



**HAL**  
open science

## LTL Model Checking of Self Modifying Code

Tayssir Touili, Xin Ye

► **To cite this version:**

Tayssir Touili, Xin Ye. LTL Model Checking of Self Modifying Code. Formal Methods in System Design, 2022, 60 (2). hal-03863361

**HAL Id: hal-03863361**

**<https://hal.science/hal-03863361>**

Submitted on 21 Nov 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# LTL Model Checking of Self Modifying Code

Tayssir Touili<sup>a</sup>, Xin Ye<sup>a,b</sup>

<sup>a</sup>LIPN, CNRS and University Paris 13, Villetaneuse, France

<sup>b</sup>East China Normal University, Shanghai, China

---

## Abstract

Self modifying code is code that can modify its own instructions during the execution of the program. It is extensively used by malware writers to obfuscate their malicious code. Thus, analysing self modifying code is nowadays a big challenge. In this paper, we consider the LTL model-checking problem of self modifying code. We model such programs using self-modifying pushdown systems (SM-PDS), an extension of pushdown systems that can modify its own set of transitions during execution. We reduce the LTL model-checking problem to the emptiness problem of self-modifying Büchi pushdown systems (SM-BPDS). We implemented our techniques in a tool that we successfully applied for the detection of several self-modifying malware. Our tool was also able to detect several malwares that well-known antiviruses such as BitDefender, Kinssoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec failed to detect.

*Keywords:* malware detection, model checking, automata

---

## 1. Introduction

Binary code presents several complex aspects that cannot be encountered in source code. One of these aspects is self-modifying code, i.e., code that can modify its own instructions during the execution of the program. Self-modifying code makes reverse code engineering harder. Thus, it is extensively used to protect software intellectual property. It is also heavily used by malware writers in order to make their malwares hard to analyse and detect by static analysers and anti-viruses. Thus, it is crucial to be able to analyse self-modifying code.

There are several kinds of self-modifying code. In this work, we consider self-modifying code caused by **self-modifying instructions**. These kind of instructions treat code as data. This allows them to read and write into code, leading to **self-modifying instructions**. These self-modifying instructions are usually **mov** instructions, since **mov** allows to access memory and read and write into it.

Let us consider the example shown in Figure 1. For simplicity, the addresses' length is assumed to be 1 byte. In the right box, we give, respectively, the binary code, the addresses of the different instructions, and the corresponding assembly code, obtained by translating syntactically the binary code at each address. For example, `0c` is the binary code of the jump `jmp`. Thus, `0c 02` is translated to `jmp 0x2` (jump to address `0x2`). The second line is translated to `push 0x9`, since `ff` is the binary code of the instruction `push`. The third instruction `mov 0x2 0xc` will replace the first byte at address `0x2` by `0xc`. Thus, at address `0x2`,

ff 09 is replaced by 0c 09. This means the instruction `push 0x9` is replaced by the jump instruction `jmp 0x9` (jump to address 0x9), etc. Therefore, this code is self-modifying: the `mov` instruction was able to modify the instructions of the program via its ability to read and write the memory. If we study this code without looking at the semantics of the self-modifying instructions, we will extract from it the Control Flow Graph **CFG a** that is in the left of the figure, and we will reach the conclusion that the call to the API function `CopyFileA` at address `0x9` cannot be made. However, you can see that the correct CFG is the one on the right hand side **CFG b**, where the call to the API function `CopyFileA` at address `0x9` can be reached. Thus, it is very important to be able to take into account the semantics of the self-modifying instructions in binary code.

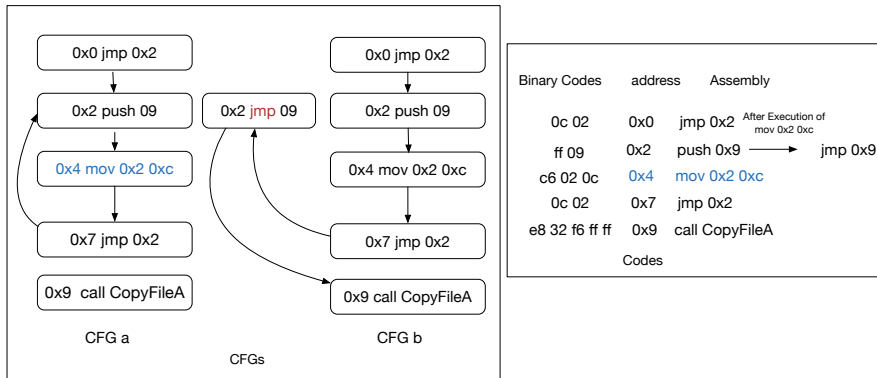


Figure 1: An Example of a Self-modifying Code

In this paper, we consider the LTL model-checking problem of self-modifying code. To this aim, we use Self-Modifying Pushdown Systems (SM-PDSs) [1] to model self-modifying code. Indeed, SM-PDSs were shown in [1] to be an adequate model for self-modifying code since they allow to mimic the program's stack while taking into account the self-modifying semantics of the transitions. This is very important for binary code analysis and malware detection, since malwares are based on calls to API functions of the operating system. Thus, antiviruses check the API calls to determine whether a program is malicious or not. Therefore, to evade from these antiviruses, malware writers try to hide the API calls they make by replacing calls by push and jump instructions. Thus, to be able to analyse such malwares, it is crucial to be able to analyse the program's stack. Hence the need to a model like pushdown systems and self-modifying pushdown systems for this purpose, since they allow to mimic the program's stack.

Intuitively, a SM-PDS is a pushdown system (PDS) with self-modifying rules, i.e., with rules that allow to modify the current set of transitions during execution. This model was introduced in [1] in order to represent self-modifying code. In [1], the authors have proposed algorithms to compute finite automata that accept the forward and backward reachability sets of SM-PDSs. In this work, we tackle the problem of LTL model-checking of SM-PDSs. Since SM-PDSs are equivalent to PDSs [1], one possible approach for LTL model checking of SM-PDS is to translate the SM-PDS to a standard PDS and then run the LTL model checking algorithm on the equivalent PDS [2, 3]. But translation

from a SM-PDS to a standard PDS is exponential. Thus, performing the LTL model checking on the equivalent PDS is not efficient.

To overcome this limitation, we propose a *direct* LTL model checking algorithm for SM-PDSs. Our algorithm is based on reducing the LTL model checking problem to the emptiness problem of Self Modifying Büchi Pushdown Systems (SM-BPDS). Intuitively, we obtain this SM-BPDS by taking the product of the SM-PDS with a Büchi automaton accepting an LTL formula  $\varphi$ . Then, we solve the emptiness problem of an SM-BPDS by computing its repeating heads. This computation is based on computing labelled *pre*\* configurations by applying a saturation procedure on labelled finite automata.

We implemented our algorithm in a tool. Our experiments show that our *direct* techniques are much more efficient than translating the SM-PDS to an equivalent PDS and then applying the standard LTL model checking for PDSs [2, 3]. Moreover, we successfully applied our tool to the analysis of 892 self-modifying malwares. Our tool was also able to detect several self-modifying malwares that well-known antiviruses like BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec were not able to detect.

**This paper is an expanded version of the conference paper [4]. Compared to [4], this journal version includes the proofs of all our results (no proof is provided in [4]).**

**Related Work.** Model checking and static analysis approaches have been widely used to analyze binary programs, for instance, in [5, 6, 7, 8, 9]. Temporal Logics were chosen to describe malicious behaviors in [10, 8, 9, 11, 12]. However, these works cannot deal with self-modifying code.

POMMADE [9, 11] is a malware detector based on LTL and CTL model-checking of PDSs. STAMAD [13, 14, 15] is a malware detector based on PDSs and machine learning. However, POMMADE and STAMAD cannot deal with self-modifying code.

Cai et al. [16] use local reasoning and separation logic to describe self-modifying code and treat program code uniformly as regular data structure. However, [16] requires programs to be manually annotated with invariants. In [17], the authors propose a formal semantics for self-modifying codes, and use that to represent self-unpacking code. This work only deals with packing and unpacking behaviours. Bonfante et al. [18] provide an operational semantics for self-modifying programs and show that they can be constructively rewritten to a non-modifying program. However, all these specifications [18, 16, 17] are too abstract to be used in practice.

In [19], the authors propose a new representation of self-modifying code named State Enhanced-Control Flow Graph (SE-CFG). SE-CFG extends standard control flow graphs with a new data structure, keeping track of the possible states programs can reach, and with edges that can be conditional on the state of the target memory location. It is not easy to analyse a binary program only using its SE-CFG, especially that this representation does not allow to take into account the stack of the program.

The authors in [20] propose abstract interpretation techniques to compute an over-approximation of the set of reachable states of a self-modifying program, where for each control point of the program, an over-approximation of the memory state at this control point is provided. Static and dynamic analysis techniques are combined to analyse self-modifying programs in [21]. Unlike our

approach, these techniques [20, 21] cannot handle the program's stack.

Unpacking binary code is also considered in [22, 23, 24, 17]. These works do not consider self-modifying **mov** instructions.

**Outline.** The rest of the paper is structured as follows: Section 2 recalls the definition of Self Modifying pushdown systems. LTL model checking and SM-BPDSs are defined in Section 3. Section 4 solves the emptiness problem of SM-BPDS. Finally, the experiments are reported in Section 5.

## 2. Self-Modifying Pushdown Systems

### 2.1. Definition

We recall in this section the definition of Self-modifying Pushdown Systems [1].

**Definition 1.** A Self-modifying Pushdown System (SM-PDS) is a tuple  $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ , where  $P$  is a finite set of control points,  $\Gamma$  is a finite set of stack symbols,  $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$  is a finite set of transition rules, and  $\Delta_c \subseteq P \times (\Delta \cup \Delta_c) \times (\Delta \cup \Delta_c) \times P$  is a finite set of modifying transition rules. If  $((p, \gamma), (p', w)) \in \Delta$ , we also write  $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$ . If  $(p, r_1, r_2, p') \in \Delta_c$ , we also write  $p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$ . A Pushdown System (PDS) is a SM-PDS where  $\Delta_c = \emptyset$ .

Intuitively, a Self-modifying Pushdown System is a Pushdown System that can dynamically modify its set of rules during the execution time: rules  $\Delta$  are standard PDS transition rules, while rules  $\Delta_c$  modify the current set of transition rules:  $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$  expresses that if the SM-PDS is in control point  $p$  and has  $\gamma$  on top of its stack, then it can move to control point  $p'$ , pop  $\gamma$  and push  $w$  onto the stack, while  $p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$  expresses that when the PDS is in control point  $p$ , then it can move to control point  $p'$ , remove the rule  $r_1$  from its current set of transition rules, and add the rule  $r_2$ .

Formally, a configuration of a SM-PDS is a tuple  $c = (\langle p, w \rangle, \theta)$  where  $p \in P$  is the control point,  $w \in \Gamma^*$  is the stack content, and  $\theta \subseteq \Delta \cup \Delta_c$  is the current set of transition rules of the SM-PDS.  $\theta$  is called the current *phase* of the SM-PDS. When the SM-PDS is a PDS, i.e., when  $\Delta_c = \emptyset$ , a configuration is a tuple  $c = (\langle p, w \rangle, \Delta)$ , since there is no changing rule, so there is only one possible phase. In this case, we can also write  $c = \langle p, w \rangle$ . Let  $\mathcal{C}$  be the set of configurations of a SM-PDS. A SM-PDS defines a transition relation  $\Rightarrow_{\mathcal{P}}$  between configurations as follows: Let  $c = (\langle p, w \rangle, \theta)$  be a configuration, and let  $r$  be a rule in  $\theta$ , then:

1. if  $r \in \Delta_c$  is of the form  $r = p \xrightarrow{(r_1, r_2)} p'$ , such that  $r_1 \in \theta$ , then  $(\langle p, w \rangle, \theta) \Rightarrow_{\mathcal{P}} (\langle p', w \rangle, \theta')$ , where  $\theta' = (\theta \setminus \{r_1\}) \cup \{r_2\}$ . In other words, the transition rule  $r$  updates the current set of transition rules  $\theta$  by removing  $r_1$  from it and adding  $r_2$  to it.
2. if  $r \in \Delta$  is of the form  $r = \langle p, \gamma \rangle \hookrightarrow \langle p', w' \rangle \in \Delta$ , then  $(\langle p, \gamma w \rangle, \theta) \Rightarrow_{\mathcal{P}} (\langle p', w' w \rangle, \theta)$ . In other words, the transition rule  $r$  moves the control point from  $p$  to  $p'$ , pops  $\gamma$  from the stack and pushes  $w'$  onto the stack. This transition keeps the current set of transition rules  $\theta$  unchanged.

Let  $\Rightarrow_{\mathcal{P}}^*$  be the transitive, reflexive closure of  $\Rightarrow_{\mathcal{P}}$  and  $\Rightarrow_{\mathcal{P}}^{\dagger}$  be its transitive closure. An execution (a run) of  $\mathcal{P}$  is a sequence of configurations  $\pi = c_0 c_1 \dots$  s.t.  $c_i \Rightarrow_{\mathcal{P}} c_{i+1}$  for every  $i \geq 0$ . Given a configuration  $c$ , the set of immediate predecessors (resp. successors) of  $c$  is  $pre_{\mathcal{P}}(c) = \{c' \in \mathcal{C} : c' \Rightarrow_{\mathcal{P}} c\}$  (resp.  $post_{\mathcal{P}}(c) = \{c' \in \mathcal{C} : c \Rightarrow_{\mathcal{P}} c'\}$ ). These notations can be generalized straightforwardly to sets of configurations. Let  $pre_{\mathcal{P}}^*$  (resp.  $post_{\mathcal{P}}^*$ ) denote the reflexive-transitive closure of  $pre_{\mathcal{P}}$  (resp.  $post_{\mathcal{P}}$ ). We remove the subscript  $\mathcal{P}$  when it is clear from the context.

We suppose w.l.o.g. that rules in  $\Delta$  are of the form  $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$  such that  $|w| \leq 2$ , and that the self-modifying rules  $r = p \xrightarrow{(r_1, r_2)} p'$  in  $\Delta_c$  are such that  $r \neq r_1$ . Note that this is not a restriction, since for a given SM-PDS, one can compute an equivalent SM-PDS that satisfies these conditions [1].

**Example 1.** Let  $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$  be a SM-PDS where  $P = \{p_1, p_2, p_3, p_4\}$ ,  $\Gamma = \{\gamma_1, \gamma_2, \gamma_3\}$ ,  $\Delta = \{r_1 : \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_1 \rangle, r_2 : \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_3, \epsilon \rangle, r_3 : \langle p_4, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_3 \rangle\}$ ,  $\Delta_c = \{r' : p_3 \xrightarrow{(r_1, r_3)} p_4\}$ . Let  $c_0 = (\langle p_1, \gamma_1 \gamma_1 \rangle, \theta_0)$  where  $\theta_0 = \{r_1, r_2, r'\}$ . Applying rule  $r_1$ , we get  $(\langle p_1, \gamma_1 \gamma_1 \rangle, \theta_0) \Rightarrow_{\mathcal{P}} (\langle p_2, \gamma_2 \gamma_1 \gamma_1 \rangle, \theta_0)$ . Then, applying rule  $r_2$ , we get  $(\langle p_2, \gamma_2 \gamma_1 \gamma_1 \rangle, \theta_0) \Rightarrow_{\mathcal{P}} (\langle p_3, \gamma_1 \gamma_1 \rangle, \theta_0)$ . Then, applying rule  $r'$ , we get  $(\langle p_3, \gamma_1 \gamma_1 \rangle, \theta_0) \Rightarrow_{\mathcal{P}} (\langle p_4, \gamma_1 \gamma_1 \rangle, \theta_1)$  where  $r'$  is self-modifying, thus, it leads the SM-PDS from phase  $\theta_0 = \{r_1, r_2, r'\}$  to phase  $\theta_1 = \theta_0 \setminus \{r_1\} \cup \{r_3\} = \{r_2, r_3, r'\}$ . Then, applying rule  $r_3$ , we get  $(\langle p_4, \gamma_1 \gamma_1 \rangle, \theta_1) \Rightarrow_{\mathcal{P}} (\langle p_2, \gamma_2 \gamma_3 \gamma_1 \rangle, \theta_1)$ . Then, applying rule  $r_2$  again, we get  $(\langle p_2, \gamma_2 \gamma_3 \gamma_1 \rangle, \theta_1) \Rightarrow_{\mathcal{P}} (\langle p_3, \gamma_3 \gamma_1 \rangle, \theta_1)$ .

## 2.2. SM-PDS vs. PDS

Let  $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$  be a SM-PDS. It was shown in [1] that:

1.  $\mathcal{P}$  can be described by an equivalent pushdown system (PDS). Indeed, since the number of phases is finite, we can encode phases in the control point of the PDS. However, this translation is not efficient since the number of control points of the equivalent PDS is  $|P| \cdot 2^{\mathcal{O}(|\Delta| + |\Delta_c|)}$ .
2.  $\mathcal{P}$  can also be described by an equivalent Symbolic pushdown system [25], where each SM-PDS rule is represented by a *single, symbolic* transition, where the different values of the phases are encoded in a symbolic way using relations between phases. This translation is not efficient neither since the size of the relations used in the symbolic transitions is  $2^{\mathcal{O}(|\Delta| + |\Delta_c|)}$ .

## 2.3. Modeling self-modifying code with SM-PDSs

### 2.3.1. Self-modifying instructions

There are different techniques to implement self-modifying code. We consider in this work code that uses self-modifying instructions. These are instructions that can access the memory locations and write onto them, thus changing the instructions that are in these memory locations. In assembly, the only instructions that can do this are the **mov** instructions. In this case, the self-modifying instructions are of the form **mov**  $l$   $v$ , where  $l$  is a location of the program that stores executable data and  $v$  is a value. This instruction replaces the value at location  $l$  (in the binary code) with the value  $v$ . This means if at location  $l$  there is a binary value  $v'$  that is involved in an assembly instruction  $i_1$ , and if by replacing  $v'$  by  $v$ , we obtain a new assembly instruction  $i_2$ , then

the instruction  $i_1$  is replaced by  $i_2$ . E.g., `ff` is the binary code of `push`, `40` is  
 195 the binary code of `inc`, `0c` is the binary code of `jmp`, `c6` is the binary code of  
`mov`, etc. Thus, if we have `mov l ff`, and if at location  $l$  there was initially the  
 value `40 01` (which corresponds to the assembly instruction `inc %edx`), then `40`  
 is replaced by `ff`, which means the instruction `inc %edx` is replaced by `push`  
`01`. If at location  $l$  there was initially the value `c6 01 02` (which corresponds to  
 200 the assembly instruction `mov edx 0x2`), then `c6` is replaced by `ff`, which means  
 the instruction `mov edx 0x2` is replaced by `push 02`.

Note that if the instructions  $i_1$  and  $i_2$  do not have the same number of  
 operands, then `mov l v` will, in addition to replacing  $i_1$  by  $i_2$ , change several  
 other instructions that follow  $i_1$ . Currently, we cannot handle this case, thus  
 205 we assume that  $i_1$  and  $i_2$  have the same number of operands.

Note also that `mov l v` is self-modifying only if  $l$  is a location of the program  
 that stores executable data, otherwise, it is not; e.g., `mov eax v` does not change  
 the instructions of the program, it just writes the value  $v$  to the register `eax`.  
 Thus, from now on, by self-modifying instruction, we mean an instruction of the  
 210 form `mov l v`, where  $l$  is a location of the program that stores executable data.  
 Moreover, to ensure that only one instruction is modified, we assume that the  
 corresponding instructions  $i_1$  and  $i_2$  have the same number of operands.

### 2.3.2. From self-modifying code to SM-PDS

We show in what follows how to build a SM-PDS from a binary program.  
 215 We suppose we are given an oracle  $\mathcal{O}$  that extracts from the binary code a  
 corresponding assembly program, together with informations about the values  
 of the registers and the memory locations at each control point of the program.  
 In our implementation, we use Jakstab [26] to get this oracle. We translate  
 the assembly program into a self-modifying pushdown system where the control  
 220 locations store the control points of the binary program and the stack mimics  
 the program's stack. The non self-modifying instructions of the program define  
 the rules  $\Delta$  of the SM-PDS (which are standard PDS rules), and can be obtained  
 following the translation of [9] that models non self-modifying instructions of  
 the program by a PDS.

As for the self-modifying instructions of the program, they define the set  
 225 of changing rules  $\Delta_c$ . As explained above, these are instructions of the form  
`mov l v`, where  $l$  is a location of the program that stores executable data. This  
 instruction replaces the value at location  $l$  (in the binary code) with the value  
 $v$ . Let  $i_1$  be the initial instruction involving the location  $l$ , and let  $i_2$  be the  
 230 new instruction involving the location  $l$ , after applying the `mov l v` instruction.  
 As mentioned previously, we assume that  $i_1$  and  $i_2$  have the same number of  
 operands (to ensure that only one instruction is modified). Let  $r_1$  (resp.  $r_2$ ) be  
 the SM-PDS rule corresponding to the instruction  $i_1$  (resp.  $i_2$ ). Suppose from  
 control point  $n$  to  $n'$ , we have this `mov l v` instruction, then we add  $n \xrightarrow{(r_1, r_2)} n'$   
 235 to  $\Delta_c$ . This is the SM-PDS rule corresponding to the instruction `mov l v` at  
 control point  $n$ .

### 3. LTL Model-Checking of SM-PDSs

#### 3.1. The linear-time temporal logic LTL

Let  $At$  be a finite set of atomic propositions. LTL formulas are defined as follows (where  $A \in At$ ):

$$\varphi := A \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2$$

Formulae are interpreted on infinite words over  $2^{At}$ . Let  $\omega = \omega^0\omega^1\dots$  be an infinite word over  $2^{At}$ . We write  $\omega_i$  for the suffix of  $\omega$  starting at  $\omega^i$ . We denote  $\omega \models \varphi$  to express that  $\omega$  satisfies a formula  $\varphi$ :

$$\begin{aligned} \omega \models A &\iff A \in \omega^0 \\ \omega \models \neg\varphi &\iff \omega \not\models \varphi \\ \omega \models \varphi_1 \vee \varphi_2 &\iff \omega \models \varphi_1 \text{ or } \omega \models \varphi_2 \\ \omega \models X\varphi &\iff \omega_1 \models \varphi \\ \omega \models \varphi_1 U \varphi_2 &\iff \exists i \geq 0, \omega_i \models \varphi_2 \text{ and } \forall 0 \leq j < i, \omega_j \models \varphi_1 \end{aligned}$$

The temporal operators G (globally) and F (eventually) are defined as follows:  $F\varphi = (A \vee \neg A)U\varphi$  and  $G\varphi = \neg F\neg\varphi$ . Let  $W(\varphi)$  be the set of infinite words that satisfy an LTL formula  $\varphi$ . It is well known that  $W(\varphi)$  can be accepted by Büchi automata:

**Definition 2.** A Büchi automaton  $\mathcal{B}$  is a quintuple  $(Q, \Gamma, \eta, q_0, F)$  where  $Q$  is a finite set of states,  $\Gamma$  is a finite input alphabet,  $\eta \subseteq (Q \times \Gamma \times Q)$  is a set of transitions,  $q_0 \in Q$  is the initial state and  $F \subseteq Q$  is the set of accepting states. A run of  $\mathcal{B}$  on a word  $\gamma_0\gamma_1\dots \in \Gamma^\omega$  is a sequence of states  $q_0q_1q_2\dots$  s.t.  $\forall i \geq 0, (q_i, \gamma_i, q_{i+1}) \in \eta$ . An infinite word  $\omega$  is accepted by  $\mathcal{B}$  if  $\mathcal{B}$  has a run on  $\omega$  that starts at  $q_0$  and visits accepting states from  $F$  infinitely often.

**Theorem.** [27] Given an LTL formula  $\varphi$ , one can effectively construct a Büchi automaton  $\mathcal{B}_\varphi$  which accepts  $W(\varphi)$ .

#### 3.2. Self Modifying Büchi Pushdown Systems

**Definition 3.** A Self Modifying Büchi Pushdown Systems (SM-BPDS) is a tuple  $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$  where  $P$  is a set of control locations,  $G \subseteq P$  is a set of accepting control locations,  $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$  is a finite set of transition rules, and  $\Delta_c \subseteq P \times 2^{\Delta \cup \Delta_c} \times 2^{\Delta \cup \Delta_c} \times P$  is a finite set of modifying transition rules of the form  $p \xrightarrow{(\sigma, \sigma')} p'$  where  $\sigma, \sigma' \subseteq \Delta \cup \Delta_c$ .

Let  $\Rightarrow_{\mathcal{BP}}$  be the transition relation between configurations as follows: Let  $\theta \subseteq \Delta \cup \Delta_c, \gamma \in \Gamma, w \in \Gamma^*$ , and  $p \in P$ , then

1. If  $r : \langle p, \gamma \rangle \hookrightarrow \langle p', w' \rangle \in \Delta$  and  $r \in \theta$ , then  $(\langle p, \gamma w \rangle, \theta) \Rightarrow_{\mathcal{BP}} (\langle p', w' w \rangle, \theta)$ .
2. If  $r : p \xrightarrow{(\sigma, \sigma')} p' \in \Delta_c$ ,  $\sigma \cap \theta \neq \emptyset$  and  $r \in \theta$ , then  $(\langle p, \gamma w \rangle, \theta) \Rightarrow_{\mathcal{BP}} (\langle p', \gamma w \rangle, \theta')$  where  $\theta' = \theta \setminus \sigma \cup \sigma'$ .

A run  $\pi$  of  $\mathcal{BP}$  is a sequence of configurations  $\pi = c_0c_1\dots$  s.t.  $c_i \Rightarrow_{\mathcal{BP}} c_{i+1}$  for every  $i \geq 0$ .  $\pi$  is accepting iff it infinitely often visits configurations having control locations in  $G$ .

Let  $c$  and  $c'$  be two configurations of the SM-BPDS  $\mathcal{BP}$ . The relation  $\Rightarrow_{\mathcal{BP}}^r$  is defined as follows:  $c \Rightarrow_{\mathcal{BP}}^r c'$  iff there exists a configuration  $(\langle g, u \rangle, \theta)$ ,  $g \in G$  s.t.  $c \Rightarrow_{\mathcal{BP}}^* (\langle g, u \rangle, \theta) \Rightarrow_{\mathcal{BP}}^+ c'$ . We remove the subscript  $\mathcal{BP}$  when it is clear



from the context. We define  $\stackrel{i}{\Rightarrow}$  as follows:  $c \stackrel{i}{\Rightarrow} c'$  iff there exists a sequence of configurations  $c_0 \Rightarrow_{\mathcal{BP}} c_1 \Rightarrow_{\mathcal{BP}} \dots \Rightarrow_{\mathcal{BP}} c_i$  s.t.  $c_0 = c$  and  $c_i = c'$ .

A head of SM-BPDS is a tuple  $(\langle p, \gamma \rangle, \theta)$  where  $p \in P$ ,  $\gamma \in \Gamma$  and  $\theta \subseteq \Delta \cup \Delta_c$ . A head  $(\langle p, \gamma \rangle, \theta)$  is repeating if there exists  $v \in \Gamma^*$  such that  $(\langle p, \gamma \rangle, \theta) \Rightarrow_{\mathcal{BP}}^r (\langle p, \gamma v \rangle, \theta)$ . The set of repeating heads of SM-BPDS is called  $Rep_{\mathcal{BP}}$ .

We assume w.l.o.g. that for every rule in  $\Delta_c$  of the form  $r : p \xrightarrow{(\sigma, \sigma')} p'$ ,  $r \notin \sigma$ . Note that this is not a restriction since a rule of the form  $r = p \xrightarrow{(\sigma, \sigma')} p'$  where  $r \in \sigma$  can be simulated by a set of rules that satisfy the above condition.

### 3.3. From LTL Model-Checking of SM-PDSs to the emptiness problem of SM-BPDSs

Let  $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$  be a self modifying pushdown system. Let  $At$  be a set of atomic propositions. Let  $\nu : P \rightarrow 2^{At}$  be a labelling function. Let  $\pi = (\langle p_0, w_0 \rangle, \theta_0)(\langle p_1, w_1 \rangle, \theta_1) \dots$  be an execution of the SM-PDS  $\mathcal{P}$ . Let  $\varphi$  be an LTL formula over the set of atomic propositions  $At$ . We say that

$$\pi \models_{\nu} \varphi \text{ iff } \nu(p_0)\nu(p_1) \dots \models \varphi$$

Let  $(\langle p, w \rangle, \theta)$  be a configuration of  $\mathcal{P}$ . We say that  $(\langle p, w \rangle, \theta) \models_{\nu} \varphi$  iff  $\mathcal{P}$  has an execution  $\pi$  starting at  $(\langle p, w \rangle, \theta)$  such that  $\pi \models_{\nu} \varphi$ .

Our goal in this paper is to perform LTL model-checking for self-modifying pushdown systems. Since SM-PDSs can be translated to standard (symbolic) pushdown systems, one way to solve this LTL model-checking problem is to compute the (symbolic) pushdown system that is equivalent to the SM-PDS (see section 2.2), and then apply the standard LTL model-checking algorithms on standard PDSs [25]. However, this approach is not efficient (as will be witnessed later in the experiments). Thus, we need a *direct* approach that performs LTL model-checking on the SM-PDS, without translating it to an equivalent PDS. Let  $\mathcal{B}_{\varphi} = (Q, 2^{At}, \eta, q_0, F)$  be a Büchi automaton that accepts  $W(\varphi)$ . We compute the SM-BPDS  $\mathcal{BP}_{\varphi} = (P \times Q, \Gamma, \Delta', \Delta'_c, G)$  by performing a kind of product between the SM-PDS  $\mathcal{P}$  and the Büchi automaton  $\mathcal{B}_{\varphi}$  as follows:

1. if  $r = \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$  and  $(q, \nu(p), q') \in \eta$ , then  $\langle \langle p, q \rangle, \gamma \rangle \hookrightarrow \langle \langle p', q' \rangle, w \rangle \in \Delta'$ . Let  $prod(r)$  be the set of rules of  $\Delta'$  obtained from the rule  $r$ , i.e., rules of  $\Delta'$  of the form  $\langle \langle p, q \rangle, \gamma \rangle \hookrightarrow \langle \langle p', q' \rangle, w \rangle$ .
2. if a rule  $r = p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$  and  $(q, \nu(p), q') \in \eta$ , then  $(p, q) \xrightarrow{(\sigma, \sigma')} (p', q') \in \Delta'_c$  where  $\sigma = prod(r_1), \sigma' = prod(r_2)$ . Let  $prod(r)$  be the set of rules of  $\Delta'$  obtained from the rule  $r$ , i.e., rules of  $\Delta'_c$  of the form  $(p, q) \xrightarrow{(\sigma, \sigma')} (p', q')$ .
3.  $G = P \times F$ .

**Remark.** Note that a rule  $r = p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$  generates rules of the form  $(p, q) \xrightarrow{(\sigma, \sigma')} (p', q') \in \Delta'_c$ , where  $\sigma = prod(r_1)$  and  $\sigma' = prod(r_2)$  are sets of rules. This is why we require that a Self Modifying Büchi Pushdown System

has modifying transition rules of the form  $p \xrightarrow{(\sigma, \sigma')} p'$  where  $\sigma, \sigma' \subseteq \Delta \cup \Delta_c$  are sets of rules.

We can show that:

**Theorem 3.1.** *Let  $(\langle p, w \rangle, \theta)$  be a configuration of the SM-PDS  $\mathcal{P}$ .  $(\langle p, w \rangle, \theta) \models_{\nu} \varphi$  iff  $\mathcal{BP}_{\varphi}$  has an accepting run from  $(\langle p, q_0 \rangle, w, \text{prod}(\theta))$  where  $\text{prod}(\theta)$  is the set of rules of  $\Delta \cup \Delta_c$  obtained from the rules of  $\theta$  as described above.*

Thus, LTL model-checking for SM-PDSs can be reduced to checking whether a SM-BPDS has an accepting run. The rest of the paper is devoted to this problem.

#### 4. The Emptiness Problem of SM-BPDSs

From now on, we fix a SM-BPDS  $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$ . Following [3], we can show that  $\mathcal{BP}$  has an accepting run starting from a configuration  $c$  if and only if from  $c$ , it can reach a configuration with a repeating head:

**Proposition 1.** *A SM-BPDS  $\mathcal{BP}$  has an accepting run starting from a configuration  $c$  if and only if there exists a repeating head  $(\langle p, \gamma \rangle, \theta)$  such that  $c \Rightarrow_{\mathcal{BP}}^* (\langle p, \gamma w \rangle, \theta)$  for some  $w \in \Gamma^*$ .*

**Proof:** “ $\Rightarrow$ ”: Let  $\sigma = c_0 c_1 \dots$  be an accepting run starting at configuration  $c$  where  $c_0 = c$  and  $c_i = (\langle p_i, w_i \rangle, \theta_i)$ . We construct an increasing sequence of indices  $i_0, i_1 \dots$  with a property that once any of the configurations  $c_{i_k}$  is reached, the rest of the run never changes the bottom  $|w_{i_k}| - 1$  elements of the stack anymore. This property can be written as follows:

$$|w_{i_0}| = \min\{|w_j| \mid j \geq 0\}$$

$$|w_{i_k}| = \min\{|w_j| \mid j > i_{k-1}\}, k \geq 1$$

Because  $\mathcal{BP}$  has only finitely many different heads, there must be a head  $(\langle p, \gamma \rangle, \theta)$  which occurs infinitely often as a head in the sequence  $c_{i_0} c_{i_1} \dots$ . Moreover, as some  $g \in G$  becomes a control location infinitely often, we can find a subsequence of indices  $i_{j_0}, i_{j_1}, \dots$  with the following property: for every  $k \geq 1$ , there exist  $v, w \in \Gamma^*$

$$c_{i_{j_k}} = (\langle p, \gamma w \rangle, \theta) \Rightarrow^r (\langle p, \gamma v w \rangle, \theta) = c_{i_{j_{k+1}}}$$

Because  $w$  is never looked at or changed in this path, we can have  $(\langle p, \gamma \rangle, \theta) \Rightarrow^r (\langle p, \gamma v \rangle, \theta)$ . This proves this direction of the proposition.

“ $\Leftarrow$ ”: Because  $(\langle p, \gamma \rangle, \theta)$  is a repeating head, we can construct the following run for some  $u, v, w \in \Gamma^*$ ,  $\theta' \subseteq (\Delta \cup \Delta_c)$  and  $g \in G$ :

$$c \Rightarrow^* (\langle p, \gamma w \rangle, \theta) \Rightarrow^* (\langle g, u w \rangle, \theta') \Rightarrow^+ (\langle p, \gamma v w \rangle, \theta) \Rightarrow^* (\langle g, u v w \rangle, \theta') \Rightarrow^+ (\langle p, \gamma v v w \rangle, \theta) \Rightarrow^* \dots$$

Since  $g$  occurs infinitely often, the run is accepting.  $\square$

330

Thus, since there exists an efficient algorithm to compute the  $\text{pre}^*$  of SM-PDSs [1], the emptiness problem of a SM-BPDS can be reduced to computing its repeating heads.

#### 4.1. The Head Reachability Graph $\mathcal{G}$

Our goal is to compute the set of repeating heads  $Rep_{\mathcal{BP}}$ , i.e., the set of heads  $(\langle p, \gamma \rangle, \theta)$  such that there exists  $v \in \Gamma^*$ ,  $(\langle p, \gamma \rangle, \theta) \Rightarrow^r (\langle p, \gamma v \rangle, \theta)$ . I.e.,  $(\langle p, \gamma \rangle, \theta) \Rightarrow^* (\langle p, \gamma v \rangle, \theta)$  s.t. this path goes through an accepting location in  $G$ . To this aim, we will compute a finite graph  $\mathcal{G}$  whose nodes are the heads of  $\mathcal{BP}$  of the form  $(\langle p, \gamma \rangle, \theta)$ , where  $p \in P$ ,  $\gamma \in \Gamma$  and  $\theta \subseteq \Delta \cup \Delta_c$ ; and whose edges encode the reachability relation between these heads. More precisely, given two heads  $(\langle p, \gamma \rangle, \theta)$  and  $(\langle p', \gamma' \rangle, \theta')$ ,  $(\langle p, \gamma \rangle, \theta) \xrightarrow{b} (\langle p', \gamma' \rangle, \theta')$  is an edge of the graph  $\mathcal{G}$  means that the configuration  $(\langle p, \gamma \rangle, \theta)$  can reach a configuration having  $(\langle p', \gamma' \rangle, \theta')$  as head, i.e., it means that there exists  $v \in \Gamma^*$  s.t.  $(\langle p, \gamma \rangle, \theta) \Rightarrow^* (\langle p', \gamma' v \rangle, \theta')$ . Moreover, we need to keep the information whether this path visits an accepting location in  $G$  or not. This information is recorded in the label of the edge  $b$ :  $b = 1$  means that the path visits an accepting location in  $G$ , i.e. that  $(\langle p, \gamma \rangle, \theta) \Rightarrow^r (\langle p', \gamma' v \rangle, \theta')$ . Otherwise,  $b = 0$ . Therefore, if the graph  $\mathcal{G}$  contains a loop from a head  $(\langle p, \gamma \rangle, \theta)$  to itself such that this loop goes through an edge labelled by 1, then  $(\langle p, \gamma \rangle, \theta)$  is a repeating head. Thus, computing  $Rep_{\mathcal{BP}}$  can be reduced to computing the graph  $\mathcal{G}$  and finding 1-labelled loops in this graph.

More precisely, we define the head reachability graph  $\mathcal{G}$  as follows:

**Definition 4.** *The head reachability graph  $\mathcal{G}$  is a tuple  $(P \times \Gamma \times 2^{\Delta \cup \Delta_c}, \{0, 1\}, \delta)$  such that  $(\langle p, \gamma \rangle, \theta) \xrightarrow{b} (\langle p', \gamma' \rangle, \theta')$  is an edge of  $\delta$  iff:*

1. *there exists a transition  $r_c : p \xrightarrow{(\sigma, \sigma')} p' \in \theta \cap \Delta_c$ ,  $\gamma = \gamma'$ ,  $\theta' = \theta \setminus \sigma \cup \sigma'$ , and  $b = 1$  iff  $p \in G$ ;*
2. *there exists a transition  $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \theta \cap \Delta$ ,  $\theta = \theta'$  and  $b = 1$  iff  $p \in G$ ;*
3. *there exists a transition  $\langle p, \gamma \rangle \hookrightarrow \langle p'', \gamma_1 \gamma' \rangle \in \theta \cap \Delta$ , for  $\gamma_1 \in \Gamma$ ,  $p'' \in P$ , s.t.  $(\langle p'', \gamma_1 \rangle, \theta) \Rightarrow_{\mathcal{BP}}^* (\langle p', \epsilon \rangle, \theta')$ , and  $b = 1$  iff  $p \in G$  or  $(\langle p'', \gamma_1 \rangle, \theta) \Rightarrow_{\mathcal{BP}}^r (\langle p', \epsilon \rangle, \theta')$*

Let  $\mathcal{G}$  be the head reachability graph. We define  $\xrightarrow{i}$  as follows: let  $(\langle p, \gamma \rangle, \theta)$  and  $(\langle p', \gamma' \rangle, \theta')$  be two heads of  $\mathcal{BP}$ . We write  $(\langle p, \gamma \rangle, \theta) \xrightarrow{i} (\langle p', \gamma' \rangle, \theta')$  iff  $\exists$  booleans  $b_1, b_2 \dots b_i \in \{0, 1\}$ ,  $\exists$  heads  $(\langle p_j, \gamma_j \rangle, \theta_j)$ ,  $0 \leq j \leq i$  s.t.  $\mathcal{G}$  contains the path  $(\langle p_0, \gamma_0 \rangle, \theta_0) \xrightarrow{b_1} (\langle p_1, \gamma_1 \rangle, \theta_1) \xrightarrow{b_2} \dots \xrightarrow{b_i} (\langle p_i, \gamma_i \rangle, \theta_i)$  where  $(\langle p_0, \gamma_0 \rangle, \theta_0) = (\langle p, \gamma \rangle, \theta)$  and  $(\langle p_i, \gamma_i \rangle, \theta_i) = (\langle p', \gamma' \rangle, \theta')$ .

Let  $\rightarrow^*$  be the reflexive transitive closure of the graph relation  $\xrightarrow{b}$ , and let  $\rightarrow^r$  be defined as follows: Given two heads  $(\langle p, \gamma \rangle, \theta)$  and  $(\langle p', \gamma' \rangle, \theta')$ ,  $(\langle p, \gamma \rangle, \theta) \rightarrow^r (\langle p', \gamma' \rangle, \theta')$  iff there is in  $\mathcal{G}$  a path between  $(\langle p, \gamma \rangle, \theta)$  and  $(\langle p', \gamma' \rangle, \theta')$  that goes through a 1-labelled edge, i.e., iff there exist heads  $(\langle p_1, \gamma_1 \rangle, \theta_1)$  and  $(\langle p_2, \gamma_2 \rangle, \theta_2)$  s.t.  $(\langle p, \gamma \rangle, \theta) \rightarrow^* (\langle p_1, \gamma_1 \rangle, \theta_1) \xrightarrow{1} (\langle p_2, \gamma_2 \rangle, \theta_2) \rightarrow^* (\langle p', \gamma' \rangle, \theta')$ .

We can show that:

**Theorem 4.1.** *Let  $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$  be a self-modifying Büchi pushdown system, and let  $\mathcal{G}$  be its corresponding head reachability graph. A head  $(\langle p, \gamma \rangle, \theta)$  of  $\mathcal{BP}$  is repeating iff  $\mathcal{G}$  has a loop on the node  $(\langle p, \gamma \rangle, \theta)$  that goes through a 1-labeled edge.*

To prove this theorem, we first need to prove the following lemma:

**Lemma 1.** *The relations  $\rightarrow^*$  and  $\rightarrow^r$  have the following properties: For any heads  $((p, \gamma), \theta_1)$  and  $((p', \gamma'), \theta_2)$ :*

- 380 (a)  $((p, \gamma), \theta_1) \rightarrow^* ((p', \gamma'), \theta_2)$  iff  $((p, \gamma), \theta_1) \Rightarrow^* ((p', \gamma'v), \theta_2)$  for some  $v \in \Gamma^*$ .
- (b)  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$  iff  $((p, \gamma), \theta_1) \Rightarrow^r ((p', \gamma'v), \theta_2)$  for some  $v \in \Gamma^*$ .

**Proof:** “ $\Rightarrow$ ”: Assume  $((p, \gamma), \theta_1) \xrightarrow{i} ((p', \gamma'), \theta_2)$ . We proceed by induction on  
385  $i$ .

- (a) **Basis.**  $i = 0$ . In this case,  $((p, \gamma), \theta_1) = ((p', \gamma'), \theta_2)$ , then we can get  $((p, \gamma), \theta_1) \Rightarrow^* ((p', \gamma'), \theta_2)$

**Step.**  $i > 0$ . Then there exist  $p_1 \in P, \gamma'' \in \Gamma^*$  and  $\theta' \subseteq \Delta \cup \Delta_c$  such that  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma''), \theta') \xrightarrow{i-1} ((p', \gamma'), \theta_2)$ . From the induction  
390 hypothesis, there exists  $u \in \Gamma^*$  such that  $((p_1, \gamma''), \theta') \Rightarrow^* ((p', \gamma'u), \theta_2)$ . Since  $((p, \gamma), \theta_1) \rightarrow ((p_1, \gamma''), \theta')$ , we have  $((p, \gamma), \theta_1) \Rightarrow^* ((p_1, \gamma''w), \theta')$  for  $w \in \Gamma^*$ , hence  $((p, \gamma), \theta_1) \Rightarrow^* ((p', \gamma'uw), \theta_2)$ . The property holds.

- (b)  $((p, \gamma), \theta_1) \rightarrow^r ((p, \gamma), \theta_1)$  cannot hold for the case  $i = 0$ .

395 **Basis.**  $i = 1$ . In this case,  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ , then we can get  $p \in G$  and  $((p, \gamma), \theta_1) \Rightarrow^r ((p', \gamma'), \theta_2)$ . The property holds.

**Step.**  $i > 0$ . As done in the proof of part (a) of this lemma, there exists  $p_1, \gamma'' \in \Gamma, \theta'' \subseteq \Delta \cup \Delta_c$  s.t.  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma''), \theta'') \xrightarrow{i-1} ((p', \gamma'), \theta_2)$ . Then if  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ , either  $((p_1, \gamma''), \theta'') \rightarrow^r ((p', \gamma'), \theta_2)$  or  
400  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma''), \theta'')$  holds. In the first case i.e.  $((p_1, \gamma''), \theta'') \rightarrow^r ((p', \gamma'), \theta_2)$ , by the induction hypothesis, we can have  $((p_1, \gamma''), \theta'') \Rightarrow^r ((p', \gamma'u), \theta_2)$ , hence,  $((p, \gamma), \theta_1) \Rightarrow^r ((p', \gamma'u), \theta_2)$  holds

The second case depends on the rule applied to get  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma''), \theta'')$  according to Definition 4.

- 405 - If this edge corresponds to a transition  $r_c : p \xrightarrow{(\sigma, \sigma')} p_1 \in \theta_1$ , then  $\gamma = \gamma'', \theta' = \theta_1 \setminus \sigma \cup \sigma'$  and  $p \in G$ . Since we can obtain  $((p, \gamma), \theta_1) \Rightarrow_{\mathcal{BP}} ((p_1, \gamma), \theta') \Rightarrow^* ((p', \gamma'uw), \theta_2)$  from part (a) and  $p \in G$ , then  $((p, \gamma), \theta_1) \Rightarrow^r ((p_1, \gamma), \theta') \Rightarrow^* ((p', \gamma'uw), \theta_2)$ . This implies that  $((p, \gamma), \theta_1) \Rightarrow^r ((p', \gamma'v), \theta_2)$  for some  $v \in \Gamma^*$ .
- 410 - If this edge corresponds to a transition  $r : \langle p, \gamma \rangle \hookrightarrow \langle p_1, \gamma'' \rangle \in \theta_1 \cap \Delta$ , then  $\theta' = \theta_1$  and  $p \in G$ . Since we can obtain  $((p, \gamma), \theta_1) \Rightarrow_{\mathcal{BP}} ((p_1, \gamma''), \theta_1) \Rightarrow^* ((p', \gamma'uw), \theta_2)$  from part (a) and  $p \in G$ , then  $((p, \gamma), \theta_1) \Rightarrow^r ((p_1, \gamma''), \theta_1) \Rightarrow^* ((p', \gamma'uw), \theta_2)$ . This implies that  $((p, \gamma), \theta_1) \Rightarrow^r ((p', \gamma'v), \theta_2)$  for some  $v \in \Gamma^*$ .

415 - If this edge corresponds to a transition  $r : \langle p, \gamma \rangle \hookrightarrow \langle p'', \gamma_1 \gamma'' \rangle \in \theta_1$ , then either  $p \in G$  or  $(\langle p'', \gamma_1 \rangle, \theta_1) \Rightarrow^r (\langle p_1, \epsilon \rangle, \theta')$  holds. If  $p \in G$ , then we have  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p'', \gamma_1 \gamma'' \rangle, \theta_1)$ . Otherwise,  $(\langle p'', v_1 \gamma'' w \rangle, \theta_1) \Rightarrow^r (\langle p_1, \gamma'' w \rangle, \theta')$ . Since we can obtain  $(\langle p_1, \gamma'' \rangle, \theta') \Rightarrow^*$   
420  $(\langle p', \gamma' u \rangle, \theta_2)$  from part (a). Therefore,  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p_1, \gamma'' \rangle, \theta') \Rightarrow^*$   
 $(\langle p', \gamma' u \rangle, \theta_2)$ . This implies that  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p', \gamma' v \rangle, \theta_2)$  for some  $v \in \Gamma^*$ .

‘ $\Leftarrow$ ’: Assume  $(\langle p, \gamma \rangle, \theta_1) \stackrel{i}{\Rightarrow} (\langle p', \gamma' v \rangle, \theta_2)$ . We proceed by induction on  $i$ .

(a) **Basis.**  $i = 0$ . In this case,  $v = \epsilon$  and  $(\langle p, \gamma \rangle, \theta_1) = (\langle p', \gamma' \rangle, \theta_2)$ , then  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$  holds.

425 **Step.**  $i > 0$ . Then there exist  $p_1 \in P, u \in \Gamma^*$  and  $\theta' \subseteq \Delta \cup \Delta_c$  such that  $(\langle p, \gamma \rangle, \theta_1) \stackrel{1}{\Rightarrow} (\langle p_1, u \rangle, \theta') \stackrel{i-1}{\Rightarrow} (\langle p', \gamma' v \rangle, \theta_2)$ . There are 2 cases:

1. Case  $\theta' = \theta_1$  : There must exist a rule  $r : \langle p, \gamma \rangle \hookrightarrow \langle p_1, u \rangle \in \Delta$  such that  $r \in \theta'$  and  $|u| \geq 1$ . Let  $l$  denote the minimal length of the stack on the path from  $(\langle p_1, u \rangle, \theta_1)$  to  $(\langle p', \gamma' v \rangle, \theta_2)$ . Then  $u$  can be written as  $u'' \gamma_1 u'$  where  $|u'| = l - 1$  (that means  $u'$  will remain on the stack for the path). Furthermore, there exists  $p'''$  such that  $(\langle p_1, u'' \rangle, \theta_1) \Rightarrow^* (\langle p''', \epsilon \rangle, \theta'')$  for some  $\theta'' \subseteq (\Delta_c \cup \Delta)$ . We have  $(\langle p, \gamma \rangle, \theta_1) \stackrel{k}{\Rightarrow} (\langle p''', \gamma_1 u' \rangle, \theta'')$  for  $k < i$ . By the induction on  $i$ , we have  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^* (\langle p''', \gamma_1 \rangle, \theta'')$ . Because  $u'$  has to remain on the stack for the rest of the path,  $v$  is of the form  $v' u'$  for some  $v' \in \Gamma^*$ .  
435 That means  $(\langle p''', \gamma_1 \rangle, \theta'') \stackrel{j}{\Rightarrow} (\langle p', \gamma' v' \rangle, \theta_2)$  for  $j < i$ . By the induction hypothesis,  $(\langle p''', \gamma_1 \rangle, \theta'') \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$  holds. Moreover, we have  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^* (\langle p''', \gamma_1 \rangle, \theta'')$ , hence  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$ .

440 2. Case  $\theta' \neq \theta_1$  : There must be a rule  $r_c : p \xrightarrow{(\sigma, \sigma')} p_1 \in \Delta_c$  such that  $r_c \in \theta_1$  and  $\sigma \cap \theta_1 \neq \emptyset$ , then  $\theta' = \theta_1 \setminus \sigma \cup \sigma'$ . After the execution of  $r_c$ , the content of the stack will remain the same, thus,  $u = \gamma$ . Then  $(\langle p, \gamma \rangle, \theta_1) \stackrel{1}{\Rightarrow} (\langle p_1, \gamma \rangle, \theta') \stackrel{i-1}{\Rightarrow} (\langle p', \gamma' v \rangle, \theta_2)$ . By the induction hypothesis to  $(\langle p_1, \gamma \rangle, \theta') \stackrel{i-1}{\Rightarrow} (\langle p', \gamma' v \rangle, \theta_2)$ , we can obtain  
445 that  $(\langle p_1, \gamma \rangle, \theta') \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$ . Since  $(\langle p, \gamma \rangle, \theta_1) \stackrel{1}{\Rightarrow} (\langle p_1, \gamma \rangle, \theta')$ , then we can have a path  $(\langle p, \gamma \rangle, \theta_1) \rightarrow (\langle p_1, \gamma \rangle, \theta') \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$  that implies  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^* (\langle p', \gamma' \rangle, \theta_2)$ . The property holds.

(b)  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p, \gamma' v \rangle, \theta_1)$  is impossible in 0 steps.

450 **Basis.**  $i = 1$ .  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p, \gamma \rangle, \theta_1)$ , then  $p \in G$ . Thus,  $(\langle p, \gamma \rangle, \theta_1) \rightarrow^r (\langle p, \gamma \rangle, \theta_1)$  holds.

**Step.**  $i > 1$ .  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p', \gamma' v \rangle, \theta_2)$  holds, then there exist  $p_1 \in P, u \in \Gamma^*$  and  $\theta' \subseteq \Delta \cup \Delta_c$  such that  $(\langle p, \gamma \rangle, \theta_1) \stackrel{1}{\Rightarrow} (\langle p_1, u \rangle, \theta') \stackrel{i-1}{\Rightarrow} (\langle p', \gamma' v \rangle, \theta_2)$ . Thus, either  $(\langle p, \gamma \rangle, \theta_1) \Rightarrow^r (\langle p_1, u \rangle, \theta')$  or  $(\langle p_1, u \rangle, \theta') \Rightarrow^r (\langle p', \gamma' v \rangle, \theta_2)$  holds.

455 The first case implies  $p \in G$ . There are 2 cases:

1. Case  $\theta' = \theta_1$  : then as in the previous proof of part (a), we can have a path  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'') \rightarrow^* ((p', \gamma'), \theta_2)$ . Since  $p \in G$ , we get by Definition 4  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'') \rightarrow^* ((p', \gamma'), \theta_2)$ . Thus, we have that  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ . The property holds.
2. Case  $\theta' \neq \theta_1$ : then as in the previous proof of part (a), we can have a path  $((p, \gamma), \theta_1) \rightarrow ((p_1, \gamma), \theta') \rightarrow^* ((p', \gamma'), \theta_2)$ . Since  $p \in G$ , we get  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma), \theta') \rightarrow^* ((p', \gamma'), \theta_2)$ . Thus, we have that  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ . The property holds.

In the second case,  $((p_1, u), \theta') \Rightarrow^r ((p', \gamma'v), \theta_2)$  holds. As previously, there are 2 cases:

1. Case  $\theta' = \theta_1$  : then as in case (a) we have  $((p_1, u), \theta_1) \Rightarrow^* ((p''', \gamma_1 u'), \theta'')$  and  $((p''', \gamma_1), \theta'') \Rightarrow^* ((p', \gamma'v'), \theta_2)$ . If  $((p_1, u), \theta_1) \Rightarrow^r ((p', \gamma'v'), \theta_2)$ , then either  $((p_1, u), \theta_1) \Rightarrow^r ((p''', \gamma_1 u'), \theta'')$  or  $((p''', \gamma_1), \theta'') \Rightarrow^r ((p', \gamma'v'), \theta_2)$ .
  - If  $((p_1, u), \theta_1) \Rightarrow^r ((p''', \gamma_1 u'), \theta'')$ , let  $u'' \in \Gamma^*$  s.t.  $u = u'' \gamma_1 u'$  and  $((p_1, u''), \theta_1) \Rightarrow^r ((p''', \epsilon), \theta'')$ , then, we have  $((p, \gamma), \theta_1) \rightarrow^r ((p''', \gamma_1), \theta'')$ . We have  $((p, \gamma), \theta_1) \xrightarrow{k} ((p''', \gamma_1 u'), \theta'')$  for  $k < i$ . By the induction on  $i$ , we have  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'')$ . Because  $u'$  has to remain on the stack for the rest of the path,  $v$  is of the form  $v'u'$  for some  $v' \in \Gamma^*$ . That means  $((p''', \gamma_1), \theta'') \xrightarrow{j} ((p', \gamma'v'), \theta_2)$  for  $j < i$ . By the induction hypothesis,  $((p''', \gamma_1), \theta'') \rightarrow^* ((p', \gamma'), \theta_2)$  holds. Moreover, we have  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'')$ , hence  $((p, \gamma), \theta_1) \rightarrow^* ((p', \gamma'), \theta_2)$ . So we can have a path  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'') \rightarrow^* ((p', \gamma'), \theta_2)$ , thus we have that  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ ;
  - If  $((p''', \gamma_1), \theta'') \Rightarrow^r ((p', \gamma'v'), \theta_2)$ , then by the induction hypothesis we have  $((p''', \gamma_1), \theta'') \rightarrow^r ((p', \gamma'), \theta_2)$ . Thus, we can have a path  $((p, \gamma), \theta_1) \rightarrow^* ((p''', \gamma_1), \theta'') \rightarrow^* ((p', \gamma'), \theta_2)$ , then we have that  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ ;
2. Case  $\theta' \neq \theta_1$  : then  $((p_1, \gamma), \theta') \Rightarrow^r ((p', \gamma'v), \theta_2)$ . By the induction hypothesis we have  $((p_1, \gamma), \theta') \rightarrow^r ((p', \gamma'), \theta_2)$ . Since  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma), \theta') \xrightarrow{i-1} ((p', \gamma'v), \theta_2)$ . By the induction hypothesis to  $((p_1, \gamma), \theta') \xrightarrow{i-1} ((p', \gamma'v), \theta_2)$ , we can obtain that  $((p_1, \gamma), \theta') \rightarrow^* ((p', \gamma'), \theta_2)$ . Since  $((p, \gamma), \theta_1) \xrightarrow{1} ((p_1, \gamma), \theta')$ , then we can have a path  $((p, \gamma), \theta_1) \rightarrow ((p_1, \gamma), \theta') \rightarrow^* ((p', \gamma'), \theta_2)$ . Thus, we have that  $((p, \gamma), \theta_1) \rightarrow^r ((p', \gamma'), \theta_2)$ ;

Thus, the property holds. □

### Proof of Theorem 4.1

We can now prove Theorem 4.1.

**Proof:** Let  $((p, \gamma), \theta)$  be a repeating head, then there exists some  $v \in \Gamma^*$ ,  $\theta \subseteq \Delta_c \cup \Delta$  such that  $((p, \gamma), \theta) \Rightarrow^r ((p, \gamma v), \theta)$ . By Lemma 1, this is the case if and only if  $((p, \gamma), \theta) \rightarrow^r ((p, \gamma), \theta)$ . From the definition of  $\rightarrow^r$ , that means that there exist heads  $((p_1, \gamma_1), \theta')$  and  $((p_2, \gamma_2), \theta'')$  such that  $((p, \gamma), \theta) \rightarrow^* ((p_1, \gamma_1), \theta') \xrightarrow{1} ((p_2, \gamma_2), \theta'') \rightarrow^* ((p, \gamma), \theta)$ . Then  $((p, \gamma), \theta)$ ,  $((p_1, \gamma_1), \theta')$  and  $((p_2, \gamma_2), \theta'')$  are all in the same loop with a 1-labelled edge. Conversely, whenever  $((p, \gamma), \theta)$  is in a component with such an edge,  $((p, \gamma), \theta) \rightarrow^r ((p, \gamma), \theta)$  holds, then Lemma 1

implies that  $(\langle p, \gamma \rangle, \theta) \Rightarrow^r (\langle p, \gamma v \rangle, \theta)$  which means that  $(\langle p, \gamma \rangle, \theta)$  is a repeating head. □

505

#### 4.2. Labelled configurations and labelled $\mathcal{BP}$ -automata

To compute  $\mathcal{G}$ , we need to be able to compute predecessors of configurations of the form  $(\langle p', \epsilon \rangle, \theta')$ , and to determine whether these predecessors were backward-reachable using some control points in  $G$  (item 3 in Definition 4). To solve this question, we will label configurations  $(\langle p'', w \rangle, \theta)$  s.t.  $(\langle p'', w \rangle, \theta) \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$  by 1 if this path went through an accepting location in  $G$ , i.e., if  $(\langle p'', w \rangle, \theta) \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ , and by 0 if not. To this aim, we define a labelled configuration as a tuple  $[(\langle p, w \rangle, \theta), b]$ , s.t.  $(\langle p, w \rangle, \theta)$  is a configuration and  $b \in \{0, 1\}$ .

Multi-automata were introduced in [2, 3] to finitely represent regular infinite sets of configurations of a PDS. Since a labelled configuration  $c = [(\langle p, w \rangle, \theta), b]$  of a SM-PDS involves a PDS configuration  $\langle p, w \rangle$ , together with the current set of transition rules (phase)  $\theta$ , and a boolean  $b$ , in order to take into account the phases  $\theta$ , and these new 0/1-labels in configurations, we extend multi-automata to labelled  $\mathcal{BP}$ -automata as follows:

**Definition 5.** Let  $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$  be a SM-BPDS. A labelled  $\mathcal{BP}$ -automaton is a tuple  $\mathcal{A} = (Q, \Gamma, T, I, F)$  where  $\Gamma$  is the automaton alphabet,  $Q$  is a finite set of states,  $I \subseteq P \times 2^{\Delta \cup \Delta_c} \subseteq Q$  is the set of initial states,  $T \subseteq Q \times ((\Gamma \cup \{\epsilon\}) \times \{0, 1\}) \times Q$  is the set of transitions,  $F \subseteq Q$  is the set of final states.

If  $(q, [\gamma, b], q') \in T$ , we write  $q \xrightarrow{[\gamma, b]}_T q'$ . We extend this notation in the obvious way to sequences of symbols: (1)  $\forall q \in Q, q \xrightarrow{[\epsilon, 0]}_T q$ , and (2)  $\forall q, q' \in Q, \forall b \in \{0, 1\}, \forall w \in \Gamma^*$  for  $w = \gamma_0 \dots \gamma_{n+1}$ ,  $q \xrightarrow{[w, b]}_T q'$  iff  $\exists q_0, \dots, q_n \in Q, b_0, \dots, b_{n+1} \in \{0, 1\}, b = b_0 \vee b_1 \vee \dots \vee b_{n+1}$  and  $q \xrightarrow{[\gamma_0, b_0]}_T q_0 \xrightarrow{[\gamma_1, b_1]}_T q_1 \dots q_n \xrightarrow{[\gamma_{n+1}, b_{n+1}]}_T q'$ . If  $q \xrightarrow{[w, b]}_T q'$  holds, we say that  $q \xrightarrow{[w, b]}_T q'$  and  $q \xrightarrow{[\gamma_0, b_0]}_T q_0 \xrightarrow{[\gamma_1, b_1]}_T q_1 \dots q_n \xrightarrow{[\gamma_{n+1}, b_{n+1}]}_T q'$  is a path of  $\mathcal{A}$ .

A labelled configuration  $[(\langle p, w \rangle, \theta), b]$  is accepted by the automaton  $\mathcal{A}$  iff there exists a path  $(p, \theta) \xrightarrow{[\gamma_0, b_0]}_T q_1 \xrightarrow{[\gamma_1, b_1]}_T q_2 \dots q_n \xrightarrow{[\gamma_n, b_n]}_T q_{n+1}$  in  $\mathcal{A}$  such that  $w = \gamma_0 \gamma_1 \dots \gamma_n$ ,  $b = b_0 \vee b_1 \vee \dots \vee b_n$ ,  $(p, \theta) \in I$ , and  $q_{n+1} \in F$ . Let  $L(\mathcal{A})$  be the set of labelled configurations accepted by  $\mathcal{A}$ .

#### 4.3. Computing $pre^*((\langle p', \epsilon \rangle, \theta'))$

Given a configuration of the form  $(\langle p', \epsilon \rangle, \theta')$ , our goal is to compute a labelled  $\mathcal{BP}$ -automaton  $\mathcal{A}_{pre^*((\langle p', \epsilon \rangle, \theta'))}$  that accepts labelled configurations of the form  $[c, b]$  where  $c$  is a configuration and  $b \in \{0, 1\}$  such that  $c \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$  (i.e.,  $c \in pre^*((\langle p', \epsilon \rangle, \theta'))$ ) and  $b = 1$  iff this path went through final control points, i.e.,  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ . Otherwise,  $b = 0$ .

Let  $p \in P$ , we define  $B(p) = 1$  if  $p \in G$  and  $B(p) = 0$  otherwise.  $\mathcal{A}_{pre^*((\langle p', \epsilon \rangle, \theta'))} = (Q, \Gamma, T, I, F)$  is computed as follows: Initially,  $Q = I = F = \{(p', \theta')\}$  and  $T = \emptyset$ . We add to  $T$  transitions as follows:

545  $\alpha_1$ : If  $r = \langle p, \gamma \rangle \hookrightarrow \langle p_1, w \rangle \in \Delta$ . If there exists in  $T$  a path  $(p_1, \theta) \xrightarrow{[w, b]}_T q$   
(in case  $|w| = 0$ , we have  $w = \epsilon$ ) with  $r \in \theta$ . Then, add  $(p, \theta)$  to  $I$ , and  
 $((p, \theta), [\gamma, B(p) \vee b], q)$  to  $T$ .

$\alpha_2$ : if  $r = p \xrightarrow{(\sigma, \sigma')} p_1 \in \Delta_c$  and there exists in  $T$  a transition  $(p_1, \theta) \xrightarrow{[\gamma, b]}_T q$   
with  $r \in \theta$ , where  $\gamma \in \Gamma$ . Then add  $(p, \theta')$  to  $I$ , and  $((p, \theta'), [\gamma, B(p) \vee b], q)$   
550 to  $T$ , for  $\theta'$  such that  $\theta = \theta' \setminus \sigma \cup \sigma'$ .

The procedure above terminates since there is a finite number of states and phases. Note that by construction,  $F = \{(p', \theta')\}$ , and, since initially  $Q = \{(p', \theta')\}$ , states of  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$  are all of the form  $(p, \theta)$  for  $p \in P$  and  $\theta \subseteq \Delta \cup \Delta_c$ .

555 Let us explain the intuition behind rule  $(\alpha_1)$ . Let  $r = \langle p, \gamma \rangle \hookrightarrow \langle p_1, w \rangle \in \Delta$ . Let  $c = (\langle p_1, ww' \rangle, \theta)$  and  $c' = (\langle p, \gamma w' \rangle, \theta)$ . Then, if  $c \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$ , then necessarily,  $c' \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$ . Moreover,  $c' \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$  iff either  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$  or  $p \in G$  (i.e.  $B(p) = 1$ ). Thus, we would like that if the automaton  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$  accepts the labelled configuration  $[c, b]$  (where  $b = 1$  means  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ ), then it should also accept the labelled configuration  
560  $[c', b \vee B(p)]$  ( $b \vee B(p) = 1$  means  $c' \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ ). Thus, if the automaton  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$  contains a path of the form  $\pi = (p_1, \theta) \xrightarrow{[w, b_1]}_T q \xrightarrow{[w', b_2]}_T q_f$  where  $q_f \in F$  that accepts the labelled configuration  $[c, b]$ , then the automaton should also accept the labelled configuration  $[c', b \vee B(p)]$ . This configuration is  
565 accepted by the run  $(p, \theta) \xrightarrow{[\gamma, B(p) \vee b_1]}_T q \xrightarrow{[w', b_2]}_T q_f$  added by rule  $(\alpha_1)$ .

Rule  $(\alpha_2)$  deals with modifying rules: Let  $r = p \xrightarrow{(r_1, r_2)} p_1 \in \Delta_c$ . Let  $c = (\langle p_1, \gamma w' \rangle, \theta)$  and  $c' = (\langle p, \gamma w' \rangle, \theta')$  s.t.  $\theta = \theta' \setminus \{r_1\} \cup \{r_2\}$ . Then, if  $c \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$ , then necessarily,  $c' \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$ . Moreover,  $c' \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$  iff either  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$  or  $p \in G$  (i.e.  $B(p) = 1$ ). Thus, we need to  
570 impose that if the automaton  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$  contains a path of the form  $(p_1, \theta) \xrightarrow{[\gamma, b_1]}_T q \xrightarrow{[w', b_2]}_T q_f$  (where  $q_f \in F$ ) that accepts the labelled configuration  $[c, b]$ ,  $b = b_1 \vee b_2$  ( $b = 1$  means  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ ), then necessarily, the automaton  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$  should also accept the labelled configuration  $[c', b \vee B(p)]$ . This configuration is accepted by the run  $(p, \theta') \xrightarrow{[\gamma, B(p) \vee b_1]}_T$   
575  $q \xrightarrow{[w', b_2]}_T q_f$  added by rule  $(\alpha_2)$ .

#### 4.3.1. Example

Let us illustrate the procedure by an example. Consider the SM-BPDS  $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$  shown in the left (i.e. part a) of Fig.2 where  $P = \{p_1, p_2, p_3, p'\}$ ,  $\Delta = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$ ,  $\Delta_c = \{r'\}$  and  $G = \{p_2\}$ . We show  
580 how to compute a  $\mathcal{BP}$ -automaton  $\mathcal{A}_{pre^*}(\langle (p', \epsilon), \theta' \rangle)$ . Let  $\mathcal{A}$  be the automaton that accepts the set  $\{(\langle p', \epsilon \rangle, \theta')\}$  with  $Q = I = F = \{(p', \theta')\}$ . Initially,  $T$  is empty. The result is obtained through the following steps:

1. First we note that  $(p', \theta') \xrightarrow{[\epsilon, b]}_T (p', \theta'), b = 0$  holds. Since  $\langle p', \epsilon \rangle$  occurs on the right hand side of rule  $r_1 \in \theta'$  and  $r_3 \in \theta'$ , moreover,  $p_1 \notin G$  i.e.  $B(p_1) = 0$  and  $p' \notin G$  i.e.  $B(p') = 0$ , then Rule  $(\alpha_1)$  adds the transition  
585  $(p_1, \theta') \xrightarrow{[\gamma_1, b_1]}_T (p', \theta')$  with  $b_1 = B(p_1) \vee b = 0$  and  $(p', \theta') \xrightarrow{[\gamma', b_2]}_T (p', \theta')$  with  $b_2 = B(p') \vee b = 0$ .



$\Delta$  :

$r_1 : \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \epsilon \rangle$     $r_2 : \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma' \rangle$   
 $r_3 : \langle p', \gamma' \rangle \hookrightarrow \langle p', \epsilon \rangle$     $r_4 : \langle p_2, \gamma' \rangle \hookrightarrow \langle p_3, \gamma_2 \rangle$   
 $r_5 : \langle p', \gamma' \rangle \hookrightarrow \langle p_2, \gamma' \rangle$     $r_6 : \langle p_2, \gamma' \rangle \hookrightarrow \langle p_1, \gamma_1 \rangle$   
 $r_7 : \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \rangle$

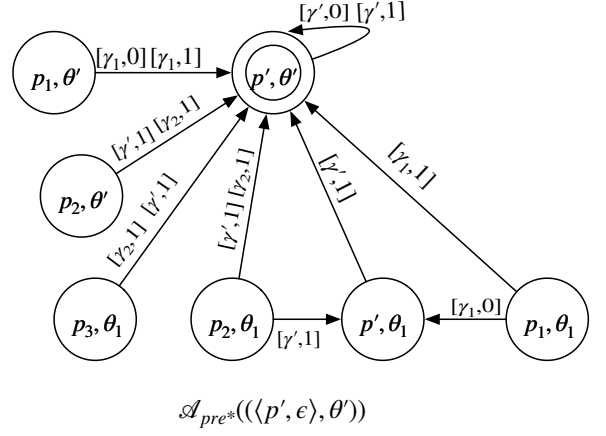
$\Delta_c$  :

$r' : p_3 \xrightarrow{(\{r_4\}, \{r_3\})} p_2$

$\theta' = \{r_1, r_2, r_3, r_5, r_6, r_7, r'\}$ ,  $\theta_1 = \{r_1, r_2, r_4, r_5, r_6, r_7, r'\}$

$G = \{p_2\}$ ,  $\Gamma = \{\gamma_1, \gamma_2, \gamma'\}$

a



b

Figure 2:  $\mathcal{BP}$ -automaton  $\mathcal{A}_{pre^*}((\langle p', \epsilon \rangle, \theta'))$

2. Now that we have the path  $(p_1, \theta') \xrightarrow{[\gamma_1, b_1]}_T (p', \theta') \xrightarrow{[\gamma', b_2]}_T (p', \theta')$ ,  $b_1 = 0$  and  $b_2 = 0$ , since  $r_2 \in \theta'$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$  then Rule  $(\alpha_1)$  adds  $(p_2, \theta') \xrightarrow{[\gamma_2, b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 \vee b_2 = 1$ .  
590
3. Since we have  $(p_1, \theta') \xrightarrow{[\gamma_1, b_1]}_T (p', \theta')$ ,  $b_1 = 0$  and  $r_6 \in \theta'$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$ , then Rule  $(\alpha_1)$  adds the transition  $(p_2, \theta') \xrightarrow{[\gamma', b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 = 1$ .
4. Now we have  $(p_2, \theta') \xrightarrow{[\gamma_2, b_1]}_T (p', \theta')$ ,  $b_1 = 1$  and  $r_7 \in \theta'$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$ , then Rule  $(\alpha_1)$  adds the transition  $(p_1, \theta') \xrightarrow{[\gamma_1, b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 = 1$ .  
595
5. Now we have  $(p_2, \theta') \xrightarrow{[\gamma', b_1]}_T (p', \theta')$ ,  $b_1 = 1$  and  $r_5 \in \theta'$ , moreover,  $p' \notin G$  i.e.  $B(p') = 0$ , then Rule  $(\alpha_1)$  adds the transition  $(p', \theta') \xrightarrow{[\gamma', b]}_T (p', \theta')$  with  $b = B(p') \vee b_1 = 1$ .
6. Since we have  $(p_2, \theta') \xrightarrow{[\gamma', b_1]}_T (p', \theta')$  and  $(p_2, \theta') \xrightarrow{[\gamma_2, b_2]}_T (p', \theta')$ ,  $b_1 = 1, b_2 = 1$ , the self-modifying rule  $r' \in \theta'$  can be applied. Moreover,  $p_3 \notin G$  i.e.  $B(p_3) = 0$  Thus, Rule  $(\alpha_2)$  adds  $(p_3, \theta_1) \xrightarrow{[\gamma_2, b]}_T (p', \theta')$  and  $(p_3, \theta_1) \xrightarrow{[\gamma', b']}_T (p', \theta')$  where  $\theta_1 = (\theta' \setminus \{r_3\}) \cup \{r_4\}$  with  $b = B(p_3) \vee b_1 = 1, b' = B(p_3) \vee b_2 = 1$ .  
600
7. Now we have  $(p_3, \theta_1) \xrightarrow{[\gamma_2, b_1]}_T (p', \theta')$ ,  $b_1 = 1$  and  $r_4 \in \theta_1$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$ , then Rule  $(\alpha_1)$  adds the transition  $(p_2, \theta_1) \xrightarrow{[\gamma', b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 = 1$ .  
605
8. Since  $(p_2, \theta_1) \xrightarrow{[\gamma', b_1]}_T (p', \theta')$ ,  $b_1 = 1$  and  $r_5 \in \theta_1$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$ , then Rule  $(\alpha_1)$  adds the transition  $(p', \theta_1) \xrightarrow{[\gamma', b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 = 1$ .  
610

9. We note that  $(p', \theta_1) \xrightarrow{[\epsilon, b_1]}_T (p', \theta_1)$ ,  $b_1 = 0$  holds. Since  $\langle p', \epsilon \rangle$  occurs on the right hand side of rule  $r_1 \in \theta_1$ , moreover,  $p_1 \notin G$  i.e.  $B(p_1) = 0$  and  $p' \notin G$  i.e.  $B(p') = 0$ , then Rule  $(\alpha_1)$  adds the transition  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b]}_T (p', \theta')$  with  $b = B(p_1) \vee b_1 = 0$ .
- 615 10. Now we have  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b_1]}_T (p', \theta_1)$ ,  $b_1 = 0$  and  $r_6 \in \theta_1$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$ , then Rule  $(\alpha_1)$  adds the transition  $(p_2, \theta_1) \xrightarrow{[\gamma', b]}_T (p', \theta_1)$  with  $b = B(p_2) \vee b_1 = 1$ .
11. Now that we have the path  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b_1]}_T (p', \theta_1) \xrightarrow{[\gamma', b_2]}_T (p', \theta')$ , since  $r_2 \in \theta_1$ ,  $b_1 = 0, b_2 = 1$ , moreover,  $p_2 \in G$  i.e.  $B(p_2) = 1$  then Rule  $(\alpha_1)$  adds  $(p_2, \theta_1) \xrightarrow{[\gamma_2, b]}_T (p', \theta')$  with  $b = B(p_2) \vee b_1 \vee b_2 = 1$ .
- 620 12. Since we have  $(p_2, \theta_1) \xrightarrow{[\gamma_2, b_1]}_T (p', \theta')$ ,  $b_1 = 1$  and  $r_7 \in \theta_1$ , Rule  $(\alpha_1)$  adds the transition  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b]}_T (p', \theta')$  with  $b = B(p_1) \vee b_1 = 1$ .
13. No further additions are possible. Thus, the procedure terminates.

The result is depicted in the right side of Fig.2

#### 625 4.3.2. Proof

Before proving that our construction is correct, we introduce the following definition:

**Definition 6.** Let  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$  be the labelled  $\mathcal{P}$ -automaton computed by the saturation procedure above. In this section, we use  $\xrightarrow{i}_T$  to denote the transition relation of  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$  obtained after adding  $i$  transitions using the saturation procedure above. Let us notice that due to the fact that initially  $Q = \{(p', \theta')\}$  and due to rules  $(\alpha_1)$  and  $(\alpha_2)$  that at step  $i$  add only transitions of the form  $(p, \theta) \xrightarrow{\gamma}_T q$  for a state  $q$  that is already in the automaton at step  $i - 1$ , then, states of  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$  are all of the form  $(p, \theta)$  for  $p \in P$  and  $\theta \subseteq \Delta \cup \Delta_c$ .

630  
635

We can show that:

**Lemma 2.** Let  $p, p'' \in P$  and  $\theta, \theta'' \subseteq \Delta \cup \Delta_c$ . Let  $w \in \Gamma^*$  and  $b \in \{0, 1\}$ . If a path  $(p, \theta) \xrightarrow{[w, b]}_T (p'', \theta'')$  is in  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$ , then  $\langle p, w \rangle, \theta \Rightarrow^* \langle p'', \epsilon \rangle, \theta''$ . Moreover, if  $b = 1$ , then  $\langle p, w \rangle, \theta \Rightarrow^r \langle p'', \epsilon \rangle, \theta''$ .

640 **Proof:** Initially, the automaton contains no transitions. Let  $i$  be an index such that  $(p, \theta) \xrightarrow{[w, b]}_i^T (p'', \theta'')$  holds. We proceed by induction on  $i$ .

**Basis.**  $i = 0$ , then  $(p'', \theta'') \xrightarrow{[\epsilon, 0]}_T (p'', \theta'')$ . This means  $p'' = p', \theta'' = \theta'$ . Since initially  $Q = \{(p', \theta')\}$ , then  $\langle p'', \epsilon \rangle, \theta'' \Rightarrow^* \langle p'', \epsilon \rangle, \theta''$  always holds.

**Step.**  $i > 0$ . Let  $t = ((p_1, \theta_1), [\gamma, b_1], (p_0, \theta_0))$  be the  $i$ -th transition added to  $\mathcal{A}_{pre^*}$  and  $j$  be the number of times that  $t$  is used in the path  $(p, \theta) \xrightarrow{[w, b]}_i^T (p'', \theta'')$ . The proof is by induction on  $j$ . If  $j = 0$ , then we have  $(p, \theta) \xrightarrow{[w, b]}_{i-1}^T (p'', \theta'')$  in the automaton, and we apply the induction hypothesis (induction on  $i$ ) then we obtain  $\langle p, w \rangle, \theta \Rightarrow^* \langle p'', \epsilon \rangle, \theta''$ . So assume

that  $j > 0$ . Then, there exist  $u, v \in \Gamma^*$ ,  $b', b'' \in \{0, 1\}$  such that  $w = u\gamma v$ ,  
650  $b = b' \vee b_1 \vee b''$  and

$$(p, \theta) \xrightarrow[i-1]{[u, b']}_T (p_1, \theta_1) \xrightarrow[i]{[\gamma, b_1]}_T (p_0, \theta_0) \xrightarrow[i]{[v, b'']}_T (p'', \theta'') \quad (1)$$

The application of the induction hypothesis (induction on  $i$ ) to  $(p, \theta) \xrightarrow[i-1]{[u, b']}_T$   
 $(p_1, \theta_1)$  gives that

$$(\langle p, u \rangle, \theta) \Rightarrow^* (\langle p_1, \epsilon \rangle, \theta_1), \text{ moreover, if } b' = 1, (\langle p, u \rangle, \theta) \Rightarrow^r (\langle p_1, \epsilon \rangle, \theta_1) \quad (2)$$

There are 2 cases depending on whether transition  $t$  was added by saturation rule  $\alpha_1$  or  $\alpha_2$ .

1. Case  $t$  was added by rule  $\alpha_1$ : There exist  $p_2 \in P$  and  $w_2 \in \Gamma^*$  such that

$$r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, w_2 \rangle \in \Delta \cap \theta_1 \quad (3)$$

and  $\mathcal{A}_{pre^*}$  contains the following path:

$$\pi' = (p_2, \theta_1) \xrightarrow[i-1]{[w_2, b_2]}_T (p_0, \theta_0) \xrightarrow[i]{[v, b'']}_T (p'', \theta''), \quad b_1 = b_2 \vee B(p_1) \quad (4)$$

Applying the transition rule  $r$ , we get that

$$(\langle p_1, \gamma v \rangle, \theta_1) \Rightarrow (\langle p_2, w_2 v \rangle, \theta_1) \quad (5)$$

By induction on  $j$  (since transition  $t$  is used  $j - 1$  times in  $\pi'$ ), we get from  
(4) that

$$(\langle p_2, w_2 v \rangle, \theta_1) \Rightarrow^* (\langle p'', \epsilon \rangle, \theta'') \text{ moreover, if } b_2 \vee b'' = 1, (\langle p_2, w_2 v \rangle, \theta_1) \Rightarrow^r (\langle p'', \epsilon \rangle, \theta'') \quad (6)$$

Putting (2), (5) and (6) together, we can obtain that

$$(\langle p, w \rangle, \theta) = (\langle p, u\gamma v \rangle, \theta) \Rightarrow^* (\langle p_1, \gamma v \rangle, \theta_1) \Rightarrow (\langle p_2, w_2 v \rangle, \theta_1) \Rightarrow^* (\langle p'', \epsilon \rangle, \theta'')$$

Furthermore, if  $b = b' \vee b_1 \vee b'' = 1$ , then  $b' = 1$  or  $b_1 \vee b'' = 1$ .

655 For the first case,  $b' = 1$ , then we can have  $(\langle p, u \rangle, \theta) \Rightarrow^r (\langle p_1, \epsilon \rangle, \theta_1)$   
from (2). Thus, we can obtain that  $(\langle p, u\gamma v \rangle, \theta) \Rightarrow^r (\langle p_1, \gamma v \rangle, \theta_1) \Rightarrow^*$   
 $(\langle p'', \epsilon \rangle, \theta'')$  i.e.  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p'', \epsilon \rangle, \theta'')$ .

660 The second case  $b_1 \vee b'' = 1$  i.e.  $B(p_1) \vee b_2 \vee b'' = 1$  implies that  
 $B(p_1) = 1$  (that means  $p_1 \in G$  and  $(\langle p_1, \gamma v \rangle, \theta_1) \Rightarrow^r (\langle p'', \epsilon \rangle, \theta'')$ ) or  
 $b_2 \vee b'' = 1$  (that implies  $(\langle p_2, w_2 v \rangle, \theta_1) \Rightarrow^r (\langle p'', \epsilon \rangle, \theta'')$  from (6)). Therefore,  
 $(\langle p, w \rangle, \theta_1) \Rightarrow^r (\langle p'', \epsilon \rangle, \theta'')$ .

2. Case  $t$  was added by rule  $\alpha_2$ : there exist  $p_2 \in P$  and  $\theta_2 \subseteq \Delta \cup \Delta_c$  such that

$$r = p_1 \xrightarrow{(\sigma, \sigma')} p_2 \in \Delta_c \cap \theta_2, \theta_2 = (\theta_1 \setminus \sigma) \cup \sigma' \quad (7)$$

and the following path in the current automaton ( self-modifying rule  
won't change the stack) with  $r \in \theta_2$ :

$$(p_2, \theta_2) \xrightarrow[i-1]{[\gamma, b'_1]}_T (p_0, \theta_0) \xrightarrow[i]{[v, b'']}_T (p'', \theta''), \quad b_1 = B(p_1) \vee b'_1 \quad (8)$$

Applying the transition rule, we can get from (7) that

$$\langle \langle p_1, \gamma v \rangle, \theta_1 \rangle \Rightarrow \langle \langle p_2, \gamma v \rangle, \theta_2 \rangle \quad (9)$$

We can apply the induction hypothesis (on  $j$ ) to (8), and obtain

$$\langle \langle p_2, \gamma v \rangle, \theta_2 \rangle \Rightarrow^* \langle \langle p'', \epsilon \rangle, \theta'' \rangle, \text{ moreover, if } b'_1 \vee b'' = 1, \langle \langle p_2, \gamma v \rangle, \theta_2 \rangle \Rightarrow^r \langle \langle p'', \epsilon \rangle, \theta'' \rangle \quad (10)$$

From (2),(9) and (10), we get

$$\langle \langle p, w \rangle, \theta \rangle = \langle \langle p, u\gamma v \rangle, \theta \rangle \Rightarrow^* \langle \langle p_1, \gamma v \rangle, \theta_1 \rangle \Rightarrow \langle \langle p_2, \gamma v \rangle, \theta_2 \rangle \Rightarrow^* \langle \langle p'', \epsilon \rangle, \theta'' \rangle$$

Furthermore, if  $b = b' \vee b_1 \vee b'' = 1$ , then  $b' = 1$  or  $b_1 \vee b'' = 1$ .

For the first case,  $b' = 1$ , then we can have  $\langle \langle p, u \rangle, \theta \rangle \Rightarrow^r \langle \langle p_1, \epsilon \rangle, \theta_1 \rangle$  from (2). Thus, we can obtain that  $\langle \langle p, u\gamma v \rangle, \theta \rangle \Rightarrow^r \langle \langle p_1, \gamma v \rangle, \theta_1 \rangle \Rightarrow^*$   
665  $\langle \langle p'', \epsilon \rangle, \theta'' \rangle$  i.e.  $\langle \langle p, w \rangle, \theta \rangle \Rightarrow^r \langle \langle p'', \epsilon \rangle, \theta'' \rangle$ . The second case  $b_1 \vee b'' = 1$   
i.e.  $B(p_1) \vee b'_1 \vee b'' = 1$  implies that  $B(p_1) = 1$  (that means  $p_1 \in G$  and  
 $\langle \langle p_1, \gamma v \rangle, \theta_1 \rangle \Rightarrow^r \langle \langle p', \epsilon \rangle, \theta' \rangle$ ) or  $b'_1 \vee b'' = 1$  (that implies  $\langle \langle p_2, \gamma v \rangle, \theta_2 \rangle \Rightarrow^r$   
670  $\langle \langle p'', \epsilon \rangle, \theta'' \rangle$  from (10)) i.e.  $\langle \langle p, w \rangle, \theta_1 \rangle \Rightarrow^r \langle \langle p', \epsilon \rangle, \theta' \rangle$ . Therefore, we can  
get that if  $b = 1$ , then  $\langle \langle p, w \rangle, \theta_1 \rangle \Rightarrow^r \langle \langle p'', \epsilon \rangle, \theta'' \rangle$ .

□

**Lemma 3.** *If there is a labelled configuration  $[\langle \langle p, w \rangle, \theta \rangle, b]$  such that  $\langle \langle p, w \rangle, \theta \rangle \Rightarrow^*$   
 $\langle \langle p', \epsilon \rangle, \theta' \rangle$ , then there is a path  $\langle p, \theta \rangle \xrightarrow{[w, b]}_T \langle p', \theta' \rangle$  in  $\mathcal{A}_{pre^*}(\langle \langle p', \epsilon \rangle, \theta' \rangle)$ . More-  
675 over, if  $\langle \langle p, w \rangle, \theta \rangle \Rightarrow^r \langle \langle p', \epsilon \rangle, \theta' \rangle$ , then  $b = 1$ .*

**Proof:** Assume  $\langle \langle p, w \rangle, \theta \rangle \xrightarrow{i} \langle \langle p', \epsilon \rangle, \theta' \rangle$ . We proceed by induction on  $i$ .

**Basis.**  $i = 0$ . Then  $\theta = \theta', p' = p$  and  $w = \epsilon$ . Initially, we have that  $Q = \{ \langle p', \theta' \rangle \}$ , therefore, by the definition of  $\rightarrow_T$ , we have  $\langle p', \theta' \rangle \xrightarrow{\epsilon}_T \langle p', \theta' \rangle$ . We cannot have  $\langle \langle p', \epsilon \rangle, \theta' \rangle \Rightarrow^r \langle \langle p', \epsilon \rangle, \theta' \rangle$  in 0-step.

**Step.**  $i > 0$ . Then there exists a configuration  $\langle \langle p'', u \rangle, \theta'' \rangle$  such that

$$\langle \langle p, w \rangle, \theta \rangle \Rightarrow \langle \langle p'', u \rangle, \theta'' \rangle \xrightarrow{i-1} \langle \langle p', \epsilon \rangle, \theta' \rangle$$

680 We apply the induction hypothesis to  $\langle \langle p'', u \rangle, \theta'' \rangle \xrightarrow{i-1} \langle \langle p', \epsilon \rangle, \theta' \rangle$ , and obtain  
that there exists in  $\mathcal{A}_{pre^*}(\langle \langle p', \epsilon \rangle, \theta' \rangle)$  a path  $\langle p'', \theta'' \rangle \xrightarrow{[u, b'']}_T \langle p', \theta' \rangle$ . If  $\langle \langle p'', u \rangle, \theta'' \rangle \Rightarrow^r$   
 $\langle \langle p', \epsilon \rangle, \theta' \rangle$ ,  $b'' = 1$ .

Let  $(p_0, \theta_0)$  be a state of  $\mathcal{A}_{pre^*}$ . Let  $w_1, u_1 \in \Gamma^*, \gamma \in \Gamma, b''_0, b''_1 \in \{0, 1\}$  be  
such that  $w = \gamma w_1, u = u_1 w_1, b'' = b''_0 \vee b''_1$  and

$$\langle p'', \theta'' \rangle \xrightarrow{[u_1, b''_0]}_T (p_0, \theta_0) \xrightarrow{[w_1, b''_1]}_T \langle p', \theta' \rangle \quad (1)$$

There are two cases depending on which rule is applied to get  $\langle \langle p, w \rangle, \theta \rangle \Rightarrow$   
 $\langle \langle p'', u \rangle, \theta'' \rangle$ .

685

1. Case  $(\langle p, w \rangle, \theta) \Rightarrow (\langle p'', u \rangle, \theta'')$  is obtained by a rule of the form:  $\langle p, \gamma \rangle \hookrightarrow \langle p'', u_1 \rangle \in \Delta$ . In this case,  $\theta'' = \theta$ . By the saturation rule  $\alpha_1$ , we have

$$(p, \theta'') \xrightarrow{[\gamma, b_0]}_T (p_0, \theta_0), \quad b_0 = B(p) \vee b_0'' \quad (2)$$

Putting (1) and (2) together, we can obtain that

$$\pi = (p, \theta'') \xrightarrow{[\gamma, b_0]}_T (p_0, \theta_0) \xrightarrow{[w_1, b_1']}_T (p', \theta') \quad (3)$$

Thus,  $(p, \theta'') \xrightarrow{[\gamma w_1, b_0 \vee b_1']}_T (p', \theta')$  i.e.  $(p, \theta) \xrightarrow{[w, b]}_T (p', \theta')$  where  $b = b_0 \vee b_1''$ .

2. Case  $(\langle p, w \rangle, \theta) \Rightarrow (\langle p'', u \rangle, \theta'')$  is obtained by a rule of the form  $p \xrightarrow{(\sigma, \sigma')} p'' \in \Delta_c$  i.e.  $\theta'' \neq \theta$ . In this case,  $u_1 = \gamma$ . By the saturation rule  $\beta_2$ , we obtain that

$$(p, \theta) \xrightarrow{[\gamma, b_0]}_T (p_0, \theta_0) \text{ where } \theta'' = \theta \setminus \{r_1\} \cup \{r_2\}, \quad b_0 = B(p) \vee b_0''. \quad (4)$$

Putting (1) and (4) together, we have the following path

$$(p, \theta) \xrightarrow{[\gamma, b_0]}_T (p_0, \theta_0) \xrightarrow{[w_1, b_1']}_T (p', \theta') \text{ i.e. } (p, \theta) \xrightarrow{[w, b]}_T (p', \theta') \text{ where } b = b_0 \vee b_1'' \quad (5)$$

690 Furthermore, if  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ , then  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p'', u \rangle, \theta'')$  or  $(\langle p'', u \rangle, \theta'') \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ .

For the first case,  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p'', u \rangle, \theta'')$ , then  $p \in G$  i.e.  $B(p) = 1$ . For the second case,  $(\langle p'', u \rangle, \theta'') \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ , we can get  $b'' = 1$  (from induction hypothesis). Thus,  $b = b_0 \vee b_1'' = B(p) \vee b_0'' \vee b_1'' = B(p) \vee b'' = 1$ . Therefore, if  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ , then we can obtain  $b = 1$ .

695

□

From these two lemmas, we get:

**Theorem 4.2.** *Let  $[c, b]$  be a labelled configuration. Then  $[c, b]$  is in  $L(\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta'))$  iff  $c \in pre^*(\langle p', \epsilon \rangle, \theta')$ . Moreover,  $c \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$  iff  $b = 1$ .*

700

**Proof:** Let  $[(\langle p, w \rangle, \theta), b]$  be a labelled configuration of  $pre^*(\langle p', \epsilon \rangle, \theta')$ . Then  $(\langle p, w \rangle, \theta) \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$ . By Lemma 2, we can obtain that there exists a path  $(p, \theta) \xrightarrow{[w, b]}_T (p', \theta')$  in  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$ . So  $[(\langle p, w \rangle, \theta), b]$  is in  $L(\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta'))$ . Moreover, if  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ , then  $b = 1$ .

Conversely, let  $[(\langle p, w \rangle, \theta), b]$  be a labelled configuration accepted by  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$

705

i.e. there exists a path  $(p, \theta) \xrightarrow{[w, b]}_T (p', \theta')$  in  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$ . By Lemma 3,  $(\langle p, w \rangle, \theta) \Rightarrow^* (\langle p', \epsilon \rangle, \theta')$  i.e.  $(\langle p, w \rangle, \theta) \in pre^*(L(A))$ . Moreover, if  $b = 1$ ,  $(\langle p, w \rangle, \theta) \Rightarrow^r (\langle p', \epsilon \rangle, \theta')$ .

□

710 4.4. Computing the Head Reachability Graph  $\mathcal{G}$

Based on the definition of the Head Reachability Graph  $\mathcal{G}$ , and on Theorem 4.2, we can compute  $\mathcal{G}$  as follows. Initially,  $\mathcal{G}$  has no edges.

- 715  $\alpha'_1$ : if  $r_c : p \xrightarrow{(\sigma, \sigma')} p' \in \Delta_c$ , then for every phase  $\theta$  such that  $r_c \in \theta$  and every  $\gamma \in \Gamma$ , we add the edge  $((p, \gamma), \theta) \xrightarrow{B(p)} ((p', \gamma), \theta_0)$  to the graph  $\mathcal{G}$ , where  $\theta_0 = \theta \setminus \sigma \cup \sigma'$ .
- $\alpha'_2$ : if  $r : \langle p, \gamma \rangle \hookrightarrow \langle p_0, \gamma_0 \rangle \in \Delta$ , then for every phase  $\theta$  such that  $r \in \theta$ , we add the edge  $((p, \gamma), \theta) \xrightarrow{B(p)} ((p_0, \gamma_0), \theta)$  to the graph  $\mathcal{G}$ .
- 720  $\alpha'_3$ : if  $r : \langle p, \gamma \rangle \hookrightarrow \langle p_0, \gamma_0 \gamma' \rangle \in \Delta$ , then for every phase  $\theta$  such that  $r \in \theta$ , we add to the graph  $\mathcal{G}$  the edge  $((p, \gamma), \theta) \xrightarrow{B(p)} ((p_0, \gamma_0), \theta)$ . Moreover, for every control point  $p' \in P$  and phase  $\theta'$  such that  $\mathcal{A}_{pre^*}(\langle p', \epsilon \rangle, \theta')$  contains a transition of the form  $t = (p_0, \theta) \xrightarrow{[\gamma_0, b]}_T (p', \theta')$ , we add to the graph  $\mathcal{G}$  the edge  $((p, \gamma), \theta) \xrightarrow{b \vee B(p)} ((p', \gamma'), \theta')$ .

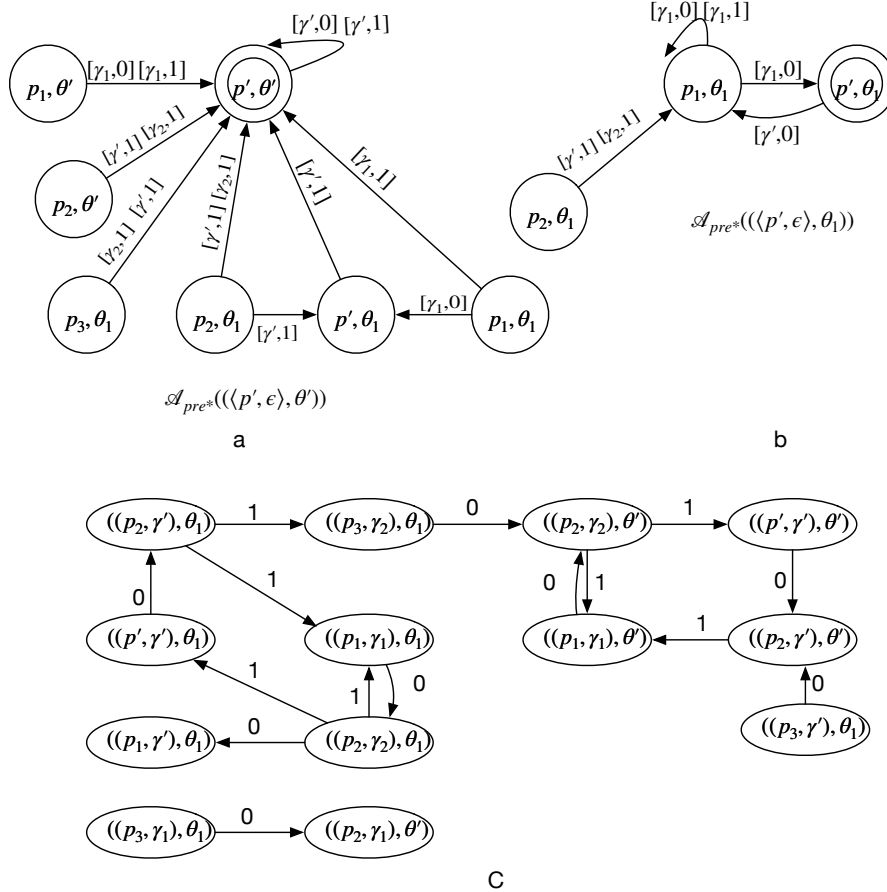


Figure 3: An Example of the SM-BPDS and the graph  $\mathcal{G}$

Items  $\alpha'_1$  and  $\alpha'_2$  are obvious. They respectively correspond to item 1 and item 2 of Definition 4 (since  $B(p) = 1$  iff  $p \in G$ ). Item  $\alpha'_3$  is based on Lemma 1 and on item 3 of Definition 4. Indeed, it follows from Lemma 1 that  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta'\rangle)$  contains a transition of the form  $(p_0, \theta) \xrightarrow{[\gamma_0, b]}_T(p', \theta')$  implies that  $\langle(p_0, \gamma_0), \theta\rangle \Rightarrow^* \langle(p', \epsilon), \theta'\rangle$ , and if  $b = 1$ , then  $\langle(p_0, \gamma_0), \theta\rangle \Rightarrow^r \langle(p', \epsilon), \theta'\rangle$ . Thus, in this case, the edge  $((p, \gamma), \theta) \xrightarrow{b \vee B(p)} \langle(p', \gamma'), \theta'\rangle$  is added to  $\mathcal{G}$  (item 3 of Definition 4) since  $\langle p, \gamma \rangle \hookrightarrow \langle p_0, \gamma_0 \gamma' \rangle \in \Delta$ .

**Example:** Let us illustrate the procedure by an example. Consider the previous example shown in Fig.2.  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta'\rangle)$  is shown in Fig. 3 (a) and  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta_1\rangle)$  is shown in Fig. 3 (b). The result  $\mathcal{G}$  shown in Fig. 3 (c) is obtained as follows:

1. Since  $r_5 \in \theta', r_5 \in \theta_1$  and  $p' \notin G$  i.e.  $B(p') = 0$ , Rule  $\alpha'_2$  adds edges  $\langle(p', \gamma'), \theta'\rangle \xrightarrow{b} \langle(p_2, \gamma'), \theta'\rangle$  and  $\langle(p', \gamma'), \theta_1\rangle \xrightarrow{b} \langle(p_2, \gamma'), \theta_1\rangle$ ,  $b = B(p') = 0$  to  $\mathcal{G}$ .
2. Because  $r_6 \in \theta', r_6 \in \theta_1$  and  $p_2 \in G$  i.e.  $B(p_2) = 1$ , Rule  $\alpha'_2$  adds edges  $\langle(p_2, \gamma'), \theta'\rangle \xrightarrow{b} \langle(p_1, \gamma_1), \theta'\rangle$  and  $\langle(p_2, \gamma'), \theta_1\rangle \xrightarrow{b} \langle(p_1, \gamma_1), \theta_1\rangle$ ,  $b = B(p_2) = 1$  to  $\mathcal{G}$ .
3. Because  $r_7 \in \theta', r_7 \in \theta_1$  and  $p_1 \notin G$  i.e.  $B(p_1) = 0$ , Rule  $\alpha'_2$  adds edges  $\langle(p_1, \gamma_1), \theta'\rangle \xrightarrow{b} \langle(p_2, \gamma_2), \theta'\rangle$  and  $\langle(p_1, \gamma_1), \theta_1\rangle \xrightarrow{b} \langle(p_2, \gamma_2), \theta_1\rangle$ ,  $b = B(p_1) = 0$  to  $\mathcal{G}$ .
4. Because  $r_4 \in \theta_1$  and  $p_2 \in G$  i.e.  $B(p_2) = 1$ , Rule  $\alpha'_2$  adds the edge  $\langle(p_2, \gamma'), \theta_1\rangle \xrightarrow{b} \langle(p_3, \gamma_2), \theta_1\rangle$ ,  $b = B(p_2) = 1$  to  $\mathcal{G}$ .
5. Now we have  $r' \in \theta_1$  and  $r_4 \in \theta_1$ , for every  $\gamma \in \Gamma$ , Rule  $\alpha'_1$  adds edges  $\langle(p_3, \gamma_1), \theta_1\rangle \xrightarrow{b} \langle(p_2, \gamma_1), \theta'\rangle$ ,  $\langle(p_3, \gamma_2), \theta_1\rangle \xrightarrow{b} \langle(p_2, \gamma_2), \theta'\rangle$  and  $\langle(p_3, \gamma'), \theta_1\rangle \xrightarrow{b} \langle(p_2, \gamma'), \theta'\rangle$ ,  $b = B(p_3) = 0$  to  $\mathcal{G}$ .
6. Since  $r_2 \in \theta'$  and  $p_2 \in G$  i.e.  $B(p_1) = 1$ , Rule  $\alpha'_3$  first adds to the graph  $\mathcal{G}$  the edge  $\langle(p_2, \gamma_2), \theta'\rangle \xrightarrow{b} \langle(p_1, \gamma_1), \theta'\rangle$ ,  $b = B(p_2) = 1$ . Then only  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta'\rangle)$  contains transitions of the form  $(p_1, \theta') \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta')$  i.e. transition  $(p_1, \theta') \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta')$ ,  $b'_1 = 0$  and  $(p_1, \theta') \xrightarrow{[\gamma_1, b'_2]}_T(p', \theta')$ ,  $b'_2 = 1$ . Then Rule  $\alpha'_3$  adds edges  $\langle(p_2, \gamma_2), \theta'\rangle \xrightarrow{b_1} \langle(p', \gamma'), \theta'\rangle$  with  $b_1 = B(p_2) \vee b'_1 = 1$  and  $\langle(p_2, \gamma_2), \theta'\rangle \xrightarrow{b_2} \langle(p', \gamma'), \theta'\rangle$  with  $b_2 = B(p_2) \vee b'_2 = 1$  to  $\mathcal{G}$ .
7. Since  $r_2 \in \theta_1$  and  $p_2 \in G$  i.e.  $B(p_1) = 1$ , Rule  $\alpha'_3$  first adds to the graph  $\mathcal{G}$  the edge  $\langle(p_2, \gamma_2), \theta_1\rangle \xrightarrow{b} \langle(p_1, \gamma_1), \theta_1\rangle$ ,  $b = B(p_1) = 1$ . Then only  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta'\rangle)$  contain transitions of the form  $(p_1, \theta') \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta')$  and  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta_1\rangle)$  contain transitions of the form  $(p_1, \theta') \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta_1)$  i.e. transition  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta_1)$  in  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta'\rangle)$  and  $(p_1, \theta_1) \xrightarrow{[\gamma_1, b'_1]}_T(p', \theta_1)$  in  $\mathcal{A}_{pre^*}(\langle(p', \epsilon), \theta_1\rangle)$ ,  $b_1 = 0$ . Then Rule  $\alpha'_3$  adds the edge  $\langle(p_2, \gamma_2), \theta_1\rangle \xrightarrow{b_2} \langle(p', \gamma'), \theta_1\rangle$ ,  $b_2 = b \vee b_1 = 1$  to  $\mathcal{G}$ .
8. No further additions are possible. Thus, the procedure terminates.

The result is depicted in Fig. 3 (c). By finding 1-labelled loops in  $\mathcal{G}$ , the repeating heads are

$$\{((p_2, \gamma_2), \theta_1), ((p', \gamma'), \theta_1), ((p_2, \gamma'), \theta_1), ((p_1, \gamma_1), \theta_1)\}$$

and

$$\{((p_2, \gamma_2), \theta'), ((p', \gamma'), \theta'), ((p_2, \gamma'), \theta'), ((p_1, \gamma_1), \theta')\}.$$

## 5. Experiments

### 5.1. Our approach vs. standard LTL for PDSs

We implemented our approach in a tool<sup>1</sup> and we compared its performance  
765 against the approaches that consist in translating the SM-PDS to an equivalent standard (or symbolic) PDS, and then applying the standard LTL model checking algorithms implemented in the PDS model-checker tool Moped [25]. All our experiments were run on Ubuntu 16.04 with a 2.7 GHz CPU, 2GB of memory. To perform the comparison, we randomly generate several SM-PDSs  
770 and LTL formulas of different sizes. For this, we use the function `int rand(void)` several times to randomly generate states and transitions. The results (CPU Execution time) are shown in Table 1. **Column Size** is the size of SM-PDS ( $S_1$  for non-modifying transitions  $\Delta$  and  $S_2$  for modifying transitions  $\Delta_c$ ). **Column LTL** gives the size of the transitions of the Büchi automaton generated from the LTL formula (using the tool LTL2BA[28]). **Column SM-PDS** gives the cost of our direct algorithm presented in this paper. **Column PDS** shows the cost it takes to get the equivalent PDS from the SM-PDS. **Column Result** reports the cost it takes to run the LTL PDS model-checker Moped [25] for the PDS we got. **Column Total** is the total cost it takes to translate the SM-PDS into  
780 a PDS and then apply the standard LTL model checking algorithm of Moped (Total=PDS+Result). **Column Symbolic PDS** reports the cost it takes to get the equivalent Symbolic PDS from the SM-PDS. **Column Result<sub>1</sub>** is the cost to run the Symbolic PDS LTL model-checker Moped. **Column Total<sub>1</sub>** is the total cost it takes to translate the SM-PDS into a symbolic PDS and then apply  
785 the standard LTL model checking algorithm of Moped. You can see that our direct algorithm (**Column SM-PDS**) is much more efficient than translating the SM-PDS to an equivalent (symbolic) PDS, and then run the standard LTL model-checker Moped. **Translating the SM-PDS to a standard PDS may take more than 20 days, whereas our direct algorithm takes only a few seconds.** Moreover, since the obtained standard (symbolic) PDS is huge,  
790 Moped failed to handle several cases (the time limit that we set for Moped is 20 minutes), whereas our tool was able to deal with all the cases in only a few seconds.

### 5.2. Malicious Behavior Detection on Self-Modifying Code

#### 795 5.2.1. Specifying Malicious Behaviors using LTL.

As described in [11], several malicious behaviors can be described by LTL formulas. We give in what follows four examples of such malicious behaviors and show how they can be described by LTL formulas:

**Registry Key Injecting:** In order to get started at boot time, many mal-  
800 wares add themselves into the registry key listing. This behavior is typically implemented by first calling the API function `GetModuleFileNameA` to retrieve

---

<sup>1</sup><https://lipn.univ-paris13.fr/~touili/smodic/>



Size	LTL	SM-PDS	PDS	Result	Total	Symbolic PDS	$Result_1$	$Total_1$
$S_1 : 5, S_2 : 2$	$ \delta :15$	<b>0.07s</b>	0.09s	0.01s	0.10s	0.08s	0.00s	0.08s
$S_1 : 5, S_2 : 3$	$ \delta :8$	<b>0.06s</b>	0.08s	0.01s	0.09s	0.09s	0.00s	0.09s
$S_1 : 11, S_2 : 4$	$ \delta :8$	<b>0.16s</b>	0.13s	0.05s	0.18s	0.10s	0.00s	0.10s
$S_1 : 5, S_2 : 3$	$ \delta :10$	<b>0.06s</b>	0.15s	0.01s	0.16s	0.09s	0.00s	0.09s
$S_1 : 110, S_2 : 4$	$ \delta :8$	<b>0.34s</b>	186.10s	0.79s	186.99s	0.35s	0.00s	0.35s
$S_1 : 255, S_2 : 8$	$ \delta :8$	<b>0.39s</b>	281.02s	0.94s	281.96s	4.82s	0.05s	4.87s
$S_1 : 255, S_2 : 8$	$ \delta :10$	<b>0.42s</b>	281.02s	0.97s	281.99s	4.82s	0.06s	4.88s
$S_1 : 110, S_2 : 4$	$ \delta :15$	<b>0.28s</b>	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
$S_1 : 255, S_2 : 8$	$ \delta :15$	<b>0.46s</b>	281.02s	1.92s	282.94s	4.82s	0.08s	4.90s
$S_1 : 110, S_2 : 4$	$ \delta :20$	<b>0.37s</b>	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
$S_1 : 255, S_2 : 8$	$ \delta :20$	<b>0.55s</b>	281.02s	1.97s	282.99s	4.82s	0.17s	4.99s
$S_1 : 255, S_2 : 8$	$ \delta :25$	<b>0.59s</b>	281.02s	1.23s	282.99s	4.82s	0.24s	5.36s
$S_1 : 2059, S_2 : 7$	$ \delta :8$	<b>0.86s</b>	19525.01s	20.71s	19545.72s	20.70s	error	-
$S_1 : 2059, S_2 : 9$	$ \delta :8$	<b>1.49s</b>	19784.7s	79.12s	19863.32	128.12s	error	-
$S_1 : 2059, S_2 : 11$	$ \delta :8$	<b>3.73s</b>	30011.67s	168.15s	30179.82s	261.07s	error	-
$S_1 : 2059, S_2 : 11$	$ \delta :28$	<b>6.88s</b>	30011.67s	169.55s	30180.22s	261.07s	error	-
$S_1 : 3050, S_2 : 10$	$ \delta :8$	<b>5.21s</b>	39101.57s	killed	-	438.27s	error	-
$S_1 : 3090, S_2 : 10$	$ \delta :8$	<b>5.86s</b>	40083.07s	killed	-	438.69s	error	-
$S_1 : 3050, S_2 : 10$	$ \delta :20$	<b>7.24s</b>	39101.57s	killed	-	438.27s	error	-
$S_1 : 3090, S_2 : 10$	$ \delta :30$	<b>8.38s</b>	40083.07s	killed	-	438.69s	error	-
$S_1 : 3090, S_2 : 10$	$ \delta :25$	<b>8.89s</b>	40083.07s	killed	-	438.69s	error	-
$S_1 : 4050, S_2 : 10$	$ \delta :8$	<b>9.21s</b>	81408.91s	killed	-	699.19s	error	-
$S_1 : 4050, S_2 : 10$	$ \delta :28$	<b>11.64s</b>	81408.91s	killed	-	699.19s	error	-
$S_1 : 4058, S_2 : 11$	$ \delta :8$	<b>9.83s</b>	93843.37s	killed	-	802.07s	error	-
$S_1 : 4058, S_2 : 11$	$ \delta :25$	<b>13.59s</b>	93843.37s	killed	-	802.07s	error	-
$S_1 : 5050, S_2 : 11$	$ \delta :8$	<b>10.34s</b>	173943.37s	killed	-	921.16s	error	-
$S_1 : 5090, S_2 : 11$	$ \delta :8$	<b>10.52s</b>	179993.54s	killed	-	929.32s	error	-
$S_1 : 5090, S_2 : 11$	$ \delta :10$	<b>12.89s</b>	179993.54s	killed	-	929.32s	error	-
$S_1 : 6090, S_2 : 11$	$ \delta :8$	<b>13.49s</b>	190293.64s	killed	-	1002.73s	error	-
$S_1 : 6090, S_2 : 11$	$ \delta :10$	<b>15.81s</b>	190293.64s	killed	-	1002.73s	error	-
$S_1 : 6090, S_2 : 11$	$ \delta :40$	<b>32.39s</b>	190293.64s	killed	-	1002.73s	error	-
$S_1 : 7090, S_2 : 11$	$ \delta :25$	<b>39.86s</b>	198932.32s	killed	-	1092.28s	error	-
$S_1 : 7090, S_2 : 11$	$ \delta :30$	<b>43.24s</b>	198932.32s	killed	-	1092.28s	error	-
$S_1 : 9090, S_2 : 11$	$ \delta :8$	<b>29.98s</b>	199987.98s	killed	-	1128.19s	error	-
$S_1 : 9090, S_2 : 11$	$ \delta :20$	<b>45.29s</b>	199987.98s	killed	-	1128.19s	error	-
$S_1 : 10050, S_2 : 12$	$ \delta :8$	<b>48.53s</b>	2134587.14s	killed	-	1469.28s	error	-
$S_1 : 10050, S_2 : 12$	$ \delta :25$	<b>59.69s</b>	2134587.14s	killed	-	1469.28s	error	-
$S_1 : 10050, S_2 : 12$	$ \delta :30$	<b>61.42s</b>	2134587.14s	killed	-	1469.28s	error	-
$S_1 : 10150, S_2 : 12$	$ \delta :35$	<b>64.17s</b>	2134633.28s	killed	-	1469.28s	error	-
$S_1 : 10150, S_2 : 14$	$ \delta :8$	<b>58.34s</b>	2181975.64s	killed	-	2849.96s	error	-
$S_1 : 10150, S_2 : 14$	$ \delta :40$	<b>82.72s</b>	2181975.64s	killed	-	2849.96s	error	-
$S_1 : 10150, S_2 : 12$	$ \delta :40$	<b>76.61s</b>	2134633.28s	killed	-	1469.28s	error	-
$S_1 : 10150, S_2 : 16$	$ \delta :45$	<b>89.83s</b>	2211008.82s	killed	-	3665.59s	error	-
$S_1 : 10150, S_2 : 12$	$ \delta :60$	<b>97.56s</b>	2134633.28s	killed	-	1469.28s	error	-
$S_1 : 10150, S_2 : 12$	$ \delta :65$	<b>105.89s</b>	2134633.28s	killed	-	1469.28s	error	-
$S_1 : 10150, S_2 : 16$	$ \delta :65$	<b>134.45s</b>	2211008.82s	killed	-	3665.59s	error	-
$S_1 : 10180, S_2 : 16$	$ \delta :65$	<b>175.29s</b>	2134643.52s	killed	-	3689.83s	error	-
$S_1 : 10180, S_2 : 16$	$ \delta :78$	<b>214.36s</b>	2134643.52s	killed	-	3689.83s	error	-

Table 1: Our approach vs standard LTL for PDSs

the path of the malware's executable file. Then, the API function RegSetValueExA is called to add the file path into the registry key listing. This malicious behavior can be described in LTL as follows:

$$\phi_{rk} = \mathbf{F}(call\ GetModuleFileNameA \wedge \mathbf{F}(call\ RegSetValueExA))$$

This formula expresses that if a call to the API function GetModuleFileNameA is followed by a call to the API function RegSetValueExA, then probably a malware is trying to add itself into the registry key listing.

**Data-Stealing:** Stealing data from the host is a popular malicious behavior that intend to steal any valuable information including passwords, software codes, bank information, etc. To do this, the malware needs to scan the disk to find the interesting file that he wants to steal. After finding the file, the malware needs to locate it. To this aim, the malware first calls the API function GetModuleHandleA to get a base address to search for a location of the file. Then the malware starts looking for the interesting file by calling the API function FindFirstFileA. Then the API functions CreateFileMappingA and MapViewOfFile are called to access the file. Finally, the specific file can be copied by calling the API function CopyFileA. Thus, this data-stealing malicious behavior can be described by the following LTL formula as follows:

$$\phi_{ds} = \mathbf{F}(call\ GetModuleHandleA \wedge \mathbf{F}(call\ FindFirstFileA \wedge \mathbf{F}(call\ CreateFileMappingA \wedge \mathbf{F}(call\ MapViewOfFile \wedge \mathbf{F}(call\ CopyFileA))))))$$

**Spy-Worm:** A spy worm is a malware that can record data and send it using the Socket API functions. For example, Keylogger is a spy worm that can record the keyboard states by calling the API functions GetAsyncKeyState and GetKeyState and send that to the specific server by calling the socket function sendto. Another spy worm can also spy on the I/O device rather than the keyboard. For this, it can use the API function GetRawInputData to obtain input from the specified device, and then send this input by calling the socket functions send or sendto. Thus, this malicious behavior can be described by the following LTL formula:

$$\phi_{sw} = \mathbf{F}((call\ GetAsyncKeyState \vee call\ GetRawInputData) \wedge \mathbf{F}(call\ sendto \vee call\ send))$$

**Appending virus:** An appending virus is a virus that inserts a copy of its code at the end of the target file. To achieve this, since the real OFFSET of the virus' variables depends on the size of the infected file, the virus has to first compute its real absolute address in the memory. To perform this, the virus has to call the sequence of instructions:  $l_1: call\ f; l_2: \dots; f: pop\ eax;$ . The instruction  $call\ f$  will push the return address  $l_2$  onto the stack. Then, the pop instruction in  $f$  will put the value of this address into the register eax. Thus, the virus can get its real absolute address from the register eax. This malicious behavior can be described by the following LTL formula:

$$\phi_{av} = \bigvee \mathbf{F}(call \wedge \mathbf{X}(top-of-stack = a) \wedge \mathbf{G}\neg(ret \wedge (top-of-stack = a)))$$

where the  $\bigvee$  is taken over all possible return addresses  $a$ , and  $top-of-stack=a$  is a predicate that indicates that the top of the stack is  $a$ . The subformula  $call \wedge \mathbf{X}(top-of-stack = a)$  means that there exists a procedure call having  $a$  as return address. Indeed, when a procedure call is made, the program pushes its corresponding return address  $a$  to the stack. Thus, at the next step,  $a$  will be on the top of the stack. Therefore, the formula above expresses that there exists a

procedure call having  $a$  as return address, such that there is no *ret* instruction  
850 which will return to  $a$ .

Note that this formula uses predicates that indicate that the top of the stack  
is  $a$ . Our techniques work for this case as well: it suffices to encode the top of  
the stack in the control points of the SM-PDS. Our implementation works for  
this case as well and can handle appending viruses.

### 855 5.2.2. Applying our tool for malware detection.

We applied our tool to detect several malwares. We use the unpack tool  
unpacker [29] to handle packers like UPX, and we use Jakstab [26] as disassem-  
bler. We consider 160 malwares from the malware library VirusShare [30], 184  
malwares from the malware library MalShare [31], 288 email-worms from VX  
860 heaven [32] and 260 new malwares generated by NGVCK, one of the best mal-  
ware generators. We also choose 200 benign samples from Windows programs.  
We consider self-modifying versions of these programs<sup>2</sup>. In these versions, the  
malicious behaviors are unreachable if the semantics of the self-modifying in-  
structions are not taken into account, i.e., if the self-modifying instructions are  
865 considered as “standard” instructions that do not modify the code, then the  
malicious behaviors cannot be reached. To check this, we model such programs  
in two ways:

1. First, we take into account the self-modifying instructions and model these  
programs using SM-PDSs as described in Section 2.3. Then, we check  
870 whether these SM-PDSs satisfy at least one of the malicious LTL formulas  
presented above. If yes, the program is declared as malicious, if not, it is  
declared as benign. Our tool was able to detect all the 892 self-modifying  
malwares as malicious, and to determine that benign programs are benign.  
We report in Table 3 some partial results of our experiments. **Column**  
875 *Size* is the number of control locations, **Column** *Result* gives the result of  
our algorithm: **Yes** means malicious and **No** means benign; and **Column**  
*cost* gives the cost to apply our LTL model-checker to check one of the  
LTL properties (**Column** *Formula*) described above. For every program,  
we consider all the formulas mentioned above. A program is declared  
880 malicious if it satisfies at least one of the formulas.
2. Second, we abstract away the self-modifying instructions and proceed as  
if these instructions were not self-modifying. In this case, we translate the  
binary codes to standard pushdown systems as described in [9]. By using  
PDSs as models, none of the malwares that we consider was detected as  
885 malicious, whereas, as reported in Table 3, using self-modifying PDSs as  
models, and applying our LTL model-checking algorithm allowed to detect  
all the 892 malwares that we considered.

**Remark.** Note that checking the formulas  $\phi_{rk}$ ,  $\phi_{ds}$ , and  $\phi_{sw}$  could be done  
using multiple *pre\** queries on SM-PDSs using the *pre\** algorithm of [1]. How-  
890 ever, this would be less efficient than performing our direct LTL model-checking  
algorithm, as shown in Table 2, where **Column** *Size* gives the number of control  
locations, **Column** *LTL* gives the time of applying our LTL model-checking al-  
gorithm; and **Column** *Multiple pre\** gives the cost of applying multiple *pre\** on

---

<sup>2</sup>Self-modifying instructions are embedded into these programs.

Example	Size	LTL	Multiple <i>pre</i> *	Example	Size	LTL	Multiple <i>pre</i> *
Tanatos.b	12315	16.261s	46.635s	Netsky.c	45	0.002s	0.092s
Win32.Happy	23	0.042s	0.075s	MyDoom-N	16980	30.231s	98.418s
Kelino.g	470	0.672s	3.446s	Netsky.b	45	0.057s	0.183s
Netsky.a	45	0.047s	0.085s	Mydoom.c	155	0.014s	0.206s
klez.c	30	0.039s	0.088s	Mydoom.v	5965	3.971s	83.988s
Netsky.d	45	0.083s	0.123s	Ardukr.d	1913	0.482s	3.212s
klez.f	27	0.054s	4.518s	Magistr.a.poly	36989	49.863s	159.195s
klez.e	27	0.094s	0.482s	Magistr.b	4670	3.987s	53.235s
Adon.1703	37	0.358s	0.884s	Adon.1559	37	0.255s	4.088s
Spam.Tedroo.AB	487	0.924s	4.894s	Alaul.c	355	0.109s	5.757s
Akez	273	0.136s	1.863s	Alcaul.d	845	0.165s	0.392s
Haharin.A	210	1.462s	4.318s	fsAutoB.F026	245	1.698s	4.503s
Haharin.dr	235	1.558s	4.312s	LdPinch.Win32.5558	2015	6.907s	8.981s
LdPinch.bx.dll	2010	6.965s	8.128s	LdPinch.fmeye	1845	6.194s	9.232s
LdPinch-15	580	1.008s	3.957s	LdPinch.e	578	1.185s	3.392s
Win32/Toga.lrfn	590	2.023s	3.978s	klez-N	6281	3.252s	78.419s
Mydoom.y	26902	12.462s	102.559s	Mydoom.j	22355	11.262s	111.617s
Plage.b	395	0.291s	3.138s	Urbe.a	123	0.376s	2.981s
Mydoom-EG	230	0.242s	6.172s	Email.W32!c	220	0.249s	5.946s
W32.Mydoom.L	235	0.288s	6.452s	Mydoom.DN.worm	220	0.299s	8.928s
Mydoom.5	228	0.307s	8.163s	Mydoom.cjdz	225	0.392s	9.968s
Mydoom.R	230	0.322s	9.086s	Win32.Mydoom	235	0.296s	7.985s
Mydoom.o@MM!zip	235	0.403s	10.323s	Win32.Mydoom.288	248	0.410s	2.983s
Sramota.avf	240	0.383s	2.691s	Mydoom	238	0.278	2.749s
Win32.Runonce	51678	92.692s	248.146s	Win32.Chur.A	51895	98.161s	298.047s
Win32.CNHacker	51095	94.952s	245.452s	Netsky.ah@MM	4480	6.991s	16.018s
Win32.Skybag	4180	6.891s	13.739s	Skybag.A	4310	6.205s	15.452s
LdPinch.by	970	4.092s	11.327s	Generic.20269	433	2.402s	9.614s
LdPinch.arr	1250	1.848s	9.986s	LdPnch-Fam	195	1.440s	4.097s
Troj.LdPinch.er	205	2.529s	6.154s	LdPinch.Gen.3	210	1.482s	4.973s

Table 2: Multiple *pre*\* v.s. our direct LTL model-checking algorithm

Example	Size	Formula	Result	cost	Example	Size	Formula	Result	cost	Example	Size	Formula	Result	cost
klez-N	6281	$\phi_{ds}$	Yes	3.252s	klez.c	30	$\phi_{ds}$	Yes	0.039s	klez.f	27	$\phi_{ds}$	Yes	0.054s
klez.d	31	$\phi_{ds}$	Yes	0.085s	Alcauld	845	$\phi_{av}$	Yes	0.165s	Alaul.c	355	$\phi_{av}$	Yes	0.109s
Akez	273	$\phi_{av}$	Yes	0.136s	Akez.Win32.1	455	$\phi_{av}$	Yes	4.008s	MyDoom.y	26902	$\phi_{ds}$	Yes	12.462s
MyDoom.j	22355	$\phi_{ds}$	Yes	11.262s	Mydoom.c	155	$\phi_{ds}$	Yes	0.014s	MyDoom-N	16980	$\phi_{ds}$	Yes	30.231s
MyDoom.v	5965	$\phi_{ds}$	Yes	3.971s	Mydoom.M	5965	$\phi_{ds}$	Yes	5.633s	Netsky.a	45	$\phi_{ds}$	Yes	0.047s
Netsky.d	45	$\phi_{ds}$	Yes	0.083s	Worm.Skybag-1	4820	$\phi_{rk}$	Yes	7.119s	Win32.Agent.R	4490	$\phi_{rk}$	Yes	7.898s
LdPinch.BX	2010	$\phi_{rk}$	Yes	6.965s	LdPinch.fmye	1845	$\phi_{rk}$	Yes	6.194s	LdPinch.Win32	2015	$\phi_{rk}$	Yes	6.907s
Lydra.a	3450	$\phi_{rk}$	Yes	8.289s	Trojan.StartPage	2985	$\phi_{rk}$	Yes	5.982s	PSW.Troj.au	2985	$\phi_{rk}$	Yes	6.198s
LdPinch-21	3180	$\phi_{rk}$	Yes	6.917s	LdPinch-R	3025	$\phi_{rk}$	Yes	7.005s	LdPinch.Gen	2990	$\phi_{rk}$	Yes	6.992s
Repah.b	221	$\phi_{rk}$	Yes	2.428s	Gibe.b	5358	$\phi_{rk}$	Yes	4.229s	Magistr.b	4670	$\phi_{rk}$	Yes	3.699s
Win32.Anar.a	215	$\phi_{ds}$	Yes	1.631s	Anar.24576	240	$\phi_{ds}$	Yes	2.738s	Anar.S	155	$\phi_{ds}$	Yes	2.093s
Kelino.l	495	$\phi_{rk}$	Yes	0.326s	Kipis.t	20378	$\phi_{rk}$	Yes	25.345s	W32.HfsAutoB.	3398	$\phi_{rk}$	Yes	5.092s
Worm.Anarxy	210	$\phi_{ds}$	Yes	1.913s	Plage.b	395	$\phi_{rk}$	Yes	0.291s	Urbe.a	123	$\phi_{rk}$	Yes	0.376s
calculation.exe	9952	$\phi_{ds}$	No	31.176s	cisvc.exe	4105	$\phi_{ds}$	No	9.114s	simple.exe	52	$\phi_{ds}$	No	0.053s
calculation.exe	9952	$\phi_{rk}$	No	14.932s	cisvc.exe	4105	$\phi_{rk}$	No	3.454s	simple.exe	52	$\phi_{rk}$	No	0.001s
calculation.exe	9952	$\phi_{av}$	No	40.118s	cisvc.exe	4105	$\phi_{av}$	No	5.092s	simple.exe	52	$\phi_{av}$	No	0.197s
calculation.exe	9952	$\phi_{sw}$	No	19.539s	cisvc.exe	4105	$\phi_{sw}$	No	2.864s	simple.exe	52	$\phi_{sw}$	No	0.003s
shutdown.exe	2529	$\phi_{ds}$	No	13.228s	loop.exe	529	$\phi_{ds}$	No	4.373s	cmd.exe	1324	$\phi_{ds}$	No	13.466s
shutdown.exe	2529	$\phi_{rk}$	No	9.152s	loop.exe	529	$\phi_{rk}$	No	8.029s	cmd.exe	1324	$\phi_{rk}$	No	6.233s
shutdown.exe	2529	$\phi_{av}$	No	19.031s	loop.exe	529	$\phi_{av}$	No	13.478s	cmd.exe	1324	$\phi_{av}$	No	9.620s
shutdown.exe	2529	$\phi_{sw}$	No	0.397s	loop.exe	529	$\phi_{sw}$	No	9.249s	cmd.exe	1324	$\phi_{sw}$	No	9.268s
notepad.exe	10529	$\phi_{rk}$	No	24.583s	java.exe	800	$\phi_{rk}$	No	15.852s	eclipse.exe	21324	$\phi_{rk}$	No	42.373s
notepad.exe	10529	$\phi_{av}$	No	89.131s	java.exe	800	$\phi_{av}$	No	18.079s	eclipse.exe	21324	$\phi_{av}$	No	63.447s
notepad.exe	10529	$\phi_{sw}$	No	22.830s	java.exe	800	$\phi_{sw}$	No	13.472s	eclipse.exe	21324	$\phi_{sw}$	No	51.693s
notepad.exe	10529	$\phi_{ds}$	No	36.119s	java.exe	800	$\phi_{ds}$	No	22.357s	java.exe	21324	$\phi_{ds}$	No	69.683s
sort.exe	8529	$\phi_{rk}$	No	29.789s	bibDesk.exe	32800	$\phi_{rk}$	No	50.279s	interface.exe	1005	$\phi_{rk}$	No	8.462s
sort.exe	8529	$\phi_{sw}$	No	34.427s	bibDesk.exe	32800	$\phi_{sw}$	No	197.628s	interface.exe	1005	$\phi_{sw}$	No	11.309s
sort.exe	8529	$\phi_{av}$	No	69.140s	bibDesk.exe	32800	$\phi_{av}$	No	408.925s	interface.exe	1005	$\phi_{av}$	No	32.193s
ipv4.exe	968	$\phi_{rk}$	No	4.186s	TextWrangler.exe	14675	$\phi_{rk}$	No	45.221s	sogou.exe	45219	$\phi_{rk}$	No	55.259s

Example	Size	Formula	Result	cost	Example	Size	Formula	Result	cost	Example	Size	Formula	Result	cost
SdBot.zk	3430	$\phi_{rk}$	Yes	23.242s	Virus.Gen	661	$\phi_{av}$	Yes	9.437s	AutoRun	240	$\phi_{rk}$	Yes	4.181s
Spam.AB	487	$\phi_{rk}$	Yes	0.924s	Haharin.A	210	$\phi_{rk}$	Yes	1.462s	Alaul.c	355	$\phi_{ds}$	Yes	0.109s
Virus.klk	5235	$\phi_{rk}$	Yes	15.863s	Virus.Agent	5340	$\phi_{rk}$	Yes	15.968s	Hoax.Gen	5455	$\phi_{rk}$	Yes	13.569s
eHeur.Virus02	420	$\phi_{ds}$	Yes	4.985s	Akez.11255	440	$\phi_{rk}$	Yes	3.985s	Akez.5	490	$\phi_{rk}$	Yes	3.958s
Weird.c	430	$\phi_{rk}$	Yes	3.929s	PEAKEZ.A	450	$\phi_{rk}$	Yes	2.998s	Weird.d	473	$\phi_{ds}$	Yes	3.302s
Win32.Runonce	51678	$\phi_{rk}$	Yes	92.692s	Chur.A	51895	$\phi_{rk}$	Yes	98.161s	WCNHacker.C	51095	$\phi_{rk}$	Yes	94.952s
Agent.xpro	533	$\phi_{rk}$	Yes	0.352s	Vilsel.lhb	15036	$\phi_{rk}$	Yes	4.972s	Generic.20269	433	$\phi_{av}$	Yes	3.489s
NewAptIgeneric	4815	$\phi_{sw}$	Yes	9.002s	NewApt.A@mm	4485	$\phi_{sw}$	Yes	8.159s	Newapt.1	4155	$\phi_{sw}$	Yes	7.885s
NGVCK1	329	$\phi_{rk}$	Yes	0.933s	NGVCK2	455	$\phi_{sw}$	Yes	1.109s	NGVCK3	2300	$\phi_{rk}$	Yes	1.388s
NGVCK4	550	$\phi_{ds}$	Yes	1.149s	NGVCK5	1555	$\phi_{rk}$	Yes	1.825s	NGVCK6	1698	$\phi_{rk}$	Yes	1.689s
NGVCK7	6902	$\phi_{av}$	Yes	14.524s	NGVCK8	2355	$\phi_{rk}$	Yes	4.254s	NGVCK9	281	$\phi_{sw}$	Yes	13.301s
NGVCK10	2980	$\phi_{rk}$	Yes	9.262s	NGVCK11	5965	$\phi_{ds}$	Yes	11.456s	NGVCK12	4529	$\phi_{ds}$	Yes	10.094s
NGVCK13	2210	$\phi_{rk}$	Yes	8.902s	NGVCK14	5358	$\phi_{ds}$	Yes	10.294s	NGVCK15	970	$\phi_{ds}$	Yes	1.912s
NGVCK16	658	$\phi_{rk}$	Yes	0.935s	NGVCK17	913	$\phi_{rk}$	Yes	1.392s	NGVCK18	90	$\phi_{rk}$	Yes	0.094s
NGVCK19	1295	$\phi_{ds}$	Yes	6.958s	NGVCK20	4378	$\phi_{ds}$	Yes	15.449s	NGVCK21	31	$\phi_{rk}$	Yes	0.097s
NGVCK22	370	$\phi_{ds}$	Yes	0.898s	NGVCK23	3955	$\phi_{ds}$	Yes	9.498s	NGVCK24	6924	$\phi_{ds}$	Yes	11.983s
NGVCK25	8127	$\phi_{ds}$	Yes	15.018s	NGVCK26	4970	$\phi_{ds}$	Yes	9.982s	NGVCK27	7989	$\phi_{ds}$	Yes	13.197s
NGVCK28	227	$\phi_{rk}$	Yes	0.098s	NGVCK29	960	$\phi_{rk}$	Yes	0.692s	NGVCK30	89	$\phi_{rk}$	Yes	0.088s
NGVCK31	550	$\phi_{rk}$	Yes	0.875s	NGVCK32	60	$\phi_{rk}$	Yes	0.059s	NGVCK33	65	$\phi_{rk}$	Yes	0.069s
NGVCK34	5990	$\phi_{ds}$	Yes	9.848s	NGVCK35	4590	$\phi_{ds}$	Yes	10.178s	NGVCK36	825	$\phi_{ds}$	Yes	2.934s
NGVCK37	80	$\phi_{rk}$	Yes	0.998s	NGVCK38	150	$\phi_{rk}$	Yes	1.093s	NGVCK39	395	$\phi_{rk}$	Yes	1.048s
mfc.dll	110	$\phi_{rk}$	No	2.014s	Uedit32	98	$\phi_{rk}$	No	0.926s	wechat.exe	12252	$\phi_{rk}$	No	45.147s
mfc.dll	110	$\phi_{sw}$	No	24.571s	Uedit32	98	$\phi_{sw}$	No	2.572s	wechat.exe	12252	$\phi_{sw}$	No	68.327s
mfc.dll	110	$\phi_{av}$	No	19.132s	Uedit32	98	$\phi_{av}$	No	36.176s	wechat.exe	12252	$\phi_{av}$	No	57.129s
mfc.dll	110	$\phi_{ds}$	No	7.746s	Uedit32	98	$\phi_{ds}$	No	6.529s	wechat.exe	12252	$\phi_{ds}$	No	54.373s
game.exe	34325	$\phi_{rk}$	No	82.424s	cycle.exe	9014	$\phi_{rk}$	No	42.555s	calender.exe	892	$\phi_{rk}$	No	35.039s
game.exe	34325	$\phi_{ds}$	No	60.119s	cycle.exe	9014	$\phi_{ds}$	No	73.306s	calender.exe	892	$\phi_{ds}$	No	42.148s
game.exe	34325	$\phi_{av}$	No	126.037s	cycle.exe	9014	$\phi_{av}$	No	110.191s	calender.exe	892	$\phi_{av}$	No	65.983s
game.exe	34325	$\phi_{sw}$	No	61.254s	cycle.exe	9014	$\phi_{sw}$	No	51.026s	calender.exe	892	$\phi_{sw}$	No	27.105s

Table 3: Experimental results

our tool	McAfee	Norman	BitDefender	Kinsoft	Avira	eScan	Kaspersky	Qihoo360	Baidu	Avast	Symantec
100%	24.8%	19.5%	31.2%	9.7%	34.1%	21.9%	53.1%	51.7%	1.4%	68.3%	82.4%

Table 4: Detection rate: Our tool vs. well known antiviruses

SM-PDSs to check the properties  $\phi_{rk}$ ,  $\phi_{ds}$ , and  $\phi_{sw}$ . It can be seen that applying our *direct* LTL model checking algorithm is more efficient. Furthermore, the appending virus formula  $\phi_{av}$  cannot be solved using multiple *pre*\* queries. Our direct LTL model-checking algorithm is needed in this case. Note that some of the malwares we considered in our experiments are appending viruses. Thus, our algorithm and our implementation are crucial to be able to detect these malwares.

### 5.2.3. Comparison with well-known antiviruses.

We compare our tool against well-known and widely used antiviruses. Since known antiviruses update their signature database as soon as a new malware is known, in order to have a fair comparison with these antiviruses, we need to consider new malwares. We use the sophisticated malware generator NGVCK available at VX Heavens [32] to generate 205 malwares. We obfuscate these malwares with self-modifying code, and we fed them to our tool and to well known antiviruses such as BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec. Our tool was able to detect all these programs as malicious, whereas none of the well-known antiviruses was able to detect all these malwares. Table 4 reports the detection rates of our tool and the well-known anti-viruses.

## 6. Conclusion and discussion

In this paper, we propose a **direct** LTL model checking algorithm for SM-PDSs. Our algorithm is based on reducing the LTL model checking problem to the emptiness problem of Self Modifying Büchi Pushdown Systems (SM-BPDSs). Intuitively, we obtain this SM-BPDS by taking the product of the SM-PDS with a Büchi automaton accepting an LTL formula  $\varphi$ . Then, we solve the emptiness problem of an SM-BPDS by computing its repeating heads. This computation is based on computing labelled *pre*\* configurations by applying a saturation procedure on labelled finite automata.

We implemented our techniques in a tool for self-modifying code analysis. We successfully used our tool to model-check more than 900 self-modifying binary codes. In particular, we applied our tool for malware detection, since malwares usually use self-modifying instructions, and since malicious behaviors can be described by LTL formulas. In our experiments, our tool was able to detect 895 malwares and to prove that 200 benign programs were benign. It was also able to detect several malwares that well-known antiviruses such as Bit-Defender, Kinsoft, Avira, eScan, Kaspersky, Avast, and Symantec failed to detect.

Malware detection is nowadays a big challenge. This work brings just a stone to the building, and helps dealing with self-modifying code. However, a lot of work remains to be done in order to have a robust tool for malware detection. Indeed, as mentioned in Section 2.3.1, this work assumes that if instruction  $i_1$

935 is replaced by  $i_2$ , then  $i_1$  and  $i_2$  must have the same number of operands. This  
of course does not hold for all malicious programs. Moreover, to disassemble  
programs, we use Jakstab [26]. This tool has a lot of limitations and offers  
sometimes rough translations, as discussed in [26]. Another limitation of our  
940 tool is that currently it considers only the four malicious behaviors described  
in Section 5.2.1. Several other malicious behaviors can be found in malwares.  
Thus, we need to study these behaviors and specify them as LTL formulas. To  
this aim, we plan to apply machine learning techniques in order to extract the  
maximum number of malicious behaviors from malwares.

## References

- 945 [1] T.Touili, X.Ye, Reachability analysis of self modifying code, in: 2017 22nd  
International Conference on Engineering of Complex Computer Systems  
(ICECCS), IEEE, 2017, pp. 120–127.
- [2] A. Bouajjani, J. Esparza, O. Maler, Reachability Analysis of Pushdown  
950 Automata: Application to Model Checking, in: International Conference  
on Concurrency Theory (CONCUR), Springer, 1997, pp. 135–150.
- [3] J.Esparza, D.Hansel, P.Rossmanith, S.Schwoon, Efficient algorithms for  
model checking pushdown systems, in: International Conference on Com-  
puter Aided Verification (CAV), Springer, 2000, pp. 232–247.
- 955 [4] T.Touili, X.Ye, Ltl model checking of self modifying code, in: 2019 24th  
International Conference on Engineering of Complex Computer Systems  
(ICECCS), IEEE, 2019, pp. 1–10.
- [5] J.Bergeron, M.Debbabi, et al., Static detection of malicious code in exe-  
cutable programs, *Int. J. of Req. Eng* 2001 (184-189) (2001) 79.
- [6] G.Balakrishnan, T. Reps, N.Kidd, A.Lal, J.Lim, et al., Model checking x86  
960 executables with codesurfer/x86 and WPDS++, in: International Confer-  
ence on Computer Aided Verification (CAV), Springer, 2005, pp. 158–163.
- [7] P.Singh, A.Lakhotia, Static verification of worm and virus behavior in bi-  
nary executables using model checking, in: IEEE Systems, Man and Cy-  
bernetics Society Information Assurance Workshop, 2003., IEEE, 2003, pp.  
965 298–300.
- [8] J.Kinder, S.Katzenbeisser, C.Schallhart, H.Veith, Detecting malicious code  
by model checking, in: International Conference on Detection of Intrusions  
and Malware, and Vulnerability Assessment, Springer, 2005, pp. 174–187.
- 970 [9] F.Song, T.Touili, Efficient malware detection using model-checking, in: In-  
ternational Symposium on Formal Methods, Springer, 2012, pp. 418–433.
- [10] P. Beaucamps, I. Gnaedig, J. Marion, Behavior abstraction in malware  
analysis, in: International Conference on Runtime Verification, Springer,  
2010, pp. 168–182.
- 975 [11] F.Song, T.Touili, Ltl model-checking for malware detection, in: Interna-  
tional Conference on Tools and Algorithms for the Construction and Anal-  
ysis of Systems (TACAS), Springer, 2013, pp. 416–431.



- [12] H.Nguyen, T.Touili, CARET model checking for malware detection, in: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, 2017.
- 980 [13] K.Dam, T.Touili, Learning malware using generalized graph kernels, in: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018, pp. 1–6.
- [14] K.Dam, T.Touili, Precise extraction of malicious behaviors, in: 42nd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, 985 IEEE, 2018, pp. 229–234.
- [15] K.Dam, T.Touili, Malware detection based on graph classification, in: ICISSP, 2017.
- [16] H.Cai, Z.Shao, A.Vaynberg, Certified self-modifying code, ACM SIGPLAN Notices 42 (6).
- 990 [17] S.Debray, K.Coogan, G.Townsend, On the semantics of self-unpacking malware code, Tech. rep. University of Arizona, Computer Science.
- [18] G.Bonfante, J. Marion, D.Reynaud-Plantey, A computability perspective on self-modifying programs, in: International Conference on Software Engineering and Formal Methods (SEFM), IEEE, 2009, pp. 231–239.
- 995 [19] A.Bertrand, M.Matias, D.Koen, A model for self-modifying code, in: International Workshop on Information Hiding, Springer, 2006, pp. 232–248.
- [20] S.Blazy, V.Laporte, D.Pichardie, Verified abstract interpretation techniques for disassembling low-level self-modifying code, Journal of Automated Reasoning 56 (3) (2016) 283–308.
- 1000 [21] K.Roundy, B.Miller, Hybrid analysis and control of malware, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2010, pp. 317–338.
- [22] K.Coogan, S.Debray, T.Kaochar, G.Townsend, Automatic static unpacking of malware binaries, in: 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 167–176. 1005
- [23] K.Gyung, et al., Renovo: A hidden code extractor for packed executables, in: Proceedings of the 2007 ACM workshop on Recurring malcode, 2007, pp. 46–53.
- [24] P.Royal, M.Halpin, et al., Polyunpack: Automating the hidden-code extraction of unpack-executing malware, in: 22nd Annual Computer Security Applications Conference (ACSAC’06), IEEE, 2006, pp. 289–300. 1010
- [25] S.Schwoon, Model-checking pushdown systems, Ph.D. thesis, Technische Universität München, Universitätsbibliothek (2002).
- 1015 [26] H. J.Kinder, Jakstab: A static analysis platform for binaries, in: International Conference on Computer Aided Verification (CAV), Springer, 2008.

- [27] M.Vardi, P.Wolper, Reasoning about infinite computations, *Inf. Comput.* 115 (1).
- [28] P.Gastin, D.Oddoux, Fast ltl to büchi automata translation, in: *International Conference on Computer Aided Verification (CAV)*, Springer, 2001.
- 1020 [29] U. Tool, Automated unpacking: A behaviour based approach, <https://github.com/malwaremusings/unpacker>.
- [30] VirusShare, vxshare, <https://virusshare.com>.
- [31] S.Cutler, malshare, <https://malshare.com>.
- [32] V.Heaven, V.heavens, <http://vxer.org/lib/>.