Improving Communicating Pushdown System Model Checking

Nicholas Kidd[1], Thomas Reps[1,2], and Tayssir Touili[3]

[1] University of Wisconsin; Madison, WI; USA. {kidd,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.
[3] LIAFA, CNRS & Université Paris Diderot, Paris, France. touili@liafa.jussieu.fr

Abstract. Communicating pushdown systems (CPDSs) are a formalism for modeling the behaviors of concurrent systems. They have been used to model concurrent C and Java programs. Once a concurrent program has been modeled as a CPDS, a reachability query is given to a CPDS model checker to determine if the property of interest is satisfied by the program model.

Our CPDS model checker implements a semi-decision procedure for answering a reachability query. For a given CPDS, the semi-decision procedure defines increasingly more-precise approximations of the reachability relation of a CPDS to determine if the query holds.

The focus of this paper is on improving the semi-decision procedure employed by the CPDS model checker. We define a new semi-decision procedure that is more precise and more efficient, and explore three new abstraction-refinement policies. When analyzing CPDSs generated from Java programs, the new policies produced median speedups of 2.4, 1.2, and 2.5, and maximum speedups of 5.4, 5.3, and 7.6.

1 Introduction

Communicating pushdown systems (CPDSs) [1, 2] are a formalism for modeling the behaviors of concurrent systems. CPDS-model checking has been used to find known bugs in concurrent C and Java programs [2, 3], and to discover an unknown bug in a model of a Windows NT Bluetooth driver that was previously thought to be correct [2].

An important aspect of CPDSs is that both single-location data-consistency errors, i.e., data races, and the often ignored but equally important *multi-location* data-consistency errors can be encoded as a reachability query for the CPDS model checker [3].[4] Indeed, a recent study of real-world concurrency-bug characteristics by Lu et al. [4] found that 34% of the non-deadlock concurrency bugs involved multiple shared-memory locations. (For an example, see [4, Fig. 6].)

Our CPDS model checker [2] implements a semi-decision procedure (SDP) that, upon termination, answers a reachability query for a given CPDS. The SDP proceeds by computing a series of increasingly precise approximations of the reachability relation of a CPDS, where a more precise approximation is computed when it is determined that the current abstraction is too coarse to answer the query.

[4] For the rest of this paper, we refer to the CPDS model checker as simply the model checker.

The focus of this paper is improving the efficiency of the algorithm that the model checker employs. We begin by relating CPDS model checking to the problem of determining the emptiness of the intersection of a set of context-free languages (§2). This shows why the CPDS model checker implements only an SDP, and briefly describes some of the abstractions that have been explored.

To explain the improvements to the model checker's SDP, we must first provide the details of pushdown systems (PDSs), CPDSs, and CPDS model checking (§3). The straightforward approach to encoding synchronization actions in CPDSs [1] causes the model checker to exhaust all resources for the benchmarks reported on in §6. We address this issue in §4 with the definition of the *Succinct SDP*, which is more precise and exponentially more succinct. than the straightforward encoding. An additional benefit of the *Succinct SDP* is that it suggests new kinds of abstraction-refinement policies for use in the model checker. We define two new policies:

1. The *Multi-step* refinement policy (§5.1) is to CPDS model checking as *counter-example guided abstraction refinement* [5, 6] is to predicate-abstraction model checking: the next more-precise abstraction is determined by using information present in the current level of abstraction.
2. The *Individual* refinement policy (§5.2) is to CPDS model checking as *lazy abstraction* [7] is to predicate-abstraction model checking: abstractions of CPDS components are refined only as needed.

We then show that these new policies are easily combined to form a third new policy, *Individual Multi-step* (§5.3). Our experimental evaluation (§6) shows that the new abstraction-refinement polices produce median speedups of 2.4, 1.2, and 2.5, and maximum speedups of 5.4, 5.3, and 7.6 when answering CPDS reachability queries generated from Java programs. The comparison is with respect to the *Succinct SDP* because reachability analysis for each query fails with the encoding from [1]. This paper makes the following contributions:

- It defines the *Succinct SDP*, which uses an encoding that is more precise and exponentially more succinct than the straightforward encoding from [1]. Using the *Succinct SDP*, the CPDS model checker is able to analyze models of real Java programs.
- It introduces three abstraction-refinement policies: *Multi-step*, *Individual*, and *Individual Multi-step*, that give rise to three new SDPs.
- It presents an experimental evaluation of the three new policies on CPDSs that model concurrent Java programs from the ConTest benchmark suite [8], and CPDSs for the three Bluetooth models reported on in [2]. On ConTest, the three policies resulted in median speedups of 2.4, 1.2, and 2.5, respectively, and maximum speedups of 5.4, 5.3, and 7.6, respectively.

2 Overview

A CPDS can be viewed as a set of context-free languages (CFLs), and a CPDS reachability query is tantamount to determining the emptiness of the intersection of the CFLs

in the set. This problem is known to be undecidable in general, and thus the model checker implements only a semi-decision procedure (SDP).

Given a set of CFLs $L_1, \ldots, L_n$, the SDP uses abstraction to define a regular over-approximation R_i for each CFL L_i, $1 \leq i \leq n$, and approximates the intersection result $L = \bigcap_{i=1}^{n} L_i$ by $R = \bigcap_{i=n}^{n} R_i$. The abstractions employed use a form of bounded precision. Namely, for a bound k, the language L_i is divided into two sets: (1) words of length less than k; and (2) words of length greater than or equal to k. The first set is referred to as the *concrete set* because it can be modeled precisely, i.e., no abstraction is required. Likewise, the second set is referred to as the *abstract set* because abstraction of this set is required to ensure decidability of the emptiness of intersection.

Because each R_i, $1 \leq i \leq n$, is an over-approximation of the corresponding CFL L_i, R is an over-approximation of L; consequently, if $R = \emptyset$ then $L = \emptyset$. The key to the SDP is that if $R \neq \emptyset$, one can determine if R contains a concrete word w from L (i.e., $|w| < k$). If no such w exists, then the abstractions must be refined, which amounts to increasing the precision bound k. Thus, the SDP uses the succession of approximations (indicated by $k, k', k'', \ldots$) $\bigcap_{i=1}^{n} R_i^k, \bigcap_{i=1}^{n} R_i^{k'}, \bigcap_{i=1}^{n} R_i^{k''}$, and so on, to determine if the actual intersection $L = \bigcap_{i=1}^{n} L_i$ is empty.

To define a regular over-approximation of the abstract set, various abstractions have been considered: The *prefix abstraction* [2] precisely models the prefix of length k of each abstract word, but loses precision by allowing any number of symbols to follow the prefix. The *suffix abstraction* [2] precisely models the suffix of length k of each abstract word, but loses precision by allowing any number of symbols to precede the suffix. The *bifix* abstraction combines the prefix and suffix abstractions so that abstract words are constrained on each end, but loses precision by allowing any number of symbols to come in between. Finally, more precise abstractions for the parts of words that lie beyond the k-bounded threshold can be used (cf. Nederhof's survey [9]). For the rest of this paper, we will only consider the prefix abstraction; however, the following improvements and new SDPs are applicable to each of the possible abstractions just described.

3 Communicating Pushdown Systems

This section presents the definitions for PDSs and CPDSs, and shows how CPDSs are used to model concurrent systems that use rendezvous-style synchronization. This is followed up by showing how to relax the synchronization primitive to k-wise synchronization, and how the prefix abstraction is used in the SDP that (attempts to) answer a reachability query for a given CPDS.

Pushdown systems are a formalism to describe *pure sequential programs* [10, 11]. They have the property that, for recursive programs, the potentially infinite set of program configurations can be represented symbolically using regular languages [12, 13].

Definition 1. *A* pushdown system *(PDS) is a tuple $\mathcal{P} = (P, \Gamma, \mathsf{Act}, \Delta, c_0)$, where P is a finite set of states; Γ is a finite set of stack symbols; Act is a finite set of actions; Δ is a finite set of transition rules of the form $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', u' \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, $a \in \mathsf{Act}$, and $u' \in \Gamma^*$; and c_0 is the initial configuration of $\mathcal{P}$. A* configuration *c of $\mathcal{P}$ is*

a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A set of configurations C is regular if for each $p \in P$, the language $\{u \in \Gamma^* \mid \langle p, u\rangle \in C\}$ is regular.

Given a PDS $\mathcal{P} = (P, \Gamma, \mathsf{Act}, \Delta, c_0)$, we define for each $a \in \mathsf{Act}$ the transition relation $\Rightarrow^a$ between configurations of $\mathcal{P}$ as follows: if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', u'\rangle \in \Delta$, then $\langle p, \gamma u\rangle \Rightarrow^a \langle p', u'u\rangle$ for every $u \in \Gamma^*$.

Definition 2. *For a PDS $\mathcal{P}$ and a regular set of configurations C, the language of $\mathcal{P}$ with respect to C, denoted by $\mathsf{Lang}(\mathcal{P}, C)$, is defined as: $\{w = a_1 \cdots a_n \in Act^* \mid \exists c \in C \,.\, c_0 \Rightarrow^{a_1} \ldots \Rightarrow^{a_n} c\}$. It is well-known that $\mathsf{Lang}(\mathcal{P}, C)$ is a context-free language.*

Example 1. Let $\mathcal{P}_{\text{ex1}} = (\{p_a, p_b\}, \{\gamma, \bot\}, \{a, b\}, \Delta_{\text{ex1}}, \langle p_a, \bot\rangle)$ be a PDS where Δ_{ex1} is defined as follows:

$$\big\{\langle p_a, \bot\rangle \overset{a}{\hookrightarrow} \langle p_a, \gamma\bot\rangle, \langle p_a, \gamma\rangle \overset{a}{\hookrightarrow} \langle p_a, \gamma\gamma\rangle, \langle p_a, \gamma\rangle \overset{b}{\hookrightarrow} \langle p_b, \epsilon\rangle, \langle p_b, \gamma\rangle \overset{b}{\hookrightarrow} \langle p_b, \epsilon\rangle\big\}.$$

The language $\mathsf{Lang}(\mathcal{P}_{\text{ex1}}, \{\langle p_b, \bot\rangle\})$ is equal to $\{a^n b^n \mid n \geq 1\}$.

Modeling Concurrent Systems with PDSs: Recently, compositions of pushdown systems called *communicating pushdown systems* have been used to model *concurrent* recursive programs [1, 2].

Definition 3. *A* communicating pushdown system *(CPDS) is a tuple $CP = (\mathcal{P}_1, \ldots, \mathcal{P}_n, Act)$ of pushdown systems, where $Act = \bigcup_{i=1}^n Act_i$ is the union of the individual action sets of the PDSs (Act_i is the set of actions of $\mathcal{P}_i$). There is a special action τ that belongs to all the sets $Act_i, 1 \leq i \leq n$, such that for all $a \in Act : \tau \cdot a = a = a \cdot \tau$. (Each τ action represents an internal action of a process, while the other actions correspond to synchronization actions.)*

For a CPDS *CP*, a *global configuration* is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$, and the initial global configuration $g_0 = (c_0^1, \ldots, c_0^n)$ consists of the initial configurations of the individual PDSs. The relation $\Rightarrow^a$ is extended to global configurations as follows:

- $(c_1, \ldots, c_n) \Rightarrow^\tau (c_1', \ldots, c_n')$ if there is an index $1 \leq i \leq n$ such that $c_i \Rightarrow^\tau c_i'$ and $c_j' = c_j$ for every $j \neq i$;
- $(c_1, \ldots, c_n) \Rightarrow^a (c_1', \ldots, c_n')$ if for every i, $1 \leq i \leq n$, $c_i \Rightarrow^a c_i'$ if $a \in \mathsf{Act}_i$ and $c_i' = c_i$ if $a \notin \mathsf{Act}_i$: all processes for which a is an action synchronize on a and move simultaneously.

A set G of global configurations is regular if it can be represented as a tuple $(C_1, \ldots, C_n)$ of regular sets of configurations of the individual PDSs, i.e., $G = \big\{(c_1, \ldots, c_n) \mid c_1 \in C_1, \ldots, c_n \in C_n\big\}$.

From now on, we fix a CPDS $CP = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathsf{Act})$, where for every $i, 1 \leq i \leq n$, $\mathcal{P}_i = (P_i, \Gamma_i, \mathsf{Act}_i, \Delta_i, c_0^i)$, and a regular set of global configurations $G = (C_1, \ldots, C_n)$.

Definition 4. *The language of CP with respect to G, denoted by $\mathsf{Lang}(CP, G)$, is defined as: $\{w = a_1 \cdots a_n \in Act^* \mid \exists g \in G . g_0 \Rightarrow^{a_1} \ldots \Rightarrow^{a_n} g\}$.*

A CPDS is a natural model of concurrent programs where processes synchronize via rendezvous. The special transition relation $\Rightarrow^{\tau}$ models a particular process of a concurrent program performing a local transition. A transition relation $\Rightarrow^{a}$ models all processes for which a is an action synchronizing on a. It is easy to see that if all of the PDSs have the same set of actions, i.e., if for every $1 \leq i \leq n$, $\mathsf{Act}_i = \mathsf{Act}$, then the following holds:

$$\mathsf{Lang}(CP, G) = \bigcap_{i=1}^{n} \mathsf{Lang}(\mathcal{P}_i, C_i). \tag{1}$$

Modeling k-wise synchronization: When using CPDSs to model real systems, it is often the case that not all processes synchronize on every action. For example, if the program model uses pairwise synchronization, each action a would be a member of exactly two action sets Act_i and Act_j, $i \neq j$. However, for Eqn. (1) to hold, all PDSs must have the same set of actions. Thus, we need to insert everywhere in the paths of $\mathcal{P}_i$ labels that correspond to the synchronization actions that are not in Act_i, but that the other PDSs can perform. In [1], this encoding is formalized via the *shuffle* operation. For $a, b \in \mathsf{Act}$ and $u, v \in \mathsf{Act}^*$, the shuffle operation $\sqcup\!\sqcup$ on words is defined as follows:

$$\begin{aligned} u \sqcup\!\sqcup \epsilon &= \{u\} = \epsilon \sqcup\!\sqcup u \\ au \sqcup\!\sqcup bv &= \big(\{a\} \cdot (u \sqcup\!\sqcup bv)\big) \cup \big(\{b\} \cdot (au \sqcup\!\sqcup v)\big), \end{aligned}$$

and for two languages L and L', $L \sqcup\!\sqcup L' = \{u \sqcup\!\sqcup v \mid u \in L, v \in L'\}$. Using the shuffle operation, we redefine the language L_i as follows:

$$L_i = \mathsf{Lang}(\mathcal{P}_i, C_i) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Act}_i)^*. \tag{2}$$

Bouajjani et al. [1] show that Eqn. (1) is then extended as follows:

$$\mathsf{Lang}(CP, G) = \bigcap_{i=1}^{n} L_i. \tag{3}$$

They also showed that relaxing the synchronization model has no effect on the expressive power of a CPDS—one can simply add "self-loops" to the rule set of each PDS to account for unused actions. That is, define a CPDS $CP' = (\mathcal{P}'_1, \ldots, \mathcal{P}'_n)$, where for $1 \leq i \leq n$, $\mathcal{P}'_i = (P_i, \Gamma_i, \mathsf{Act}, \Delta'_i, c_0^i)$ is $\mathcal{P}_i$ with the rule set Δ_i augmented as follows:

$$\Delta'_i = \Delta_i \cup \{\langle p, \gamma\rangle \xhookrightarrow{a} \langle p, \gamma\rangle \mid p \in P_i, \gamma \in \Gamma_i, a \in (\mathsf{Act} \setminus \mathsf{Act}_i)\}.$$

Example 2. Let $\mathcal{P}_{\text{ex2}} = (\{p_a, p_b\}, \{\gamma, \bot\}, \{a, b, c\}, \Delta_{\text{ex2}}, \langle p_a, \bot\rangle)$ be a PDS where $\Delta_{\text{ex2}} = \Delta_{\text{ex1}} \cup \big\{\langle p, x\rangle \xhookrightarrow{c} \langle p, x\rangle \mid p \in \{p_a, p_b\}, x \in \{\gamma, \bot\}\big\}$—$\mathcal{P}_{\text{ex2}}$ is $\mathcal{P}_{\text{ex1}}$ with an unused action c and PDS rules added that explicitly account for it. The language $\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \bot\rangle\})$ is $\{(c^*a)^n(c^*b)^n c^* \mid n \geq 1\}$.

The language $\mathsf{Lang}(CP, G)$ defined by Eqn. (3) is equivalent to the language $\mathsf{Lang}(CP', G)$ defined by Eqn. (1). That is, the following holds:

$$\mathsf{Lang}(CP, G) = \bigcap_{i=1}^{n} L_i = \bigcap_{i=1}^{n} \mathsf{Lang}(\mathcal{P}'_i, C_i) = \mathsf{Lang}(CP', G).$$

Reachability Analysis of CPDSs [2]: The goal of CPDS model checking is to determine if G is reachable in *CP*. Because G is reachable iff $\mathsf{Lang}(CP, G) \neq \emptyset$, and because determining the emptiness of the intersection of a set of CFLs is known to be undecidable, this problem is also undecidable.

As discussed in §2, decidability is retained via a k-bounded abstraction that defines a regular over-approximation for the language of each PDS. We now formally define the prefix abstraction.

Given a bound k and a language $L \subseteq \mathsf{Act}^*$, the *prefix abstraction* $\alpha_k(L)$ is the language: $\{w \mid w \in L \wedge |w| < k\} \cup \{\lfloor w \rfloor^k w' \mid w \in L \wedge |w| \geq k \wedge w' \in \mathsf{Act}^*\}$, where for a word w and a bound k, $\lfloor w \rfloor^k$ denotes the prefix of w of length k.

Example 3. Given $\mathcal{P}_{\text{ex2}}$ from Ex. 2 and bound $k = 3$, $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \bot\rangle\})\big)$ computes the language $\big\{ab\big\} \cup \big\{aaaw,\ aabw,\ aacw,\ abcw,\ acaw,\ acbw,\ accw,\ caaw,\ cabw,\ cacw,\ ccaw,\ cccw \mid w \in \mathsf{Act}^*\big\}$.

Prefix abstractions, likewise suffix and bifix abstractions, can be represented by a finite set consisting of two types of words: (1) *concrete* words are words in the concrete set whose length is less than k, and they are represented exactly; and (2) *abstract* words are prefixes of length k that summarize an infinite set of words from the abstract set. The set is finite because there are only a finite number of words w such that $|w| \leq k$.

Example 4. The language $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \bot\rangle\})\big)$ from Ex. 3 can be represented by the finite set $\big\{ab, aaa, aab, aac, abc, aca, acb, acc, caa, cab, cac, cca, ccc\big\}$. (All of the words except "ab" are abstract words.)

For *CP* and G and using the prefix abstraction, the model checker sets the initial bound as $k = 1$, and defines $R = \bigcap_{i=1}^{n} \alpha_k(L_i)$. If $R = \emptyset$, then G is not reachable. Otherwise, let w be a minimal length word in R. If $|w| < k$, then G is reachable, else k is incremented and the process repeats.

4 Improved Reachability Analysis of CPDSs

The CPDS reachability analysis described in §3 is inefficient because of the need to account for the right operand of the shuffle term in Eqn. (2). That is, explicitly adding rules to PDS $\mathcal{P}_i$ to account for the unused actions (i.e., $\mathsf{Act} \backslash \mathsf{Act}_i$) causes (i) the abstraction to lose more precision than necessary, and (ii) answer sets to be possibly of exponentially greater size than with the technique described in this section. In fact, when only using $\alpha_k(L_i)$ to approximate the language of $\mathcal{P}_i$, our model checker exhausted all resources on even the simplest of queries. For example, using the improved $\beta_k(L_i)$ abstraction defined below, the model checker took only 2.76 seconds to determine reachability for a Bluetooth model (see §6), whereas using the abstraction $\alpha_k(L_i)$, it ran out of memory on a dual-core Xeon processor with 4 GB of memory.

Observe that for the prefix abstraction $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \epsilon\rangle\}\big)$, all of the regularity of the unused action c, i.e., the c^* from $\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \epsilon\rangle\}\big)$ in Ex. 2, has been lost. In addition, without the unused action c as in Ex. 1, the finite set that represents $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex1}}, \{\langle p_b, \epsilon\rangle\}\big)$ is merely $\big\{ab, aaa, aab\big\}$. In general, explicitly modifying $\mathcal{P}_i$ to account for unused actions, as in Ex. 2, causes the set that represents

$\alpha_k(L_i)$ to be exponentially larger in the size of $\mathsf{Act} \setminus \mathsf{Act}_i$ (e.g., aaa blows up to $\{aaa, aac, aca, acc, caa, cac, cca, ccc\}$). We avoid this inefficiency *and* define a more precise abstraction, β_k, by leveraging the fact that we can use the shuffle operation to account for unused actions.

Recall that for a PDS $\mathcal{P}_i$, the *concrete* language of interest is L_i defined by Eqn. (2). Instead of directly applying the prefix abstraction to L_i (i.e., $\alpha_k(L_i)$), we gain precision by pulling the shuffle operation *outside of* the approximation:

$$\beta_k(L_i) = \alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Act}_i)^*.$$

It is easy to see that $\beta_k(L_i)$ is an over-approximation of L_i, and that incrementing k obtains a more precise over-approximation, i.e., for every i, $L_i \subseteq \beta_{k+1}(L_i) \subseteq \beta_k(L_i)$.

The β_k abstraction avoids the exponential increase that occurs when additional PDS rules are introduced to account for unused actions. Also, β_k provides a *more precise* over-approximation of L_i because the following holds:

$$\beta_k(L_i) = \alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C)\big) \sqcup\!\sqcup (\mathsf{Act}\setminus\mathsf{Act}_i)^* \subseteq \alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C) \sqcup\!\sqcup (\mathsf{Act}\setminus\mathsf{Act}_i)^*\big) = \alpha_k(L_i).$$

Moreover, performing the shuffle *after* computing the prefix abstraction is easy. Let $\mathcal{A}_i = (Q_i, \Sigma_i, \delta_i, q_0, F)$ be the automaton (defined in the usual way) that accepts the language $\alpha_k\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big)$. The shuffle operation $\mathsf{Lang}(\mathcal{A}_i) \sqcup\!\sqcup (\mathsf{Act} \setminus \mathsf{Act}_i)^*$ is performed directly on $\mathcal{A}_i$ by augmenting the transition relation δ_i with the following set of additional transitions: $\{(q, a, q) \mid q \in Q_i \wedge a \in (\mathsf{Act} \setminus \mathsf{Act}_i)\}$.

Example 5. Let $\mathsf{Act} = \{a, b, c\}$. The language $\beta_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \bot\rangle\})\big)$ is $\{c^*ac^*bc^*\} \cup \{c^*ac^*aw \mid w \in \mathsf{Act}^*\}$. This is a more precise approximation than the language $\alpha_3\big(\mathsf{Lang}(\mathcal{P}_{\text{ex2}}, \{\langle p_b, \bot\rangle\})\big)$ in Ex. 3 because for abstract words, the former enforces that two "a" actions occur before a "b" action.

For a bound k, the *Succinct* ***SDP*** uses the abstraction $R = \bigcap_{i=1}^{n} \beta_k(L_i)$ in place of $R = \bigcap_{i=1}^{n} \alpha_k(L_i)$. Either R is empty, which proves that G is unreachable, or else R is not empty, in which case the model checker needs to establish whether there exists a concrete word $w \in R$.

Because the β_k abstraction performs the shuffle *after* approximating L_i, the length of a word $w \in R$ is no longer sufficient to determine if w is concrete. That is, for a word $w \in R$ and PDS $\mathcal{P}_i$, the actions that are not executed by $\mathcal{P}_i$ and that were introduced in $\beta_k(L_i)$ because of the shuffle operation must be projected out. Let w be the sequence of actions $a_1 \cdots a_{|w|}$; then for each i, $1 \leq i \leq n$, $\pi_i(w) = \pi_i(a_1) \cdots \pi_i(a_{|w|})$, where the mapping π_i is defined as follows:

$$\pi_i(a) = \begin{cases} a \text{ if } a \in \mathsf{Act}_i \\ \tau \text{ if } a \in \mathsf{Act} \setminus \mathsf{Act}_i. \end{cases}$$

If the length of the projection is less than k (i.e., $|\pi_i(w)| < k$), then this corresponds to a concrete execution for process $\mathcal{P}_i$. We now formally define the *Succinct* ***SDP***.

Succinct SDP

1. $k = 1$

2. $R = \bigcap_{i=1}^{n} \beta_k(L_i)$
3. if $R = \emptyset$ return false. (G is not reachable.)
4. let $w \in R : \nexists w' \in R \wedge |w'| < |w|$
5. if $\mathsf{max}(|\pi_1(w)|, \ldots, |\pi_n(w)|) < k$ return true. (G is reachable.)
6. $k = k + 1$
7. goto 2

The *Succinct SDP* uses the shuffled prefix abstraction β_k to compute regular over-approximations of the languages of the individual PDSs, and then takes their intersection (step 2). If the intersection result R is empty, then—because β_k produces over-approximations—so is the language $\mathsf{Lang}(CP, G)$ (step 3). Otherwise, the *Succinct SDP* finds a minimal-length word $w \in R$ (step 4). Step 5 checks whether w is concrete for all of the PDSs.

Example 6. Let $CP_3 = (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathsf{Act})$, where $\mathsf{Act} = \{\tau, a, b, c, d\}, \mathsf{Act}_1 = \{\tau, b, d\}$, $\mathsf{Act}_2 = \{\tau, a, c\}$, and $\mathsf{Act}_3 = \{\tau, a, b, c, d\}$. Assume that at $k = 3$, a minimal-length word $w = abcdb \in R$ has been found. Then $\pi_1(w) = bdb$, $\pi_2(w) = ac$, and $\pi_3(w) = abcdb$. Because w is abstract for $k = 3$ ($\pi_1(w) \geq 3$ and $\pi_3(w) \geq 3$), the *Succinct SDP* refines the value of k to be 4, and continues at step 2.

Comparing α_k and β_k: Let the α-SDP be the SDP described in §3 with $\beta_k(L_i)$ replaced by $\alpha_k(L_i)$ in step 2, and does not use projected lengths in step 5, but instead checks whether $|w| < k$. The following theorem compares the two SDPs.

Theorem 1. *If the α-SDP decides reachability of G in CP at an abstraction k, then the Succinct SDP will decide reachability of G in CP at an abstraction $k' \leq k$.*

Proof. If G is reachable, then the α-SDP finds a concrete word $w \in \mathsf{Lang}(CP, G)$ at abstraction k, and $|w| = k - 1$. Because of over-approximation, if $w \in \mathsf{Lang}(CP, G)$, then w will be in $\bigcap_{i=1}^{n} \beta_k(L_i)$, and thus the *Succinct SDP* will find w at abstraction k (or another concrete word $w' \in \mathsf{Lang}(CP, G)$ such that $|w'| = k - 1$). However, because of the increased precision, i.e., $\beta_k(L_i) \subseteq \alpha_k(L_i)$, the *Succinct SDP* can find a concrete word w' at an abstraction $k' < k$.

Otherwise, assume the α-SDP proves that $\mathsf{Lang}(CP, G) = \emptyset$ at abstraction k. For $1 \leq i \leq n$, $\beta_k(L_i) \subseteq \alpha_k(L_i)$. Hence, if $\bigcap_{i=1}^{n} \alpha_k(L_i) = \emptyset$, then $\bigcap_{i=1}^{n} \beta_k(L_i) = \emptyset$. Because $\beta_k(L_i) \subseteq \alpha_k(L_i)$, it is possible for the *Succinct SDP* to prove emptiness at an abstraction $k' < k$. □

5 Abstraction-Refinement-Policy Extensions

The *Succinct SDP* has a simple abstraction-refinement policy: increase the search depth by one (step 6). We now present two extensions to the default abstraction-refinement policy, and conclude the section by showing how they are easily combined to form a third. The extensions are presented for the prefix-abstraction case; however, they can be straightforwardly adapted for the other k-bounded abstractions discussed in §2.

5.1 Multi-Step Abstraction Refinement

The first opportunity for improvement over *Succinct SDP* comes from the fact that step 6 does not make use of any additional information that is present in the intersection result R. Specifically, at step 6, the *Succinct SDP* has found that w is a minimal-length word, and that $\mathsf{max}(|\pi_1(w)|, \ldots, |\pi_n(w)|) \geq k$. (We will use π_{max} to denote the result of the max computation.) To determine whether or not w is concrete, a value for k equal to $\pi_{\mathsf{max}} + 1$ must be used, and thus the *Multi-step SDP* defines the next value of k to be $\pi_{\mathsf{max}} + 1$. (The *Multi-step SDP* is presented side-by-side with the *Succinct SDP*, with boxes indicating the modified steps.)

Succinct SDP	***Multi-step SDP***
1. $k = 1$	1. $k = 1$
2. $R = \bigcap_{i=1}^{n} \beta_k(L_i)$	2. $R = \bigcap_{i=1}^{n} \beta_k(L_i)$
3. if $R = \emptyset$ return false	3. if $R = \emptyset$ return false
4. let $w \in R : \nexists w' \in R \wedge |w'| < |w|$	4. let $w \in R : \nexists w' \in R \wedge |w'| < |w|$
5. if $\pi_{\mathsf{max}} < k$ return true.	5. if $\pi_{\mathsf{max}} < k$ return true.
6. $k = k + 1$	$\boxed{6.}$ $k = \pi_{\mathsf{max}} + 1$
7. goto 2	7. goto 2

The benefit of the *Multi-step SDP* is that the model checker is able to converge more quickly on a value for k that results in finding a counterexample (i.e., a concrete word), and avoids useless word in doing so. This is illustrated in Ex. 7.

Example 7. Revisiting Ex. 6 where $w = abcdb$, the *Multi-step SDP* determines that $\pi_{\mathsf{max}} = 5$ because $\pi_3(w) = abcdb$. Thus, the next abstraction uses a value of 6 for k instead of 4, and the model checker avoids analysis for $k = 4$ and $k = 5$.

5.2 Individual Abstraction Refinement

The *Multi-step SDP* addresses the fact that the *Succinct SDP* is naïve in determining what the next value of k should be. We now define the *Individual SDP*, which addresses *when* a new abstraction should be computed.

Let w be a word in the intersection R. Then for PDSs $\mathcal{P}_i$ and $\mathcal{P}_j$, $i \neq j$, it need not be the case that $|\pi_i(w)| = |\pi_j(w)|$ because Act_i and Act_j can be different. Due to this difference, if the shortest word w is an abstract word then it is not necessarily an abstract word for *both* $\mathcal{P}_i$ and $\mathcal{P}_j$. That is, if w has the global property of being abstract, it may in fact be (locally) concrete for some subset of the PDSs. If w is concrete for $\mathcal{P}_i$ (i.e., $|\pi_i(w)| < k$), then the approximation $\mathcal{A}_i$ for the language $\mathsf{Lang}(\mathcal{P}_i, C_i)$ need not be refined. We now present the *Individual SDP* side-by-side with the *Succinct SDP*, with boxes indicating the modified steps.

Succinct SDP	*Individual* SDP
1. $k = 1$	1. for $i \in 1..n : k_i = 1$
2. $R = \bigcap_{i=1}^{n} \beta_k(L_i)$	2. $R = \bigcap_{i=1}^{n} \beta_{k_i}(L_i)$
3. if $R = \emptyset$ return false	3. if $R = \emptyset$ return false
4. let $w \in R : \nexists w' \in R \wedge \lvert w' \rvert < \lvert w \rvert$	4. let $w \in R : \nexists w' \in R \wedge \lvert w' \rvert < \lvert w \rvert$
5. if $\pi_{\mathsf{max}} < k$ return true	5. if $\bigwedge_{i=1}^{n} \lvert \pi_i(w) \rvert < k_i$ return true
6. $k = k + 1$	6. for $i \in 1..n$: if $\lvert \pi_i(w) \rvert \geq k_i$ then $k_i = k_i + 1$
7. goto 2	7. goto 2

The *Individual* SDP uses n local abstraction levels $k_1, \ldots, k_n$. In step 6, each k_i is only incremented as needed. This extension allows the *Individual* SDP to focus its refinement efforts on only those PDSs that require it.

Example 8. Given CP_3 from Ex. 6, assume that $k_1 = k_2 = k_3 = 3$ and that $w = abcdb$ is the same minimal-length string found. Because $\pi_2(w) = ac$ and $|ac| < k_2$, k_2 is not refined for the next round of approximation by *Individual* SDP. Similarly, because $|\pi_1(w)| \geq k_1$ and $|\pi_3(w)| \geq k_3$, the values for k_1 and k_3 are incremented for the next round. Thus, the next abstraction uses the k_i values $(4, 3, 4)$.

There are two benefits to performing abstraction refinement at the level of an individual PDS. First, it is possible that a counterexample can be found using a very coarse approximation for some of the PDSs in *CP*. This permits the model checker to avoid unnecessarily computing more precise over-approximations (i.e., the β_ks) for such PDSs. Second, before performing the shuffle operation defined by β_{k_i}, the language $\alpha_{k_i}\big(\mathsf{Lang}(\mathcal{P}_i, C_i)\big)$ is represented by a finite set S. Because the size of S can be exponential in the abstraction level k_i, when the model checker is able to use a coarser abstraction for $\mathcal{P}_i$, it does so using smaller sets, and thus using less memory.

5.3 Individual Multi-Step Abstraction Refinement

The logical next step takes advantage of these improvements at the same time.

***Individual Multi-step* SDP**

1. for $i = 1..n : k_i = 1$
2. $R = \bigcap_{i=1}^{n} \beta_{k_i}(L_i)$
3. if $R = \emptyset$ return false.
4. let $w \in R : \nexists w' \in R \wedge |w'| < |w|$
5. if $\bigwedge_{i=1}^{n} (|\pi_i(w)| < k_i)$ return true
6. for $i \in 1..n$: if $|\pi_i(w)| \geq k_i$ then $k_i = |\pi_i(w)| + 1$
7. goto 2

To combine the two new abstraction-refinement policies, only step 6 needs to be modified with respect to the *Individual* SDP. The change is to define the next value of k_i to be $|\pi_i(w)| + 1$ versus $k_i + 1$. By doing so, the *Individual Multi-step* SDP is able to take advantage of the time and space savings provided by the new abstraction-refinement policies. This is seen in the running example Ex. 9.

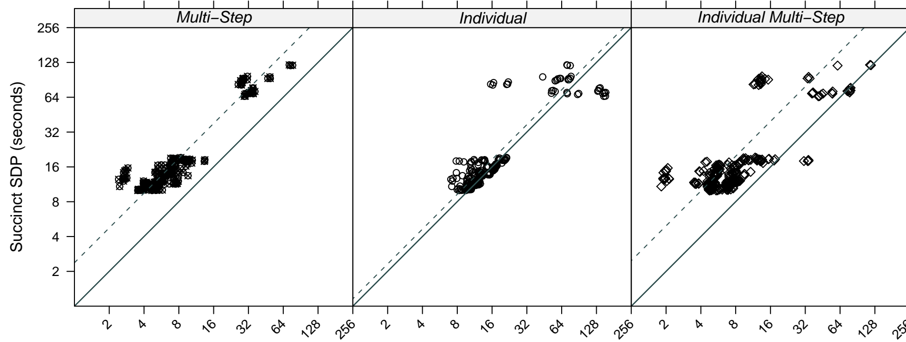

Fig. 1. Performance comparison on 313 queries of each new SDP compared to *Succinct SDP*. Points above the solid diagonal line denote queries in which the new SDP executed faster. The dashed diagonal lines show the median speedups of 2.4,1.2, and 2.5, respectively.

Example 9. Revisiting Ex. 8 with $k_1 = k_2 = k_3 = 3$ and $w = abcdb$, the *Individual Multi-step SDP* would refine k_1 and k_3; however, it is able to make a better choice for the next value of k_3 because $|\pi_3(w)| = 5$. The *Individual Multi-step SDP* would choose the next abstraction to have values $(4, 3, 6)$.

To summarize, the refinement decisions of the four algorithms are as follows:

Succinct SDP	*Multi-step SDP*	*Individual SDP*	*Individual Multi-step SDP*
$k = 4$	$k = 6$	$(k_1, k_2, k_3) = (4, 3, 4)$	$(k_1, k_2, k_3) = (4, 3, 6)$

6 Experimental Evaluation

We extended the model checker with the capability to use each of the new SDPs presented in this paper. We performed a comparison study using the four SDPs on a set of CPDSs generated by the EMPIRE tool—EMPIRE is a tool to statically verify user-specified data-consistency properties in concurrent Java programs [3]—and on the Bluetooth models BT_1, BT_2, and BT_3 reported on in [2]. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 4 GB of memory.

Analyzing Java Programs: We analyzed Java programs from the CONTEST benchmark suite [8], which consists of small programs with specific concurrency errors. For our evaluation, we only analyzed benchmark programs whose bug was listed as "Non-atomic". The performance of the CPDS model checker is more dependent on the use of synchronization and less on the size of the program, and thus the programs from the CONTEST suite are able to stress the model checker. When EMPIRE attempts to verify a data-consistency property, it generates a multitude of CPDSs: one CPDS for each pair of shared-memory locations that are being analyzed for a concurrency bug. Because some benchmarks contain several such locations, many CPDSs are generated.

In total, we ran the model checker on 2273 CPDSs, and it determined reachability on 1918 of those. To ensure that the model checker explores a non-trivial search space, we only compare the running times for CPDSs where the model checker using the

Succinct SDP took time greater than or equal to 10.0 seconds. This filters the set of CPDSs down to 313. The results of our evaluation are shown in Fig. 1.

Fig. 1 shows log-log plots of the times to execute the *Succinct SDP* (y-axis) versus the *Multi-step*, *Individual*, and *Individual Multi-Step* SDPs (x-axis), respectively, for the 313 queries. The solid diagonal lines represent equal running times, and points above it are queries where the abstraction-refinement policies produced a performance improvement. The dashed diagonal lines show the median speedups for each SDP versus the *Succinct SDP*, and are discussed in more detail later.

Our evaluation shows that using the *Multi-step SDP* produces a large performance improvement: more than 80% of the analyzed CPDSs enjoyed a speedup of at least 2, with a median speedup of 2.4 and maximum speedup of 5.4. For the *Individual SDP*, the analysis time for 30% of the CPDSs incurred a performance degradation. The reason for this is that the *Individual SDP* can cause the CPDS model checker to enter into pathologically bad behavior where the refinements occur in a round-robin fashion. However, using the *Individual SDP* results in a performance improvement for 70% of the reported benchmarks, with a median speedup of 1.2 and a maximum speedup of 5.3. Analysis with the *Individual Multi-step SDP* resulted in the largest maximum and median speedups of 7.6 and 2.5, respectively. Thus, to obtain the best performance, one should use the combined abstraction-refinement policy. However, future work should include being able to detect when the pathological behavior of *Individual SDP* occurs so that the model checker can resort to using only the *Multi-step SDP*.

Finally, the dashed lines show the median speedup. The *Multi-step SDP* is consistently grouped around the median speedup line, indicating that it produces a reliable speedup on all queries. The *Individual Multi-step SDP* has more variation because it is a combination of the *Multi-step SDP* and the *Individual SDP*. When both perform well, the *Individual Multi-step SDP* has a large performance improvement, and likewise when the *Individual SDP* degrades performance, the *Individual Multi-step SDP* does not result in as much speedup.

Analyzing Bluetooth Models: We also analyzed the Bluetooth models from [2]: BT_1 is the buggy model from [14], BT_2 is the corrected model proposed by the authors of [14], which is bug-free if there are two processes, but has a bug for a three-process configuration [2], and BT_3 is a modified version of BT_2 that has been verified correct for three processes (see [2] more details). The time to analyze each model using the four SDPs is given in Tab. 2. Tab. 1 shows the sequence of refinements used by the *Individual Multi-step SDP* . A bold k_i value indicates a refinement step. For models BT_1, BT_2, and BT_3, the *Succinct SDP* single steps to abstraction levels 6, 9, and 7, respectively. We see that this results in performing unnecessary abstraction refinement that is avoided by the *Individual Multi-step SDP*.

BT_1	BT_2	BT_3
$(2,2,2,2,2)$	$(2,2,2,2,2,2)$	$(2,2,2,2,2,2)$
$(\mathbf{4},2,\mathbf{4},\mathbf{4},\mathbf{4})$	$(\mathbf{4},\mathbf{4},\mathbf{4},\mathbf{4},\mathbf{4},2)$	$(\mathbf{4},\mathbf{4},\mathbf{4},\mathbf{4},\mathbf{4},2)$
$(4,\mathbf{5},4,\mathbf{5},4)$	$(\mathbf{7},4,\mathbf{7},\mathbf{7},4,\mathbf{7})$	$(\mathbf{7},4,\mathbf{7},\mathbf{7},4,\mathbf{7})$
$(4,5,4,\mathbf{6},4)$	$(\mathbf{9},4,7,7,4,7)$	

Table 1. *Individual Multi-step SDP* 's refinement steps for analyzing the Bluetooth models. Each table entry is the k_i-tuple used during an analysis round.

	Succinct SDP	*Multi-step SDP*	*Individual SDP*	*Individual Multi-step SDP*
BT_1	2.76	2.61	1.52	1.41
BT_2	135.57	117.32	130.27	110.7
BT_3	7.48	4.82	7.12	4.51

Table 2. Time in seconds to analyze the Bluetooth models using the four SDPs.

7 Extending Abstraction-Refinement Policies

In each of the three SDPs defined in §5, step 4 reads: "4. let $w \in R : \nexists w' \in R \wedge |w'| < |w|$". This step finds a minimal-length word $w \in R$ to use as the basis for determining the next abstraction. This is a *greedy* heuristic because abstraction refinement is based solely on the first minimal-length word encountered.

The greedy heuristic precluded us from establishing the analog of Thm. 1 for comparing the SDPs from §5 to the *Succinct SDP*. The problem is that the non-uniformity of the π_i functions allows the SDPs from §5 to make different refinement choices depending on which minimal-length word is chosen. For a bound k and a word $w \in R$, we define the function $h(w) = (|\pi_1(w)|, \ldots, |\pi_n(w)|)$, which returns a tuple of the lengths of the projected words $\pi_i(w), 1 \leq i \leq n$, of w with respect to each of the PDSs of *CP*. Consider two minimal-length words w_1 and w_2. Because of the non-uniformity of the projection functions, the tuples $h(w_1)$ and $h(w_2)$ are not necessarily equal. For the *Individual Multi-step SDP*, there are two possibilities when this occurs:

1. w_1 is a concrete word and w_2 is not, or vice versa.[5] If the wrong minimal-length word is chosen, the *Individual Multi-step SDP* will perform an additional round of abstraction refinement when it is not needed.
2. If both w_1 and w_2 are abstract—for some $i, j, 1 \leq i, j \leq n$, $|\pi_i(w_1)| \geq k_i$ and $|\pi_j(w_2)| \geq k_j$—the *Individual Multi-step SDP* could choose an over-refined abstraction when a coarser abstraction would have sufficed.

Thus, it could be the case that the *Individual Multi-step SDP* jumps to an abstraction $(k_1, \ldots, k_n)$ where $\max(k_1, \ldots, k_n)$ is greater than an abstraction k that could be used by the *Succinct SDP* to determine if G is reachable in *CP*.

We next define an algorithm to replace the greedy heuristic for choosing w in step 4, and define the *Individual Multi-step SDP'*. We then prove that if the *Succinct SDP* finds a word $w \in \mathsf{Lang}(CP, G)$ at abstraction k, then the *Individual Multi-step SDP'* will find a word $w' \in \mathsf{Lang}(CP, G)$ such that $\max\big(h(w')\big) = k$. For the case when $\mathsf{Lang}(CP, G) = \emptyset$, we show that the *Individual Multi-step SDP'* could need to be more precise than the *Succinct SDP* by a factor of n, i.e., the number of PDSs.

7.1 Eliminating Greediness

The objective is to minimize the maximal bound k_i necessary to determine the emptiness of $\mathsf{Lang}(CP, G)$. To do so, we choose the word w at step 4 to be the word whose

[5] For expository purposes, we only consider the *Individual Multi-step SDP*, although the discussion that follows also pertains to the *Multi-step SDP* and *Individual SDP*.

maximum projected k_i value is a minimal-maximum for all words in R. That is, the desired word w should have the property that $\nexists w' \in R$. $\max(h(w')) < \max(h(w))$.

We associate with each word $w \in R$ the tuple $h(w) = (x_1, \ldots, x_n)$, and between two tuples t_1 and t_2, define the partial order $t_1 \sqsubseteq t_2$ iff $\bigwedge_{i=1}^{n} t_1^i \leq t_2^i$, where t^i is the ith component of tuple t. For two tuple sets T_1 and T_2, define the tuple-set join function

$$T_1 \sqcup T_2 = \{t \mid t \in (T_1 \cup T_2) \wedge \nexists t' \in (T_1 \cup T_2) \; . \; t' \sqsubset t\}.$$

(The tuple-set join function is the join for the powerset lattice of antichains of tuples of non-negative integers.) The goal is then to compute $S = \bigsqcup_{w \in R}\{h(w)\}$, and to choose a word w such that $\max(h(w))$ is a minimum for all $t \in S$.

We now define a replacement for step 4 that chooses such a word. Because there are potentially infinitely many words in R, the replacement step is cast as solving a weighted-graph problem over a graph that is constructed from $R = (Q, \Sigma, \delta, q_0, F)$.

We define a labeled graph $G = (V, E, \Sigma)$, where there is a vertex $v_q \in V$ for each state $q \in Q$, and there is an edge $(v_q, \sigma, v_{q'}) \in E$ for each transition $(q, \sigma, q') \in \delta$. Associated with each vertex v is a set of tuples of the form $(x_1, \ldots, x_n)$ of non-negative integers. The tuples denote the projected lengths of the words that label paths from v_{q_0} to v in the graph. It is a set because it is necessary to consider all incomparable projected-length tuples that might arise from different paths in the graph to the same vertex. We define the value of a vertex v, $val(v)$, as follows.

$$val(v) = \begin{cases} \{(0,0,0)\} & v = v_{q_0} \\ \bigsqcup_{(v',\sigma,v) \in E}\{t + h(\sigma) \mid t \in val(v')\} & v \neq v_{q_0}. \end{cases}$$

We extract a tuple from the set $S = \bigsqcup_{q_f \in F} val(v_{q_f})$ such that $\mathsf{max}(t)$ is a minimum for all $t \in S$. Finally, the word that is chosen is the word $w \in R$ such that w corresponds to the path whose projected length is the tuple t. (The construction above is easily extended to provide such a witness word.)

7.2 Comparison with the *Succinct SDP*

To eliminate greediness, we modify step 4 from the *Individual Multi-step SDP* to choose w as described in §7.1. This defines the *Individual Multi-step SDP′*, for which we can prove an analog of Thm. 1 for the case where G is reachable. For the case where G is unreachable, we prove an upper bound on the maximum amount of extra precision that the *Individual Multi-step SDP′* could require.

Theorem 2. *If the Succinct SDP decides that G is reachable in CP at abstraction k, then the Individual Multi-step SDP′ will decide that G is reachable in CP at abstraction $(k_1, \ldots, k_n)$ such that $\pi_{max} = k$.*

Proof. If G is reachable, then $\mathsf{Lang}(CP, G) \neq \emptyset$, and there exists a word $w \in \mathsf{Lang}(CP, G)$ such that $\pi_{\mathsf{min}} = \mathsf{max}\big(h(w)\big)$ is a minimum value for all words in $\mathsf{Lang}(CP, G)$. In this case, the *Succinct SDP* will find w (or a concrete word w'' such that $\mathsf{max}\big(h(w'')\big) = \pi_{\mathsf{min}}$) at abstraction $k = \pi_{\mathsf{min}} + 1$. The *Individual Multi-step SDP′* minimizes on the maximum of the projected lengths of the words in the over-approximation, hence it

must be the case that it will always choose a tuple t such that $\mathsf{max}(t) \leq \pi_{\mathsf{min}}$. Therefore, for $1 \leq i \leq n$, it will always choose a value k_i such that $k_i \leq \pi_{\mathsf{min}} + 1$. Hence the *Individual Multi-step* SDP' is always able to find a concrete word w' at an abstraction $(k_1, \ldots, k_n)$ such that for $1 \leq i \leq n$, $k_i \leq k$, because it can find w at that abstraction, i.e., $w' = w$. □

Theorem 3. *If the Succinct* SDP *decides that* G *is unreachable in CP at abstraction* k*, then the Individual Multi-step* SDP' *will decide that* G *is unreachable in CP at abstraction* $(k_1, \ldots, k_n)$ *such that* π_{max} *is at most* $k(n-1)$*.*

Proof. If the *Succinct* SDP proves that G is unreachable in *CP* at abstraction k, then this could require the *Individual Multi-step* SDP' to reach a stage where it uses the abstraction $(k-1, \ldots, k-1)$, but has still not decided reachability. In the worst case, it could proceed from $(k-1, \ldots, k-1)$ to $(k_1, \ldots, k_n)$ such that for all $1 \leq i \leq n$ where $i \neq g$, $k_i = k$, and $k_g = k(n-1)+1$. This can occur when for some $1 \leq g \leq n$, $\mathsf{Act}_g = \mathsf{Act}$, and for all $1 \leq i, j \leq n$, $i, j \neq g$, $\mathsf{Act}_i \cap \mathsf{Act}_j = \emptyset$; i.e., all PDSs synchronize only with PDS $\mathcal{P}_g$. Thus, to push the abstraction of each PDS $\mathcal{P}_i$, $i \neq g$, to level k, it could require that the abstraction for $\mathcal{P}_g$ be increased to $k(n-1)+1$. □

We see that, in the worst case, the *Individual Multi-step* SDP' could need to be more precise by roughly a factor of n to prove that G is not reachable. In practice (see §6), we observed that on 9 of the 1854 CPDSs for which *Succinct* SDP determined that G was not reachable, the *Individual Multi-step* SDP times-out. However, a bad scenario for a CPDS is very rare (9/1854 ≈ 0.5%); moreover, we have never observed a worst-case scenario like the one used in the proof of Thm. 3.

At present, we have not implemented the non-greedy search method due to the complexity of solving the path problem, which is an artifact of the height of the lattice of antichains. Future work includes an evaluation of this strategy to see if using the *Individual Multi-step* SDP' eliminates the time-out for the 9 CPDSs.

8 Related Work

Multi-step abstraction refinement draws inspiration from the research on *counterexample guided abstraction refinement* (CEGAR) [5, 6]. In a similar spirit, CEGAR determines the next-more-precise abstraction by taking into account knowledge gained from the current abstraction. Our work is not specific to refining the Boolean programs that result from predicate abstraction, but instead on how to choose the next abstraction of the program model.

Individual abstraction refinement draws inspiration from two pieces of prior work. First, in BLAST, Henzinger et al. [7] use what they term *lazy abstraction*, which allows the model of the program being analyzed to have varying degrees of precision. This allows BLAST to focus its abstraction-refinement process on only those regions of code that have been found to be too imprecise. For the concurrent programs that we are interested in, we focus at the level of a component of the model of the concurrent program. Second, the *heterogeneous abstractions* of Yahav and Ramalingam [15] allow the program model to focus on selected regions of the heap. Our analyses instead focus on the synchronization messages used by a component of the concurrent system.

References

1. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
2. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: TACAS. (2006)
3. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. In: VMCAI. (2009)
4. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In: ASPLOS. (2008)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
6. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: ICSE. (2003)
7. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. (2002)
8. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Conc. and Comp.: Prac. and Exp. **19**(3) (2007)
9. Nederhof, M.J.: Practical experiments with regular approximation of context-free languages. CoRR (1999)
10. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: FOSSACS'99. Volume LNCS 1578. (1999)
11. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. (2001)
12. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proc. CONCUR. (1997)
13. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Elec. Notes in Theor. Comp. Sci. (1997)
14. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI. (2004)
15. Yahav, E., Ramalingam, G.: Verifying safety properties using separation and heterogeneous abstractions. In: PLDI. (2004)