Ranking Abstraction as Companion to Predicate Abstraction*

Ittai Balaban[1], Amir Pnueli[12], and Lenore D. Zuck[3]

[1] New York University, New York, {balaban,amir}@cs.nyu.edu
[2] Weizmann Institute of Science,
[3] University of Illinois at Chicago, lenore@cs.uic.edu

Abstract. Predicate abstraction has become one of the most successful methodologies for proving safety properties of programs. Recently, several abstraction methodologies have been proposed for proving liveness properties. This paper studies "ranking abstraction" where a program is augmented by a non-constraining progress monitor, and further abstracted by predicate-abstraction, to allow for automatic verification of progress properties. Unlike most liveness methodologies, the augmentation does not require a complete ranking function that is expected to decrease with each helpful step. Rather, the inputs are component rankings from which a complete ranking function may be formed.
The premise of the paper is an analogy between the methods of ranking abstraction and predicate abstraction, one ingredient of which is refinement: When predicate abstraction fails, one can refine it. When ranking abstraction fails, one must determine whether the predicate abstraction, or the ranking abstraction, need be refined. The paper presents strategies for determining which case is at hand.
The other part of the analogy is that of automatically deriving deductive proof constructs: Predicate abstraction is often used to derive program invariants for proving safety properties as a boolean combination of the given predicates. Deductive proof of progress properties requires well-founded ranking functions instead of invariants. We show how to obtain concrete global ranking functions from abstract programs.
We demonstrate the various methods on examples with nested loops, including a bubble sort algorithm on linked lists.

1 Introduction

Predicate abstraction has become one of the most successful methodologies for proving safety properties of programs. However, with no extension it cannot be used to verify general liveness properties. In this paper, we present a framework, based on predicate abstraction and *ranking abstraction*, for verification of both safety and progress properties. Ranking abstraction, introduced in [8], is based on an augmentation of the concrete program. The augmentation is parameterized by a set of well founded ranking

* This research was supported in part by NSF grant CCR-0205571, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

functions. Based on these, new *compassion* (strong fairness) requirements as well as transitions are generated, all of which are synchronously composed with the program in a non-constraining manner. Unlike most methodologies, the ranking functions are not expected to decrease with each transition of the program.

The basic premise presented in this paper is that there is a duality between the activities that lead to verification of safety properties via predicate abstraction, and those that lead to verification of progress properties via ranking abstraction. This duality is expressed through the following components:

- *The initial abstraction.* Heuristics are applied to choose either an initial set of predicates, or a set of core well founded ranking functions.
- *Refinement.* A too-coarse initial abstraction leads to spurious abstract counterexamples. Depending on the character of the counterexample, either a predicate, or a ranking, refinement is performed.
- *Generation of deductive proof constructs.* Predicate abstraction is often used as an automatic method to generate an inductive invariant as a boolean combination of the given predicates. Dually, ranking abstraction can be used to generate a global concrete ranking function that decreases with every step of the program, as a lexicographical combination of the core ranking functions.

We demonstrate the use of ranking refinement in order to prove termination of a canonical program with nested loops and unbounded random assignments, as well as a bubble sort algorithm on unbounded linked lists. Both examples entail the use of additional heuristics in order to synthesize core ranking functions.

The framework, as well as all experiments, have been implemented using the TLV programmable model-checker [1]. The contribution of the paper is as follows: At the informal, conceptual level, it strives to convince the reader that the duality between invariance and progress, present in deductive frameworks, extends to how one approaches automatic verification of each kind of property. More concretely, it suggests a formal framework, based on two specific abstraction methods for proving both safety and progress properties. This includes heuristics for choosing separate refinement methodologies based on the form of counterexamples, and a method for automatically deriving a global well founded program ranking function.

The paper is organized as follows: Section 2 describes the computational model of *fair discrete systems* as well as predicate and ranking abstractions. Furthermore, it motivates the use of ranking abstraction by demonstrating its value, compared to a typical deductive method. Section 3 formalizes the different notions of abstraction refinement. Section 4 presents a method for extracting the auxiliary constructs necessary for a deductive proof of a response property from a successful application of the ranking-abstraction method. These include a set of ranking functions and helpful assertions. This section deals with the restricted case of sequential programs, which do not assume any fairness requirements. These restrictions are removed in Section 5 which performs a similar extraction of auxiliary constructs for the case of a system representing a concurrent program, or any other system with weak fairness requirements. Finally, Section 6 summarizes and concludes.

Related Work

The body of work most comparable to ours is [6], where Cook, Podelski, and Rybalchenko present a framework for verifying termination, that formalizes duel refinements – of transition predicate abstraction and of transition invariants. Transition invariants can be described as composite transition relations of a program that has been augmented with a ranking function, and assume the role of ranking functions in our framework. Comparable to our work, the algorithm in [6], when presented with an abstract counterexample, analyzes the cause of its "spuriousness", and refines either the predicate abstraction or the transition invariant. While our framework is inherently applicable to systems with (weak and strong) fairness constraints, the framework in [6] lacks any notion of fairness. Therefore it must be extended in order to support concurrent programs. Furthermore in this work we show how to extract deductive proofs of progress for sequential programs and a wide class of concurrent programs.

Dams, Gerth, and Grumberg [7] point out the duality between verification of safety and progress of programs. Like us, they aim to lift this duality to provide tools for proving progress properties, whose functionality is analogous to similar tools used for safety. Specifically, they propose a heuristic for discovering ranking functions from a program's text. In contrast, we concentrate on an analogy with predicate abstraction, a particular method for safety. Our approach is broader, however, in that we suggest a general framework for safety and progress properties where each of the activities in a verification process has an instantiation with respect to each of the dualities.

In [12] Podelski and Rybalchenko present a method for synthesis of linear ranking functions. The method is complete for unnested loops, and is embedded successfully in a broader framework for proving liveness properties [11], as well as in [6]. This method is one of several candidates that can be embedded in our framework, when ranking refinement is called for.

The topic of refinement of state abstraction, specifically predicate abstraction, has been widely studied. A number of existing works in this area are [5, 3], and [4].

2 The Formal Framework

In this section we present our computational model, as well as the methods of predicate abstraction and ranking abstraction.

2.1 Fair Discrete Systems

As our computational model, we take a *fair discrete system* (FDS) $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- V — A set of *system variables*. A *state* of $\mathcal{D}$ provides a type-consistent interpretation of the variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.
- Θ — The *initial condition*: An assertion (state formula) characterizing the initial states.

- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an $\mathcal{D}$-successor state $s' \in \Sigma$. We assume that every state has a ρ-successor.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements (assertions); A computation must include infinitely many states satisfying each of the justice requirements.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q \rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

For an assertion ψ, we say that $s \in \Sigma$ is a ψ-state if $s \models \psi$.

A *run* of an FDS $\mathcal{D}$ is a possibly infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is an $\mathcal{D}$-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1} \rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.

A *computation* of $\mathcal{D}$ is an infinite run that satisfies

- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Compassion* – for every $\langle p, q \rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

Since we focus here on termination properties of sequential programs, we define a state s to be *terminal* if it has itself as the only successor. Justice assumptions ensure that, if possible, a computation takes non-idle steps, i.e. steps leading from a state s to a successor different from s. Extending the framework to general, possibly concurrent, systems requires distinction between *deadlock* states and *terminal* states. The extension is straightforward, and for simplicity of exposition we ignore this distinction here. An FDS is defined to be *terminating* if every computation contains (eventually reaches) a terminal state. To simplify the treatment of sequential programs, we assume that a state s that has a successor $s' \neq s$ cannot also have itself as a successor. This guarantees progress without the need for explicit fairness requirements. In particular it implies that idling steps are only possible from terminal states.

Once we move to concurrent programs, where fairness requirements play a significant role, we will remove this assumption.

Ranking A well-founded domain is a pair $(\mathcal{W}, \succ)$ such that $\mathcal{W}$ is a set and $\succ$ is a partial order over $\mathcal{W}$ that admits no infinite $\succ$-decreasing chains. A *ranking function* is a function mapping program states to some well-founded domain.

An assertion that, like a transition relation, refers to both unprimed and primed copies of the system variables is called a *bi-assertion*. A bi-assertion $\beta(V, V')$ is called *well founded over assertion* p if there does not exist an infinite sequence of states $s_0, s_1, \ldots$, such that $s_0 \models p$ and $\langle s_i, s_{i+1} \rangle \models \beta$, for every $i \geq 0$. If $p = 1$ (*true*), then we say simply that β is *well founded*.

In order to prove that β is well founded over p, it is sufficient to find an auxiliary assertion φ and a well-founded ranking δ, such that

$$p \to \varphi \qquad \text{and} \qquad \varphi(V) \land \beta(V, V') \to \varphi(V') \land \delta(V) \succ \delta(V')$$

In this case, we say that the well-founded ranking δ *proves the well-foundedness of* β *over* p.

Termination of an FDS corresponding to a sequential program is often proved by finding a well-founded ranking δ such that δ decreases on every non-idle step. We refer to such a ranking function as *adequate for* (proving) *termination*. With appropriate assumptions on the FDS (e.g., countable non-determinism), every terminating (sequential) FDS has such well founded ranking.

The same method can also be applied to prove more general progress properties such as $p \implies \diamondsuit q$ over sequential programs. To handle this case, we define a state to be *pending* (with respect to p, q) if it is reachable by a q-free path from a reachable p-state. Then, we should find a well-founded ranking δ such that δ decreases on every non-idle step that departs from a pending state.

2.2 Predicate Abstraction

The material here is a summary of [8] and [2]. We fix an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ whose set of states is Σ. We consider a set of *abstract variables* $V_A = \{u_1, \ldots, u_n\}$ that range over finite domains. An *abstract state* is an interpretation that assigns to each variable u_i a value in the domain of u_i. We denote by Σ_A the (finite) set of all abstract states. An *abstraction mapping* is presented by a set of equalities

$$\alpha_\mathcal{E} : \quad u_1 = \mathcal{E}_1(V),\ \ldots,\ u_n = \mathcal{E}_n(V),$$

where each $\mathcal{E}_i$ is an expression over V ranging over the domain of u_i. The abstraction $\alpha_\mathcal{E}$ induces a semantic mapping $\alpha_\mathcal{E} : \Sigma \mapsto \Sigma_A$, from the states of $\mathcal{D}$ to the set of abstract states.

Usually, most of the abstract variables are boolean, and then the corresponding expressions $\mathcal{E}_i$ are predicates over V. This is why this type of abstraction is often referred to as *predicate abstraction*. The abstraction mapping $\alpha_\mathcal{E}$ can be expressed succinctly by:

$$V_A = \mathcal{E}(V)$$

Throughout the rest of the paper, when there is no ambiguity, we shall refer to $\alpha_\mathcal{E}$ simply as α. For an assertion $p(V)$, we define its α-abstraction (with some overloading of notation) by:

$$\alpha(p) : \ \exists V.(V_A = \mathcal{E}(V) \ \land \ p(V))$$

The semantics of $\alpha(p)$ is $\|\alpha(p)\| : \{\alpha(s) \mid s \in \|p\|\}$. Note that $\|\alpha(p)\|$ is, in general, an over-approximation – an abstract state S is in $\|\alpha(p)\|$ iff *there exists* some concrete p-state that is abstracted into S. A bi-assertion $\beta(V, V')$ is abstracted by:

$$\alpha^2(p) : \ \exists V, V'.(V_A = \mathcal{E}(V) \ \land \ V'_A = \mathcal{E}(V') \ \land \ \beta(V, V'))$$

The abstraction α is said to be *precise with respect to the assertion* p if $\alpha(\neg p) = \neg\alpha(p)$, implying that we cannot have a p-state and a $(\neg p)$-state both being abstracted into the same abstract state. For a temporal formula ψ in positive normal form (where negation is

applied only to state assertions), ψ^α is the formula obtained by replacing every maximal state sub-formula p in ψ by $\alpha(p)$.

In all cases discussed in this paper, we assume that the considered abstractions are precise with respect to the assertions appearing within the system definition and the property specification, and all temporal specifications of properties are given in positive normal form. Hence, we can restrict to the over-approximation semantics.

The abstraction of $\mathcal{D}$ by α is the system

$$\mathcal{D}^\alpha \;=\; \langle V_A, \alpha(\Theta), \alpha^2(\rho), \bigcup_{J \in \mathcal{J}} \alpha(J), \bigcup_{(p,q) \in \mathcal{C}} \langle \alpha(p), \alpha(q) \rangle \, \rangle$$

An abstraction α is called *idle preserving* if whenever $s_a \neq s_b$, where s_b is a ρ-successor of s_a, then $\alpha(s_a) \neq \alpha(s_b)$. Thus, the abstraction α does not generate new idling steps. In this paper we restrict our attention to idle-preserving abstractions. This is justified by the fact that all of our FDS's are derived from programs, and all considered abstractions preserve the values of the program counters.

The soundness of predicate abstraction is derived from [8]:

Theorem 1. *For a system $\mathcal{D}$, abstraction α, and a temporal formula ψ:*

$$\mathcal{D}^\alpha \models \psi^\alpha \qquad \Longrightarrow \qquad \mathcal{D} \models \psi$$

Thus, if an abstract system satisfies an abstract property, then the concrete system satisfies the concrete property.

2.3 Ranking Abstraction

State abstraction often does not suffice to verify progress properties. We consider *ranking abstraction*, a method of augmenting the concrete program in a non-constraining manner, in order to measure progress of program transitions, with respect to a ranking function. Once a program is augmented, a conventional state abstraction can be used to preserve the ability to monitor progress in the abstract system. This method was introduced in [8].

Ranking abstraction allows us to get away with finding a set of possible ingredients for ranking functions, without having to design a comprehensive single ranking function which is usually required in deductive verification of termination. This is accomplished by means of augmenting the system with several non-constraining monitors, and predicates abstracting the resulting system.

Fix some system $\mathcal{D}\colon \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ and some well-founded domain $(\mathcal{W}, \succ)$, and let δ be some ranking function over the domain. Let *dec* be a fresh variable (not in V). The *augmentation of $\mathcal{D}$ by δ*, written $\mathcal{D}{+}\delta$, is the system

$$\mathcal{D}{+}\delta\colon \langle V \cup \{dec\}, \Theta, \rho \;\wedge\; \rho_\delta, \mathcal{J}, \mathcal{C} \cup \{(dec > 0, dec < 0)\} \rangle$$

where the conjunct ρ_δ is defined by:

$$\rho_\delta\colon dec' = \begin{cases} 1 & \delta \succ \delta' \\ 0 & \delta = \delta' \\ -1 & \text{otherwise} \end{cases}$$

Thus, $\mathcal{D}+\delta$ behaves exactly like $\mathcal{D}$ and, in addition, keeps track of whether δ decreases, remains the same, or otherwise. The new compassion requirement captures the restriction that δ cannot decrease infinitely often without increasing infinitely often, which follows immediately from the well-foundedness of $\mathcal{W}$. For the pervasive case that δ ranges over the naturals, we can express ρ_δ as $dec' = sign(\delta - \delta')$.

Since augmentation does not constrain the behavior of $\mathcal{D}$, any property over V is valid over $\mathcal{D}$ iff it is valid over $\mathcal{D}+\delta$. In order to verify a liveness property of $\mathcal{D}$, the augmentation $\mathcal{D}+\delta$ can be predicate abstracted and checked for satisfiability of the abstracted property. Note that we do not require that δ decreases on every (non-idle) step. As demonstrated below, it suffices to have δ capture some of the behavioral aspects of a "comprehensive" ranking function.

Example 1 (Nested Loops). Consider program NESTED-LOOPS in Fig. 1(a). In this program, the statements "$x :=\ ?$" and "$y :=\ ?$," in lines 0 and 1 respectively, denote random assignments of arbitrary positive integers to variables x and y. An initial attempt to prove termination of this program is to define the ranking function $\delta_y = y$. The augmentation $\mathcal{D}+\delta_y$ is shown in Fig. 1(b). Note that statements that in the original program assign values to y, are now replaced with a simultaneous assignment to *both* y and the augmentation variable dec_y. In the case of control statements such as *while*, the augmentation is not displayed explicitly. However, it is implicitly assumed that the assignment $dec_y := 0$ is executed in parallel with any of these statements. Note that the assignments to dec_y have been optimized in some of the statements, replacing the expression $sign(y - y')$ by its values which are known to be 1 and 0 at the execution of statements 4 and 5, respectively.

```
x, y : natural init x = 0, y = 0
[ 0 : x := ?
    1 : while x > 0 do
    [ 2 : y := ?
        3 : while y > 0 do
        [ 4 : y := y − 1 ]
      5 : x := x − 1 ]
  6 : ]
```

(a) Program NESTED-LOOPS

```
x, y : natural     init x = 0, y = 0
dec_y : {−1, 0, 1}
compassion (dec_y > 0, dec_y < 0)
[ 0 : (x, dec_y) := (?, 0)
  1. while x > 0 do
  [ 2 : (y, dec_y) := (?, sign(y − y'))
      3 : while y > 0 do
      [ 4 : (y, dec_y) := (y − 1, 1) ]
    5 : (x, dec_y) := (x − 1, 0) ]
  6 : ]
```

(b) Program AUGMENTED-NESTED-LOOPS

Fig. 1. Program NESTED-LOOPS and its augmented version

While this augmentation is not sufficient to prove program termination, it can be used to prove termination of the inner loop (lines 3, 4).

Consider the abstraction:

$$\alpha : \quad \mathit{\Pi} = \pi,\ X = (x > 0),\ Y = (y > 0),\ \mathit{Dec}_y = dec_y$$

where Π is the abstract program counter. The resulting abstract program is presented in Fig. 2. Note that α introduces nondeterministic assignments to both X and Y (lines 4 and 5). It is now possible to verify, e.g. by model-checking, the termination of the inner loop.

Deductive verification of termination of the inner loop consisting of statements 3 and 4, requires the use of the adequate ranking function $2y + (\pi = 3)$ (where the boolean expression $(\pi = 3)$ evaluates to 1 on states in which π equals 3) or the function $\langle y, \pi = 3\rangle$ ranging over lexicographic pairs. However, supplying the model checker with the "ingredient rank" y suffices for the application of the ranking abstraction method. Obviously, to obtain the termination of the complete program, we'd need to also consider the variable x. ▪

$X, Y : \{0,1\}$ **init** $Y = 0, X = 0$
$Dec_y : \{-1, 0, 1\}$
compassion $(Dec_y > 0, Dec_y < 0)$
$$\left[\begin{array}{l}0 : (X, Dec_y) := (1, 0)\\ \quad 1 : \textbf{while } X \textbf{ do}\\ \quad\left[\begin{array}{l}2 : (Y, Dec_y) := (1, \{-1, 0, 1\})\\ \quad 3 : \textbf{while } Y \textbf{ do}\\ \quad\left[4 : (Y, Dec_y) := (\{0,1\}, 1)\right]\\ 5 : (X, Dec_y) := (\{0,1\}, 0)\end{array}\right]\\ 6 :\end{array}\right]$$

Fig. 2. Program ABSTRACT-AUGMENTED-NESTED-LOOPS

In a dual way to the computation of an abstraction of a concrete assertion, we can concretize an abstract assertion. Let Φ be an abstract assertion. The *concretization of* Φ, denoted by $\alpha^{-1}(\Phi)$, is defined as

$$\alpha^{-1}(\Phi) : \quad \exists V_A.(V_A = \mathcal{E}(V) \ \wedge \ \Phi(V_A))$$

For example, consider the abstract assertion $\Phi : \Pi = 3 \wedge X = 1 \wedge Y = 0$. Its concretization is given by $\alpha^{-1}(\Phi) : \pi = 3 \wedge x > 0 \wedge y = 0$.

As shown in Example 1, it is sometimes necessary to include several δ's in order to obtain a termination proof, by considering simultaneous augmentations by a set of ranking functions. A *ranking core* is a set of ranking functions. Let $\mathcal{R}$ be the ranking core $\{\delta_1, \ldots, \delta_k\}$. The augmentation $\mathcal{D}{+}\mathcal{R}$ is the system

$$\mathcal{D}{+}\mathcal{R} : (\mathcal{D}{+}\delta_1){+}\{\delta_2, \ldots, \delta_k\}$$

Just like the case of predicate abstraction, we lose nothing (except efficiency) by adding potentially redundant rankings. The main advantage here over direct use of ranking functions within deductive verification is that one may contribute as many elementary ranking functions as one wishes. It is then left to a model-checker to sort out their interaction and relevance. To illustrate this, consider a full deductive proof of termination of program NESTED-LOOPS. Due to the unbounded nondeterminism of the

random assignments, a deductive termination proof is necessarily based on a ranking function over lexicographic tuples, an example of which is the following:

$$\langle (\pi = 0), 4x + 3(\pi = 1) + 2(\pi = 2) + (\pi \in \{3, 4\}), 2y + (\pi = 3) \rangle$$

With ranking abstraction, however, one need only provide the well-founded ranking core $\mathcal{R} = \{x, y\}$.

To abbreviate the notation, we will write $\mathcal{D}^{\mathcal{R},\alpha}$ as shorthand for $(\mathcal{D}+\mathcal{R})^\alpha$. Note that, when we perform ranking abstraction w.r.t a core $\mathcal{R} : \delta_1, \ldots, \delta_k$, we use an abstraction mapping that extends α by the additional definitions:

$$Dec_1 = dec_1, \; \ldots, \; Dec_k = dec_k.$$

Since augmentation induced by the ranking core $\mathcal{R}$ does not constrain the behavior of the original FDS $\mathcal{D}$, it follows that every $\sigma : s_0, s_1, \ldots$, a computation of $\mathcal{D}$, gives rise to $\widetilde{\sigma} : \widetilde{s_0}, \widetilde{s_1}, \ldots$, a computation of $\mathcal{D}+\mathcal{R}$ agreeing with σ on all variable except for the *Dec* variables associated with $\mathcal{R}$. The computation $\widetilde{\sigma}$ can be abstracted into $\sigma^\alpha : S_0, S_1, \ldots$, a computation of $\mathcal{D}^{\mathcal{R},\alpha}$, such that $S_i = \alpha(\widetilde{s_i})$, for all $i \geq 0$. Thus, the set of computations of $\mathcal{D}$ is, modulo augmentation and abstraction, a subset of the computations of $\mathcal{D}^{\mathcal{R},\alpha}$. By arguments similar to the ones used in the proof of Theorem 1, we can establish the soundness of the ranking abstraction method ([8]).

Theorem 2. *For a system $\mathcal{D}$, abstraction α, ranking core $\mathcal{R}$, and a temporal formula ψ:*

$$\mathcal{D}^{\mathcal{R},\alpha} \models \psi^\alpha \qquad \Longrightarrow \qquad \mathcal{D} \models \psi$$

Thus, if a ranking-abstracted system satisfies an abstract property, then the concrete system satisfies the concrete property.
Ranking abstraction is more powerful than predicate abstraction, because we can also establish for it the following claim of completeness ([8]):

Theorem 3. *The method of ranking abstraction is complete. Namely, for every system $\mathcal{D}$ and temporal formula ψ, such that $\mathcal{D} \models \psi$, there exist an abstraction α and ranking core $\mathcal{R}$ such that $\mathcal{D}^{\mathcal{R},\alpha} \models \psi^\alpha$.*

3 Abstraction Refinement

In this section we will show that, similarly to predicate abstraction, ranking abstraction also possesses a counterexample guided refinement process. Assume that, wishing to check that $\mathcal{D} \models \psi$, we model checked $\mathcal{D}^{\mathcal{R},\alpha} \models \psi^\alpha$ and obtained an abstract counterexample σ^α. There are two possibilities. Either there exists a concrete computation σ, such that σ^α is the abstraction of σ, or σ^α cannot be concretized. In the first case, σ is a true counterexample, implying that ψ is not valid over $\mathcal{D}$. In the second case, this means that our abstraction is too coarse and needs to be refined.

The process of counterexample guided refinement has to distinguish between these two cases, and in the case of a spurious counterexample, to utilize the failure to concretize in order to refine the two abstraction components: α and $\mathcal{R}$. Note that the situation here is more complex than simple predicate abstraction, because the refinement may call for a refinement of α or of $\mathcal{R}$, or of both.

3.1 Abstract Runs and Their Concretizations

Let $\mathcal{D}\colon \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be a system, $\mathcal{P}$ be a predicate base, α be the abstraction mapping $\exists V. V_A = \mathcal{P}(V)$, and α^{-1} be α's inverse, i.e., the mapping $\exists V_A . V_A = \mathcal{P}(V)$, which we fix for the duration of this section. We refer to runs of $\mathcal{D}$ and $\mathcal{D}^{\mathcal{R},\alpha}$ as *concrete* and *abstract* runs, respectively. In this section, we assume that, unless explicitly stated otherwise, all runs are finite. For a run $\xi : s_0, \ldots, s_m$, we denote by $|\xi| = m$ the *length* of ξ.

Consider an abstract run $\Xi\colon S_0, \ldots, S_m$. A concrete run $\xi : s_0, \ldots, s_m$ is called a *concretization of* Ξ if $\alpha(s_i) = S_i$, for all $i \in [0..m]$. Rather then considering a single state concretization of the abstract run Ξ, we may wish to derive a characterization of all possible concretizations of Ξ. This may be captured by a sequence $\varphi_0, \ldots, \varphi_m$ of assertions over V. We thus define the *symbolic concretization of* Ξ (*with respect to* $\mathcal{D}$) to be the sequence $\gamma(\Xi)\colon \varphi_0, \ldots, \varphi_m$ of concrete assertions inductively as follows:

$$\varphi_i\colon \begin{cases} \Theta \;\wedge\; \alpha^{-1}(S_0) & i = 0 \\ (\varphi_{i-1} \diamond \rho) \;\wedge\; \alpha^{-1}(S_i) & i \in [1..m] \end{cases}$$

where $\varphi \diamond \rho$ is the assertion characterizing the states that are ρ-successors of a φ-state.

We sometimes refer to φ_i as $\gamma(\Xi)[i]$, or simply $\gamma[i]$ if Ξ is understood from the context.

Example 2. Recall program NESTED-LOOPS of Fig. 1(a), and consider the abstraction and ranking core of Example 1, that is:

$$\begin{aligned} \alpha &: X = (x > 0),\; Y = (y > 0),\; Dec_y = dec_y \\ \mathcal{R} &: \{\delta_1 = y\} \end{aligned}$$

The abstract system is shown in Fig. 2. Consider an abstract run $\Xi\colon S_0, \ldots, S_6$ of the system, where

$$\begin{array}{ll} S_0 : \langle \Pi{:}0,\ X{:}0,\ Y{:}0,\ Dec_y{:}0\rangle & S_1 : \langle \Pi{:}1,\ X{:}1,\ Y{:}0,\ Dec_y{:}0\rangle \\ S_2 : \langle \Pi{:}2,\ X{:}1,\ Y{:}0,\ Dec_y{:}0\rangle & S_3 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_y{:}-1\rangle \\ S_4 : \langle \Pi{:}4,\ X{:}1,\ Y{:}1,\ Dec_y{:}0\rangle & S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}0,\ Dec_y{:}1\rangle \\ S_6 : \langle \Pi{:}5,\ X{:}1,\ Y{:}0,\ Dec_y{:}0\rangle & \end{array}$$

The symbolic concretization of Ξ is $\varphi_0, \ldots \varphi_6$ where:

$$\begin{array}{l} \varphi_0\colon\; \pi = 0 \;\wedge\; x = 0 \;\wedge\; y = 0 \;\wedge\; dec_y = 0 \\ \varphi_1\colon\; \pi = 1 \;\wedge\; x > 0 \;\wedge\; y = 0 \;\wedge\; dec_y = 0 \\ \varphi_2\colon\; \pi = 2 \;\wedge\; x > 0 \;\wedge\; y = 0 \;\wedge\; dec_y = 0 \\ \varphi_3\colon\; \pi = 3 \;\wedge\; x > 0 \;\wedge\; y > 0 \;\wedge\; dec_y = -1 \\ \varphi_4\colon\; \pi = 4 \;\wedge\; x > 0 \;\wedge\; y > 0 \;\wedge\; dec_y = 0 \\ \varphi_5\colon\; \pi = 5 \;\wedge\; x > 0 \;\wedge\; y = 0 \;\wedge\; dec_y = 1 \\ \varphi_6\colon\; \pi = 6 \;\wedge\; x > 0 \;\wedge\; y = 0 \;\wedge\; dec_y = 0 \end{array}$$

┛

The following claim establishes the relation between symbolic concretizations of abstract runs and concrete runs:

Claim 1 (Feasibility). For every abstract run Ξ, $m \in [0..|\Xi|]$, and concrete state s, s satisfies $\gamma[m]$ iff there exists a concrete run $s_0, \ldots, s_m = s$ that is a concretization of $\Xi[0..m]$.

Proof. Assume that $s \models \gamma[m]$. Proceeding from m down to 0, we will construct a sequence of states $s = s_m, s_{m-1}, \ldots, s_0$, such that, for each $i = 0, \ldots, m$, $\Xi[i] = \alpha(s_i)$ and $s_i \models \gamma[i]$, s_0 is initial, and, for each $i \in [0..m-1]$, s_{i+1} is a ρ-successor of s_i.

For every $i \in [1..m]$ assume that we already constructed s_i, such that $s_i \models \gamma[i]$. The fact that s_i satisfies $\gamma[i] = (\gamma[i-1] \diamond \rho) \ \wedge \ \alpha^{-1}(\Xi[i])$ implies that $\Xi[i] = \alpha(s_i)$ and that there exists a state s_{i-1} which is a ρ-predecessor of s_i and satisfies $\gamma[i-1]$.

For $i = 0$, the fact that $s_0 \models \gamma[0] = \alpha^{-1}(S_0) \ \wedge \ \Theta$ implies that s_0 is an initial state such that $S_0 = \alpha(s_0)$.

Thus, the state sequence $\xi : s_0, \ldots, s_m = s$ is a concretization of $\Xi[0..m]$.
In the other direction the claim is straightforward. □

A corollary of Claim 1 is that an abstract run $\Xi[1..m]$ can be concretized iff $\gamma(\Xi)[m]$ is satisfiable.

The assertion $\gamma(\Xi)[i]$ characterizes all the states that can appear at position i of a concretization of the abstract run Ξ. We will generalize this notion by defining a bi-assertion $\beta_{i,j}(\Xi)$, for $0 \leq i \leq j \leq |\Xi|$, such that $\langle s_a, s_b \rangle \models \beta_{i,j}$ iff there exists $\xi : s_0, \ldots, s_{|\Xi|}$, such that $s_i = s_a$ and $s_j = s_b$. This bi-assertion will be used when attempting to concretize "abstract cycles" that are obtained in counterexamples. The generic presentation of the bi-assertion is $\beta_{i,j}(V_0, V)$ (rather than $\beta_{i,j}(V, V')$), where V_0 is a fresh copy of the system variables, and records the values of variables at state s_i.

Let $\Xi = S_0, \ldots, S_m$ be an abstract run. The bi-assertion $\beta_{i,j}(\Xi)$ is defined inductively, for all j, $i \leq j \leq m$ by:

$$\beta_{i,j} = \begin{cases} V = V_0 \ \wedge \ \alpha^{-1}(S_i) & j = i \\ (\beta_{i,j-1} \diamond \rho) \ \wedge \ \alpha^{-1}(S_j) & j > i \end{cases}$$

In this definition, $V = V_0$ is an abbreviation for $\bigwedge_{x \in V}(x = x_0)$, which states equality between all V-variables and their corresponding V_0-counterparts. The expression $\beta_{i,j-1} \diamond \rho$ stands for

$$\exists \widetilde{V} : (\beta_{i,j-1}(V_0, \widetilde{V}) \wedge \rho(\widetilde{V}, V))$$

Note in particular that this expression preserves the values of the V_0-variables from $\beta_{i,j-1}$ to $\beta_{i,j}$.

Example 3. Continuing Example 2, we compute $\beta_{1,1}, \ldots, \beta_{1,6}$ as follows:

$$\begin{aligned}
&\beta_{1,1} : \textit{init} \ \wedge \ \pi = \pi_0 \ \wedge \ x = x_0 \ \wedge \ y = y_0 \ \wedge \ \textit{dec}_y = \textit{dec}_y^0 \\
&\beta_{1,2} : \textit{init} \ \wedge \ \pi = 2 \ \wedge \ x = x_0 \ \wedge \ y = y_0 \ \wedge \ \textit{dec}_y = 0 \\
&\beta_{1,3} : \textit{init} \ \wedge \ \pi = 3 \ \wedge \ x = x_0 \ \wedge \ y > 0 \ \wedge \ \textit{dec}_y = -1 \\
&\beta_{1,4} : \textit{init} \ \wedge \ \pi = 4 \ \wedge \ x = x_0 \ \wedge \ y > 0 \ \wedge \ \textit{dec}_y = 0 \\
&\beta_{1,5} : \textit{init} \ \wedge \ \pi = 3 \ \wedge \ x = x_0 \ \wedge \ y = 0 \ \wedge \ \textit{dec}_y = 1 \\
&\beta_{1,6} : \textit{init} \ \wedge \ \pi = 5 \ \wedge \ x = x_0 \ \wedge \ y = 0 \ \wedge \ \textit{dec}_y = 0
\end{aligned}$$

where $init : \pi_0 = 1 \wedge x_0 > 0 \wedge y_0 = 0 \wedge dec_y^0 = 0$. ■

3.2 Counterexample Guided Abstraction Refinement

The verification (or refutation) of a progress property ψ over an FDS begins with a (possibly empty) user-provided initial ranking $\mathcal{R}$ and a predicate abstraction $\mathcal{P}$. Following [13], initially $\mathcal{P}$ is chosen to be the set of atomic state formulas occurring in ρ, Θ, $\mathcal{J}$, $\mathcal{C}$ and the concrete formula ψ, excluding formulas that refer to control and primed variables.

Let ψ^α be the formula $\alpha(\psi)$. We start by model checking the validity of $\alpha(\psi)$ over $\mathcal{D}^{\mathcal{R},\alpha}$. If ψ^α is valid then we can conclude that $S \models \psi$. Otherwise, a counterexample is found in the form of a computation of $\mathcal{D}^{\mathcal{R},\alpha}$ that does not satisfy ψ^α. If such a computation exists then a standard model checker will return a counterexample that is finitely represented as a "lasso" – an abstract run of the form $\Xi_1; \Xi_2^\omega$ where $\Xi_1 : S_0, \ldots, S_{k-1}$ is a finite abstract run, and $\Xi_2 : S_k, \ldots, S_{m-1}$ is a finite sequence of consecutive abstract states. As in the case of predicate abstraction refinement, we first attempt to concretize the counterexample $\Xi : S_0, \ldots, S_k, \ldots, S_{m-1}, S_m = S_k$. Namely, we compute $\gamma(\Xi) : \varphi_0, \ldots, \varphi_m$ and $\beta_{k,m}(\Xi)$. The following may occur:

Case 1. The counterexample Ξ cannot be concretized.
This case is identified by observing that $\varphi_m = \gamma[m]$ is unsatisfiable. This is a typical scenario in state abstraction refinement – the abstraction is too coarse, and should be refined so as to eliminate the spurious counterexample. One can apply any of the known predicate refinement techniques, e.g., [5, 3, 4]. For all following cases, we may assume that φ_m is satisfiable.

Case 2. The concretization of the counterexample contains a cycle compatible with Ξ_2 — the property is not valid.
This case is identified by observing that $\varphi_k(V) \wedge \beta_{k,m}(V, V)$ is satisfiable. This implies that there exists a state s such that $s \models \varphi_k$ and $\langle s, s \rangle \models \beta_{k,m}$, and therefore, there exists a state concretization of Ξ of the form $\xi : s_0, \ldots, s_k, \ldots, s_m = s_k = s$. It follows that the infinite concrete run $s_0, \ldots, s_{k-1}(s_k, \ldots, s_{m-1})^\omega$ is a computation of $\mathcal{D}$ which violates ψ. We conclude that ψ is not $\mathcal{D}$-valid.

Case 3. The infinite abstract run $\Xi_1; \Xi_2^\omega$ cannot be concretized — the abstract counterexample is spurious; perform ranking refinement.
This case is identified by observing that the bi-assertion $\beta_{k,m}$ is well-founded over φ_k. Obviously, if $\Xi_1; \Xi_2^\omega$ could be concretized by the infinite concrete run $s_0, s_1, \ldots,$ then we would have had an infinite state sequence, namely $s_k, s_{k+L}, s_{k+2L}, \ldots,$ where $L = m - k$, such that $s_k \models \varphi_k$ and $\beta_{k,m}$ holds between any two consecutive states in this sequence. This would have contradicted the fact that $\beta_{k,m}$ is well-founded over φ_k. We conclude that the counterexample is spurious.

This case is a typical scenario in ranking abstraction refinement – the ranking is too coarse, and should be refined to eliminate the spurious counterexample. The ranking core is refined by adding to it a well-founded ranking that proves the well-foundedness of $\beta_{k,m}$ over φ_k.

A number of methods have been proposed to synthesize such functions from well-founded relations, among them in [12, 7]. In Subsection 3.3 we present an additional heuristic for the domain of unbounded linked lists.

Case 4. The infinite abstract run $\Xi_1; \Xi_2^\omega$ can be concretized — the property is not valid.

This case can be identified by observing that the bi-assertion $\beta_{k,m}$ is not well-founded over φ_k. From the fact that $\beta_{k,m}$ is not well-founded, we can infer the existence of an infinite sequence $s_k, s_{k+L}, s_{k+2L}, \ldots$. This sequence can be enriched by filling in the missing states to form a concretization of $\Xi_1; \Xi_2^\omega$ that is a computation of $\mathcal{D}$ violating the property ψ.

In this case we can declare the property ψ to be invalid over the concrete program.

The process is described in Fig. 3.

CEGAR($\mathcal{D}, \psi, \mathcal{P}, \mathcal{R}$)
1. **let** $\mathcal{D}_A = \mathcal{D}^{\mathcal{R},\alpha}$;
2. **let** $\psi_A = \psi^\alpha$;
3. **if** $\mathcal{D}_A \models \psi_A$ **then**
   4. **return "success"**
   else
   5. **let** $C = S_0 \cdots S_{k-1}(S_k \cdots S_{m-1})^\omega$ be a computation of $\mathcal{D}_A$ such that $C \models \neg\psi_A$;
   6. **let** $\Xi = S_0, \ldots, S_k, \ldots, S_{m-1}, S_m = S_k$;
   7. **Compute** $\gamma(\Xi) : \varphi_0, \ldots, \varphi_m,$ and $\beta_{k,m}(\Xi)$;
   8. **if** $(\exists i : 0 \leq i \leq m : \neg sat(\varphi_i))$ **then** —— Case 1
      9. **let** $\mathcal{P}'$ be a predicate refinement of $\mathcal{P}$ induced by the failure to concretize Ξ;
      10. **return** CEGAR($\mathcal{D}, \psi, \mathcal{P}', \mathcal{R}$);
   11. **else if** $sat(\varphi_k(V) \wedge \beta_{k,m}(V, V))$ **then** —— Case 2
      12. **let** $\xi : s_0, \ldots, s_k, \ldots, s_m = s_k$ be the concrete run concretizing Ξ;
      13. **Return** "Property not valid. Counterexample: ξ";
   14. **else if** ($\beta_{k,m}$ is well-founded over φ_k) **then** —— Case 3
      15. **let** δ be a well-founded ranking proving the well-foundedness of $\beta_{k,m}$ over φ_k;
      16. **return** CEGAR($\mathcal{D}, \psi, P, \mathcal{R} \cup \{\delta\}$);
   17. **else return** "Property not valid. Counterexample: $s_k, s_{k+L}, s_{k+2L}, \ldots$"; —— Case 4

Fig. 3. CounterExample Guided Abstraction Refinement algorithm

Lines 1 and 2 abstract the system and the property respectively with respect to the ranking core $\mathcal{R}$ and the predicate base $\mathcal{P}$. Line 3 (model-)checks whether the abstract property holds over the abstract program. If so, the algorithm returns "success" (line 4). Else, a finitely-representable counterexample is produced (by a model checker) in line 5, from which we construct the "lasso" $\Xi : S_0, \ldots, S_{m-1}, S_m = S_k$ at line 6. Line 7 computes the symbolic concretization $\gamma(\Xi)$ and the bi-assertion $\beta_{k,m}(\Xi)$. Line

8 checks whether the symbolic concretization is satisfiable. If it is not satisfiable (Case 1), then predicate refinement is applied (line 9) and the algorithm is re-started with the augmented predicate base (line 10).

If the symbolic concretization is satisfiable, then we check in line 11 whether $\varphi_k(V) \wedge \beta_{k,m}(V, V)$ is satisfiable. If it is satisfiable (Case 2) then we can construct a concrete lasso $\xi : s_0, \ldots, s_k, \ldots, s_{m-1}, s_m = s_k$ concretizing Ξ that is therefore a concrete counterexample.

If the above two tests were answered negatively, we check in line 14 whether the bi-assertion $\beta_{k,m}(\Xi)$ is well-founded over φ_k. If it is (Case 3) then we know that the abstract counterexample is spurious. In line 15, we attempt to construct a well-founded ranking δ which proves the well-foundedness over φ_k of $\beta_{k,m}(\Xi)$. If we succeed to identify such a δ then it is added to the ranking core as a refinement, and we reiterate the algorithm with the extended ranking core. This step is the least constructive in the algorithm, and the best that can be offered is a set of heuristics for finding a well-founded ranking that can prove the well-foundedness of a given bi-assertion.

Finally, if all preceding tests fail, we reach line 17 (Case 4). In this case, $\beta_{k,m}$ is known not to be well-founded. This implies that there exists a concrete counter example, but not necessarily one that can be presented in finite terms. The best that we can do is present to the user a prefix of a potentially infinite counterexample, as explained in the preceding discussion of Case 4.

The algorithm may not terminate (assuming even an extremely powerful model checker). For one, predicate refinement is not guaranteed to terminate. Similarly, ranking refinement may not terminate. Furthermore the test at line 14, which decides whether we are in Case 3 or Case 4, is, in general, undecidable.

Example 4 (Termination of NESTED-LOOPS*).* Recall program NESTED-LOOPS presented in Fig. 1(a) and the termination property expressed as $(\pi = 0) \Longrightarrow \diamondsuit(\pi = 4)$. Following Example 2, we begin with the initial abstraction and ranking used in Example 1. The first iteration of CEGAR results in an abstract counterexample consisting of the following single-state lasso prefix S_0 and the repeating period $(S_1, \ldots, S_6)$ where $S_0, \ldots, S_6$ are as in Example 2. The abstract lasso derived from this counterexample is $\Xi : S_0, S_1, \ldots, S_6, S_7 = S_1$. We follow the computation of Example 3 to obtain the bi-assertions $\beta_{1,1}, \ldots, \beta_{1,6}$, and also compute

$$\beta_{1,7}\colon \textit{init} \wedge \pi = 1 \wedge x = x_0 - 1 \wedge x > 0 \wedge y = 0 \wedge \textit{dec}_y = 0$$

It follows that $\beta_{1,7}(V_0, V)$ implies $x_0 > x > 0$, which is well-founded. A well-founded ranking function proving well-foundedness of $\beta_{1,7}$ is $\delta_2 = x$ over the domain $(\mathbb{N}, >)$. Thus we refine $\mathcal{R}$ and continue with the same predicate base and the refined ranking core $\mathcal{R}'\colon \{\delta_1, \delta_2\}$. At this point, the abstraction of $S{+}\mathcal{R}'$ by α is sufficient to verify the termination property of the program. ∎

3.3 Synthesizing Elementary Ranking Functions

A number of methods have been suggested for synthesis of ranking functions that establish (prove) well-foundedness of a well-founded bi-assertion. In our examples we

have used the simple heuristic of searching for simple linear constraints implied by the transition relation of a control-flow loop ([12] provides a more general method for doing this. Indeed, their method is complete). For example, given a set of variables V and a bi-assertion β, we check validity of implications such as $\beta \to v > v'$, for each $v \in V$. As demonstrated, this has been sufficient in dealing with the NESTED-LOOPS program. A more general approach based on linear algebra may look for ranking functions that are linear combinations of system variables.

Such an extraction is useful in two contexts in which bi-assertions may arise. The first has been demonstrated in the ranking refinement process. The second is related to the determination of the ranking components that should be placed in the initial ranking core. This can be based on a heuristic that analyzes the various loops in the program. Assume a control loop identified by a sequence of locations $L = \ell_1, \ldots, \ell_n = \ell_1$, such that, for each $i = 1, \ldots, n-1$, ℓ_{i+1} can be reached from ℓ_i in a single step. For a loop L, we can define a sequence of bi-assertions as follows:

$$\beta_{i,j} = \begin{cases} V_0 = V \ \wedge \ \pi = \ell_i & j = i \\ (\beta_{i,j-1} \diamond \rho) \ \wedge \ \pi = \ell_j & j > i \end{cases}$$

It only remains to check whether the bi-assertion $\beta_L = \beta_{1,n}$ is well-founded, and identify well-founded ranking functions that prove the well-foundedness of β_L. Such an identification is, in general, undecidable, but we can use any of the heuristics mentioned above, such as linear analysis.

We have used a variant of this heuristic to deal with programs that manipulate unbounded pointer structures. One such program is BUBBLE SORT, shown in Fig. 4. This is a parameterized system with H denoting the maximal size of a singly-linked pointer structure (or *heap*). The heap itself is represented by the array Nxt. In addition there are a number of *pointer* variables, such as x and y, that are also parameterized by H. In the program, as well as in the ranking functions we will use, the assertion $Nxt^*(u, v)$ denotes the second-order formula

$$\exists n, \{u = u_0, u_1, \ldots, u_n = v\}. \forall i \in [1 \ldots n]. u_i = Nxt[u_{i-1}]$$

That is, $Nxt^*(u, v)$ is satisfied when there is a sequence of graph nodes starting at u and ending at v, that are consecutive with respect to Nxt.

In order to synthesize a ranking function for BUBBLE SORT and similar programs, our strategy is to seek constraints on graph reachability. One such form of constraint is

$$\beta_L \to (Nxt^*(v, v') \ \wedge \ v \neq v')$$

where β_L is the bi-assertion associated with the loop L and v is a pointer variable. Under the assumption that a singly-linked list emanating from v is acyclic, such a constraint suggests the ranking function $\{i \mid Nxt^*(v, i)\}$ over the domain $(2^{\mathbb{N}}, \supset)$. Indeed, while proving termination of BUBBLE SORT, one of the functions discovered automatically by refinement was $\{i \mid Nxt^*(yn, i)\}$, a function that serves to prove termination of the nested loop ($L =$ lines $2 \ldots 9$).

```
x, y, yn, prev, last : [0..H]
Nxt                  : array [0..H] of [0..H] where Nxt*(x, nil)
D                    : array [0..H] of bool
0 :  (prev, y, yn, last) := (nil, x, Nxt[x], nil);
1 :  while last ≠ Nxt[x] do
       2 : while yn ≠ last do
             3 : if (D[y] > D[yn]) then
                   4 : (Nxt[y], Nxt[yn]) := (Nxt[yn], y);
                   5 : if (prev = nil) then
                           6 :  x := yn
                       else
                           7 :  Nxt[prev] := yn;
                   8 : (prev, yn) := (yn, Nxt[y])
                 else
                   9 : (prev, y, yn) := (y, yn, Nxt[y])
10 : (prev, y, yn, last) := (nil, x, Nxt[x], y);
11 :
```

Fig. 4. Program BUBBLE SORT

4 Extracting A Deductive Proof

There are situations in which verification alone is not sufficient, and an actual proof is required. This is the case, for example, when the verification effort is embedded in a larger proof-generating effort, either because we consider only a component of the system, or are verifying a property that is only a part of the full specification. When dealing with safety properties, it is straightforward to generate a concrete logical formula that represents an inductive invariant, based on the set of reachable abstract states, to be used as the basis of a deductive proof. The analogous constructs in the case of a response-property proof consist of an assertion that over-approximates the set of pending states, and a well-founded, always-decreasing ranking function.

In this section we present algorithms that extract the necessary auxiliary constructs from a successful application of the ranking abstraction method. The algorithm is based on the LTL model-checking algorithm of [9]. For simplicity, we consider here the case of an FDS derived from a sequential program and, therefore, has no justice or compassion requirements. We will consider the more general case of concurrent programs in the next section.

4.1 Extracting A Deductive Proof of Invariance Properties

For the sake of completeness and emphasizing the analogy between predicate abstraction and ranking abstraction, we present here the process of the extraction of a deductive proof of an invariance property from a successful application of predicate abstraction.

In Fig. 5, we present the deductive rule INV for establishing the validity of the invariance property $\Box\, p$. The application of the rule calls for the identification of an

Rule INV
For assertions p, φ,
I1. $\Theta \rightarrow \varphi$
I2. $\varphi \wedge \rho \rightarrow \varphi'$
I3. $\varphi \rightarrow p$
$\Box\, p$

Fig. 5. Deductive rule INV

auxiliary assertion φ that, together with p, satisfies premises I1 — I3. We refer to an assertion that satisfies premises I1 and I2 as *inductive*.

Let $\mathcal{D}$ be an FDS for which we wish to verify the invariance property $\psi : \Box\, p$. Assume that we employed the predicate abstraction $\alpha : V_A = \mathcal{P}(V)$ and verified, by model checking, that $\mathcal{D}^\alpha \models \Box\, p^\alpha$. By soundness of the predicate abstraction method we can conclude that $\mathcal{D} \models \Box\, p$. It only remains to extract a deductive proof of this fact.

In Fig. 6, we present algorithm EXTRACT-INVARIANCE, which extracts an auxiliary assertion φ from the abstracted system $\mathcal{D}^\alpha$. The algorithm computes first in Φ an

Algorithm EXTRACT-INVARIANCE $(\mathcal{D}, \alpha)$
1. **Compute** $\mathcal{D}^\alpha$;
2. **let** $\Phi := \Theta^\alpha \diamond (\rho^\alpha)^*$;
3. **let** $\varphi := \alpha^{-1}(\Phi)$;

Fig. 6. An algorithm for extracting an inductive assertion

abstract assertion that characterizes all abstract states that are reachable in $\mathcal{D}^\alpha$. It then concretizes Φ into φ by applying the concretization mapping α^{-1}.

The correctness of the algorithm is stated by the following claim:

Claim 2 (Extraction of inductive assertion).
For any $\mathcal{D}$ and α, the assertion φ extracted by Algorithm EXTRACT-INVARIANCE is inductive over $\mathcal{D}$. If $\mathcal{D}^\alpha \models \Box\, p^\alpha$ then also $\varphi \rightarrow p$ is valid.

It follows that if we apply the extraction algorithm to a system after a successful application of the predicate abstraction method, then the extracted assertion φ satisfies all the premises of rule INV.

Example 5 (Extracting a deductive proof of invariance for NESTED-LOOPS*).*
Consider program NESTED-LOOPS presented in Fig. 1(a). For this program, we wish to prove the invariance of the assertion $p : \pi = 6 \rightarrow \neg(y > 0)$, claiming that when execution reaches location 6, then $y = 0$. Applying the predicate abstraction α introduced in Example 1, we obtain the abstract program presented in Fig. 2 when we omit all references to variable Dec_y. The property $\Box\, p$ is abstracted by α into the abstract property $\Box(\Pi = 6 \rightarrow Y \neq 1)$.

Computing the set of reachable states in this abstract program we obtain a set that is captured by the following abstract assertion:

$$\begin{array}{lllll} \Phi: & (X \rightarrow \Pi \in [1..5]) & \wedge & (\Pi \in [2..5] \rightarrow X) & \wedge \\ & (Y \rightarrow \Pi \in [3..4]) & \wedge & (\Pi = 4 \rightarrow Y) & \end{array}$$

Concretizing by α^{-1}, we obtain the following candidate assertion for φ:

$$\begin{array}{lllll} \varphi: & (x > 0 \rightarrow \pi \in [1..5]) & \wedge & (\pi \in [2..5] \rightarrow x > 0) & \wedge \\ & (y > 0 \rightarrow \pi \in [3..4]) & \wedge & (\pi = 4 \rightarrow y > 0) & \end{array}$$

It is not difficult to verify independently that φ is indeed inductive, and that φ implies $\pi = 6 \rightarrow \neg(y > 0)$. ▟

4.2 Deductive Rules for Response Properties

Moving to response properties, we consider a property of the form $p \Longrightarrow \diamondsuit\, q$. In this section we restrict our attention to FDS's that are derived from sequential programs. This implies that the FDS has no fairness requirements, and that idling steps are allowed only from terminal states. A basic proof rule BASIC-RESPONSE for the deductive verification of such a property is presented in Fig. 7

Rule BASIC-RESPONSE
For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions p, q, φ,
and ranking function $\Delta : \Sigma \mapsto \mathcal{A}$

$$\begin{array}{llll} \text{B1.} & p & \Longrightarrow & q \vee \varphi \\ \text{B2.} & \varphi \wedge \rho & \Longrightarrow & q' \ \vee \ \varphi' \wedge \Delta \succ \Delta' \\ \hline & p & \Longrightarrow & \diamondsuit\, q \end{array}$$

Fig. 7. Deductive rule BASIC-RESPONSE

The rule calls for the identification of an auxiliary assertion φ and a ranking function Δ over the well-founded domain $\mathcal{A}$. Assertion φ is intended to be an over-approximation of the set of pending states w.r.t assertions p and q. It is possible to view this rule as stating that the property $\psi : p \Longrightarrow \diamondsuit\, q$ is valid over $\mathcal{D}$ whenever the transition relation ρ, when restricted to the pending states (or their over-approximation φ), forms a well-founded bi-assertion. The well-founded ranking Δ is a ranking that proves the well-foundedness of the bi-assertion derived from ρ.

In practice, it is often useful to partition φ into several disjoint assertions that cover different cases. This leads to rule SEQUENTIAL-RESPONSE, which is presented in Fig. 8.

The rule uses assertions $\varphi_0, \ldots, \varphi_m$, where $\varphi_0 = q$. It is not difficult to see that if we can find a set of constructs (assertions and ranking functions) satisfying the premises

Rule SEQUENTIAL-RESPONSE
For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
and ranking functions Δ_0 , Δ_1 , ..., Δ_m where each $\Delta_i : \Sigma \mapsto \mathcal{A}$
R1. $p \implies \bigvee_{j=0}^{m} \varphi_j$
For each $i = 1, \ldots, m$,
R2. $\varphi_i \wedge \rho \implies \bigvee_{j=0}^{m} (\varphi'_j \wedge \Delta_i \succ \Delta'_j)$
$p \implies \diamondsuit\, q$

Fig. 8. Deductive rule SEQUENTIAL-RESPONSE

of rule SEQUENTIAL-RESPONSE, we can immediately construct the appropriate constructs necessary for rule BASIC-RESPONSE. This can be done by taking

$$
\begin{array}{ll}
\varphi & : \varphi_1 \vee \cdots \vee \varphi_m \\
\Delta & : \textbf{case} \\
& \qquad \varphi_1 : \Delta_1 \\
& \qquad \cdots \\
& \qquad \varphi_m : \Delta_m \\
& \textbf{otherwise} : 0 \\
& \textbf{end-case}
\end{array}
$$

It is customary to refer to assertions $\varphi_0, \ldots, \varphi_m$ as the *helpful assertions*.

In the rest of the section we will show how the constructs needed for rule SEQUENTIAL-RESPONSE, i.e. the ranks $\Delta_0, \ldots, \Delta_m$ and helpful assertions $\varphi_0, \ldots, \varphi_m$, can be extracted from a successful application of the ranking abstraction method.

4.3 Extracting the Ranking Functions

Let $\mathcal{D}, \mathcal{R}$, and α be a concrete system, a ranking core, and an abstraction mapping, respectively. Assume that $\psi : p \implies \diamondsuit\, q$ is the response property we wish to verify over $\mathcal{D}$. Let $\mathcal{D}^{\mathcal{R},\alpha}$ be the abstract system and ψ^α be the abstracted property.

The extraction process proceeds in two steps, where in the first step we extract the ranking functions $0 = \Delta_0, \Delta_1, \ldots, \Delta_m$ and, in the second step, we construct the assertions $q = \varphi_0, \varphi_1, \ldots, \varphi_m$. The well-founded domain $\mathcal{A}$ will be constructed incrementally together with the construction of the Δ_i's.

We start by constructing a transition graph $G : \langle N, E \rangle$, whose nodes are given by $N = \mathit{pend} \cup \{g\}$, where *pend* is the set of all pending states, and g is a special *goal* node representing all q^α-states that are reachable from a pending state in one step. Recall that the pending states are all the states that are reachable by a q^α-free path from a reachable p^α-state. The edges consist of all transitions connecting one pending state to another. We also include an edge connecting $n \in \mathit{pend}$ to g, if there exists a transition connecting state n to any non-pending state. For simplicity, we omit all edges connecting any node to itself, because they cannot contribute to the ranking. We will refer to the nodes of the graph as $N = \{g, S_1, \ldots, S_m\}$, where g is the goal node and $S_1, \ldots, S_m$ are the abstract pending states. We refer to G as the *pending graph* of system $\mathcal{D}^{\mathcal{R},\alpha}$.

The ranking function will be represented as a mapping $\mathit{Rank} : N \rightarrow$ TUPLES, where TUPLES is the type of lexicographic tuples whose elements are either natural numbers

or ranking functions present in the ranking core $\mathcal{R}$. The ranking *Rank* is initialized as *Rank*$[n] = \bot$ for each $n \in N$, where $\bot$ is the empty tuple. Then the recursive procedure RANK-GRAPH(G), shown in Fig. 9, is invoked. In each iteration, the algorithm updates the mapping *Rank* by concatenating additional components (natural numbers or elements of the ranking core) to the right of tuples.

Algorithm RANK-GRAPH(G)
Input: a graph $G = (N, E)$ representing the pending states for the abstract system.
Output: *Rank*, an array $N \mapsto$ TUPLES
Initially: For every $n \in N$, *Rank*$(n) = \bot$.
Rank(G):

1. Decompose G into a sorted list of MSCCs (maximal strongly connected components) $G = C_0, ..., C_k$, where sortedness means that $i > j$ whenever there is an edge from a C_i-node into a C_j-node;
2. For every node $n \in C_i$, append i to *Rank*(n);
3. Perform the following for each non-singleton MSCC C in the decomposition:
   (a) If for some compassion requirement $(Dec_j > 0, Dec_j < 0)$, C has some nodes with $Dec_j > 0$, but no nodes with $Dec_j < 0$, then append δ_j to *Rank*(n) of every node $n \in C$; if no such j exists, report "failure" and halt;
   (b) Let D be the subgraph obtained by removing every edge in C leading into a $Dec_j > 0$ node;
   (c) Call Rank(D);

Fig. 9. Procedure RANK-GRAPH, which constructs a ranking function from the transition graph of a terminating abstract system.

When Algorithm RANK-GRAPH terminates, it produces a list of ranking functions $\Delta_0, \Delta_1, \ldots, \Delta_m$, where Δ_0 is the rank associated with node g (usually 0), while $\Delta_1, \ldots, \Delta_m$ correspond to abstract states $S_1, \ldots, S_m$, respectively.

```
          X, Y          : {0,1}        init Y = 0, X = 0
          Dec_y, Dec_x  : {-1,0,1}
compassion {(Dec_x > 0, Dec_x < 0), (Dec_y > 0, Dec_y < 0)}
  [ 0 : (X, Dec_y, Dec_x) := (1, 0, -1)
  [ 1 : while X do
  [      [ 2 : (Y, Dec_y, Dec_x) := (1, -1, 0)
  [      [ 3 : while Y do
  [      [       [ 4 : (Y, Dec_y, Dec_x) := ({0,1}, 1, 0) ]
  [      [ 5 : (X, Dec_y, Dec_x) := ({0,1}, 0, 1)          ]
  [ 6 :                                                     ]
```

Fig. 10. Abstraction of Program NESTED-LOOPS augmented with ranking core $\{y, x\}$

Example 6 (Extracting a Ranking Function for NESTED-LOOPS*).* We illustrate the algorithm by extracting the ranking function for the deductive proof of termination of program NESTED-LOOPS, given the abstract program shown in Fig. 10. This is a version of the augmented abstract version from Fig. 2, after refinement with the additional ranking function $\delta_2 : x$. The response property we wish to verify is that of termination, which can be specified by the formula $at_\ell_0 \implies \diamondsuit\, at_\ell_6$, where at_ℓ_i is an abbreviation of the assertion $\pi = i$.

Fig. 11 visualizes iterations in the progress of the algorithm, as a series of graphs of the pending states of the abstract FDS, with nodes representing states and directed edges representing transitions. The goal state for the property appears as a graph node that is labeled with $\Pi = 6$ at the bottom of each diagram.

As the algorithm proceeds, each node is associated with a tuple that denotes the ranking generated thus far.

Fig. 13 summarizes the process as a table. The ranking procedure proceeds as follows: Initially the graph of Fig. 11 is decomposed into components $0, \ldots, 4$, and nodes are assigned ranks according to the index of their MSCC. This is shown in Fig. 11(a) and in the "Iteration 1" column of Fig. 13. Since nodes $\{S_2, \ldots .S_8\}$ form an MSCC that violates the compassion requirement $\langle Dec_x > 0, Dec_x < 0\rangle$, the corresponding ranking function x is appended to their ranking tuples, as shown in Fig. 11(b) and in the "Iteration 2" column. The ranking procedure is now applied recursively to the subgraph that consists of nodes $\{S_2, \ldots .S_8\}$ and all edges except for the one entering the $(Dec_x > 0)$ node (S_8). The subgraph, which is no longer strongly connected due to edge removal, is re-decomposed into MSCCs, and each tuple is updated (by concatenation to the right) with the new MSCC indices (shown in Fig. 11(c) and in the "Iteration 3" column). The component consisting of nodes S_4 and S_5 is now a non-singleton MSCC, and it violates the compassion requirement $\langle Dec_y > 0, Dec_y < 0\rangle$. Therefore, the corresponding ranking function y is appended to the rankings of nodes S_4 and S_5, as shown in Fig. 11(d) and in the "Iteration 4" column. Again, the procedure is applied recursively to the subgraph with nodes S_4 and S_5 that has all edges but the one leading into the $(Dec_y > 0)$ node (S_5). The rankings of nodes S_4 and S_5 are appended index values corresponding to a new sorting (Fig. 11(d) and column "Iteration 5"). At this point there are no non-singleton MSCCs left in the graph and the procedure terminates. The final ranking, with zeroes padded to the right where appropriate, is shown in Fig. 12 and in the "final ranking" column. ▟

Let $\Delta_i = (a_1, \ldots, a_r)$ and $\Delta_j = (b_1, \ldots, b_r)$ be two ranks. The formula

$$gt(\Delta_i, \Delta_j) : \quad \bigvee_{k=1}^{r} (a_1 = b'_1) \wedge \cdots \wedge (a_{k-1} = b'_{k-1}) \wedge (a_k \succ b'_k)$$

formalizes the condition for $\Delta_i \succ \Delta'_j$ in lexicographic order. In general, we cannot determine whether the formula $gt(\Delta_i, \Delta_j)$ is true or false, because some of the a_k, b_k can be functions such as x or y. Let E be a consistent conjunction whose conjuncts are expressions of the form $\delta_k = \delta'_k$ or $\delta_k \succ \delta'_k$ for some $\delta_k \in \mathcal{R}$. We say that Δ_i *dominates* Δ_j *under* E, written $\Delta_i \succ_E \Delta_j$, if the implication $E \rightarrow gt(\Delta_i, \Delta_j)$ is valid. Thus, E lists some assumptions about the relations between δ_k and δ'_k under which $gt(\Delta_i, \Delta_j)$ can be evaluated.

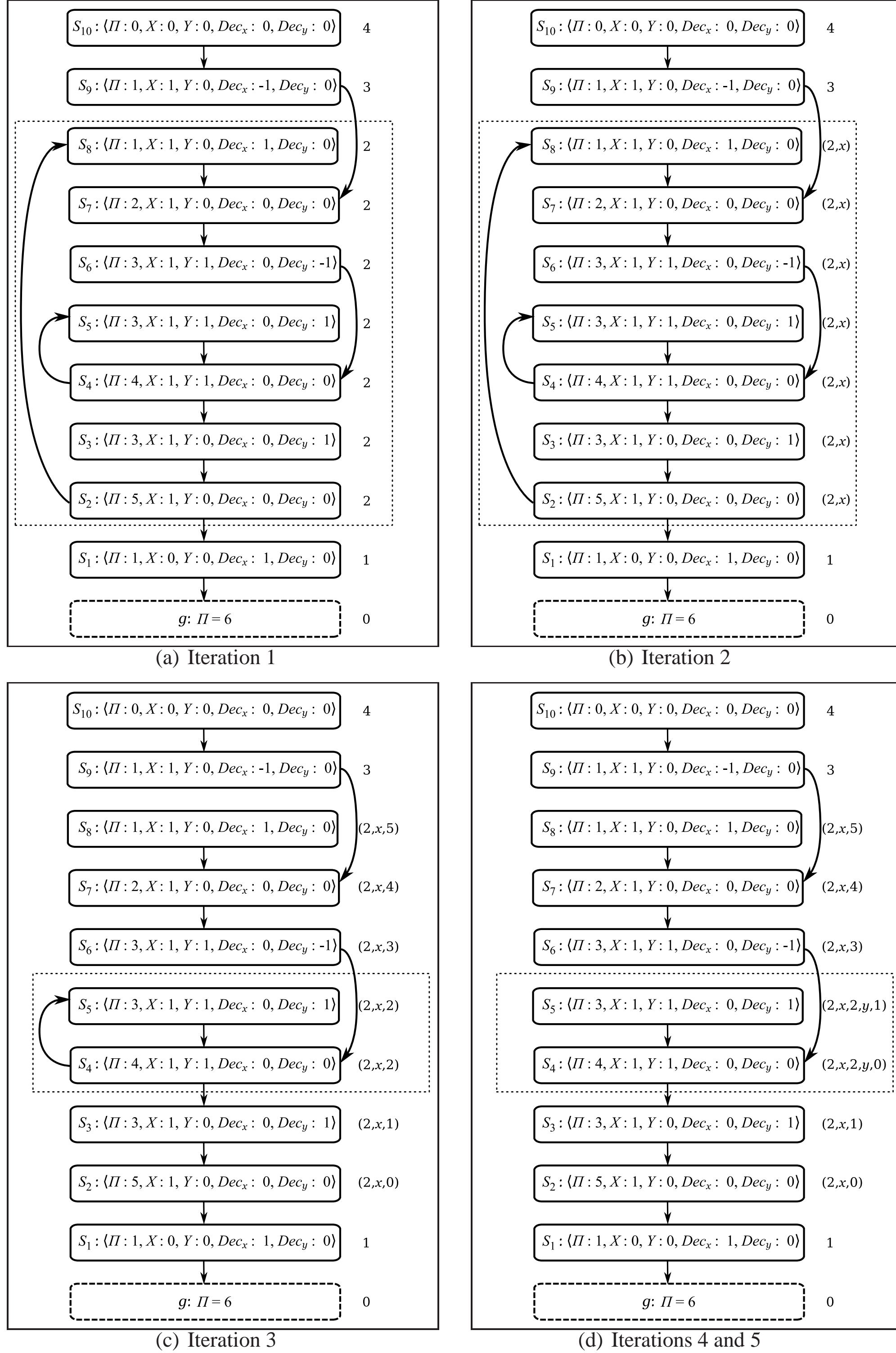


Fig. 11. Progress of procedure RANK-GRAPH over the transition graph of the abstraction of NESTED-LOOPS. The goal summary state is denoted by node g. To the right of each node i is a tuple denoting *Rank*$[i]$ as computed in the relevant iteration of the algorithm.

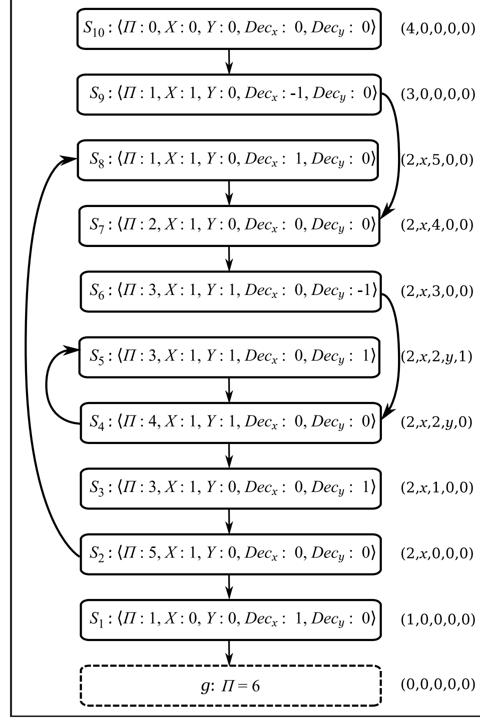


Fig. 12. The end result of RANK-GRAPH, as applied to the abstraction of NESTED-LOOPS

For example, for $\Delta_i : (2, x, 2, y, 0)$, $\Delta_j : (2, x, 2, y, 1)$ and $E : x = x' \wedge y > y'$, Δ_i dominates Δ_j under E, that is, $\Delta_i \succ_E \Delta_j$. A special environment is $E_0 = \bigwedge_{\delta_k \in \mathcal{R}} (\delta_k = \delta'_k)$, which assumes that all ranking components are equal. If Δ_i dominates Δ_j under E_0, we denote this fact by $\Delta_i > \Delta_j$. Each abstract state S induces an environment, denoted $E(S)$, which, for each $\delta_k \in \mathcal{R}$, contains the equality $\delta_k = \delta'_k$ iff $S[\mathit{Dec}_k] = 0$, and contains the inequality $\delta_k \succ \delta'_k$ iff $S[\mathit{Dec}_k] = 1$. We denote the fact that S_i dominates S_j under the environment $E(S)$ by writing $S_i \succ_S S_j$.

For example, consider the abstract states $S_4 : \langle \Pi{:}4,\ X{:}1,\ Y{:}1,\ \mathit{Dec}_x{:}0,\ \mathit{Dec}_y{:}0 \rangle$ and $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ \mathit{Dec}_x{:}0,\ \mathit{Dec}_y{:}1 \rangle$, and their associated ranks $\Delta_4 : (2, x, 2, y, 0)$ and $\Delta_5(2, x, 2, y, 1)$. According to the definition $E(S_5) = (x = x' \wedge y > y')$ and $E(S_4) = (x = x' \wedge y = y')$. It follows that both $\Delta_4 \succ_{S_5} \Delta_5$ and $\Delta_5 \succ_{S_4} \Delta_4$ are true. Following are some important properties of pending graphs and their final rankings:

P1. There is a rank decrease $\Delta_i \succ_{S_j} \Delta_j$ across every edge (S_i, S_j), with associated ranks Δ_i, Δ_j.
P2. If Δ_i and Δ_j are the ranks associated with states S_i and S_j, respectively, then $S_i \neq S_j$ implies $\Delta_i \neq \Delta_j$.
P3. The relation $>$ between ranks is a total order over the set of final ranks computed by Algorithm RANK-GRAPH.
P4. If $\Delta_i \succ_{S_j} \Delta_j$ and $\Delta_j > \Delta_k$, then $\Delta_i \succ_{S_j} \Delta_k$.

node	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Final Ranking
S_{10}	(4)					$(4,0,0,0,0)$
S_9	(3)					$(3,0,0,0,0)$
S_8	(2)	$(2,x)$	$(2,x,5)$			$(2,x,5,0,0)$
S_7	(2)	$(2,x)$	$(2,x,4)$			$(2,x,4,0,0)$
S_6	(2)	$(2,x)$	$(2,x,3)$			$(2,x,3,0,0)$
S_5	(2)	$(2,x)$	$(2,x,2)$	$(2,x,2,y)$	$(2,x,2,y,1)$	$(2,x,2,y,1)$
S_4	(2)	$(2,x)$	$(2,x,2)$	$(2,x,2,y)$	$(2,x,2,y,0)$	$(2,x,2,y,0)$
S_3	(2)	$(2,x)$	$(2,x,1)$			$(2,x,1,0,0)$
S_2	(2)	$(2,x)$	$(2,x,0)$			$(2,x,0,0,0)$
S_1	(1)					$(1,0,0,0,0)$
g	(0)					$(0,0,0,0,0)$

Fig. 13. Iterative Ranking for NESTED-LOOPS

P5. If states S_i and S_j agree on the values of their non-*Dec* variables, then they have the same set of successors.

These properties can be proven by induction on the steps in Algorithm RANK-GRAPH. The induction hypothesis can be phrased as an invariant that includes properties P1 and P2 restricted to the case that the considered S_i and S_j belong to disjoint MSCC's. This invariant is expected to hold whenever we enter step 3 in algorithm RANK-GRAPH.

4.4 Forming an Abstract Verification Diagram

The ranked pending graph still contains too many details. In particular, it assigns different ranks to two abstract states that agree on all non-*Dec* variables. For example, states $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}1\rangle$ and $S_6 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}-1\rangle$ are assigned different ranks in the ranked graph of Fig. 12. In the next step, we group together all abstract states that agree on the values of all non-*Dec* variables.

To eliminate this redundant distinction, we form an *abstract verification diagram* in the spirit of [10]. This is a directed graph whose nodes are labeled with assertions $\Phi_0, \Phi_1, \ldots, \Phi_m$ and are also ranked by a well-founded ranking. Such diagrams are often used to provide a succinct representation of the auxiliary constructs needed for a proof rule such as SEQUENTIAL-RESPONSE.

The abstract verification diagram is constructed as follows:

1. For each set of states in the ranked pending graph that agree on the values of their non-*Dec* variables we construct a node and label it by an assertion Φ. Assertion Φ is a conjunction that specifies the values of all non-*Dec* variables.
   Thus, the set consisting of state $S_5 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}1\rangle$ and state $S_6 : \langle \Pi{:}3,\ X{:}1,\ Y{:}1,\ Dec_x{:}0,\ Dec_y{:}-1\rangle$ will be represented by a single node labeled by the assertion $\Phi : \Pi = 3 \wedge X = 1 \wedge Y = 1$.
   For simplicity, we write $S \in \Phi$ as synonymous to $S \models \Phi$.
2. We draw an edge from the node (labeled by the assertion) Φ_i to node Φ_j, whenever there are states $S_i \in \Phi_i$ and $S_j \in \Phi_j$ such that S_i is connected to S_j in the ranked pending graph.

3. A node Φ is ranked by a rank Δ which is the $>$-minimum among the ranks associated with the states that are grouped in Φ.
   Thus, the rank assigned to node $\Phi : \Pi = 3 \wedge X = 1 \wedge Y = 1$, which has been obtained by grouping together states S_5 and S_6 with associated ranks $(2, x, 2, y, 1)$ and $(2, x, 3)$, is $(2, x, 2, y, 1)$, which is the smaller of the two ranks.

In Fig. 14(a) we present the abstract verification diagram obtained for program NESTED-LOOPS.
A property supporting the correctness of the construction is

P6. If Φ_i is connected to Φ_j in the verification diagram and $S_j \in \Phi_j$, then $\Delta_i \succ_{S_j} \Delta_j$, where Δ_i, Δ_j are the ranks associated with Φ_i, Φ_j, respectively.

The proof of this property is based on the fact that the rank of an assertion Φ is defined as the $>$-minimum over the ranks of all states included in Φ, and on the "semi-transitivity" property P4.

4.5 Obtaining the Concrete Helpful Assertions

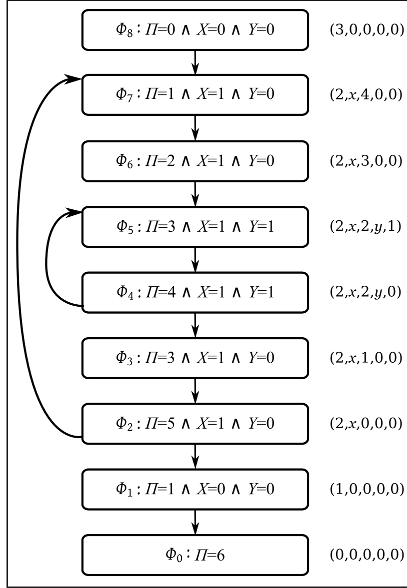
As the last step in the extraction of the auxiliary constructs needed by rule SEQUENTIAL-RESPONSE, we compute the concrete helpful assertions $\varphi_0, \ldots, \varphi_m$. These are obtained simply by concretization of the abstract assertions $\Phi_0, \ldots, \Phi_m$. That is, for each $i = 0, \ldots, m$, we let $\varphi_i = \alpha^{-1}(\Phi_i)$. Thus, for program NESTED-LOOPS, we obtain the helpful assertions presented in the table of Fig. 14(b).

A property that leads to the overall correctness of the construction is given by:

P7. If concrete states s_i, s_j satisfy $s_i \models \varphi_i$ and $s_j \models \varphi_j$, and s_j is a $\mathcal{D}$-successor of s_i, then $s_i[\Delta_i] \succ s_j[\Delta_j]$.

We will sketch the proof of this property. Since s_j is a $\mathcal{D}$-successor of s_i, there exist $\widetilde{s}_i, \widetilde{s}_j$, states of the augmented system $\mathcal{D}+\mathcal{R}$, such that $\widetilde{s}_i \models \varphi_i, \widetilde{s}_j \models \varphi_j$, and $\widetilde{s}_j$ is a $\mathcal{D}+\mathcal{R}$-successor of $\widetilde{s}_i$. Abstracting these states by α, we obtain abstract states S_i, S_j, such that $S_i \models \Phi_i, S_j \models \Phi_j$, and S_j is a $\mathcal{D}^{\mathcal{R},\alpha}$-successor of S_i (and hence Φ_i is connected to Φ_j in the abstract verification diagram). By property P6, it follows that $\Delta_i \succ_{S_j} \Delta_j$. The definition of $\succ_{S_j}$ implies that $\Delta_i \succ \Delta_j$ under the assumptions that the ranking components δ_k decrease or increase as determined by the values of the *Dec*$_k$ variables in state S_j. However, these values are the same as they are in $\widetilde{s}_j$ and therefore, faithfully represent whether δ_k decrease or increase on the transition from s_i to s_j. It therefore follows that $s_i[\Delta_i] \succ s_j[\Delta_j]$.
We can now summarize the complete algorithm for extraction of the auxiliary constructs necessary for a successful application of rule SEQUENTIAL-RESPONSE.

1. Construct the pending graph and apply Algorithm RANK-GRAPH in order to assign well-founded ranks to the nodes in the graph.
2. Construct an abstract verification diagram by abstracting away the *Dec*-variables.
3. Derive the helpful assertions by concretization of the assertions labeling the nodes in the abstract verification diagrams.



(a) Abstract Verification Diagram for NESTED-LOOPS

Index	φ_i	Δ_i
8	$\pi = 0 \wedge x = 0 \wedge y = 0$	$(3, 0, 0, 0, 0)$
7	$\pi = 1 \wedge x > 0 \wedge y = 0$	$(2, x, 4, 0, 0)$
6	$\pi = 2 \wedge x > 0 \wedge y = 0$	$(2, x, 3, 0, 0)$
5	$\pi = 3 \wedge x > 0 \wedge y > 0$	$(2, x, 2, y, 1)$
4	$\pi = 4 \wedge x > 0 \wedge y > 0$	$(2, x, 2, y, 0)$
3	$\pi = 3 \wedge x > 0 \wedge y = 0$	$(2, x, 1, 0, 0)$
2	$\pi = 5 \wedge x > 0 \wedge y = 0$	$(2, x, 0, 0, 0)$
1	$\pi = 1 \wedge x = 0 \wedge y = 0$	$(1, 0, 0, 0, 0)$
0	$\pi = 6$	$(0, 0, 0, 0, 0)$

(b) Assertions and Rankings for NESTED-LOOPS

Fig. 14. Final, simplified, result of RANK-GRAPH, with the resulting concrete helpful assertions

5 Extracting Deductive Proofs for Concurrent Programs

In the previous Section we showed how to extract a deductive proof of response properties for FDS's derived from sequential programs and, therefore, containing no fairness requirements. In this Section we extend the method to FDS's containing justice requirements that can, therefore, represent the majority of concurrent programs.

Recall that in Subsection 2.1 we stated that, since we focus on sequential programs, we can restrict to systems where idle steps are only allowed from terminal states. Now, when discussing concurrent systems, we lift this restriction. As a matter of fact, we assume that every state has an idle successor. We, however, still restrict our attention to the subclass of *just discrete systems* (JDS), which allow an arbitrary number of justice requirements, but no native compassion requirements. The ranking abstraction method introduces its own compassion requirements into the augmented system prior to abstraction, but we allow no compassion requirements in the original system $\mathcal{D}$.

5.1 A Deductive Rule for Response under Justice

In the case of sequential programs, we could require a well-founded ranking that decreases on *every* non-idle step. This is no longer possible in the case of concurrent programs or in the presence of justice requirements. Here, we partition the space of pending steps into regions characterized by assertions $\varphi_1, \ldots . \varphi_m$ where for each $i = 1, \ldots, m$, $\varphi_i \rightarrow \neg J_i$, so that any step from a φ_i-state that causes J_i to be fulfilled should cause the

ranking to decrease. However, as long as we remain within φ_i, we may take an arbitrary number of steps and the rank need not decrease.

In Fig. 15 we present proof rule RESPONSE which establishes the response property $p \Longrightarrow \Diamond q$ for a JDS $\mathcal{D}$. Premise R1 of the rule requires that any p-state is also an φ_i-state for some $i = 0, \ldots, m$. Premise R2 of the rule requires that any step from a φ_i-state ($i > 0$) either causes the ranking to decrease or preserves the value of Δ_i, provided we stay in the φ_i-region. By premise R3, justice requirement J_i is not satisfied by any φ_i-state. It follows that every infinite run that enters the pending domain without leaving it either causes the ranking to decrease infinitely many times, which is impossible, or stays forever within some φ_i-region from a certain point on. However, in the latter case, justice requirement J_i will be satisfied only finitely many times. It follows that such a run cannot be a computation since it violates the justice requirement associated with J_i. We conclude that no computation can stay contained forever within the domain of pending states. Hence any computation that enters the pending domain must eventually exit, and satisfy q.

Rule RESPONSE
For a well-founded domain $\mathcal{A} : (W, \succ)$,
assertions $p, q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
justice requirements $J_1, \ldots, J_m$,
and ranking functions $\Delta_0, \Delta_1, \ldots, \Delta_m$ where each $\Delta_i : \Sigma \mapsto \mathcal{A}$

R1. $p \Longrightarrow \bigvee_{j=0}^{m} \varphi_j$

For each $i = 1, \ldots, m$,

R2. $\varphi_i \wedge \rho \Longrightarrow (\varphi_i' \wedge \Delta_i = \Delta_i') \vee \bigvee_{j=0}^{m} (\varphi_j' \wedge \Delta_i \succ \Delta_j')$

R3. $\varphi_i \Longrightarrow \neg J_i$

$p \Longrightarrow \Diamond q$

Fig. 15. Deductive rule RESPONSE

5.2 Applying Ranking Abstraction to Concurrent Programs

The method of ranking abstraction can be applied, with no change, to JDS's derived from concurrent programs.

We illustrate this application on program UP-DOWN, presented in Fig. 16, for which we wish to prove the response property $(\pi_1 = 0 \wedge \pi_2 = 0) \Longrightarrow \Diamond(\pi_1 = 4)$, where π_1 and π_2 are the location counters for P_1 and P_2 respectively. To distinguish between the locations of processes P_1 and P_2, we denote them by ℓ_i, and m_j, respectively. We also use the notation at_ℓ_i to denote $\pi_1 = i$, and, similarly, we use at_m_j to denote $\pi_2 = j$. The justice requirements of the programs are given by $\mathcal{J} = \{\neg at_\ell_0, \neg at_\ell_1, \neg at_\ell_2, \neg at_\ell_3, \neg at_m_0\}$. Thus, every statement at location ℓ (i.e., ℓ_i or m_j) is associated with a justice requirement of the form $\neg at_\ell$, guaranteeing that the statement is eventually executed, and execution does not remain stuck at ℓ. Employing the predicate base $\mathcal{P} : \{x > 0, y > 0\}$ and the ranking core $\mathcal{R} : \{\delta_y : y\}$, we obtain the abstraction

$$\alpha: \quad \Pi_1 = \pi_1,\ \Pi_2 = \pi_2,\ X = (x > 0),\ Y = (y > 0),\ Dec_y = dec_y,$$

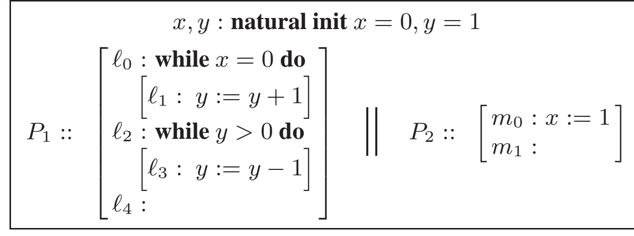


Fig. 16. Program UP-DOWN

Augmenting and abstracting program UP-DOWN, we obtain the abstract program ABSTRACT-UP-DOWN, presented in Fig. 17.

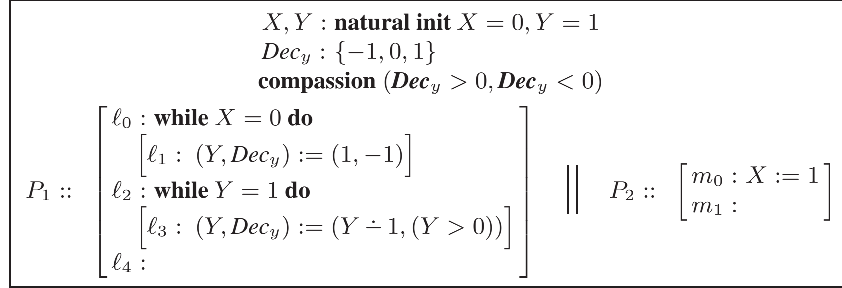


Fig. 17. Program ABSTRACT-UP-DOWN

The program uses the operation $\dot{-}$ which is defined by $Y \dot{-} 1 = \max(Y-1, 0)$. The justice requirements of the abstract program are the same as for the concrete program – $\mathcal{J} = \{\neg at_\ell_0, \neg at_\ell_1, \neg at_\ell_2, \neg at_\ell_3, \neg at_m_0\}$, except that at_ℓ_i and at_m_j are now interpreted as $\Pi_1 = i$ and $\Pi_2 = j$, respectively.

Model checking the abstracted property $\Psi^\alpha : (\Pi_1 = 0 \wedge \Pi_2 = 0) \Longrightarrow \diamondsuit(\Pi_1 = 4)$ over program ABSTRACT-UP-DOWN, we find out that it is valid. We conclude that the concrete program UP-DOWN terminates.

5.3 Extracting a Deductive Proof

We proceed to outline the algorithm by which we can extract the necessary ingredients for a deductive proof by rule RESPONSE from a successful application of the ranking abstraction method. As in the sequential case, the extraction process proceeds in three steps. In the first step we construct the pending graph, and assign ranks to the abstract states belonging to this graph. That step also assigns a helpful justice requirement to each abstract state (as required by rule RESPONSE). The second step constructs an abstract verification diagram which contains an abstracted versions of the helpful assertions. In the third and final step we construct the (concrete) helpful assertions.

Once again, we start by constructing a transition graph $G : \langle N, E \rangle$, which represents the set of pending states plus a goal state. In the just version of the construction, it is important to label the edges by a label that can be viewed either as the transition that leads from one state to the next, or the justice requirement which the transition causes to be satisfied. This correspondence results from the fact that every transition τ in the program can be associated with a justice requirement J_τ that holds iff τ is

disabled. To illustrate this construction, we present in Fig. 18 the labeled transition graph corresponding to the pending states of program ABSTRACT-UP-DOWN. As seen in the diagram, we represent the justice requirements $\neg at_\ell_i$ and $\neg at_m_j$ simply by the locations ℓ_i and m_j, respectively.

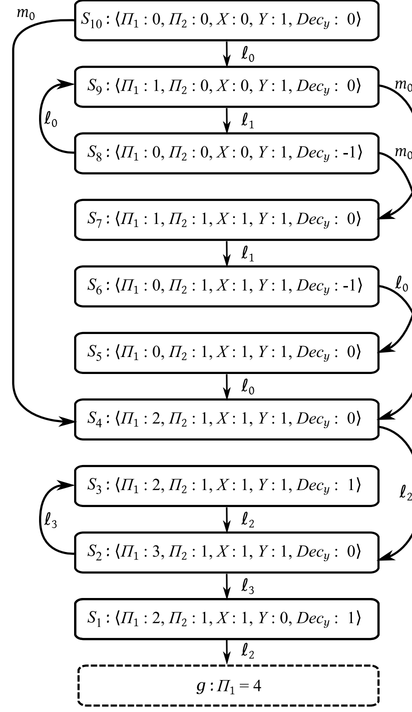


Fig. 18. Pending and goal states for program ABSTRACT-UP-DOWN.

Having constructed the pending transition graph G we proceed to analyze it and determine the ranks associated with the abstract states. A basic process in the algorithm for rank determination is the decomposition of subgraphs into their MSCC's. An MSCC C is said to be *just with respect to justice requirement* J_i if C contains a state satisfying J_i. Component C is defined to be *just* if it is just with respect to all justice requirements. In Fig. 19, we present the algorithm for computing the ranks for a pending graph produced for a JDS.

In the table of Fig. 20, we present the progress of algorithm RANK-JUST-GRAPH when applied to the pending graph of program ABSTRACT-UP-DOWN, which is given in Fig. 18. The last column in the table lists, for each node, the justice requirement identified as helpful for that node. These entries are determined in line 3 of Algorithm RANK-JUST-GRAPH.
In the first iteration, the MSCC decomposition yields the following sorted list:

$$g,\ S_1,\ \{S_2,\ S_3\},\ S_4,\ S_5,\ S_6,\ S_7,\ \{S_8,\ S_9\},\ S_{10}$$

Algorithm RANK-JUST-GRAPH(G)
Input: a graph $G = (N, E)$ representing the pending states for the abstract system.
Output: *Rank*, an array $N \mapsto$ TUPLES
Initially: For every $n \in N$, $Rank(n) = \bot$.
Just-Rank(G):

1. Decompose G into a sorted list of MSCCs $G = C_0, ..., C_k$;
2. For every node $n \in C_i$, append i to $Rank(n)$;
3. For each unjust MSCC C, identify the justice requirement J_i that causes the injustice, and mark J_i as the requirement that is helpful for C.
4. Perform the following for each just MSCC C in the decomposition, excluding node g:
   (a) If for some compassion requirement $(Dec_j > 0, Dec_j < 0)$, C has some nodes with $Dec_j > 0$, and no nodes with $Dec_j < 0$, then append δ_j to $Rank(n)$ of every node $n \in C$; if no such j exists, report "failure" and halt;
   (b) Let D be the subgraph obtained by removing every edge in C leading into a $Dec_j > 0$ node;
   (c) Call Just-Rank(D);

Fig. 19. Procedure RANK-JUST-GRAPH, which constructs a ranking function from the transition graph of a terminating abstract JDS.

Consequently, we assign to nodes $g, S_1, \ldots, S_{10}$ the sequence of ranks:

$$0,\ 1,\ 2,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 7,\ 8$$

Next, we examine each of the components, excluding g. We find that the only just component is $\{S_2, S_3\}$. This is because each of the other components is unjust w.r.t some justice requirement, as shown by

Component	S_1	S_4	S_5	S_6	S_7	$\{S_8,\ S_9\}$	S_{10}
Violates	$\neg at_\ell_2$	$\neg at_\ell_2$	$\neg at_\ell_0$	$\neg at_\ell_0$	$\neg at_\ell_1$	$\neg at_m_0$	$\neg at_m_0$

Component $\{S_2, S_3\}$ is just. Therefore we search for a compassion requirement that is violated by the component. Indeed, we observe that $(Dec_y > 0, Dec_y < 0)$ is violated because State S_3 has a positive value of Dec_y but there is no state assigning a negative value to Dec_y in this component. We therefore augment the ranks of S_2 and S_3 by the ranking element $\delta_y : y$, remove the edges entering S_3, and invoke the procedure Just-Rank with a graph D whose nodes are $\{S_2, S_3\}$ and which has the single edge $(S_3 \rightarrow S_2)$. Decomposing the subgraph D, we obtain the decomposition S_2, S_3. Consequently, in the 3rd (and last) iteration, we append to nodes S_2, S_3 the ranks $0, 1$, respectively.

Note that, once we identify that some components are unjust, we do not process them any further. Note also that, while the sequential version of the ranking computation algorithm always terminates with a graph consisting of singleton components, the just version may leave several components intact, such as $\{S_8,\ S_9\}$.

In Fig. 21(a), we present a ranked version of the pending graph. Edges labeled with the helpful justice requirements are drawn in bold type.

Node	Iteration 1	Iteration 2	Iteration 3	Final Ranking	Helpful Justice Req.
S_{10}	8			$(8,0,0)$	m_0
S_9	7			$(7,0,0)$	m_0
S_8	7			$(7,0,0)$	m_0
S_7	6			$(6,0,0)$	ℓ_1
S_6	5			$(5,0,0)$	ℓ_0
S_5	4			$(4,0,0)$	ℓ_0
S_4	3			$(3,0,0)$	ℓ_2
S_3	2	$(2,y)$	$(2,y,1)$	$(2,y,1)$	ℓ_2
S_2	2	$(2,y)$	$(2,y,0)$	$(2,y,0)$	ℓ_3
S_1	1			$(1,0,0)$	ℓ_2
g	0			$(0,0,0)$	

Fig. 20. Progress of Algorithm RANK-JUST-GRAPH

On successful termination of Algorithm RANK-JUST-GRAPH, we can claim the following properties:

C1. For every two states S_i, S_j which belong to disjoint MSCC's, and such that S_i is connected to S_j, there is a rank decrease $\Delta_i \succ_{S_j} \Delta_j$, where Δ_i, Δ_j are the ranks associated with S_i, S_j, respectively.

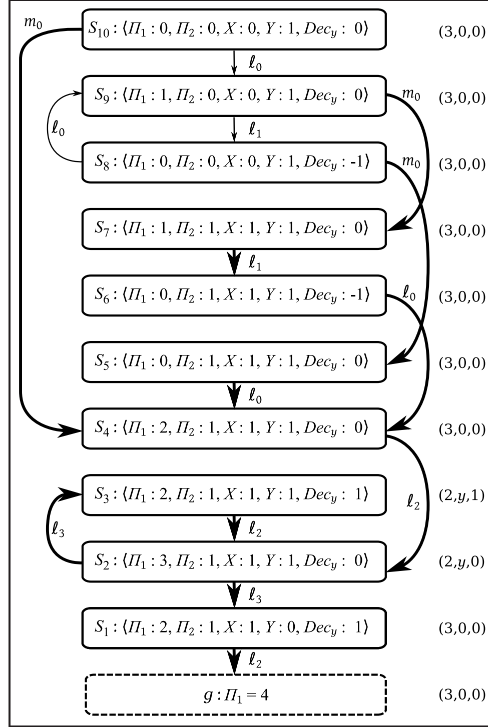
C2. For every two states S_i, S_j and their associated ranks Δ_i, Δ_j, then $\Delta_i = \Delta_j$ iff S_i and S_j belong to the same MSCC.

In addition, properties P3–P5 of Section 4 are also true here.

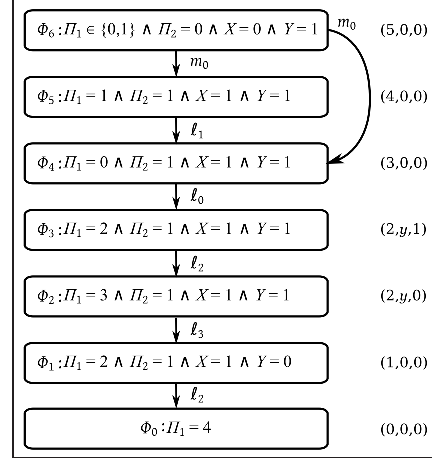
5.4 Forming an Abstract Verification Diagram

In the second step of the extraction process, we form the abstract verification diagram merging together each MSCC of the pending graph into a single assertion.

1. In the first step we merge together abstract states that differ only in their *Dec* variables. This is done by identifying two such abstract states S_i and S_j, retaining the representative with the smaller rank, and redirecting edges previously connecting to the node with higher ranks into the node with the lower rank.
   Thus, in the graph of Fig. 18, we can merge S_{10} into S_8, S_6 into S_5, and S_4 into S_3.
2. Next, we construct for each MSCC of the graph resulting from the previous step a single assertion Φ which is a disjunction of the valuations of the non-*Dec* variables of all the states contained in the MSCC.
   Thus, the assertion corresponding to the MSCC which contains the two states $S_8 : \langle \Pi_1{:}0,\ \Pi_2{:}0,\ X{:}0,\ Y{:}1,\ Dec_y{:}-1\rangle$ and $S_9 : \langle \Pi_1{:}1,\ \Pi_2{:}0,\ X{:}0,\ Y{:}1,\ Dec_y{:}0\rangle$ is $\Phi : \Pi_1 \in \{0,1\} \wedge \Pi_2 = 0 \wedge X = 0 \wedge Y = 1$.
3. We draw an edge labeled by J connecting node Φ_i to node Φ_j, whenever there are states $S_i \in \Phi_i$ and $S_j \in \Phi_j$ such that S_i is connected to S_j by a J-labeled edge in the ranked pending graph.



(a) A ranked version of the Pending graph for program ABSTRACT-UP-DOWN.



(b) Abstract verification diagram for program ABSTRACT-UP-DOWN.

Fig. 21. The ranking produced by RANK-JUST-GRAPH on the Pending graph of program ABSTRACT-UP-DOWN, and the subsequent abstract verification diagram.

4. A node Φ is ranked by a rank Δ which is the common rank associated with the states that belong to the MSCC.

In Fig. 21(b), we present the abstract verification diagram obtained from the graph of Fig. 21(a).
An important property of the abstract verification diagram is the following:

P6. If Φ_i is connected to Φ_j in the verification diagram and $S_j \in \Phi_j$, then $\Delta_i \succ_{S_j} \Delta_j$, where Δ_i, Δ_j are the ranks associated with Φ_i, Φ_j, respectively.

5.5 Obtaining the Concrete Helpful Assertions

As the last step in the extraction of the auxiliary constructs needed by rule RESPONSE, we compute the concrete helpful assertions $\varphi_0, \ldots, \varphi_m$. As in the sequential case, these are obtained simply by concretization of the abstract assertions $\Phi_0, \ldots, \Phi_m$. In the table presented in Fig. 22, we present the auxiliary constructs extracted for program UP-DOWN.

As in the sequential case, the following property leads to the overall correctness of the construction:

P7. If concrete states s_i, s_j satisfy $s_i \models \varphi_i$ and $s_j \models \varphi_j$, and s_j is a $\mathcal{D}$-successor of s_j, then $s_i[\Delta_i] \succ s_j[\Delta_j]$.

i	φ_i	Δ_i	J_i
6	$at_\ell_{0,1} \wedge at_m_0 \wedge x = 0 \wedge y > 0$	$(5, 0, 0)$	$\neg at_m_0$
5	$at_\ell_1 \wedge at_m_1 \wedge x > 0 \wedge y > 0$	$(4, 0, 0)$	$\neg at_\ell_1$
4	$at_\ell_0 \wedge at_m_1 \wedge x > 0 \wedge y > 0$	$(3, 0, 0)$	$\neg at_\ell_0$
3	$at_\ell_2 \wedge at_m_1 \wedge x > 0 \wedge y > 0$	$(2, y, 1)$	$\neg at_\ell_2$
2	$at_\ell_3 \wedge at_m_1 \wedge x > 0 \wedge y > 0$	$(2, y, 0)$	$\neg at_\ell_3$
1	$at_\ell_2 \wedge at_m_1 \wedge x > 0 \wedge Y = 0$	$(1, 0, 0)$	$\neg at_\ell_2$
0	at_ℓ_4		

Fig. 22. Extracted auxiliary constructs for program UP-DOWN

6 Conclusion

The work in this paper is a direct continuation of [2], where a framework was presented for automatic computation of predicate and ranking abstractions, with a specific application to the domain of unbounded pointer structures (aka Shape Analysis). That framework requires all predicates and component ranking functions to be provided by the user. Here we have extended it with dual means of refinement for both types of abstraction.

We have shown two heuristics for synthesizing component ranking functions, one for a linear domain and another for a domain of unbounded pointer structures. These have been surprisingly effective in proving termination of a number of example programs. In the near future we plan to explore richer heuristics in the domain of shape analysis.

In the last two sections we have shown how a deductive proof of a response property can be extracted from a successful application of the ranking-abstraction method. First, we consider the simpler case of systems with no fairness requirements, which correspond to sequential programs. We then indicated how the extraction process can be applied to systems with justice requirements, which can be used to model concurrent programs.

References

1. A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.

2. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VM-CAI'2005: Verification, Model Checking, and Abstraction Interpretation*, pages 164–180, 2005.
3. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, 2002.
4. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
6. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Static Analysis Symposium*, pages 87–101, 2005.
7. D. Dams, R. Gerth, and O. Grumberg. A heuristic for the automatic generation of ranking functions. In G. Gopalakrishnan, editor, *Workshop on Advances in Verification*, pages 1–8, 2000.
8. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
9. O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.
10. Z. Manna and A. Pnueli. Temporal verification diagrams. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Comp. Sci.*, pages 726–765. Springer-Verlag, 1994.
11. A. Podelski and A. Rybalchenko. Software model checking of liveness properties via transition invariants. Research Report MPI-I-2003-2-004, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, December 2003.
12. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.