text{def}}{=} \bigsqcup_{n \geq 0} (Post^{\#})^n(\alpha(s_0))$.

Even though the fixed point g^* is guaranteed to exist, it might take an infinite number of steps to compute it. To avoid this case, the widening operator (∇) is applied to the increasing sequence $g^1, g^2, \ldots$ to accelerate the convergence of the algorithm. ∇ fulfill the following conditions: (1) $C_1 \bigsqcup C_2 \subseteq C_2 \nabla C_2$, (2) for every infinite sequence $s_0, s_1, \ldots$ the sequence $C_0, C_1, \ldots$, where $C_0 = s_0, C_i = C_{i-1} \nabla s_i$, is not strictly increasing.

1.2.4 Graphs

We use a standard notation for (directed) graphs, denoted as tuples of the form (V, E), with $E \subseteq V \times V$. We define *(vertex) labeled graphs* over a set of labels VL as graphs with labels for each vertex and denote them as (V, E, ν) where $\nu : V \to VL$ is the *vertex-labeling function*. For the rest of the paper we fix VL, a finite set of labels and we denote by *Graphs* the set of all labeled graphs with labels VL. Also, unless explicitly stated otherwise, whenever we say graph, we refer to a labeled graph. A *partial graph homomorphism* h from a graph $G = (V, E)$ to $G' = (V', E')$ written $h : G \to G'$ is a partial mapping $h : V \rightharpoonup V'$ such that $(v, w) \in E$ implies $(h(v), h(w)) \in E'$. If h is total, it is simply called morphism. If it is bijective and its inverse is also a homomorphism, then it is called *isomorphism*. Two graphs G and G' for which an isomorphism exists are called *isomorphic*, which we denote by $G \cong G'$. For labeled graphs, we additionally require that (partial) homomorphisms respect the vertex labeling, i.e. $\nu'(h(v)) = \nu(v)$, for all nodes $v \in \mathrm{dom}(h)$. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$, written $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. For a set $V' \subseteq V$ of vertices of a graph $G = (V, E)$, we denote by $G[V'] = (V', E \cap V' \times V')$ the subgraph *induced* by V'. We further denote by $\preceq$ the quasi-ordering induced by subgraph isomorphisms, i.e., $G \preceq H$ iff G is isomorphic to a subgraph of H. We write $G \cong H$ if G and H are isomorphic.

1.3 Related work on the analysis of message-passing concurrency

Analysis of concurrent system is a problem that has been often studied. However, due to the complexity, it has never really been solved. On the other hand, many different analysis techniques have been developed for particular kinds of systems. After an overview of the π-calculus, we will review the closely related work using infinite-state model checking techniques to analysis concurrent programs. Then, we will presents some different approaches to the same problem, e.g. compositional verification. More specific related work will also be presented in the relevant chapters.

1.3.1 The π-Calculus

Informally, the π-calculus[88, 89] is considered to be the λ-calculus of message passing concurrency. It tries to be minimal, but keeps a great modeling power. It can model synchronous and asynchronous communication, changing network topologies, dynamic creation of processes, etc. A fundamental concept of the π-calculus is the notion of names which decouple processes from addresses. Names are very similar to channels, as processes can send and receive messages through them. However, processes can use many names and many processes can receive from the same name. This is different from the actor model in which only a single actor can receive from a channel. Furthermore, names are also first-class values that can be sent and created.

Syntax. In π-calculus, a process (formula) P is defined by the following grammar:

$$\begin{array}{rcll} P & ::= & x(y).P & \text{(input prefix)} \\ & | & \overline{x}\langle y\rangle.P & \text{(output prefix)} \\ & | & \sum_i a_i(b_i).P_i & \text{(external choice)} \\ & | & P \mid P & \text{(parallel composition)} \\ & | & !P & \text{(replication)} \\ & | & (\nu x)P & \text{(name creation)} \\ & | & 0 & \text{(unit process)} \end{array}$$

Remark 1 *The* internal choice *a construct which is generally used to encode nondeterministic branching is omitted in the above definition. However, it can be emulated using the following construct:*

$$A \oplus B \Leftrightarrow (\nu a)(\nu b)(\overline{a} \mid \overline{b} \mid a().b().A + b().a().B)$$

We consider systems of recursive equations in the polyadic π-calculus that have a specific normal form inspired by Amadio and Meyssonnier [9].

Assume a countable infinite set of names with typical elements x, y and a countable infinite set of process identifiers with typical elements A, B. We assume that each name and identifier has an associated *arity* in $\mathbb{N}$. We denote by $\vec{x}$ a (possibly empty) vector over names and denote by $[\vec{x}/\vec{y}]$ a substitution on names. Hereby, a prefix π is either an *input prefix* of the form $x(\vec{y})$ or an *output prefix* of the form $\overline{x}(\vec{y})$. All parameter vectors occurring in process terms must respect the arities of names and identifiers. We call the terms of the form $A(\vec{x})$ *threads*. We write Π in order to denote indexed parallel composition and Σ for indexed external choice. We allow both input and output prefixes below Σ and generalize the reduction rules accordingly. We further write $(\nu\vec{x})$ for $(\nu x_1)\ldots(\nu x_n)$ where $\vec{x} = x_1, \ldots, x_n$. An occurrence of a name x in a process term P is called *free* if it is not below a (νx) or an input prefix $y(x)$. We denote by $\mathsf{fn}(P)$ the set of all free occurring names in P. We say that P is *closed* if $\mathsf{fn}(P) = \emptyset$.

A *process* $\mathcal{T}$ is a pair $(I, \mathcal{E})$ where I is an *initial* configuration and $\mathcal{E}$ is a finite set of parametric equations $A(\vec{x}) = P$ such that (1) every process identifier in P and I is defined by exactly one equation in $\mathcal{E}$ and (2) $\mathsf{fn}(P) \subseteq \{\vec{x}\}$. We assume that each equation in $\mathcal{E}$ has the following form:

$$A(\vec{x}) = \sum_{i\in I} \pi_i.(\nu\vec{x}_i)(\prod_{j\in J_i} A_j(\vec{x}_j))$$

A *configuration* is a closed process term of the following normal form:

$$(\nu\vec{x})(\prod_{i\in I} A_i(\vec{x}_i))$$

Similar normal forms can be found in [9] and in [80] under the name of *anchored fragment*.

Operational semantics. Informally, threads are doing the computations in a π-calculus process. On the other hand, the *names* give the topology of the 'network' of threads. They do not perform computations, but are the support for the messages.
Here is an informal description of the types of formula in the grammar:

$$\overline{x}\langle z\rangle.P \mid x(y).Q \rightarrow P \mid Q[z/y]$$
$$\overline{a}\langle b\rangle.P \mid \sum_{i\in I} a_i(b_i).Q_i \quad \rightarrow \quad P \mid Q_x[b/b_x] \quad \text{where } a_x = a$$
$$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$$
$$P \rightarrow Q \Rightarrow (\nu x)P \rightarrow (\nu x)Q$$
$$P \equiv P' \wedge Q \equiv Q' \wedge \rightarrow Q \Rightarrow P' \rightarrow Q'$$

Figure 1.4: Reduction rules for π-calculus

$P \equiv Q$ if P and Q are equal up to renaming of bound names	
$P \mid Q \equiv Q \mid P$	(commutativity of $\mid$)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(associativity of $\mid$)
$P \mid 0 \equiv P$	(elimination of unit)
$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	(reordering of restricted names)
$(\nu x)0 \equiv 0$	(elimination of unused name)
$(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ if $x \notin \mathsf{fn}(P)$	(scope extrusion)

Figure 1.5: Axioms defining the structural congruence relation $\equiv$

- $x(y).P$ means that a name is received from x, bound to y, and the process continues as P.
- $\overline{x}\langle y\rangle.P$ means that the name y is send on x and the process continues as P.
- $\sum_i a_i(b_i).P_i$ receive one of the available messages or blocks when there is no message at all.
- $P_1 \mid P_2$ denotes the concurrent execution of P_1 and P_2.
- $!P$ means that P can be replicated as many time as needed.
- $(\nu x)P$ creates a new fresh name x in the scope of P.
- 0 is a process whose execution is finished.

Here is a *partial* overview of the evaluation of a π-calculus process. The evaluation mixes the application of reduction rules and congruence rules. The reduction rules are detailed in Figure 1.4 and the congruence rules in Figure 1.5. The first two reduction rules are the most important as they corresponds to the reception of a message. For more details, refer to [87, 88, 89].

The congruence relation can be extended to terms with replication by adding the axioms in Figure 1.6. The resulting congruence relation (which we also denote by $\equiv$) corresponds to the *extended congruence relation* studied in [43], where it is also shown to be decidable. Note that ! distributes over parallel composition but not restriction.

Figure 1.1 and 1.2 shows Scala actors abstracted to the π-calculus.

$$!P \equiv P \mid !P \qquad \text{(replication)}$$
$$!(P \mid Q) \equiv !P \mid !Q \qquad \text{(distributivity of !)}$$
$$!!P \equiv !P \qquad \text{(idempotence of !)}$$
$$!0 \equiv 0 \qquad \text{(replication of unit)}$$

Figure 1.6: Additional axioms defining the *extended structural congruence*

Transition systems defined by π-calculus equations. Given a process $\mathcal{T} = (I, \mathcal{E})$, we define a transition relation $\rightarrow_{\mathcal{E}}$ on configurations that captures the usual π-calculus reduction rules as follows. Let P and Q be configurations then we have $P \rightarrow_{\mathcal{E}} Q$ if and only if the following conditions hold:

1. $P \equiv (\nu \vec{u})(A(\vec{v}) \mid B(\vec{w}) \mid P')$,
2. the defining equation of A in $\mathcal{E}$ is of the form $A(\vec{x}) = x(\vec{x'}).(\nu \vec{x''})(M) + M'$,
3. the defining equation of B in $\mathcal{E}$ is of the form $B(\vec{y}) = \overline{y}(\vec{y'}).(\nu \vec{y''})(N) + N'$,
4. $\sigma = [\vec{v}/\vec{x}, \vec{w}/\vec{x'}, \vec{z}_A/\vec{x''}, \vec{w}/\vec{y}, \vec{z}_B/\vec{y''}]$ where $\vec{z} = \vec{z}_A, \vec{z}_B$ are fresh names,
5. $\sigma(x) = \sigma(y)$,
6. $Q \equiv (\nu \vec{u}, \vec{z})(\sigma(M) \mid \sigma(N) \mid P')$.

We denote by $\rightarrow_{\mathcal{E}}^*$ the reflexive transitive closure of the relation $\rightarrow_{\mathcal{E}}$. We say that a configuration P is *reachable* in process $\mathcal{T}$ if and only if $I \rightarrow_{\mathcal{E}}^* P$. Finally, we denote by $\mathrm{Reach}(\mathcal{T})$ the set of all reachable configurations of process $\mathcal{T}$.

1.3.2 State-space exploration approaches

The simplest model for concurrent program is probably a finite state automaton. The parallel composition of two processes is simply the product of two automata. Unfortunately, this approach runs directly into the state explosion problem. The number of state in the automaton is exponential in the number of processes considered. Also, this approach is limited to a fixed number of processes, i.e. no (or bounded) process creation. Thus, more general models have been considered.

Petri nets. Petri nets, originally developed by Carl Adam Petri to model chemical reactions [95], found applications in the modeling of concurrent and distributed processes. A Petri net can be seen as a generalization of finite automaton where a state in not a single place (or state) but a bag of places, usually represented by drawing the appropriate number of token into the places. Furthermore, the transition relation moves set of token from places to places, rather than a single token from one place to another. More formally:

Definition 6 (Petri net) *A* Petri net *is a tuple (S, T, W, M_0) where S is a finite set of places, T is a finite set of transitions, $W : (S,T) \cup (T,S) \rightarrow \mathbb{N}$ is a (multi)set of arcs, and M_0 is the initial marking. A marking M is a map: $S \rightarrow \mathbb{N}$. We denote by $\mathcal{M}(S)$ the set of all markings over S. A transition $t \in T$ is fireable at M iff for all $s \in S$, $M(s) \geq W(s,t)$. Firing t at M gives M' defined as $M'(s) = M(s) - W(s,t) + W(t,s)$.*

Given an initial marking M_0 and a target marking M_t, we can ask whether M_t can be *covered* from M_0, i.e. if there is marking $M_{t'}$ and a sequence of transition such that $M_0 \longrightarrow^* M_{t'}$ and $M_{t'} \geq M_t$ where $\geq$ is the pointwise ordering on the marking. We will define formally the covering problem and covering set shortly after the introduction of well-structured transition systems. The covering problem is sometime known as the control-state reachability problem, since we can ask question such as "can the system reach a configuration where a process gets into an error state?" On the other hand, it is weaker than reachability. For instance, we cannot check that a program is deadlock-free using the covering problem. The covering set is the set of all states that can be covered.

The (perhaps) most common method to answer the covering question is to compute the *Karp&Miller tree* [73]. The Karp&Miller tree is an reachability tree augmented with *acceleration*. Acceleration is used to compute the limit of sequences of transition that can be fired infinitely often. Markings are extended to ω-markings: $S \to \mathbb{N} \cup \{\omega\}$ where omega is a *limit* representing an arbitrarily large number of token. Figure 1.7 shows a simple Petri net and its corresponding Karp&Miller tree.

Definition 7 (Karp&Miller tree [73]) *The* Karp&Miller tree *for a Petri net $P = (S, T, W, M_0)$ is a labelled rooted tree (V, E, r, l) where V is the set of vertices, $E\,(\subseteq V \times T \times V)$ is the set of edges, $r\,(\in V)$ is the root, and $l\,(V \to S)$ is the labelling function.*

Let $a \prec b$ denotes that a is an ancestor of b. Additionally:

- *The root r is labelled with M_0, i.e. $l(r) = M_0$.*
- *For every node n, if there exists p such that $p \prec n \wedge l(p) = l(n)$ then n has no successor. Otherwise, n has one successor for every fireable $t \in T$.*
- *For every edges $(m, t, n) \in E$,if there is p such that $p \prec m \wedge l(p) \leq t(l(m))$ then $l(n)(i) = \begin{cases} \omega & \text{if } l(p)(i) < t(l(m))(i) \\ t(l(m))(i) & \text{otherwise} \end{cases}$. Otherwise, $l(n) = t(l(m))$*

The union of all the leaves of the Karp&Miller tree is the covering set of the corresponding Petri net P. Intuitively, the covering set is a set the contains all the markings that P can cover.

There exists many extension for Petri nets. It is worth mentioning transfer and reset nets which enrich Petri nets with edges that consume all the tokens in a place, either to move them in another place (transfer) or remove them completely (reset). These two extensions are monotonic, thus the covering problem is still decidable. However, it requires different algorithms. More details can be found in [49]. Petri nets can also be extended with inhibitory arcs (not fireable while a place is not empty). But this extension is not monotonic anymore and it becomes a Turing-complete model of computation.

Well-structured transition systems (WSTS). A general principle behind the analysis of Petri net and other infinite state systems like lossy-channel systems [5] was identified and formalized in the notion of well-structured transition systems [3, 49]. WSTSs rely on the monotonicity of the transitions and the structure/ordering of the states to make some problem like the covering problem decidable. Since their formalization WSTS are one of the main workhorses to prove decidability results.

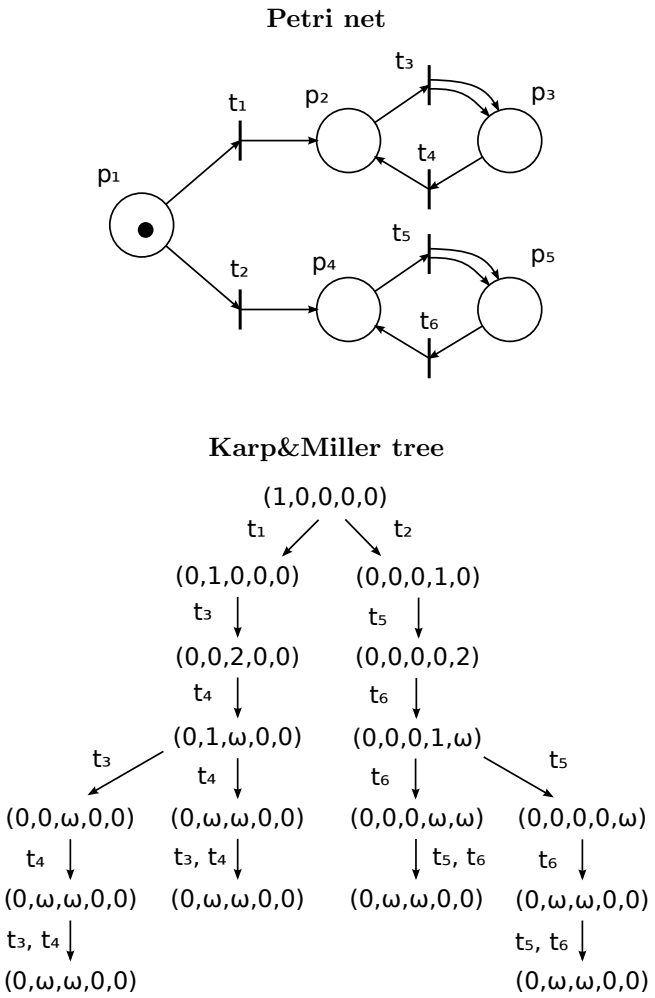


Figure 1.7: Petri net and its Karp&Miller tree [73, Example 4.1]

Definition 8 (Well-structured transition systems) *A* well-structured transition system *is a transition system $\mathcal{T} = (S, s_0, \to, \leq)$ where S is a set of states, $s_0 \in S$ an initial state, $\to \subseteq S \times S$ a transition relation, and $\leq \subseteq S \times S$ a relation satisfying the following two conditions:*

well-quasi-ordering *: $\leq$ is a well-quasi-ordering on S; and*

compatibility *: $\leq$ is upward compatible with respect to $\to$, i.e., for all s_1, s_2, t_1 such that $s_1 \leq t_1$ and $s_1 \to s_2$, there exists t_2 such that $t_1 \to^* t_2$ and $s_2 \leq t_2$.*

The compatibility property is also known as the monotonicity property. It also comes in different flavor. It can be strict ($s_1 < t_1 \Rightarrow s_2 < t_2$) or not. There is also a version known as strong monotonicity ($t_1 \to t_2$). Strong means that one transition has to be simulated by exactly one transition.

Even though S only needs to be a quasi-order in practice it is most of the time a partial order (also antisymmetric). Hence we might also refer to the state-space of a WSTS as a poset.

Definition 9 (Covering problem) *Given a WSTS $(S, s_0, \to, \leq)$ and a state $t \in S$, the* covering problem *asks whether there exists a state $t' \in S$ such that $s_0 \to^* t'$ and $t \leq t'$.*

The covering problem is decidable for WSTS by using a backward search algorithm [3]. An application of the covering problem is asks whether, given a *bad state* s, there exists a reachable state s' of the system that covers the bad state, i.e., $s_0 \to^* s'$ and $s \leq s'$.

Definition 10 (Covering set) *The* covering set *of a well-structured transition system $\mathcal{T}$, denoted* $\mathrm{Cover}(\mathcal{T})$*, is defined by* $\mathrm{Cover}(\mathcal{T}) = {\downarrow}\mathrm{lfp}^{\subseteq}(\lambda X.{\downarrow}S_0 \cup \mathsf{post}(X))$.

Unfortunately, some other problems like computing the covering set which are decidable for Petri nets are not decidable for WSTS [41].

Domains of limits. More recently, generic forward search algorithms, such as the Expand-Enlarge-Check algorithm [52], were developed and classes of WSTS where the covering set is computable were identified [47, 48, 27]. Those algorithm rely on so-called *adequate domain of limits* (ADL) [52, 47]. An adequate domain of limits for the well-quasi-ordering of a WSTS provides an effective representation of all downward-closed sets of configurations, i.e., ADLs are the key for ensuring termination of forward analyses of WSTSs. The concept of limits have been formalized using an axiomatization [52], the topological notion of Noetherian space [58, 59, 47], and the order-theoretic ideal completion [47].

Process Calculi and WSTS. The use of the WSTS framework to analyze π-calculus and other related process calculi is not new [9, 94, 80, 82]. The fragment of the π-calculus known as depth-bounded systems [80] is one of the most general. It is defined semantically as we will see later in Section 1.3.3, but it subsumes many fragments that are defined syntactically [9, 35], in terms of type systems [94, 106, 22], or as abstraction [40].

1.3.3 Depth-bounded systems

The techniques we will develop in the rest of this document are mostly targeted toward depth-bounded systems [80]. We use depth-bounded systems due to the flexibility they offer in modeling concurrent and distributed systems. For the moment, we use the most common definition of depth-bounded systems which is based on the π-calculus. In Chapter 4, we will transfer those ideas to an even more flexible graph-rewriting framework. In the rest of this section, we are dealing with π-calculus processes in the normal form defined in Section 1.3.1.

Definition 11 (Nesting depth (of restriction)) *The* nesting of restrictions *$nest_\nu$ of a process term is measured recursively as follows $nest_\nu(0) = nest_\nu(A(\vec{x})) = nest_\nu(P_1 + P_2) = 0$, $nest_\nu((\nu x)P) = 1 + nest_\nu(P)$, and $nest_\nu(P_1 \mid P_2) = \max\{nest_\nu(P_1), nest_\nu(P_2)\}$.*

Definition 12 (Depth) *The* depth *of a process term P is the minimal nesting of restrictions of process terms in the congruence class of P:*

$$depth(P) = \min\{\, nest_\nu(Q) \mid Q \equiv P \,\}.$$

Definition 13 (Depth-Boundedness) *A set of configurations $\mathcal{C}$ is called* depth-bounded *if there is $k_D \in \mathbb{N}$ such that $depth(P) \leq k_D$ for all $P \in \mathcal{C}$. A process $\mathcal{T}$ is called* depth-bounded *if its set of reachable configurations* $\mathrm{Reach}(\mathcal{T})$ *is depth-bounded.*

Example 1 *The following equations describe a simple client-server system where a server can spawn clients and answer their requests. The server is given by process identifier $Server(x, y)$. The channel x is used for communication with clients. An Env(ironement) process is used to create of new clients.*

$$\begin{aligned}
Server(x) &= x(z).(Reply(z) \mid Server(x)) \\
Client(u, x) &= (u().Client(u, x)) \oplus (Client(u, x) \mid Request(x, u)) \\
Env(x) &= (\nu u)(Client(u, x)) \mid Env(x) \\
Reply(u) &= \overline{u}().0 \\
Request(x, u) &= \overline{x}(u).0
\end{aligned}$$

If the initial configuration is given by $(\nu x)(Env(x) \mid Server(x))$ then the depth of all reachable configurations is bounded by 2.

Definition 14 (wqo for depth-bounded systems) *We define the following natural quasi-ordering $\leq$ on configurations of processes: $P \leq Q$ if and only if $P \equiv (\nu\vec{x})P'$ and $Q \equiv (\nu\vec{x})(P' \mid R)$ for some process term P', R.*

Proposition 2 (Meyer [80])

- *$\leq$ is a well-quasi-order on depth-bounded sets of configurations.*
- *A depth-bounded process $\mathcal{T} = (I, \mathcal{E})$ induces a well-structured transition system $(\mathrm{Reach}(\mathcal{T}), I, \rightarrow_E, \leq)$.*

Notice that the system is well-structured only on Reach($\mathcal{T}$). This is a semantic restriction and cannot be checked automatically. In fact, it is undecidable to determine whether a system is depth-bounded.

Further works by Meyer [81, 82] link π-calculus to Petri nets, using a bound both in depth and in *breadth*. In Section 3.3 and 3.4 we will show how Petri nets and depth-bounded systems are related in term of state-space. Intuitively, Petri nets cannot represent a client-server system with unboundedly many clients, each having unboundedly many messages in its mailbox. A Petri net cannot only represent unboundedly many clients with bounded mailboxes or finitely many clients with unbounded mailboxes. A depth-bounded system can represent both at the same time.

1.3.4 Other approaches to the verification of mobile processes.

Instead of exploring the state-space of all the processes executing as the same time, like the methods we saw above. Compositional verification methods have been studied. The idea is to look only at processes in isolation and compose them in an appropriate framework which can prove global properties form the local information of each process. Those methods sidestep the state-explosion problem at the cost of automation. Except for particular cases those methods requires some human insight about how processes should be composed.

Compositional verification has been studied in the context of model-checking, e.g. [8], but in a context where mobility occurs session types [67, 18] and contracts [68, 101, 26, 25] are more adapted. Session and contracts are based on the behavioral types Such types describe the interactions that one processes can do as contracts or the composition of multiple processes as protocols. This approach allows the verification of stronger properties like deadlock freedom but on a more constrained classes of systems. Those type systems suffer from restrictions that correspond to what can be proved safe in a set of rules where type checking is decidable. Depending on the program one needs to choose the appropriate typing systems and many safe programs do not type.

Session types. The idea behind session types is that composing processes leads to protocols. There are two kinds of types: (1) the protocol describing the overall behavior of all the processes, (2) the end-point types describing the behavior of the individual participants. Since composing processes is extremely difficult, session types ask the user to give the protocol. Then it is possible to project the protocol onto the end-points (individual processes), and check that the end-points implement their role in the protocol. Instead of the classical bottom-up typing, session types work top-down. Session Types provide strong guarantees in term of progress and preservation. However, the number of participants in a protocol is either fixed [67] or given as a parameter [18].

Contracts. A different approach to typing message passing processes are contracts. Contracts are closer to the usual typing systems in the sense that processes are typed bottom-up. A contract defines the operations that a process must do. Contracts have a notion of duality with the environment. The process mirror the environment by swapping sending and receiving. Subtyping

(behaves the same when communicating with the same partner) is also an important component. There exists a typing system [68] for π-calculus without channel creation where types are CCS processes. Then it is possible to reason on CCS which is easier than reasoning on the π-calculus. [101, 26] bring back the restriction operator, but at the expense of decidability. Furthermore, an assume-guarantee rule is given for CCS in order to check the processes compositionally. Discharging the hypotheses of such rules reduces to checking weak open simulation of CCS processes. The weakness of that approach is the model checking of CCS processes is undecidable. [25] uses a different approach and refines the traditional analysis by distinguishing between multiple terminal behaviors: deadlock, success, and an autonomous process taking infinitely many internal steps.

1.4 Contributions

This work puts together the different pieces needed to build an analyzer for depth-bounded systems.

Our first contribution is the development of an adequate domain of limits for depth-bounded processes, presented in Chapter 2. For this purpose we show that downward-closed sets of configurations in depth-bounded processes are characterized by finite unions of regular languages of unranked trees.

In Chapter 3 we propose an abstract interpretation framework that computes precise approximation of covering sets for WSTS, capturing the key insights of acceleration-based algorithms, yet is guaranteed to terminate. The abstract domain of our analysis is based on the ideal completion of the well-quasi-ordering of the analyzed WSTS and an accompanying widening operator. The widening operator mimics the effect of acceleration, but loses enough precision to guarantee termination.

In Chapter 4 we present Picasso, a static analyzer that takes a DBS as input and computes an over-approximations of its covering set. However, Picasso implements an algorithm that exploits the monotonic structure of DBS and often yields precise results. Picasso implements the domain of limits and the abstract interpretation framework described in the earlier parts. We describe the structure of Picasso, some of the key implementation details and how we have used Picasso to automatically verify safety and liveness properties of complex concurrent systems such as nonblocking and distributed algorithms, as well as sequential object-oriented code.

Finally, we show how the covering set can be use as the starting point of further analysis. In Section 5.1, we focus on termination of depth-bounded systems and the implementation of that method in Picasso. In Section 5.2, we generalize the notion of state-machine interface from single object to groups of interacting objects.

Chapter 2

Toward a forward analysis of depth-bounded systems: domain of limits

Depth-bounded processes form one of the most expressive known fragments of the π-calculus for which interesting verification problems are still decidable. In this paper we develop an adequate domain of limits for the well-structured transition systems that are induced by depth-bounded processes. An immediate consequence of our result is that there exists a forward algorithm that decides the covering problem for this class. Unlike backward algorithms, the forward algorithm terminates even if the depth of the process is not known a priori. More importantly, our result suggests a whole spectrum of forward algorithms that enable the effective verification of a large class of mobile systems.

This chapter is joint work with Thomas A. Henzinger and Thomas Wies. It was published in FoSSaCS 2010 as "Forward Analysis of Depth-Bounded Processes" [112].

2.1 Motivation

We are interested in the verification of π-calculus processes [88, 89], i.e., message passing systems that admit unbounded creation of processes and name mobility. We can think of a configuration of such a system as a graph [86, 70]. The vertices of the graph are the processes labelled by their current local state. Edges between processes indicate whether the respective processes share a channel, i.e., whether they are able to communicate with each other. We refer to such a graph as the *communication topology* of a configuration.

The most expressive known fragment of the π-calculus for which interesting verification problems are still decidable is the class of depth-bounded processes [80]. Intuitively, in a depth-bounded process there is a bound on the length of all simple paths in all reachable configuration graphs (the graphs may contain cycles). A typical example of a depth-bounded process is a server-client architecture where a server answers requests of clients and where each client

only knows the name of the server but not the names of other clients. Both the number of simultaneously active clients as well as the number of pending requests for the server can be unbounded.

In this chapter we are concerned with the covering problem for depth-bounded processes. More precisely about the representation of downward-closed sets for depth-bounded systems which are a key component of *forward* algorithms to solve the covering problem. Intuitively, the covering problem asks whether a system can reach a configuration that contains some process that is in a local error state. A decision procedure for the covering problem therefore enables the automated verification of an interesting class of safety properties. Meyer showed in [80] that depth-bounded processes are well-structured transition systems (WSTS) [45, 3, 49]. This implies that the covering problem for depth-bounded processes of *known depth* can be decided using a standard backward algorithm for WSTSs. The question whether the covering problem is decidable for the entire class of depth-bounded processes was open.

We present the first forward algorithm for this problem. Unlike backward algorithms, our algorithm terminates even if the bound of the system is not known a priori. We thus show that the covering problem is decidable for the entire class. Our algorithm is an instance of the expand, enlarge, and check algorithm schema for WSTSs that exhibit a so-called *adequate domain of limits* (ADL) [52, 47]. An adequate domain of limits for the well-quasi-ordering of a WSTS provides an effective representation of all downward-closed sets of configurations, i.e., ADLs are the key for ensuring termination of forward analyses of WSTSs. Our main technical contribution is the development of an adequate domain of limits for depth-bounded processes. For this purpose we show that downward-closed sets of configurations in depth-bounded processes are characterized by finite unions of regular languages of unranked trees.

Besides our theoretical interest in forward analysis of π-calculus processes there are also practical considerations that make forward algorithms more appealing than their backward counterparts. A backward analysis needs to consider all possible unifications between names that may enable processes to synchronize. A forward analysis instead knows which names are equal and which are not. In practice, the search space of a forward analysis is therefore often significantly smaller than the search space of a backward analysis. We give an example that demonstrates this phenomenon in Section 2.2. While the forward algorithm that we consider in this paper is mainly of theoretical interest, our adequate domain of limits suggests a whole spectrum of forward algorithms that enable the effective verification of a large class of mobile systems. This spectrum ranges from acceleration-based algorithms in the style of the Karp&Miller tree [73, 44, 48] to approximation algorithms based on abstract interpretation [32].

2.2 The Covering Problem for Depth-Bounded Processes

Given a depth-bounded process $\mathcal{T} = (I, \mathcal{E})$ which induces a WSTS $(\text{Reach}(\mathcal{T}), I, \rightarrow_E, \leq)$. We are interested in the covering problem for this WSTS.

Forward vs. backward algorithms. The standard algorithm [3] for deciding the covering problem for a WSTS is a backward algorithm that works as follows. Starting from the configuration t that is to be covered one computes the set of backward-reachable configurations of the upward closure of t and then checks whether this set contains the initial configuration. The well-quasi-ordering ensures that the backward analysis terminates.

In the WSTS $(\mathrm{Reach}(\mathcal{T}), I, \rightarrow_E, \leq)$ that is induced by a depth-bounded process $\mathcal{T}$ we implicitly restrict the transition relation $\rightarrow_E$ to the forward-reachable configurations $\mathrm{Reach}(\mathcal{T})$. The predecessor configurations with respect to this restricted transition relation are not effectively computable, i.e., the backward algorithm is not applicable to this WSTS. On the other hand, predecessor configurations for the unrestricted transition relation are effectively computable, but the induced set of backward-reachable configurations is in general not depth-bounded (and thus not well-quasi-ordered by $\leq$). A backward algorithm can only be effectively applied to the WSTS $(\mathcal{C}(k), I, \rightarrow_E, \leq)$. Here $\mathcal{C}(k)$ is the set of all configurations of depth k and k is the maximal depth of configurations in $\mathrm{Reach}(\mathcal{T})$, i.e., $\mathrm{Reach}(\mathcal{T}) \subseteq \mathcal{C}(k)$. Since k must be known in advance, Meyer's result only implies that the covering problem is decidable for depth-bounded processes of known depth. We will show that there exists a forward algorithm that overcomes this limitation.

Besides the theoretical deficiency of backward algorithms there is also a practical reason why forward algorithms are more attractive. We explain this with an example.

Example 2 *Consider the parameterized process $\mathcal{T}(n)$ for $n \in \mathbb{N}$ that is given by the initial configuration $I(n)$:*

$$(\nu x, z, y, y_1, \ldots, y_n)(\mathit{Buffer}_n(x, z, y_1, \ldots, y_n) \mid \mathit{Env}(z, x, y))$$

and the equations $\mathcal{E}(n)$:

$$\begin{array}{rcll}
\mathit{Buffer}_0(x,z) & = & x(y).\mathit{Buffer}_1(x,z,y) & \\
\mathit{Buffer}_i(x,z,y_1,\ldots,y_i) & = & x(y).\mathit{Buffer}_{i+1}(x,z,y_1,\ldots,y_i,y) & \\
& + & \overline{z}(y_1).\mathit{Buffer}_{i-1}(x,z,y_2,\ldots,y_i) & \text{for } 0 < i < n \\
\mathit{Buffer}_n(x,z,y_1,\ldots,y_n) & = & \overline{z}(y_1).\mathit{Buffer}_{n-1}(x,z,y_2,\ldots,y_n) & \\
\mathit{Env}(z,x,y) & = & \overline{x}(y).(\nu u)(\mathit{Env}(z,x,u)) \;+\; z(u).\mathit{Env}(z,x,u) &
\end{array}$$

The process $\mathcal{T}(n)$ models a finite FIFO buffer that stores data sent by the environment in a queue of maximal length n. The queue is modeled using the parameter lists of the process identifiers Buffer_i.

Suppose we want to check that the configuration $P \equiv (\nu x, z)(\mathit{Buffer}_0(x, z))$ is coverable in $\mathcal{T}(n)$. The number of representatives for the set of configurations that are backward-reachable from the upward-closure of P grows exponential in n. The reason is that in one of the continuations of the choices that define $\mathit{Buffer}_i(x, z, y_1, \ldots, y_i)$ the parameter y_1 does not occur. A backward algorithm that computes the predecessors for the execution of this choice has no knowledge about the name that the parameter y_1 denotes. It has to guess whether it is a fresh name or whether it is equal to one of the other names appearing in the continuation. On the other hand, a forward algorithm always knows which name the parameter y_1 denotes. Therefore, the set of representatives for the

configurations that are forward reachable from $I(n)$ grows only linear in n. It is this phenomenon that makes forward algorithms more appealing for the analysis of π-calculus processes.

2.3 An Adequate Domain of Limits

Most forward algorithms for solving the covering problem of WSTSs compute the *cover*, i.e., the downward-closure of the forward-reachable configurations and then check whether this set contains the configuration to be covered. In order to effectively compute the cover, one needs to find a *completion* of the wqo set that contains all the limits of downward-closed sets. The canonical example is the completion for the well-quasi-ordering on markings of Petri nets. It is given by vectors over the set $\mathbb{N}_\omega$ of natural numbers extended with the limit ordinal ω. This completion is the basis for the Karp-Miller algorithm [73] that computes the covering tree of a Petri net. The notion of an *adequate domain of limits* [52, 47] formalizes the completions of wqo sets.

An *adequate domain of limits* (ADL) [52] for a well-quasi-ordered set $(X, \leq)$ is a tuple $(Y, \sqsubseteq, \gamma)$ where Y is a set disjoint from X; (L1) the map $\gamma : Y \cup X \to 2^X$ is such that $\gamma(z)$ is downward-closed for all $z \in X \cup Y$, and $\gamma(x) = {\downarrow}\{x\}$ for all $x \in X$; (L2) there is a limit point $\top \in Y$ such that $\gamma(\top) = X$; (L3) $z \sqsubseteq z'$ if and only if $\gamma(z) \subseteq \gamma(z')$; and (L4) for any downward-closed set D of X, there is a finite subset $E \subseteq Y \cup X$ such that $\gamma(E) = D$, where γ is extended to sets as expected: $\gamma(E) = \bigcup_{z \in E} \gamma(z)$. A *weak adequate domain of limits* (WADL) [47] for $(X, \leq)$ is a tuple $(Y, \sqsubseteq, \gamma)$ satisfying (L1),(L3), and (L4). Note that any weak adequate domain of limits can be extended to an adequate domain of limits.

2.3.1 Limit Configurations

We now describe a weak adequate domain of limits for depth-bounded configurations. In order to finitely represent the limits of infinite downward-closed sets we need to be able to express that certain subterms in a configuration can be replicated arbitrarily often. A natural solution to this problem is to extend configurations with the replication operator ! that is used as a recursion primitive in alternative definitions of the π-calculus [88, 89]. Instead of using replication to express recursion, we use it to effectively represent infinite sets of configurations.

A *limit configuration* E is constructed recursively from process identifiers $A(\vec{x})$, parallel composition $E_1 \mid E_2$, name restriction $(\nu x)E$ and replication $!E$. We extend the congruence relation $\equiv$ from configurations to limit configurations by adding the axiom $!E \equiv (E \mid !E)$. We carry over the definitions of the transition relations of processes and the quasi-ordering $\leq$ from configurations to limit configurations by replacing the congruence relation in the definitions with the extended congruence relation. We then define the denotation $\gamma(E)$ of a limit configuration E as its downward-closure restricted to non-limit configurations:

$$\gamma(E) = \{\, P \mid P \text{ configuration and } P \leq E \,\}$$

The quasi-ordering $\sqsubseteq$ on limit configurations that is required for the adequate domain of limits is defined by condition (L3).

Example 3 *Consider again the client-server process presented in Example 1. The following limit configuration denotes the cover of this process:*

$$\begin{array}{l}(\nu x)((\nu y)(New(y) \mid Server(x,y)) \\ \quad \mid !(\nu z)(Client(z,x) \mid Answer(z)) \\ \quad \mid !(\nu z)(Client(z,x) \mid Request(x,z)))\end{array}$$

We now state the main technical result of this paper. Given a finite set of process identifiers PI, we denote by $\mathcal{C}(PI,k)$ the set of all configurations over PI that have depth at most k. We further denote by $\mathcal{L}(PI,k)$ the set of all limit configurations over PI whose elements denote sets of k-bounded configurations such that $\mathcal{L}(PI,k)$ itself does not contain the configurations in $\mathcal{C}(PI,k)$.

Theorem 1 *Let $k \in \mathbb{N}$ and let PI be a finite set of process identifiers. Then $(\mathcal{L}(PI,k), \sqsubseteq, \gamma)$ is a weak adequate domain of limits for the well-quasi-ordered set $(\mathcal{C}(PI,k), \leq)$.*

In the remainder of this section we prove Theorem 1.

2.3.2 Tree Encoding of Depth-Bounded Configurations

We first relate depth-bounded configurations with graphs of bounded tree-depth, which in turn can be encoded into trees of bounded height. The construction is similar to the one used in [80]. However, we prove that the tree encodings of depth-bounded configurations are not just well-quasi-ordered, but in fact better-quasi-ordered.

Communication topology. We use standard notation for (undirected) graphs. A *labelled graph* over a finite set of labels L is a tuple (G, l_v, l_e) where G is a graph, $l_v : V(G) \to L$ is a *vertex labelling function*, and $l_e : V(E) \to L$ is an *edge labelling function*.

Let $\mathcal{T} = (I, \mathcal{E})$ be a process. Let further n be the maximal arity of all vectors of names occurring in I and $\mathcal{E}$, and let $\mathcal{A}$ be the set of all process identifiers occurring in $I, \mathcal{E}$. Define the set of labels $L \stackrel{\text{def}}{=} 2^{\{0,\ldots,n\}} \cup \mathcal{A} \cup \{\bullet\}$ where $\bullet$ is distinct from all process identifiers. Let P be a configuration of process $\mathcal{T}$ of the form

$$(\nu \vec{x})(\Pi_{j \in J} A_j(\vec{x}_j))$$

where $\vec{x} = x_1, \ldots, x_m$, and the index sets $\{1..m\}$, and J are disjoint. The function ct maps P to a labelled graph over L as follows: the graph consists of vertices corresponding to threads and names occurring in the configuration. Each thread vertex is labelled by the process identifier of the corresponding thread in the configuration. There are edges between thread vertices and name vertices indicating that one of the names in the parameter vector of the thread is the name associated with that name vertex. Formally, $\mathsf{ct}(P)$ is a graph $((V,E), l_v, l_e)$ where

- V is a union of disjoint sets of vertices $\{v_j\}_{j \in J}$ and $\{v_1, \ldots, v_m\}$,
- $E = \{\, \{v_j, v_i\} \mid j \in J \ \wedge\ 1 \leq i \leq m \ \wedge\ x_{j_r} = x_i \text{ for some } 1 \leq r \leq n \,\}$,
- $l_v(v_k) = \begin{cases} A_k & \text{if } k \in J \\ \bullet & \text{otherwise,} \end{cases}$

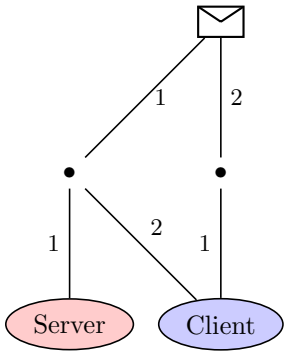


Figure 2.1: $\mathsf{ct}(\,(\nu x)(Server(x) \mid (\nu y)(Client(y,x)|Messages(x,y)))\,)$

- $l_e(\{v_j, v_i\}) = \{\, r \mid j \in J \ \wedge\ 1 \leq i \leq m \ \wedge\ x_{j_r} = x_i \,\}$.

We call $\mathsf{ct}(P)$ the *communication topology* of configuration P. Figure 2.1 shows an example.

Tree-depth. We relate depth-bounded sets of configurations to sets of graphs of bounded tree-depth [93]. A *path* π in a graph G is a sequence $v_1, \ldots, v_n$ of vertices in $V(G)$ that are consecutively connected by edges in $E(G)$. We say that π *connects* vertices v_1 and v_n. We call π *simple path* if for all $1 \leq i < j \leq n$, $v_i \neq v_j$. A *tree* T is a graph such that every pair of distinct vertices in T is connected by exactly one path and this path is simple. A *rooted tree* is a tree with a dedicated root vertex. A *rooted forest* is a disjoint union of rooted trees. The *height* of a vertex v in a rooted forest F, denoted $height(F, v)$, is the number of vertices on the path from the root (of the tree to which v belongs) to v. The *height* of F is the maximal height of the vertices in F. Let v, w be vertices of F and let T be the tree in F to which w belongs. The vertex v is an *ancestor* of vertex w in F, denoted $v \preceq w$, if v belongs to the path connecting w and the root of T. The *closure* $clos(F)$ of a rooted forest F is the graph consisting of the vertices of F and the edge set $\{\, \{v, w\} \mid v \preceq w, v \neq w \,\}$. The *tree-depth* $\mathsf{td}(G)$ of a graph G is the minimum height of all rooted forests F such that $G \subseteq clos(F)$. The tree-depth of a labelled graph is the tree-depth of the enclosed graph. Finally, we say that a set of graphs $\mathcal{G}$ has *bounded tree-depth* if there exists $k \in \mathbb{N}$ such that all graphs $G \in \mathcal{G}$ have tree-depth at most k.

Proposition 3 *A set of configurations $\mathcal{C}$ is depth-bounded iff its communication topologies $\mathsf{ct}(\mathcal{C})$ have bounded tree-depth.*

The proof of Proposition 3 uses Meyer's characterization of sets of depth-bounded configurations in terms of sets of graphs that are bounded in the length of the simple paths [80, Theorem 1]. One can easily show that a set of graphs is bounded in the length of the simple paths if and only if it has bounded tree-depth.

We now relate the ordering on configurations $P \leq Q$ with an ordering on the underlying communication topologies. Given two labelled graphs G_1 and G_2, we say G_1 is (isomorphic to) a *subgraph* of G_2, written $G_1 \hookrightarrow G_2$, iff there exists an injective label-preserving homomorphism from G_1 to G_2.

Lemma 1 *Let P and Q be configurations. Then $P \leq Q$ iff $\mathsf{ct}(P) \hookrightarrow \mathsf{ct}(Q)$.*

Proof. Recall that $P \leq Q$ means the $P \equiv (\nu x)P'$ and $Q \equiv (\nu x)(P' \mid (\nu y)Q')$. Then the proof follows directly from the construction of ct. Notice that the homomorphism from P to Q restricted to the nodes with label $\bullet$ corresponds to the name substitution done in $\equiv$. □

Tree encoding. A labelled rooted tree over a finite set of labels L is a pair (T, l) where T is a rooted tree and $l : V(T) \to L$ a vertex labelling function. We extend the relation $\hookrightarrow$ to rooted labelled trees, as expected, and we say that a tree T_1 is a *subtree* of tree T_2 whenever $T_1 \hookrightarrow T_2$ holds. In the following, we fix a finite set of labels L. Let L_k be the set of all isomorphism classes of labelled graphs G over labels $L \cup (L \times \{1..k\})$ such that G has at most k vertices. Clearly, since L is finite, L_k is finite.

Given a labelled graph G over labels L that has tree-depth at most k, we can construct a labelled rooted tree (T, l) over the set of labels L_k from G as follows. First, let F be a rooted forest of minimal height whose closure contains the graph induced by G. The rooted tree T is constructed from the forest F by extending F with a fresh root vertex r that has edges to all the roots of the trees in F. The labelling function l is defined as follows. Let $v \in V(T)$ be a vertex in T. If $v = r$ then $l(r)$ is the empty graph. Otherwise v is a vertex in F (and thus in G). Let P be the subgraph of G that is induced by the vertices on the path from v to the root (of the tree in F to which v belongs). Now construct a graph P_h from P by adding to the label of each vertex of P its height in F. Then $l(v)$ is the isomorphism class of P_h. Since G has tree-depth at most k, $P_h \in L_k$. Thus, l is well-defined. Let Trees_k be the function mapping a labelled graph G of tree-depth at most k to the set of all labelled rooted trees over L_k that can be constructed from G as described above. We denote by $\mathrm{rng}(\mathit{Trees}_k)$ the set of labelled trees $\bigcup \{ \mathit{Trees}_k(G) \mid G \text{ labelled graph over } L \text{ with } \mathsf{td}(G) \leq k \}$.

Lemma 2 *Let $k \in \mathbb{N}$ and T_1, T_2 be trees in $\mathrm{rng}(\mathit{Trees}_k)$. If T_1 is a subtree of T_2 then $G_1 \hookrightarrow G_2$ for all G_1, G_2 with $T_1 \in \mathit{Trees}_k(G_1)$ and $T_2 \in \mathit{Trees}_k(G_2)$.*

Let T be a rooted tree and $x, y \in V(T)$ two vertices. The infimum of x and y, denoted $x \inf y$, is the vertex $z \in V(T)$ with the greatest height such that $z \preceq x$ and $z \preceq y$. Given rooted trees T_1 and T_2, a function φ is an *inf-preserving embedding* from T_1 into T_2 iff (1) $\varphi : V(T_1) \to V(T_2)$ is injective, and (2) for all $x, y \in V(T_1)$, $\varphi(x \inf y) = \varphi(x) \inf \varphi(y)$. An embedding between two rooted labelled trees over the same set of labels is *label-preserving* iff it maps vertices to vertices with the same label.

Clearly, if a tree is a subtree of another tree then there exists an inf and label preserving embedding between these trees. For trees that result from the tree encoding of configurations the converse holds, too. Vertices of different height in such trees have always different labels. Thus, an inf and label-preserving embedding between such trees also preserves antecedence of vertices.

Lemma 3 *Let $k \in \mathbb{N}$ and T_1, T_2 be trees in $\mathrm{rng}(\mathit{Trees}_k)$. Then the following two properties are equivalent:*

1. *there exists an inf and label-preserving embedding from T_1 to T_2;*

2. T_1 is a subtree of T_2.

Laver [75] proved a variation of Kruskal's tree theorem for trees labelled by a bqo set, namely that countable rooted trees labelled by a bqo set are a bqo under inf-preserving embedding. Similar to Friedman's special case of Kruskal's tree theorem, we get the special case of Laver's theorem that rooted trees labelled by a finite set of labels are better-quasi-ordered by inf and label-preserving embedding. Thus, together with Lemma 2 we get the following proposition.

Proposition 4 *For any $k \in \mathbb{N}$, $(\mathrm{rng}(\mathit{Trees}_k), \hookrightarrow)$ is a bqo set.*

2.3.3 Limit Configurations as Ideal Completions

Finkel and Goubault-Larrecq [47] characterize the minimal candidates for the WADLs of a wqo set X in terms of its ideal completion. This means that the set of all downward-closed directed subsets of X forms a WADL for X. We use this observation to prove that limit configurations form WADLs for depth-bounded configurations.

Proposition 5 *The directed downward-closed sets of depth-bounded configurations are exactly the denotations of limit configurations.*

By [47, Proposition 3.3] the above proposition implies Theorem 1. In our proof of Proposition 5 we characterize the tree encodings of downward-closed sets of configurations in terms of the languages of *hedge automata* [29, Chapter 8].

Hedge automata. A *(nondeterministic) finite hedge automaton* $\mathcal{A}$ over a finite alphabet Σ is a tuple (Q, Σ, Q_f, Δ) where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a finite set of transition rules of the following form:

$$a(R) \to q$$

where $a \in \Sigma$, $q \in Q$, and $R \subseteq Q^*$ is a regular language over Q. These languages R occuring in the transition rules are called *horizontal languages*.

A *run* of $\mathcal{A}$ on a rooted labelled tree T with vertex label function $l : V(T) \to \Sigma$ is a vertex label function $r : V(T) \to Q$ such that for each vertex $v \in V(T)$ with $a = l(v)$ and $q = r(v)$ there is a transition rule $a(R) \to q$ with $r(v_1) \dots r(v_n) \in R$ where $v_1, \dots, v_n$ are the immediate successors of v in T. In particular, to apply a rule to a leaf, the empty word ϵ has to be in the horizontal language of the rule R.

A rooted labelled tree T is *accepted* by $\mathcal{A}$ if there is a run r of $\mathcal{A}$ on T such that r labels the root of T by a final state. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all rooted labelled trees over Σ that are accepted by $\mathcal{A}$.

Finite partitions of well-quasi-ordered sets. In order to characterize the horizontal languages of the constructed hedge automaton we will define equivalence classes on the vertices of the individual levels of the tree encodings. For this purpose, the following definition will be useful. Let $(X, \leq)$ be a well-quasi-ordered set. We call a partition $\mathcal{P} \subseteq 2^X$ of X an *infinite chain partition* if and only if (1) $\mathcal{P}$ is finite and (2) for all $Y \in \mathcal{P}$, either Y is a singleton or Y contains an infinite chain C such that $Y \leq C$.

Proposition 6 *If $(X, \leq)$ is a countable well-quasi-ordered set then there exists an infinite chain partition of X.*

Proof. We can construct an infinite chain partition $\mathcal{P}$ of X recursively using the following procedure: according to [38, Theorem 5], X can be partitioned into finitely many irreducible subsets $Y_1, \ldots, Y_n$. By [38, Proposition 3], for each $1 \leq i \leq n$, Y_i contains a chain C_i with $Y_i \leq C_i$. For each $1 \leq i \leq n$, check if Y_i contains an infinite chain with this property. If it does then add Y_i to $\mathcal{P}$. Otherwise pick one finite chain C_i with $Y_i \leq C_i$. Since C_i is finite it contains a greatest element y_i. Then let $Z_i = \{ y \in Y_i \mid y_i \leq y \}$ be the set of elements in Y_i that are equivalent to y_i wrt. the quasi-ordering $\leq$. Since Y_i contains no infinite chains that are large in Y_i, the set Z_i is finite. Then add all singletons $\{z\}$ with $z \in Z_i$ to $\mathcal{P}$ and recursively apply the above procedure on the well-quasi-ordered set $(Y_i - Z_i, \leq)$. Clearly, if this procedure terminates then the resulting set $\mathcal{P}$ is an infinite chain partition of X. Thus, assume that the procedure does not terminate. Then the algorithm constructs a strictly decreasing infinite sequence $Y_1 \supseteq Y_2 \supseteq \ldots$ of subsets of Y with $Y_i - Y_{i+1} > Y_{i+1} - Y_{i+2}$ for all $i \in \mathbb{N}$. Define $X_i = Y_i - Y_{i+1}$ then each X_i is nonempty, i.e., we can choose $x_i \in X_i$ for each $i \in \mathbb{N}$ such that we get an infinite descending chain $x_1 > x_2 > \ldots$ of elements in X. This contradicts the fact that $\leq$ is well-founded. □

In order to prove Proposition 5, we first prove that every directed downward-closed set of depth-bounded configurations is the denotation of a limit configuration.

Lemma 4 *Let D be a downward-closed set of configurations then there exists a limit configuration E such that $D = \gamma(E)$.*

For proving the lemma, let $D = (P_i)_{i \in \mathbb{N}}$ be a downward-closed directed family of configurations and let k be the maximal tree-depth of the graphs in $\mathsf{ct}(D)$. Choose some $Q_0 \in D$ whose communication topology has tree-depth k. Using Q_0 construct an ascending chain $D' = (Q_i)_{i \in \mathbb{N}}$ as follows: for each $i \in \mathbb{N}$ choose $Q_i \in D$ such that $P_i \leq Q_i$ and $Q_{i-1} \leq Q_i$. Such Q_i exists for each $i \in \mathbb{N}$ since D is directed and, by induction, $Q_{i-1} \in D$. Then by construction (1) $D = {\downarrow} D'$ and (2) all elements in D' have tree-depth k. Let $(G_i)_{i \in \mathbb{N}}$ be the family of labelled graphs $G_i = \mathsf{ct}(Q_i)$. Now for each $i \in \mathbb{N}$ choose a tree $T_i \in \mathit{Trees}_k(G_i)$ such that the family $\mathcal{T} = (T_i)_{i \in \mathbb{N}}$ is an ascending chain with respect to the subtree relation. Such a family exists because the G_i are ordered by subgraph isomorphism and all G_i have the same tree-depth. Without loss of generality we assume that the vertex sets of all trees T_i are pairwise disjoint.

Let $V = \bigcup_{i \in \mathbb{N}} V(T_i)$, $E = \bigcup_{i \in \mathbb{N}} E(T_i)$, and let l be the union of all the vertex labelling functions of the labelled trees T_i. The height of the vertices in the trees T_i range from 1 to $k+1$. For a vertex $v \in V$ of height $h > 1$ we denote by $\mathit{parent}(v) \in V$ the parent of v in the tree T_i to which v belongs. Similarly, for a vertex $v \in V$ we denote by $\mathit{Children}(v)$ the set of all vertices that are direct successors of v in the tree to which v belongs. We extend the functions *parent* to sets of vertices, as expected. Furthermore, let $T(v)$ be the subtree rooted in v of the tree T_i with $v \in V(T_i)$. For all $1 \leq h \leq k+1$, let V_h be the set of all vertices in V that have height h. For all h we extend the relation $\hookrightarrow$ from

labelled rooted trees to vertices in V_h as expected: for all $v, w \in V_h$, $v \hookrightarrow w$ iff $T(v) \hookrightarrow T(w)$. From Proposition 4 and Property 3 of Proposition 1 follows that for all h, $(V_h, \hookrightarrow)$ is a bqo.

We will now construct a finite hedge automaton $\mathcal{A}$ from the family of trees $\mathcal{T}$ whose language is both small and large in $\mathcal{T}$. For this purpose we define an equivalence relation on each V_h that partitions V_h into finitely many equivalence classes. These equivalence classes serve as the states of the automaton.

For each $i \in \mathbb{N}$ fix some injective label-preserving homomorphism $\phi_i : V(T_i) \to V(T_{i+1})$ and denote by $\phi_{[i,j]}$ the composition $\phi_{j-1} \circ \cdots \circ \phi_i$ if $j > i$ and the identify function id if $j = i$. Then define an equivalence relation $\sim$ on V as follows: for all $v_i \in V(T_i)$ and $v_j \in V(T_j)$

$$v_i \sim v_j \quad \text{iff} \quad \begin{array}{l} i \leq j \text{ and } \phi_{[i,j]}(v_i) = v_j \text{ or} \\ i \geq j \text{ and } \phi_{[j,i]}(v_j) = v_i \end{array}$$

Now, recursively define an equivalence relation $\simeq_h$ on V_h for each $1 \leq h \leq k+1$ as follows: for $h = 1$ we simply have $v \simeq_1 w$ for all $v, w \in V_1$. In order to define $\simeq_h$ for $h > 1$ we need some intermediate definitions. Given an equivalence class U in the quotient of V_{h-1} wrt. $\simeq_{h-1}$, let $\mathit{Children}(U)$ be the set of equivalence classes $\tilde{v}$ in the quotient $V_h/_\sim$ such that some $v \in \tilde{v}$ has a parent in U. Since $(V_h, \hookrightarrow)$ is a bqo, and $\mathit{Children}(U) \subseteq 2^{V_h}$, it follows from Proposition 1 that $(\mathit{Children}(U), \hookrightarrow_1)$ is also a bqo and thus a wqo. Furthermore, $\mathit{Children}(U)$ is countable. Thus, by Proposition 6 there exists an infinite chain partition of $\mathit{Children}(U)$. For each U, choose one such infinite chain partition $\mathcal{P}(U)$ of $\mathit{Children}(U)$. Then for $v, w \in V_h$ we define: $v \simeq_h w$ iff there exists $U \in V_{h-1}/_{\simeq_{h-1}}$ such that (1) $\mathit{parent}(v), \mathit{parent}(w) \in U$ and (2) there is $P \in \mathcal{P}(U)$ such that $v, w \in \bigcup P$.

Lemma 5 *Each $\simeq_h$ is indeed an equivalence relation on V_h and that each $\simeq_h$ partitions V_h into finitely many equivalence classes.*

Proof. By induction on h:

$h = 1$**:** directly follows from the definition of $\simeq_1$.

$h > 1$**:** By induction, we know that $V_{h-1}/_{\simeq_{h-1}}$ has finitely many equivalence classes. Additionally, $\mathcal{P}(U)$ partition the children into finitely many sets. These two facts imply that $\simeq_h$ partitions V_h into finitely many equivalence classes. Furthermore, proving that $\simeq_h$ is an equivalence relation requires proving reflexivity, symmetry, and transitivity. The first two follows directly from the definition. Transitivity additionally relies in the fact that $\mathcal{P}(U)$ is a partition.

□

Furthermore, using the definition of infinite chain partitions one can easily prove the following properties.

Lemma 6 *Let $U \in V_h/_{\simeq_h}$ then*

1. *all $v \in U$ have the same label,*
2. *U is directed with respect to $\hookrightarrow$,*

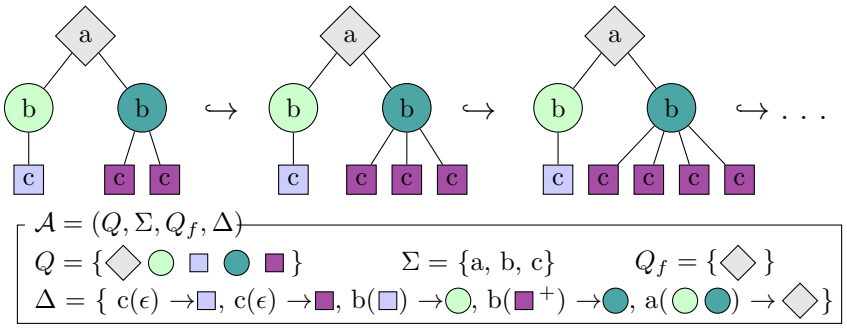


Figure 2.2: A chain of labelled trees with the equivalence classes under the relations $\simeq_h$ and the constructed hedge automaton

3. *if* $h = 1$ *then* U *contains exactly the root vertices of all the trees* T_i*,*
4. *if* $h > 1$ *then* $parent(U) \subseteq U'$ *for some* $U' \in V_{h-1}/_{\simeq_{h-1}}$ *and*
   (a) *either all vertices in* U' *have at most one child in* U *or*
   (b) *every* $v \in U$ *is contained in a proper infinite chain* $C \subseteq U$ *and for every finite subset* $V \subseteq U$ *there exists* $v' \in U'$ *such that* $V \hookrightarrow_1 Children(v') \cap U$*.*

Now let $\simeq$ be the union of all the relations $\simeq_h$. Then $\simeq$ is an equivalence relation on V that partitions V into finitely many equivalence classes. For an equivalence class $U \in V/_\simeq$, let $C(U)$ be the set of all equivalence classes that contain children of vertices in U. Furthermore, let $l(U)$ be the unique label of all vertices in U, and let $m(U)$ denote 1 if every parent of a vertex $v \in U$ has at most one child in U and, otherwise, let $m(U)$ denote the symbol $+$. Now define the hedge automaton $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where:

- $Q = V/_\simeq$,
- $\Sigma = L_k$,
- $Q_f = V_1/_\simeq$,
- Δ consists of transition rules of the following form for each $U \in V/_\simeq$
  - $l(U)(U_1^{m(U_1)} \cdots U_n^{m(U_n)}) \rightarrow U$ if $C(U) = \{U_1, \ldots, U_n\}$
  - $l(U)(\epsilon) \rightarrow U$ if $C(U) = \emptyset$.

Figure 2.2 depicts a chain of trees and the constructed automaton $\mathcal{A}$. The equivalence classes in the quotient $V/\simeq$ are highlighted in the trees.

Using Lemma 6 we can now prove that the language accepted by $\mathcal{A}$ is both small and large in $\mathcal{T}$.

Lemma 7 *$\mathcal{L}(\mathcal{A})$ is small in $\mathcal{T}$, i.e., $\forall T \in \mathcal{L}(\mathcal{A})\, \exists i \in \mathbb{N} :\ T \hookrightarrow T_i$.*

Proof. For proving the lemma, let T_U be a tree labelled by L_k and r a run of $\mathcal{A}$ on T_U. We show by induction on the height of T_U that if $r(w) = U$ for the root w of T_U then there exists $v \in U$ such that $T_U \hookrightarrow T(v)$.

If $h = 1$ then T_U consists of a single root vertex w that is a leaf. Then the transition rule in Δ used to label w in r is of the form $l(U)(\epsilon) \to U$. Thus, by construction of $\mathcal{A}$ all trees $T(v)$ for vertices $v \in U$ consist of the single leaf vertex v labeled by $l(U)$, i.e., $T_U \hookrightarrow T(v)$ for all $v \in U$.

If $h > 1$ then the transition rule in Δ used to label w must have the form

$$l(U)(U_1^{m_1} \cdots U_n^{m_n}) \to U$$

with $C(U) = \{U_1, \ldots, U_n\}$ and $m_i = m(U_i)$ for all $1 \leq i \leq n$. Let

$$T_{1,1}, \ldots, T_{1,r_1}, \ldots, T_{n,1}, \ldots, T_{n,r_n}$$

be the subtrees of T_U rooted at the children of w such that r labels the root of each tree $T_{i,j}$ by U_i. These trees have height $h - 1$ and r is a run of $\mathcal{A}$ on each of these trees. Thus, by induction hypothesis there exist vertices

$$v_{1,1}, \ldots, v_{1,r_1} \in U_1 \ \ldots \ v_{n,1}, \ldots, v_{n,r_n} \in U_n$$

with $T_{i,j} \hookrightarrow T(v_{i,j})$ for all $1 \leq i \leq n$, $1 \leq j \leq r_i$. If two vertices $v_{i,j}$ and $v_{i',j'}$ coincide then we must have $i = i'$. Thus, $r_i > 1$ and $m(U_i) = +$, i.e., by construction of $\mathcal{A}$, there are vertices in U that have more than one child in U_i. Then U_i satisfies property 4.(b) of Lemma 6, i.e., U_i contains a proper infinite chain C with $v_{i,j} \in C$. Hence, we can choose two vertices $v'_{i,j}, v'_{i',j'} \in C$ that are (1) distinct, (2) disjoint from all other $v_{i,j}$, and (3) satisfy $T_{i,j} \hookrightarrow T(v'_{i,j})$ and $T_{i,j} \hookrightarrow T(v'_{i',j'})$. Therefore, without loss of generality assume that all the $v_{i,j}$ are distinct. Now for any $1 \leq i \leq n$ we can find $v_i \in U$ such that

$$\{v_{i,1}, \ldots, v_{i,r_i}\} \hookrightarrow_1 \mathit{Children}(v_i) \cap U_i$$

Namely, if $r_i = 1$ then $v_i = \mathit{parent}(v_{i,1})$ and if $r_i > 1$ then such v_i exists by property 4.(b). Now, using the fact that U is directed we can inductively construct an upper bound $v \in U$ of all the v_i with respect to the wqo $\hookrightarrow$. Then we have by construction:

$$\{v_{1,1}, \ldots, v_{1,r_1}, \ldots, v_{n,1}, \ldots, v_{n,r_n}\} \hookrightarrow_1 \mathit{Children}(v)$$

We conclude that $\mathit{Children}(w) \hookrightarrow_1 \mathit{Children}(v)$ and $l(v) = l(U)$, i.e., $T_U \hookrightarrow T(v)$, which concludes the induction proof. □

Using a similar inductive proof we can show that $\mathcal{L}(A)$ is large in $\mathcal{T}$.

Lemma 8 *$\mathcal{L}(\mathcal{A})$ is large in $\mathcal{T}$, i.e., $\forall i \in \mathbb{N}\, \exists T \in \mathcal{L}(\mathcal{A}) :\ T_i \hookrightarrow T$.*

Note that by construction of $\mathcal{A}$ the tree encoding operation can be reversed on the trees in $\mathcal{L}(\mathcal{A})$. Let $D_{\mathcal{A}}$ be the corresponding set of configurations. From Lemmas 7,8,2, and 1 follows that $D = {\downarrow}D' = {\downarrow}D_{\mathcal{A}}$. From $\mathcal{A}$ we can now easily construct a limit configuration E whose denotation is the downward closure of $D_{\mathcal{A}}$. It follows that $D = {\downarrow}D_{\mathcal{A}} = \gamma(E)$ which proves Lemma 4.

Lemma 9 *For any limit configuration E, $\gamma(E)$ is a downward-closed directed set.*

Clearly $\gamma(E)$ is downward-closed. For proving that $\gamma(E)$ is directed, we can again construct a hedge automaton $\mathcal{A}$ from E, such that the tree encoding operation can be reversed on all trees accepted by $\mathcal{A}$ and the downward-closure of the resulting configurations $D_{\mathcal{A}}$ coincides with $\gamma(E)$. Using a simple pumping argument for the language $\mathcal{L}(\mathcal{A})$ we can show that for every two trees $T_1, T_2 \in \mathcal{L}(\mathcal{A})$ there exists a tree $T \in \mathcal{L}(\mathcal{A})$ such that $T_1 \hookrightarrow T$ and $T_2 \hookrightarrow T$. It follows that $D_{\mathcal{A}}$ is directed and thus $\gamma(E)$.

2.4 Forward Analysis of Depth-Bounded Processes

The expand, enlarge, and check (EEC) algorithm of Geeraerts et al. [52] is a forward algorithm that decides the covering problem for *effective* WSTSs with appropriate adequate domain of limits.

A WSTS $(X, x_0, \rightarrow, \leq)$ and an adequate domain of limits $(Y, \sqsubseteq, \gamma)$ for the wqo $(X, \leq)$ are *effective* if the following conditions are satisfied: (E1) X and Y are recursively enumerable; (E2) for any $x_1, x_2 \in X$, one can decide whether $x_1 \rightarrow x_2$; (E3) for any $z \in X \cup Y$ and for any finite subset $Z \subseteq X \cup Y$, one can decide whether $\mathit{Post}(\gamma(z)) \subseteq \gamma(Z)$; and (E4) for any finite subsets $Z_1, Z_2 \subseteq X \cup Y$, one can decide whether $\gamma(Z_1) \subseteq \gamma(Z_2)$.

Theorem 2 (Geeraerts et al. [52]) *There exists an algorithm to decide the covering problem for effective WSTSs with an adequate domain of limits.*

We argue that the WSTS induced by a depth-bounded process together with its WADL of limit configurations are effective. The conditions (E1) and (E2) are clearly satisfied. Also given a limit configuration z we can compute a finite set of limit configurations denoting $\mathit{Post}(\gamma(z))$. Note further that Proposition 5 implies that for any finite subsets $Z_1, Z_2 \subseteq \mathcal{L}(PI, k)$, $\gamma(Z_1) \subseteq \gamma(Z_2)$ holds if and only if for all $z_1 \in Z_1$ there exists $z_2 \in Z_2$ such that $\gamma(z_1) \subseteq \gamma(z_2)$. The inclusion problem $\gamma(z_1) \subseteq \gamma(z_2)$ can be reduced to the language inclusion problem for deterministic hedge automata, which is decidable. For this purpose, one computes deterministic hedge automata from the finitely many tree encodings of the configurations of z_1 and z_2 and then checks whether the language of some automaton of z_1 is included in the language of some automaton of z_2. Thus conditions (E3) and (E4) are also satisfied.

Finally, let us explain why the EEC algorithm terminates on depth-bounded systems even if the bound of the system is not known a priori. The idea of the algorithm is to simultaneously enumerate two infinite increasing chains. The first chain $X_0 \subseteq X_1 \ldots$ is a sequence of finite subsets of X that contains all reachable configurations of the analyzed system. The second chain $Y_0 \subseteq Y_1 \subseteq \ldots$ is a sequence of finite subsets of Y that contains all limits Y. In each iteration i the algorithm computes an under and an over-approximation of the analyzed system for the current pair (X_i, Y_i) of elements in the chain. These approximations are such that the under-approximation is guaranteed to detect that t can be covered if X_i contains a path to a covering state. The over-approximation is guaranteed to detect that t cannot be covered if Y_i can express $\downarrow \mathit{Post}^*(\downarrow s_0)$ and this set does not cover t. The conditions on the chains ensure that one of the two conditions eventually holds for some $i \in \mathbb{N}$.

For deciding the covering problem of depth-bounded systems we can now simply enumerate the sets $\mathcal{C}(PI) = \bigcup_{k\in\mathbb{N}} \mathcal{C}(PI,k)$ and $\mathcal{L}(PI) = \bigcup_{k\in\mathbb{N}} \mathcal{L}(PI,k)$. Then in each iteration of the EEC algorithm the pair (X_i, Y_i) is contained in some limit domain $\mathcal{C}(PI,k), \mathcal{L}(PI,k))$ and the conditions on the chains for termination of the EEC algorithm are still satisfied.

Theorem 3 *The covering problem for depth-bounded processes is decidable.*

Complexity. Not only depth-bounded systems inherit the nonprimitive recursive complexity lower-bound of the nets it subsumes [107]. Encoding the Ackermann function, for instance, can be done easily using a Church-like encoding for the integers. Since the computation of the Ackermann function terminates the system is depth-bounded.

2.5 Further related work

Depth-bounded processes are semantically defined in terms of reachable configurations. The idea of using nesting of π-calculus names as a measure of the expressiveness a system goes back to [106] where it takes the form of a type system. This question of expressiveness and name dependencies was investigated further in [22, 55]. Ostrovský [94] uses depth-boundedness and well-structured transition systems in the context of decidability of session types. While checking depth-boundedness is in general undecidable, many fragments of the π-calculus that are defined syntactically [9, 35] or in terms of type systems [94, 106, 22] are subsumed by depth-bounded processes. Our result carries over to these fragments. Further related work can be found in the context of graph rewriting systems. Bauer and Wilhelm [17] developed an overapproximating shape analysis for graph rewriting systems whose reachable configurations have a star-like shape. Such systems are bounded in the length of the acyclic paths. Our result naturally generalizes to such systems and promises complete algorithms for their verification.

The control reachability problem for the π-calculus has been studied in [90, 9, 113, 36]. The approaches taken in [90, 113] consider only finitary systems that impose a bound on the number of threads that can be dynamically created. Delzanno [36] considers an abstraction-based approach that applies to the full asynchronous π-calculus. This approach is sound but in general incomplete. Amadio and Meyssonnier [9] considers *input-bounded systems*, a syntactically defined fragment of the asynchronous π-calculus that allows some form of name creation and name mobility. Input-bounded systems and depth-bounded systems are incomparable. Unlike depth-bounded systems, input-bounded systems cannot truly model the dynamic creation of an unbounded set of threads by a given thread such that all of these threads remain active and communicate. Because of such restrictions, input-bounded systems are less interesting from a practical point of view. Conversely, input-bounded systems are not depth-bounded because they enable the creation of unbounded chains of inactive threads. We suspect that there is a relaxation of the depth-boundedness condition that subsumes both fragments and for which control-reachability is still decidable.

Chapter 3

Bridging the gap between theory and practice: ideal abstraction

In the previous chapter we developed a adequate domain of limits for depth-bounded systems. Even though, we can use these limits to solve the covering problem, it does not yet give a efficient and practical analysis. Furthermore, we are interested not only in the covering problem, but also in the covering set which can be used for further analysis as we will see in Chapter 5. For this pats we are also broadening our scope and consider additional kinds of WSTS such as Petri nets and lossy-channel systems. Among the most popular algorithms to compute the covering set are acceleration-based forward algorithms like the Karp&Miller tree. Termination of these algorithms can only be guaranteed for flattable WSTS. Yet, many WSTS of practical interest are not flattable and the question whether any given WSTS is flattable is itself undecidable. We therefore propose an analysis that computes the covering set and captures the essence of acceleration-based algorithms, but sacrifices precision for guaranteed termination. Our analysis is an abstract interpretation whose abstract domain builds on the ideal completion of the well-quasi-ordered state space, and a widening operator that mimics acceleration and controls the loss of precision of the analysis. We present instances of our framework for various classes of WSTS.

This chapter is joint work with Thomas A. Henzinger and Thomas Wies. It was published in VMCAI 2012 as "Ideal Abstractions for Well-Structured Transition Systems" [114].

3.1 Motivation

One of the great successes in applying model checking techniques to the analysis of infinite state systems has been achieved by studying the class of *well-structured transition systems* (WSTS) [3, 49, 73, 48, 47, 71, 52, 51]. Interesting classes of WSTS include Petri nets [95] and their monotonic extensions [41], lossy channel systems [5], and dynamic process networks such as depth-bounded

processes [80].

Many interesting verification problems are decidable for WSTS. In particular, the verification of a large class of safety properties can be reduced to the *coverability problem*, which is decidable for WSTS that satisfy only a few additional mild assumptions [3]. In this paper, we are not just interested in solving the coverability problem, but in the more general problem of computing the *covering set* of a WSTS T. The covering set $\mathrm{Cover}(T)$ is defined as the downward-closure of the reachable states of the system $\mathrm{Cover}(T) = {\downarrow}Post^*({\downarrow}\{s_0\})$. With the help of the covering set one can decide the coverability problem, but also answer other questions of interest such as boundedness (which asks whether $\mathrm{Cover}(T)$ is finite) and U-boundedness (which asks whether $\mathrm{Cover}(T) \cap U$ is finite for some upward-closed set U). While coverability is decidable for most WSTS, boundedness is not [41], i.e., the covering set is not always computable. Therefore, our goal is to compute precise over-approximations of the covering set, instead of computing this set exactly. In this paper, we present a new analysis based on abstract interpretation [32, 31] that accomplishes this goal.

One might question the rational of using an approximate analysis for solving decidable problems such as coverability. However, in practice one often uses coverability to give approximate answers to verification problems that are undecidable even for WSTS (such as general reachability). Thus, completeness is not always a primary concern. Also, one should bear in mind that even though coverability is decidable, its complexity is non-primitive recursive for many classes of WSTS [107], i.e., from a practical point of view the problem might as well be undecidable. Nevertheless, the techniques that have been developed for solving the coverability problem provide important algorithmic insights for the design of good approximate analyses.

Among the best understood algorithms for computing the exact covering set of a WSTS are acceleration-based algorithms such as the Karp&Miller tree construction for Petri nets [73] or the more general clover algorithm [48]. These algorithms exploit the fact that every downward-closed subset of a well-quasi-ordering can be effectively represented as a finite union of order ideals [58, 47]. The covering set is then computed by identifying sequences of transitions in the system that correspond to loops leading from smaller to larger states in the ordering, and then computing the exact set of ideals covering the states reachable by arbitrary many iterations of these loops. This process is referred to as ω- or lub-acceleration. Since acceleration is exact, these algorithm compute the exact covering set of a WSTS, whenever they terminate. Since the covering set is not always computable, termination is only guaranteed for so-called *flattable* systems [48]. In a flattable WSTS the covering set can be obtained by a finite sequence of lub-accelerations of finite sequences of transitions. In particular, this means that every nested loop of transitions can be decomposed into a finite sequence of simple loops. Many WSTS of practical interest do not satisfy this property. We provide an example of such a system in the next section.

We are the first to propose an abstract interpretation framework that computes precise approximation of covering sets for WSTS, captures the key insights of acceleration-based algorithms, yet is guaranteed to terminate even on non-flattable WSTS. The abstract domain of our analysis is based on the ideal completion of the well-quasi-ordering of the analyzed WSTS and an accompanying widening operator. The widening operator mimics the effect of acceleration, but loses enough precision to guarantee termination. Instead of accelerating

$$
\begin{array}{rcl}
\textit{Equations}: & & \\
\mathrm{client}(C,S) & = & C().\mathrm{client}(C,S) \oplus (\overline{S}(C).0 \mid \mathrm{client}(C,S)) \\
\mathrm{server}(S) & = & S(C).(\overline{C}().0 \mid \mathrm{server}(S)) \\
\mathrm{env}(S) & = & \mathrm{env}(S) \mid (\nu\, C)\mathrm{client}(C,S) \\
\textit{Initial state}: & (\nu\, S)(\mathrm{server}(S) \mid \mathrm{env}(S)) &
\end{array}
$$

Figure 3.1: A π-calculus process implementing a client-server protocol.

loops that lead from sets of smaller to sets of larger states, our widening operator only accelerates the difference between these sets of states, independently of the actual sequence of transitions that produced them. We present instances of our framework for the WSTS classes of Petri nets, lossy channel systems, and depth-bounded process networks.

3.2 Overview of the Analysis

We start with an example of a non-flattable system and illustrate how our analysis computes its covering set. Our example is given by the π-calculus process shown in Figure 3.1. The process models a concurrent system that implements a client-server protocol using asynchronous message passing. The process consists of one single server thread, an environment thread, and an unbounded number of client threads. Each type of threads is defined by a recursive π-calculus equation. In each loop iteration of a client, the client nondeterministically chooses to either wait for a response from the server on its own dedicated channel C, or to send a new request to the server. Requests are sent asynchronously and modeled as threads that wait for the server to receive the client's channel name over the server's dedicated channel S and then terminate immediately. In each iteration of the server loop, the server waits for incoming requests on its own channel S and then asynchronously sends a response back to the client using the client's channel name C received in the request. The environment thread models the fact that new clients can enter the system at anytime. In each iteration of the environment thread, it spawns a new client thread with its own dedicated fresh channel name. The initial state of the system consists only of the server and the environment thread.

The states of a π-calculus process can be represented as a *communication graph* with nodes corresponding to threads (labeled by their id) and edges corresponding to channels (labeled by channel names). The left hand side of Figure 3.2 shows the communication graph representing the process:

$$\mathrm{server}(S) \mid \mathrm{client}(C_1,S) \mid \overline{S}(C_1).0 \mid \mathrm{client}(C_2,S) \mid \mathrm{env}(S)$$

The graph of Figure 3.2 are more compact than the one obtained using the ct function defined in Section 2.3.2. Here the name nodes are implicit because we look at systems respecting the unique receiver condition, i.e. only a single process receive from a name and this name does not change over time. This allows us to draw more compact graphs but does not change the theoretical results.

The transition relation on processes is monotone with respect to the ordering on processes that is induced by subgraph isomorphism between their commu-

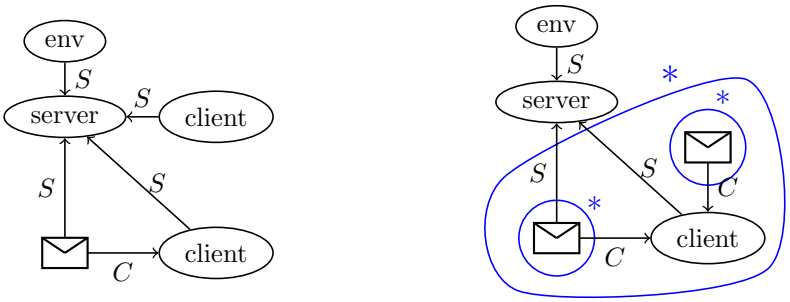


Figure 3.2: Communication graph of the system in Figure 3.1 and the symbolic representation of the covering set of this system

nication graphs, i.e., a process represented by a communication graph G can take all transitions of processes represented by the subgraphs of G. We call a set of graphs *depth-bounded*, if there exists a bound on the length of all simple paths in all graphs in the set. A *depth-bounded process* [80] is a process whose set of reachable communication graphs is depth-bounded. The subgraph isomorphism ordering is a well-quasi-ordering on sets of depth-bounded graphs, i.e., depth-bounded processes are WSTS. The process defined in Figure 3.1 is depth-bounded because the longest simple path in any of its reachable communication graphs has length at most 2. We now explain our analysis through this example.

Our analysis computes an over-approximation of the covering set of the analyzed WSTS, i.e., the downward-closure (with respect to the well-quasi-ordering) of its reachable set of states. The elements of the abstract domain of the analysis are the downward-closed sets. In our example, these are sets of communication graphs that are downward-closed with respect to the subgraph ordering. A finite downward-closed set of graphs can be represented by the maximal graphs in the set. The downward-closure of a single graph is an ideal of the subgraph ordering. Thus, any finite downward-closed set is a finite union of ideals. For well-quasi-orderings this is true for arbitrary downward-closed sets, including infinite ones. We symbolically represent the infinite ideals of the subgraph ordering by graphs where some subgraphs are marked with the symbol '*'. These markings of subgraphs can be nested. Such a symbolic graph represents the downward-closure of all graphs that result from (recursively) unfolding the marked subgraphs arbitrarily often. We call this type of symbolic graphs *nested graphs*. The right hand side of Figure 3.2 shows such a nested graph. It represents a downward-closed set of communication graphs of our example system that is also the covering set of the system. The covering set consists of all graphs that contain one server thread, one environment thread, and arbitrarily many clients with arbitrarily many unprocessed request and response messages each.

Our analysis works as follows: it starts with a set of nested communication graphs that represents the downward-closure of the initial states of the system. Then it iterates a fixed point functional that is composed of the following two steps: (1) compute the set of nested communication graphs that represent the

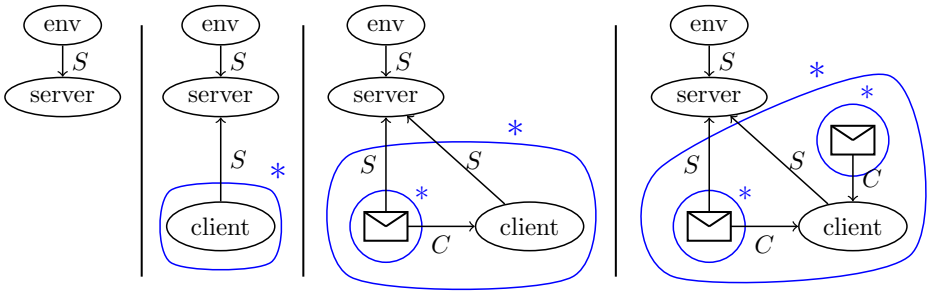


Figure 3.3: Sequence of symbolic communication graphs produced by the analysis of the system in Figure 3.1

downward-closure of the post states of the states represented by the current set of nested graphs, and (2) widen the resulting set of nested graphs with respect to the sequence of iterates that have been computed in the previous steps. The widening step compares the nested graphs in the new iterate pairwise to the nested graphs obtained in the previous iterates. If a nested graph in the new iterate is larger than some nested graph in a previous iterate then the larger graph must contain a subgraph that is not contained in the smaller one. This subgraph in the larger graph is then marked with a '*'. The intuition behind the widening is that, because of monotonicity of the transition relation, the sequence of transitions that lead from the smaller to the larger graph can be repeated arbitrarily often, which results in graphs with arbitrarily many copies of the new subgraph. Figure 3.3 shows a sequence of nested graphs obtained during the analysis of the client-server example. The final nested graph in the sequence represents the covering set of the system. This nested graph is also the fixed point obtained by our analysis, i.e., in this example the analysis does not lose precision.

Note that the covering set of our example system cannot be computed by a finite number of accelerations of finite sequences of transitions, i.e., the system is not flattable. This is reflected by the nesting of marked subgraphs in the nested graph that represents the covering set. To obtain this covering set via acceleration, one would need to compute the set of states reachable by a transfinite sequence of transitions resulting from ω-acceleration of a sequence of transition that is already infinite. The infinite sequence of transition that is to be accelerated corresponds to the creation of a client by the environment thread, followed by infinitely many exchanges of request and response messages between this client and the server. Since acceleration-based algorithms such as the clover algorithm [48] cannot accelerate infinite sequences of transitions, they do not terminate on our example system.

Ordinal numbers, Height of a poset and Cofinal sets. Throughout this part we will use ordinal numbers, and we assume the reader to be familiar with the basic aspect of their use. [50] can be consulted as reference. Furthermore, we use the less known natural sum and natural product of ordinals to bound the order type of the combination of ordered sets. The reader can consult [24] for the definitions of those operations.

Since we are working with poset rather than totally ordered set, we need

to introduce a few more terms. The *height* of a poset P is the ordinal number which has the same order type as the longest chain contained in P. We refer to the height of an element x, denoted $h(x)$, as the order type of the longest strictly ascending chain that finishes with x. The *width* of a poset is the size of its longest antichain. In the case of wqo, the width is finite.

In a set A equipped with a relation $\leq$, a subset B of A is *cofinal* iff for every $a \in A$ there exists a $b \in B$ such that $a \leq b$. The *cofinality* ($cf(A)$) of a poset A is the cardinality of the smallest cofinal subset of A. We also refer the reader to [38, 39] which provide good insights about the role of cofinal sets in well-quasi-ordered spaces.

3.3 Acceleration and Non-flattable Systems

We now show that the example of Figure 3.2 is not flattable. We investigate the order type of the chains that goes from the initial state (bottom) to the limit configurations of the cover (top). Furthermore, we are only interested in chain such that it is possible to get from one elements of the chain to the next one with a finite number of transition of the system. This restriction captures the kind of chains that an acceleration based algorithm explores (flat traces). Without this restriction it is possible to find a chain of order type ω by the cofinality of the space. First, we show that the longest chain in this example has order-type $\omega^2 \cdot 2$. Then, we show that any chain with the restriction mentioned above has at least order-type ω^2.

To convince ourself that the length of the longest chain is $\omega^2 \cdot 2$ we can simply show a chain of such length. Consider the chain of length ω that adds one client and then adds many requests. By repeating this chain ω times we can make a chain of length ω^2. To that chain we can append another chain of length ω that adds server's replies to one client. And, again, repeat this process for all the clients to obtain a chain of length $\omega^2 \cdot 2$. We still need to prove that there is no longer chain. Let observe that before adding any requests we must add the corresponding client first. This prevents the existence, for instance of a chain of length $\omega + 1$ where the requests are added before the client. The same reasoning can be done for the server's replies. The requests and the replies together gives a chain of at most $\omega \cdot 2$. Because of the ordering used (embedding) we know that the ordering of any sequence of state that include element of such a chain must the respect the ordering of that chain. In the state space pf the system there can be ω such chains of length $\omega \cdot 2$. By [24] we can bound any chain in the state space by $\omega^2 \cdot 2$.

To argue about the length of the shortest chain from bottom to top, we need to look at what are the successors of a configuration in the ordering. When we look at the transitions of the system, It is possible to add a client (transition of the env process), to add a request (transition of a client process), to consume a request and send a reply (transition of the server), or consume a reply (transition of the client). In order of getting a ω^2 lower bound on the length of a chain from bottom to top, we can ignore the replies from the server and the reception of requests and replies to focus only on the client and the requests. Hence we consider only two kind of transitions: adding a client and a client sending a request. We can first add all the clients which take ω operations. Then we need to add the messages to each clients. There is two ways of doing this. Either

adding a finite number of message to each clients and repeating until the system is saturated, or first saturating one clients with requests and then proceeding to the next client. In both cases $\omega \cdot \omega (= \omega^2)$ operations are required.

In Section 3.4 we show results about the height of state-spaces for Petri Nets, Lossy Channels Systems, and Depth-Bounded Processes. These observations help better understanding why some systems are necessarily non-flattable.

In Section 3.5 we briefly comment on another cause of non-flattability, the absence of iterated sequence.

3.4 Height of the State-spaces for PN, LCS, and DBP

To better understand the complexity of widening and the difference between acceleration and widening it is worth looking in more details at the shape of the state spaces of Petri nets, lossy-channel systems, and depth-bounded systems. One of the interesting questions is the height of the state space, i.e. what is the longest chain in the state space. Acceleration of simple loops is generating chains of length ω where there is a finite number of elements in the ordering between any two consecutive elements of the chains. Therefore, to guarantee convergence when the state space admits chains with length at least ω^2 we need to use a widening operator. For a PN with n places the height is $\omega \cdot n$, see Proposition 7. In the case of a DBP with depth n and e the number of equation defining the system, the height of the space is $\omega^{n+1} \cdot e$, see Proposition 9. For a LCS with n channels and m the number of different messages, we do not have a tight bound. Proposition 8 presents the upper and lower bound we found.

State space of Petri nets.

Proposition 7 *A PN with n places has a state space of the height $\omega \cdot n$.*

Proof. The state-space of a petri-net with n places is $\mathbb{N}^n$. We observe that the longest chain in $\mathbb{N}^n$ is a well-order in the disjoint union of $\mathbb{N} \uplus \mathbb{N} \uplus \dots$ (n times). This follows that fact that when the place p has x tokens there cannot be any element further in the chain with p having less than x tokens. By [24, Theorem 1], we know that order type of this chain is less than or equal to $\omega \cdot n$. We now prove that this bound is tight by showing a chain of that length. The lowest element in the ordering is when all the places are empty. The highest where all the places contains ∞ many tokens. A strictly ascending ω-chain takes at least one place that contains finitely many element to the limit (∞). Hence, there can by at most n such chain. Concatenating these chains gives the height: $\omega \cdot n$. □

State space of lossy-channel systems.

Proposition 8 *A LCS with n channels and m different messages has a state space with height between $\omega^m \cdot n + 1$ and $\Sigma_{i=0}^{m} \omega^{m-i} * n$.*

Proof.

Lower bound: To show the lower bound, we start by showing a chain over a *single* channel of length $\omega^m + 1$ going from ϵ (bottom) to $(\Sigma_{a\in\Sigma}\, a)*$ (top). The chains starts at ϵ, then add a_1 until it reaches a_1^*, then adds a_2 and goes back to adding a_1. When $(a_1^* a_2)^*$ is reached, the scheme is repeated with a_3, a_4, ... To get the length of this chain let consider the case of $|a_1^*| = \Sigma_0^\omega\, 1 = \omega$. Similarly, $|(a_1^* a_2)^*| = \Sigma_0^\omega(\omega + 1) = \omega^2 + 1$. For all the messages we get $\omega^m + 1$. Then repeating this process for each channel we get the lower bound of $\omega^m \cdot n + 1$.

Upper bound: Using natural operations we can easily derive an upper bound of $\Sigma_{i=0}^m\, \omega^{m-i} * n$.

□

State space of depth-bounded systems.

Proposition 9 *A DBP with depth n and where e is the number of equation defining the system has a state space height bounded by $\omega^{n+1} \cdot e$.*

Proof. The equations defining the systems are crucial in determining the state space. So we will show that there exists some equations which state-space can reach that bound. We can prove this by induction over n. We assumes that the equations defining the system are in a normal form where the equations starts by a prefix, see [9]. Hence, there are no equations without name parameter. Since the definition of DBP does not allows free name, we start our induction with $n = 1$.

$n = 1$: In this case there are top-level names defining scopes that contain processes referencing the names corresponding to the scope. The scopes do not intersect. Within a scope there is at most e different kinds of processes. Furthermore, there can be many disjoint scopes. We can start our chain by creating ω-scopes containing one element (empty scopes are not allowed). Then we can have an ω-chain to add one element to each scope. Hence, we get a chain of length ω^2 to saturate each scope with a single element. We can repeat this process for each of the equation label, i.e e times. This chain gives an height of $\omega^2 \cdot e$.

$n \to n + 1$: Increasing the nesting means that must be one or more processes referencing the top-level name. These processes may or may not already be in another scope. In the worst case there is all the names are at the bottom of the nesting and references all the $n+1$ names. No imagine that we restrict ourself to a single process, i.e. $e = 1$, and apply our induction hypothesis. Each scope of depth n gives a chain of length ω^{n+1}. We can have ω scopes of depth $n+1$, hence create a chain of length $\omega^{n+1}\omega = \omega^{n+2}$. Then we repeat this process e times to saturate the system with all the possible elements to obtain a chain of $\omega^{n+2} \cdot e$.

We now should prove that this bound is a real upper bound. In order to do so we shall use [24, Theorems 1-2] and induction over n.

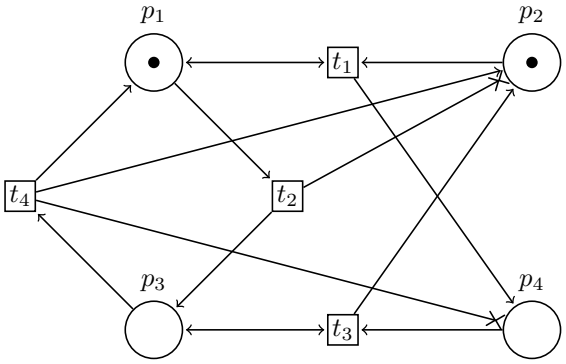


Figure 3.4: Reset net wit no iterated sequence.

$n = 1$: This case is equivalent to an ω-chain of PN with e places. The order type of that chain can be seen as the natural product of ω and $\omega \cdot e$. Hence the bound is $\omega^2 \cdot e$.

$n \to n + 1$: We can use the same reasoning as the part about reaching the bound which amounts to taking the natural product between ω and $\omega^{n+1} \cdot e$. This proves that the bound is $\omega^{n+2} \cdot e$.

□

Remark 2 *A tight bound can be directly derived from [24, Theorems 1-2]. However, we feel that showing explicitly a chain that reaches the bound gives a better grasp of what the state-space looks like.*

3.5 WSTS without iterated sequence

Even when the state-space has an height less than ω^2 algorithms based on acceleration may not finish. The reason is that the procedure finishes only if the cover can be generated using a finite number of acceleration, i.e. the system has a finite number of behaviors. This fact is captured by the flattability condition [15]. Only flat (without nested loop) or flattable system can be analyzed by acceleration. For instance, nested loops can give rise to longer and longer sequences of transitions, leading to divergence of the algorithm. Figure 3.4 presented in [41] shows a reset net with such a behaviour. To get from $(1, i, 0, 0)$ to $(1, i + 1, 0, 0)$ the firing sequence is $t_1^i t_2 t_3^i t_4$. Hence, getting to $(1, \omega, 0, 0)$ requires the firing of infinitely many different sequences. Even though, flattable systems are very common [76], it is undecidable to know if a system is flattable a priori [27].

3.6 Ideal Abstraction

We next describe our abstract interpretation framework for computing over-approximations of the covering sets of WSTS. For this purpose we fix a WSTS $T = (S, S_0, \to, \leq)$ throughout the rest of this section.

3.6.1 Concrete and Abstract Domain

Following the framework of abstract interpretation [32, 31], a static analysis is defined by lattice-theoretic domains and by fixed point iteration over the domains. The concrete domain $\mathcal{D}$ of our analysis is the powerset domain over the states S of WSTS T:

$$\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(S)(\subseteq, \emptyset, S, \cup, \cap)$$

Since our analysis is to compute an over-approximation of the covering set of T, which is a downward-closed set, we define the abstract domain $\mathcal{D}_\downarrow$ as the set of all downward-closed subsets of S, again ordered by subset inclusion:

$$\mathcal{D}_\downarrow \stackrel{\text{def}}{=} \{\, {\downarrow}X \mid X \subseteq S \,\}\,(\subseteq, \emptyset, S, \cup, \cap)$$

One can easily verify that $\mathcal{D}_\downarrow$ is a complete lattice. This choice of the abstract domain suggests the following abstraction function $\alpha_\downarrow : \mathcal{D} \to \mathcal{D}_\downarrow$ and concretization function $\gamma_\downarrow : \mathcal{D}_\downarrow \to \mathcal{D}$ defined as $\alpha_\downarrow(X) \stackrel{\text{def}}{=} {\downarrow}X$ and $\gamma_\downarrow(Y) \stackrel{\text{def}}{=} Y$.

Proposition 10 *The pair $(\alpha_\downarrow, \gamma_\downarrow)$ forms a Galois insertion between domains $\mathcal{D}$ and $\mathcal{D}_\downarrow$.*

According to [31], the Galois insertion $(\alpha_\downarrow, \gamma_\downarrow)$ defines the *best abstract post operator* $\mathsf{post}_\downarrow$ on the abstract domain $\mathcal{D}_\downarrow$:

$$\mathsf{post}_\downarrow \stackrel{\text{def}}{=} \alpha_\downarrow \circ \mathsf{post} \circ \gamma_\downarrow$$

We next show that we can effectively represent the elements of $\mathcal{D}_\downarrow$ and, for all practical purposes, effectively compute $\mathsf{post}_\downarrow$ on this representation. To obtain this representation, we exploit the fact that any downward-closed subset of a wqo-set $S(\leq)$ is a finite union of ideals of $S(\leq)$.

Denote by $\mathcal{P}_{\text{fin}}(\mathit{Idl}(S))$ the finite sets of ideals of $S(\leq)$ and define the quasi-ordering $\sqsubseteq$ on $\mathcal{P}_{\text{fin}}(\mathit{Idl}(S))$ as the point-wise extension of $\subseteq$ from the ideal completion $\mathit{Idl}(S)$ of $S(\leq)$ to $\mathcal{P}_{\text{fin}}(\mathit{Idl}(S))$:

$$L_1 \sqsubseteq L_2 \stackrel{\text{def}}{\iff} \forall I_1 \in L_1.\, \exists I_2 \in L_2.\, I_1 \subseteq I_2$$

Let $\mathcal{D}_{\mathit{Idl}}$ be the quotient of $\mathcal{P}_{\text{fin}}(\mathit{Idl}(S))$ with respect to the equivalence relation $\sqsubseteq \cap \sqsubseteq^{-1}$. For notational convenience we use the same symbol $\sqsubseteq$ for the quasi-ordering on $\mathcal{P}_{\text{fin}}(\mathit{Idl}(S))$ and the partial ordering that it defines on the quotient $\mathcal{D}_{\mathit{Idl}}$. We further identify the elements of $\mathcal{D}_{\mathit{Idl}}$ with the finite sets of maximal ideals, i.e., for all $L \in \mathcal{D}_{\mathit{Idl}}$ and $I_1, I_2 \in L$, if $I_1 \subseteq I_2$ then $I_1 = I_2$.

Now, define the function $\gamma_{\mathit{Idl}} : \mathcal{D}_{\mathit{Idl}} \to \mathcal{D}_\downarrow$ as $\gamma_{\mathit{Idl}}(L) \stackrel{\text{def}}{=} \bigcup L$.

Proposition 11 *The function γ_{Idl} is an order-isomorphism.*

Proof. Every downward-closed subset of a wqo-set $S(\leq)$ is a finite union of ideals from $\mathit{Idl}(S)$ (this follows, e.g., from [38, Theorem 5] or from [47, Proposition 3.2 and Proposition 3.3]). Thus γ_{Idl} is surjective.

Assume γ_{Idl} is not injective. Then there exist distinct $L_1, L_2 \in \mathcal{D}_{\mathit{Idl}}$ such that $\gamma_{\mathit{Idl}}(L_1) = \gamma_{\mathit{Idl}}(L_2)$. Since $L_1 \neq L_2$, either $L_1 \not\sqsubseteq L_2$ or $L_2 \not\sqsubseteq L_1$. Assume wlog. that $L_1 \not\sqsubseteq L_2$. Then there exists $I_1 \in L_1$ such that for all $I_2 \in L_2$, $I_1 \not\subseteq I_2$.

Hence there exists $x \in I_1$ such that $x \notin \bigcup L_2$. It follows $\bigcup L_1 \neq \bigcup L_2$, which is a contradiction. Thus, γ_{Idl} is injective.

Monotonicity of γ_{Idl} follows immediately from the definitions of γ_{Idl} and $\sqsubseteq$. For proving that the inverse of γ_{Idl} is monotone, let $X_1, X_2 \in \mathcal{D}_{\downarrow}$ such that $X_1 \subseteq X_2$. We first prove a small lemma: let $L \in \mathcal{D}_{Idl}$ and $I \in Idl(S)$ such that $I \subseteq \bigcup L$ and $I \neq \emptyset$. Let n be the cardinality of L. Since I is nonempty, so must be L, i.e., $n \geq 1$. We prove by induction on n that there exists $J \in L$ such that $I \subseteq J$. Choose arbitrarily some $J \in L$ then $I \subseteq J \cup \bigcup(L \setminus \{J\})$. Since both J and $\bigcup(L \setminus \{J\})$ are downward-closed and I is directed, it follows that either $I \subseteq J$ or $I \subseteq \bigcup(L \setminus \{J\})$; otherwise there would exist distinct $x, y \in I$ such that $x \in J$ but not in $I \subseteq \bigcup(L \setminus \{J\})$, and vice versa. The fact that both of these sets are downward-closed then would imply that x and y have no common upper bound in I, contradicting that I is directed. Now, if $I \subseteq J$ holds then we are done. Thus assume $I \not\subseteq J$. Then $n > 1$ because I is non-empty. From the induction hypothesis follows that there exists $J' \in L \setminus \{J\}$ such that $I \subseteq J'$. Since $J' \in L$, this concludes the prove of the lemma. Using this lemma we can easily conclude that $\gamma_{Idl}^{-1}(X_1) \sqsubseteq \gamma_{Idl}^{-1}(X_2)$, i.e., γ_{Idl}^{-1} is monotone.

It follows that γ_{Idl} is an order-isomorphism between $\mathcal{D}_{Idl}$ and $\mathcal{D}_{\downarrow}$. □

Let $\sqcup$ and $\sqcap$ be the least upper bound and greatest lower bound operators on the poset $\mathcal{D}_{Idl}(\sqsubseteq)$. These operators exist because $\mathcal{D}_{\downarrow}$ is a complete lattice and $\mathcal{D}_{\downarrow}$ and $\mathcal{D}_{Idl}$ are order-isomorphic according to Proposition 11. The following proposition then follows immediately.

Proposition 12 *$\mathcal{D}_{Idl}(\sqsubseteq, \emptyset, \{S\}, \sqcup, \sqcap)$ is a complete lattice.*

Let $\alpha_{Idl} : \mathcal{D}_{\downarrow} \to \mathcal{D}_{Idl}$ be the inverse of γ_{Idl}. Since γ_{Idl} is an order-isomorphism, the pair $(\alpha_{Idl}, \gamma_{Idl})$ forms a Galois insertion between $\mathcal{D}_{\downarrow}$ and $\mathcal{D}_{Idl}$.

Let $\alpha = \alpha_{Idl} \circ \alpha_{\downarrow}$ and $\gamma = \gamma_{\downarrow} \circ \gamma_{Idl}$. Then (α, γ) forms a Galois insertion between concrete domain $\mathcal{D}$ and abstract domain $\mathcal{D}_{Idl}$. Let $\mathsf{post}_{Idl} = \alpha \circ \mathsf{post} \circ \gamma$ be the induced best abstract post operator on $\mathcal{D}_{Idl}$ and let F_{Idl} be the function $F_{Idl} = \lambda L.\, \alpha(S_0) \sqcup \mathsf{post}_{Idl}(L)$. The following proposition is then a simple consequence of Proposition 11.

Proposition 13 *The least fixed point of F_{Idl} on $\mathcal{D}_{Idl}$ is the covering set of T:*

$$\gamma(\mathrm{lfp}^{\sqsubseteq}(F_{Idl})) = \mathrm{Cover}(T) \ .$$

Can we compute $\mathrm{lfp}^{\sqsubseteq}(F_{Idl})$? In general the answer is "no" for various reasons. First, we may not be able to compute the iterates of the abstract functional F_{Idl}, respectively, decide the fixed point test on the abstract domain. However, for the classes of WSTS that are of practical interest, this is not a problem: We say that the ideal completion $Idl(S)$ of a WSTS $T = (S, S_0, \to, \leq)$ is *effective* if (i) for all $I_1, I_2 \in Idl(S)$, checking $I_1 \subseteq I_2$ is decidable, and (ii) for all $I \in Idl(S)$, $\mathsf{post}_{Idl}(\{I\})$ is computable. It follows from [47, Theorem 3.4] that all WSTS with a so called *effective adequate domain of limits* [52] also have an effective ideal completion. Classes of WSTS with this property include, e.g., Petri nets and their monotone extensions [52], lossy channel systems [47], and depth-bounded processes, cf. Chapter 2.

Thus, assume that T has an effective ideal completion. Then, for any $L \in \mathcal{D}_{Idl}$ we can compute $F_{Idl}(L)$ and decide $F_{Idl}(L) \sqsubseteq L$. However, this is not

yet sufficient for guaranteeing termination. In general, the covering set of a WSTS is not computable, i.e., we cannot expect that the sequence of iterates $(\bigsqcup_{i \leq n} F^i_{Idl}(\emptyset))_{n \in \mathbb{N}}$ stabilizes. In fact, even if the exact covering set $\mathrm{Cover}(T)$ is computable for a particular WSTS, the sequence of fixed point iterates might not stabilize because the abstract domain $\mathcal{D}_{Idl}$ has (typically) infinite height. To ensure termination of our analysis, we next define an appropriate widening operator for the abstract domain $\mathcal{D}_{Idl}$.

3.6.2 Widening

Let us first recall the notion of set-widening operators [33]. A *set-widening operator* for a poset $X(\leq)$ is a partial function $\nabla : \mathcal{P}(X) \rightharpoonup X$ that satisfies the following two conditions:

- *Covering*: For all $Y \subseteq X$, if $\nabla(Y)$ is defined then for all $y \in Y$, $y \leq \nabla(Y)$.
- *Termination*: For every ascending chain $\{x_i\}_{i \in \mathbb{N}}$ in $X(\leq)$, the sequence $y_0 = x_0$, $y_i = \nabla(\{x_0, \ldots, x_i\})$, for all $i > 0$, is well-defined and an ascending stabilizing chain.

In the following, we define a general set-widening operator for the abstract domain $\mathcal{D}_{Idl}$. The reason for using a set-widening operator instead of the more popular pair widening operator is that we want to enable the widening operator to take into account the whole history of the previous iterates of the fixed point computation. This allows us to use widening to mimic the effect of acceleration for computing the exact covering set of flattable WSTS.

The set-widening operator on the abstract domain $\mathcal{D}_{Idl}$ is obtained by lifting a given set-widening operator for the ideal completion $Idl(S)$. This underlying widening operator on ideals is a parameter of the analysis because it is domain-specific for each class of WSTS. In the next section, we will describe several such widening operators for common classes of WSTS.

In general, extending a widening operator from a base domain to its finite powerset is non-trivial [11]. We can simplify this task by making a stronger assumption about the ordering $\leq$ on the base set S: we assume that $S(\leq)$ is not just a wqo, but a bqo. This ensures that the ideal completion $Idl(S)$ is itself a bqo with respect to the subset inclusion ordering. Using this fact we can then lift the set-widening operator on ideals to sets of ideals. From a practical point of view, requiring a bqo is not a real restriction, since all wqos of WSTS occurring in practice are actually bqos.

Assume that ∇_S is a set-widening operator on the poset $Idl(S)(\subseteq)$. Then define the operator $\nabla : \mathcal{P}(\mathcal{D}_{Idl}) \rightharpoonup \mathcal{D}_{Idl}$ as follows: for $C \subseteq \mathcal{D}_{Idl}$, if C is a finite ascending chain $C = \{L_i\}_{0 \leq i \leq n}$ in $\mathcal{D}_{Idl}(\sqsubseteq)$ let $\nabla(C)$ be defined recursively by

$$\begin{aligned} \nabla(\{L_0\}) &= L_0 \\ \nabla(\{L_0, \ldots, L_i\}) &= \nabla(\{L_0, \ldots, L_{i-1}\}) \sqcup \\ &\quad \{\, \nabla_S(\mathcal{I}) \mid \mathcal{I} \text{ maximal ascending chain in } \nabla(\{L_0, \ldots, L_{i-1}\}) \,\} \end{aligned}$$

for all $0 < i \leq n$. In all other cases let $\nabla(C)$ be undefined.

Proposition 14 *If $S(\leq)$ is a bqo then ∇ is a set-widening operator for $\mathcal{D}_{Idl}(\sqsubseteq)$.*

Proof. Proving the covering property is easy. We thus focus on the termination property. Let $C = \{L_i\}_{i\in\mathbb{N}}$ be an infinite ascending chain in $\mathcal{D}_{Idl}(\sqsubseteq)$. For deriving a contradiction, assume that the ascending chain $\{W_i\}_{i\in\mathbb{N}}$ defined by $W_0 = L_0$, $W_{i+1} = \nabla(\{L_0, \ldots, L_{i+1}\})$ is not stabilizing. Since the chain $\{W_i\}_{i\in\mathbb{N}}$ is strictly increasing it follows from the definition of $\sqsubseteq$ that for every $i \geq 1$ there exists $I_i \in W_i$ such that for all $I \in W_{i-1}$, $I_i \not\subseteq I$. Thus consider the sequence $\{I_i\}_{i\geq 1}$. Then we have by transitivity of $\sqsubseteq$ that for all $i > 1$, $1 \leq j < i$, and $I \in W_j$, $I_i \not\subseteq I$. Since $S(\leq)$ is a bqo, so are $\mathcal{P}(S)(\subseteq)$ and $Idl(S)(\subseteq)$. From the properties of wqos and thus bqos follows that the sequence $\{I_i\}_{i\geq 1}$ contains a subsequence $\{I_{i_k}\}_{k\in\mathbb{N}}$ with $i_k < i_{k+1}$ for all $k \in \mathbb{N}$, such that $\{I_{i_k}\}_{k\in\mathbb{N}}$ is an ascending chain in $Idl(S)(\subseteq)$. Since ∇_S is a set-widening operator for $Idl(S)(\subseteq)$, it follows from the termination property that the sequence defined by $J_0 = I_{i_0}$, $J_{k+1} = \nabla_S(\{I_{i_0}, \ldots, I_{i_k}\})$ is an ascending stabilizing chain. Let j be an index where the chain has stabilized, i.e., $J_j = J_{j+1}$. By definition of ∇, we have that for all $k \in \mathbb{N}$, $J_k \in W_{i_k}$. In particular, $J_j \in W_{i_j}$. The covering property of ∇_S then implies that $I_{i_{j+1}} \subseteq J_{j+1} = J_j \in W_j$, which is a contradiction. □

We now define our analysis in terms of the widening sequence $\{W_i\}_{i\in\mathbb{N}}$ as follows:

$$W_0 = \emptyset \qquad \textit{and} \qquad W_{i+1} = \nabla(\{W_0, \ldots, W_i, F_{Idl}(W_i) \sqcup W_i\})$$

Note that for computing the image of ∇ in step $i + 1$ we can reuse W_i. The properties of set-widening operators, Proposition 13, and Proposition 14 imply the soundness and termination of the analysis.

Theorem 4 *If $S(\leq)$ is a bqo then the sequence $\{W_i\}_{i\in\mathbb{N}}$ stabilizes and its least upper bound approximates the covering set of T, i.e., $Cover(T) \subseteq \gamma(\bigcup\{W_i\}_{i\in\mathbb{N}})$.*

Trace Partitioning. Note that, unlike acceleration, the widening operator ∇ does not take into account whether each widened chain of ideals is actually correlated by some sequence of transition in the system. This incurs an additional loss of precision that is not needed to ensure termination of the analysis. To avoid this loss of precision, we can refine the above analysis via combination with an appropriate trace partitioning domain [103]. The resulting analysis is a generalized Karp-Miller tree construction where acceleration has been replaced by widening.

3.7 Set-Widening Operators for Ideal Completions

We now discuss several instantiations of our analysis for different classes of WSTS by presenting the corresponding ideal completions and set-widening operators on ideals. We discuss, in turn, Petri nets, lossy channel systems, and depth-bounded processes.

3.7.1 Petri Nets

For the complete definition of Petri nets we refer the reader to Section 1.3.2. Furthermore, the pointwise ordering of markings is a bqo [85]. The ideal completion $Idl(\mathcal{M}(S))$ of the markings of a Petri net can be represented by extended markings, which are functions $S \to \mathbb{N} \cup \{\omega\}$ [58]. The ordering on extended markings is given by $M \leq M'$ iff for all $s \in S$, $M'(s) = \omega$ or $M(s) \in \mathbb{N}$ and $M(s) \leq M'(s)$.

Widening for Petri Nets. The set-widening operator ∇_{PN} for a Petri Net corresponds to the usual acceleration used in the Karp-Miller tree construction for Petri nets. For a finite ascending chain $\{M_i\}_{0 \leq i \leq n}$ we define $\nabla_{\mathsf{PN}}(\{M_i\}_{0 \leq i \leq n}) = M$ where $M(s) = \omega$ if $M_n(s) > M_0(s)$ and $M_n(s)$ otherwise. Clearly this set-widening operator satisfies the covering condition. It also satisfies termination, since the set of places S is finite.

Precision of the Widening and Monotonic Extensions of Petri Nets. For standard Petri nets the above widening operator corresponds to the acceleration used in the Karp-Miller tree construction. In fact, for this class of WSTS our analysis does not lose precision. The reason is that in Petri nets sequences of firing transitions σ that increase the value of a marking M by some δ, $\sigma(M) = M + \delta$, do the same for all larger markings $M' \geq M$, i.e., $\sigma(M') = M' + \delta$.

For monotonic extensions of Petri nets the situation is more complicated. Monotonic extensions of Petri nets include transfer nets and reset nets. In a reset net a transition can consume all the tokens present in one place in a single step. In a transfer net a transition can transfer all the tokens from one place to another place in a single step. In both cases we can use the same widening as for standard Petri nets, but the analysis may lose precision because neither transfer nets nor reset nets are flattable, in general. However, for a concrete net the loss of precision does not depend on the flattability of the net in consideration, i.e., there are non-flattable nets where the result of the analysis is exact and flat nets were the analysis over-approximates the actual covering set.

3.7.2 Lossy Channel Systems

A *lossy channel system* (LCS) [5] is a tuple (S, s_0, C, M, δ) where S is a finite set of control locations, s_0 is the initial location, C is a finite set of channels, M is a finite set of messages, and δ is a set of transitions. A state of an LCS is a tuple (s, w) where $s \in S$ and w is a mapping $C \to M^*$ denoting the content of the channels. A transition t is a tuple (s_1, Op, s_2) where $s_1, s_2 \in S$ and Op is of the form $c\,!/?\,m$ ($c \in C$,$m \in M$). The system can go from state (s_1, w_1) to (s_2, w_2) by firing transition t iff $Op = c!m \wedge w_2(c) \leq w_1(c)m$ or $Op = c?m \wedge mw_2(c) \leq w_1(c)$, the remaining channels are unchanged. The systems are called *lossy* because messages can be dropped from channels before and after performing a send or receive operation. The ordering on states $\leq$ is defined as $(s, w) \leq (s', w')$ iff $s = s'$ and for all $c \in C$, $w(c)$ is a subword of $w'(c)$. The subword ordering is a bqo [85] and thus so is the ordering $\leq$ on states. In the following we describe a widening on the content of individual channels. Its extension to states is defined as expected.

The downward-closed sets of the subword ordering are exactly the languages of simple regular expressions (SRE) [4], which are defined by the following grammar:

$$\begin{array}{rcl} \mathit{atom} & ::= & (m+\epsilon) \mid (m_1 + \ldots + m_n)^* \\ \mathit{product} & ::= & \epsilon \mid \mathit{atom}\ \mathit{product} \\ \mathit{SRE} & ::= & \mathit{product}\ [\ +\ \mathit{SRE}] \end{array}$$

The ideals of the subword ordering are the languages denoted by the products in SRE. The ordering on the ideals is language inclusion.

Widening for LCS. The first step in defining the widening operator on channel contents is to define a notion of difference on the corresponding ideals. For a product p we denote by $|p|$ the number of atoms appearing in p and for $1 \leq i \leq |p|$ we denote by $p[i]$ the ith atom of p.

Let p, q be products. If $p \leq q$ then we can find a mapping $\iota : [1, |p|] \to [1, |q|]$ such that (i) ι is monotone, i.e., for all $i, j \in [1, |p|]$ if $i \leq j$ then $\iota(i) \leq \iota(j)$, (ii) for all $i \in [1, |p|]$ the language of $p[i]$ is included in the language of $q[\iota(i)]$, and (iii) for all $i, j \in [1, |p|]$ if $\iota(i) = \iota(j)$ and $q[\iota(i)]$ is of the form $(a + \epsilon)$ then $i = j$. We call ι an *inclusion mapping* for $p \leq q$. Note that we consider an interval $[l, r]$ to be empty if $l > r$, i.e., if $p = \epsilon$ then the inclusion mapping exists trivially.

Let p and q be atoms such that $p \leq q$ and let ι be an inclusion mapping for $p \leq q$. We define an extrapolation operator χ_{LCS} for p, q and ι as follows. If $p = \epsilon$ then $\chi_{\mathsf{LCS}}(p, q, \iota) = (\sum_i q[i])^*$. Otherwise, let $i_1, \ldots, i_n$ be the increasing sequence of indices in the range of ι. For each $j \in [1, n-1]$ define the interval $d_j = [i_j + 1, i_{j+1} - 1]$. Furthermore, define $d_0 = [1, i_1 - 1]$ and $d_n = [i_n, |q|]$. For all $j \in [0, n]$, define $s_j = \left(\sum_{i \in d_j} q[i]\right)^*$. Note that s_j is equivalent to ϵ if d_j is empty and, otherwise, s_j is equivalent to an atom of the form $\left(\sum_k m_k\right)^*$ where the m_k are the messages appearing in the atoms $q[i]$ for $i \in d_j$. Then define $\chi_{\mathsf{LCS}}(p, q, \iota) = s_0\, q[i_1] \ldots s_{k-1}\, q[i_k]\, s_k$.

Inclusion mappings are not necessarily unique. We therefore fix for each ascending sequence of products $p_1 \leq p_2 \ldots$ a corresponding sequence $\iota_1, \iota_2, \ldots$ such that (1) for all i, ι_i is an inclusion mapping for $p_i \leq p_{i+1}$, and (2) for every two ascending chains of products that share a common prefix, the corresponding sequences of inclusion mappings agree on this prefix.

Let $\pi = \{p_i\}_{0 \leq i \leq n}$ be an ascending chain of products with $n > 0$. The set-widening of π is then defined as $\nabla_{\mathsf{LCS}}(\pi) = \chi_{\mathsf{LCS}}(p_0, p_n, \iota_{0,n})$ where $\iota_{0,n}$ is the composition of the fixed sequence of inclusion mappings for π, $\iota_{0,n} = \iota_{n-1} \circ \cdots \circ \iota_0$.

Theorem 5 *∇_{LCS} is a set widening operator.*

Proof. The covering requirement for ∇_{LCS} follows easily from its definition. In the following, we prove the termination requirement. Let $p_0 \leq p_1 \ldots$ be an ascending chain of products and let $\iota_0, \iota_1, \ldots$ be the associated sequence of inclusion mappings. First, note that if ι is an inclusion mapping for $p \leq q$ and ι' is an inclusion mapping for $q \leq r$ then $\iota' \circ \iota$ is an inclusion mapping for $p \leq r$. Thus, using induction on n, we can easily prove that for all $n \geq 1$, $\iota_{0,n}$ is an inclusion mapping for $p_0 \leq p_n$, i.e., $\chi_{\mathsf{LCS}}(p_0, p_n, \iota_{0,n})$ is well-defined. Hence,

define the sequence $\{y_n\}_{n\in\mathbb{N}}$ as $y_0 = p_o$ and $y_n = \chi_{\mathsf{LCS}}(p_0, p_n, \iota_{0,n})$ for all $n > 0$. Using the definition of χ_{LCS} and the ι_n one can easily construct an inclusion mapping between y_n and y_{n+1}, for any $n \in \mathbb{N}$. It follows that the sequence $\{y_n\}_{y\in\mathbb{N}}$ is an ascending chain. Furthermore, from the definition of χ_{LCS} it follows immediately that for all $n \geq 0$, $|y_n| \leq 2|p_0| + 1$. Since the number of possible atoms is finite (for a fixed set of messages M), the number of atoms in products of the sequence is bounded, and the sequence is ascending, it follows that the sequence must be stabilizing. □

Note that one cannot use the operator χ_{LCS} to define a standard pair widening operator ∇ on ideals of the subword ordering: $\nabla(p, q) = \chi_{\mathsf{LCS}}(p, q, \iota)$ where ι is an inclusion mapping for $p \leq q$. As a counterexample for termination of this operator consider the following sequence of ideals: $x_0 = \epsilon$, $x_1 = (a + \epsilon)$, $x_2 = a^*(b + \epsilon)$, $x_3 = a^*b^*(a + \epsilon)$, etc. Applying ∇ pairwise on consecutive elements of the sequence leads to the following diverging sequence: $y_0 = x_0 = \epsilon$, $y_1 = \nabla(y_0, x_1) = a^*$, $y_2 = \nabla(y_1, x_2) = a^*b^*$, $y_3 = \nabla(a^*b^*, x_3) = a^*b^*a^*$, etc. On the other hand, the set-widening operator ∇_{LCS} produces the stabilizing sequence: $y_0 = x_0 = \epsilon$, $y_1 = \nabla_{\mathsf{LCS}}(\{x_0, x_1\}) = a^*$, $y_2 = \nabla_{\mathsf{LCS}}(\{x_0, x_1, x_2\}) = (a + b)^*$, $y_3 = \nabla_{\mathsf{LCS}}(\{x_0, x_1, x_2, x_3\}) = (a + b)^*$, etc. For termination, it is crucial that the maximal length of the products provided as first argument of χ_{LCS} is bounded throughout all widening steps. This is for instance ensured by fixing the first argument of χ_{LCS} to one particular element of the widened sequence (e.g., the first element as in the definition of ∇_{LCS}). However, one can also define more precise set-widening operators than ∇_{LCS}. For instance, one may define a set-widening operator ∇_k that is parameterized by a natural number k as follows: $\nabla_k(\{p_0, \ldots, p_n\}) = \chi_{\mathsf{LCS}}(q, p_n, \iota)$ where q is the largest product in $\{p_0, \ldots, p_n\}$ with at most k atoms, and ι the inclusion mapping for $q \leq p_n$. The termination proof for ∇_k closely follows the proof of termination for ∇_{LCS}.

The intuition for the termination of the widening is that the number of distinct symbols from the set of messages M that appear in the atoms contained in the widened sequence of ideals increases monotonically. Since M is finite, the sequence stabilizes, at the latest with the ideal consisting of the single atom $\left(\sum_{m\in M} m\right)^*$.

3.7.3 Depth-Bounded Processes

Depth bounded systems were introduced in Section 1.3.3. In Chapter 2 that the covering problem is decidable for this class. As for many other classes of WSTS, the problem has non-primitive recursive complexity. This makes depth-bounded systems a particularly interesting target for approximative analyses.

In Chapter 2 we have shown that the ideals of the ordering on depth-bounded configurations can be represented by extending process terms with a replication operator ! to encode that certain subprocesses may be repeated arbitrarily often. We call these terms *limit process terms*. For instance the covering set of the example discussed in Section 3.2 is denoted by the following limit process term:

$$(\nu S)(\text{server}(S) \mid \text{env}(S) \mid !(\nu C)(\text{client}(C, S) \mid !(\overline{S}(C).0) \mid !(C().0)))$$

Remark 3 *In the definition of depth-bounded systems the ordering refers to the set of reachable states. This conflicts with the fact that we computing an*

$$\frac{}{\vec{x}, P, 0 \vdash P \leq P}\ \textsc{Eq}$$

$$\frac{\vec{x}, P, F \vdash P \leq Q}{\vec{x}, P, F \mid F' \vdash P \leq Q \mid F'}\ \textsc{|-Frame}$$

$$\frac{\vec{x}, R, F \vdash P \leq !Q}{\epsilon, !P, !((\nu\vec{x})(R \mid F)) \vdash !P \leq !Q}\ \textsc{!-Frame}$$

$$\frac{\vec{x}\vec{x'}, R, F \vdash P \leq Q}{\vec{x}z\vec{x'}, R, F \vdash (\nu z)P \leq (\nu z)Q}\ \nu\textsc{-Subtr}$$

$$\frac{\vec{x}, R, F \vdash P \leq Q}{\vec{x}, R \mid S, F \vdash P \mid S \leq Q \mid S}\ \textsc{|-Subtr}$$

$$\frac{\vec{x}, R, F \vdash P' \leq Q' \quad P \equiv P' \quad Q \equiv Q'}{\vec{x}, R, F \vdash P \leq Q}\ \textsc{Struct}$$

Figure 3.5: Anti-frame inference rules

overapproximation of the covering set. During the analysis, we might explore non-reachable states. Therefore, the ordering is not guaranteed to be a bqo. In this section, we assume that we know the depth of the systems or at least have an upper bound. This is required to guarantee the termination of the analysis. In practice, we have implemented the analysis without knowledge of the bound. Even though, we know how to make examples where the abstraction leads to a non-terminating analysis we have never encountered such example in practice.

Widening for depth-bounded systems. We first define an extrapolation operator χ_{DBP} on pairs of limit process terms, which we then lift to a set-widening operator ∇_{DBP}. The extrapolation operator relies on a set of inference rules for checking validity of clauses of the form $P \leq Q$ where P, Q are limit process terms. The inference rules do not just prove $P \leq Q$ but do a bit more: given P and Q, the rules derive judgments of the form $\vec{x}, R, F \vdash P \leq Q$. The semantics is that if $\vec{x}, R, F \vdash P \leq Q$ can be derived then $(\nu\vec{x})R \equiv P$ and $(\nu\vec{x})(R \mid F) \equiv Q$. We call F an *anti-frame*[1] of $P \leq Q$. The anti-frame captures the difference between process terms P and Q and is crucial for defining the widening. The *anti-frame inference* relation $(\nu\vec{x})(R \mid F) \equiv Q$ is defined by the rules given in Figure 3.5.

Proposition 15 *Let R, F, P, and Q be process terms and let $\vec{x}$ be a list of names then $\vec{x}, R, F \vdash P \leq Q$ iff $(\nu\vec{x})R \equiv P$ and $(\nu\vec{x})(R \mid F) \equiv Q$.*

Proof. The left-to-right direction is proved by induction on the rules defining the relation $\vec{x}, R, F \vdash P \leq Q$. The case for rule |-Frame is trivial. The case for rule !-Frame follows from idempotence of ! and the fact that $\equiv$ is a congruence relation. The case for rule ν-Subtr follows again from the fact that $\equiv$ is a congruence relation and the axiom for reordering of restricted names. The case

[1] We use the term "anti-frame" because of the similarity to abduction in entailment provers for separation logic [23].

$$\frac{}{\vec{x}, P, 0 \triangleright P \leq P}\ \textsc{Eq}$$

$$\frac{\vec{x}, P, F \triangleright P \leq Q}{\vec{x}, P, F \mid !F' \triangleright P \leq Q \mid F'}\ \textsc{|-Extrapolate}$$

$$\frac{\vec{x}, R, F \triangleright P \leq !Q}{\epsilon, !P, !(\nu\vec{x})(R \mid F) \triangleright !P \leq !Q}\ \textsc{!-Frame}$$

$$\frac{\vec{x}\vec{x'}, R, F \triangleright P \leq Q}{\vec{x}z\vec{x'}, R, F \triangleright (\nu z)P \leq (\nu z)Q}\ \nu\textsc{-Subtr}$$

$$\frac{\vec{x}, R, F \triangleright P \leq Q}{\vec{x}, R \mid S, F \triangleright P \mid S \leq Q \mid S}\ \textsc{|-Subtr}$$

$$\frac{\vec{x}, R, F \triangleright P' \leq Q' \quad P \equiv P' \quad Q \equiv Q'}{\vec{x}, R, F \triangleright P \leq Q}\ \textsc{Struct}$$

Figure 3.6: Extrapolation rules

for rule |-Subtr follows directly from the fact that $\equiv$ is a congruence relation. Finally, the case for rule Struct follows from the fact that $\equiv$ is an equivalence relation.

For proving the other direction assume $(\nu\vec{x})(R \mid F) \equiv Q$ and $(\nu\vec{x})R \equiv P$. Using rule |-frame we infer $\epsilon, R, F \vdash R \leq R \mid F$. Repeated application of rule ν-Subtr then gives $\vec{x}, R, F \vdash (\nu\vec{x})R \leq (\nu\vec{x})(R \mid F)$. Using the assumption together with rule Struct finally gives $\vec{x}, R, F \vdash P \leq Q$. □

We now define the extrapolation operator χ_{DBP}. First, we modify the anti-frame inference rules to get the extrapolation rules of Figure 3.6. The modification is carefully done in a way that if $P \leq Q$ then there exist some $(R, F, \vec{x})$ such that $\vec{x}, R, F \triangleright P \leq Q$. This essentially follows from Proposition 15. We fix one such $(R, F, \vec{x})$ for each $P \leq Q$. The extrapolation $\chi_{\mathsf{DBP}}(P, Q)$ is then defined as: $\chi_{\mathsf{DBP}}(P, Q) = (\nu\vec{x})(R \mid F)$.

Lemma 10 *Let P, Q be limit process terms such that $P \leq Q$. Then $P \leq \chi_{\mathsf{DBP}}(P, Q)$ and $Q \leq \chi_{\mathsf{DBP}}(P, Q)$.*

Proof. Notice that from a proof that $P \leq Q$ using the anti-frame inference rules, we gets an extrapolation proof tree with the same structure. Now we just need to show that at each step the extrapolated part is larger than the anti-frame. Only for the |-Frame and |-Extrapolate differs and this trivially holds since $F' \leq !F'$. □

Unit Limits. A limit process term P is called *unit* if for all limit process terms R, Q and contexts C such that $P \equiv C[R \mid Q]$, neither $R \leq Q$ nor $Q \leq R$. Furthermore, we assume there are no redundant !, i.e. there is no C, R such that $C[!!R]$. We denote by *Unit* the set of all units.

We introduce units in order to facilitate the proof of convergence of the widening operator for depth-bounded systems. We will prove that the extrapo-

lation/widening progress on the units. Progress combined with Lemma 11 gives use convergence of the widening.

Lemma 11 *The set Unit is finite modulo congruence of limit process terms.*

Proof. By induction on the syntax tree of configurations:

Process IDs: A system is defined by a finite set of equations and due to the depth-boundedness there is only a finite number of names that can be used as parameters of the processes.

Restriction: Due to depth-boundedness, there are only a bounded number of nested restrictions.

Replication: By induction hypothesis, there are a finite number of possible subtrees. Furthermore, replication is idempotent.

Parallel composition: But by definition of units, all of the elements in a parallel composition should be different. Since $\leq$ is a bqo, we know that antichains are finite. Therefore, there is only a finite number of elements picked from a finite set (induction hypothesis).

Thus, we conclude that there is only a finite number of units. □

We also need to prove that any configuration P is covered by some unit U. This concretely means that the entire state-space can be covered by units.

Lemma 12 *For any configuration P there exists an unit $U \in Unit$ such that $P \leq U$.*

Proof. As long as P contains a context C such that $P \equiv C[R \mid Q]$ and $R \leq Q$ replace $R \mid Q$ by $!Q$ to obtain $P' \equiv C[!Q]$. Then repeat this as long as P' is not an unit. We can easily see that $P \leq P'$. Therefore, when this procedure finishes we have an U such that $P \leq U$. Furthermore, at each step the number of nodes in the syntax tree of P decreases which means that procedure eventually terminates. □

For a set of limit process terms π we denote by $\mathsf{units}(\pi)$ the set of units that are subsumed by elements of π, i.e., $U \in \mathsf{units}(\pi)$ iff $U \leq P$ for some $P \in \pi$.

Lemma 13 *Let P, Q be two limit configurations such that $P < Q$. Then $\mathsf{units}(P) \subsetneq \mathsf{units}(\chi_{\mathsf{DBP}}(P, Q))$.*

Proof. Due to the strictness of $P < Q$, we know that Q contains a least one subterm that cannot be covered by P. This means that applying the extrapolation rules gives an inference tree that contains a path starting from a |-Extrapolate and ending at the root where $\mathsf{units}(P) \subsetneq \mathsf{units}((\nu\vec{x})(R \mid F))$ hold at each step of the inference. First we need to show that the path actually starts by |-Extrapolate and that the invariant holds at the initial step. Then we do induction on that path and the extrapolation rules to show that the invariant continues to hold. The invariant holding at the root of the extrapolation tree implies the theorem.

The path must start with a |-Extrapolate since it is the only rules that allows removing element on the Q side only which is required since the inequality between P and Q is strict. We know the $P \leq Q$ and $P < Q \mid F'$. We can now distinguish two case:

when $P \geq F'$ by the strictness of $P < Q \mid F'$ we know that it is not a unfolding of a replication in P that covers F'. Furthermore, the extrapolation contains $!F'$ and it is only possible to cover a replication with another replication. Thus, we can conclude that $\mathsf{units}(!F') \not\subseteq \mathsf{units}(P)$.

when $P \not\geq F'$ it follows that $\mathsf{units}(!F') \not\subseteq \mathsf{units}(P)$

Then we need to show that the formula continues to hold at each step of the inference, i.e. each rule. We observe that for these cases, the new units returned by the extrapolation have some replicated components and that the only way of covering a replicated part is by another replicated part.

Eq: This rules is a leaf rule and since our path starts with |-Extrapolate the Eq rule is not part of the path.

ν-Subtr: The units are the P side and the extrapolated side are changed in the same way. Thus, the units not in P are not in $(\nu z)P$.

Struct: The rule does not changes the units since $\equiv$ preserves $\leq$.

|-Extrapolate: Follows directly from the induction hypothesis.

|-Subtr: By induction, we know that $\mathsf{units}(P) \subsetneq \mathsf{units}((\nu\vec{x})(R \mid F))$ and because we are in the path the exhibit the strictness of the inequality between P and Q we know that $P \mid S \not\leq (Q \mid S)$. The later part implies that there are no replicated part in S that covers Q. Therefore, the units in S does not cover that units of $(\nu\vec{x})(R \mid F)$ because it will contains some additional replicated units.

!-Frame: By induction, we know that $\mathsf{units}(P) \subsetneq \mathsf{units}((\nu\vec{x})(R \mid F))$ and because we are in the path the exhibit the strictness of the inequality between P and Q we know that $!P \not\geq !Q$. However, troubles can arise because ! is idempotent, so the units in $(\nu\vec{x})(R \mid F)$ and $!(\nu\vec{x})(R \mid F)$ might be the same. We show by contradiction that this cannot be the case.

Assume that $\mathsf{units}(!P) = \mathsf{units}(!(\nu\vec{x})(R \mid F))$. Because the new unit is introduced by extrapolation it is a replicated product (! distribute over |), thus $F \equiv !F' \equiv \Pi_i\, !f_i$. If it is not covered by $\mathsf{units}(P)$ but by $\mathsf{units}(!P)$ it means that the new unit in F are not affected by !. Since ! does not distribute over ν it means that the part in F does not uses any of the name in $\vec{x}$. Then, our assumption implies that $\mathsf{units}(\Pi_i\, !f_i) \subseteq \mathsf{units}(!P)$. $!P < !Q$ implies $\Pi_i\, !f_i \not\leq !P$ which in turn implies that there is an i such that $f_i \not\leq !P$. Therefore, $f_i \in {\downarrow}\mathsf{units}(F)$ and $f_i \notin {\downarrow}\mathsf{units}(!P)$ which contradicts our additional hypothesis and conclude the proof.

□

For an ascending chain of limit process terms $\pi = (P_i)_{i\in\mathbb{N}}$ we denote by $\nabla^*_{\mathsf{DBP}}(\pi)$ the *widening chain* $(Q_i)_{i\in\mathbb{N}}$ where $Q_0 = P_0$ and $Q_i = \chi_{\mathsf{DBP}}(Q_{i-1}, P_i)$

if $Q_{i-1} \leq P_i$ and $Q_i = P_i$ otherwise, for all $i > 0$. Then $\nabla_{\mathsf{DBP}}(\pi)$ is the last element of the widening chain $\nabla^*_{\mathsf{DBP}}(\pi)$.

Theorem 6 *∇_{DBP} is a set-widening operator.*

Proof. Assume an infinite ascending chain π such that π is not stabilizing.

First, we show that $\mathsf{units}(\pi) \subseteq \mathsf{units}(\nabla^*_{\mathsf{DBP}}(\pi))$. By definition of the widening chain and by Lemma 10, we know that every element of $\nabla^*_{\mathsf{DBP}}(\pi)$ is the same or greater as the corresponding element in π.

Then, we prove by contradiction that $\nabla^*_{\mathsf{DBP}}(\pi)$ is stabilizing. Because $\leq$ is a bqo and neither π nor $\nabla^*_{\mathsf{DBP}}(\pi)$ are stabilizing, χ_{DBP} will be applied infinitely many time on two configurations where the later one is strictly greater. By Lemma 13, we conclude that we are generating infinitely many new different units which contradicts Lemma 11. Thus, $\nabla^*_{\mathsf{DBP}}(\pi)$ is stabilizing and ∇_{DBP} is a set-widening operator. □

The intuition behind the termination argument for the operator ∇_{DBP} is that for an infinite ascending chain of limit processes, ∇_{DBP} gradually saturates the finitely many nesting levels of restrictions in the process terms of the chain, which corresponds to the subsumption of new unit limits.

3.8 Further Related Work.

In Section 3.2 we have already explained, in detail, the connection of our work with acceleration-based algorithms for computing the covering set. We discuss further connections with algorithms for solving the related coverability problem. The simplest algorithm for this problem is a backward analysis described in [3]. In practice, backward algorithms tend to be less efficient than forward algorithms, especially for dynamic process networks where the pre operator is expensive to compute, see example 2. Therefore, many attempts have been made at deriving complete forward algorithms for this problem. The most general solutions are described in [52] and [51].

The expand, enlarge, and check algorithm [52] decides the covering problem using a combination of an under-approximating and an over-approximating forward analysis. The over-approximating analysis relies on a so-called adequate domain of limits for the representation of downward-closed sets, which is actually the ideal completion of the underlying well-quasi ordering [47]. Ganty et al. propose an alternative algorithm [51] based on abstract interpretation. Unlike our approach, this algorithm uses a finite abstract domain that represents downward-closed sets by complements of upward-closed sets. The algorithm then relies on a complete refinement scheme to refine the abstraction for a specific coverability goal. Both algorithms [52, 51] compute an over-approximation of the covering set as a byproduct of the analysis, namely an invariant whose complement contains the coverability goal. To ensure completeness, the precision of this computed invariant is geared towards proving the specific instance of the coverability problem. Instead, our analysis computes a precise approximation of the covering set that is independent of any specific coverability instance.

Another research direction goes toward a better understanding of flattability and completeness of acceleration based aglrorithm. [47] propose the clover

algorithm that is complete for flattable system and [27] identify the class of trace-bounded WSTS. Trace-bounded WSTS are interesting because (1) their are flattable and (2) it is decidable to determine whether a system is trace-bounded.

Summary

We proposed a novel abstract interpretation framework to compute precise approximations of the covering set of WSTS. Our analysis captures the essence of acceleration-based algorithms that compute the exact covering set but only terminate on flattable systems. By replacing acceleration with widening we ensure that our analysis always terminates. We discussed several concrete instances of our framework including the application to depth-bounded process networks, which are typically non-flattable.

Chapter 4

Implementation: Picasso

We implemented the results presented in Chapter 2 and Chapter 3 in the Picasso tool, a static analyzer for depth-bounded systems. Furthermore, Picasso generalises the π-calculus version of depth-bounded systems to graph rewriting systems, thus offering more flexibility. Picasso computes the covering set of depth-bounded systems and can also generate abstraction of depth-bounded systems as numerical transition systems. We have used Picasso to automatically verify safety and liveness properties of complex concurrent systems such as nonblocking and distributed algorithms, as well as sequential object-oriented code.

This chapter is joint work with Thomas A. Henzinger and Thomas Wies. It has not yet been published.

4.1 Overview

Graph rewriting systems [42] provide a formalism for describing concurrent computations. In particular, they can model concurrent programs with message passing, dynamic thread creation, dynamically changing communication topology, and complex shared heap structures.

In this part, we express depth-bounded systems, not in terms of π-calculus, but as graph rewriting systems. This new formalism, make depth-bounded systems more intuitive to understand and more flexible to use. A graph rewriting systems is depth-bounded if there exists a bound on the length of all acyclic paths in all reachable graphs of the system. Often, the graph rewriting models of concurrent programs are depth-bounded or have natural depth-bounded abstractions that preserve important properties such as safety, cf. Chapter 3, and progress guarantees, cf. Section 5.1.

We present Picasso, a static analyzer that takes a DBS as input and computes an over-approximation of its covering set. In general, the covering set of a DBS is not computable, thus the over-approximation. However, Picasso implements an algorithm that exploits the monotonic structure of DBS and often yields precise results. Picasso accepts DBS in two input formats: a low-level format, in which the system is specified directly in terms of graph rewriting rules, and a high-level process algebra notation, which is compiled to graph

rewriting rules. In addition to computing the covering set, PICASSO can also produce a numeric program that abstracts the input DBS and is subject to further analysis by other verification tools.

We have successfully used PICASSO to verify safety and liveness properties of DBS that model complex infinite state systems. This includes non-blocking implementations of concurrent data structures and distributed message-passing algorithms with an unbounded number of threads and messages.

4.2 Example

Map-reduce example. We show how PICASSO works on an example that is inspired by an implementation of a map-reduce algorithm [62, Chapter 9] in SCALA using the SCALA actor library. [1] An actor is a heap object that has an associated thread of execution. All actors execute concurrently and communicate through asynchronous message passing, i.e., references between actors on the heap can also be thought of as communication channels.

The map-reduce algorithm takes a collection of data elements as input and computes a set of key/value pairs. The algorithm consists of two stages: the mapping and the reduction stage. In the mapping stage each input data value is mapped to a set of key/value pairs. In the reduction stage, the values across all input data that are associated with the same key are aggregated and then reduced to a single key/value pair. A typical example of an application of map-reduce is counting key words in web documents. The input data is a set of web documents. The mapping stage counts the occurrences of the key words in each document. The reduction stages sums up the occurrences for all documents.

In the implementation of the mapping stage, the `master` actor creates a `worker` actor for each element of the input data and sends the data to the workers, which apply the mapping function and send the lists of key/value pairs back to the `master`. In the reduction stage, the `master` creates a `reducer` actor for each key and sends it the aggregated values for that key. The `reducer` then reduces all these values to a single value which is again sent back to the `master`.

DBS model. We model the example using a set of graph rewriting rules that define a depth-bounded system. Heap objects (including actors) are represented by nodes in the graph and references between objects by edges. Each node is labeled by the internal state of the associated object and each edge is labeled by the name of the associated reference variable. We require that the set of node labels is finite. This is ensured by applying a form of predicate abstraction to the internal state of each object, yielding a finite partition of internal states into equivalence classes, each of which is represented by a unique label. For instance the node corresponding to an actor is labeled with the class of the object and the value of the program counter of the associated thread, e.g. `master1` for an instance of the `master` actor at program location 1.

Graph rewriting rules. We use an adaptation of graph rewriting systems with single pushout [42] on labeled directed graphs. A rewriting rule is composed of two graphs, the left-hand-side (LHS) and the right-hand-side (RHS),

[1] The full example and the analysis output generated by PICASSO are available at http://pub.ist.ac.at/~zufferey/picasso/mapReduce-report.html

and a partial morphism m between the two graphs. Intuitively, applying a rule to a graph G has the following semantics: the LHS is tested for inclusion in G; if no subgraph of G matches the LHS, the rule does not apply. Otherwise, if a subgraph of G matches the LHS, it is replaced by the RHS. Hereby, the morphism m is used to preserve the edges connecting nodes outside of the matching subgraph to nodes inside that subgraph.

Rule extensions: wildcards and guards. PICASSO provides two extensions of the standard graph rewriting rules to obtain more succinct representations of models:

- *Wildcards:* A wildcard is a special node label that is denoted by the underscore symbol _. When matching the LHS of a rule in a graph G, a wildcard matches against all possible labels and is instantiated by the matching. This is useful for translating code that applies to an object regardless of what its internal state is. Using wildcards, we can model the behavior of such code using a single rewriting rule.
- *Guards:* A rewriting rule can be guarded by an inhibitory graph. The semantics is that the rewriting rule should not fire on a graph G if the inhibitor can be matched to a subgraph of G.

Figure 4.1 shows a transition that models a step of the reduction stage in which a reducer has reduced its last value, sends the accumulated value back to the master, and then terminates. The inhibitor graph in the guard is used to express that all values have been reduced. The wildcard is used to match the master actor, regardless of what its internal state is. The blue edges indicate morphisms.

Analysis approach. Our map-reduce model is a DBS because the rewriting rules do not create arbitrarily long chains of nodes in the reachable graphs. The model is parameterized by the number of input data values and keys. Hence, the number of created mappers, reducers, and key/value pairs is unbounded. Nevertheless, DBS can be effectively analyzed because they are well-structured with respect to the subgraph ordering [80]. The underlying analysis algorithm of PICASSO is based on *ideal abstraction* [117], which is an abstract interpretation for computing precise over-approximations of the covering set of a WSTS. PICASSO implements ideal abstraction for the concrete case of DBS.

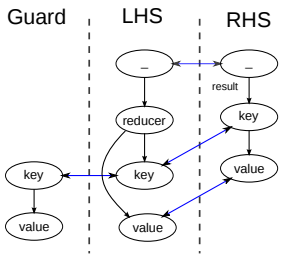


Figure 4.1: A rewriting rule

Nested graphs. Ideal abstraction exploits the fact that every downward-closed subset of a well-quasi ordered set can be decomposed into a finite union of order ideals. This yields a finite representation of infinite downward-closed sets such as the covering set of a WSTS, and is key to an effective analysis of such systems. The ideals of DBS can be represented as finite hedge automata [112], or more intuitively as *nested graphs*. A nested graph represents the downward-closure of all graphs that are obtained by recursively unfolding the nested sub-

graph components. For example, Figure 4.2 shows a nested graph that represents the downward-closure of the set of initial states of the map-reduce example. We indicate the nested subgraphs by surrounding them with a dashed box. That is, the graph in Figure 4.2 represents a master actor with an arbitrary number of associated input data values.

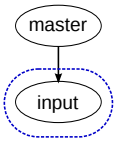


Figure 4.2: Initial states of map-reduce

Computing the Covering Set. For some classes of WSTS, the covering set is computable precisely. A prominent example are Petri nets for which the covering set can be computed using acceleration-based algorithms such as the Karp&Miller tree construction. Unfortunately, the covering set of a DBS is not computable in general. Ideal abstraction solves this problem by mimicking the Karp&Miller tree construction for Petri nets [73], but replacing acceleration with widening to ensure termination at the cost of precision. In the case of DBS, this analysis computes an *abstract coverability tree* whose nodes are nested graphs. The tree is constructed starting from the root note which represents the initial states of the system. Whenever an unprocessed leaf node of the tree is not yet subsumed by a processed node, it is expanded. The expansion adds successor nodes for all nested graphs that are obtained by exhaustive symbolic application of the rewriting rules of the system. Figure 4.3 shows the individual steps of a symbolic application of the rewriting rule in Fig. 4.1. These steps are as follows:

- *Unfolding:* When the LHS matches a part of a nested subgraph, we unfold the nested subgraph to get a concrete copy of the matched part.
- *Inhibiting:* The extension of rewriting rules with guards potentially breaks the monotonicity property that is required for WSTS. Therefore, we use the idea of monotonic abstraction [2] to obtain a sound implementation of guarded rewriting rules. Monotonic abstraction adds additional transitions to the model to enforce monotonicity. Concretely, if the inhibitor graph matches a graph G, we remove the matched part from G to obtain G' on which we apply the rewriting rule as usual. Since G' is not necessarily reachable in the original system, guarded rules introduce a potential source of incompleteness to the analysis (the other source of incompleteness is widening).
- *Rewriting:* The unfolded match of the LHS is replaced by the RHS.
- *Folding:* To maintain a compact representation of nested graphs in the tool, we fold nested graphs by removing subgraphs which are subsumed within the nested graph itself.

The final nested graph that results from the rule application in Fig. 4.3 is identical to the nested graph to which the rule has been applied. Hence, it is subsumed and does not need to be expanded further. All nested graphs that are not subsumed after an expansion step are recursively widened with their ancestor nodes in the tree. This ensures that the construction of the abstract coverability tree eventually terminates. The set of all nodes in the resulting tree represents a downward-closed inductive invariant of the system and, hence, subsumes the covering set.

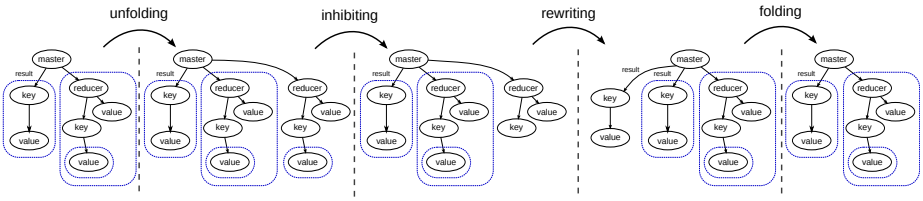


Figure 4.3: Symbolic application of the rewriting rule from Fig. 4.1 to a nested graph

4.3 From π-calculus to graphs

We now present in a formal way the graph rewriting systems that Picasso handles and how limits, originally the replication in π-calculus, are represented as nested graphs.

Graph transformation systems. We use an adaptation of the standard notion of graph transformation systems with the single pushout approach [42] to labeled directed graphs.

A *rewriting rule* is a partial morphism $r : G_L \rightharpoonup G_R$, where G_L is called *left-hand side* and G_R is called *right-hand side*. A *match* of r is a total injective morphism $m : G_L \to G$. Given a rule r and a match $m : G_L \to G$, a *rewriting step* is the *pushout* of r and m, which consists of a graph H and two graph morphisms $r' : G \rightharpoonup H$, $m' : G_R \to H$ such that $m' \circ r = r' \circ m$ and for every pair of morphisms $r'' : G \rightharpoonup H'$ and $m'' : G_R \rightharpoonup H'$ there exists a unique morphism $f : H \rightharpoonup H'$ with $f \circ m' = m''$ and $f \circ r' = r''$. It is known that pushouts are guaranteed to exist, that they are unique up to isomorphism and that they can be effectively constructed.

A *graph transformation system* (GTS) $\mathcal{R}$ is a tuple (R, G_0), where R is a set of rewriting rules and G_0 an initial graph. A GTS $\mathcal{R} = (R, G_0)$ induces a transition system $\mathcal{T}(\mathcal{R}) = (\mathit{Graphs}, G_0, R, \xrightarrow{R})$ where R is a finite set of rewriting rules, and $\xrightarrow{R}$ is the union of all relations $\xrightarrow{r}$, for $r \in R$. The subgraph ordering $\preceq$ is monotonic with respect to graph rewriting.

Lemma 14 *Let $\mathcal{R} = (R, G_0)$ be a GTS, then $\preceq$ is monotonic with respect to $\xrightarrow{R}$.*

Proof. Given three graphs G, G', H, a rewrite rule $r : G_L \rightharpoonup G_R$, and a match m such that $G \preceq G'$ and r and m applied to G gives $r' : G \rightharpoonup H$ and $m' : G_R \to H$. We construct a graph H' such that $H \preceq H'$ and a morphism $r'' : G' \rightharpoonup H'$ such that r'' and m' are the pushout of r and m on G', H'. r'' is obtained by adding to r in the following way: $r'' = r' \cup \{(v, v) \mid v \in G' \setminus G\}$. □

The *depth* of a graph G is the length of the longest simple path in the undirected version of G, obtained by taking the symmetric closure of the edges. For $k \in \mathbb{N}$, we denote by $\mathcal{G}_{\leq k}$ the set of all graphs with depth at most k. We say that a set of graphs $\mathcal{G}$ is *depth-bounded* if $\mathcal{G} \subseteq \mathcal{G}_{\leq k}$ for some $k \in \mathbb{N}$. A *depth-bounded system* (DBS) is a GTS $\mathcal{R} = (R, G_0)$, whose reachable configuration

graphs are depth-bounded, i.e., $\mathrm{Reach}(\mathcal{T}(\mathcal{R})) \subseteq \mathcal{G}_{\leq k}$, for some $k \in \mathbb{N}$. We call k a *bound* of the system. From Proposition 4 it follows that $\preceq$ is a wqo on depth-bounded sets of graphs.

Lemma 15 *For any $k \in \mathbb{N}$, $(\mathcal{G}_{\leq k}, \preceq)$ is a bqo.*

Proof. Using Laver's theorem [75], see Proposition 4. □

Thus, Lemmas 14 and 15 imply that depth-bounded GTSs induce WSTSs.

Theorem 7 *Let $\mathcal{R} = (R, G_0)$ be a DBS, then $(\mathrm{Cover}(\mathcal{R}), G_0, R, \xrightarrow{R}, \preceq)$ is a WSTS.*

Nested graphs. We represent downward-closed depth-bounded sets of graphs as finite sets of *nested graphs*. Formally, a *nested graph* $\widehat{G}$ is a tuple (V, E, ν, l) where (V, E, ν) is a labeled graph and $l : V \to \mathbb{N}$ maps each vertex to its *nesting level*. We abuse notation and denote the labeled graph of a nested graph $\widehat{G}$ by G. We extend the notion of homomorphism to nested graphs as expected, i.e., homomorphisms on nested graphs also preserve the nesting levels of vertices.

Intuitively, a nested graph $\widehat{G}$ represents the set of *concrete* graphs that can be obtained by recursively unfolding the nested subgraphs of $\widehat{G}$ arbitrarily often. In the following, we make these notions formal.

We define a *one-step unfolding* relation on nested graphs $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$, denoted $\widehat{G} \rightsquigarrow \widehat{H}$, as follows. For $i \geq 1$, denote all vertices at nesting level i or higher by $V_{\geq i} = \{\, v \in V \mid l(v) \geq i \,\}$. Unfolding involves *duplicating* the subgraph induced by $V_{\geq i}$ and reducing the nesting level of all vertices in the copy of $V_{\geq i}$ by one. Formally, we have $\widehat{G} \rightsquigarrow \widehat{H}$ iff for some $i \geq 1$ there exists a partition U, W_1, W_2 of V' and a homomorphism $h : H \to G$ such that $H[U \cup W_1] \cong G \cong H[U \cup W_2]$, $H[W_1] \cong G[V_{\geq i}] \cong H[W_2]$ under (natural restrictions of) h, $W_1 \times W_2 \cap E' = \emptyset$, for all $v' \in V' \setminus W_2$, $l'(v') = l(h(v'))$, and for all $v' \in W_2$, $l'(v') = l(h(v')) - 1$. When required, we refer to the underlying homomorphism by saying $\widehat{G} \rightsquigarrow \widehat{H}$ *under* h.

We then define the concretization $\gamma(\widehat{G})$ of a nested graph $\widehat{G}$ as the downward closure (with respect to the embedding order) of the set of all unfoldings of $\widehat{G}$:

$$\gamma(\widehat{G}) = {\downarrow}\left\{ H \mid \widehat{G} \rightsquigarrow^* \widehat{H} \right\}$$

We extend γ to sets of nested graphs $\widehat{\mathcal{G}}$ as expected: $\gamma(\widehat{\mathcal{G}}) = \bigcup_{\widehat{G} \in \widehat{\mathcal{G}}} \gamma(\widehat{G})$.

Example 4 *In Figure 4.4, we show a nested graph $\widehat{G}$ one the left and a possible unfolding on the right. The left replicated subgraph is unfolded three times; the right one twice; the inner ones twice and 0 time respectively.*

Inclusion of Nested Graphs. We next show that inclusion on nested graphs is decidable. Let $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$ be nested graphs. Define the relation $\sqsubseteq$ on nested graphs as $\widehat{G} \sqsubseteq \widehat{H}$ iff $\gamma(\widehat{G}) \subseteq \gamma(\widehat{H})$. An *inclusion mapping* for $\widehat{G}$ and $\widehat{H}$ is a homomorphism $\widehat{h} : (V, E, \nu) \to (V', E', \nu')$ satisfying the following additional properties: (i) for all $v \in V$, $l(v) \leq l'(\widehat{h}(v))$; (ii) $\widehat{h}$ is injective with respect to level 0 vertices in V': for all $v, w \in V$, $v' \in V'$,

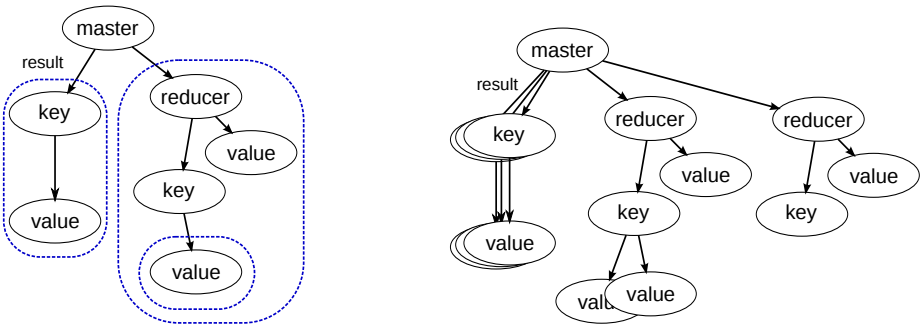


Figure 4.4: One the left a nested graph and, one the right, one possible unfolding of the graph.

$\widehat{h}(v) = \widehat{h}(w) = v'$ and $l'(v') = 0$ implies $v = w$; (iii) for all distinct $u, v, w \in V$ such that $\widehat{h}(u) = \widehat{h}(v)$, and u and v are both neighbors of w, $l(u) > l(w)$ and $l(v) > l(w)$.

Theorem 8 *Let $\widehat{G}$ and $\widehat{H}$ be nested graphs. Then $\widehat{G} \sqsubseteq \widehat{H}$ iff there exists an inclusion mapping $\widehat{h} : \widehat{G} \to \widehat{H}$. The problem of deciding the existence of $\widehat{h}$ is NP-complete.*

To see that the problem is in NP, note that each of the conditions for inclusion mapping can be checked in polynomial time. NP-hardness follows from the fact that the problem subsumes the subgraph isomorphism problem.

Nested graph rewriting. We lift application of rewrite rules to nested graphs by using inclusion mappings as the notion of a *match*. Intuitively, inclusion mappings allow us to apply the rewrite rule to an unfolding of the graph that contains the left-hand-side of the rule as a subgraph. Formally, we extend the notion of pushout to nested graphs in a natural way by using the homomorphisms defined on nested graphs. For a rewriting rule $r : G_L \to G_R$, naturally lift the notion and define $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$. A *match* of $\widehat{r}$ is an inclusion mapping $\widehat{m} : \widehat{G_L} \to \widehat{G}$.

Lemma 16 *Given a rule $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$ and a match $\widehat{m} : \widehat{G_L} \to \widehat{G}$, there exists a nested graph $\widehat{G'}$ and an injective inclusion mapping $\widehat{h} : \widehat{G_L} \to \widehat{G'}$ such that $\widehat{G} \rightsquigarrow^* \widehat{G'}$. Moreover, $\widehat{G'}$ and $\widehat{h}$ can be constructed in polynomial time.*

Let $\widehat{G'}$ be the nested graph and $\widehat{h} : \widehat{G_L} \to \widehat{G'}$ the injective inclusion mapping, as described in Lemma 16. Then there exists a pushout $\widehat{r'} : \widehat{G'} \rightharpoonup \widehat{H}$, $\widehat{h'} : \widehat{G_R} \to \widehat{H}$ for $\widehat{r}$ and $\widehat{h}$. This pushout defines a *rewriting step of nested graphs* $\widehat{G} \xrightarrow{\widehat{r}} \widehat{H}$.

Guarded rewrite rules and monotonic abstraction. A *guarded rewriting rule* is a pair of partial morphism $(i : G_L \rightharpoonup G_I, r : G_L \rightharpoonup G_R)$, where $r : G_L \rightharpoonup G_R$ is a rewriting rule and G_I if the guard (or inhibitor). Intuitively, given a match $m : G_L \to G$ the rule can be applied only if there is no match $m' : G_I \to G$ such that $m \circ i \subseteq m'$. Unfortunately, this extension breaks the compatibility requirement of WSTS. It is trivial to encode Petri net with inhibitory edges

with guarded rewrite rules. This reduction shows that guards make the model Turing-complete. Therefore, we use the idea of monotonic abstraction [2] to give a new monotonic semantics to guarded rules. In our case, we remove the parts matched by the guard and apply the rule on the trimmed graph.

Let (i, r) be a rewrite rule. We denotes by V_i the nodes which are in the pre-image of i and $V_{\neg i}$ the nodes which are in G_I but not in the pre-image of i.

A rewrite rule (i, r) is *well-formed* if for all G and all match $m : G_L \to G$ there is no match $m' : G_I \to G$ such that $m'(V_{\neg i}) \cup m(G_L) \neq \emptyset$. Intuitively, well-formedness means that the guard does not removes nodes which are used by the rewrite rule. It can be enforced by simple sufficient check such as requiring i to be surjective.

To apply a well-formed guarded rewrite rule (i, r) and a match $m : G_L \to G$ we need to distinguish two cases:

While there is some $m' : G_I \to G$ such that $m \circ i \subseteq m'$: We restrict G to $G' = G \setminus m'(V_{\neg i})$, and m to m'' similarly. Then we check for the existence of a new m'.

Otherwise: The rule is the pushout defined by r and m.

4.4 The Tool

Picasso is available for download at [116]. The website provides a binary and source code distribution of the tool, as well as read-only access to the SVN repository that we use for development.

External dependencies. Picasso is written in the Scala programming language and runs on the JVM. Many of the computation steps in the construction of the abstract coverability tree, such as inclusion tests and the computation of matches, are reduced to satisfiability of propositional formulas. Picasso uses Sat4J [20] to solve the generated SAT instances. When Picasso is used to compute the numerical abstraction of a DBS, it additionally requires Princess [104] and Z3 [1] to discharge quantifier elimination and satisfiability queries in (quantified) linear integer arithmetic.

4.4.1 Input formats

Picasso has two main frontends: the graph rewriting frontend presented in Section 4.2 and a simple high-level language inspired by actor-model languages. In fact, this language just provides a convenient syntax for sets of recursive equations in the π-calculus [88][2]. The programs written in the actor language are compiled into a graph rewriting system and then analyzed as such. Further examples as well as instructions to run Picasso can be found at [116].

We describe the two input languages in the extended Backus-Naur form. Grammar rules are in *italic*, terminal symbols in `texttt`. {} denotes repetition and [] denotes optional terms.

[2] Picasso actually derives its name from this fact. It stands for Pi-calculus-based static software analyzer.

EBNF grammar of the graph rewriting systems. The graph rewriting input language is offers a lot of flexibility and allows one to write very compact program that can be efficiently analyzed by PICASSO. The graph language, at first, was not supposed to be PICASSO primary input, but a convenient way of testing the backend by providing an input that directly matches the data-structures used in the backend. For instance, we will see that a transition has a forward and backward mapping (`==>`,`<==`). Both could be simplified into the same mapping to make the input simpler, but PICASSO uses both mapping in its internal representation and this particularity stayed in the language.

A graph rewriting system is given as an initial graph and a set of graph rewriting rules (transitions). The graph are labelled directed graphs where the labels are both on the nodes and the edges. Therefore, nodes have both an identifier and a label. The identifier is unique to each node. On the other hand, many nodes can have the same labels. We first describe the graphs (i.e. nodes and edges) and then the transitions.

A node consists of an identifier and a label. In case the label is not known, it is possible to have a special wildcard label (denoted _). Furthermore, it is possible to specify the multiplicity (nesting depth) of a node using *.

node ::= `(` *ident* `,` *label* `)`
| `(` *ident* `,` `_` `)`
| *node*`*`

An edge is given by a pair of nodes. Optionally, the edge can be labelled.

edge ::= *node* `->` *node* [`[` *label* `]`]

A graph is given by a sequence of edges. In case a node is not connected, it can be declared by prefixing it with "node".

graph ::= { *edge* | `node` *node* }

A mapping is a dictionary between nodes identifiers. It is given by a sequence of pairs.

mapping ::= { *ident* `->` *ident* }

A transition transform a subgraph (`pre`) into another one (`post`). The pre is matched within a larger graph (subgraph) and then replaced by the post. To preserve the connections with the rest of the graph during the replacement, the nodes in pre and post need to be connected by mappings. The forward mapping (`==>`) maps non-wildcard nodes from pre into non-wildcard nodes of post. The backward mapping (`<==`) maps wildcard nodes of post to wildcard nodes of pre. It is also possible to specify an inhibitory pattern that prevent the transition from firing.

transition ::= `transition` *stringLit*
`pre` *graph*
`post` *graph*
`==>` *mapping*
`<==` *mapping*
[`no` *graph*]

A system is given by an initial state (a graph), a sequence of transitions, and, optionally, a target state.

system ::= `init` *graph* { *transition* }

```
init  (e, env) -> (s, server) [S]

transition "new client"
pre   (e, env) -> (s, server) [S]
post  (e, env) -> (s, server) [S]
      (c, client) -> (s, server) [S]
==>   e -> e
      s -> s
<==

transition "request"
pre   (c, client) -> (s, server) [S]
post  (c, client) -> (s, server) [S]
      (m, msg) ->  (s, server) [S]
      (m, msg) ->  (c, client) [C]
==>   s -> s
      c -> c
<==

transition "reply"
pre   (m, msg) -> (s, server) [S]
      (m, msg) -> (c, _) [C]
post  (r, reply) -> (c, _) [C]
      node (s, server)
==>   s -> s
<==   c -> c

transition "receive reply"
pre   (r, reply) -> (c, _) [C]
post  node (c, _)
==>
<==  c -> c
```

Figure 4.5: Example of a graph rewriting system for Picasso

Example 5 *Figure 4.5 shows a simple client-server example in the spirit of Figure 3.1. The process consists of one single server thread, an environment thread, and an unbounded number of client threads. In each loop iteration of a client, the client non-deterministically chooses to either wait for a response from the server, or to send a new request to the server. Requests are sent asynchronously and carry both the address of the server and the client. In each iteration of the server loop, the server waits for incoming requests and then asynchronously sends a response back to the client using the client's address received in the request. The environment thread models the fact that new clients can enter the system at anytime. In each iteration of the environment thread, it spawns a new client thread. The initial state of the system consists only of the server and the environment thread.*

EBNF grammar of the basic frontend. The so-called basic input language is a layer of syntactic sugar on top of the asynchronous π-calculus. The goal is to have an human readable language that looks like some of the early

actor languages. Channels are orthogonal to actors, as names are orthogonal to threads in π-calculus.

A program written in this language is first transformed by PICASSO into a graph rewriting system and the processed in the same way a graph program.

The messages are described/matched using expressions/patterns. Then *process* provides a simple imperative language. Reception is done using a `select` statement that corresponds to the Σ in π-calculus. Each case of a select specify a channel and a pattern that the message to receive must match. Like in the orignal semantic of the actor model, messages are exchanged asynchronously and does not preserve the ordering between the messages.

A *literal* is either `true`, `false`, or a string literal (*stringLit*).

A pattern is either a wild card, a literal, an identifier, or a constructor.

pattern ::= `_`
| *literal*
| *ident* [`(` *pattern* { `,` *pattern* } `)`]

An expression is either a wild card (for undefined values), a literal, an identifier, or a constructor. The special `newChannel()` expression is not interpreted as a constructor but as the restriction operator (ν) of the π-calculus.

expression ::= `_`
| *literal*
| *ident* [`(` *expression* { `,` *expression* } `)`]

A process is a sequence of statements where statements can be variables (or constant) declarations, affectation, sending a message, creating a new actor, an if statement, a while loop, or a receiving a message.

process ::= `begin` [*process* { `;` *process* }] `end`
| `var` *ident* `:=` *expression*
| `val` *ident* `:=` *expression*
| *ident* `:=` *expression*
| *expression* `!` *expression*
| *expression*
| `new` *ident* `(` [*expression* { `,` *expression* }] `)`
| `if` *expression* `then` *process* [`else` *process*]
| `while` *expression* `do` *process*
| `select` { `case` *expression* `?` *pattern* `=>` *process*}

An actor is just a syntactic unit that binds the free parameters occurring in a process.

actor ::= *ident* `(` [*ident* { `,` *ident* }] `)` *process*

The initial configuration of the system is a list of actors. Identifiers in the arguments of the actors are assume to be top-level bounds names.

initial ::= `initial` *ident* `(` [*ident* { `,` *ident* }] `)`
{ `;` *ident* `(` [*ident* { `,` *ident* }] `)` }

Example 6 *Figure 4.6 shows a simple ping-pong example inspired from the example in Figure 1.1 and 1.2. There are two main difference compared to the* SCALA *version. First, we do not yet support integers. Therefore, we have*

```
Ping(self, pong)
  while true do
    select
      case self ? Start() => pong ! Ping(self)
      case self ? SendPing() => pong ! Ping(self)
      case self ? Pong() =>
        if random() then self ! SendPing()
        else begin
          pong ! Stop();
          exit()
        end

Pong(self)
  while true do
    select
      case self ? Ping(sender) => sender ! Pong()
      case self ? Stop() => exit()

Main()
  begin
    val ping := newChannel();
    val pong := newChannel();
    new Ping(ping, pong);
    new Pong(pong);
    ping ! Start()
  end

initial
  Main()
```

Figure 4.6: Example of an actor program for Picasso

replaced the counter by non-deterministic choice. Second, in scala each actor has its own channel. In our language the notions of channels (names) and processes are decoupled like in the π-calculus. Thus, the receive operations has to explicitly specify from which channel it is receiving.

4.4.2 Analyses and outputs

The default analysis of Picasso is to compute the covering set of a given depth-bounded systems. Picasso produces an HTML report that contains the textual input and a graphical representation of the rewrite rules, the nested graph representing the initial states, and the computed covering set. If the input system is expressed in terms of the high-level actor language, then the control-flow automaton of each actor is also included in the report.

Algorithm. Picasso is designed to be modular and different algorithms to compute the covering set have been implemented. The main algorithm is base on the Karp&Miller tree algorithm with a few optimisations to reduce the size of the tree. Algorithm 1 contains an high-level pseudo-code sketch of the algorithm used. We are building the abstract coverability tree using a depth-first search

and we are never cutting a subtree when a new branch subsumes an already computed branch. In that sense, our optimization produces slightly smaller trees compared to the full tree. The resulting tree may still contain redundancies, i.e., we are not computing the smallest tree as in the minimal covering tree algorithm [46] and thus avoid the problem of that algorithm [53].

Theorem 9 *Algorithm 1 computes* Cover($\mathcal{T}$).

Proof.

Correctness: Line 2 ensures that the algorithm computes an inductive invariant. If the test returns true then the algorithm has already visited a state s' which is greater or equal the current state s. This follows from the fact that cover variable contains a downward-closed set. By monotonicity, s' can cover all the states that s can cover. Therefore, we can stop exploring the successors from s. In this step we rely on the fact that s' (and its successors) will never be removed from the tree (unlike [46]).

Termination: Assuming that the algorithm do not terminates would create infinite strictly ascending chain on line 5. (The bqo prevents infinite descending chain or antichain.) Therefore, contradicting the property of ∇ which ensures that every ascending chain eventually stabilizes.

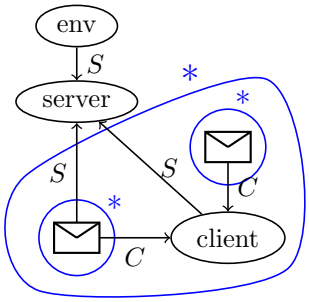
□

Algorithm 1 Algorithm to compute the covering set of $\mathcal{T}$

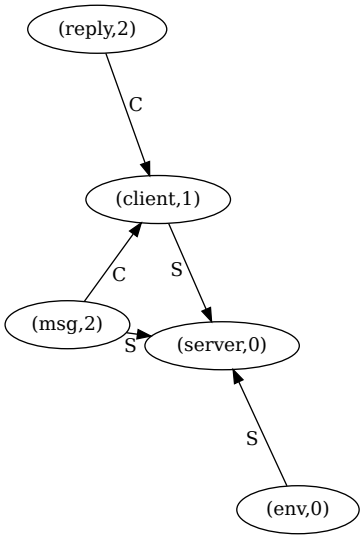
Input: a depth-bounded system $\mathcal{T}$
Output: Cover($\mathcal{T}$)

```
1:  function BUILD_TREE(ancestors, node, cover)
2:      if node.conf ∈ cover then
3:          return cover
4:      else
5:          node.conf ← ∇( ancestors.filter(_.conf ≤ node.conf), node.conf)
6:          for all c1 ∈ Post(node.conf) do
7:              n ← new node()
8:              n.conf ← c1
9:              node.children.append(n)
10:             cover ← BUILD_TREE(ancestors + node, n, ↓(cover + node.conf))
11:         end for
12:         return cover
13:     end if
14: end function
15: root ← new node;
16: root.conf = s_0;
17: return BUILD_TREE([], root, ∅)
```

On top of Algorithm 1 Picasso implements some optimizations to make the algorithm more scalable. For instance, at line 5, we do not use all the nodes, but we select them with an exponential back-off strategy. This allows Picasso to



(a) Nested graph



(b) PICASSO's representation of the same graph

Figure 4.7: Comparison between two representations of nested graphs

explore deeper covering trees. Furthermore, to counter the slowdown incurred by the growth of the tree we have implemented a restart policy. After 5 minutes PICASSO restarts the exploration by using the unexplored leaves of the current tree as roots for new trees. The new search continues with the current value of cover.

Numerical abstraction of depth-bounded systems. The computed covering set can be used to verify safety properties of the analyzed system. In addition, PICASSO supports an analysis for proving (fair) termination of the input depth-bounded systems [13]. In this mode, the report will contain a numerical abstraction of the depth-bounded systems that is expressed in the format of the termination checker ARMC [98]. When ARMC is available, PICASSO will directly run it on the numerical abstraction. If ARMC is able to prove termination of the numerical abstraction, then also the original system is guaranteed to terminate. The numerical abstraction can also be printed in the NUMERICAL TRANSITION SYSTEMS [69] (NTS) format and then analyzed with tools supporting this format, e.g. such as FLATA [21] or ELDARICA [66].

Remark 4 (Picasso's representation of nested graphs.) *Due to the internal representation of nested graphs in* PICASSO *and some laziness of the programmer,* PICASSO *displays nested graphs in a sightly more primitive and less user-friendly way than most of the figures in this documents. Instead of showing the replicated subgraphs by surrounding them with line,* PICASSO *prints the node in the format (*label, depth*) where the depth is the nesting depth of the node in the π-calculus sense, i.e. how many replication operator surrounds the terms. Figure 4.7 shows two version of the same graph, one in each format.*

Example 7 *Figure 4.8 and 4.9 shows the output of* PICASSO *on Example 5.*

The first figure shows a graphical representation of the rewrite rules given as text to Picasso. *This does not involve analysis per se, but it is useful to check the rules are correct. It is easier to spot mistakes on the pictures than on the text. The transitions have a graph for the left-hand-side (LHS) and one for the right-hand-side (RHS). The forward mapping is shown in blue and the backward mapping in green.*

The second picture shows the result of the analysis, i.e. the covering set and, optionally, the Karp&Miller tree. In the Karp&Miller tree, the black arrows between graph represents the successor relation and the blue arrow the covering relation. When a graph is covered by an already existing graph the exploration successors of that node are not explored.

Example 8 *Figure 4.10, 4.11 and 4.12 shows* parts *of the output of* Picasso *on example 6.*

Figure 4.10 shows that control-flow automata (CFA) of the `Ping` *and* `Pong` *actors. The initial states of the CFAs are shown as octagons. Note that the CFAs don't have final states, instead the action* {} *which stands for* `exit()` *plays that role. The convention for the naming of the state is* `actor_name(#line.#col)`.

Then the CFAs are translated into rewrite rules. Figure 4.11 shows the translation of a few selected kinds of statements. The total number of rewrite rule for this example is 18. Nodes carries as label either the control location of an actor or `name`. `names` *nodes represent channels (in the* π*-calculus) sense. Edges from actors are labelled by the name of the variable referencing the pointed node. A message (or constructor in the grammar) is encoded in a similar way. The edges have numbers that corresponds to the position of the argument and a special* `to` *edge points to the recipient.*

Finally, Figure 4.12 shows the Karp&Miller tree. Since this example is bounded (number of processes and messages), the nested graphs representation does not give any advantage compared to an explicit representation. Therefore, we show only the Karp&Miller tree as most of its nodes are part of the covering set.

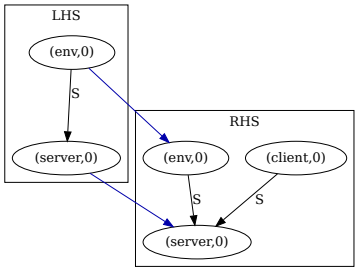
As we already mentioned Picasso is also able to generate (labelled) numerical abstraction that simulates a given depth-bounded systems. The main purpose of this abstraction is proving termination of depth-bounded systems. In Section 5.1 we will see how the abstraction is generated along with examples.

4.5 Evaluation

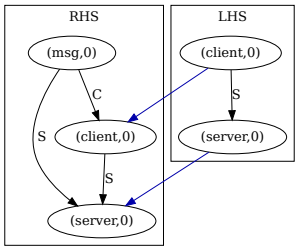
Picasso combines the ideal abstraction domain with a trace partitioning domain [103]. The resulting analysis is a generalized Karp&Miller tree construction with widening instead of acceleration. The implementation is parameterized by the concrete ideal completion and the widening operator on ideals that are used in the analysis. The tool Picasso and the example programs are available on-line [116].

For the analysis of our examples we have implemented a generalization of the ideal abstraction domain and widening operator for depth-bounded processes that we described in Sec. 3.7.3. The representation of ideals used in the implementation more closely resembles the communication graphs with nested

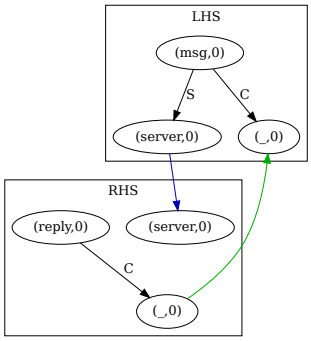
```
transition "new client"
pre   (e, env) -> (s, server) [S]
post  (e, env) -> (s, server) [S]
      (c, client) -> (s, server) [S]
==>   e -> e
      s -> s
<==
```




```
transition "request"
pre   (c, client) -> (s, server) [S]
post  (c, client) -> (s, server) [S]
      (m, msg) ->  (s, server) [S]
      (m, msg) ->  (c, client) [C]
==>   s -> s
      c -> c
<==
```




```
transition "reply"
pre   (m, msg) -> (s, server) [S]
      (m, msg) -> (c, _) [C]
post  (r, reply) -> (c, _) [C]
      node (s, server)
==>   s -> s
<==   c -> c
```




```
transition "receive reply"
pre   (r, reply) -> (c, _) [C]
post  node (c, _)
==>
<==  c -> c
```

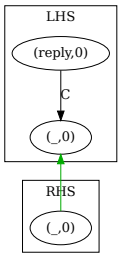
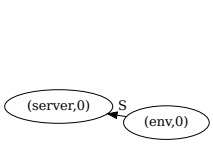
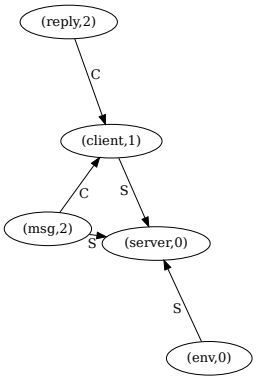


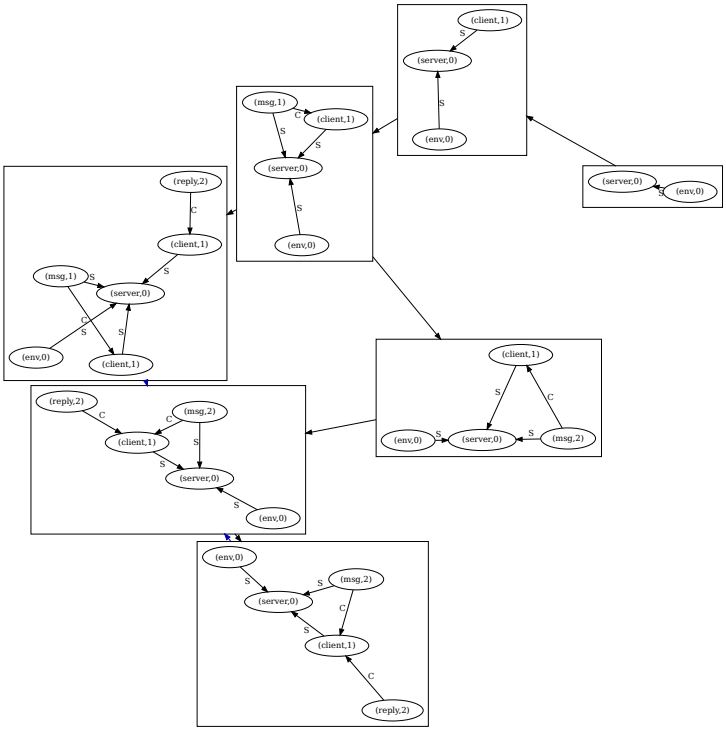
Figure 4.8: Transitions as text and image (generated by Picasso)



(a) Initial state

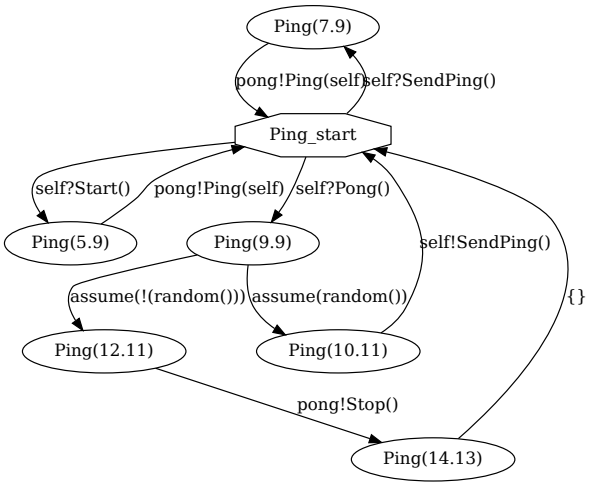


(b) Covering set

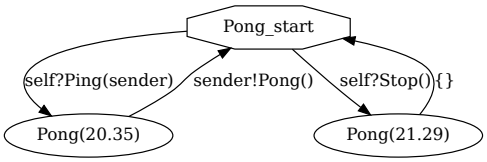


(c) Karp&Miller tree

Figure 4.9: Initial state, covering set, and Karp&Miller tree (generated by Picasso)

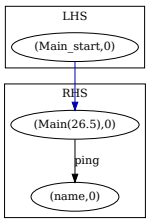


(a) `Ping(self, ping)`

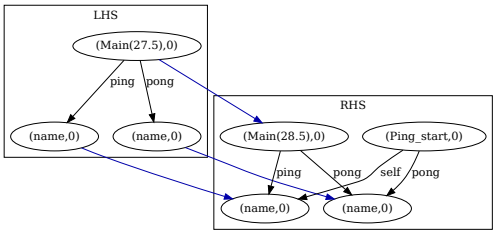


(b) `Pong(self)`

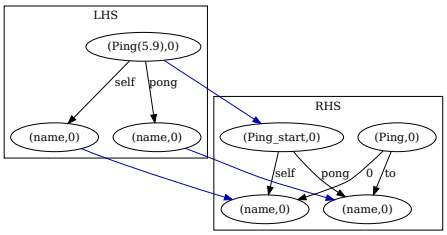
Figure 4.10: CFAs for `Ping` and `Pong` (generated by Picasso)



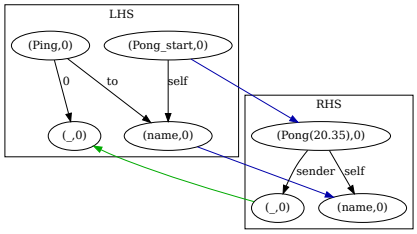
(a) `ping := new-channel()`



(b) `create(Ping, ping, pong)`



(c) `ping!Ping(self)`



(d) `self?Ping(sender)`

Figure 4.11: Translation of statements into rewrite rules (generated by Picasso)

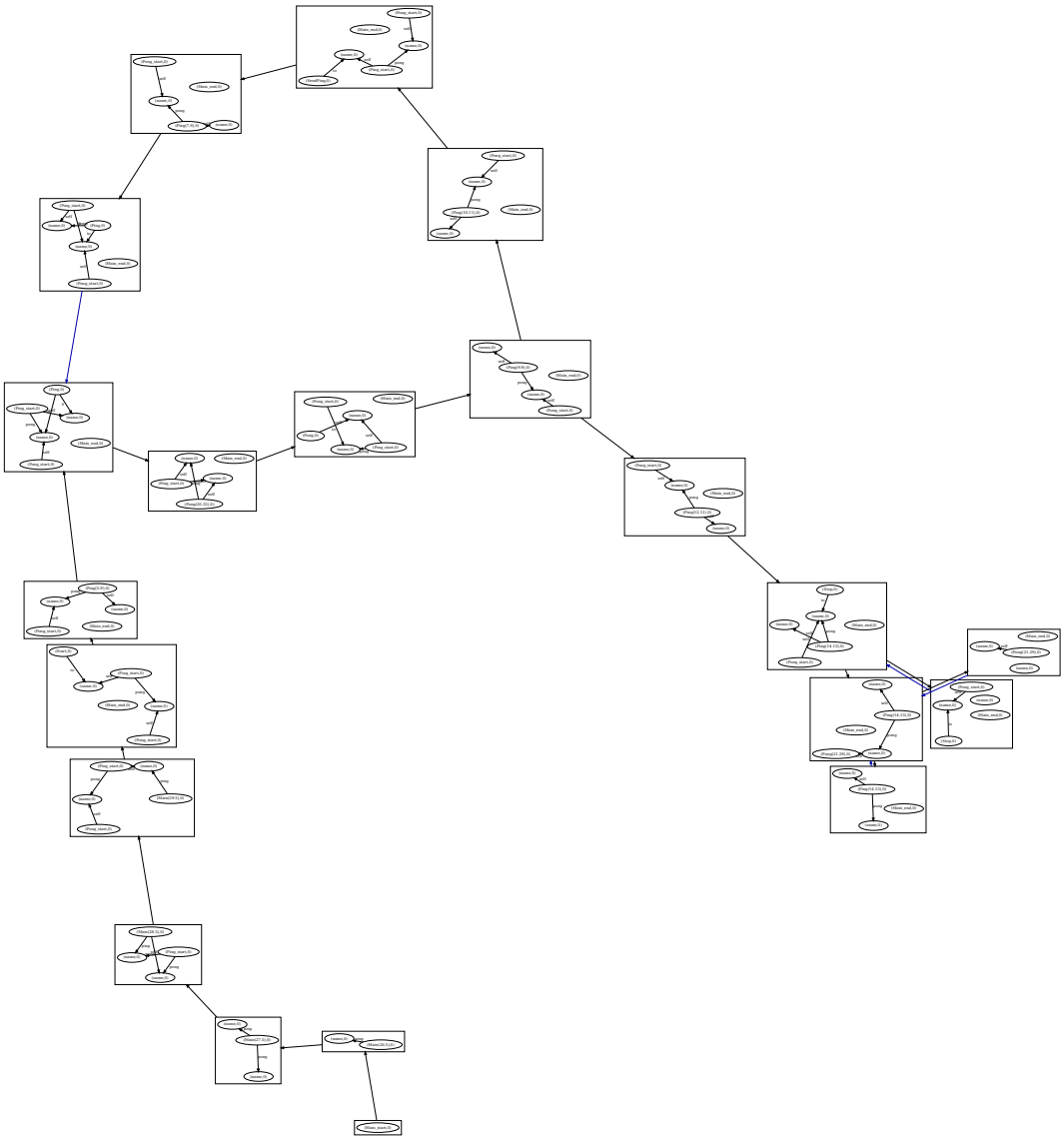


Figure 4.12: Karp&Miller tree for Example 6 (generated by PICASSO)

repeated substructures described in Sec. 3.2. This representation admits process nodes in communication graphs with arbitrarily many outgoing edges. Such nodes correspond to process identifiers in π-calculus process terms with unbounded (but unordered) parameter lists. To represent the limit elements we annotate the nodes in the graph with natural numbers indicating the nesting depth of the nodes. Testing the ordering on states is done by computing morphisms between the corresponding graphs. The morphisms take into account the nesting structure by allowing mappings to nodes of higher nesting depth to be non-injective. The actual test is encoded into a set of Boolean constraints and passed to a SAT solver. The morphisms are then reconstructed from the obtained satisfying assignments. The algorithm constructs a Karp&Miller tree using a depth-first search. When the tree is extended with a new node, widening is applied to the chains on the path to the root of the tree that contain the new node. Among the smaller ancestors of a node, not all are used for the widening. Instead, nodes are selected using an exponential back-off strategy. When the depth of the constructed tree becomes too large, the algorithm tends to slow down significantly. For such cases, we have implemented a restart policy. When a restart occurs, the leaves of the current tree are used as roots to construct new trees. The restart policy ensures that, for larger examples, the analysis terminates within reasonable time. The current implementation uses restart intervals of 5 minutes. The implementation exploits parallelism and makes use of multiple cores when possible.

We ran our experiments on a machine with two AMD Opteron 2431 processors and a total of 12 cores. We found that memory consumption was not an issue for the analysis of our examples. The examples that we have considered are depth-bounded processes, which are inspired by Scala programs. These Scala programs use the Scala actor library [61] for the implementation of dynamic process networks. Table 4.1 summarizes the results of our experiments. The `ping-pong` example is the "*Hello World*" of actor programming and is taken from the tutorial for the Scala actor library. All remaining examples follow a client-server type of communication with an unbounded number of clients. These examples cover common patterns that arise in message passing programs. The second and third program are variations of the example presented in Section 3.2. In the third program, we added a timeout to the receive operations of clients. We model the timeout by letting the clients send `Timeout` messages to themselves. This pattern is often used in programs based on the Scala actor library. The `genericComputeServer` example is the message passing skeleton of a tutorial for remote actors [10]. The example implements a compute server that accepts computation tasks from clients and then executes them. The second version uses actors to model the closures that are sent to the server. This model is obtained using the usual reduction of high-order π-calculus to the standard π-calculus. The `liftChatLike` example is the message-passing skeleton extracted from a chat application based on the lift web framework [78]. Since our implementation does not yet support collections, the broadcast pattern that is used in the original implementation has been changed into a polling pattern. The `round_robin_`k example is a load balancer that routes requests to a pool of k workers. Increasing the value of k greatly increase the number of interleavings that the analysis has to consider. With added support for collections, we can analyze a generic `round_robin_`k, which should also reduce the symmetry in the model.

Name	tree size	cov. set size	time
`ping-pong`	17	14	0.6 s
`client-server`	25	2	1.9 s
`client-server-with-TO`	184	5	12.8 s
`genericComputeServer`	57	4	4.6 s
`genericComputeServer-fctAsActor`	98	8	14.8 s
`liftChatLike`	1846	21	1830.9 s
`round_robin_2`	830	63	48.8 s
`round_robin_3`	3775	259	737.8 s
`round_robin_4`	22749	1108	26088 s

Table 4.1: Experimental results: the columns indicate the number of nodes in the Karp&Miller tree, the number of ideals in the covering set, and the running time.

Our experiments indicate that our analysis produces sufficiently precise approximations of the covering set to be useful for program verification and program understanding. The main bottle neck of our analysis is the explosion caused by interleavings of the transitions of individual processes. We did not yet explore techniques such as partial order reduction to tackle this problem.

We also considered concurrent data structure implementations such as Treiber's stack and the Michael-Scott queue. Such systems are usually not depth-bounded due to the linking structure of heap objects. However, they have natural depth-bounded abstraction that preserve progress guarantees such as lock-freedom [13], which can be reduced to fair termination. The experimental results are in Section 5.1.5.

Furthermore, we looked at depth-bounded systems encodings of object-oriented features like dynamic dispatch within a proof-of-concept Scala compiler plug-in (available as part of the sources). The compiler plug-in takes the parsed and typed abstract syntax tree of a Scala program, identifies actors with their related method calls, and produces a depth-bounded systems. The compiler plug-in was developed using Scala 2.9 and does not work on later version of Scala.

4.6 Related tools

The covering problem for Petri nets and their monotonic extensions is still a very active subject of research and tools are readily available. Among them are Mist [52], BFC [72], IIC [74]. Picasso is different from these tool in the sense that depth-bounded systems have a much larger state space than Petri nets. This leads to more difficulties in comparing two states (linear in Petri nets, NP-complete in depth-bounded systems), also acceleration/widening is more complex as we have seen in Section 3. These facts impact the scalability of Picasso. On the other hand, we can model systems that do not fit into Petri nets.

Joshi and König have also studied graph transformation systems that are well-structured with respect to the graph minor ordering [71]. Our approach targets a different application domain. We consider rewriting rules with injective matching. With this semantics graphs rewriting systems are not monotonic

wrt the graph minor ordering, i.e. not well-structured under the graph minor ordering. On the other hand, the graph minor ordering is a wqo for arbitrary graphs, while the subgraph ordering is a wqo only for graphs bounded in the length of their simple paths. Thus, the two approaches consider orthogonal classes of WSTS.

Related to the model-checking of π-calculus and actors we find two tools: SOTER [40] and PETRUCHIO [83].

SOTER is a safety verifier for core ERLANG programs, i.e. functional actor programs. SOTER takes as input a core ERLANG program and generate a Petri net abstraction which is then analysed with BFC. Due to the use of a Petri net backend, the precision of SOTER is somewhat limited. Unbounded creation of actors is allowed, but all the actors of one type share the same mailbox.

PETRUCHIO is a tool for the verification of temporal properties of π-calculus processes. PETRUCHIO reduces a fragment of the π-calculus, actually a fragment of depth-bounded systems, to Petri nets [81, 82]. Then, model checking is used to check the Petri net against an LTL specification.

Compared to both tools PICASSO is capable of handling a larger class of systems with more precision at the cost of scalability (SOTER), or the complexity of the properties verified (PETRUCHIO).

Chapter 5

Extensions: termination of depth-bounded systems, dynamic package interfaces

The decision to compute the covering set rather than trying to solve the simpler covering problem came about because we expected to extract useful information from the covering sets. High-level information, like the communication topology of mobile processes, can be extracted from the covering set and returned as part of the analysis result. Therefore, even without a clear safety property to check the analysis returns meaningful information.

From the experimental in Table 4.1, we saw that for many systems the covering set, or an overapproximation of it, is actually quite small. Even though we compute an over-approximation of the covering set, the analysis has a very good precision and returns the exact covering set in many cases. This encouraged us to look further into using the covering set to get more informations. In the next sections we will see how to us the covering set as starting points of two additional analyses: (1) termination and (2) state-machine-like interface for groups of interacting objects.

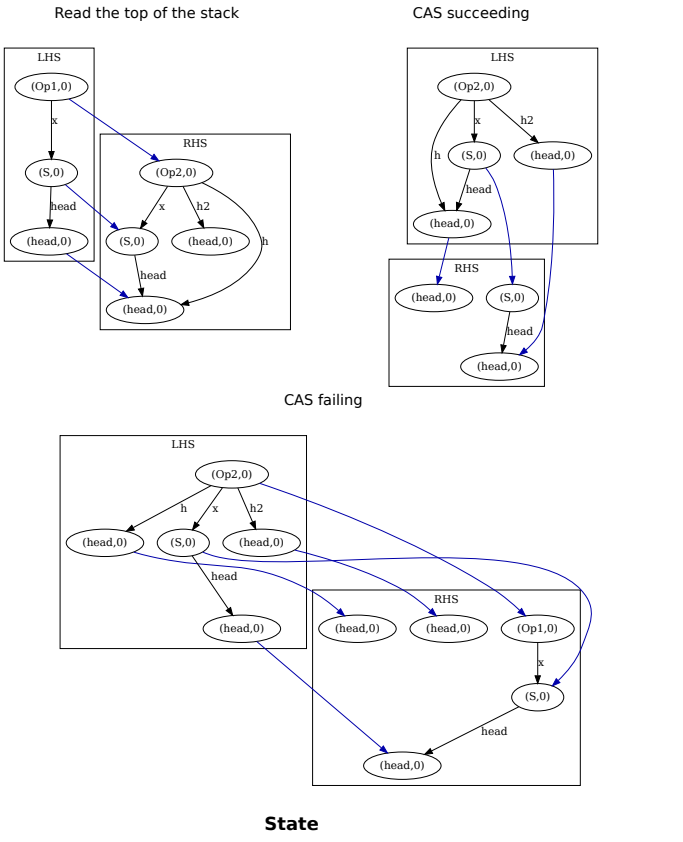
Before going into the details, let us recall some of the properties of the covering set. First and foremost, the covering set is an inductive invariant which means that it contains the initial state and is closed under the transition relation. Therefore, it subsumes all the behaviors of that the system may exhibit. The number of ideals in the covering set depends on how "monotonic" the systems is. In Figure 5.1, we show a system and its covering set and in Figure 5.2 we apply the transitions on the covering set and show the intermediate steps as in Figure 4.3. In Figure 5.2 the covering set is in the center and the three transitions each of the with three graphs are around it. For each transition the unfolding, rewriting, and folding are shown. These nested graphs connected by partial morphisms are the basic ingredients of the following two extensions. The example of Figure 5.1 will be explained in more details in Section 5.1.2.

Code: abstracted operation in Treiber's stack

```
    void op() {
        do {
Op1:        t = S->Top;
            x = ... ;
Op2:    } while (!CAS(&S->Top,t,x));
    }
```

Graph rewrite rules

(manually extracted from the code)



State

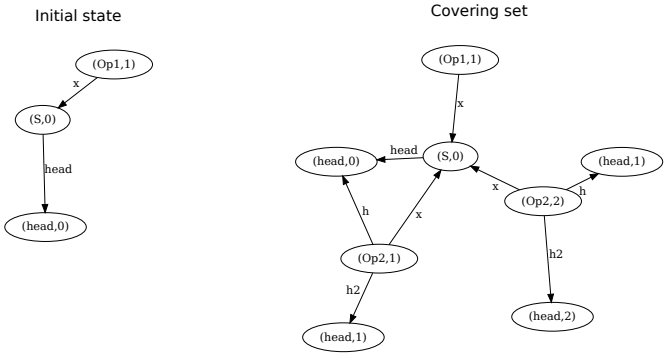


Figure 5.1: Simplified Treiber's stack (graphs generated by Picasso)

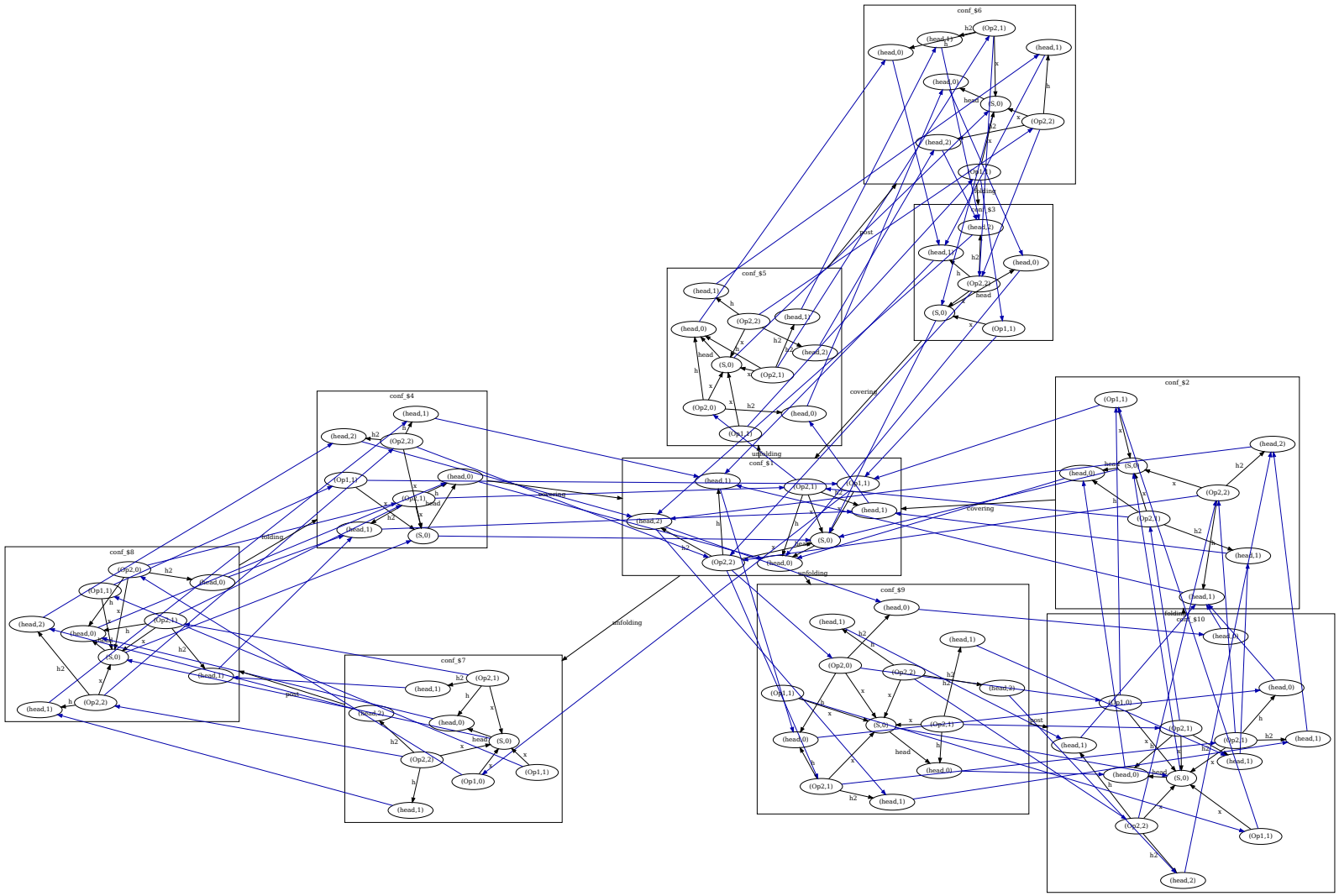


Figure 5.2: Transitions applied to the covering set

5.1 Structural Counter Abstraction

Depth-Bounded Systems can model a wide range of concurrent infinite-state systems including those with dynamic thread creation, dynamically changing communication topology, and complex shared heap structures. In [13] we present the first method to automatically prove fair termination of depth-bounded systems. Our method uses a numerical abstraction of the system which we obtain by systematically augmenting an over-approximation of the system's reachable states with a finite set of counters. This numerical abstraction can be analyzed with existing termination provers. What makes our approach unique is the way in which it exploits the well-structuredness of the analyzed system. We have implemented our work in Picasso and used it to automatically prove liveness properties of complex concurrent systems, including nonblocking algorithms such as Treiber's stack and several distributed processes. Many of these examples are beyond the scope of termination analyses that are based on traditional counter abstractions. In this section, we recall the main result of [13] and describe in more details the implementation of the method in Picasso.

This section is joint work with Kshitij Bansal, Eric Koskinen, and Thomas Wies. It was published in TACAS 2013 as "Structural Counter Abstraction" [13]. The author's contribution in this part is mostly related to the implementation. The theory required to understand the method and its implementation is quickly recalled to make the thesis self-contained, but should not be considered as a contribution. For the details of the methods, we refer the reader to the original publication [13] and the corresponding technical report [14].

5.1.1 Overview

Depth-bounded systems are also among the most expressive classes of WSTS, subsuming *e.g.* Petri nets and their monotonic extensions [82]. They can model a wide range of concurrent systems including those with dynamic thread creation, dynamically changing communication topology, and complex shared heap data structures. Many concurrent systems are depth-bounded. For instance, Actor-style message-passing systems often fall into this class. Other systems have natural depth-bounded abstractions that preserve important properties. For example, consider the lock-free stack due to Treiber [109] (see Figure 5.3), which uses atomic *compare-and-swap* instructions to implement nonblocking stack operations. This algorithm can be abstracted to a depth-bounded system by ignoring the order of the elements in the stack. This abstraction preserves the termination/progress behavior of the algorithm. Similar depth-bounded abstractions can be obtained for a wide variety of concurrent algorithms.

In [13], we presented the first method to automatically prove fair termination of depth-bounded systems. The method focuses on the notion of *weak fairness*. Many liveness properties of practical interest (including progress guarantees: wait-, lock-, and obstruction-freedom) are reducible to termination under weak fairness. The problem is difficult; it subsumes the structural termination problem for transfer nets (i.e. termination for all possible input markings), which is undecidable [79]. Despite this difficulty, we show that one can build on existing verification techniques for WSTS to obtain an approximate analysis for this

problem that is both practical and sufficiently precise to prove fair termination of complex systems.

Furthermore, the technique also extends to other fairness conditions. For instance, the implementation of the method in Picasso focus on strong fairness. In Section 5.1.6, we discuss how weak and strong fairness relate on the example we considered.

The key insight of [13] is that we can automatically construct a precise numerical abstraction of a depth-bounded system from a precomputed inductive invariant of the system. In the case of depth-bounded systems this inductive invariant is an over-approximation of the covering set. The covering set is given as a finite set of nested graphs, as we saw in Chapter 4. Thus, each nested graph is a symbolic representation of the (infinite) set of concrete graphs obtained by unfoldings. We associate a counter with each of the nested subgraphs, tracking how often it can be unfolded. From these augmented nested graphs we then compute a numerical transition system that simulates the depth-bounded system. This so-called *structural counter abstraction* can then be analyzed using existing termination provers. The number and meaning of counters in the numerical abstraction is not fixed *a priori* but, instead, depends on the structure of the reachable configuration graphs (described by the inductive invariant). Our method thus provides a more precise alternative to traditional counter abstractions [96, 16, 37] for concurrent systems.

The benefit of our approach is that it can utilize existing reachability analyses for depth-bounded systems to obtain the inductive invariant, cf. Chapter 3, and existing termination analyses for numerical programs [98, 30, 66]. We have implemented our method in Picasso and applied it to prove liveness properties of various concurrent systems, including nonblocking algorithms such as Treiber's stack, as well as distributed processes. These systems are beyond the scope of traditional counter abstraction techniques. The numerical abstractions obtained by our analysis are expressible in terms of transfer nets. However, we have chosen this encoding to support a more effective analysis using existing termination provers for integer programs.

Our technique enables the automated verification of liveness properties for a large class of concurrent infinite-state systems. What makes our approach unique is the way in which it exploits the monotonicity of the system. Our algorithmic technique of computing a numerical abstraction from an inductive invariant, introduced in this paper, promises applications beyond liveness properties. For instance, the same technique can be used to strengthen an inductive invariant of a depth-bounded system with numerical constraints, enabling proofs of complex safety properties.

5.1.2 Motivating Example

Consider Treiber's stack [109], a non-blocking algorithm, given in the C-like code in Fig. 5.3. The algorithm implements a stack with a simple linked-list. The two operations, `push` and `pop` use the *compare-and-swap* (`CAS`) instruction to atomically modify a location in memory. `CAS(l,v,v')` atomically examines the value at location `l` and, if it is equivalent to `v`, sets `l` to value `v'`. In this section, we will describe how we are able to prove lock-freedom of this algorithm via a reduction to fair termination of a depth-bounded system.

We can represent Treiber's stack algorithm as a depth-bounded system, by

```
struct node {
  struct node *next;
  value t data;
};

struct stack {
   struct node *Top;
};
struct stack *S;

void push(value t v) {
  struct node *t, *x;
  x = alloc();
  x->data = v;
  do { t = S->Top; x->next = t; }
  while (!CAS(&S->Top,t,x))
}
```

```
void init() {
  S = alloc();
  S->Top = NULL;
}

value t pop() {
  struct node *t, *x;
  do {
    t = S->Top;
    if (t == NULL) return EMPTY;
    x = t->next;
  } while (!CAS(&S->Top,t,x));
  return t->data;
}
```

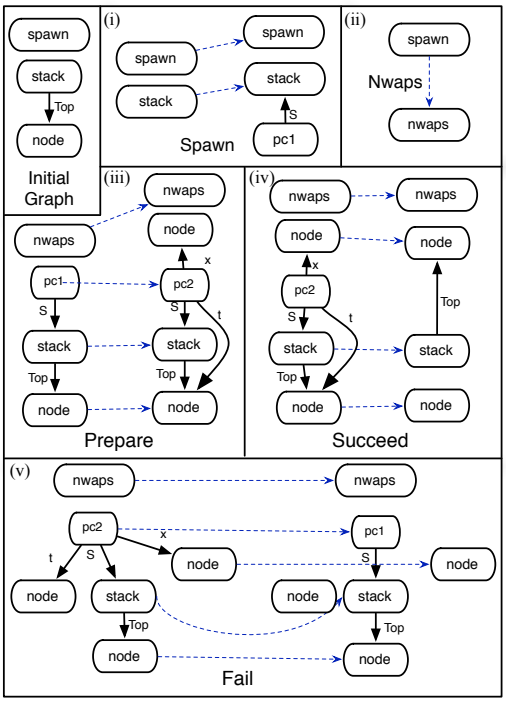


Figure 5.3: Source code of Treiber's stack [109] and its abstraction as a graph transformation system.

abstracting over the values and order of the elements in the stack. In the depth-bounded abstraction of Treiber's stack, the graphs represent the state of the heap, *i.e.*, the linked list implementing the stack, and thread objects describing the local states of all clients currently executing push and pop operations. The abstraction is obtained from the concrete transition system of Treiber's stack by ignoring the values of `next` pointers connecting the vertices in the linked

list of the stack. In this abstraction, there may still be unboundedly many elements in the stack as well as unboundedly many clients operating on the stack. However, since the list vertices are no longer connected, they can no longer form simple paths of arbitrary length in the heap graph. This abstraction preserve terminatino of the original stack but is not adatped to prove other properties like functional correctness. At this level of abstraction, `push` and `pop` become indistinguishable. Both operations have the same control-flow structure: they iteratively read the top of the stack and attempt to modify it until the `CAS` operation succeeds. The actual modification of the stack is non-deterministic in both operations.

In Fig. 5.3 we give the graph rewriting system for the depth-bounded abstraction of Treiber's stack. The initial state is a graph consisting of the vertex spawn, indicating that clients can be spawned, and the stack and its Top element which is some node. There are five rewrite rules. (i) The Spawn rule replaces a stack vertex with an identical stack vertex that is connected to a new vertex pc1 representing a client in an initial thread state before the `CAS` (pc1 refers to its owning stack via edge S). The dotted line indicates how the left-hand-side of the rule is replaced by the right-hand-side: the stack vertex on the left is replaced with the stack vertex on the right. (ii) Spawning may cease when the Nwaps rule is applied. Here, the spawn vertex is replaced with a nwaps vertex. The effect is that both the Spawn and Nwaps rules are disabled, but the remaining rules now become enabled. (iii) In the Prepare rule, a client reads the stack's Top pointer and prepares a new element (pointed to by x) to be pushed or popped onto the stack. There are then two cases that correspond to whether or not the `CAS` operation succeeds (depending on whether the local pointer `t` agrees with Top). (iv) In the Succeed case, the stack is updated to point to the new element and the old element is disregarded. This is a generalization that encompasses both push and pop. (v) Alternatively, the `CAS` may fail, as given by the Fail case. The stack is unchanged and the client forgets what it read and retries.

We can prove that Treiber's stack is lock-free by showing that its depth-bounded abstraction always terminates modulo a weak fairness constraint. The fairness constraint is that the Nwaps rule cannot be continuously enabled without being applied, i.e., a fair run of the system will only spawn finitely many clients. It does not matter whether we allow process spawning only in an initial phase (as in our model), or at any time.

The key contribution of this paper is a technique that automatically constructs a precise numerical abstraction of a depth-bounded system from a given inductive invariant of the system. We refer to this numerical abstraction as the *structural counter abstraction*. The structural counter abstraction then enables us to prove weakly fair termination of the system. Our approach utilizes existing reachability analyses for well-structured transition systems to obtain the inductive invariant, and existing termination analyses for numerical programs to prove termination of the structural counter abstraction. In the following, we explain the construction of the counter abstraction for Treiber's stack.

Nested graphs. Above we saw that graph rewrite rules transform a subcomponent of a concrete graph into another concrete graph. However, we will need to work with (potentially infinitely many) instances of graph subcomponents. So we instead work with *nested graphs* (formal definitions provided in Section 4.3)

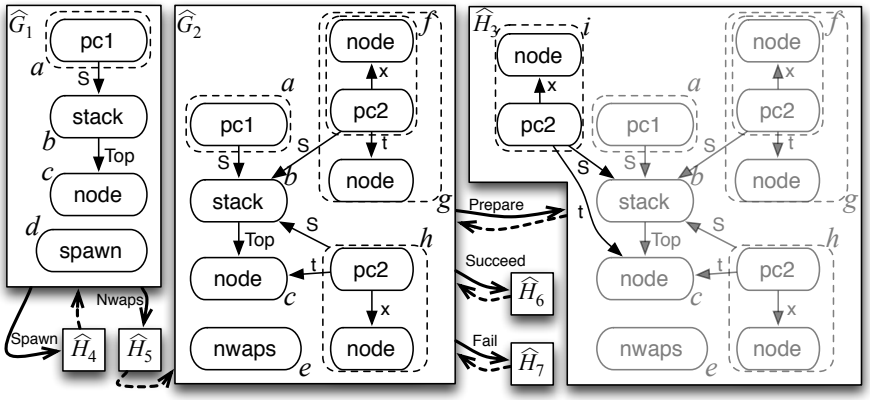
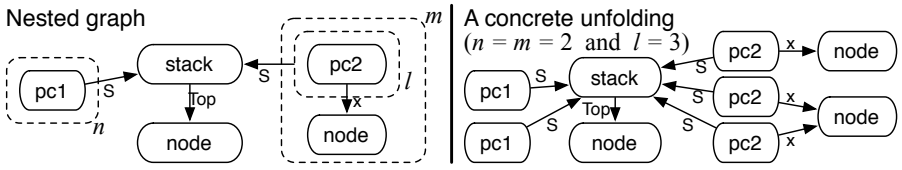


Figure 5.4: Structural counter abstraction for Treiber's stack. Numerical transition constraints are omitted for readability. Here the inductive invariant is given by nested graphs $\widehat{G}_1$ and $\widehat{G}_2$.

in which subcomponents are given counters that indicate an upper bound on how many times they may be duplicated. For Treiber's stack, consider this abstract graph on the left hand side:



The set of concrete graphs represented by this nested graph are those in which the dotted subcomponents are repeated some number of times but at most as many times as determined by the associated counter. For instance, the left dotted subgraph is repeated at most n times. A component may itself contain nested sub-components. An example of an unfolded concrete graph is given on the right hand side. Notice that the pc2 vertices occur at different frequencies per node vertex. Also note that counters always refer to the *total* number of copies of their component. This representation can be thought of as a more precise alternative to counter abstractions [96, 16, 37], in that we associate counters with nested graph components rather than merely program locations. We say that a nested graph $\widehat{G}_1$ is *covered* by nested graph $\widehat{G}_2$ if the set of concrete graphs obtainable from unfoldings of $\widehat{G}_2$ is contained within the set of concrete graphs obtainable from unfoldings of $\widehat{G}_1$. Determining whether $\widehat{G}_2$ covers $\widehat{G}_1$ is decidable and, as we will see, helps ensure that the structural counter abstraction can be effectively computed.

Obtaining the structural counter abstraction. We begin with a nested graph representation of the inductive invariant. For Treiber's stack the invariant is $\widehat{G}_1$ and $\widehat{G}_2$ in Fig. 5.4. This invariant (obtained, *e.g.*, via [117]) is a finite set of nested graphs and is an over-approximation of the reachable states of

the system. $\widehat{G}_1$ describes states in which spawning may still occur (indicated with a spawn vertex) and $\widehat{G}_2$ describes states in which spawning has ceased (indicated with a nwaps vertex) and arbitrarily many clients have performed Prepare, Suceed or Fail.

We begin to construct the structural counter abstraction by associating a counter variable with each subcomponent of each nested graph in the inductive invariant. For example in Fig. 5.4, we have established counter variables a, b, c, d with components of $\widehat{G}_1$ and additional counter variables e, f, g, h in $\widehat{G}_2$. In our generation of the structural counter abstraction, we leverage the fact that the invariant is *closed* under rewrite rules. That is, whenever we apply a rewrite rule to a nested graph $\widehat{G}$ in the inductive invariant, we obtain $\widehat{H}$ that is already covered by some other nested graph $\widehat{G}'$ in the invariant.

To construct the abstraction, we apply each rewrite rule, one at a time, for every possible match in one of the nested graphs in the invariant. For example, in Fig. 5.4 we can apply the Prepare rule as follows. We first unfold one instance of the pc1 vertex a in $\widehat{G}_2$, obtaining a separate pc1 vertex to which we apply the Prepare rule. This produces a new nested graph $\widehat{H}_3$ that extends $\widehat{G}_2$ with a new subgraph. We add a new counter variable i for this new subgraph in $\widehat{H}_3$. Notice that, because the inductive invariant is maximal, $\widehat{H}_3$ is covered by the existing graph $\widehat{G}_2$ (hence the dotted edge from $\widehat{H}_3$ to $\widehat{G}_2$). It is covered because the isomorphic subgraphs with associated counters i and h in $\widehat{H}_3$ can both be represented by the subgraph with associated counter h in $\widehat{H}_3$. From the point of view of the concrete graph transformation system, we can think of this covering edge as an ϵ-transition: every rewrite rule that is susequently applied to $\widehat{H}_3$ can also be applied to $\widehat{G}_2$. The structural counter abstraction is a numerical transition system that reflects the corresponding changes to the counter values when rewrite and covering edges between nested graphs are taken. There are several other possible instances where rules can be applied to this inductive invariant. (These involve graphs $\widehat{H}_4$, $\widehat{H}_5$, $\widehat{H}_6$, and $\widehat{H}_7$ which have been omitted for lack of space.) For example, one can apply the Spawn rule in $\widehat{G}_1$ and obtain $\widehat{H}_4$ which has two pc1 subgraphs. This new graph $\widehat{H}_4$ is, again, covered by $\widehat{G}_1$ and the two pc1 subgraphs can be merged into the pc1 subgraph in $\widehat{G}_1$.

Structural counter abstraction. The structural counter abstraction is represented as a simple control-flow graph program $\mathcal{N} = (\mathit{Locs}, s_0, \mathit{Vars}, \Delta)$. Here, *Locs* refers to the control locations. There is one location per nested graph in the inductive invariant, respectively, per nested graph obtained by application of a rewriting rule. The variables *Vars* are the structural counters in the nested graphs, and Δ is a set of commands that change the counter values according to the rewriting and covering steps. s_0 is the initial state. An excerpt of the structural counter abstraction for Treiber's stack that captures parts of Fig. 5.4 is as follows:

$$
\begin{array}{rcl}
\mathcal{N} & \equiv & (\{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7\}, s_0, \{a, b, c, \dots\}, \{(\ell_2, \delta_{23}, \ell_3), (\ell_3, \delta_{32}, \ell_2), \ldots\}) \text{ where} \\
s_0 & \equiv & (\ell_1, \{b \mapsto 1, c \mapsto 1, d \mapsto 1, \_ \mapsto 0\}) \\
\delta_{23} & \equiv & a' = a - 1 \wedge i' = i + 1 \wedge Id|_{\{a,i\}} \quad \delta_{32} \equiv h' = h + i \wedge i' = 0 \wedge Id|_{\{h,i\}}
\end{array}
$$

$Id|_S$ is the identity mapping on the variables, excluding those in S. The transition constraint δ_{23} captures the application of the Prepare rule on $\widehat{G}_2$ yielding

$\widehat{H_3}$. The transition constraint δ_{32} captures the covering transition from $\widehat{H_3}$ back to $\widehat{G_2}$. The initial state s_0 encodes the initial graph of the system which consists of one spawn, one stack, and one node vertex. The fairness constraints on the original system can be translated to fairness constraints on the structural counter abstraction in a straightforward manner. The structural counter abstraction we produce is then fit to be analyzed by an existing termination analysis tool such as Terminator [30] or ARMC [98].

Prototype. In Section 5.1.5 we describe the implementation in PICASSO that automates all steps required to prove fair termination of depth-bounded systems: generation of the inductive invariant, construction of the structural counter abstraction, and the final termination proof. It is able to prove fair termination of the Treiber stack model in less than 10 seconds. A simple counter abstraction that distinguishes only between processes at different control locations would yield a system with fair infinite traces. It is crucial to distinguish between the processes at location pc2 that may still succeed and those that are bound to fail. This is achieved by our more fine-grained structural counter abstraction.

5.1.3 Weakly Fair Termination of Depth-Bounded Systems

In this section, we formally define the class of systems that we consider in this paper and the type of questions that we answer about these systems.

Let $\mathcal{T} = (S, s_0, Act, \longrightarrow)$ be a labeled transition system. A *finite trace* π of $\mathcal{T}$ is a sequence $s_0\, a_0\, s_1\, a_1 \ldots a_{n-1}\, s_n$, with $s_i \in S$ and $a_i \in Act$ such that $s_i \xrightarrow{a_i} s_{i+1}$, for all $0 \leq i < n$; we define infinite traces $s_0\, a_0\, s_1\, a_1 \ldots$ correspondingly. We say that an action $a \in Act$ is *enabled* in a state s, if there exists a state s' such that $s \xrightarrow{a} s'$. Let $\mathcal{F} = \{A_0, \ldots, A_m\}$ be a set of subsets of Act. An infinite trace $s_0\, a_0\, s_1\, a_1 \ldots$ is *weakly fair* with respect to $\mathcal{F}$ if for every A_j, $0 \leq j \leq m$, there are infinitely many i such that $a_i \in A_j$ or there are infinitely many i such that no action in A_j is enabled in s_i.

Definition 15 *Given a transition system $\mathcal{T}$ and a finite set $\mathcal{F}$ of sets of actions of $\mathcal{T}$, the* weakly fair non-termination problem *asks whether there exists an infinite trace π of $\mathcal{T}$ such that π is weakly fair with respect to $\mathcal{F}$. We refer to the complementary problem as the* weakly fair termination problem *(WFT).*

Theorem 10 ([13]) *Weakly fair termination is undecidable for depth-bounded systems.*

5.1.4 Structural Counter Abstraction

We now see the formal treatment of how one obtains the structural abstraction of a given depth-bounded system and how it is used to give approximate answers to the weakly fair termination problem. For the remainder of this section, let $\mathcal{R}$ be a depth-bounded system. We systematically construct the structural counter abstraction of $\mathcal{R}$ from an inductive invariant of $\mathcal{R}$. However, we are not interested in arbitrary inductive invariants but in those that are downward-closed with respect to graph embedding. Since graph embedding is a wqo on depth-bounded graphs, such downward-closed sets are finite unions of ideals of

the embedding order, see Chapter 4. Each ideal can itself be finitely represented and we can compute symbolically the effect of transition on this representation. This enables us to compute a form of closure on the inductive invariant that yields the structural counter abstraction. We start by formalizing this representation of ideals.

Constructing the structural counter abstraction. In the following, we assume that $\widehat{\mathcal{I}}$ is a finite set of nested graphs such that $\gamma(\widehat{\mathcal{I}})$ is a downward-closed inductive invariant of $\mathcal{R}$. From $\widehat{\mathcal{I}}$ we then construct the structural counter abstraction. The precision of this abstraction depends on the precision of $\widehat{\mathcal{I}}$. The most precise downward-closed inductive invariant of $\mathcal{R}$ is the covering set $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. Unfortunately, this set is in general not computable for depth-bounded systems[1]. Thus, we employ existing the method from Chapter 3 that compute downward-closed inductive approximations of the covering set. In practice, these algorithms often yield precisely $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. This is confirmed by our experiments in Section 5.1.5.

Let G_0 be the initial graph of $\mathcal{R}$ and let $\widehat{G_0}$ be the nested graph obtained by equipping G_0 with a nesting level function mapping all nodes to 0. Further, let R be the set of rewriting rules of $\mathcal{R}$. We define a set of *rewriting edges* E_R as follows: $E_R = \left\{ (\widehat{G}, r, \widehat{H}) \mid \widehat{G} \in \widehat{\mathcal{I}}, r \in R, \widehat{H} \in \widehat{\mathcal{G}}, \widehat{G} \xrightarrow{\hat{r}} \widehat{H} \right\}$. That is, E_R describes the set of one step rule applications on the nested graphs in the inductive invariant. The set E_R is finite up to isomorphism of nested graphs. Next, define the set $\widehat{\mathcal{J}} = \left\{ \widehat{G_0} \right\} \cup \left\{ \widehat{H} \mid (\widehat{G}, r, \widehat{H}) \in E_R \right\}$. From the fact that $\widehat{\mathcal{I}}$ is an inductive invariant it follows that, for all $\widehat{H} \in \widehat{\mathcal{J}}$ there exists $\widehat{G} \in \widehat{\mathcal{I}}$ such that $\widehat{H} \sqsubseteq \widehat{G}$. Fix one such $\widehat{G}$ for each $\widehat{H} \in \widehat{\mathcal{J}}$ and let E_C be the set of all pairs $(\widehat{H}, \widehat{G})$. We call the elements of E_C *covering edges*. Let $\mathcal{E} = E_R \cup E_C$. In Fig 5.4, we saw this construction for the example of Treiber's stack starting with an inductive invariant. The solid edges between nested graphs correspond to rewrite edges and the dashed ones to covering edges. At the end of Section 5.1.2, we also saw an excerpt of the counter abstraction, next we describe how this is done in general.

The abstraction is a tuple $\mathcal{N} = (\mathit{Locs}, s_0, \mathit{Vars}, \Delta)$ where $\mathit{Locs} = \left\{ \ell_{\widehat{G}} \mid \widehat{G} \in \widehat{\mathcal{I}} \cup \mathcal{J} \right\}$ is a set of control locations, $\mathit{Vars} = \left\{ x_v \mid v \in V(\widehat{G}), \widehat{G} \in \mathcal{I} \cup \mathcal{J} \right\}$ is a set of counter variables , one for each vertex of a nested graph in $\mathcal{I} \cup \mathcal{J}$, and $\Delta = \{ \delta_e \mid e \in \mathcal{E} \}$ is a set of commands, one for each edge in $\mathcal{E}$. The command δ_e associated with an edge $e = (\widehat{G}, \widehat{H})$ is of the form $(\ell_{\widehat{G}}, \rho_e, \ell_{\widehat{H}})$ where ρ_e is a *transition constraint* over primed and unprimed versions of the variables in Vars. The initial state of $\mathcal{N}$ is $s_0 = (\ell_{\widehat{G_0}}, \eta_0)$ where η_0 is a function mapping counters to natural numbers and defined as $\eta_0(x_v) = 1$ if $v \in V(\widehat{G_0})$, and 0 otherwise. Further, let $\sigma_{\mathcal{R}} : \Delta \rightharpoonup R$ be a partial mapping defined as $\sigma_{\mathcal{R}}(\delta_e) = r$ if e is a rewriting edge for rule r.

The definition of the transition constraint δ_e for an edge $e \in \mathcal{E}$ depends on whether e is a rewriting or a covering edge. We first consider the case that e is a rewriting edge $(\widehat{G}, r, \widehat{H})$. In order to perform a rewrite (which only transforms level-0 vertices) we need to unfold the graph $\widehat{G}$. As mentioned in Lemma 16,

[1] This follows from the undecidability of place-boundedness of transfer nets [41].

this can be done efficiently giving us $\widehat{G} \rightsquigarrow^* K$. Each unfolding step gives a homomorphism, which can be composed together to give $h : K \to \widehat{G}$. Further, from the pushout we get a partial homomorphism $r' : K \rightharpoonup \widehat{H}$. Let V be the vertices of $\widehat{G}$, U the vertices of K, and W the vertices of $\widehat{H}$. Further, let U_0 be the level-0 vertices of K and define $\overline{U_0} = U \setminus U_0$. Similarly, let W_0 be the level-0 vertices of $\widehat{H}$. Then, the transition constraint ρ_e for e is given by the conjunction of the following constraints:

$$x_v = \sum_{u \in h^{-1}(v) \cap \overline{U_0}} x'_{r'(u)} + \left|h^{-1}(v) \cap U_0\right|, \quad \text{for all } v \in V \tag{5.1}$$

$$x'_w = 1, \quad \text{for all } w \in W_0 \tag{5.2}$$

$$y' = 0, \quad \text{for all } y \in \mathit{Vars} \setminus \{\, x_w \mid w \in W \,\} \tag{5.3}$$

During unfolding of $\widehat{G}$ to $\widehat{H}$, if some vertex v with count x_v is duplicated, then constraint (5.1) ensures that all counts for the duplicates sum up to x_v. Level-0 vertices get a special treatment, since they may be transformed by the rewrite rule. Similarly, (5.2) takes care of level-0 vertices in the rewritten graph. The constraint (5.3) encodes that only counters of vertices associated with the successor location have non-zero values. For covering edges $e = (\widehat{H}, \widehat{G})$, we use the inclusion mapping $\widehat{h} : \widehat{H} \to \widehat{G}$ between the two nested graphs to define the transition constraint δ_e. Let W be the vertices of $\widehat{G}$, W_0 the level-0 vertices of $\widehat{G}$, and V the vertices of $\widehat{H}$. The inclusion mapping encodes which vertices $v \in V$ are collapsed to a single vertex $w \in W$, yielding the constraint

$$x'_w = \sum_{v \in \widehat{h}^{-1}(w)} x_v, \quad \text{for all } w \in W \tag{5.4}$$

Then δ_e is the conjunction of constraint (5.4) and constraints (5.2) and (5.3), which are the same as in the case of a rewriting edge.

Finally, the fairness constraints $\mathcal{F}_\mathcal{R}$ of $\mathcal{R}$ can be translated to fairness constraints $\mathcal{F}_\mathcal{N}$ of $\mathcal{N}$ using the partial function $\sigma_\mathcal{R}$ as follows: $\mathcal{F}_\mathcal{N} = \{\, \sigma_\mathcal{R}^{-1}(R_i) \mid R_i \in \mathcal{F}_\mathcal{R} \,\}$.

The numerical abstraction induces a transition system $\mathcal{T}(\mathcal{N}) = (S, s_0, \Delta, \xrightarrow{\Delta})$ with states $S = \mathit{Locs} \times \mathbb{N}^{\mathit{Vars}}$, i.e., a program location along with an evaluation of the counters. The transition relation $\xrightarrow{\Delta}$ is as expected. The details of the following soundness theorem may be found in [13].

Theorem 11 (Soundness [13]) *If $(\mathcal{T}(\mathcal{R}), \mathcal{F}_\mathcal{R})$ has a weakly fair infinite trace, then so does $(\mathcal{T}(\mathcal{N}), \mathcal{F}_\mathcal{N})$.*

5.1.5 Evaluation

We implemented a prototype of our algorithm as an extension to the Picasso [117, 116] tool. Picasso takes as input a depth-bounded systems and computes a so called *abstract coverability tree* (ACT). The nodes of the ACT are nested graphs and its construction is similar to the Karp-Miller tree for Petri nets. The maximal nodes in the ACT form a downward-closed inductive invariant, $\widehat{\mathcal{I}}$, of the input system. From this invariant we generate a structural counter abstraction, $\mathcal{N}$, that is optimized and then analyzed with the ARMC [98] termination prover.

A naive implementation of the method described in Section 5.1.4 produced structural counter abstractions that were too big for current termination provers. For instance, for Treiber's stack, having one variable for each vertex of each nested graph in the inductive invariant and those obtained by applying rewrite rules led to an abstraction with over 170 variables and 40 transitions. We therefore optimized the generation of the abstraction to get a smaller counter program with the same termination properties. When we generate the constraints for a transition, we decompose the transition into three steps: unfolding, morphism, and covering. These steps lead to many intermediate locations and transitions. We eliminate the intermediate steps by using the quantifier elimination procedure for linear integer arithmetic in Princess [104]. We collect the constraints generated for each step and quantify away the variables at the intermediate locations. The resulting constraint describes a single transition with the same source and target locations as the original three-step transition, using only the variables at those locations. Furthermore, we observed that in many places constant values are assigned to the variables because they represent nodes on nesting level 0. We propagate the constant values using a combination of lightweight abstract interpretation and constraint propagation. We use an abstract domain that maps the variables to $\mathbb{N} \cup \bot$. A variable v is mapped to a value n in $\mathbb{N}$ when we can deduce that v is always equal to n, otherwise v is mapped to $\bot$.From the abstract fixed point we extract variable/value pairs and eliminate the variables by replacing them with their associated values. Lastly, instead of using one variable per node and graph, we reuse the variables across different graphs. The renaming is done by finding a minimal coloring of a graph where the nodes are variables and there is an edge between two nodes if the corresponding variables are used at the same location.For Treiber's stack, we reduced the abstraction to 6 variables and 4 transitions.

Transition predicates. We observed that ARMC finds easily the predicates that involve one or two variables, but not the predicates requiring more variables. Fortunately, ARMC can take transition predicates as part of its input. We manually give hints to Picasso in the form of variables names, usually corresponding to control-states. Those names are turned into transition predicates by summing the variables. For example, in the numerical abstraction of Treiber's stack we specified a simple predicate indicating that the sum of all the process counters was either unchanged or decreasing. We also implemented an heuristic that looks at a small number of cycles in the control flow graph of the abstraction and look for decreasing groups of variables by querying an SMT-solver. Then the corresponding transition predicates are generated. As the number of elementary cycles grows quickly with the size of the system, it is sometimes helpful to manually give hints about the predicates.

Results. Table 5.1 summarizes the results of our experiments. Our implementation is parallelized and ran on a server using 26 cores. Memory consumption was not an issue. We examined a collection of depth-bounded transition systems, including distributed processes and concurrent algorithms. The examples and the tool can be downloaded from the Picasso web site [116]. We applied our method to prove global progress properties of those systems. Fairness is used to limit the number of clients, requests, and failures. Details about the encoding

Example	#loc	#v	#t	$\widehat{\mathcal{I}}$	$\mathcal{N}$	ARMC	**Total**
Split/merge	4	3	9	1.5	6.8	0.1	8.4
Work stealing, 3 processors	4	4	20	1.7	13.1	0.2	15.0
Work stealing, parameterized	2	3	4	1.5	5.6	0.1	6.2
Compute server job queue	2	5	4	1.6	6.1	0.1	7.8
Chat room	5	34	80	9.8	61.3	5 min	6 min
Map reduce	6	10	15	2.0	8.8	0.2	11.0
Map reduce with failure	6	15	21	2.3	11.1	0.9	14.3
Treiber's stack (coarse-grained)	2	6	4	1.9	7.2	0.2	9.3
Treiber's stack (fine-grained)	3	14	13	2.7	14.2	1.2	17.1
Herlihy/Wing queue	3	16	25	3.8	24.9	6.5	34.2
Michael/Scott queue (dequeue only)	4	7	23	2.8	13.0	0.6	16.4
Michael/Scott queue (enqueue only)	7	15	53	3.8	43.7	7.6	55.1
Michael/Scott queue	9	31	224	25.0	265.0	3 wks	3 wks

Table 5.1: Experimental results. The columns show the number of locations, variables, and transitions in the counter abstraction, and the running times, in seconds, for computing the inductive invariant, constructing the abstraction, and for proving termination.

of fairness constraints can be found in Section 5.1.6. Our experiments show that our approach can quickly prove termination of complex systems. The structural counter abstraction is concise and maintains the necessary information in order to prove termination.

The split/merge example is a parallel computation where a master sends jobs to a pool of workers. We also proved termination of (non-)parameterized versions of a work stealing algorithm. From [62] we considered systems obtained from Scala implementations of a chat room and a map reduce algorithm (with and without failure). The chat room example shows the interaction of an unbounded number of clients, each one posting an arbitrary number of messages. For the map reduce examples, one version models failures and the other does not. the first version do not consider failures. In the second version, both mappers and reducers may crash after which they get recreated. To ensure termination, we added the constraint that processes cannot crash infinitely often without succeeding infinitely often. As shared memory examples, we considered the model of Treiber's stack [109] described in Section 5.1.2 as well as a more fine-grained variant with push and pop modeled independently. We analyzed a model of the Herlihy/Wing concurrent queue [64] which requires an additional fairness constraint to ensure that dequeue operations cannot execute without enqueue operations ever taking steps. This is needed because the dequeue operation retries if the queue is empty. With this additional fairness constraint we can also prove termination of this example. Finally, we also considered the Michael/Scott queue [84] where the order between the elements is abstracted. This example results in an abstraction that is very large for today's termination provers. We therefore also show the results for simpler models where enqueue and dequeue operations are considered in isolation.

5.1.6 Fairness constraints in PICASSO

Strong fairness. PICASSO and ARMC do not directly support weak-fairness. However, we can encode strong fairness constraints in the structural counter abstraction with *fairness counters*. These counters are either decremented by one or incremented by an arbitrary finite amount in the relevant transitions. For example, let t_1 and t_2 be two transitions and (t_1, t_2) a Streett fairness condition. The Streett condition tell us that if t_1 occurs infinitely often, then t_2 must also occur infinitely often. This can be encoded in the following way. Let v a fresh variable in the structural counter abstraction. In t_1 we add the constraint $v' = v - 1$ and in t_2 we add $v' \geq 0$. This principle generalizes to Rabin and (co-)Büchi fairness conditions.

Furthermore, we have extended PICASSO to support such fairness constraints, expressed in the input graph rewriting rules. The graph rewriting operation corresponding $v' \geq 0$ cannot be expressed by the formalism presented in Section **??**. For this purpose we extend the graph rewriting rules to allow nodes of non-zero nesting level on the right-hand-side of rewriting rules. This single modification in the parser was enough to express strong fairness constraints.

When weak and strong fairness meet. The main difference between strong an weak fairness is that in addition to transition firing, weak fairness also considers whether a transition is enabled. This cannot currently be expressed in PICASSO. Fortunately, the weak-fairness condition that we use are also expressible as strong fairness condition. The key insight is that we can statically know when a transition is enabled. For instance, the transition that spawns client is always enabled until the "Spawn to Nwaps" transition fires. We can rephrase this condition as a co-Büchi condition saying that the spawning of client does not occur infinitely often. For the experimental evaluation we use the co-Büchi condition.

5.1.7 Related Work

We present breifly other works related to the structural counter abstraction. A more thorough comparison with existing literature is available in [13].

DBS were first introduced by Meyer in [80] as a fragment of the π-calculus. In his paper, he showed that DBS are well-structured and that termination (without fairness constraints) is decidable. Termination without fairness has only limited practical applications because the initial state of the system is fixed. With a fixed initial state one cannot model systems with an infinite set of reachable states without losing termination, since we only consider finitely branching systems.

Numerical abstractions for the analysis of concurrent systems have been previously explored *e.g.* in [96, 16, 37]. Our work is a more precise alternative to these approaches. Rather than using a fixed number of counters (one for each program location) that count how many threads are at each program location in a given state, we use a reachability analysis to introduce counters that also take into account data dependencies between individual threads.

The idea of using reachability analyses to obtain numerical abstractions of programs whose states can be described by graphs is by itself not new. In

particular, such techniques have been studied for proving safety and liveness properties of heap manipulating programs [19, 99, 60]. Our technique differs substantially from these approaches in the way the numerical abstraction is computed. Specifically, our technique is based on *ideal abstractions* [117] for computing over-approximations of the covering sets of WSTS and it exploits the monotonicity of the analyzed system. Finally, the abstract domain of nested graphs can model unbounded recursive unfolding structures that are difficult to capture using traditional shape analysis domains.

An application of our results is proving nonblocking properties of concurrent algorithms. Others have considered approaches directly targeted on this goal. Gotsman *et al.* [57] describe a thread-modular proof technique.

5.2 Dynamic Package Interfaces

Programmers using software components have to follow protocols that specify when it is legal to call particular methods with particular arguments. For example, one cannot use an iterator over a set once the set has been changed directly or through another iterator. We generalize the notion of state-machine interfaces for single objects to group of interacting objects and formalize it in what we call *dynamic package interfaces* (DPI). We also give an methods to statically compute a sound abstraction of a DPI. States of a DPI represent (unbounded) sets of heap configurations and edges represent the effects of method calls on the heap. We introduce a novel heap abstract domain based on depth-bounded systems to deal with potentially unboundedly many objects and their relations. We have implemented our algorithm and show that it is effective in computing representations of common patterns of package usage, such as relationships between viewer and label, container and iterator, and JDBC statements and cursors.

This part is joint work with Shahram Esmaeilsabzali, Rupak Majumdar, and Thomas Wies. It is still work in progress and has not yet been published.

5.2.1 Motivation

Modern object-oriented programming practice uses packages to encapsulate components, allowing programmers to use these packages through well-defined application programming interfaces (APIs). While programming languages such as Java or C# provide a clear specification of the *static* APIs of components in terms of classes and their (typed) methods, there is usually no specification of the *dynamic* behavior of packages that constrain the temporal ordering of method calls on different objects. For example, one should invoke the *lock* and *unlock* methods of a lock object in alternation; any other sequence raises an exception. More complex constraints connect method calls on objects of different classes. For example, in the Java Database Connectivity (JDBC) package, a ResultSet object, which contains the result of a database query executed by a Statement object, should first be closed before its corresponding Statement object can execute a new query.

In practice, such temporal constraints are not formally specified, but explained through informal documentation and examples, leaving programmers susceptible to bugs in the usage of APIs. Being able to specify dynamic interfaces for components that capture these temporal constraints clarify constraints imposed by the pacakge on client code. Moreover, program analysis tools may be able to automatically check whether the client code invokes the component correctly according to such an interface.

Previous work on mining dynamic interfaces through static and dynamic techniques has mostly focused on the single-object case (such as a lock object) [111, 7, 77, 63, 56], and rarely on more complex collaborations between several different classes (such as JDBC clients) interacting through the heap [102, 91, 100]. In this paper, we propose a systematic, static approach for extraction of dynamic interfaces from existing object-oriented code. Our work is closely related to the Canvas project [102]. Our new formalization can express structures than could not be expressed in previous work (i.e. nesting of graphs).

More precisely, we work with *packages*, which are sets of classes. A configuration of a package is a concrete heap containing objects from the package as well as references among them. A *dynamic package interface* (DPI) specifies, given a history of constructor and method calls on objects in the package, and a new method call, if the method call can be executed by the package without causing an error. In analogy with the single-object case, we are interested in representations of DPIs as finite state machines, where states represent sets of heap configurations and transitions capture the effect of a method call on a configuration. Then, a method call that can take the interface to a state containing erroneous configurations is not allowed by the interface, but any other call sequence is allowed.

The first stumbling block in carrying out this analogy is that the number of states of an object, that is, the number of possible valuations of its attributes, as well as the number of objects living in the heap, can both be unbounded. As in previous work [63, 102], we can bound the state space of a single object using *predicate abstraction*, that tracks the abstract state of the object defined by a set of logical formulas over its attributes. However, we must still consider unboundedly many objects on the heap and their inter-relationships. Thus, in order to compute a dynamic interface, we must address the following challenges.

1. The first challenge is to define a finite representation for possibly unbounded heap configurations and the effect of method calls. For single-object interfaces, states represent a subset of finitely-many attribute valuations, and transitions are labeled with method names. For packages, we have to augment this representation for two reasons. First, the number of objects can grow unboundedly, for example, through repeated calls to constructors, and we need an abstraction to represent unbounded families of configurations. Second, the effect of a method call may be different depending on the receiver object and the arguments, and it may update not only the receiver and other objects transitively reachable from it, but also other objects that can reach these objects.

2. The second challenge is to compute, in finite time, a dynamic interface using the preceding representation. For single-object interfaces [7, 63], interface construction roughly reduces to abstract reachability analysis against the most general client (a program that non-deterministically calls all available methods in a loop). For packages, it is not immediate that abstract reachability analysis will terminate, as our abstract domains will be infinite, in general.

We address these challenges as follows. First, we describe a novel shape domain for finitely representing infinite sets of heap configurations as recursive unfoldings of nested graphs. Technically, our shape domain combines predicate abstraction [105, 97], for abstracting the internal state of objects, with sets of depth-bounded graphs represented as nested graphs, cf. Chapter 4. Each node of a nested graph is labelled with a valuation of the abstraction predicates that determine an equivalence class for objects of a certain class.

Second, we describe an algorithm to extract the DPI from this finite state abstraction based on abstract reachability analysis of depth-bounded graph rewriting systems. We use the insight that the finite state abstraction can be reinterpreted as a numerical program is a way resembling to the counter abstraction of

Section 5.1. The analysis of this numerical program yields detailed information about how a method affects the state of objects when it is called on a concrete heap configuration, and how many objects are effected by the call.

We have implemented our algorithm on top of PICASSO and we have applied our method on a set of standard benchmarks written in a Java-like OO language, such as container-iterator, JDBC query interfaces, etc. In each case, we show that our algorithm produces an intuitive DPI for the package within a few seconds. This DPI can be used by a model checking tool to check conformance of a client program using the package to the dynamic protocol expected by the package.

5.2.2 Overview

We illustrate our approach through a simple example.

Example. Figure 5.5 shows two classes Viewer and Label in a package, adapted from [91], and inspired by an example from Eclipse's ContentViewer and IBaseLabelProvider classes. A Label object throws an exception if its run or dispose method is called after the dispose method has been called on it. There are different ways that this exception can be raised. For example, if a Viewer object sets its f reference to the same Label object twice, after the second call to set, the Label object, which is already disposed, raises an exception. As another example, for two Viewer objects that have their f reference attributes point to the same Label object, when one of the objects calls its done method, if the other object calls its done method an exception will be raised. An *interface* for this package should provide possible configurations of the heap when an arbitrary client uses the package, and describe all usage scenarios of the public methods of the package that do not raise an exception.

Dynamic Package Interface. Intuitively, an interface for a package summarizes all possible ways for a client to make calls into the package (i.e., create instances of classes in the package and call their public methods). In the case of single-objects, where all attributes are scalar-valued, interfaces are represented as finite-state machines with transitions labeled with method calls [111, 7, 63]. Each state s of the machine represents a set $[\![s]\!]$ of states of the object, where a state is a valuation to all the attributes. (In case there are infinitely many states, the methods of [7, 63] abstract the object relative to a finite set of predicates, so that the number of states is finite.) An edge $s \xrightarrow{m} t$ indicates that calling the method $m()$ from any state in $[\![s]\!]$ takes the object to a state in $[\![t]\!]$. Some states of the machine are marked as errors: these represent inconsistent states, and method calls leading to error states are disallowed.

Below, we generalize such state machines to packages.

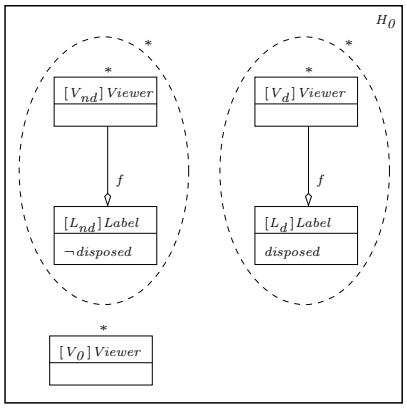
States: Ideals over Shapes. The first challenge is that the notion of a state is more complex now. First, there are arbitrarily many states: for each n, we can have a state with n instances of Label (e.g., when a client allocates n objects of class Label); moreover, we can have more complex configurations where there are arbitrarily many viewers, each referring to a single Label, where the Label

```
class Viewer {
  Label f;
  public void Viewer() {
    f := null; }
  public void run() {
    if (f != null) f.run(); }
  public void done() {
    if (f != null) f.dispose(); }
  public void set(Label l){
    if (f != null) f.dispose();
    f := l; }
}
```

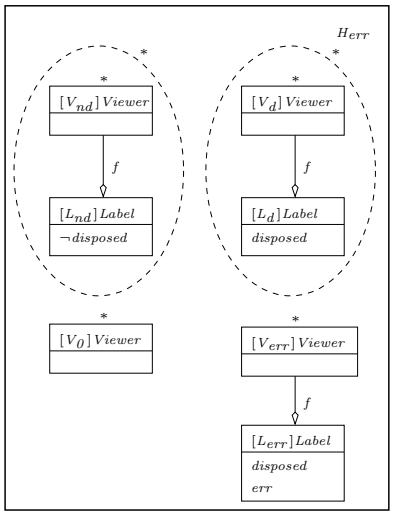
(a) The Viewer class

```
class Label {
  boolean disposed;
  public void Label() {
    disposed := false;
  }
  protected void run() {
    if (disposed) throw new Exception(); }
  protected void dispose() {
    if (disposed) throw new Exception();
    disposed := true; }
}
```

(b) The Label class



(c) Abstract heap H_0



(d) Abstract heap H_{Err}

Figure 5.5: A package consisting of Viewer and Label classes and its two abstract heaps

may have disposed = true or not. We call sets of (potentially unbounded) heap configurations *abstract heaps*.

Our first contribution is a novel finite representation for abstract heaps. We represent abstract heaps using a combination of *parametric shape analysis* [105] and *ideal abstractions for depth-bounded systems*. As in shape analysis, we fix a set of unary predicates, and abstract each object w.r.t. these predicates. For example, we track the predicate `disposed`(l) to check if an object l of type Label has disposed set to true. Additionally, we track references between objects by representing the heap as a nested graph whose nodes represent predicate abstractions of objects and whose edges represent references from one object to another. Unlike in parametric shape analysis, references are always determinate and the abstract domain is therefore still infinite.

Figure 5.5c shows an abstract heap H_0 for our example. There are five nodes in the abstract heap. Each node is labelled with the name of its corresponding class and a valuation of predicates, and represents an object of the specified

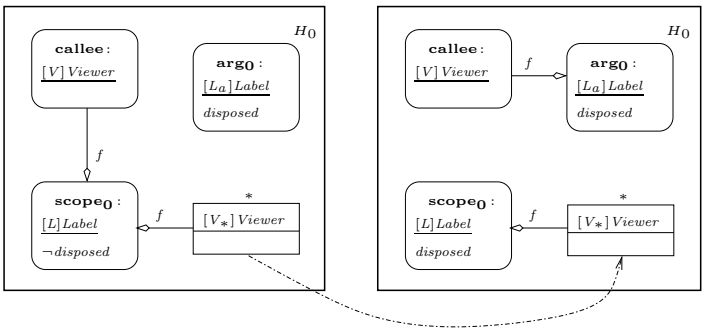
class whose state satisfies the predicates. Some nodes have an identifier in square brackets in order to easily refer to them. For instance, V_{nd} represents a Viewer object and L_d represents a Label object for which disposed is true. Edges between nodes show field references: the edge between the V_d and L_d objects that is labeled with f shows that objects of type V_d have an f field referring to some object of type L_d. Finally, nodes and subgraphs can be marked with a "*". Intuitively, the "*" indicates an arbitrary number of copies of the pattern within the scope of the "*". For example, since V_d is starred, it represents arbitrarily many (including zero) Viewer objects sharing a Label object of type L_d. Similarly, since the subgraph over nodes V_d and L_d is starred, it represents configurations with arbitrarily many Label objects, each with (since V_d is starred as well) arbitrarily many viewers associated with it.

Figure 5.5d shows a second abstract heap H_{err}. This one has two extra nodes in addition to the nodes in H_0, and represents erroneous configurations in which the Label object is about to throw an exception in one of its methods. (We set a special error-bit whenever an exception is raised, and the node with object type L_{err} represents an object where that bit is set.)

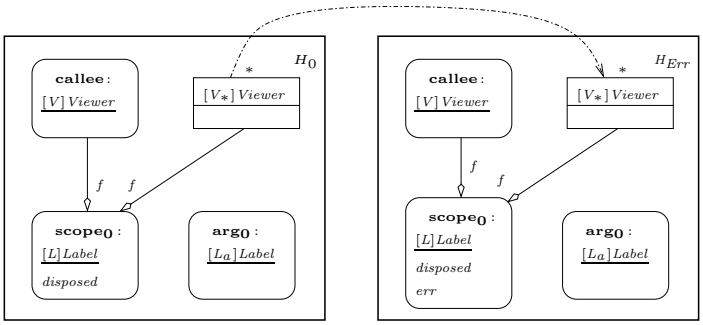
Technically (see Sections 5.2.9 and 5.2.9), nested graphs represent ideals of downward-closed sets (relative to graph embedding) of configurations of depth-bounded predicate abstractions of the heap. While the abstract state space is infinite, it is well-structured structured, and we can compute the covering set, c.f. Chapter 3.

Transitions: Object Mappings. Suppose we get a finite set $\mathcal{S}$ of abstract heaps represented as above. The second challenge is that method calls may have parameters and may change the state of the receiver object as well as objects reachable from it or even objects that can reach the receiver. As an example, consider a set container object with some iterators pointing to it. Removing an element through an iterator can change the state of the iterator (it may reach the end), the set (it can become empty), as well as other iterators associated with the set (they become invalidated and may not be used to traverse the set). Thus, transitions cannot simply be labeled with method names, but must also indicate which abstract objects participate in the call as well as the effect of the call on the abstract objects. The interface must describe the effect of the heap in all cases, and all methods. In our example, we can enumerate 14 possible transitions from H_0. To complete the description of an interface, we have to (1) show how a method call transforms the abstract heap, and (2) ensure that each possible method call from each abstract heap in $\mathcal{S}$ ends up in an abstract heap also in $\mathcal{S}$.

Consider invoking the set method of a viewer in the abstract heap H_0. There are several choices: one can choose in Figure 5.5c an object of type V_d, V_{nd}, or V_0 as the callee, and pass it an object of type L_d or L_{nd}. Note that the method call captures the scenario in which one *representative* object is chosen from each node and the method is executed. Recall that, because of stars, a single node may represent multiple objects. Figure 5.6a shows how the abstract heap is transformed if we choose a viewer pointing to a label which is not disposed as the callee and pass it a disposed label as argument. The box on the left specifies the *source* heap before the method call and the box on the right specifies the *destination* heap after the method call. A representative object in a method



(a) Object mapping for $V.\texttt{set}(L_a)$



(b) Object mapping for $V.\texttt{set}(L_a)$

Figure 5.6: Two object mappings for the package in Figure 5.5

call is graphically shown by a rounded box and has a *role name* that prefixes its object type. The source heap includes three representative objects with role names: **callee**, $\mathbf{arg_0}$, and $\mathbf{scope_0}$. The **callee** and $\mathbf{arg_0}$ role names determine the callee object and the parameter object of the method call, respectively. The $\mathbf{scope_0}$ is a *Label* object that is in the *scope* of the method call: i.e., the method call affects its type or the valuation of its predicates. Lastly, there is a fourth object in the left box that is not a single representative, but a starred object V_* that represents all viewers other than the callee object that reference the object with role $\mathbf{scope_0}$. The following properties hold: first, both the source and the destination of the transition are H_0, hence, the method call transforms objects in the abstract heap H_0 back to H_0; second, any object in H_0 that is not mentioned in the source box is untouched by the method call. Third, each object in the left box is mapped to another representative object in the right box: The representative objects can be traced via their role names while the other objects via the arrows that specifies their new types (to model non-determinism, such an arrow can be a multi-destination arrow). Thus, $V.\texttt{set}(L_a)$ transforms the callee object by changing its reference f to the L_a object that was the parameter of the method call. The object L that the callee referenced before the method call get the value of its disposed predicate changed to true after the method call. All other objects represented by V_* that reference L continue referencing that object.

The second transition, in Figure 5.6b, shows what happens if set is called on V_d with any label. This time, an error occurs, since the method call tries to

dispose an already disposed label. This is indicated by a transformation to the error node H_{err}, and thus, is not allowed in the interface.

Algorithm for Interface Computation. Our second contribution is an algorithm and a tool for computing the dynamic package interfaces in form of a state machine, as described above. Conceptually, the DPI of a package is computed in two steps: (i) computing the *covering set* of the package, which includes all possible configurations of the package, in a finite form; and (ii) computing the object mappings of the package using the covering set.

Computing the Covering Set. We introduce three layers of abstraction to obtain an overapproximation of the covering set of a package in a finite form. First, using a fixed set of predicates over the attributes of classes, we introduce a predicate abstraction layer. Second, we remove from this predicate abstraction those reference attributes of classes that can create a chain of objects with an unbounded length; these essentially correspond to recursive data structures, such as linked lists. We call these two abstraction layers the *depth-bounded abstraction*. The soundness of depth-bounded abstraction follows soundness arguments similar to the ones for classic abstract interpretation. However, unlike the classic abstract interpretation of non-object–oriented programs, the depth-bounded abstraction of object-oriented packages does not in general result in a finite representation; e.g., we may still have an unbounded number of iterator and set objects, with each iterator object being connected to exactly one set object.

Our third abstraction layer, namely, *ideal abstraction*, ensures a finite representation of the covering set of a package. The domain of ideal abstraction is essentially the same as the domain of nested graphs. The key property of this abstraction layer is that it can represent an unbounded number of depth-bounded objects as the union of a finite set of ideals, each of which itself is represented finitely. The soundness of this abstraction layer follows from the general soundness result for the ideal abstraction of depth-bounded systems.

To compute the covering set of a package, we use a notion of *most general client*. Intuitively, the most general client [63] runs in an infinite loop; in each iteration of the loop, it non-deterministically either allocates a new object, or picks an already allocated object, a public method of the object, a sequence of arguments to the method, and invokes the method call on the object.

Computing the Object Mappings. The object mappings are computed using the covering set as starting point. To compute the object mappings we let the most general client run one more time using the covering set as starting state of the system. During that run we record what effect the transitions have. For a particular transition we record, among other information, what are the starting and ending abstract heaps and the corresponding *unfolded*, representative objects. The nodes of the unfolded heap configurations are tagged with their respective roles in the transition. Finally, we record how the objects are modified and extract the mapping of the object mapping.

In our example, there are two maximal nodes: H_0 and H_{err}, where H_{err} denotes the error configurations. H_0 and H_{err} together represent the covering set of the package. Accordingly, the interface shows that H_0 captures the "most

general" abstract heap in the use of this package; each "correct" method call corresponds to an object mapping over H_0. We omit showing the remaining 12 object mappings of the interface.

5.2.3 Concrete Semantics

We now present a core OO language.

Syntax. For a set of symbols X (including variables), we denote by $\mathsf{Exp}.X$ and $\mathsf{Pred}.X$ the set of expressions and predicates respectively, constructed with symbols drawn from X. We assume there are two special variables this and null.

In our language, a package consists of a collection of class definitions. A class definition consists of a class name, a constructor method, a set of fields, and a set of method declarations partitioned into public and protected methods. A constructor method has the same name as the class, a list of typed arguments, and a body. We assume fields are typed with either a finite scalar type (e.g., Boolean), or a class name. The former are called *scalar* fields and the latter *reference* fields. Intuitively, reference fields refer to other objects on the heap. Methods consist of a signature and a body. The signature of a method is a typed list of its arguments and its return value. The body of a method is given by a control flow automaton over the fields of the class. Intuitively, any client can invoke public methods, but only other classes in the package can invoke protected ones.

A control flow automaton (CFA) over a set of variables X and a set of operations $\mathsf{Op}.X$ is a tuple $F = (X, Q, q_0, q_f, T)$, where Q is a finite set of *control states*, $q_0 \in Q$ (resp. $q_f \in Q$) is a designated initial state (resp. final state), and $T \subseteq Q \times \mathsf{Op}.X \times Q$ is a set of edges labeled with operations.

For our language, we define the set $\mathsf{Op}.X$ of *operations* over X to consist of: (i) *assignments* $\mathsf{this}.x := e$, where $x \in X$ and $e \in \mathsf{Exp}.X$; (ii) *assumptions*, $\mathrm{assume}(p)$, where $p \in \mathsf{Pred}.(\{\mathsf{this}\} \cup X)$, (iii) *construction* $\mathsf{this}.x = \mathsf{new}(C(\bar{a}))$, where C is a class name and $\bar{a}$ is a sequence in $\mathsf{Exp}.X$, and (iv) *method calls* $\mathsf{this}.x := \mathsf{this}.y.m(\bar{a})$, where $x, y \in X$.

Formally, a class $C = (A, c, M_p, M_t)$, where A is the set of fields, c is the constructor, M_p is the set of public methods, and M_t is the set of protected methods. We use C also for the name of the class. A package P is a set of classes.

We make the following assumptions. First, all field and method names are disjoint. Second, each class has an attribute ret used to return values from a method to its callers. Third, all CFAs are over disjoint control locations. Fourth, a package is well-typed, in that assignments are type-compatible, called methods exist and are called with the right number and types of arguments, etc. Finally, it is not clear how the pushdown system and depth-bounded systems mixes and whether there exists an bqo that may accomodate both. Therefore, we omit recursive method calls from our the analysis.

A *client* I of a package P is a class with exactly one method main, such that (i) for each $x \in I.A$, we have the type of x is either a scalar or a class name from P, (ii) in all method calls $\mathsf{this}.x = \mathsf{this}.y.m(\bar{a})$, m is a public method of its class, and (iii) edges of main can have the additional *non-deterministic assignment* $\mathrm{havoc}(\mathsf{this}.x)$. An OO program is a pair (P, I) of a package P and a client I.

Concrete Semantics. We give the semantics of an OO program as a labeled transition system. A *transition system* $\mathcal{S} = (X, X_0, \rightarrow)$ consists of a set X of states, a set $X_0 \subseteq X$ of initial states, and a transition relation $\rightarrow \subseteq X \times X$. We write $x \rightarrow x'$ for $(x, x') \in \rightarrow$.

Fix an OO program $S = (P, I)$. It induces a transition system $(\mathit{Conf}, U_0, \rightarrow)$, with configurations Conf, initial configurations U_0, and transition relation $\rightarrow$ as follows.

Let $\mathcal{O}$ be a countably infinite set of *object identifiers* (or simply objects) and let $\mathit{class} : O \rightarrow P \cup \{I, \mathit{nil}\}$ be a function mapping each object identifier to its class. A *configuration* $u \in \mathit{Conf}$ is a tuple $(O, \mathit{this}, q, \nu, \mathit{st})$, where $O \subseteq \mathcal{O}$ is a finite set of currently allocated *objects*, $\mathit{this} \in O$ is the *current object* (i.e., the receiver of the call to the method currently executed), q is the *current control state*, which specifies the control state of the CFA at which the next operation will be performed, ν is a sequence of triples of object, variable, and control location (the program stack), and st is a *store*, which maps an object and a field to a value in its domain. We require that O contains a unique *null* object *null* with $\mathit{class}(\mathit{null}) = \mathit{nil}$. We denote by Conf the set of all configurations of S.

The set of *initial configurations* $U_0 \subseteq \mathit{Conf}$ is the set of configurations $u_0 = (\{\mathit{null}, o_I\}, \mathit{this}, \mathsf{main}.q_0, \varepsilon, \mathit{st})$ such that (i) $\mathit{class}(o_I) = I$, (ii) the current object $\mathit{this} = o_I$, (iii) the value of all reference fields of all objects in the store is *null* and all scalar fields take some default value in their domain, and (iv) the control state is the initial state of the CFA of the main method of I and the stack is empty.

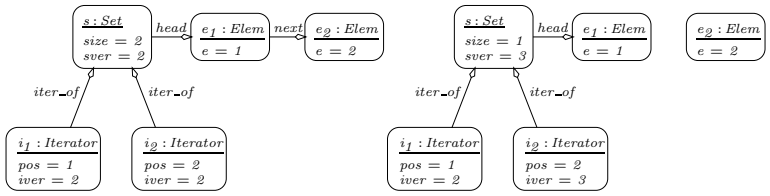
Given a store, we write $\mathit{st}(e)$ and $\mathit{st}(p)$ for the value of an expression e or predicate p evaluated in the store st, computed the usual way.

The transitions in $\rightarrow$ are as follows. A configuration $(O, \mathit{this}, q, \nu, \mathit{st})$ moves to configuration $(O', \mathit{this}', q', \nu', \mathit{st}')$ if there is an edge (q, op, q') in the CFA of q such that

- $\mathit{op} = \mathsf{this}.x := e$ and $O' = O$, $\mathit{this}' = \mathit{this}$, $\nu' = \nu$, and $\mathit{st}' = \mathit{st}[(\mathit{this}, x) \mapsto \mathit{st}(e)]$.
- $\mathit{op} = \mathrm{assume}(p)$ and $O' = O$, $\mathit{this}' = \mathit{this}$, $\nu' = \nu$, $\mathit{st}(p) = 1$, and $\mathit{st}' = \mathit{st}$.
- $\mathit{op} = \mathsf{this}.x := \mathsf{this}.y.m(\bar{a})$ and $O' = O$, $\mathit{this}' = \mathit{this}$, $\nu' = (\mathit{this}, x, q')\nu$, and $q' = m.q_0$, and the formal arguments of m are assigned values $\mathit{st}(\bar{a})$ in the store.
- $\mathit{op} = \mathsf{this}.x := \mathsf{new}(C(\bar{a}))$ and $O' = O \uplus \{o\}$ for a new object o with $\mathit{class}(o) = C$, $\mathit{this}' = o$, $\nu' = (\mathit{this}, x, q')\nu$, and $q' = c.q_0$ for the constructor c of C, and the formal arguments of c are assigned values $\mathit{st}(\bar{a})$ in the store.
- $\mathit{op} = \mathrm{havoc}(\mathsf{this}.x)$: $O' = O$, $\mathit{this}' = \mathit{this}$, and $\mathit{st}' = \mathit{st}[(\mathit{this}, x) \mapsto v]$, where v is some value chosen non-deterministically from the domain of x.

Finally, if q is the final node of a CFA and $\nu = (o, x, q)\nu'$, and the configuration $(O, \mathit{this}, q, \nu, \mathit{st})$ moves to $(O, o, q, \nu', \mathit{st}')$, where $\mathit{st}' = \mathit{st}[o.x \mapsto \mathit{st}(\mathit{this}.\mathsf{ret})]$. If none of the rules apply, the program terminates.

To model error situations, we assume that each class has a field *err* which is initially 0 and set to 1 whenever an error is encountered (e.g., an assertion is violated). An error configuration is a configuration u in which there exists an



(a) Before removing an element via i_2 (b) After removing the element

Figure 5.7: Two configurations of set and iterator package

object $o \in u.O$ such that $o.err = 1$. An OO program is *safe* if it does not reach any error configuration.

Example 9 *Figure 5.7 depicts two configurations for a set of objects belonging to a "set and iterator" package. For the sake of brevity, we do not show the code for this package, but the functionality of the package is standard. The package has three classes, namely,* `Set`*,* `Iterator`*, and* `Elem`*. The* `Elem` *class can create a linked list to store the elements of a* `Set` *object. An* `Iterator` *object is used to traverse the elements of its corresponding* `Set` *object via its* `pos` *attribute as an index. It can also remove an element of the* `Set` *object through its* `remove` *method. An* `Iterator` *object can perform these operations only if it has the same* version *as its corresponding* `Set` *object. The* `Iterator` *version is stored in the* `iver` *field and the* `Set` *version in* `sver`*. In this example, we focus on the* `remove` *method. The* `remove` *method of an* `Iterator` *object invokes the* `delete` *method of its corresponding* `Set` *object, passing its* `pos` *attribute as a parameter. The* `delete` *method, in turn, deletes the* `pos`*th* `Elem` *object that is accessible through its* `head` *attribute. The version attributes of both the* `Iterator` *and* `Set` *objects are incremented, while the version attributes of other* `Iterator` *objects remain the same. The two configurations in Figure 5.7 are abbreviated to show only the information relevant to this example.*

The configuration

$$u = (\{s, i_1, i_2.e_1, e_2\}, s, ., \langle(i_2, ., .)\rangle, \{((i1, iver), 2), ((i2, iver), 2), \cdots\}),$$

depicted in Figure 5.7(a), is one of the configurations during the execution of i_2.`remove`*, namely the configuration immediately after executing* *this*.`iter_of.delete(`*this*.`pos)`*. After a number of steps, the computation reaches configuration*

$$u' = (\{s, i_1, i_2.e_1, e_2\}, s, ., \varepsilon, \{((i1, iver), 2), ((i2, iver), 3), \cdots\}),$$

depicted in Figure 5.7(b), which is the configuration after o_2.`remove()` *has completed and the control has returned to the client,* I*. At* u'*,* i_2 *still has the same version (*$i_2.iver$*) as* s*, (*$s.sver$*), but* i_1 *has a different version now. Thus,* i_1 *cannot traverse or remove an element of* s *any more.*

5.2.4 Dynamic Package Interface (DPI)

For a package P, its *dynamic package interface* is essentially a set of *abstract object graphs* representing heap configurations together with a set of *object mappings* over them, one for each distinct method invocation.

Each abstract object graph represents an unbounded number of heap configurations. An object mapping for a method invocation specifies how the objects of a source heap configuration are transformed to the objects of a destination heap configuration. Object mappings use an extended notion of object graphs with *role labelling* to identify the callee and the arguments of the method calls. Up to isomorphism, the set of object mappings of a DPI specify the effect of all possible public method calls on distinct heap configurations of a package.

In the remainder of this section, in Section 5.2.6, we present the notions of abstract object graphs and unfolded object graphs, followed by the notion of object mapping, in Section 5.2.7. In Section 5.2.8, we present DPI formally.

5.2.5 Most General Client and Granularity of the DPI

To simplify the presentation of DPI, we currently do not consider concurrency. This means that the most general client is a sequential program, i.e. one method is called only when the previous method returned. Therefore, the states of the DPI are only at the boundary of method calls in the most general client. In those configuration the stack is empty and can be omitted. This explains why the abstract object graphs do not have a stack.

Just before a call, the most general client guesses the object on which to call the method and the parameter of the method. These steps are also not explicitly represented in the DPI, but in the unfolded objects graphs attached to the object mappings. There are roles for the callee and the arguments of the method. Furthermore, the object mappings (Section 5.2.7) represent the whole execution of methods as a single step. Obviously, this assumes that the methods are terminating.

5.2.6 Abstract Object Graphs

An *abstract object graph* H over a package P is a tuple (AL, AR, O, l, st, nl) with

- AL and AR: sets of *object labels* and *reference fields*, respectively,
- O: a set of *object nodes* identifiers,
- $st : (O \times AR) \nrightarrow O$ the *reference edge* function,
- $l : O \to AL$ the *object labelling* function,
- $nl : O \to \mathbb{N}_0$, the *nesting level* function.

We call an object node with nesting level zero an *object instance* and otherwise call it an *abstract object*. An abstract object represents an unbounded number of object instances. If an object node is connected via a reference label to another object node in st, it means that one or more object instances (depending on their relative nesting levels) in the source node have reference attributes pointing to an object instance in the destination node. We denote by *class* the function from AL to P that extracts the class information from a label.
An abstract object graph is *well-formed* if: $\forall(o_1, r, o_2), st(o_1, r) = o_2 \Rightarrow nl(o_1) \geq nl(o_2)$. This constraint is necessary because it should not be possible for an object instance to reference more than one object instance with the same reference attribute.

Abstract object graphs are a more structured version of the labeled digraphs used previously, in the sense that they have a semantics that can directly be related to the concrete semantics of the simple OO language introduced before. Furthermore, since they are a specific case of the graphs used before, the previous results also apply to them.

Example 10 *Let us consider the graph in Figure 5.5c, which is an abstract object graph. Let the object node labelled with* $[V_{nd}]$ *Viewer be denoted by* x*, then* V_{nd} *is the identifier that we use to refer to* x *in the description, and we have* $l(x) = Viewer$*, which tells the class of* x *and the predicates and their valuation (none in this case). Finally, we have* $nl(x) = 2$*.*

An *unfolded object graph* G over P is a tuple $(AL, R, AR, O, l, st, n, nl)$ where

- (AL, AR, O, l, st, nl) is an abstract object graph over P,
- R is a set of *object role* labels, and
- $n : R \to O$ is a *role name* function.

An object of an unfolded object graph may have a role name in addition to its label. A role name indicates the fixed responsibility of the object instance during a method call.

An unfolded object graph can be obtained from an abstract object graph by unfolding the graph and adding a role function. The unfolding step copies a subgraph with nesting level greater than 0 and decreases the nesting level of the copy by one. This process is repeated until all the roles can be assigned to object instances.

An unfolded object graph is *well-formed* if:

i. The role name function labels all object instances and only them: $\forall o \in O,\ \exists r \in R,\ n(r) = o \Leftrightarrow nl(o) = 0$;

ii. the role name function is injective: $\forall r_1, r_2 \in R,\ n(r_1) = n(r_2) \Rightarrow r_1 = r_2$.

Henceforth, we consider only well-formed abstract object graphs and well-formed unfolded object graphs. We denote the set of all abstract object graphs and the set of all unfolded object graphs over P as $\mathcal{H}_P$ and $\mathcal{G}_P$, respectively.

In our analysis, each unfolded object graph $G \in \mathcal{G}_P$ corresponds to a unique abstract object graph $H \in \mathcal{G}_H$, as we will see in the next section. We assume the *source* function $src : \mathcal{G}_P \to \mathcal{H}_P$, which determines the abstract object graph of an unfolded object graph.

Example 11 *Let us consider the graph inside the box in the left hand side of Figure 5.6a, which is an unfolded object graph whose source is* H_0 *in Figure 5.5c. Let the object node labelled with* **callee**$:[V_{nd}]$ *Viewer be* x*, then* $l(x) = Viewer$*,* $nl(x) = 0$*, and* $n(x) =$ **callee**.

The DPI shows the state of the system (i.e., the package together with its most general client) at the call and return points of public methods in the package. In those states, the stack of the client is empty and *this* always refers to the most general client. Therefore, we omit this information in abstract object graphs. The roles in abstract graphs can be seen as a projection of the internal state of the most general client on the objects in the heap. That is, the object instance of the most general client itself is not represented as a node in the graphs.

5.2.7 Object Mapping

Notation. For a package P, we denote by $\mathcal{M}_P$ the set of all its public methods: $\mathcal{M}_P = \bigcup_{C \in P} C.M_p$. For a public method $m(C_1, \cdots, C_n)$ of a class C, we define its *signature* as $sig(m) = \{(C, \textbf{callee}), (C_1, \textbf{arg}_\textbf{0}), \cdots, (C_n, \textbf{arg}_\textbf{n})\}$.

An *object mapping* of a method $m \in \mathcal{M}_P$ is a tuple (m, G, G', k) where $G, G' \in \mathcal{G}_P$, $k \subseteq G.O \times G'.O$ is a relation, and the following conditions are satisfied:

- G includes object instances for $sig(m)$:

$$\forall (C, s) \in sig(m),\ \exists o \in G.O,\ \ class(G.l(o)) = C \wedge G.n(o) = s;$$

- $dom(k) = G.O$;
- k preserves the class of an object: $\forall (o_1, o_2) \in k,\ \ class(G.l(o_1)) = class(G'.l(o_2))$;
- k is functional on object instances: $\forall (o_1, o_2), (o_1, o_3) \in k,\ G.nl(o_1) = 0 \Rightarrow o_2 = o_3$;
- k preserves the nesting level of object instances: $\forall (o_1, o_2) \in k,\ G.nl(o_1) = 0 \Leftrightarrow G'.nl(o_2) = 0$;
- k preserves the role names of object instances: $\forall (o_1, o_2) \in k,\ G.nl(o_1) = 0 \Rightarrow G.n(o_1) = G'.n(o_2)$.

For a set $M \subseteq \mathcal{G}_P$, by $Maps_P(M)$ we denote the set of all object mappings (m, G, G', k) of package P such that $G, G' \in M$.

An object mapping is a compact representation of the effect that a method call has on the objects of a package. The mapping specifies how objects are transformed by the method call. A pair $(o_1, o_2) \in k$ indicates that each concrete object represented by the abstract object o_1 might become part of the target abstract object o_2. The total number of concrete objects is always preserved. Because abstract object graphs can represent more than one concrete state, there can be more than one object mapping associated with a given method call and source graph, as well as multiple target objects for each source object in the source graph of one object mapping.

Example 12 *Let us consider the two unfolded object graphs inside the boxes in the left and right hand side of Fig. 5.6a. Denote these two graphs by G and G'. Figure 5.6a then represents the object mapping:* $(\texttt{set}, G, G', \{(V, V), (L_a, L_a), (L, L), (V_*, V_*)\})$.

Note that in addition to **callee** and $\textbf{arg}_\textbf{0}$ role names, the object mapping in Figure 5.6a also uses $\textbf{scope}_\textbf{0} \in G.R$, which labels an object instance that is not part of the signature of the method. The $\textbf{scope}_\textbf{i}$ role names are used to label all such object instances. One last type of role names that are used by object mappings is $\textbf{new}_\textbf{i}$ role names, which label the objects that are created by a method call. To improve the readability of some figures we omit abstract objects that are not modified. We show only the objects part of the connected component affected by the call.

5.2.8 Definition: DPI

A DPI of a package P is a tuple $(\mathcal{H}, \mathcal{G}, \Omega, \mathcal{E})$ where

- $\mathcal{H} \subseteq \mathcal{H}_P$ is a finite set of abstract object graphs,
- $\mathcal{G} \subseteq \mathcal{G}_P$ is a finite set of unfolded object graphs,
- $\Omega \subseteq \mathit{Maps}_P(\mathcal{G})$ the set of *object mappings*; and
- $\mathcal{E} \subseteq \mathcal{H}$ the set of *error* abstract object graphs.

The DPI $(\mathcal{H}, \mathcal{G}, \Omega, \mathcal{E})$ is *well-formed* if:

i. the unfolded graphs come from $\mathcal{H}$: $\forall G \in \mathcal{G},\ \mathit{src}(G) \in \mathcal{H}$

ii. it is *safe*: $\forall (m, G, G') \in \Omega,\ \mathit{src}(G) \in (\mathcal{H} - \mathcal{E})$; and

iii. it is *complete* in that a non-error covering abstract object graph has a mapping for all methods:

$$\forall H \in (\mathcal{H} - \mathcal{E}),\ \forall o \in H.O,\ \forall m \in \mathit{class}(G.l(o)).M_p,\ \exists (m, G, G') \in \Omega,\ \mathit{src}(G) = H.$$

Well-formed DPIs characterize the type of interface that we are interested in computing for OO packages. Following the analogy between a DPI and an FSM, the set of abstract object graphs correspond to the "states" of the state machine and the set of object mappings correspond to the "transitions". Section 5.2.9 describes how a well-formed DPI can be computed for a package soundly via an abstract semantics that simulates the concrete semantics of Section 5.2.3. Henceforth by a DPI, we mean a well-formed DPI.

A DPI can be understood in two ways. The first interpretation comes directly from the abstract OO program semantics of Section 5.2.9. The second interpretation views the DPI as a counter program. In this program each $H \in \mathcal{H}$ has a control location and for each node in $H.O$ there is a counter variable. The value of a counter keeps track of the number of concrete objects that are represented by the corresponding abstract object node. Object mappings can be translated into updates of the counters. Further details of that interpretation can be found in Section 5.2.10, 5.1, and [13].

5.2.9 Abstract Semantics for Computing DPI

In this section, we present the abstraction layers that we use to compute the DPI of a package. First, we present our *depth-bounded abstract* domain, which ensures that any chain of objects of a package has a bounded depth when represented in this domain. Then, we present our *ideal abstract* domain, which additionally ensures that any number of objects of a package are represented finitely. Finally, we describe how the DPI of a package can be computed by encoding the ideal abstract interpretation of a package as a numerical program.

Depth-Bounded Abstract Semantics

We now present an abstract semantics for OO programs. Given an OO program S, our abstract semantics of S is a transition system $S^{\#}_{\mathsf{h}} = (\mathit{Conf}^{\#}, U^{\#}_0, \rightarrow^{\#}_{\mathsf{h}})$ that is obtained by an abstract interpretation [31] of S. Typically, the system $S^{\#}_{\mathsf{h}}$ is still an infinite state system. However, the abstraction ensures that $S^{\#}_{\mathsf{h}}$ belongs to the class of *depth-bounded systems*. Depth-bounded systems are well-structured transition systems that can be effectively analyzed, and this will enable us to compute the dynamic package interface.

Heap Predicate Abstraction. We start with a heap predicate abstraction, following shape analysis [105, 97]. Let AP be a finite set of *unary abstraction predicates* from $\mathsf{Pred}.(\{x\} \cup \mathcal{C}.A)$ where x is a fresh variable different from this and null. For a configuration $u = (O, \cdot, st)$ and $o \in O$, we write $u \models p(o)$ iff $st[x \mapsto o](p) = 1$. Further, let AR be a subset of the reference fields in $\mathcal{C}.A$. We refer to AR as *binary abstraction predicates*. For an object $o \in \mathcal{O}$, we denote by $AR(o)$ the set $AR \cap \mathit{class}(o).A$.

The concrete domain D of our abstract interpretation is the powerset of configurations $D = \mathcal{P}(\mathit{Conf})$, ordered by subset inclusion. The abstract domain $D^{\#}_{\mathsf{h}}$ is the powerset of *abstract configurations* $D^{\#}_{\mathsf{h}} = \mathcal{P}(\mathit{Conf}^{\#})$, again ordered by subset inclusion. An abstract configuration $u^{\#} \in \mathit{Conf}^{\#}$ is like a concrete configuration except that the store is abstracted by a finite labelled graph, where nodes are object identifiers, edges correspond to the values of reference fields in AR, and node labels denote the evaluation of objects on the predicates in AP. That is, the abstract domain is parameterized by both AP and AR.

Formally, an abstract configuration $u^{\#} \in \mathit{Conf}^{\#}$ is a tuple $(O, \mathit{this}, q, \nu, \eta, st)$ where $O \subseteq \mathcal{O}$ is a finite set of object identifiers, $\mathit{this} \in O$ is the current object, $q \in F.Q$ is the current control location, ν is a finite sequence of triples (o, x, q) of objects, variables, and control location, $\eta : O \times AP \rightarrow \mathbb{B}$ is a *predicate valuation*, and st is an *abstract store* that maps objects in $o \in O$ and reference fields $a \in AR(o)$ to objects $st(p, a) \in O$. Note that we identify the elements of $\mathit{Conf}^{\#}$ up to isomorphic renaming of object identifiers.

The meaning of an abstract configuration is given by a concretization function $\gamma_{\mathsf{h}} : \mathit{Conf}^{\#} \rightarrow D$ defined as follows: for $u^{\#} \in \mathit{Conf}^{\#}$ we have $u \in \gamma_{\mathsf{h}}(u^{\#})$ iff (i) $u^{\#}.O = u.O$; (ii) $u^{\#}.\mathit{this} = u.\mathit{this}$; (iii) $u^{\#}.q = u.q$; (iv) $u^{\#}.\nu = u.\nu$; (v) for all $o \in u.O$ and $p \in AP$, $u^{\#}.\eta(o, p) = 1$ iff $u \models p(o)$; and (vi) for all objects $o \in O$, and $a \in AR(o)$, $u.st(o, a) = u^{\#}.st(o, a)$. We lift γ_{h} pointwise to a function $\gamma_{\mathsf{h}} : D^{\#}_{\mathsf{h}} \rightarrow D$ by defining $\gamma_{\mathsf{h}}(U^{\#}) = \bigcup \{ \gamma_{\mathsf{h}}(u^{\#}) \mid u^{\#} \in U^{\#} \}$. Clearly, γ_{h} is monotone. It is also easy to see that γ_{h} distributes over meets because for each configuration u there is, up to isomorphism, a unique abstract configuration $u^{\#}$ such that $u \in \gamma_{\mathsf{h}}(u^{\#})$. Hence, let $\alpha_{\mathsf{h}} : D \rightarrow D^{\#}_{\mathsf{h}}$ be the unique function such that $(\alpha_{\mathsf{h}}, \gamma_{\mathsf{h}})$ forms a Galois connection between D and $D^{\#}_{\mathsf{h}}$, i.e., $\alpha_{\mathsf{h}}(U) = \bigcap \{ U^{\#} \mid U \subseteq \gamma_{\mathsf{h}}(U^{\#}) \}$.

The abstract transition system $S^{\#}_{\mathsf{h}} = (\mathit{Conf}^{\#}, U^{\#}_0, \rightarrow^{\#}_{\mathsf{h}})$ is obtained by setting $U^{\#}_0 = \alpha_{\mathsf{h}}(U_0)$ and defining $\rightarrow^{\#}_{\mathsf{h}} \subseteq \mathit{Conf}^{\#} \times \mathit{Conf}^{\#}$ as follows. Let $u^{\#}, v^{\#} \in \mathit{Conf}^{\#}$. We have $u^{\#} \rightarrow^{\#}_{\mathsf{h}} v^{\#}$ iff $v^{\#} \in \alpha_{\mathsf{h}} \circ \mathsf{post}.S \circ \gamma_{\mathsf{h}}(u^{\#})$.

Theorem 12 *The system $S^{\#}_{\mathsf{h}}$ simulates the concrete system S, i.e., (i) $U_0 \subseteq \gamma_{\mathsf{h}}(U^{\#}_0)$ and (ii) for all $u, v \in \mathit{Conf}$ and $u^{\#} \in \mathit{Conf}^{\#}$, if $u \in \gamma_{\mathsf{h}}(u^{\#})$ and $u \rightarrow v$,*

then there exists $v^\# \in \mathit{Conf}^\#$ *such that* $u^\# \rightarrow_{\mathsf{h}}^\# v^\#$ *and* $v \in \gamma_{\mathsf{h}}(v^\#)$.

Proof. *(Sketch)* We can use the framework of abstract interpretation [32] to prove the theorem. By definition, $(\alpha_{\mathsf{h}}, \gamma_{\mathsf{h}})$ forms a Galois connection between D and $D_{\mathsf{h}}^\#$. Furthermore, $u^\# \rightarrow_{\mathsf{h}}^\# v^\#$ iff $v^\# \in \alpha_{\mathsf{h}} \circ \mathsf{post}.S \circ \gamma_{\mathsf{h}}(u^\#)$. □

Depth-Boundedness. Let $u^\# \in \mathit{Conf}^\#$ be an abstract configuration. A *simple path* of length n in $u^\#$ is a sequence of distinct objects $\pi = o_1, \ldots, o_n$ in $u^\#.O$ such that for all $1 \leq i < n$, there exists a_i with $u^\#.st(o_i, a_i) = o_{i+1}$ or $u^\#.st(o_{i+1}, a_i) = o_i$ (the path is not directed). We denote by $\mathit{lsp}(u^\#)$ the length of the longest simple path of $u^\#$. We say that a set of abstract configurations $U^\# \subseteq \mathit{Conf}^\#$ is *depth-bounded* if $U^\#$ is bounded in the length of its simple paths, i.e., there exists $k \in \mathbb{N}$ such that $\forall u^\# \in U^\#, \mathit{lsp}(u^\#) \leq k$ and the size of the stack $|u^\#.\nu| \leq k$.

We show that under certain restrictions on the binary abstraction predicates AR, the abstract transition system $S_{\mathsf{h}}^\#$ is a well-structured transition system. For this purpose, we define the *embedding order* on abstract configurations. An *embedding* for two configurations $u^\#, v^\# : \mathit{Conf}^\#$ is a function $h : u^\#.O \to v^\#.O$ such that the following conditions hold: (i) h preserves the class of objects: for all $o \in u^\#.O$, $\mathit{class}(o) = \mathit{class}(h(o))$; (ii) h preserves the current object, $h(u^\#.\mathit{this}) = v^\#.\mathit{this}$; (iii) h preserves the stack, $\bar{h}(u^\#.\nu) = v^\#.\nu$ where $\bar{h}$ is the unique extension of h to stacks; (iv) h preserves the predicate valuation: for all $o \in u^\#.O$ and $p \in AP$, $u^\#.\eta(o,p)$ iff $v^\#.\eta(h(o),p)$; and (v) h preserves the abstract store, i.e., for all $o \in u^\#.O$ and $a \in AR(o)$, we have $h(u^\#.st^\#(o,a)) = v^\#.st^\#(h(o),a)$. The embedding order $\preceq$: $\mathit{Conf}^\# \times \mathit{Conf}^\#$ is then as follows: for all $u^\#, v^\# : \mathit{Conf}^\#$, $u^\# \preceq v^\#$ iff $u^\#$ and $v^\#$ share the same current control location $(u^\#.q = v^\#.q)$ and there exists an injective embedding of $u^\#$ into $v^\#$.

Lemma 17 *(1) The embedding order is monotonic with respect to abstract transitions in* $S_{\mathsf{h}}^\# = (\mathit{Conf}^\#, U_0^\#, \rightarrow_{\mathsf{h}}^\#)$. *(2) Let* $U^\#$ *be a depth-bounded set of abstract configurations. Then* $(U^\#, \preceq)$ *is a bqo.*

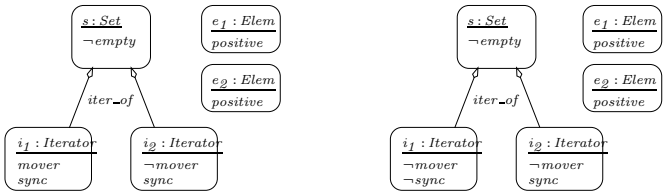
Proof. The first part follows form the definitions. For the second part, we can reduce it to the result from Chapter 4. We just need to encode the stack into the graph. The stack itself can be easily encoded as a chain with special bottom and top node. The assumption that the stack is bounded guarantees that can still apply Lemma 15. □

If the set of reachable configurations of the abstract transition system $S_{\mathsf{h}}^\#$ is depth-bounded, then $S_{\mathsf{h}}^\#$ induces a well-structured transition system.

Theorem 13 *If* $\mathit{Reach}(S_{\mathsf{h}}^\#)$ *is depth-bounded, then* $(\mathit{Reach}(S^\#), U_0^\#, \rightarrow_{\mathsf{h}}^\#, \preceq)$ *is a WSTS.*

Proof. The theorem follows from Lemma 17 and [80, Theorem 2]. □

In practice, we can ensure depth-boundedness of $\mathit{Reach}(S_{\mathsf{h}}^\#)$ syntactically by choosing the set of binary abstraction predicates AR such that it does not



(a) Before removing an element via i_2 (b) After removing the element

Figure 5.8: Two depth-bounded abstract configurations

contain reference fields that span recursive data structures. Such reference fields are only allowed to be used in the defining formulas of the unary abstraction predicates. Recursive data structures can be dealt with only if they are private to the package, i.e. not exposed to the user. In that case the predicate abstraction can use a more complex domain that understand such shapes, e.g. [105]. In the next section, we assume that the set $\mathit{Reach}(S^{\#}_{\mathsf{h}})$ is depth-bounded and we identify $S^{\#}_{\mathsf{h}}$ with its induced WSTS.

Example 13 *Figure 5.8 depicts the two corresponding, depth-bounded abstract configurations of the concrete configurations in Figure 5.7. The objects are labelled with their corresponding unary predicates. A labelled arrow between two objects specifies that the corresponding binary predicate between two object holds. The set of unary abstraction predicates consists of:*

$$\begin{array}{lcl} \mathit{empty}(x) & \equiv & x.\texttt{size} = 0 \\ \mathit{synch}(x) & \equiv & x.\texttt{iver} = x.\texttt{iter\_of.sver} \\ \mathit{mover}(x) & \equiv & x.\texttt{pos} < x.\texttt{iter\_of.size} \\ \mathit{positive}(x) & \equiv & x.\texttt{e} > 0 \end{array}$$

The set of binary abstraction predicates is $AR = \{\texttt{iter\_of}\}$. *If we had also included* `head` *and* `next` *in* AR, *the resulting abstraction would not have been depth bounded.*

Ideal Abstraction

In our model, the errors are local to objects. Thus, we are looking at the control-state reachability question. This means that the set of abstract error configurations is upward-closed with respect to the embedding order $\preceq$, i.e., we have $U^{\#}_{\mathit{err}} = {\uparrow}\, U^{\#}_{\mathit{err}}$. From the monotonicity of $\preceq$ we therefore conclude that $\mathit{Reach}(S^{\#}_{\mathsf{h}}) \cap U^{\#}_{\mathit{err}} = \emptyset$ iff $\mathit{Cover}(S^{\#}_{\mathsf{h}}) \cap U^{\#}_{\mathit{err}} = \emptyset$. This means that if we analyze the abstract transition system $S^{\#}_{\mathsf{h}}$ modulo downward closure of abstract configurations, this does not incur an additional loss of precision. We exploit this observation as well as the fact that $S^{\#}_{\mathsf{h}}$ is well-structured to construct a finite abstract transition system whose configurations are given by downward-closed sets of abstract configurations. We then show that this abstract transition system can be effectively computed.

Every downward-closed subset of a wqo is a *finite* union of ideals. In Chapter 3, we formalized an abstract interpretation coined *ideal abstraction*, which exploits this observation to obtain a generic terminating analysis for computing

an over-approximation of the covering set of a WSTS. We next show that ideal abstraction applies to the depth-bounded abstract semantics by providing an appropriate finite representation of ideals and how to use it to compute the DPI. The abstract domain $D^{\#}_{\mathsf{idl}}$ of the ideal abstraction is given by downward-closed sets of abstract configurations, which we represent as finite sets of ideals. The concrete domain is $D^{\#}_{\mathsf{h}}$. The ordering on the abstract domain is subset inclusion. The abstraction function is downward closure.

Formally, we denote by $Idl(Conf^{\#})$ the set of all depth-bounded ideals of abstract configurations with respect to the embedding order. Define the quasi-ordering $\sqsubseteq$ on $\mathcal{P}_{\text{fin}}(Idl(Conf^{\#}))$ as the point-wise extension of $\subseteq$ from the ideal completion $Idl(Conf^{\#})$ of $Conf^{\#}(\preceq)$ to $\mathcal{P}_{\text{fin}}(Idl(Conf^{\#}))$:

$$\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \iff \forall I_1 \in \mathcal{I}_1.\, \exists I_2 \in \mathcal{I}_2.\, I_1 \subseteq I_2$$

The abstract domain $D^{\#}_{\mathsf{idl}}$ is the quotient of $\mathcal{P}_{\text{fin}}(Idl(Conf^{\#}))$ with respect to the equivalence relation $\sqsubseteq \cap \sqsubseteq^{-1}$. For notational convenience we use the same symbol $\sqsubseteq$ for the quasi-ordering on $\mathcal{P}_{\text{fin}}(Idl(Conf^{\#}))$ and the partial ordering that it induces on $D^{\#}_{\mathsf{idl}}$. We further identify the elements of $D^{\#}_{\mathsf{idl}}$ with the finite sets of maximal ideals, i.e., for all $L \in D^{\#}_{\mathsf{idl}}$ and $I_1, I_2 \in L$, if $I_1 \subseteq I_2$ then $I_1 = I_2$. The abstract domain $D^{\#}_{\mathsf{idl}}$ is defined as $\mathcal{P}_{\text{fin}}(Idl(Conf^{\#}))$. The concretization function $\gamma_{\mathsf{idl}} : D^{\#}_{\mathsf{idl}} \to D^{\#}_{\mathsf{h}}$ is $\gamma_{\mathsf{idl}}(\mathcal{I}) = \bigcup \mathcal{I}$. Further, define the abstraction function $\alpha_{\mathsf{idl}} : D^{\#}_{\mathsf{h}} \to D^{\#}_{\mathsf{idl}}$ as $\alpha_{\mathsf{idl}}(U^{\#}) = \left\{ I \in Idl(Conf^{\#}) \mid I \subseteq {\downarrow}U^{\#} \right\}$. From the ideal abstraction framework, it follows that $(\alpha_{\mathsf{idl}}, \gamma_{\mathsf{idl}})$ forms a Galois connection between $D^{\#}_{\mathsf{h}}$ and $D^{\#}_{\mathsf{idl}}$. The overall abstraction is then given by the Galois connection (α, γ) between D and $D^{\#}_{\mathsf{idl}}$, which is defined by $\alpha = \alpha_{\mathsf{idl}} \circ \alpha_{\mathsf{h}}$ and $\gamma = \gamma_{\mathsf{h}} \circ \gamma_{\mathsf{idl}}$. We define the *abstract post operator* $\mathsf{post}^{\#}$ of S as the most precise abstraction of $\mathsf{post}.S$ with respect to this Galois connection, i.e., $\mathsf{post}^{\#}.S = \alpha \circ \mathsf{post}.S \circ \gamma$.

In the following, we assume the existence of a *sequence widening operator* $\nabla_{\mathsf{idl}} : Idl(Conf^{\#})^{+} \rightharpoonup Idl(Conf^{\#})$. The operator ∇_{idl} is an adaptation of the operator ∇_{DBP} from Section 3.7.3.

The ideal abstraction computes the covering set from which we can extract a finite labeled transition system $S^{\#}_{\mathsf{idl}}$ whose configurations are ideals of abstract configurations. The transitions corresponds to applying the rules to the ideals of the covering set followed by transitions labeled with ϵ, which we refer to as *covering transitions*. We call $S^{\#}_{\mathsf{idl}}$ the *abstract covering system* of $S^{\#}_{\mathsf{h}}$. This is because the set of reachable configurations of $S^{\#}_{\mathsf{idl}}$ over-approximates the covering set of $S^{\#}_{\mathsf{h}}$, i.e., $Cover(S^{\#}_{\mathsf{h}}) \subseteq \gamma_{\mathsf{idl}}(Reach(S^{\#}_{\mathsf{idl}}))$.

Formally, we define $S^{\#}_{\mathsf{idl}} = (\mathcal{I}_{\mathsf{idl}}, \mathcal{I}_0, \rightarrow^{\#}_{\mathsf{idl}})$ as follows. Given a downward-closed inductive invariant $\mathcal{C}$ of $S^{\#}_{\mathsf{idl}}$ the set of configurations $\mathcal{I}_{\mathsf{idl}} \subseteq Idl(Conf^{\#})$ is composed of the maximal incomparable ideals in $\mathcal{C}$. The initial configurations $\mathcal{I}_0$ are given by $\{s \in \mathcal{I}_{\mathsf{idl}} \mid \alpha_{\mathsf{idl}}(U^{\#}_0) \subseteq s\}$. The transition relation $\rightarrow^{\#}_{\mathsf{idl}} \subseteq \mathcal{I}_{\mathsf{idl}} \times \mathcal{I}_{\mathsf{idl}}$ is defined as the smallest set satisfying the following condition: for every $I \in \mathcal{I}_{\mathsf{idl}}$ and $I' \in \mathsf{post}^{\#}.S \circ \gamma_{\mathsf{idl}}(I)$, there is an $I'' \in \mathcal{I}_{\mathsf{idl}}$ such that $I' \subseteq I''$ and $(I, I'') \in \rightarrow^{\#}_{\mathsf{idl}}$.

Theorem 14 *The abstract covering system $S^{\#}_{\mathsf{idl}}$ is computable and finite.*

Proof. *(Sketch)* Following the result from Chapter 3, we can effectively compute an inductive overapproximation $\mathcal{C}$ of the covering set of $S^{\#}_{\mathsf{idl}}$. From

Lemma 4, we have a finite representation of $\mathcal{C}$. Finally, $\rightarrow^{\#}_{\mathsf{idl}}$ can be effectively computed as we will see in the remainder of the section. □

We now state our main soundness theorem.

Theorem 15 (Soundness) *The abstract covering system $S^{\#}_{\mathsf{idl}}$ simulates S, i.e., (i) $U_0 \subseteq \gamma(\mathcal{I}_0)$ and (ii) for all $I \in \mathcal{I}_{\mathsf{idl}}$ and $u, v \in Reach(S)$, if $u \in \gamma(I)$ and $u \rightarrow v$, then there exists $J \in \mathcal{I}_{\mathsf{idl}}$ such that $v \in \gamma(J)$ and $I \rightarrow^{\#}_{\mathsf{idl}} J$.*

Proof. *(Sketch)* The abstract covering system is just a lifting of the original transition system to a finite-state system by partitioning the states into a finite number of sets given by the incomparable ideals in covering set or an overapproximation of it. The lifting relies on the monotonicity property of the underlying WSTS to ensures simulation. The transition relation $\rightarrow^{\#}_{\mathsf{idl}}$ maps states from ideal to ideal while ensuring that the target ideal contains at least one larger state. □

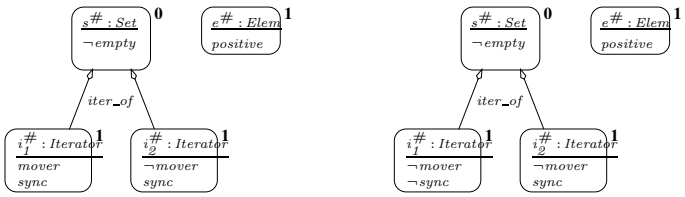
In the rest of this section we explain how we represent ideals of abstract configurations and how the operations for computing the abstract covering system are implemented.

Representing Ideals of Abstract Configurations. The ideals of depth-bounded abstract configurations are recognizable by regular hedge automata [112]. We can encode these automata into abstract configurations $I^{\#}$ that are equipped with a *nesting level function*. The nesting level function indicates how the substructures of the abstract store of $I^{\#}$ can be replicated to obtain all abstract configurations in the represented ideal.

Formally, a *quasi-ideal configuration* $I^{\#}$ is a tuple $(O, this, q, \nu, \eta, st, nl)$ where $nl : O \rightarrow \mathbb{N}$ is the nesting level function and $(O, this, q, \nu, \eta, st)$ is an abstract configuration, except that η is only a partial function $\eta : O \times AP \rightharpoonup \mathbb{B}$. We denote by $QIdlConf^{\#}$ the set of all quasi-ideal configurations. We call $I^{\#} = (O, this, q, \nu, \eta, st, nl)$ simply *ideal configuration*, if η is total and for all $o \in O$, $a \in AR(o)$, $nl(o) \geq nl(st(o, a))$. We denote by $[I^{\#}]$ the *inherent* abstract configuration $(O, this, q, \nu, \eta, st)$ of an ideal configuration $I^{\#}$. Further, we denote by $IdlConf^{\#}$ the set of all ideal configurations and by $IdlConf^{\#}_0$ the set of all ideal configurations in which all objects have nesting level 0. We call the latter *finitary* ideal configurations.

Meaning of Quasi-Ideal Configurations. An *inclusion mapping* between quasi-ideal configurations $I^{\#} = (O, this, q, \nu, st, nl)$ and $J^{\#} = (O', this', q', \nu', st', nl')$ is an embedding $h : O \rightarrow O'$ that satisfies the following additional conditions: (i) for all $o \in O$, $nl(o) \leq nl'(h(o))$; (ii) h is injective with respect to level 0 vertices in O': for all $o_1, o_2 \in O$, $o' \in O'$, $h(o_1) = h(o_2) = o'$ and $nl'(o') = 0$ implies $o_1 = o_2$; and (iii) for all distinct $o_1, o_2, o \in O$, if $h(o_1) = h(o_2)$, and o_1 and o_2 are both neighbors of o, then $nl'(h(o_1)) = nl'(h(o_2)) > nl'(h(o))$.

We write $I^{\#} \leq_h J^{\#}$ if $q = q'$, and h is an inclusion mapping between $I^{\#}$ and $J^{\#}$. We say that $I^{\#}$ is *included* in $J^{\#}$, written $I^{\#} \leq J^{\#}$, if $I^{\#} \leq_h J^{\#}$ for some h.



(a) Before removing an element via $i_2^{\#}$ (b) After removing the element

Figure 5.9: Two ideal abstract configurations

We define the meaning $[\![I^{\#}]\!]$ of a quasi-ideal configuration $I^{\#}$ as the set of all inherent abstract configurations of the finitary ideal configurations included in $I^{\#}$:

$$[\![I^{\#}]\!] = \left\{ [J^{\#}] \mid J^{\#} \in IdlConf_0^{\#} \wedge J^{\#} \leq I^{\#} \right\}$$

We extend this function to sets of quasi-ideal configurations, as expected.

Proposition 16 *Ideal configurations exactly represent the depth-bounded ideals of abstract configurations, i.e.,* $\left\{ [\![I^{\#}]\!] \mid I^{\#} \in IdlConf^{\#} \right\} = Idl(Conf^{\#})$.

Since the relation $\leq$ is transitive, we also get:

Proposition 17 *For all* $I^{\#}, J^{\#} \in QIdlConf^{\#}$, $I^{\#} \leq J^{\#}$ *iff* $[\![I^{\#}]\!] \subseteq [\![J^{\#}]\!]$.

It follows that inclusion of (quasi-)ideal configurations can be decided by checking for the existence of inclusion mappings, which is an NP-complete problem.

Quasi-ideal configurations are useful as an intermediate representation of the images of the abstract post operator. They can be thought of as a more compact representation of sets of ideal configurations. In fact, any quasi-ideal configuration can be reduced to an equivalent finite set of ideal configuration. We denote the function performing this reduction by $reduce : QIdlConf^{\#} \to \mathcal{P}_{\text{fin}}(IdlConf^{\#})$ and we extend it to sets of quasi-ideal configurations, as expected.

Example 14 *Figure 5.9 depicts the two corresponding, ideal abstract configurations of the two depth-bounded abstract configurations in Figure 5.8. The nesting level of each object is shown by the number next to it. When the abstract configurations in Figure 5.8 are considered as finitary ideal configurations, then they are included in their corresponding ideal configurations in Figure 5.9. The two inclusion mappings between the corresponding configurations in Figure 5.8 and Figure 5.9 are* $\left\{(i_1, i_1^{\#}), (i_2, i_2^{\#}), (s, s^{\#}), (e_1, e^{\#}), (e_2, e^{\#})\right\}$.

Note that since the nesting level of $s^{\#}$: *Set in both ideal configurations is zero, it is not possible to define inclusion mapping when there are more than one concrete set object. However, if the nesting levels of the set and iterator objects are incremented, then such an inclusion mapping can be defined.*

Computing the Abstract Post Operator. We next define an operator $\mathsf{Post}^{\#}.S$ that implements the abstract post operator $\mathsf{post}^{\#}.S$ on ideal configurations. In the following, we fix an ideal configuration $I^{\#} = (O, \mathit{this}, q, \nu, st, nl)$ and a transition $t = (q, op, q')$ in S. For transitions not enabled at $I^{\#}$, we set $\mathsf{Post}^{\#}.S.t(I^{\#}) = \emptyset$.

We reduce the computation of abstract transitions $[I^{\#}] \to u^{\#}$ to reasoning about logical formulas. For efficiency reasons, we implicitly use an additional Cartesian abstraction [12] in the abstract post computation that reduces the number of required theorem prover calls. For a set of variables X, we assume a *symbolic weakest precondition* operator $\mathsf{wp} : \mathsf{Op}.(\mathcal{C}.A) \times \mathsf{Pred}.(X \cup \mathcal{C}.A) \to \mathsf{Pred}.(X \cup \mathcal{C}.A)$ that is defined as usual. In addition, we need a symbolic encoding of abstract configurations into logical formulas. For this purpose, define a function $\Gamma : O \to \mathsf{Pred}.(O \cup \mathcal{C}.A)$ as follows: given $o \in O$, let $O(o)$ be the subset of objects in O that are transitively reachable from o in the abstract store st, then $\Gamma(o)$ is the formula

$$\Gamma(o) = \mathrm{distinct}(O(o) \cup O(\mathit{this})) \wedge \mathsf{this} = \mathit{this} \wedge \mathsf{null} = \mathit{null} \wedge$$
$$\bigwedge_{o' \in O(o) \cup O(\mathit{this})} \left(\bigwedge_{p \in AP} \eta(o', p) \cdot p(o') \wedge \bigwedge_{a \in AR(o')} o'.a = st(o'.a) \right)$$
$$\text{where } \eta(o', p) \cdot p(o') = \begin{cases} p(o') & \text{if } \eta(o', p) = 1 \\ \neg p(o') & \text{if } \eta(o', p) = 0. \end{cases}$$

Now, let $\mathcal{J}^{\#}$ be the set of all quasi-ideal configurations $J^{\#} = (O, \mathit{this}, q', \nu, \eta', st', nl)$ that satisfy the following conditions:

- $\Gamma(\mathit{this}) \wedge q$ is satisfiable, if $op = \mathrm{assume}(q)$;
- for all $o \in O$, $p \in AP$, if $\Gamma(o) \models \mathsf{wp}(op, p(o))$, then $\eta'(o, p) = 1$, else if $\Gamma(o) \models \mathsf{wp}(op, \neg p(o))$, then $\eta'(o, p) = 0$, else $\eta'(o, p)$ is undefined;
- for all $o, o' \in O$, $a \in AR(o)$, if $\Gamma(o) \wedge \Gamma(o') \models \mathsf{wp}(op, o.a = o')$, then $st'(o, a) = o'$, else if $\Gamma(o) \wedge \Gamma(o') \models \mathsf{wp}(op, o.a \neq o')$, then $st'(o, a) \neq o'$.

Then define $\mathsf{Post}^{\#}.S.t(I^{\#}) = \mathit{reduce}(\mathcal{J}^{\#})$.

5.2.10 Computing the Dynamic Package Interface

We now describe how to compute the dynamic package interface for a given package P. The computation proceeds in three steps. First, we compute the OO program $S = (P, I)$ that is obtained by extending P with its most general client I. Next, we compute the abstract covering system $S^{\#}_{\mathsf{idl}}$ of S as described in Sections 5.2.9 and 5.2.9. We assume that the user provides sets of unary and binary abstraction predicates AP, respectively, AR that define the heap abstraction. Alternatively, we can use heuristics to guess these predicates from the program text of the package. For example, we can add all branch conditions in the program description as predicates. Finally, we extract the package interface from the computed abstract covering system. We describe this last step in more detail.

We can interpret the abstract covering system as a numerical program. The control locations of this program are the ideal configurations in $S^{\#}_{\mathsf{idl}}$. With each

abstract object occurring in an ideal configuration we associate a counter. The value of each counter denotes the number of concrete objects represented by the associated abstract object. While computing $S^{\#}_{\mathsf{idl}}$, we do some extra book keeping and compute for each transition of $S^{\#}_{\mathsf{idl}}$ a corresponding numerical transition that updates the counters of the counter program. These updates capture how many concrete objects change their representation from one abstract object to another. A formal definition of such numerical programs can be found in [13].

The dynamic package interface $DPI(P)$ of P is a numerical program that is an abstraction of the numerical program associated with $S^{\#}_{\mathsf{idl}}$. The control locations of $DPI(P)$ are the ideal configurations in $S^{\#}_{\mathsf{idl}}$ that correspond to call sites, respectively, return sites to public methods of classes in P, in the most general client. A connecting path in $S^{\#}_{\mathsf{idl}}$ for a pair of such call and return sites (along with all covering transitions connecting ideal configurations on the path) corresponds to the abstract execution of a single method call. We refer to the restriction of the numerical program $S^{\#}_{\mathsf{idl}}$ to such a path and all its covering transitions as a *call program*. Each object mapping of $DPI(P)$ represents a summary of one such call program. Hence, an object mapping of $DPI(P)$ describes, both, how a method call affects the state of objects in a concrete heap configuration and how many objects are effected.

Note that a call program may contain loops because of loops in the method executed by the call program. The summarization of a call program therefore requires an additional abstract interpretation. The concrete domain of this abstract interpretation is given by transitions of counter programs, i.e., relations between valuations of counters. The concrete fixed point is the transitive closure of the transitions of the call program. The abstract domain provides an appropriate abstraction of numerical transitions. How precisely the package interface captures the possible sequences of method calls depends on the choice of this abstract domain and how convergence of the analysis of the call programs is enforced. We chose a simple abstract domain of *object mappings* that distinguishes between a constant number, respectively, arbitrary many objects transitioning from an abstract object on the call site of a method to another on the return site. However, other choices are feasible for this abstract domain that provide more or less information than object mappings.

5.2.11 Experiences

We have implemented our system by extending Picasso. Picasso uses an ideal abstraction to compute abstract coverability DAGs of depth-bounded graph rewriting systems. Our extension of Picasso computes a dynamic package interface from a graph rewriting system that encodes the semantics of the method calls in a package.[2]

For a graph-rewriting system that represents a package, our tool first computes its covering set. Using the elements of the covering set, it then performs unfolding over them with respect to all distinct method calls to derive the object mappings of the DPI of the package. The computation of the covering elements and the object mappings are carried out as described in the previous section.

In addition to the Viewer and Label example, described in Section 5.2.2, we

[2]Our tool and the full results of our experiments can be found at: http://pub.ist.ac.at/~zufferey/picasso/dpi/index.html

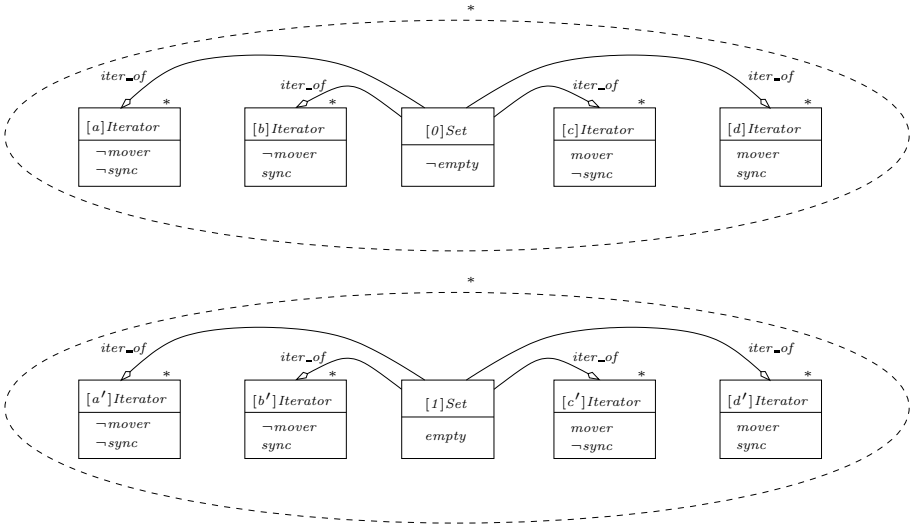
have experimented with other examples: a set and iterator package, which we used as our running example in the previous sections, and the JDBC statement and result package. In the remainder of this section, we present the DPIs for these packages.

Set and Iterator. We considered a simple implementation of the Set and Iterator classes in which the items in a set are stored in a linked list. The Iterator class has the usual next, has_next, and remove methods. The Set class provides a method iterator, which creates an Iterator object associated with the set, and an add method, which adds a data element to the set. The interface of the package is meant to avoid raising exceptions of types NoSuchElementException and ConcurrentModificationException. A NoSuchElementException is raised whenever the next method is called on an iterator of an empty list. A ConcurrentModificationException is raised whenever an iterator accesses the set after the set has been modified, either through a call to the add method of the set or through a call to the remove method of another iterator. An iterator that removes an element can still safely access the set afterwards. (Similar restrictions apply to other Collection classes that implement Iterable.)

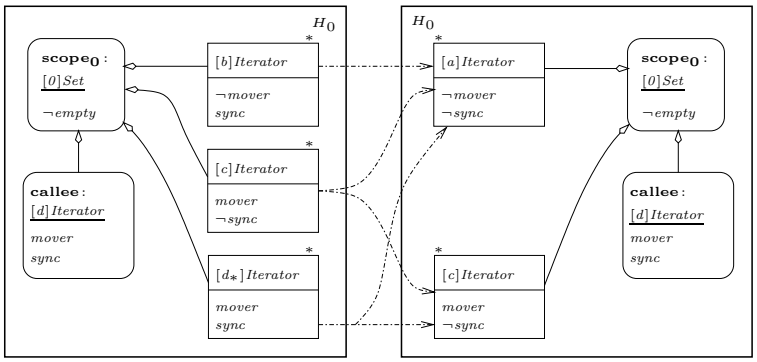
We used the following predicates. The unary abstraction predicate $empty(s)$ determines whether the size of a Set object s is zero or not. For Iterator objects, we specified two predicates that rely on the attributes of both the Set and the Iterator classes. The predicate $sync(i)$ holds for an Iterator object i that has the same version as its associated Set object. The predicate $mover(i)$ specifies that the position of an Iterator object i in the list of its associated Set object is less than the size of the set.

Our algorithm computes the maximal configurations H_0, shown in Figure 5.10a. There are also four error abstract heap configurations, which correspond to different cases in which one of the two exceptions is raised for an Iterator object. Figure 5.10b and 5.10c show the object mappings of two transitions. For the sake of clarity, we have omitted the name of the reference attribute *iter_of* in the mappings. While both transitions invoke the remove() method on an Iterator object whose *mover* and *sync* predicates are true, they have different effects because they capture different concrete heaps represented by the same abstract heap H_0. The first transition shows the case when the callee object remains a mover, i.e., its pos field does not refer to the last element of the list. The second transition shows the case when the callee object becomes a non-mover; i.e., before the call to remove, its pos field refers to the last element of the linked list. In both transitions, the other Iterator objects that reference the same Set object all become unsynced. Some of these objects remain movers while some of them become non-movers. In both cases, the callee remains synced. There are two other symmetric transitions that capture the cases in which the Set object becomes empty.

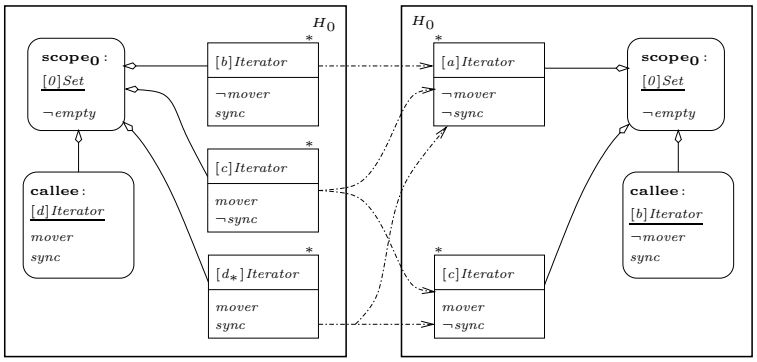
JDBC (Java Database Connectivity) is a Java technology that enables access to databases of different types. We looked at three classes of JDBC for simple query access to databases: Connection, Statement, and ResultSet. A Connection object provides a means to connect to a database. A Statement object can execute an SQL query statement through a Connection object. A ResultSet object stores the result of the execution of a Statement object. All objects can be closed



(a) Abstract heap configuration H_0 of the set-iterator package using predicates: $empty(s) \equiv s.\texttt{size} = 0$, $synch(i) \equiv i.\texttt{iver} = i.\texttt{iter\_of.sver}$, and $mover(i) \equiv i.\texttt{pos} < i.\texttt{iter\_of.size}$.



(b) Object mapping for d.remove()



(c) Another possible object mapping for d.remove()

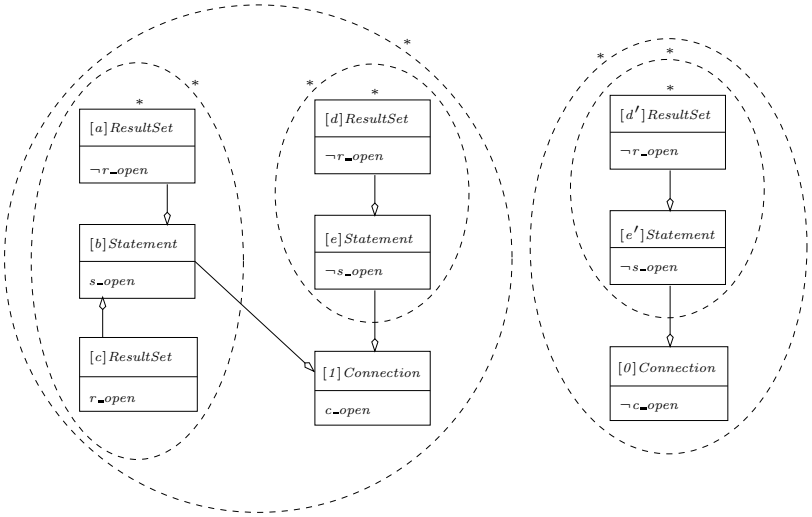
Figure 5.10: Set-iterator DPI: The abstract heap of the package together with two of its object mappings

explicitly. If a Statement object is closed, its corresponding ResultSet object is also implicitly closed. Similarly, if a Connection object is closed, its corresponding Statement objects are implicitly closed, and so are the open ResultSet objects of these Statement objects. Java documentation states: "By default, only one ResultSet object per Statement object can be open at the same time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All execution methods in the Statement interface implicitly close a statement's current ResultSet object if an open one exists."

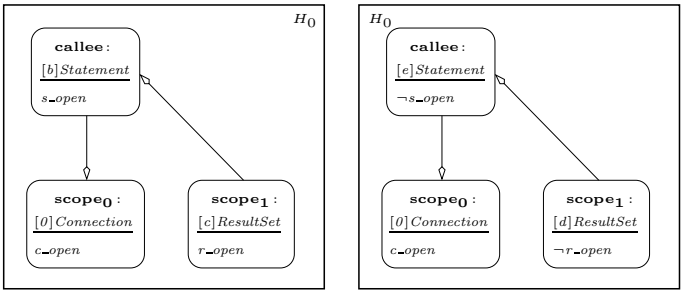
Figure 5.11a shows the maximal abstract heap H_0 computed by our tool. It represents all safe configurations in which the Connection object is either open or closed. Each type of object has a corresponding "open" predicate that specifies whether it is open or not. The node c is of particular interest, as it demonstrates the preciseness of our algorithm: It has the same nesting level as the node b, which means that an open Statement object can have at most one open ResultSet object associated with it. We omit showing abstract heaps capturing erroneous configurations. Lastly, Figure 5.11b shows the object mapping for the close method call on an open Statement object with an open ResultSet object. The mapping takes the Statement object and the open ResultSet object to their corresponding closed objects. All other objects remain the same.

Summary

We have formalized DPIs for OO packages with inter-object references, developed a novel ideal abstraction for heaps, and given a sound and terminating algorithm to compute DPIs on the (infinite) abstract domain. In contrast to previous techniques for multiple objects based on mixed static-dynamic analysis [91, 100], our algorithm is guaranteed to be sound. While our algorithm is purely static, an interesting future direction is to effectively combine it with dual, dynamic [54, 34, 100] and template-based [110] techniques.



(a) Abstract heap configurations H_0 of JDBC package



(b) Object mapping for b.close()

Figure 5.11: JDBC DPI: The abstract heap of JDBC together with one of its object mappings

Chapter 6

Conclusion

Our goal was to develop a framework for the analysis of message-passing concurrency. We choose to focus on depth-bounded systems due to their ability to faithfully model a wide variety of systems. First, we looked only a π-calculus, thus, at distributed systems. Later, after moving to a graph-based formalism, we extended our analysis to shared memory concurrency.

At the beginning our focus was only on safety properties such as the control-state-reachability problem. We developped an abstraction compute an inductive invariant, i.e. an overapproximation of the covering set, which provides the answer to any covering question. Later, we build additional analyses on top of our safety analysis, e.g. we could generate integer programs that simulate the input systems and used those to prove termination of the original systems.

Another goal was to have fully-automatic analyses. Because most of the problems we looked at are undecidable, our algorithms computes sound over-approximations of the exact solutions. Most of the methods presented in this thesis have been implemented in the Picasso tool. With Picasso, we can show that our abstractions are precise, thus, useful in practice.

First, by giving a finite representation to downward-closed sets for depth-bounded systems, we made the first step toward an analysis of depth-bounded systems using forward search. We gave two formalizations for the limit elements: the first one based on the π-calculus and the second one, more amenable to implementation, based on graphs. In π-calculus the limits are represented using the replication operator. In the graph formalism we mark subgraphs as repeatable. The subgraphs correspond to the scope of the replication operator in π-calculus. As the replication operator subgraphs can be nested (up to the depth-of the system). Therefore, we call them nested graphs. Nested graphs are implemented in the Picasso tool and form the basic data structure used by the analyses.

Then, we propose an abstract interpretation framework to compute a sound approximations of the covering set of WSTS. The goal is to capture the essence of acceleration-based algorithms which compute the covering set and generalize it to classes of systems for which acceleration does not work. Acceleration-based algorithms terminate only on flattable systems, i.e. systems that can be saturated with a finite number of simple loops. Unfortunately, due to

the shape of the state-space, depth-bounded systems are typically not flattable. Nested loops needs to be considered by the analysis. We sidestep this problem by replacing acceleration (precise) with widening (overapproximation) to ensure that our analysis always terminates. We discuss several concrete instances of our framework including Petri nets, lossy-channel systems, and depth-bounded systems. The analysis is implemented in Picasso and experiments show that the analysis is often precise, which makes it a useful tool for verification and program analysis.

The decision to focus on the covering set rather than the easier covering problem came from our expectation to extract useful information from the covering set. In the context of π-calculus and mobile processes the communication topology, among other information, can be extracted from the covering set. Therefore, even without a clear safety property to check the analysis can return useful information. Later, we build more complex analysis using the covering set as starting point.

[13] presents a method to check whether a depth-bounded system terminates when subject to some weak fairness constraints. The method builds an numerical abstraction of the system. Unlike traditional counter abstraction the number of counters is not fixed a priori, but is determined by the structure of the covering set. This give us a way of finding complex termination arguments with an high degree of automation. We implemented the method in Picasso and show that it is sufficiently precise to prove termination of lock-free algorithms and distributed systems.

We also started to generalize the notion of state-machine interfaces in the object-oriented world from single object to groups of interacting objects in what we call dynamic package interfaces. Similar to the structural counter abstraction, the method takes the covering set of a depth-bounded systems, applies the post operator and tracks changes in the configuration graphs. For interfaces, the tracking tells us how a method called on some object not only affect the callee but the other objects (pointing-to, pointed-by) around it. We have an early implementation of the method in Picasso and applied it to common OO usage patterns like containers and iterators.

At the time of writing this thesis, the work on dynamic package interfaces is still ongoing. On the short term we are also exploring the use of the structural counter abstraction to strengthen invariant represented by the covering set and prove more complex safety properties. One could, for instance, prove that the number of some elements, e.g. tasks or processes, is preserved over time. We also want to continue the work on making the analysis more scalable and provide a frontend which is closer to a more feature-rich programming language. For the moment, computation of the covering set is quite sensitive to the interleavings of processes, i.e. the length of the path between two comparable configurations. Therefore, an efficient frontend has to expose only the synchronization points of a process and deal separately with the "local" computations. There is some interesting challenges in integrating the techniques used in software-model checkers, e.g. CEGAR, with the infinite-state backend of Picasso to achieve a better scalability and proving more complex properties, e.g. dependent on data such as integers. The exploration done with the Scala compiler plugin

shows that our analysis is applicable to actor programs but the implementation requires a large amount of time and has many software engineering challenges. A direct encoding of the features of Scala into π-calculus has an high cost in term of performance. For instance, the dynamic dispatch introduces a level of indirection that makes function calls expensive and increases the size of the state-space. To achieve good performance the frontend needs to performs many intermediate analysis to efficiently remove such feature and to produce a system that is compact enough for the backend to analyse.

On the longer term there is still many open question related to the analysis of concurrent programs with unbounded process creation. We approached concurrency through the lens of well-structured systems. Within that framework monotonicity plays a crucial role. However, precise characterisation of monotonicity is still lacking. Existing characterisation of monotonicity in term of strength, i.e. how many transition of a larger state are needed to simulate a smaller state, or strictness, i.e. whether the simulation preserve strictness in the ordering, does not tell anything about the practicality of the analysis.

The problem of these characterisations is that they are checked on every states and every transitions. These global and static checks do not reflect what happen during the state-space exploration. A finer way of looking at monotonicity is required. By looking at the computation of the covering set we observed the following phenomenon: the number of ideals summarizing the set of explored state first increases, then decreases as the set get closer to the covering set. Furthermore, ideals that are closer to being *fully saturated* tend to simulate smaller states with less transitions, e.g., in our case, 0 instead of 1. Therefore, many transitions do not have any effect on the later stage of the exploration. At that point the analysis usually gets faster and converges quickly. Understanding this phenomenon and quantifying it is an worthy challenge likely to have a impact. It would give a better understanding of which type of systems are suitable for the kind of saturation procedure we use and also give us better heuristics to steer search along path that can be easily saturated. This would help us to better understand concurrent software and give more effective analyses.

Bibliography

[1] The Z3 Theorem Prover. research.microsoft.com/projects/Z3.

[2] P.A. Abdulla, G. Delzanno, and A. Rezine. Monotonic abstraction in parameterized verification. *Electronic Notes in Theoretical Computer Science*, 223:3–14, 2008.

[3] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General Decidability Theorems for Infinite-State Systems. In *LICS*, pages 313–321. IEEE, 1996.

[4] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *FMSD*, 25(1):39–65, 2004.

[5] Parosh Aziz Abdulla and Bengt Jonsson. Verifying Programs with Unreliable Channels. In *LICS*, pages 160–170, 1993.

[6] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT CSAIL, 1986.

[7] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL'05*, pages 98–109. ACM, 2005.

[8] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *LICS*, pages 207–218. IEEE Computer Society, 1996.

[9] Roberto M. Amadio and Charles Meyssonnier. On Decidability of the Control Reachability Problem in the Asynchronous pi-Calculus. *Nord. J. Comput.*, 9(1):70–101, 2002.

[10] Tim Azzopardi. Generic compute server in Scala using remote actors. http://tiny.cc/yjzva, 2008. Accessed Nov 2011.

[11] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening Operators for Powerset Domains. *Software Tools for Technology Transfer*, 8(4/5):449–466, 2006.

[12] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[13] Kshitij Bansal, Eric Koskinen, Thomas Wies, and Damien Zufferey. Structural counter abstraction. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2013.

[14] Kshitij Bansal, Eric Koskinen, Thomas Wies, and Damien Zufferey. Structural Counter Abstraction. Technical Report TR2012-947, New York University, 2013.

[15] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Ph. Schnoebelen. Flat Acceleration in Symbolic Model Checking. In *ATVA*, pages 474–488, 2005.

[16] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic Counter Abstraction for Concurrent Software. In *CAV*, volume 5643 of *LNCS*, pages 64–78. Springer, 2009.

[17] Jörg Bauer and Reinhard Wilhelm. Static Analysis of Dynamic Communication Systems by Partner Abstraction. In *SAS*, pages 249–264, 2007.

[18] Andi Bejleri, Pierre-Malo Deniélou, Raymond Hu, and Nobuko Yoshida. Parameterised Multiparty Session Types. In *International Conference on Foundations of Software Science and Computation Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145, March 2010.

[19] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV*, pages 386–400. Springer, 2006.

[20] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[21] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In Touili et al. [108], pages 227–242.

[22] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. Recursive Definitions in Channel Based Calculi. In *ICALP*, pages 133–144, 2003.

[23] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[24] Philip W. Carruth. Arithmetic of ordinals with applications to the theory of ordered Abelian groups. *Bull. Amer. Math. Soc.*, 48:262–271, 1942.

[25] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *CONCUR 2009: Proceedings of the 20th International Conference on Concurrency Theory*, pages 211–228, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–57, New York, NY, USA, 2002. ACM.

[27] Pierre Chambart, Alain Finkel, and Sylvain Schmitz. Forward Analysis and Model Checking for Trace Bounded WSTS. In Lars M. Kristensen and Laure Petrucci, editors, *Proceedings of the 32nd International Conference on Applications and Theory of Petri Nets (ICATPN'11)*, Lecture Notes in Computer Science, Kanazawa, Japan, 2011. Springer.

[28] William Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT CSAIL, 1981.

[29] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available from: http://tata.gforge.inria.fr/, 2008. release November, 18th 2008.

[30] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[31] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[33] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[34] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA*, pages 85–96. ACM, 2010.

[35] Mads Dam. Model Checking Mobile Processes. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 1993.

[36] Giorgio Delzanno. A Symbolic Procedure for Control Reachability in the Asynchronous Pi-calculus: Extended Abstract. *Electr. Notes Theor. Comput. Sci.*, 98:21–33, 2004.

[37] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the Automated Verification of Multithreaded Java Programs. In *TACAS*, volume 2280, pages 173–187. Springer, 2002.

[38] Reinhard Diestel. Relating Subsets of a Poset, and a Partition Theorem for WQOs. *Order*, 18(3):275–279, 2001.

[39] Reinhard Diestel and Oleg Pikhurko. On the cofinality of infinite partially ordered sets: Factoring a poset into lean essential subsets. *Order*, 20:53–66, 2003. 10.1023/A:1024449306316.

[40] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for Erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 137–140. ACM, 2012.

[41] Catherine Dufourd, Alain Finkel, and Ph. Schnoebelen. Reset Nets Between Decidability and Undecidability. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998.

[42] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corbibradini. *Handbook of Graph Grammars and Computing by Graph Transformations Vol.1*. World Scientific, River Edge, NJ, USA, 1997.

[43] Joost Engelfriet and Tjalling Gelsema. Multisets and Structural Congruence of the pi-Calculus with Replication. *Theor. Comput. Sci.*, 211(1-2):311–337, 1999.

[44] Alain Finkel. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In *ICALP*, pages 499–508, 1987.

[45] Alain Finkel. Reduction and covering of infinite reachability trees. *Inf. Comput.*, 89(2):144–179, 1990.

[46] Alain Finkel. The Minimal Coverability Graph for Petri Nets. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer, 1991.

[47] Alain Finkel and Jean Goubault-Larrecq. Forward Analysis for WSTS, Part I: Completions. In *STACS*, volume 09001 of *Dagstuhl Sem. Proc.*, pages 433–444, 2009.

[48] Alain Finkel and Jean Goubault-Larrecq. Forward Analysis for WSTS, Part II: Complete WSTS. In *ICALP (2)*, pages 188–199, 2009.

[49] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[50] Roland Fraïssé. *Theory of Relations*, volume 145 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2000.

[51] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. In *VMCAI*, pages 49–64, 2006.

[52] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.

[53] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the Efficient Computation of the Minimal Coverability Set for Petri Nets. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762, pages 98–113, 2007.

[54] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE*, pages 430–440. IEEE, 2009.

[55] Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia. On the Expressiveness of Infinite Behavior and Name Scoping in Process Calculi. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2004.

[56] D. Giannakopoulou and C.S. Pasareanu. Interface Generation and Compositional Verification in JavaPathfinder. In *FASE*, volume 5503 of *LNCS*, pages 94–108. Springer, 2009.

[57] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*. ACM, 2009.

[58] Jean Goubault-Larrecq. On Noetherian Spaces. In *LICS*, pages 453–462. IEEE Computer Society, 2007.

[59] Jean Goubault-Larrecq. Noetherian Spaces in Verification. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 2010.

[60] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.

[61] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[62] Philipp Haller and Frank Sommers. *Actors in Scala*. Artima, January 2012.

[63] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 31–40. ACM, 2005.

[64] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[65] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[66] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 247–251. Springer, 2012.

[67] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, 2008.

[68] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121 – 163, 2004.

[69] Radu Iosif, Filip Konecny, and Marius Bozga. The numerical transition systems library. `http://nts.imag.fr/`. retrived on May 1, 2013.

[70] D. Janssens, M. Lens, and G. Rozenberg. Computation graphs for actor grammars. *J. Comput. Syst. Sci.*, 46(1):60–90, 1993.

[71] Salil Joshi and Barbara König. Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems. In *CAV*, volume 5123 of *LNCS*, pages 214–226. Springer, 2008.

[72] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2012.

[73] Richard M. Karp and Raymond E. Miller. Parallel Program Schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.

[74] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *CAV*, 2013.

[75] Richard Laver. On Fraïssé's Order Type Conjecture. *Ann. of Math.*, 93(1):89–111, 1971.

[76] Jérôme Leroux and Grégoire Sutre. Flat Counter Automata Almost Everywhere! In *ATVA*, pages 489–503, 2005.

[77] Z. Li and Y.Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, pages 306–315. ACM, 2005.

[78] Lift. Lift web framework. http://liftweb.net/.

[79] Richard Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci.*, 297(1-3):337–354, 2003.

[80] R. Meyer. On Boundedness in Depth in the pi-Calculus. In *TCS*, volume 273 of *IFIP 273*, pages 477–489. Springer, Springer, 2008.

[81] Roland Meyer. A theory of structural stationarity in the pi-calculus. *Acta Inf.*, 46(2):87–137, 2009.

[82] Roland Meyer and Roberto Gorrieri. On the Relationship between pi-Calculus and Finite Place/Transition Petri Nets. In *CONCUR*, pages 463–480, 2009.

[83] Roland Meyer and Tim Strazny. Petruchio: From Dynamic Networks to Nets. In Touili et al. [108], pages 175–179.

[84] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*, 1996.

[85] E. C. Milner. Basic WQO- and BQO-Theory. *Graphs and order*, 1985.

[86] Robin Milner. Flowgraphs and Flow Algebras. *J. ACM*, 26(4):794–818, 1979.

[87] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[88] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[89] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[90] Ugo Montanari and Marco Pistore. Checking Bisimilarity for Finitary pi-Calculus. In *CONCUR*, pages 42–56, 1995.

[91] M.G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA*, pages 77–96. ACM, 2005.

[92] Crispin St John Alvah Nash-Williams. On better-quasi-ordering transfinite sequences. *Proc. Camb. Phil. Soc.*, 64:273–290, 1968.

[93] Jaroslav Nešetřil and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *Eur. J. Comb.*, 27(6):1022–1041, 2006.

[94] Karol Ostrovský. *On Modelling and Analysing Concurrent Systems*. PhD thesis, Chalmers University of Technology and Gotebörg University, 2005.

[95] Carl Adam Petri and Wolfgang Reisig. Scholarpedia, 3(4):6477. http://www.scholarpedia.org/article/Petri_net, 2008.

[96] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *CAV*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.

[97] A. Podelski and T. Wies. Boolean Heaps. In *SAS*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.

[98] Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.

[99] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. Heap Assumptions on Demand. In *CAV*, volume 5123 of *LNCS*, pages 314–327. Springer, 2008.

[100] M. Pradel, C. Jaspan, J. Aldrich, and T.R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE'12*, pages 925–935. IEEE, 2012.

[101] Sriram Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In Patrick Cousot, editor, *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 375–394. Springer Berlin / Heidelberg, 2001.

[102] G. Ramalingam, Alex Varshavsky, John Field, Deepak Goyal, and Shmuel Sagiv. Deriving specialized program analyses for certifying component-client conformance. In Jens Knoop and Laurie J. Hendren, editors, *PLDI*, pages 83–94. ACM, 2002.

[103] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[104] Philipp Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[105] M. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3):217–298, 2002.

[106] Davide Sangiorgi. pi-Calculus, Internal Mobility, and Agent-Passing Calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.

[107] Philippe Schnoebelen. Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In *MFCS*, pages 616–628, 2010.

[108] Tayssir Touili, Byron Cook, and Paul Jackson, editors. *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*. Springer, 2010.

[109] R.K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[110] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Autom. Softw. Eng.*, 18(3-4):263–292, 2011.

[111] J. Whaley, M.C. Martin, and M.S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, 2002.

[112] Thomas Wies, Damien Zufferey, and Thomas A. Henzinger. Forward Analysis of Depth-Bounded Processes. In *FoSSaCS 2010*, volume 4349 of *LNCS*, pages 94–108. Springer, 2010.

[113] Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A Logical Encoding of the pi-Calculus: Model Checking Mobile Processes Using Tabled Resolution. In *VMCAI*, pages 116–131, 2003.

[114] D. Zufferey, T. Wies, and T.A. Henzinger. Ideal Abstractions for Well-Structured Transition Systems. In *VMCAI*, volume 7148 of *LNCS*, pages 445–460. Springer, 2012.

[115] Damien Zufferey. Verification of concurrent asynchronous message passing programs. Master's thesis, EPFL, 2009.

[116] Damien Zufferey and Thomas Wies. Picasso Analyzer. http://ist.ac.at/~zufferey/picasso/.

[117] Damien Zufferey, Thomas Wies, and Thomas A. Henzinger. Ideal Abstractions for Well-Structured Transition Systems. In *VMCAI*, pages 445–460, 2012.