Beyond Regular Model Checking (Extended Abstract) *

Dana Fisman and Amir Pnueli

Dept of Computer Science, The Weizmann Institute of Science, Rehovot, Israel.
{dana,amir}@wisdom.weizmann.ac.il

Abstract. In recent years, it has been established that *regular model checking* can be successfully applied to several parameterized verification problems. However, there are many parameterized verification problems that cannot be described by regular languages, and thus cannot be verified using regular model checking. In this study we try to practice symbolic model checking using classes of languages more expressive than the regular languages. We provide three methods for the uniform verification of non-regular parameterized systems.

1 Introduction

During the last two decades, several formal methods have been developed to answer the *verification problem* of finite-state systems. The verification problem asks the question of whether a given reactive system is correct relative to some specification. Although many interesting concurrent programs are in fact finite state, they are often given semantically in terms of a parameter n, representing the number of concurrent processes. Such a schematic program really represents an infinite family of uniformly defined programs. Such programs are often referred to as *parameterized systems.* A challenging problem is to provide methods for the *uniform* verification of parameterized systems, i.e., proving correctness for all possible programs obtained by instantiating the parameter. We refer to this problem as the *parameterized verification problem.* In 1986 Apt and Kozen proved that, in general, the parameterized verification problem is undecidable, even when each instance is finite-state [3]. However, for specific families the problem may be solvable.

Model Checking (MC) is an automatic technique for answering the verification problem. In this framework, specification are usually expressed by a propositional temporal logic and programs are modeled by state-transition systems. The model checking procedure performs an exhaustive search of the state space of the system to determine whether the system satisfies the specification. The use of an exhaustive state-space exploration limits the application of model checking to finite-state systems.

However, it is possible to perform a state-space exploration of infinite-state systems as well, by using an implicit representation for sets of states. The framework of model checking where sets of states are represented implicitly using some

* This work was supported in part by the European Commission (FET project ADVANCE, contract No IST-1999-29082), and carried out at the John von Neumann Minerva Center for the Verification of Reactive Systems.

symbolic representation is known as *symbolic model checking (SMC)* [11]. Symbolic model checking can be applied to infinite-state systems, although in this case, the termination of the procedure is not guaranteed.

Regular model checking is an application of symbolic model checking where regular expressions are used to represent symbolically sets of states [21]. Regular model checking can be applied to any verification problem that is expressible using regular languages. Such an application is successful if the regular model checking procedure has terminated. In recent years, it has been established that regular model checking can be successfully applied to several types of parameterized verification problems.

However, many interesting parameterized systems cannot be handled by regular model checking since the class of regular languages is not strong enough to express them. An example for such a verification problem is the Peterson algorithm for mutual exclusion among n processes [23]. The existence of such examples, is the main motivation for this study.

In this study we try to practice symbolic model checking using as symbolic representation classes of languages more expressive than the regulars. As a first attempt, we use context-free languages at the last step of the symbolic model checking procedure, while regular languages are used in the procedure until this last step. This seemingly slight change, already enables us to verify mutual exclusion for the Peterson algorithm.

By carefully examining the model checking procedure, one can compile a list of the requirements a class of languages must meet in order to be *adequate* for symbolic model checking [21]. Such a list consists of several operations the class must be effectively closed under, and several questions that must be effectively decidable for the class. We recognize that the class of languages accepted by deterministic pushdown automata $\mathcal{L}_{DPDA}$, meets all requirements but two: the class is not closed under projection and there is no known efficient algorithm to decide equivalence. We thus direct our effort to find a class, which is a subset of the class $\mathcal{L}_{DPDA}$, and possesses efficient algorithms for computing projection and deciding equivalence. This class must also satisfy the rest of the requirements.

We succeed to define a sub-class of $\mathcal{L}_{DPDA}$, which we denote $\mathcal{L}_{DPDA-M}$, for which there exists a semi-algorithm[1] to compute projection. In addition, there exists an efficient algorithm to answer the equivalence problem for this class. This class also satisfies all other requirements. Thus, we establish a class, which is more expressive than the class of regular languages, and yet is adequate for symbolic model checking. The Peterson example can be symbolically model checked, using languages in this class. Note that the notion of adequacy achieved here is weaker than the one introduced in [21] because termination failure can also occur in the computation of projection. However, due to the general undecidability of the problem, a semi-algorithm is the best we can hope for in any case.

[1] By semi-algorithm we mean a computational procedure that is not guaranteed to halt, but is guaranteed to give a correct answer in all cases at which it does halt. In practice, we use semi-algorithms by running them up to a prescribed time limit.

Recall that the standard fix-point computation in the symbolic model checking procedure is not guaranteed to terminate when applied to infinite-state systems. This difficulty must be addressed when considering a class of languages to be adequate for symbolic model checking. In order for such a class to be *practically* adequate for symbolic model checking, it must also provide means to tackle this difficulty. Indeed, for regular model checking, many techniques to overcome this problem have been developed (see related work section). The common idea behind these techniques lies in calculating the effect of taking an arbitrary number of system-transitions in one step, often refer to as calculating meta-transitions or "accelerations".

For a special case of the class $\mathcal{L}_{DPDA-M}$ there exists an algorithm to compute projection. In addition, all techniques developed for calculating meta-transitions for regular model checking, can be applied to this (sub-)class. This class is therefore *practically* adequate for symbolic model checking. The Peterson example can also be verified using languages in this class.

Due to space limitation, further details and proofs are omitted, these can be found in the full version of the paper.[2]

Related Work Regular model checking has been advocated by [21] and [30] as a uniform paradigm for algorithmic verification of several classes of parameterized and infinite-state systems. The use of regular languages to express state properties goes back to [13]. The problem of calculating meta-transitions is one of the most laborious problems in this field of research. It has been thoroughly researched [1, 2, 18], resulting in several corrective techniques, such as acceleration [24, 4], calculation of the transitive closure [20, 9] and widening [22, 9, 15].

The study of model checking pushdown systems has recently been receiving growing attention [12, 5, 10, 6, 7, 30, 16]. Nevertheless, these efforts do not pertain to this study for two reasons: first, we use the pushdown automata to represent the set of states, while the above work considers the pushdown automata as the system being analyzed. Furthermore, a fundamental difference is that all systems previously considered share a common characteristic: a regular language represents their state space, whereas our main interest is systems whose state space cannot be represented by a regular language.

There has been other works also pursuing symbolic representations which are more expressive than regular languages. Studies by Boigelot and Wolper [29] and Comon and Jurski [14] give symbolic representations for configurations of systems with counters.

Perhaps the work most related to this research is by Bouajjani and Habermehl [8]. They define a symbolic representation, denoted CQDD, which is an extension of the QDDs defined by [5]. While QDDs are finite automata, CQDDs are a combination of restricted finite automata and linear constrains on the number of occurrences of symbols. The CQDDs are a symbolic representation which is more expressive than regular languages. As an example, they show that CQDDs accept the language $a^n b^m a^n b^m$. However, to our understanding, the CQDDs are not strong enough to express the Peterson example, which we were able to verify using all our methods.

2 Preliminaries

Definition 1. Let Σ be an alphabet. A *bi-letter* over Σ is an element of $\Sigma \times \Sigma$. A *bi-word* over Σ is a string of bi-letters over Σ (i.e., an element of $(\Sigma \times \Sigma)^*$).

[2] the full version of the paper can be found in `www.wisdom.weizmann.ac.il/~dana`

A *bi-language* over Σ is a set of bi-words over Σ (i.e., a subset of $(\Sigma \times \Sigma)^*$). We use the notations $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right]$ to denote respectively the bi-letter (a, b) and the bi-word $\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix}\right]\left[\begin{smallmatrix} a_2 \\ b_2 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} a_n \\ b_n \end{smallmatrix}\right]$ where $u = a_1 a_2 ... a_n$ and $v = b_1 b_2 ... b_n$.

Given a bi-language L over Σ, we denote by $L\Downarrow_1$ ($L\Downarrow_2$) the *projection* of L on the first (respectively, second) coordinate. Given a language L over Σ, we denote by $L \times \Sigma^*$ the language $\left\{ \left[\begin{smallmatrix} w \\ u \end{smallmatrix}\right] \;:\; w \in L,\, u \in \Sigma^*,\, |u| = |v| \right\}$, referred to as *left lifting*. Similarly, we define *right lifting* and use the notation $\Sigma^* \times L$. We use the standard notations $\overline{L}$ and $L_1 \cap L_2$ to denote the complement of a language and the intersection of two languages respectively.

Definition 2. A *system* is a quadruple $(\Sigma, \chi, \Theta, \rho)$ where

- Σ is a finite alphabet.
- $\chi \subseteq \Sigma^*$ is a language over Σ, denoting the set of *states*.
- $\Theta \subseteq \chi$ is a language over Σ, denoting the set of *initial states*.
- $\rho \subseteq \chi \times \chi$ is a bi-language over Σ, denoting the *transition relation*.

2.1 Symbolic Model Checking and Regular Model Checking

The *verification problem* for a system M and a property φ is to decide whether φ holds over all computations of M. *Model checking* is an automatic technique for answering the verification problem. In *symbolic model checking (SMC)*, some symbolic representation $\mathcal{C}$ is used to represent sets of states. In the framework of symbolic model checking using the representation $\mathcal{C}$, the system and the property to be verified are represented as expressions in $\mathcal{C}$: The system is defined using $\mathcal{C}$-expressions to describe its components, which are essentially sets of states or relations over sets of states. The invariance property is defined using some expression in $\mathcal{C}$ to describe the sets of states satisfying it. Then, the model checking procedures operate by manipulating sets of states (instead of individual states) through operations on the expressions of $\mathcal{C}$. Symbolic model checking can be applied to infinite-state systems, although in this case, the termination of the procedure is not guaranteed. We refer to this problem as the *convergence problem*.

Regular model checking [21, 30] is an application of symbolic model checking, where regular expressions are used as the (symbolic) assertional language for describing sets of states. Using this framework, the system's components, and the property to be verified are defined by means of regular expressions. Usually, we assume a given alphabet Σ to represent a local state. The set of global states χ, the set of initial states Θ and the property φ, are specified by a regular expression (defining a regular language) over Σ. The transition relation ρ is specified by a regular expression using bi-letters over Σ (sometimes referred to as a bi-regular expression).

Example 1. (TOKEN-STRING)
Consider an array of processes that pass a token from left to right. We define the alphabet $\Sigma = \{1, 0\}$ to denote the local state of a process, i.e., that the process has or has not the token. A global state of a system consisting of n processes

is defined by a word of length n, each letter describes the state of one process. The set of global states χ is defined to be the language of all words of positive length, given by the regular expression $(0+1)^+$. The set of initial states Θ, is given by the regular expression 10^*, indicating that the leftmost process holds the token. The transition relation ρ can be specified by the bi-regular expression $\left(\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]+\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\right)^* \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] \left(\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]+\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\right)^*$ defined over $\Sigma = \{0,1\}$. Alternatively, we can specify the transition relation by the length-preserving rewrite rule $x\ 1\ 0\ y \rightarrow x\ 0\ 1\ y$ (where x and y are arbitrary words over $\{0,1\}^*$). The invariance property stating that there is exactly one token at all times, can be given by $\varphi = 0^*\ 1\ 0^*$.

3 Adequacy of Classes of Automata for SMC

Regular model checking is an important application of symbolic model checking which enables the verification of several classes of parameterized and infinite-state systems. Nevertheless, there are instances where a regular language is not expressive enough to describe the system or the property at hand. Our aim is to find a class of languages that is more expressive than the regular languages, yet is adequate for symbolic model checking. Naturally, we are looking for the largest class of languages, which are still amenable to symbolic model checking.

The *rich-language symbolic model checking methodology* described in [21] lists a set of minimal requirements from an assertional language, in order for it to be adequate for symbolic model checking. We reformulate these requirements here in terms of classes of languages rather than assertions. We classify the languages involved in the symbolic model process, according to the operations applied to them. Our intention is to allow deployment of different classes of languages within the symbolic model checking process. We present *backward* and *forward model checking procedures* in terms of this classification.

Let $\mathcal{M}$, $\mathcal{R}$ and $\mathcal{A}$ be three classes of languages. Let $M_\varphi \in \mathcal{M}$ and $A_\varphi \in \mathcal{A}$ be languages adequate for specifying the property to be verified. Let $A_\Theta \in \mathcal{A}$ and $M_\Theta \in \mathcal{M}$ be languages adequate for representing the initial state of the system. Let $R_\rho \in \mathcal{R}$ be a bi-language adequate for representing the transition relation of the system, augmented by the identity relation (idle transition). We will use the auxiliary languages $M_0, M_1, M_2, \ldots \in \mathcal{M}$ to represent system states. The following procedures describe backward and forward model checking:

Procedure *Backward MC*
  $M_0 := \overline{M_\varphi}$
  For $i = 0, 1, \ldots$ **repeat**
    $M_{i+1} := ((\Sigma^* \times M_i) \cap R_\rho)\Downarrow_1$
  until $M_{i+1} = M_i$
return $M_i \cap A_\Theta = \emptyset$

Procedure *Forward MC*
  $M_0 := M_\Theta$
  For $i = 0, 1, \ldots$ **repeat**
    $M_{i+1} := ((M_i \times \Sigma^*) \cap R_\rho)\Downarrow_2$
  until $M_{i+1} = M_i$
return $M_i \cap \overline{A_\varphi} = \emptyset$

The classes $\mathcal{M}$, $\mathcal{R}$ and $\mathcal{A}$ are *adequate* for symbolic model checking, if the following requirements hold:

1. $\mathcal{R}$ is adequate for representing ρ, and either $\mathcal{M}$ and $\mathcal{A}$ are adequate for specifying φ and Θ respectively (for *Backward MC*) or $\mathcal{M}$ and $\mathcal{A}$ are adequate for specifying Θ and φ respectively (for *Forward MC*).

2. Either $\mathcal{M}$ is effectively closed under complementation (for *Backward MC*) or $\mathcal{A}$ is effectively closed under complementation (for *Forward MC*).
3. $\mathcal{M}$ is effectively closed under lifting.
4. $\mathcal{M}$ is effectively closed under intersection with $\mathcal{R}$.
5. $\mathcal{M}$ is effectively closed under projection.
6. Either $\mathcal{M}$ is effectively closed under intersection with $\mathcal{A}$ and emptiness is effectively decidable for $\mathcal{M}$, or $\mathcal{A}$ is effectively closed under intersection with $\mathcal{M}$ and emptiness is effectively decidable for $\mathcal{A}$.
7. Equivalence of two languages in $\mathcal{M}$ is effectively decidable.

Assuming that ρ always includes the identity relation (implementing the idling step), it is unnecessary to require closure under union (as required in [21]).

3.1 Meeting the Requirements

The following combinations appear to meet all the requirements:

1. Taking $\mathcal{M} = \mathcal{R} = \mathcal{A}$ to be the class of regular languages $\mathcal{L}_{FA}$. This leads to regular model checking.
2. Taking $\mathcal{M} = \mathcal{R} = \mathcal{L}_{FA}$ and letting $\mathcal{A}$ be the class of languages accepted by deterministic pushdown automata $\mathcal{L}_{DPDA}$. This combination is demonstrated in Section 4 by verifying Peterson's mutual exclusion algorithm for an arbitrary number of processes.

A more challenging step is considering $\mathcal{M}$ to be the $\mathcal{L}_{DPDA}$ class, leaving the $\mathcal{L}_{FA}$ class for $\mathcal{R}$ and $\mathcal{A}$. Requirement 3 is met as $\mathcal{L}_{DPDA}$ is closed under inverse homomorphism, which is a generalization of lifting. Requirements 2, 4 and 6 are met as $\mathcal{L}_{DPDA}$ is closed under complementation and intersection with a regular language, and the emptiness problem for $\mathcal{L}_{DPDA}$ is decidable. However, requirement 5 is not met: $\mathcal{L}_{DPDA}$ is not closed under projection. Requirement 7, decidability of the equivalence problem, was an open question until recently. In 1997 it was proven positively by Senizergues [26]. However, the algorithm he provides is not effective [28].

We thus concentrate our efforts to find a sub-class of $\mathcal{L}_{DPDA}$ which is adequate for symbolic model checking. Therefore, we must find effective algorithms to compute projection and to decide equivalence for languages in this class. In addition, this class must satisfy the rest of the closure and decidability properties discussed above. This is the topic of Section 5.

4 SMC using Context-free and Regular Languages

In this section we concentrate on the framework of symbolic model checking where the class $\mathcal{A}$ is chosen to be the $\mathcal{L}_{DPDA}$ class, while $\mathcal{M}$ and $\mathcal{R}$ are the $\mathcal{L}_{FA}$ class. If we choose to apply backward model checking we can describe the property to be verified by some language in the $\mathcal{L}_{DPDA}$ class, but then we have to compromise with a regular language for describing the initial condition. Alternatively, we can choose to apply forward model checking, which results in the initial condition being in $\mathcal{L}_{DPDA}$, while the property must be described by a regular language.

4.1 Peterson's Algorithm

We focus on the verification of the Peterson algorithm for mutual exclusion among N processes presented in the figure below.

The Peterson algorithm can be explained as follows. Each process $P[i]$ has a priority variable $y[i]$. The range of the priority variable are the numbers from 0 to $N-1$. In addition there are $N-1$ signature variables $s[1], s[2], \ldots s[N-1]$. The domain of the signature variables consists of the processes indices $1, 2, ..., N$. The variable $s[j]$ holds the signature (index) of the last process that received priority j. Assume process $P[i]$ has priority j. In order to increment its priority (to $j+1$), it must be either the process with the highest priority, or not the last who signed in the signature $s[j]$. A process can enter the critical section if it has the highest priority $(N-1)$ and it is <u>not</u> the last who signed in (in the signature of the highest priority, $s[N-1]$). When a process exits the critical section its priority is reset.

$$
\begin{array}{l}
N : \textbf{natural initially } N > 1 \\
y \ \ : \textbf{array } [1..N] \textbf{ of } [0..N-1] \textbf{ initially } y = 0 \\
s \ \ : \textbf{array } [1..N-1] \textbf{ of } [1..N]
\end{array}
$$

$$
\mathop{\Big\|}_{i=1}^{N} P[i] :: \left[
\begin{array}{l}
t : \textbf{integer} \\
\ell_0\text{:}\ \ \textbf{loop forever do} \\
\quad \left[
\begin{array}{l}
\ell_1\text{:}\ \ \textbf{Non-Critical} \\
\ell_2\text{:}\ \ \textbf{for } t := 1 \textbf{ to } N-1 \textbf{ do} \\
\quad \left[
\begin{array}{l}
\ell_3\text{:}\ \ (y[i], s[t]) := (t, i) \\
\ell_4\text{:}\ \ \textbf{await } s[t] \neq i \ \vee \ \forall j \neq i : y[j] < y[i]
\end{array}
\right] \\
\ell_5\text{:}\ \ \textbf{Critical} \\
\ell_6\text{:}\ \ y[i] := 0
\end{array}
\right]
\end{array}
\right]
$$

Program Peterson(N).

4.2 Verification of Peterson's Algorithm

In order to apply our method, we must first model the system by languages as defined in Definition 2. We encode a state of the Peterson system as a word of the form

$$
\underbrace{\bigcirc \cdots \bigcirc}_{y=0} \mid \underbrace{\overset{\overset{s[1]}{\downarrow}}{\bigcirc} \cdots \bigcirc}_{y=1} \mid \underbrace{\overset{\overset{s[2]}{\downarrow}}{\bigcirc} \cdots \bigcirc}_{y=2} \mid \ \cdots \ \mid \underbrace{\overset{\overset{s[N-1]}{\downarrow}}{\bigcirc} \cdots \bigcirc}_{y=N-1} \mid \underbrace{\bigcirc \cdots \bigcirc}_{y=N}
$$

where all processes $P[i]$ within partition $k = 0, 1, \ldots, N-1$ have their priority variable $y[i]$ set to k. The leftmost process in partition k is the one signed in the signature $s[k]$. Processes in the rightmost partition are the ones which are in the critical section. Note that there are $N+1$ partitions, separated by N border markers. The set of global states can thus be given by the language $\chi = \{w \in (\bigcirc + \mid)^* : \ \ \sharp(\mid, w) = \sharp(\bigcirc, w) > 1\}$. This is the language over the alphabet $\{\bigcirc, \mid\}$, each of whose words has an equal number of "$\mid$" and "$\bigcirc$".

The initial state can be described by the regular expression $\Theta : \quad \bigcirc^* \mid^*$, in which all processes have their priority variable y set to 0 and thus are located in the first partition.

The transition relation ρ can be specified using five length-preserving rewrite rules: $\rho_1, \rho_2, \rho_3, \rho_4$ and ρ_{id} defined as follows:

$$
\begin{array}{lcll}
\rho_1 & : & x \bigcirc \mid y \mapsto x \mid \bigcirc y & \text{where } x \in \bigcirc^*, y \in (\mid + \bigcirc)^* \\
\rho_2 & : & x \mid \bigcirc \mid y \mapsto x \mid \mid \bigcirc y & \text{where } x \in (\mid + \bigcirc)^*, y \in \mid^* \\
\rho_3 & : & x \bigcirc \bigcirc \mid y \mapsto x \bigcirc \mid \bigcirc y & \text{where } x, y \in (\mid + \bigcirc)^* \\
\rho_4 & : & x \bigcirc \mapsto \bigcirc x & \text{where } x \in (\mid + \bigcirc)^* \\
\rho_{id} & : & x \mapsto x & \text{where } x \in (\mid + \bigcirc)^*
\end{array}
$$

The first rewrite rule states that a process can move unconditionally from the first partition to the second partition. This corresponds to increasing the priority variable y from 0 to 1. The second rule allows a single process within a partition can move to the next partition provided that all the partitions to it's right are empty. This corresponds to the situation where the process has the highest priority. The third rule allows a process to move to the next partition provided that there is a process to its left in the same partition. This corresponds to the situation where the process is not the last to sign in the signature corresponding to its current priority. The fourth rule describes an exit from the critical section,while the last rule captures the stuttering step.

Performing forward regular model checking, we obtain the following set of reachable states:

$$R : \qquad \bigcirc^* \mid (\bigcirc^+ \mid)^* \bigcirc^* \mid^* \quad + \quad \bigcirc^* \mid (\bigcirc^+ \mid)^* \mid^+ \bigcirc \mid^*.$$

The interpretation being: to the right of any empty partition $(\mid \mid)$ there can be at most one process $(\bigcirc)$. The negation of mutual exclusion is described by $x \bigcirc \bigcirc$, which represents the situation in which there are at least two processes in the critical section. This has a non-empty intersection with the set of reachable states, given by: $\bigcirc^* \mid (\bigcirc^+ \mid)^* \bigcirc^* \bigcirc \bigcirc$. Thus, we are unable to verify the Peterson algorithm using purely regular model checking.

However, when we express the negation of the property by an $\mathcal{L}_{DPDA}$-language which can be described by the intersection $(x \bigcirc \bigcirc) \cap \chi$, where χ requires an equal number of sticks and stones, we obtain an empty intersection. Thus, by mixing regular with context-free languages, we are able to verify mutual exclusion for the Peterson algorithm for an arbitrary number of processors, whereas using purely regular model checking, we failed to do so. An elaborated example and further details can be found in the full paper.[3]

5 SMC using a Cascade Product 1DPDA ∘ DFA

In this section we seek to find a subset of $\mathcal{L}_{DPDA}$ which is adequate for symbolic model checking. Given a single state DPDA M we define the class $\mathcal{L}_{DPDA-M}$. We

[3] the full version of the paper can be found in `www.wisdom.weizmann.ac.il/~dana`

provide a semi-algorithm for computing projection for this class, and an efficient algorithm to decide equivalence. This class, in addition, is closed under all basic operations required in order to be adequate for symbolic model checking.

5.1 The Classes $\mathcal{L}_{DPDA-M}$

We concentrate on deterministic pushdown automata with no epsilon transitions, DPDAs [19]. We often consider single-state DPDAs with an empty set of accepting conditions, to which we refer as a 1DPDA. A 1DPDA $M = \langle \Sigma, \{q\}, q, \Gamma, \perp, \Delta, \emptyset \rangle$ can be represented by the quadruple $\langle \Sigma, \Gamma, \perp, \widehat{\Delta} \rangle$ where $\widehat{\Delta} : \Sigma \times \Gamma \rightarrow Com(\Gamma)$. The following definitions are needed for the sequel.

Definition 3. (*cascade product*)
Let $R = \langle V \times \Gamma, S, s_0, \delta, F \rangle$ be a DFA and $\phi : V \rightarrow \Sigma$ a substitution mapping each letter of V to a letter in Σ. The *cascade product* $M \circ_\phi R$ is the DPDA $\langle V, S, s_0, \Gamma, \perp, \rho, F \rangle$ where $\rho(s, \sigma, z) = (\delta(s, \langle \sigma, z \rangle), \Delta(\phi(\sigma), z))$.

Definition 4. (*stack-consistent with M*)
A DPDA A is said to be *stack-consistent with M* (*M-consistent*), if there exist a substitution $\phi : V \rightarrow \Sigma$ and a DFA $R = \langle V \times \Gamma, S, s_0, \delta, F \rangle$ such that $A = M \circ_\phi R$. We say that A is *M-consistent w.r.t* (with repect to) ϕ.

Let $A = M \circ_\phi R$. A run π_A of A on the word $w = \sigma_1 \sigma_2 \cdots \sigma_n$ can be decomposed into two runs: a run π_M of M on $\phi(w)$, and a run π_R of R on w:

$$\begin{array}{lccccccc}
\pi_A : & (s_0, \gamma_0) & \overset{\sigma_1/x_1}{\longmapsto} & (s_1, \gamma_1) & \overset{\sigma_2/x_2}{\longmapsto} & \cdots & \overset{\sigma_n/x_n}{\longmapsto} & (s_n, \gamma_n) \\
\pi_M : & \gamma_0 & \overset{\phi(\sigma_1)/x_1}{\longmapsto} & \gamma_1 & \overset{\phi(\sigma_2)/x_2}{\longmapsto} & \cdots & \overset{\phi(\sigma_n)/x_n}{\longmapsto} & \gamma_n \\
\pi_R : & s_0 & \overset{\sigma_1, \top(\gamma_0)}{\longmapsto} & s_1 & \overset{\sigma_2, \top(\gamma_1)}{\longmapsto} & \cdots & \overset{\sigma_n, \top(\gamma_{n-1})}{\longmapsto} & s_n
\end{array}$$

The decision upon the stack command at the i-th step x_i is governed only by the 1DPDA M, which is not aware of the state s_i of R. Yet, the DFA R can look at the top symbol on the stack of M ($\top(\gamma)$ denotes the top symbol of stack γ). Automata R and M move simultaneously: when A reads a letter $\sigma_i \in V$, the DFA R makes a move as if it was reading the pair $(\sigma_i, \top(\gamma_{i-1}))$, where γ_{i-1} is the current stack of M, while M makes a move as if it was reading $\phi(\sigma_i)$. The automaton A accepts a word w if, when the run terminates, R is in state $s_n \in F$.

An automaton $A = \langle V, S, s_0, \Gamma, \perp, \rho, F \rangle$ which is M-consistent w.r.t ϕ can be characterized by the pair $\langle R, \phi \rangle$ where R is the DFA such that $A = M \circ_\phi R$.

Definition 5. (*The M-stack consistent class,* $\mathcal{L}_{DPDA-M}$)
Given a 1DPDA M, we define the M-consistent class $\mathcal{L}_{DPDA-M}$ to be the class of languages that are accepted by some M-consistent DPDA.

The M-consistent automaton A can be viewed as a case in which the automaton has been decomposed into a stack-manipulator, which behaves exactly like M in its decisions about the stack transformations, and a finite-state controller R which affects the selection of the next state. Two M-consistent automata A_1 and A_2 share the same stack-manipulator and may differ in their respective finite-state controllers, as well as in their respective substitutions.

Claim. The class $\mathcal{L}_{DPDA-M}$ is *effectively closed* under complementation, lifting and intersection with a regular language.

Claim. Equivalence and Emptiness are *effectively decidable* for the class $\mathcal{L}_{DPDA-M}$.

5.2 Computing Projection

Viewing the claims above, for any 1DPDA M the class $\mathcal{L}_{DPDA-M}$ satisfies requirements 1,2,3,4,6 and 7. Thus, if the class is also effectively closed under projection, then it is adequate for symbolic model checking (in collaboration with the $\mathcal{L}_{FA}$ class). Below, we provide a semi-algorithm for computing projections.

Let A be a DPDA which is M-consistent w.r.t ϕ, and let R be A's characteristic DFA. For simplicity we assume the input alphabet of A is $\Sigma \times \Sigma$, where Σ is the input alphabet of M. Also, we assume A is M-consistent w.r.t $\phi = \Downarrow_2$ and we wish to calculate the projection of $\mathcal{L}(A)$ on its first coordinate. That is, we would like to compute another DPDA $\widetilde{A}$ over the alphabet Σ which is M-consistent w.r.t the identity relation and accepts the projection of A on its first coordinate.

We claim that procedure *Project* described below calculates the correct projection (unless it aborts). The procedure can be explained as follows. To simplify notations, we present ρ as a relation, a subset of $S \times \Sigma \times \Gamma \times Com(\Gamma) \times S$ instead of a function from $S \times \Sigma \times \Gamma$ to $S \times Com(\Gamma)$.

Procedure *Project*
Input: a DPDA $A = \langle \Sigma \times \Sigma, S, s_0, \Gamma, \perp, \rho, F \rangle$, a positive integer k
Output: a DPDA $\widetilde{A} = \langle \Sigma, \widetilde{S}, \widetilde{s_0}, \Gamma, \perp, \widetilde{\rho}, \widetilde{F} \rangle$

1. *Annotation*:
   $\widehat{A} = \langle \Sigma \times \Sigma, \widehat{S}, \widehat{s_0}, \Gamma, \perp, \widehat{\rho}, \widehat{F} \rangle := Annotate(A, k)$
2. *Projection*:
   $\ddot{A} := \langle \Sigma, \ddot{S}, \ddot{s_0}, \Gamma, \perp, \ddot{\rho}, \ddot{F} \rangle$ where $\ddot{S} = \widehat{S}$, $\ddot{s_0} = \widehat{s_0}$, $\ddot{F} = \widehat{F}$ and
   $\ddot{\rho} = \{(s, z_1, \sigma_1, x_1, s') \ : \ \exists z_2, \sigma_2, x_2 : \ (s, z_1, z_2, \sigma_1, \sigma_2, x_1, x_2, s') \in \widehat{\rho}\}$
3. *Determinization*:
   $\ddot{R} := \langle \Sigma \times \Gamma, \ddot{S}, \ddot{s_0}, \ddot{\delta}, \ddot{F} \rangle$ where $\ddot{\delta}(s, (\sigma, z)) = s' \iff \exists x : \ (s, z, \sigma, x, s') \in \ddot{\rho}$
   $\widetilde{R} = \langle \Sigma \times \Gamma, \widetilde{S}, \widetilde{s}, \widetilde{\delta}, \widetilde{F} \rangle := Subset\_Construction(\ddot{R})$

Return $\widetilde{A} := M \circ_{id} \widetilde{R}$

In Phase 1 procedure *Project* calls procedure *Annotate* described below. Procedure *Annotate* guesses how the 1DPDA M will operate when looking at the first coordinate instead of the second. For each edge (s, s') labeled by $(\left[\begin{smallmatrix}\sigma_1\\ \sigma_2\end{smallmatrix}\right], z_2, x_2)$[4] the procedure aims to find all possible stack letters z_1 and stack commands x_1 such that if M looks at the first coordinate then, when moving from state s to s', it will have z_1 on the top of the stack and decide on the stack command x_1.

For each state, it saves an information describing the difference between the actual stack of M and the "guessed stack" (the stack of M if it was looking at the first coordinate). For this it uses the notation $\langle \beta_1, \beta_2 \rangle$ with the intention that if the maximal common prefix of the actual stack and the guessed stack is w then the "guessed stack" is described by $w\beta_1$ and the actual stack is described

[4] If $(s, z, x, \sigma, s') \in \rho$ we say that the edge (s, s') is labeled by (σ, z, x).

by $w\beta_2$. An original state s of A may appear more than once in $\widehat{A}$, each time labeled with a different notation $\langle\beta_1,\beta_2\rangle$.

Note that z_2 must be the top symbol of $w\beta_2$. The procedure will choose z_1 to be the top symbol of $w\beta_1$. The stack command x_1 is then determined by $\Delta(\sigma_1, z_1)$. Assuming that the difference between the guessed stack and actual stack in state s is $\langle\beta_1,\beta_2\rangle$, and the stack commands for the guessed and actual stack are x_1 and x_2, we can compute the new difference $\langle\beta_1',\beta_2'\rangle$ for state s'. Let γ_1 and γ_2 be the result of applying x_1 and x_2 to the stack contents $w\beta_1$ and $w\beta_2$ respectively. Given w' is the maximal common prefix of γ_1 and γ_2, then $\beta_1' = \gamma_1/w'$ and $\beta_2' = \gamma_2/w'$ (where u/v denotes the right division of u by v).

Procedure ***Annotate***
Input: $A = \langle\Sigma\times\Sigma, S, s_0, \Gamma, \perp, \rho, F\rangle$, a positive integer k
Output: $\widehat{A} = \langle\Sigma\times\Sigma, \widehat{S}, \widehat{s_0}, \Gamma, \perp, \widehat{\rho}, \widehat{F}\rangle$ where $\widehat{S}\subseteq S\times(\Gamma^{\leq k}\times\Gamma^{\leq k})$ and $\widehat{\rho}\subseteq S\times\Sigma^2\times\Gamma^2\times Com(\Gamma)^2\times S$.
$\widehat{s_0} := (s_0,\langle\epsilon,\epsilon\rangle)$; $\quad f\mathsf{reg}(\widehat{s_0}) := Compute\_Stack\_Language(A, s_0)$
$\widehat{\rho} := \emptyset$; $\quad \widehat{S} := \{\widehat{s_0}\}$; $\quad Q := \{\widehat{s_0}\}$
While $Q\neq\emptyset$ **do**
Pick $\widehat{s} = (s,\langle\beta_1,\beta_2\rangle)\in Q$
For all $(r, r', z_1, z_2, x_1, x_2, \sigma_1, \sigma_2, \beta_1, \beta_2, \beta_1', \beta_2')\in \mathit{ReachableTransitions}$ s.t. $(s,\sigma_1,\sigma_2,z_2,x_2,s')\in\rho$ and $r = f\mathsf{reg}(s)$ **repeat**
If $|\beta_1'|\geq k\ \vee\ |\beta_2'|\geq k$ **then abort**
$\widehat{s'} := (s',\langle\beta_1',\beta_2'\rangle)$
If $\widehat{s'}\notin\widehat{S}$ **then** $f\mathsf{reg}(\widehat{s'}) := r'$ **else** $f\mathsf{reg}(\widehat{s'}) := f\mathsf{reg}(\widehat{s'})\cup r'$
$\widehat{\rho} := \widehat{\rho}\cup\{(\widehat{s}, z_1, z_2, \sigma_1, \sigma_2, x_1, x_2, \widehat{s'})\}$; $\quad \widehat{S} := \widehat{S}\cup\{\widehat{s'}\}$; $\quad Q := Q\cup\{\widehat{s'}\}$
end for all
$Q := Q\setminus\{\widehat{s}\}$
end while
$\widehat{s_0} := \{(s,\langle\beta_1,\beta_2\rangle)\ :\ s = s_0\}$; $\quad \widehat{F} := \{(s,\langle\beta_1,\beta_2\rangle)\ :\ s\in F\}$
end procedure

To guarantee termination, the procedure uses the second parameter – a bound $k\in N$. The procedure aborts if the length of either β_1 or β_2 exceeds k. It may be the case that the procedure decides to abort when exploring an *unreachable* edge. We say that an edge (s,s') labeled by $(\left[\begin{smallmatrix}\sigma_1\\ \sigma_2\end{smallmatrix}\right], z_2, x_2)$ where s is annotated $\langle\beta_1,\beta_2\rangle$ is unreachable if there is no prefix w such that $w\beta_2$ is in the reachable stack language of s. To avoid this situation, the procedure computes for each state s the reachable stack language of s. It uses a labeling function $f\mathsf{reg}$ that labels each state with its reachable stack language, $\mathcal{L}(s)$. When exploring the edge (s,s') the procedure checks that $w\beta_2$ is in the reachable stack language of s, given by $r = f\mathsf{reg}(s)$. The procedure updates the reachable stack language of s', $f\mathsf{reg}(s')$, to contain the set of words obtained by applying x_2 to a word in r whose top symbol is z_2.

The complete relation between all involved components, is summarized as follows. Let $z_1, z_2\in\Gamma$, $\sigma_1,\sigma_2\in\Sigma$, $\beta_1,\beta_2,\beta_1',\beta_2'\in\Gamma^*$, $x_1,x_2\in Com(\Gamma)$,

$r, r' \in \mathcal{R}(\Gamma)$. $(r, r', z_1, z_2, x_1, x_2, \sigma_1, \sigma_2, \beta_1, \beta_2, \beta_1', \beta_2') \in$ *ReachableTransitions* if and only if the following holds:

1. $\exists w \in \Gamma^*$ such that $w\beta_2 \in r$
2. $z_1 = \top(w\beta_1)$ and $z_2 = \top(w\beta_2)$
3. $x_1 = \Delta(\sigma_1, z_1)$ and $x_2 = \Delta(\sigma_2, z_2)$
4. $\beta_1' = x_1(w\beta_1)/w'$ and $\beta_2' = x_2(w\beta_2)/w'$
   where $w' = \mathit{max\_common\_prefix}(x_1(w\beta_1), x_2(w\beta_2))$
5. $r' = \{x_2(uz_2) \mid uz_2 \in r\}$

The search is conducted on-the-fly starting at the initial state. For the initial state both the actual stack and guessed stack are $\perp$ and hence the initial state is annotated by $\langle \epsilon, \epsilon \rangle$. The reachable stack language of the initial state is computed by calling procedure *Compute_Stack_Languages* (see for example [17]). Then, for each state s labeled $f\mathsf{reg}(s) = r$ and annotated by $\langle \beta_1, \beta_2 \rangle$, and for each out-going edge (s, s') labeled by $(\left[\begin{smallmatrix}\sigma_1\\ \sigma_2\end{smallmatrix}\right], z_2, x_2)$, it annotates the state s' by $\langle \beta_1', \beta_2' \rangle$, updates $f\mathsf{reg}(s')$ to contain r' and adds the annotation z_1, x_1 to the edge, where $(r, r', z_1, z_2, x_1, x_2, \sigma_1, \sigma_2, \beta_1, \beta_2, \beta_1', \beta_2')$ satisfy the conditions of a reachable transition.

In Phase 2 procedure *Project* simply projects each edge on the first coordinate. From each edge on it eliminates the second component σ_2 of the bi-letter $\left[\begin{smallmatrix}\sigma_1\\ \sigma_2\end{smallmatrix}\right]$, the stack letter z_2 and the stack command x_2.

In Phase 3, as a first step the procedure extracts from the PDA $\ddot{A}$ the finite-state automaton $\ddot{R}$. Note that $\ddot{R}$ is non-deterministic (i.e. it is an NFA): from one state s there could be two outgoing edges (s, s') and (s, s'') with the same label. This non-determinism is the result of the projection performed in the previous phase. Hence, as a second step, the algorithm applies the subset construction and obtains the DFA $\widetilde{R}$.

In the full version of the paper, we also consider the simpler case where the cascade product DPDA $\circ$ DFA is degenerate, i.e. the DFA does not look at the stack of the 1DPDA. In this case the language is given by a simple product of a DPDA and a DFA.

6 Conclusions and Future Work

Our research tried to give an answer to the verification problem of parameterized system, which cannot be answered using regular model checking. We presented three methods to handle such problems. All suggested methods have been successfully used to prove mutual exclusion of the Peterson algorithm for an arbitrary number of processes [23] and termination of a termination detection algorithm extracted from Ricart and Agrawalas' algorithm [25].

It is left to study the relation ship between these methods. Is one stronger than the other or are they incomparable, and one works on some instances while the second works on other.

In this paper we consider only safety properties. It is important to extend our methods to verify liveness properties as well.

In order to apply regular model checking or any of the methods we have developed here to a given verification problem, the verification problem must be modeled by languages and bi-languages. In several cases, in order to model the system, we used an encoding which is an abstraction of the system. It is interesting to see if one can automatically verify the correctness of such an abstraction, or even more, automatically produce a correct encoding (abstraction).

References

1. P.A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. *CAV'98*, *LNCS*, volume 1427, p. 305–318. Springer-Verlag, 1998.
2. P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. *CAV'99*, volume 1633 of *LNCS*, p. 134 Springer-Verlag, 1999.
3. K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.
4. J-P. Bodeveix and M. Filali. Experimenting acceleration methods for the validation of infinite state systems. In *DSVV'2000*, Taipei, Taiwan, April 2000.
5. B. Boigelot and P. Godefroid. Symbolic verification of coomunication protocols with infinite state spaces using QDDs. *CAV'96*, volume 1102 of *LNCS*, p. 1–12. Springer-Verlag, 1996.
6. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, LNCS Springer-Verlag, 1997.
7. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. *CONCUR'97*, *LNCS*, volume 1243, p. 135. Springer-Verlag, 1997.
8. A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with non-regular sets of configurations. In ICALP'97, volume 1256 of *LNCS* Springer-Verlag, 1997.
9. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touilli. Regular model checking. *CAV'00*, volume 1855 of *LNCS*, p. 403–418. Springer-Verlag, 2000.
10. A. Bouajjani and O. Maler. Reachability analysis of push-down automata. In *Workshop on Infinite-state Systems*, Pisa, 1996.
11. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
12. O. Burkat and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.
13. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *CONCUR'95*, p. 395–407, Philadelphia, PA, August 1995.
14. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*, LNCS, p. 268–279, 1998.
15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL'97.
16. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. *CAV'00*, volume 1855 of *LNCS*, p. 232–247. Springer-Verlag, 2000.
17. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proc. Infinity'97, Electronic Notes in Theoretical Computer Science*, 1997.
18. L. Fribourg and H. Olsen. Reachability sets of parametrized rings as regular languages. In *Proc. Infinity'97, Electronic Notes in Theoretical Computer Science*, volume 9, July 1997.
19. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.
20. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf, editor, *Proceedings of TACAS'00*, LNCS, 2000. To Appear.
21. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *CAV'97*, LNCS , p. 424–435. Springer-Verlag, 1997.
22. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL'97*, Paris, 1997.
23. G.L. Peterson. Myths about the mutual exclusion problem. *Info. Proc. Lett.*, 12(3):115, 1981.
24. A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. *CAV'00*, volume 1855 of *LNCS*, p. 328–343. Springer-Verlag, 2000.
25. G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM*, 24(1):9–17, 1981. Corr. ibid. 1981, p.581.
26. G. Senizergues. The equivalence problem for deterministic pushdown automata is decidable. volume 1256 of *LNCS*, p. 671–681, 1997.
27. E. Shahar. The tlv system and its application. Master's thesis, Weizmann Institute, 1996.
28. Colin Stirling. Decidability of dpda equivalence. *FODDACS'01*, LNCS Springer-Verlag, 2001.
29. P. Wolper and B. Boigelot. Symbolic verification with periodic sets. In D. Dill, editor, *CAV'94*, volume 818 of *LNCS*, p. 55–67. Springer-Verlag, 1994.
30. Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. *CAV'98*, *LNCS*, volume 1427, p. 88–97. Springer-Verlag, 1998.