A Bypass of Cohen's Impossibility Result:

Extended Version for SSN - 29 November 2004

Jan A. Bergstra^{1,2} and Alban Ponse¹

University of Amsterdam, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands www.science.uva.nl/research/prog/

Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands www.phil.uu.nl/en/

Abstract

Detecting illegal resource access in the setting of network communication or grid computing is similar to the problem of virus detection as put forward by Fred Cohen in 1984. We disucuss Cohen's impossibility result on virus detection, and introduce "risk assessment of security hazards", a notion that is decidable for a large class of program behaviors.

Keywords: Malcode, Program algebra, Polarized process algebra, Virus, Worm.

1 Introduction

Virus, Worm, Trojan horse, Malcode.... There is a vast amount of literature on these matters, and opinions seem to differ wildly. Many authors agree that malcode contains all others, and that both a virus and a worm can replicate.¹ Furthermore, a worm is more autonomous than a virus. Some authors claim that a virus can only replicate as a consequence of actions of users, and that sound education and awareness can protect users from acting with such effect. So, a virus uses a user for its replication; that user may or may not be a victim of the virus' harmful action at the same time. Unclear is if each of these users must be a human one or if background processes in a machine can also be "used" as users.

This paper is about virus detection and discusses two fundamental questions. First, we consider Cohen's result about the impossibility of a uniform tool (or algorithm) for detecting viruses in all programs [4]. This is done in the setting of the program algebra PGA [1]. Then, we define an associated notion of testing — *security hazard risk assessment* — with which the occurrence of security hazards is decidable for a large class of program behaviors. However, if divergence (the absence of halting) is considered also as a security hazard, decidability is lost.

The paper is organized as follows: in Section 2, we introduce some basics of program algebra. Then, in Section 3, we consider Cohen's impossibility result and some related issues. In Section 4 we introduce our notion of security risk assessment. In Section 5 we discuss a variant of the Halting Problem that applies to the present setting. The paper is ended with an Appendix.

2 Basics of Program Algebra

Program algebra (PGA, [1]) provides a very simple notation for sequential programs and a setting in which programs can be systematically analyzed and mapped onto behaviors. Program behaviors are

¹See Appendix A for some quotes about these topics.

modeled in BPPA, Basic Polarized Process Algebra. Finally, we consider in this section some other program notations based on program algebra.

2.1 The program Algebra PGA

In PGA we consider *basic* instructions a, b, ..., given by some collection B. Furthermore, for each $a \in B$ there is a *positive test* instruction +a and a *negative test* instruction -a. The control instructions are *termination*, notation!, and (relative) jump instructions #k ($k \in \mathbb{N}$). Program expressions in PGA, or shortly PGA-programs, have the following syntax:

- each PGA-instruction is a PGA-program,
- if X and Y are PGA-programs, so is their concatenation X; Y,
- if X is a PGA-program, so is its repetition X^{ω} .

The behavior associated with the execution of PGA-programs is explained below in Section 2.2. Instruction congruence of programs has a simple axiomatization, given in Table 1.

Table 1: Axioms for PGA's instruction sequence congruence

$$\begin{array}{ccccc} (X;Y);Z&=&X(Y;Z) & (\mathrm{PGA1}) & & X^\omega;Y&=&X^\omega & (\mathrm{PGA3}) \\ (X^n)^\omega&=&X^\omega & \text{for } n>0 & (\mathrm{PGA2}) & & (X;Y)^\omega&=&X(Y;X)^\omega & (\mathrm{PGA4}) \end{array}$$

The axioms PGA1-4 imply *Unfolding*, i.e. the law $X^{\omega} = X; X^{\omega}$, and PGA2-4 may be replaced by Unfolding and the proof rule $Y = X; Y \Rightarrow Y = X^{\omega}$.

2.2 BPPA, Basic Polarized Process Algebra

Execution of PGA-programs is modeled in BPPA, Basic Polarized Process Algebra. Given B, now considered as a collection of *actions*, it is assumed that upon execution each action generates a Boolean reply (true or false). Now, behavior is specified in BPPA by means of the following constants and operations:

Termination. The constant $S \in \text{BPPA}$ represents (successful) termination.

Inaction. The constant $D \in BPPA$ represents the situation in which no subsequent behavior is possible. (Sometimes D is called *deadlock* or *divergence*.)

Post conditional composition. For each action $a \in B$ and behavioral expressions P and Q in BPPA, the post conditional composition $P \subseteq a \supseteq Q$ describes the behavior that first executes action a, and continues with P if true was generated, and Q otherwise.

Action prefix. For $a \in B$ and behavioral expression $P \in BPPA$, the action prefix $a \circ P$ describes the behavior that first executes a and then continues with P, irrespective of the Boolean reply. Action prefix is a special case of post conditional composition: $a \circ P = P \le a \ge P$.

2.3 Behavior Extraction: from PGA to BPPA

The behavior extraction operator |X| assigns a behavior to program X. Instruction sequence equivalent programs have of course the same behavior. Behavior extraction is defined by the thirteen equations in Table 2, where $a \in B$ and u is a PGA-instruction.

Table 2: Equations for behavior extraction on PGA

```
\begin{array}{llll} |!| &= S & & |!;X| &= S & & |\#k| &= D \\ |a| &= a \circ D & & |a;X| &= a \circ |X| & & |\#0;X| &= D \\ |+a| &= a \circ D & & |+a;X| &= |X| \unlhd a \trianglerighteq |\#2;X| & & |\#1;X| &= |X| \\ |-a| &= a \circ D & & |-a;X| &= |\#2;X| \unlhd a \trianglerighteq |X| & & |\#k+2;u| &= D \\ |\#k+2;u;X| &= |\#k+1;X| & & |\#k+2;u| &= |\#k+1;X| & & |\#k| &= |\#k+1;X| & & |\#k| &= |\#
```

Some examples: $|(\#0)^{\omega}| = |\#0; (\#0)^{\omega}| = D$ and, further taking action prefix to bind stronger than post conditional composition,

```
|-a;b;c| = |\#2;b;c| \le a \ge |b;c|
= |\#1;c| \le a \ge b \circ |c|
= |c| \le a \ge b \circ c \circ D
= c \circ D \lhd a \rhd b \circ c \circ D.
```

In some cases, these equations can be applied (from left to right) without ever generating any behavior, e.g.,

```
\begin{aligned} |(\#1)^{\omega}| &= |\#1; (\#1)^{\omega}| = |(\#1)^{\omega}| = \dots \\ |(\#2; a)^{\omega}| &= |\#2; a; (\#2; a)^{\omega}| = |\#1; (\#2; a)^{\omega}| = |(\#2; a)^{\omega}| = \dots \end{aligned}
```

In such cases, the extracted behavior is defined as D.

It is also possible that behavioral extraction yields an infinite recursion, e.g., $|a^{\omega}| = |a; a^{\omega}| = a \circ |a^{\omega}|$, and therefore,

```
\begin{array}{rcl} |a^{\omega}| &=& a \circ |a^{\omega}| \\ &=& a \circ a \circ |a^{\omega}| \\ &=& a \circ a \circ a \circ |a^{\omega}| \\ &\vdots \end{array}
```

In such cases the behavior of X is infinite, and can be represented by a finite number of behavioral equations, e.g., $|(a; +b; \#3; -b; \#4)^{\omega}| = P$ and

$$\begin{array}{ll} P &=& a \circ (P \unlhd b \trianglerighteq Q), \\ Q &=& P \unlhd b \trianglerighteq Q. \end{array}$$

Note. Observe that the following identity holds: $|X| = |X; (\#0)^{\omega}|$. This identity characterizes that for a finite program object (i.e., a finite sequence of instructions), a missing termination instruction yields inaction. Conversely, this identity makes six out of the thirteen equations in Table 2 derivable (namely, those for programs of length 1 and the equation |#k+2; u| = D).

2.4 The program notations PGLB and PGLC

The program notation PGLB is obtained from PGA by adding backwards jumps $\mbox{$\#$}k$ and leaving out the repetition operator. For example, $+a; \#0; \mbox{$\#$}2$ behaves as $(+a; \#0)^{\omega}$. This can be defined with help of a projection function pglb2pga that translates PGLB-programs in a context-dependent fashion to PGA-programs. For a PGLB-program X we write $|X|_{pglb} = |pglb2pga(X)|$ (see further [1]).

The language PGLC is the variant of PGLB in which termination is modeled implicitly: a program terminates after the last instruction has been executed and that instruction was no jump into the

program, or after a jump outside the program. The termination instruction! is not present in PGLC. For example,

```
 |+a; #2; \ | #2; +b|_{pglc} = |+a; #2; \ | #2; +b; !; !|_{pglb} 
 = |(+a; #2; #6; +b; !; !; #0; #0)^{\omega}| 
 = P
```

for $P = b \circ S \leq a \geq P$ (see [1] for precise definitions of $|X|_{pglc}$ and $|Y|_{pglb}$.)

3 Forecasting Security Hazards

In this section we introduce the setting in which we will analyze code security risks. We then recall Cohen's impossibility result on forecasting security hazards and draw some conclusions.

3.1 A Setting with Security Hazards

Let P be some behavior that uses communication with other devices — further called *reactors* — H_f , H_g and H_e :

$$\begin{array}{c|c} \hline P & \xrightarrow{e} & H_e & \text{(external focus)} \\ \\ f \downarrow & \searrow g & H_g & \text{(low risk focus, no security hazard)} \\ \hline \\ H_f & \text{(high risk focus, security risk)} \end{array}$$

Such communication will be modelled using "focus-method" notation: the reply of a basic instruction e.m will be determined by the reactor H_e . Likewise, instructions with focus f or g communicate with H_f and H_g , respectively.

Let furthermore skip be the identity on H_f . Now, execution is secure if no f.m is called until termination or first call of some e.m (to the external focus). A behavior can have low risk actions (secure execution expected) and high risk actions (insecure execution expected). For example,

```
\begin{array}{ccc} S & & --\text{ a low risk behavior,} \\ f.\textit{skip} \circ S & & --\text{ a high risk behavior,} \\ f.\textit{skip} \circ S \unlhd g.m \trianglerighteq g.\textit{skip} \circ S & --\text{ risk depends on } H_g. \end{array}
```

3.2 SHFT, a Security Hazard Forecasting Tool

In this section we discuss the impossibility of a tool (algorithm) that forecasts security hazards. Let SHFT be a Security Hazard Forecasting Tool with focus shft, thus a reactor that forecasts a security hazard. As assumed earlier, a security hazard is in this simple setting a call (action) f.m for some m. Furthermore, let shft.test be the test that uses SHFT in the following way: in

```
P \subseteq shft.test \supseteq Q,
```

the action shft.test returns true if P has a security hazard, and false if Q has no security hazard.

Theorem 1. A Security Hazard Forecasting Tool cannot exist.

Proof. Consider $S \subseteq shft.test \supseteq f.skip \circ S$. If the test action shft.test returns false, $f.skip \circ S$ will be performed, which is a security hazard; if true is returned, then S is performed and no security hazard arises.

The behavior in the proof above illustrates the impossibility of prediciting that a behavior (or a program) contains a virus, a general phenomenon that was described in Cohen's seminal 1984-paper [4] and that will be further referred to as *Cohen's impossibility result*. For the sake of completeness, we recall Cohen's line of reasoning. In the pseudo-code in Figure 1 (taken from [4]), D is a decision procedure that determines whether a program is (contains) a virus, ~D stands for its negation, and next labels the remainder of some (innocent) program.

```
program contradictory-virus:=
{1234567;
subroutine infect-executable:=
 {loop:file = get-random-executable-file;
 if first-line-of-file = 1234567 then goto loop;
 prepend virus to file;
subroutine do-damage:=
 {whatever damage is to be done}
subroutine trigger-pulled:=
 {return true if some condition holds}
main-program:=
 {if ~D(contradictory-virus) then
    {infect-executable;
    if trigger-pulled then do-damage;
    }
 goto next;
}
```

Figure 1: Cohen's program contradictory-virus

In PGLC, the program contradictory-virus can be represented by the following term CV:

```
CV = #8; Pre; #3; -shft.test(CV); \#8; Next
```

where Pre abbreviates the six instructions that model the security hazard:

and Next models the remainder of the program. Applying behavior extraction on this program yields

```
\begin{split} |\mathtt{CV}|_{pglc} &= \ |\mathtt{Next}|_{pglc} \unlhd \mathit{shft}.\mathtt{test}(\mathtt{CV}) \trianglerighteq |\mathtt{Pre}; \#3; -\mathit{shft}.\mathtt{test}(\mathtt{CV}); \backslash \#8; \mathtt{Next}|_{pglc} \\ &= \ |\mathtt{Next}|_{pglc} \unlhd \mathit{shft}.\mathtt{test}(\mathtt{CV}) \trianglerighteq |\mathtt{Pre}; \mathtt{Next}|_{pglc}. \end{split}
```

So, $S \subseteq shft.test \supseteq f.skip \circ S$ is indeed a faithful characterization of Cohen's impossibility result.

Even with the aid of universal computational power, the problem whether a behavior has a security hazard (issues an f.m call) is undecidable. In Section 5.2 we give a proof of this fact.

3.3 Modifying SHFT

Alternatively, we can modify our definitions concerning forecasting security hazards:

- 1. If all code is malcode, SHFT should always return true. This is correct. It follows that Cohen's impossibility result depends on the assumption that for some *P*, SHFT returns false.
- 2. We can restrict our class of behaviors: if security hazards occur only in the lefthand-sides of behavioral expressions, as in $f.skip \circ S \subseteq shft.test \supseteq S$, then a negative reply (false) is always correct (using the definition that malcode is code for which the extracted behavior contains a security hazard).

Cohen's impossibility result needs the notion of a secure run (no security hazards), as well as a secure program or behavior (a behavior that will have secure runs only). So, Cohen's impossibility result emerges if:

- secure runs exist,
- secure programs (polarized processes) exist,
- there is a full match between these two, and
- forecasting is possible.

Now there is a difficulty with forecasting: if *shft.test* returns false one hopes to proceed in such a way that the security hazard is avoided (why else do the test?). But that is not sound as was shown above. Thus we conclude: this type of forecasting security hazards is a problematic idea for security assessment.

Yet another perspective on these problems is to consider the option that test actions shft.test may also yield a third truth value M (Meaningless). At the same time we can avoid the problem that the true-reply of the test shft.test is about a different behavior (the left-argument) as the false-reply, making such a type of test (a little) more plausible. So, consider the following modification of shft.test: in

$$P \triangleleft shft.test2 \triangleright Q$$
,

the action shft.test2 returns true if P has a security hazard, false if both P and Q have no security hazard, and M in all other cases. 'the behavior associated with the reply M can be taken D, but other options are possible (see e.g. [3]). This seems a more consistent definition. However, in

$$S \subseteq shft.test2 \supseteq f.skip \circ S$$

the test action shft.test2 returns M, showing that forecasting security hazards in the style of Cohen remains a problematic issue.

4 Security Risk Assessment

In this section we introduce a *security risk assessment* tool, taking into account the above-mentioned considerations. This tool turns out to be a more plausible modeling of testing the occurrence of security hazards. However, if we add divergence (the absence of halting) as a security risk, this tool can not exist.

4.1 SHRAT, a Security Hazard Risk Assessment Tool

The following security risk assessment tool SHRAT with focus *shrat* may be conceived of as assessing a security hazard risk. In

$$P \subseteq shrat.ok \supseteq Q$$

the test action *shrat.ok* returns true if P is secure, and false if P is insecure (then P is avoided and Q is done instead). This seems to be a more rational test, because it only tests a property of a single behavior (its left argument). Using an external focus e, the test action *shrat.ok* in

$$(P_1 \unlhd e.m \trianglerighteq P_2) \unlhd shrat.ok \trianglerighteq Q$$

yields true because e.m is seen as an action that is beyond control of assessing security hazards. For testing shrat.ok actions we can employ a backtracking model: at $P \subseteq shrat.ok \supseteq Q$,

- temporarily remove loaded program (or behavior),
- place P instead

• execute until
$$\begin{cases} \text{true} & \Rightarrow commit, \\ external \ call \ \Rightarrow commit, \\ hazard & \Rightarrow backtrack. \end{cases}$$

So, for a behavior $(P_1 \unlhd shrat.ok \trianglerighteq P_2) \unlhd shrat.ok \trianglerighteq Q$, first evaluate the leftmost shrat.ok test action. If this yields true, then the rightmost does as well, otherwise evaluate $P_2 \unlhd shrat.ok \trianglerighteq Q$. For finite behaviors this is a terminating procedure, and not problematic. Some examples of the reply of shrat.ok:

```
\begin{split} S & \unlhd \mathit{shrat}.ok \trianglerighteq Q \Rightarrow \mathsf{true} \\ D & \unlhd \mathit{shrat}.ok \trianglerighteq Q \Rightarrow \mathsf{true} \\ (P_1 \unlhd f.m \trianglerighteq P_2) \unlhd \mathit{shrat}.ok \trianglerighteq Q \Rightarrow \mathsf{false} \end{split}
```

Evaluation of *shrat.ok* actions can be extended to a larger class of behaviors. A polarized *regular* behavior over B is defined by a finite system of equations over $\overline{P} = P_1, ..., P_n$ (for some $n \ge 1$) of the following form:

$$P_1 = F_1(\overline{P})$$

$$\vdots$$

$$P_n = F_n(\overline{P})$$

with
$$F_i(\overline{P}) ::= S \mid D \mid P_{i,1} \leq a_i \geq P_{i,2}$$
 where $P_{i,j} \in \{P_1, ..., P_n\}$ and $a_i \in B$.

Consider $P_1 \subseteq shrat.ok \supseteq Q$, thus $F_1(\overline{P}) \subseteq shrat.ok \supseteq Q$. Again we can decide the outcome of the test action shrat.ok by doing a finite number of substitutions, linear in n. (Loops and divergence are not considered security hazards.) This leads us to the following result:

Theorem 2. For regular behaviors, the tool SHRAT is possible.

We give an example: if

$$\begin{array}{ll} P_1 &=& P_2 \unlhd a \trianglerighteq P_1 \\ P_2 &=& P_1 \unlhd f.\mathit{skip} \trianglerighteq P_1 \quad (=f.\mathit{skip} \circ P_1), \end{array}$$

then shrat.ok in $(P_2 \le a \ge P_1) \le shrat.ok \ge Q$ yields true if it does in both

$$P_1 \unlhd shrat.ok \trianglerighteq Q$$
 and $P_2 \unlhd shrat.ok \trianglerighteq Q$.

Obviously, it does not in the latter case, so this behavior equals Q.

So, evaluation of the reply of *shrat.ok* is decidable for regular behaviors. This even remains the case if a stack is added as a reactor (based on the decidability of DPDA-equivalence [5]). We conclude that Cohen's impossibility result does not apply in this case; apparently, that result is about forecasting. Of course, the decidability of the reply of *shrat.ok* actions is lost if a Turing Tape is used as a reactor (see Section 5).

4.2 Divergence Risk Assessment

If we consider divergence as a security hazard, say by focus *drat* and reactor DRAT (Divergence Risk Assessment Tool), we have a totally different situation: in the behavior defined by

$$P = P \subseteq drat.ok \supseteq S$$

we then obviously want that the test action *drat.ok* returns the answer false. It is well-known that (in general) DRAT can not exist, as it would solve the Halting Problem (further discussed in Section 5).

Now, involving divergence as a security hazard in shrat.ok actions, we also find that in

$$P = P \triangleleft shrat.ok \triangleright f.m \circ S$$

the test should yield false (otherwise divergence). However, this yields a problem: in

$$P = P \le shrat.ok \ge S$$

this goes wrong: the termination problem (Turing impossibility result) "wins", and hence the back-tracking model is not suitable anymore. We conclude that SHRAT (a Security Hazard Risk Assessment Tool) does not exist if D (divergence) is considered a security hazard.

5 Digression: Using a Turing Machine

In this section we elaborate on a variant of the Halting Problem, which we call the Security Hazard Property (SHP), formalizing the situation that the execution of a certain program in a certain initial state establishes a security hazard. In order to give a concise presentation, we rely on a few notations and explanations given in [3], and we will be very precise about the part of that paper that is used below.

5.1 Preliminaries

Behaviors as considered in the above arise from PGA-programs (or programs in PGLB or PGLC). A more fundamental view on generators of behavior is the so-called SPI, the *Sequence of Primitive Instructions*. Of course, each program in PGA, PGLB or PGLC represents a SPI, but not each (computable) SPI can be represented by a program in one of the above-mentioned program notations, a simple example being

$$a; b; a; b; b; a; b^3; a; b^4; \dots$$

The above SPI defines a behavior that is not regular, and because each behavior definable by a PGA-program (PGLC-program) is regular, it is clear that we need an extended setting to specify such behaviors.

One such extension is the use of reactors, as was sketched in the above. In [2] we provide a formal treatment of the (potential) interaction of a behavior P with reactors H_e , H_g and H_f . Notation for that situation is the expression

$$((P/_fH_f)/_gH_g)/_eH_e$$
 or equivalently, $P/_fH_f/_gH_g/_eH_e$.

The operator $/_h$ (that takes as its left-argument a behavior and as its right-argument a reactor) is called the *use operator*, where h is some dedicated focus. In the previous part of the paper we considered all communications of P with a reactor H_h implicit and wrote P instead. In other words, an expression like $P \subseteq h.m \supseteq Q$ as occurring in the previous part of this paper is considered to abbreviate

$$(P \leq h.m \geq Q)/_hH_h$$
,

and this type of interaction is formalized in [2]. Furthermore, the variants of PGA and PGLC that use instructions with a focus explicitly intended for use-applications are called PGA:FMN and PGLC:FMN, where FMN abbreviates focus-method notation.

In the next section we prove that even with the availability of Turing computational power, it is undecidable that security hazards can be predicted. For this proof we use the notation and terminology explained in [3, Sections 3 and 4]. However, the material presented in Section 3.4 of [3] is not used in this exposition.

5.2 Proof of the Undecidability of Virus-Detection

The Security Hazard Property (SHP) can be modeled as follows: a PGLCi:FMN program p executing on the ETMT with initial configuration $\hat{\mathbf{b}} w \mathbf{b}$ (w a bit sequence) has a security hazard, notation $(p, w) \in \text{SHP}$, if

```
pgaEA(|p|_{pqlc}, etmt:ETMT(\hat{b}wb))
```

issues an action of the form f.m. We stipulate that program $q \in PGLCi:FMN$ solves the question whether $(p, w) \in SHP$ in the following way:

```
pgaEA(|q|_{pqlc}, etmt:ETMT(\hat{b}p; wb))
```

where p is stored as a bit sequence always halts, and after halting, the tape configuration is of the form

```
\begin{split} \mathsf{ETMT}(\,\hat{\mathbf{b}}\,1\sigma) \text{ for some } \sigma & \text{ if } \mathsf{pgaEA}(|p|_{pglc}, \, \textit{etmt} : \mathsf{ETMT}(\,\hat{\mathbf{b}}\,w\,\mathbf{b}\,)) \text{ has a security hazard,} \\ & \text{ thus } (p,w) \in \mathsf{SHP}, \end{split} \mathsf{ETMT}(\,\hat{\mathbf{b}}\,0\rho) \text{ for some } \rho & \text{ if } \mathsf{pgaEA}(|p|_{pglc}, \, \textit{etmt} : \mathsf{ETMT}(\,\hat{\mathbf{b}}\,w\,\mathbf{b}\,)) \text{ has no security hazard,} \\ & \text{ i.e., } (p,w) \not\in \mathsf{SHP}. \end{split}
```

Of course, it is assumed that the program q in no circumstance itself issues an f.m action, i.e., $(q, w) \notin SHP$ for all tape configurations w, and that its execution always terminates.

Theorem 3. The security hazard property is unsolvable by means of any program in PGLCi:FMN.

Proof. Suppose the contrary, i.e., there exists a program $q \in PGLCi:FMN$ that solves SHP. Let program s be defined as follows:

```
s = etmt.dup; q; r
```

with r = etmt.mv:right; -etmt.test:1; f.skip; etmt.mv:begin. Consider the question $(s,s) \stackrel{?}{\in}$ SHP. We show below that both assumptions $(s,s) \not\in$ SHP and $(s,s) \in$ SHP lead to a contradiction. Hence, s cannot exist, and thus q cannot exist.

First, assume that $(s, s) \notin SHP$. Then

```
\begin{split} & \text{pgaEA}(|s|_{pglc}, \ \textit{etmt}\text{:ETMT}(\,\hat{\mathbf{b}}\,s\,\mathbf{b}\,)) \\ & \quad \downarrow \tau \quad (\textit{dup}) \\ & \text{pgaEA}(|q;r|_{pglc}, \ \textit{etmt}\text{:ETMT}(\,\hat{\mathbf{b}}\,s;s\,\mathbf{b}\,)). \end{split}
```

Because $q \in PGLCi:FMN$, the program q; r first executes q (which terminates successfully by assumption) and then starts with the first instruction of r. Thus,

```
pgaEA(|q; r|_{pglc}, etmt:ETMT(\hat{\mathbf{b}} s; s \mathbf{b}))

\downarrow \tau (by program q)

pgaEA(|r|_{pglc}, etmt:ETMT(\hat{\mathbf{b}} 0 \sigma \mathbf{b}))
```

for some string σ . The remaining behavior is displayed in Figure 2, and results in

```
pgaEA(|f.skip; etmt.mv:begin|_{pglc}, etmt:ETMT(b \hat{0}\sigma b)).
```

This last AnArch clearly represents a security hazard because of the first instruction f.skip, and therefore $(s, s) \in SHP$. Contradiction.

```
\begin{split} & \operatorname{pgaEA}(|r|_{pglc},\ etmt:\operatorname{ETMT}(\hat{\mathbf{b}}\,0\sigma\,\mathbf{b}\,)) \\ &= \\ & \operatorname{pgaEA}(|etmt.mv:right;-etmt.test:1;\ f.skip;\ etmt.mv:begin|_{pglc},\ etmt:\operatorname{ETMT}(\hat{\mathbf{b}}\,0\sigma\,\mathbf{b}\,)) \\ &\downarrow \tau \quad (etmt.mv:right) \\ & \operatorname{pgaEA}(|-etmt.test:1;\ f.skip;\ etmt.mv:begin|_{pglc},\ etmt:\operatorname{ETMT}(\mathbf{b}\,\hat{\mathbf{0}}\sigma\,\mathbf{b}\,)) \\ &\downarrow \tau \quad (-etmt.test:1) \\ & \operatorname{pgaEA}(|f.skip;\ etmt.mv:begin|_{pglc},\ etmt:\operatorname{ETMT}(\mathbf{b}\,\hat{\mathbf{0}}\sigma\,\mathbf{b}\,)). \end{split}
```

Figure 2: Critical state of the behavior in case $(s, s) \notin SHP$ in the proof of Thm. 3

Now assume that $(s,s) \in SHP$. The resulting computation is displayed in Figure 3 (for some string ρ). Here the last configuration represents halting without having executed any f.m action, and therefore $(s,s) \notin SHP$ and again a contradiction occurs.

```
\begin{aligned} & \text{pgaEA}(|s|_{pglc},\ etmt: \text{ETMT}(\,\hat{\mathbf{b}}\,s\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (dup) \\ & \text{pgaEA}(|q;r|_{pglc},\ etmt: \text{ETMT}(\,\hat{\mathbf{b}}\,s;s\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (\text{by program }q) \\ & \text{pgaEA}(|r|_{pglc},\ etmt: \text{ETMT}(\,\hat{\mathbf{b}}\,1\rho\,\mathbf{b}\,)) \\ & = \\ & \text{pgaEA}(|etmt.mv:right; -etmt.test:1;\ f.skip; etmt.mv:begin|_{pglc},\ etmt: \text{ETMT}(\,\hat{\mathbf{b}}\,1\rho\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (etmt.mv:right) \\ & \text{pgaEA}(|-etmt.test:1;\ f.skip; etmt.mv:begin|_{pglc},\ etmt: \text{ETMT}(\,\mathbf{b}\,\hat{\mathbf{1}}\rho\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (-etmt.test:1) \\ & \text{pgaEA}(|etmt.mv:begin|_{pglc},\ etmt: \text{ETMT}(\,\mathbf{b}\,\hat{\mathbf{1}}\rho\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (etmt.mv:begin|_{pglc},\ etmt: \text{ETMT}(\,\mathbf{b}\,\hat{\mathbf{1}}\rho\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (etmt.mv:begin|_{pglc},\ etmt: \text{ETMT}(\,\mathbf{b}\,\hat{\mathbf{1}}\rho\,\mathbf{b}\,)) \\ & \downarrow \tau \quad (etmt.mv:begin) \\ & \sqrt{\quad \text{with ETMT's configuration: } etmt: \text{ETMT}(\,\hat{\mathbf{b}}\,\mathbf{1}\rho\,\mathbf{b}\,). \end{aligned}
```

Figure 3: The case that $(s, s) \in SHP$ in the proof of Thm. 3

So our supposition was definitely wrong, i.e., there is no program $q \in PGLCi:FMN$ that solves the security hazard property.

References

- [1] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
- [2] J.A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
- [3] J.A. Bergstra and A. Ponse. Execution architectures for program algebra. Logic Group Preprint Series 230, http://preprints.phil.uu.nl/lgps/?lang=en, Dept. of Philosophy, Utrecht University, 2004.
- [4] F. Cohen. Computer viruses theory and experiments, 1984. http://vx.netlux.org/lib/afc01.html. Version including some corrections and references: *Computers & Security* 6(1): 22-35, 1987.
- [5] C.P. Stirling. Decidability of DPDA equivalence. *Theoretical Computer Science*, 255, 1-31, 2001.

A Appendix - Terminology and Quotes from the Web

Ball State University. What is a Computer Virus?² A computer virus is a type of computer program that is designed to hide in the background and replicate itself from one computer to another by attaching itself to existing programs or parts of the operating system. Users can unknowingly transmit a virus from one system to another by copying infected program files from one machine to another or by using an infected program downloaded from the Internet. Computer viruses often infect many programs on the same system or even parts of the operating system in an attempt to spread themselves as far as possible.

A computer virus will often have a "trigger" such as a specific date or a specific program being run that will perform a benign event such as flashing a message on the users screen. Some viruses will perform more malicious deeds however, deleting or scrambling users files or their entire system. Some viruses also slow down a users system, disable certain functions, or cause erratic system behavior.

Computer Worm (Definition).³ Worms are very similar to viruses in that they are computer programs that replicate themselves and that often, but not always, contain some functionality that will interfere with the normal use of a computer or a program.

The difference is that unlike viruses, worms exist as separate entities; they do not attach themselves to other files or programs. A worm can spread itself automatically over the network from one computer to the next. Worms take advantage of automatic file sending and receiving features found on many computers.

Webopedia. <u>Virus.</u>⁴ A program or piece of code that is loaded onto your computer without your knowledge and runs against your wishes. Viruses can also replicate themselves. All computer viruses are manmade. A simple virus that can make a copy of itself over and over again is relatively easy to produce. Even such a simple virus is dangerous because it will quickly use all available memory and bring the system to a halt. An even more dangerous type of virus is one capable of transmitting itself across networks and bypassing security systems.

²www.bsu.edu/ucs/article/0,1370,6254-1978-3303,00.html

³www.bsu.edu/ucs/article/0,1370,6254-1978-4488,00.html

 $^{^4 {\}tt www.webopedia.com/TERM/v/virus.html}$

Since 1987, when a virus infected ARPANET, a large network used by the Defense Department and many universities, many antivirus programs have become available. These programs periodically check your computer system for the best-known types of viruses.

Some people distinguish between general viruses and worms. A worm is a special type of virus that can replicate itself and use memory, but cannot attach itself to other programs.

<u>Worm.</u>⁵ A program or algorithm that replicates itself over a computer network and usually performs malicious actions, such as using up the computer's resources and possibly shutting the system down.

<u>Trojan horse</u>. A destructive program that masquerades as a benign application. Unlike viruses, Trojan horses do not replicate themselves but they can be just as destructive. One of the most insidious types of Trojan horse is a program that claims to rid your computer of viruses but instead introduces viruses onto your computer.

The term comes from a story in Homer's Iliad, in which the Greeks give a giant wooden horse to their foes, the Trojans, ostensibly as a peace offering. But after the Trojans drag the horse inside their city walls, Greek soldiers sneak out of the horse's hollow belly and open the city gates, allowing their compatriots to pour in and capture Troy.

⁵www.webopedia.com/TERM/w/worm.html

⁶www.webopedia.com/TERM/T/Trojan_horse.html