Language Strength Reduction[⋆]

Nicholas Kidd[1], Akash Lal[1⋆⋆], and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, WI; USA; {kidd, akash, reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA

Abstract. This paper concerns methods to check for atomic-set serializability violations in concurrent Java programs. The straightforward way to encode a reentrant lock is to model it with a context-free language to track the number of successive lock acquisitions. We present a construction that replaces the context-free language that describes a reentrant lock by a regular language that describes a non-reentrant lock. We call this replacement *language strength reduction.* Language strength reduction produces an average speedup (geometric mean) of 3.4. Moreover, for 2 programs that previously exhausted available space, the tool is now able to run to completion.

1 Introduction

Vaziri et al. [1] define an *atomic set* as a set of memory locations that share a consistency property, and a *unit-of-work* as a code fragment that preserves the consistency property. They specify eleven forbidden *data-access patterns* on atomic sets; and show that an atomic-set serializability violation occurs iff one of the data-access patterns is observed during a unit-of-work.

Empire [2] is a static violation[3] detector for Java. Empire abstracts a concurrent Java program into a program written in the Empire Modeling Language (EML). An EML program consists of a finite set of processes, a finite set of global variables, and a finite set of locks. Each process consists of a set of (recursive) functions with the standard control operators. As in Java, an EML lock is reentrant, i.e., it can be acquired multiple times by the process that owns the lock, but it also must be successively released the same number of times. An EML lock is acquired and released by entering and exiting, respectively, a function that is synchronized on the lock. (Java synchronized blocks are modeled as inlined anonymous function invocations in EML.) EML provides a `unit` block that denotes a unit-of-work. The `unit` blocks are allowed to be nested. An example EML process is given in Fig. 1.

An execution trace of an EML process is described by a string of actions, where an action corresponds to reading (writing) a variable, acquiring (releasing) a lock, or entering (exiting) a `unit` block. The set of all execution traces of an

⋆ Supported by NSF under grants CCF-0540955 and CCF-0524051.
⋆⋆ Supported by a Microsoft Research Fellowship.
[3] For this paper, the term "violation" means "atomic-set serializability violation".

```
lock: l;
var : v;

process P0 {
    synchronized(l) get { read v; }

    synchronized(l) set { write v; }

    synchronized(l) testAndSet {
        get();
        if( * )
            set();
    }

    main {
        unit {
            testAndSet()
        }
    }
}
```

Fig. 1: Example program that makes use of reentrant locking.

EML process is described by a context-free language (CFL) of actions. Similarly, the set of all behaviors of an EML lock is described by a CFL. Finally, the set of all interleaved execution traces of an EML program is described by the intersection of a set S of CFLs—one for each EML process and one for each EML lock. Intersection ensures that the mutual-exclusion property of locks is obeyed.

Violation detection is performed by determining the emptiness of the intersection of the set S, augmented with a regular language L_{data} that defines a data-access pattern. Determining the emptiness of the intersection of two or more CFLs is undecidable. This issue is addressed by translating the EML program, along with L_{data}, into a communicating pushdown system (CPDS) [3,4], for which a semi-decision procedure is implemented in the CPDS model checker [4]. The semi-decision procedure over-approximates each CFL by a regular language and then checks whether the intersection of the regular languages is empty (which is decidable). If the intersection is empty or it contains a valid string in each of the original CFLs, then the model checker terminates. Otherwise, the process is repeated using a tighter regular over-approximation of each CFL.

If a language is known to be regular (e.g., L_{data}), then the CPDS model checker can be directed to treat it as such (determining if a CFL is regular is also undecidable). This has two key advantages:

1. *Precision* increases because the model checker uses the exact language.
2. *Cost* decreases because the model checker avoids approximating a CFL.

This paper presents a generic technique that we use to reduce the number of CFLs necessary to model an EML program. It is based on the observation that the CFL for an EML lock can be replaced by a regular language because the EML lock's acquisitions and releases are synchronized with function calls and returns. We call the process of replacing a CFL by a regular language *language strength reduction*. For an EML program with m processes and n locks, applying language strength reduction allows the program to be described by m CFLs and n regular languages, versus $m + n$ CFLs.

Contributions. The observation that pushdown automata are closed under intersection when the stacks are synchronized was formalized in the work of Alur and Madhusudan [5,6]. They defined nested-word languages, which make stack operations explicit in the words of the language, and nested-word automata (NWA), which accept such languages. They showed that these languages are closed under intersection. However, their result does not apply in our setting.

In our setting, a CPDS consists of a set of extended weighted pushdown systems (EWPDSs) [7]. EWPDSs are a generic formalism for modeling recursive programs (cf. §3.1). They are able to model more powerful program abstractions than the pushdown automata used in [5,6]. (EWPDSs can model infinite-state data abstractions, as opposed to pushdown automata, which can only model finite-state data abstractions.) EWPDSs allow one to compute meet-over-all-valid-paths (MOVP) values for the abstraction, which goes beyond the capabilities of the approach proposed by Alur and Madhusudan. The MOVP values capture the set of behaviors of the program modeled by the EWPDS.

Our approach is similar in spirit to [6]. We use an NWA A to model the locking behavior of an EML process. We define the nested-word language of an EWPDS (cf. §4.1) by associating a nested-word with every path of the EWPDS. This makes its stack operations explicit. We give a generic construction that combines A with an EWPDS $\mathcal{E}$ to produce another EWPDS $\mathcal{E}_A$ whose nested-word language is the intersection of the nested-word languages of $\mathcal{E}$ and A. Computing the MOVP value over $\mathcal{E}_A$ captures the set of all behaviors of the program modeled by $\mathcal{E}$ that respect the locking behaviors described by A.

The key to language strength reduction is distinguishing between the lock acquisitions and releases that change the owner of a lock l and those that do not. We show how to achieve this using the NWA A. We then transfer this ability to the EWPDS $\mathcal{E}$ for an EML process via the construction of $\mathcal{E}_A$. This enables us to perform language strength reduction for the lock l (cf. §5).

Our work makes the following contributions:

- We define the notion of the nested-word language of an EWPDS (§4.1). We give a construction to combine an NWA A with an EWPDS $\mathcal{E}$ to produce another EWPDS $\mathcal{E}_A$ (§4.2). This generalizes previous results, and permits verification to be performed using a broader class of abstract domains (see Defn. 2).
- We show how the construction allows one to perform language strength reduction (§5).
- We analyzed 5 programs from the concurrency testing benchmark suite by Eytani et al. [8]. Our technique obtained an average speedup of 3.4 on 3 of the programs. Moreover, for the 2 programs that previously exhausted available space, the tool is now able to run to completion.

$$\begin{array}{lll}
S & \to U & \\
M & \to \epsilon & \mid M\ M \mid (\ M\ ) \\
U & \to M & \mid M\ U \mid (\ U
\end{array}$$

(a)

$$\begin{array}{lllll}
S & \to U^o & & & \\
M^o & \to \epsilon & \mid M^o\ M^o & \mid (_o\ M^n\ )_o \\
U^o & \to M^o & \mid M^o\ U^o & \mid (_o\ U^n \\
M^n & \to \epsilon & \mid M^n\ M^n & \mid (_n\ M^n\ )_n \\
U^n & \to M^n & \mid M^n\ U^n & \mid (_n\ U^n
\end{array}$$

(b)

$$S \to (_o\ )_o\ S \mid (_o \mid \epsilon$$

(c)

Fig. 2: (a) Grammar for the CFL of a reentrant lock. (b) Grammar that distinguishes between outermost and nested parentheses. (c) Grammar for the regular language of a non-reentrant lock.

The remainder of the paper is organized as follows: §2 provides an overview. §3 presents definitions and examples. §4 presents the nested-word language of an EWPDS and the construction that combines an NWA with an EWPDS. §5 presents the language-strength-reduction transformation. §6 describes our experiments. §7 discusses related work.

2 Overview

Consider the EML process in Fig. 1. Let "(" and ")" denote entering and exiting a `synchronized(l)` function, "[" and "]" denote entering and exiting a `unit` block, and R_v and W_v denote reading and writing to the variable v, respectively. The program path

Path 1: $main \to testAndSet \to get \to set \to testAndSet \to main$

can be described by the word w_{path} = "$[((R_v)(W_v))]$". Removing all symbols that do not model a change in the state of the lock l produces the word w_l = "$(()())$". In general, due to recursion, the language that describes the set of possible program behaviors with respect to l is a partially-balanced matched-parenthesis language, whose grammar is shown in Fig. 2(a).

For *Path 1*, there are two distinct types of lock acquisitions: ownership-changing acquisitions (OC) and non-ownership-changing acquisitions (nOC). The dual also holds for lock releases. With respect to w_l, these two distinct types correspond to outermost parentheses, denoted by "$(_o)_o$", and nested parentheses, denoted by "$(_n)_n$", respectively. Using this notation, w_l can be rewritten as "$(_o(_n)_n(_n)_n)_o$". Fig. 2(b) extends this to the language level by distinguishing between the outermost and nested parentheses of Fig. 2(a).

Observation 1 *With respect to the executions of an EML program, only the OC lock acquisitions and releases enforce mutual exclusion. For a program trace, projecting out the nOC lock acquisitions and releases does not change the set of instructions that are guarded by locks.*

Projecting out the nested parentheses for w_l results in "$(_o)_o$". Performing the projection on the grammar in Fig. 2(b) results in a regular language whose grammar is shown in Fig. 2(c).

This paper presents a technique that allows us to use the simpler language in Fig. 2(c) in place of the language in Fig. 2(a). We call this replacement *language strength reduction*. Language strength reduction provides the precision and cost benefits highlighted by items 1 and 2 of §1.

Language strength reduction relies on the ability to distinguish between the OC and nOC lock acquisitions of an EML process. In §3.3, we show how this distinction can be captured by an NWA. Having defined the language of Fig. 2(b) via an NWA A, we combine it with the EWPDS $\mathcal{E}$ that represents an EML process. This results in another EWPDS $\mathcal{E}_A$ on which we then project out all nOC lock acquisitions and releases—the end result being that each EML lock is modeled by the regular language shown in Fig. 2(c) in the CPDS model checker. Using the simpler language of Fig. 2(c) leads to the speedups reported in §6.

The goal of language strength reduction is to model reentrant locks with non-reentrant locks without sacrificing soundness or precision. This problem can be tackled by either source-code modification or by manipulating the program model. In our model checker's tool chain, a CPDS is produced from a concurrent Java program, and thus we followed the approach of modifying the EWPDSs that make up the generated CPDS. A benefit of this approach is that we have developed generic techniques that apply to a declarative specification of the set of locks. That is, given the set of lock names, the techniques we present perform language strength reduction automatically.

3 Definitions and Examples

3.1 Extended Weighted Pushdown Systems

Definition 1. *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of* states*, Γ is a finite set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$, then $\langle p, \gamma u \rangle \Rightarrow \langle p', u'u \rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side [9]. A rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, $u \in \Gamma^*$, is called a *push*, *step*, or *pop* rule if $|u| = 2$, $|u| = 1$, or $|u| = 0$, respectively.

A PDS naturally models a program's control flow. The standard approach is as follows: P contains a single state p, Γ corresponds to the nodes of the program's interprocedural control flow graph (ICFG), and Δ corresponds to edges of the program's ICFG (see Fig. 3). We denote the entry point of a program's main function by e_{main}, and let $c_{\text{init}} = \langle p, e_{\text{main}} \rangle$. A *run* of $\mathcal{P}$ is a rule

Rule	Control flow modeled
$\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle p, n_c \rangle \hookrightarrow \langle p, e_f\ r_c \rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f \rangle \hookrightarrow \langle p, \varepsilon \rangle$	Return from f at exit x_f

Fig. 3: The encoding of an ICFG's edges as PDS rules.

sequence $\rho = [r_1, \ldots, r_j]$ that transforms c_{init} into some other configuration c.[4] We denote the set of all runs of $\mathcal{P}$ by *Runs*($\mathcal{P}$), which represents the set of all interprocedurally-valid paths in the program.

An extended weighted pushdown system (EWPDS) is obtained by augmenting a PDS with a weight domain [10,3] and a set of merging functions [7]. Weights encode the effect that each statement (or PDS rule) has on the data state of the program. Merging functions are used to fuse the local state of the calling procedure as it existed just before the call with the global state produced by the called procedure.

Definition 2. *A* **weight domain** *is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set whose elements are called* **weights***,* $\overline{0}, \overline{1} \in D$*, and* $\oplus$ *(the combine operation) and* $\otimes$ *(the extend operation) are binary operators on* D *such that*

1. $(D, \oplus)$ *is a commutative monoid with* $\overline{0}$ *as its neutral element, and where* $\oplus$ *is idempotent (i.e., for all* $a \in D$*,* $a \oplus a = a$*).* $(D, \otimes)$ *is a monoid with the neutral element* $\overline{1}$*.*
2. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have*
   $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.
3. $\overline{0}$ *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$*,* $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*.*
4. *In the partial order* $\sqsubseteq$ *defined by* $\forall a, b \in D,\ a \sqsubseteq b$ *iff* $a \oplus b = a$*, there are no infinite descending chains.*

Example: The Prefix Weight Domain for CPDSs [3]. For a CFL L over finite alphabet Σ, the prefix abstraction precisely models each word $w \in L$ whose length is less than a bound k. If $|w| \geq k$, then w is approximated by the regular language $w|_k \Sigma^*$, where $w|_k$ denotes the prefix of w of length k. Because there are only a finite number of words and prefixes whose lengths are less than or equal to k, the prefix abstraction produces a regular approximation of L.

For two words $w_1 = a_1 \ldots a_i$ and $w_2 = b_1 \ldots b_j$, let $w_1 \bowtie_k w_2$ be the word $(a_1 \ldots a_i b_1 \ldots b_j)|_k$. We extend $\bowtie_k$ to finite sets in the obvious way. For a finite alphabet Σ and bound k, let D be the powerset of $\bigcup_{0 \leq i \leq k} \Sigma^i$. The prefix weight domain is defined as $\mathcal{S}|_k = (D, \cup, \bowtie_k, \emptyset, \{\epsilon\})$.

Definition 3. *A function* $m : D \times D \rightarrow D$ *is a* **merging function** *with respect to a weight domain* $(D, \oplus, \otimes, \overline{0}, \overline{1})$ *if it satisfies the following properties:*

[4] It is not necessary to restrict the definition of a run to start from the initial configuration. However, this simplifies the discussion.

	Rules	Weight	d_{const}
1	$\langle p, e_{\text{main}} \rangle \hookrightarrow \langle p, n_{16} \rangle$	$\overline{1}$	
2	$\langle p, n_{16} \rangle \hookrightarrow \langle p, n_{17} \rangle$	$\{ \, [\, \}$	
3	$\langle p, n_{17} \rangle \hookrightarrow \langle p, e_{\text{testAndSet}}\ n_{18} \rangle$	$\{ \, (\, \}$	$\{ \,) \, \}$
4	$\langle p, e_{\text{testAndSet}} \rangle \hookrightarrow \langle p, n_{10} \rangle$	$\overline{1}$	
5	$\langle p, n_{10} \rangle \hookrightarrow \langle p, e_{\text{get}}\ n_{11} \rangle$	$\{ \, (\, \}$	$\{ \,) \, \}$
6	$\langle p, e_{\text{get}} \rangle \hookrightarrow \langle p, x_{\text{get}} \rangle$	$\{R_v\}$	
7	$\langle p, x_{\text{get}} \rangle \hookrightarrow \langle p, \epsilon \rangle$	$\overline{1}$	
8	$\langle p, n_{11} \rangle \hookrightarrow \langle p, n_{12} \rangle$	$\overline{1}$	
9	$\langle p, n_{12} \rangle \hookrightarrow \langle p, e_{\text{set}}\ x_{\text{testAndSet}} \rangle$	$\{ \, (\, \}$	$\{ \,) \, \}$
10	$\langle p, e_{\text{set}} \rangle \hookrightarrow \langle p, x_{\text{set}} \rangle$	$\{W_v\}$	
11	$\langle p, x_{\text{set}} \rangle \hookrightarrow \langle p, \epsilon \rangle$	$\overline{1}$	
12	$\langle p, x_{\text{testAndSet}} \rangle \hookrightarrow \langle p, \epsilon \rangle$	$\overline{1}$	
13	$\langle p, n_{18} \rangle \hookrightarrow \langle p, x_{\text{main}} \rangle$	$\{ \,] \, \}$	
14	$\langle p, x_{\text{main}} \rangle \hookrightarrow \langle p, \epsilon \rangle$	$\overline{1}$	
15	$\langle p, n_{11} \rangle \hookrightarrow \langle p, x_{\text{testAndSet}} \rangle$	$\overline{1}$	

Fig. 4: EWPDS rules that encode EML process `P0` from Fig. 1 (subscripts correspond to the line numbers). Only the constant weight d_{const} is shown for the merging functions.

1. **Strictness.** *For all $a \in D$, $m(\overline{0}, a) = m(a, \overline{0}) = \overline{0}$.*

2. **Distributivity.** *The function distributes over $\oplus$. For all $a, b, c \in D$,*

$$m(a \oplus b, c) = m(a, c) \oplus m(b, c) \text{ and } m(a, b \oplus c) = m(a, b) \oplus m(a, c)$$

Example: The Prefix Merging Functions of Empire. The prefix merging functions used by Empire are of the form $\lambda d_1.\lambda d_2.d_1 \otimes d_2 \otimes d_{\text{const}}$, where d_{const} is either $\overline{1}$ for invoking non-synchronized functions, or $\{ \,) \, \}$ for invoking a function that is synchronized on a lock l. Note that placing the close-parenthesis symbol, corresponding to the release of a lock, inside of a merge function accurately reflects the behavior of returning from a synchronized function.

Definition 4. *Let $\mathcal{M}$ be the set of all merging functions on weight domain $\mathcal{S}$, and let Δ_2 denote the set of push rules of a PDS $\mathcal{P}$. An* **extended weighted pushdown system** *is a quadruple $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a weight domain, $f : \Delta \to D$ is a map that assigns a weight to each rule of $\mathcal{P}$, and $g : \Delta_2 \to \mathcal{M}$ assigns a merging function to each rule in Δ_2.*

Example: An EWPDS for an EML process. For an EML process π, an EWPDS $\mathcal{E}\langle \pi \rangle$ is generated using the schema from Fig. 3, the prefix weight domain, and the prefix merging functions. Fig. 4 presents the rules that encode process `P0` from Fig. 1.

Run of an EWPDS. A run of an EWPDS $\mathcal{E}$ is simply a run of its underlying PDS. We denote the set of all runs of $\mathcal{E}$ by $Runs(\mathcal{E})$, and the set of runs ending in configuration c as $Runs(\mathcal{E}, c)$. Using f and g, we can associate a value to a run ρ, denoted by $val(\rho)$. To do so, we define the helper functions $val[r]$, *build*, and *flatten*. The function $val[r](z, S)$ takes a weight and a weight-rule stack, and returns a weight and weight-rule stack:

$$
\mathit{val}[r](z,S) = \begin{cases} (z \otimes f(r), S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \\ (\overline{1}, (z,r)||S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \\ (g(r_c)(z_c, f(r_c)\otimes z \otimes f(r)), S') & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\epsilon\rangle \\ & \quad \text{and } S = (z_c, r_c)||S' \\ (z \otimes f(r), S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\epsilon\rangle \text{ and } S = \emptyset \end{cases}
$$

The function $\mathit{build}(\rho)$ maps a run to a weight and weight-rule stack as follows:

$$
\begin{aligned} \mathit{build}([]) &= (\overline{1}, \emptyset) \\ \mathit{build}([r_1,\ldots,r_j]) &= \mathit{val}[r_j](\mathit{build}([r_1,\ldots,r_{j-1}])) \end{aligned}
$$

The function $\mathit{flatten}(z,S)$ "flattens" a weight and weight-rule stack by using the extend ($\otimes$) operation:

$$
\begin{aligned} \mathit{flatten}(z,\emptyset) &= z \\ \mathit{flatten}(z,(z_c,r_c)||S') &= \mathit{flatten}(z_c \otimes f(r_c) \otimes z, S') \end{aligned}
$$

Given these definitions, $\mathit{val}(\rho) = \mathit{flatten}(\mathit{build}(\rho))$.

Example: Valuation of Path 1. Using the EWPDS rules of Fig. 4, and for a prefix bound $k > 10$, one can verify that $\mathit{val}([r_1,\ldots,r_{14}]) = \{\ [((R_v))(W_v))]\ \}$, which is the set containing only the word given in §2 for *Path 1*.

Definition 5. *For EWPDS $\mathcal{E}$ and a set of configurations C, the **meet-over-all-valid-paths** value* $\mathrm{MOVP}_{\mathcal{E}}(C)$ *is defined as* $\bigoplus\{\mathit{val}(\rho) \mid \rho \in \mathit{Runs}(\mathcal{E},c), c \in C\}$.

The MOVP value captures the net effect of all paths leading to a set of configurations. An algorithm for computing MOVP is given in [7].

Example: MOVP **for EML process** `P0` **from Fig. 1.** Let $\mathcal{E}\langle\texttt{P0}\rangle$ be the EWPDS for process `P0` with rules given in Fig. 4. For a prefix bound $k > 10$, $\mathrm{MOVP}_{\mathcal{E}\langle\texttt{P0}\rangle}(\langle p, x_{\mathrm{main}}\rangle) = \{\ [((R_v)(W_v))]\ ,\ [((R_v))]\ \}$. The first string describes the path that follows the true branch of the `if` statement at line 11 in Fig. 1, and the second string describes the path that follows the false branch. Because process `P0` has only two valid paths and $k > 10$, the MOVP weight precisely describes the behavior of process `P0`. However, if k was instead the value 8, then the result of the same MOVP computation would be $\{\ [((R_v)(W_v)\Sigma^*\ ,\ [((R_v))]\ \}$. Note that the first string has been approximated by an infinite set of strings.

3.2 Communicating Pushdown System

A CPDS consists of a set of EWPDSs $\mathcal{E}_1,\ldots,\mathcal{E}_n$, where each EWPDS $\mathcal{E}_i$ uses the prefix weight domain and merging functions, and a set of target configurations $C_1,\ldots,C_n$. The CPDS model checker computes: $S = \bigcap_{1\le i\le n} \mathrm{MOVP}_{\mathcal{E}_i}(C_i)$. The set S is the intersection of the prefix abstractions for each CFL that is modeled by an EWPDS. If $S = \emptyset$, then so is the intersection of the CFLs. Otherwise, let w be the shortest word in S. If $|w| = k$, then k is incremented and the process repeats. Otherwise, w represents a concrete execution of the EML program that reaches the target configurations.

3.3 Nested Word Automata

Alur et al. [6] define a nested word to be a pair (w, v), where w is a word $a_1 \dots a_k$ over a finite alphabet and v, the *nesting relation*, is a subset of $\{1, 2, \dots, k\} \times (\{1, 2, \dots, k\} \cup \{\infty\})$. The nesting relation denotes a set of *properly nested* hierarchical edges of a nested word. For a valid nesting relation, $v(i, j)$ implies $i < j$, and for all i', j' such that $v(i', j')$ holds and $i < i'$, then either $j < i'$ or $j' < j$. Given v, i is a *call position* if $v(i, j)$ holds for some j, a *return position* if $v(k, i)$ holds for some k, and an *internal position* otherwise.

A set of nested words is *regular* if it can be modeled by a *nested-word automaton* (NWA) [6]. An NWA A is a tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation that consists of three components:

- $\delta_c \subseteq Q \times \Sigma \times Q$ defines the transition relation for call positions.
- $\delta_i \subseteq Q \times \Sigma \times Q$ defines the transition relation for internal positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ defines the transition relation for return positions.

Starting from q_0, an NWA A reads a nested word $nw = (w, v)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and the nesting relation. That is, if A is in state q when reading input symbol σ at position i in w, then if i is a call or internal position, A makes a transition to q' using $(q, \sigma, q') \in \delta_c$ or $(q, \sigma, q') \in \delta_i$, respectively. Otherwise, i is a return position and $v(j, i)$ holds for some j. Let q_c be the state A was in just before the transition it made on the j^{th} symbol; then A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A accepts nw [6].

We use $L(A)$ to denote the nested-word language that A accepts, and $L(A, q)$ to denote the nested-word language such that for each nested word $nw \in L(A, q)$, A is left in state q after reading nw. We extend this notion to sets of states in the obvious way. Thus, $L(A) = L(A, F)$.

An NWA Template for Lock Behavior. For an EML lock l and process π with set of functions *Sync* synchronized on l and set of functions *Fun* not synchronized on l, the locking behavior of π on l is defined by an NWA $A\langle\pi\rangle = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{\boxtimes, \square\}$, Σ is the set of control locations of π, $q_0 = \square$, $F = Q$, and δ is defined in Tab. 1. (The transitions in Tab. 1 are instantiated for all $q \in Q$, $e_{\text{sync}} \in \{e_f \mid f \in \mathit{Sync}\}$, $e \in \{e_f \mid f \in \mathit{Fun}\}$, $x_{\text{sync}} \in \{x_f \mid f \in \mathit{Sync}\}$, $x \in \{x_f \mid f \in \mathit{Fun}\}$, and $\sigma \in (\Sigma - \{e_{\text{sync}}, x_{\text{sync}}, e, x\})$.)

δ_c	δ_r	δ_i
$(q, e_{\text{sync}}, \boxtimes)$	$(\boxtimes, q_c, x_{\text{sync}}, q_c)$	(q, σ, q)
(q, e, q)	(q, q, x, q)	

Table 1: An NWA template for the locking behavior of an EML process.

$A\langle\pi\rangle$ consists of two states: locked ($\boxtimes$) and unlocked ($\square$). The entry to and exit from a function `f` are denoted by $e_{\mathtt{f}}$ and $x_{\mathtt{f}}$, respectively. When an

l-synchronized function is called, $A\langle\pi\rangle$ makes a transition to the locked state via the transitions $(q, e_{\text{sync}}, \boxtimes)$. When returning from a function, the state of the caller is restored. For example, the transitions $(\boxtimes, q_c, x_{\text{sync}}, q_c)$ ensure that $A\langle\pi\rangle$ goes to state q_c of the caller.

Template Usage. For EML process `P0` from Fig. 1, let $\mathcal{E}\langle\texttt{P0}\rangle$ be the EWPDS that models `P0` with the rules shown in Fig. 4, and let $A\langle\texttt{P0}\rangle$ be the NWA that results from instantiating the above template with `P0`. With respect to the locking behavior of `P0`, $\mathcal{E}\langle\texttt{P0}\rangle$ cannot distinguish between OC and nOC lock acquisitions and releases, while $A\langle\texttt{P0}\rangle$ is able to do so via its state space. The transitions $(\square, e_{\text{sync}}, \boxtimes)$ and $(\boxtimes, e_{\text{sync}}, \boxtimes)$ in δ_c are the OC and nOC lock acquisitions, respectively; and transitions $(\boxtimes, \boxtimes, e_{\text{sync}}, \boxtimes)$ and $(\boxtimes, \square, e_{\text{sync}}, \square)$ in δ_r are the nOC and OC lock releases, respectively.

We show how to combine $\mathcal{E}\langle\texttt{P0}\rangle$ and $A\langle\texttt{P0}\rangle$ in §4 to construct another EWPDS $\mathcal{E}_A\langle\texttt{P0}\rangle$, such that $\mathcal{E}_A\langle\texttt{P0}\rangle$ contains the same behaviors as $\mathcal{E}\langle\texttt{P0}\rangle$, but is able to distinguish between the OC and nOC lock acquisitions and releases. Once such a distinction can be made, we leverage *Observation 1* to remove all nOC lock acquisitions and releases from $\mathcal{E}_A\langle\texttt{P0}\rangle$. This makes it possible to model an EML lock with the trivial language shown in Fig. 2(c).

4 Combining an NWA with an EWPDS

We first define the notion of the *nested-word language* of an EWPDS, which establishes a relationship between the NWA and EWPDS formalisms. Additionally, it allows us to formally reason about the construction of §4.2 that combines an NWA with an EWPDS.

4.1 The Nested-Word Language of an EWPDS

The nested-word language of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$, denoted by $L(\mathcal{E})$, is defined in terms of the set of runs of $\mathcal{E}$. Intuitively, if (w, v) is a nested word in $L(\mathcal{E})$, w consists of the sequence of left-hand-side stack symbols $\gamma_1 \dots \gamma_j$ for a run $[r_1, \dots, r_j]$ of $\mathit{Runs}(\mathcal{E})$, and v encodes the matching calls and returns. We additionally require that the valuation of the run not be equal to the weight zero, i.e., $\mathit{val}(\rho) \neq \bar{0}$. This notion is formalized by defining the function *post*, which maps a run of $\mathcal{E}$ to a nested word. The function *post* is defined recursively in terms of the helper function $\mathit{post}[r](w, v)$.

For a nested word $nw = (w, v)$ and rule $r \in \Delta$, $\mathit{post}[r](w, v)$ is defined as follows:

$$\mathit{post}[r](w,v) = \begin{cases} (w\gamma, v) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \\ (w\gamma, (v - \{\langle i, \infty\rangle\}) \cup \{\langle i, |w\gamma|\rangle\}) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle, \\ & \quad i = \mathit{max}(\{j \mid \langle j, \infty\rangle \in v\}) \\ (w\gamma, v \cup \{\langle |w\gamma|, \infty\rangle\}) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\ \gamma''\rangle \end{cases}$$

Using $post[r]$, we define the function $post([r_1 \dots r_j])$[5] as follows:

$$\begin{aligned} post([]) &= (\epsilon, \emptyset) \\ post([r_1, \dots, r_j]) &= post[r_j](post([r_1, \dots, r_{j-1}])) \end{aligned}$$

Definition 6. *For an EWPDS $\mathcal{E}$, the nested-word language $L(\mathcal{E})$ is defined as $L(\mathcal{E}) = \{post(\rho) \mid \rho \in Runs(\mathcal{E}) \wedge val(\rho) \neq \overline{0}\}$.*

We will sometimes wish to further restrict $L(\mathcal{E})$ by an acceptance criterion, which we call *φ-acceptance.*

Definition 7. *The **φ-accepted** nested-word language for an EWPDS $\mathcal{E}$ and function $\varphi : D \to \mathbb{B}$ is defined as $L^{\varphi}(\mathcal{E}) = \{post(\rho) \mid \rho \in Runs(\mathcal{E}) \ \wedge \ val(\rho) \neq \overline{0} \wedge \ \varphi(val(\rho))\}$.*

4.2 Construction

The construction that combines an EWPDS $\mathcal{E}$ with an NWA A produces another EWPDS $\mathcal{E}_A$. The weight domain of $\mathcal{E}_A$ models the transition relation of A in addition to the original weight domain of $\mathcal{E}$. This is accomplished via a *relational weight domain.*

Definition 8. *A **weighted relation** on a set G, with weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$, is a function from $(G \times G)$ to D. The composition of two weighted relations R_1 and R_2 is defined as $(R_1; R_2)(g_1, g_3) = \oplus\{w_1 \otimes w_2 \mid \exists g_2 \in G : w_1 = R_1(g_1, g_2), w_2 = R_2(g_2, g_3)\}$. The union of the two weighted relations is defined as $(R_1 \cup R_2)(g_1, g_2) = R_1(g_1, g_2) \oplus R_2(g_1, g_2)$. The identity relation is the function that maps each pair (g, g) to $\overline{1}$ and others to $\overline{0}$. The reflexive transitive closure is defined in terms of these operations, as usual. If R is a weighted relation and $R(g_1, g_2) = z$, then we write $g_1 \xrightarrow{z} g_2 \in R$.*

Definition 9. *If $\mathcal{S}$ is a weight domain with set of weights D and G is a finite set, then the **relational weight domain** on $(G, \mathcal{S})$ is defined as $(2^{G \times G \to D}, \cup, ;, \emptyset, id)$: weights are weighted relations on G, combine is union, extend is weighted relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the weighted identity relation on $(G, \mathcal{S})$.*

This weight domain can be encoded symbolically using techniques such as algebraic decision diagrams [11].

The weight domain of $\mathcal{E}_A$ will be a relational weight domain on $(G, \mathcal{S})$, where G encodes the state space of A, and $\mathcal{S}$ is the weight domain of $\mathcal{E}$. Intuitively, for a run ρ of $\mathcal{E}_A$, the valuation $val(\rho)$ in $\mathcal{E}_A$ is a weighted relation R such that if $q_1 \xrightarrow{z} q_2 \in R$, then (i) the valuation $val(\rho)$ in $\mathcal{E}$ must be equal to z, and (ii) starting from state q_1, A can make a transition to state q_2 on the

[5] $post[r](nw)$ is not always defined because of *max*; and thus neither is *post*. However, for a run of a PDS from the initial configuration, both will always be defined.

nested word $post(\rho)$. We now introduce some notation needed to show how this is accomplished by the construction.

First, for an NWA $A = (Q, \Sigma, q_0, \delta, F)$, we define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. The relational weight domain of $\mathcal{E}_A$ is over the finite set $Q \times \Sigma_\epsilon$. The pairing of Q with Σ_ϵ is used below to properly model the return relation δ_r of A. We denote an element (q, σ) of this set by q^σ, but omit σ when $\sigma = \epsilon$.

Second, we define the restriction of δ_i to σ, denoted by $\delta_i^{|\sigma}$, to be the relation with $(q_1, q_2) \in \delta_i^{|\sigma}$ iff $(q_1, \sigma, q_2) \in \delta_i$. Note that by representing (q_1, q_2) as $(q_1^\epsilon, q_2^\epsilon)$, $\delta_i^{|\sigma}$ can be embedded into $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$ using only states in which $q \in Q$ is paired with ϵ (i.e., q^ϵ). Henceforth, we abuse notation and use $\delta_i^{|\sigma}$ to mean the version that is embedded in $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$. We define $\delta_c^{|\sigma}$ similarly. $\delta_i^{|\sigma}$ and $\delta_c^{|\sigma}$ will be the relational part of the weights that annotate step and push rules in $\mathcal{E}_A$. By restricting δ_i (δ_c) to σ, a run of $\mathcal{E}_A$ enforces that $\mathcal{E}$ and A are kept in lock step (see *Construction 1*).

Third, we define the function *expand*(σ), which takes as input a symbol $\sigma \in \Sigma$ and generates the relation $\{(q^\epsilon, q^\sigma) \mid q \in Q\}$. This is used to pass the return location to $\mathcal{E}_A$'s merging functions, which is needed for properly modeling the return relation δ_r of A.

Fourth, we define $\hat{\delta}$ so that $(q_r^\sigma, q_c, q) \in \hat{\delta}$ iff $(q_r, q_c, \sigma, q) \in \delta_r$. Notice that $\hat{\delta}$ combines the input symbol σ used in δ_r with the return state. This is used by $\mathcal{E}_A$'s merging functions to receive the return location passed via *expand*.

Construction 1. The combination of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ and an NWA $A = (Q, \Sigma, q_0, \delta, F)$ is modeled by an EWPDS $\mathcal{E}_A$ that has the same underlying PDS as $\mathcal{E}$, but with a new weight domain and new assignments of weights and merging functions to rules: $\mathcal{E}_A = (\mathcal{P}_A, \mathcal{S}_A, f_A, g_A)$, where $\mathcal{P}_A = \mathcal{P}$, $\mathcal{S}_A = (D_A, \oplus_A, \otimes_A, \overline{0}_A, \overline{1}_A)$ is the relational weight domain on the set $Q \times \Sigma_\epsilon$ and weight domain $\mathcal{S}$, and f_A and g_A are defined as follows:

1. For step rule $r = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle \in \Delta$, $f_A(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_i^{|n_1}\}$.
2. For push rule $r = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle \in \Delta$, $f_A(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_c^{|n_c}\}$ and

$$g_A(r)(w_c, w_x) = \left\{ q_1 \xrightarrow{z} q_2 \mid \exists a, b : \begin{pmatrix} q_1 \xrightarrow{z_1} a \in w_c \\ \wedge\ a \xrightarrow{z_2} b \in (f_A(r) \otimes w_x) \\ \wedge\ \hat{\delta}(b, a, q_2) \end{pmatrix}, z = g(r)(z_1, z_2) \right\}$$

3. For pop rule $r = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$, $f_A(r) = \{q \xrightarrow{f(r)} q^x \mid (q, q^x) \in expand(x)\}$.

The properties of *Construction 1* are that (i) $\mathcal{E}_A$'s nested-word language is the intersection of those of $\mathcal{E}$ and A, and (ii) the behaviors of $\mathcal{E}_A$ (summarized by its MOVP values) are those of $\mathcal{E}$ restricted by A. Formally, these are captured by Thm. 1 and Cor. 1.

Theorem 1. *An NWA A combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_A$ such that $L^{\varphi}(\mathcal{E}_A) = L(A, Q) \cap L(\mathcal{E})$, where for a run ρ of $\mathcal{E}_A$ with $z = val(\rho)$, $\varphi(z) = \exists q \in Q : q_0 \xrightarrow{y} q \in z$, and $y \neq \overline{0}$.*

Proof. See [12].

Corollary 1. *An NWA A combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_A$ such that $\mathrm{MOVP}_{\mathcal{E}_A}(C) = \bigoplus\{val(\rho) \mid \rho \in Runs(\mathcal{E}, c), c \in C, post(\rho) \in L(A, Q)\}$.*

Complexity of $\mathcal{E}_A$ versus $\mathcal{E}$. The complexity of computing MOVP on an EWPDS is proportional to the *height* of the weight domain, which is defined to be the length of the longest descending chain in the domain.[6] If H is the height of the weight domain of $\mathcal{E}$, then the height of the weight domain of $\mathcal{E}_A$ is $H|Q|^2$, where Q is the set of states of A. Because $\mathcal{E}$ and $\mathcal{E}_A$ have the same PDS, the complexity of computing MOVP on $\mathcal{E}_A$ only increases by a factor of $|Q|^2$.

5 Language Strength Reduction for the Empire Tool

Thm. 1 and Cor. 1 show that the EWPDS $\mathcal{E}_A$ created by *Construction 1* is able to model both $\mathcal{E}$ and A simultaneously (for nested words in their intersection). This capability allowed us to use language strength reduction to improve the Empire tool's performance. To make the discussion clear, we focus on EML process `P0` from Fig. 1. The first three steps are as follows:

1. $\mathcal{E}\langle\texttt{P0}\rangle$ is generated using the original Empire translation. Recall that the weight domain of $\mathcal{E}\langle\texttt{P0}\rangle$ is the prefix weight domain.
2. Let *Locks* be the set of locks of the EML program. For each lock $l \in \mathit{Locks}$, an NWA $A_l\langle\texttt{P0}\rangle$ is generated using the NWA template from §3.3. Define $A\langle\texttt{P0}\rangle$ to be $\bigcap_{l \in \mathit{Locks}} A_l\langle\texttt{P0}\rangle$. The state space Q of $A\langle\texttt{P0}\rangle$ is equal to $2^{|\mathit{Locks}|}$. That is, each $q \in Q$ represents a set of locks that are held. Note that in Fig. 1, there is only one lock l, and thus $A\langle\texttt{P0}\rangle = A_l\langle\texttt{P0}\rangle$, and $Q = \{\square, \boxtimes\}$.
3. $\mathcal{E}_A\langle\texttt{P0}\rangle$ is generated from $\mathcal{E}\langle\texttt{P0}\rangle$ and $A\langle\texttt{P0}\rangle$ using *Construction 1*. The NWA template from §3.3 is instantiated for $A\langle\texttt{P0}\rangle$, and thus $L(A\langle\texttt{P0}\rangle) = L(\mathcal{E}\langle\texttt{P0}\rangle)$. Hence, $\mathcal{E}_A\langle\texttt{P0}\rangle$ contains the same behaviors as $\mathcal{E}\langle\texttt{P0}\rangle$. Additionally, due of Thm. 1, $\mathcal{E}_A\langle\texttt{P0}\rangle$ is able to distinguish between OC and nOC lock acquisitions and releases in the same manner as $A\langle\texttt{P0}\rangle$.

From Fig. 2(a) to Fig. 2(b) The weight domain of $\mathcal{E}_A\langle\texttt{P0}\rangle$ is a relational weight domain over Q and the prefix weight domain of $\mathcal{E}\langle\texttt{P0}\rangle$. In $\mathcal{E}\langle\texttt{P0}\rangle$, the rule

[6] EWPDSs can also be used when the height is unbounded, provided there are no infinite descending chains. To simplify the discussion of complexity, we assume the height to be finite.

(a)	(b)	(c)
$[((R_v)(W_v\mathbf{\Sigma}^*$	$[(_o(_nR_v)_n(_nW_v\mathbf{\Sigma}^*$	$[(_oR_vW_v)_o]$

Table 2: For *Path 1* of $\mathcal{E}_A\langle\texttt{P0}\rangle$, a prefix bound of 7, and $\rho = [r_1, \ldots, r_{14}]$ from Fig. 4, cols. (a) and (b) present $\mathit{val}(\rho)(\square,\square)$ before and after distinguishing between OC and nOC lock acquisitions and releases, respectively. Col. (c) presents $\mathit{val}(\rho)(\square,\square)$ after removing all nOC lock acquisitions and releases from $\mathcal{E}_A\langle\texttt{P0}\rangle$. Note that for cols. (a) and (b), the valuation is an approximation, whereas col. (c) is able to describe *Path 1* exactly within the given prefix bound.

$r = \langle p, n_{12}\rangle \hookrightarrow \langle p, e_{\text{set}}\ x_{\text{testAndSet}}\rangle$ is annotated with the weight $\{\ (\ \}$. In $\mathcal{E}_A\langle\texttt{P0}\rangle$, r is annotated with the weight

$$R = \{\square \xrightarrow{\{\ (\ \}} \boxtimes, \boxtimes \xrightarrow{\{\ (\ \}} \boxtimes\}$$

Observe that the state space of $A\langle\texttt{P0}\rangle$ is encoded in the weight, and that $R(\square,\boxtimes)$ denotes an OC lock acquisition, and $R(\boxtimes,\boxtimes)$ denotes an nOC lock acquisition. This is represented in R by annotating the two open-parenthesis symbols with the open and nested subscripts, respectively:

$$R = \{\square \xrightarrow{\{\ (_o\ \}} \boxtimes, \boxtimes \xrightarrow{\{\ (_n\ \}} \boxtimes\}$$

In other words, we perform the following transformation: For a weighted relation R, if $R(\square,\boxtimes) = \{\ (\ \}$, then $R(\square,\boxtimes) = \{\ (_o\ \}$; and if $R(\boxtimes,\boxtimes) = \{\ (\ \}$, then $R(\boxtimes,\boxtimes) = \{\ (_n\ \}$. Performing this transformation, and its dual for lock releases, on the weight of each rule and merging function induces a homomorphism, with respect to lock acquisitions and releases, from Fig. 2(a) to Fig. 2(b), on the language computed by $\text{MOVP}(\mathcal{E}_A\langle\texttt{P0}\rangle)$. This is illustrated by the weighted valuations for *Path 1* in Tab. 2, columns (a) and (b).

From Fig. 2(b) to Fig. 2(c) Once $\mathcal{E}_A\langle\texttt{P0}\rangle$ is able to distinguish between OC and nOC lock acquisitions and releases, we leverage *Observation 1* to remove all nOC lock acquisitions and releases. For a weighted relation R, if $R(\boxtimes,\boxtimes) = \{\ (_n\ \}$, then we set $R(\boxtimes,\boxtimes) = \overline{1}$. Performing this transformation, and its dual for lock releases, induces a homomorphism on the language from Fig. 2(b) to Fig. 2(c). This is exemplified by the weighted valuation of *Path 1* in Tab. 2, columns (b) and (c). Note that the valuation shown in column (c) is *not* an approximation like those in columns (a) and (b). This is because the string that describes *Path 1* is shorter after performing language strength reduction.

Removing nested parentheses that denote nOC acquisitions and releases guarantees that all lock acquisitions and releases modeled by $\mathcal{E}$ are OC. Thus, all EML locks can now be modeled in the CPDS by the trivial language shown in Fig. 2(c). Because the open and close-parenthesis symbols for nOC acquisitions and releases have been removed, a path in an EML process that uses reentrant locking can now be described by a shorter string. This can be seen in Tab. 2. In fact, there is now no cost to model a successive synchronized call, including recursive synchronized functions. Thus, in some cases, the CPDS model checker can find the same counterexample using a smaller bound k.

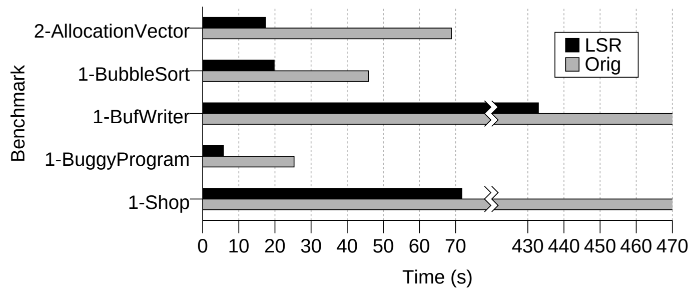

Fig. 5: Execution time for the CPDS model checker with the original encoding (Orig) and after language strength reduction (LSR).

6 Experiments

We implemented *Construction 1* and the transformations from §5 in the Empire tool. Five Java benchmark programs from the concurrency-testing benchmark by Eytani et al. [8] were analyzed. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 16 GB of memory. The machine ran a Windows XP Professional x64 Edition host OS, and an Ubuntu guest OS configured with the 32-bit Linux kernel 2.6.22. Ubuntu ran on top of VMware Server 1.0.4. A virtual machine was required because the CPDS model checker is only 32-bit Linux compatible.

Each benchmark was analyzed with the original encoding (Orig) and then again after applying language strength reduction (LSR). The analysis times are shown in Fig. 5. The Y-axis of Fig. 5 gives the benchmark names, with each name being preceded by the number of locks in the EML program (e.g., "BufWriter" uses 1 lock). For benchmark programs "AllocationVector", "BubbleSort", and "BuggyProgram", the average speedup (geometric mean) is 3.4. In addition, analysis of the benchmark programs "BufWriter" and "Shop" exhausted all resources in the original version of Empire, whereas the analysis ran to completion after performing language strength reduction.

7 Related Work

Alur and Madhusudan [5,6] introduced the concept of an NWA. For program verification, they showed that a property specification and a program can be modeled by NWAs, and that verification can be solved by taking their intersection. Our work extends this result to property checking where the program is specified by an EWPDS and the property by an NWA. Because EWPDSs allow programs to be abstracted using more than just predicate-abstraction domains (i.e., abstract programs can be more than just Boolean programs), our work has broadened the class of program abstractions for which one can use an NWA as the property specification.

Chaudhuri and Alur [13] instrument a `C` program with an NWA that defines a property specification. This approach diffuses the NWA throughout the program proper. Our approach combines the NWA with an EWPDS, but keeps the NWA separated by modeling it using weights. This is beneficial for reporting error-paths back to a user when model checking a `C` program because the internals of the NWA are not exposed in the error-path. Additionally, by keeping the NWA separated in the weight domain, one can use symbolic encoding of weights [9] for handling the potentially exponential size of the NWA.

Kahlon et al. [14,15] analyze concurrent recursive programs that use nested locking, where nested locking means that all locks are released in the opposite order in which they are acquired. Their locks, however, are not reentrant and are not syntactically scoped. If one enforces syntactically scoped locks, then one can apply our techniques for language strength reduction to model a program with reentrant locks using only non-reentrant locks. This would produce a model to which their model-checking algorithm could be applied.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL. (2006)
2. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Static detection of atomic-set serializability violations. Technical Report TR-1623, Univ. of Wisconsin (October 2007)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
4. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: TACAS. (2006)
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC. (2004)
6. Alur, R., Madhusudan, P.: Adding nesting structure to words. In: DLT. (2006)
7. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: CAV. (2005)
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. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
10. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP (2005)
11. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: CAD. (1993)
12. Kidd, N., Lal, A., Reps, T.: Advanced queries for property checking. Technical Report TR-1621, Univ. of Wisconsin (October 2007)
13. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: SPIN. (2007)
14. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV. (2005)
15. Kahlon, V., Yang, Y., Sankaranarayan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: CAV. (2005)