Available online at www.sciencedirect.com
ScienceDirect
Journal of Applied Logic 5 (2007) 170–192
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Execution architectures for program algebra

Jan A. Bergstra [a,b], Alban Ponse [a,*]

[a] *University of Amsterdam, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*
[b] *Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

Received 8 September 2004; accepted 10 October 2005

Available online 3 May 2006

Abstract

We investigate the notion of an execution architecture in the setting of the program algebra PGA, and distinguish two sorts of these: *analytic* architectures, designed for the purpose of explanation and provided with a process-algebraic, compositional semantics, and *synthetic* architectures, focusing on how a program may be a physical part of an execution architecture. Then we discuss in detail the Turing machine, a well-known example of an analytic architecture. The logical core of the halting problem—the inability to forecast termination behavior of programs—leads us to a few approaches and examples on related issues: *forecasters* and *rational agents*. In particular, we consider architectures suitable to run a Newcomb Paradox system and the Prisoner's Dilemma.

Keywords: Halting problem; Execution of programs; Program algebra; Turing machine

1. Introduction

The program algebra PGA introduced in [8] aims at the clarification of the concept of a program at the simplest possible level. In this paper we will use PGA as a vehicle to study fundamentals of the execution of programs. Having available a rigid definition of what a program is, the subject of how programs may be used raises compelling questions. This paper focuses on the notion of an *execution architecture*. This notion is more general than that of a machine and admits many different forms of interaction between a program and its context.

First, we consider programs modelled independently of any execution environment by means of *threads*. In this view, it is unavoidable to contemplate computable threads as a general semantic category. It turns out that computable instruction streams can describe all computable threads. Using finite programs only (regular instruction streams), an execution architecture with the well-known Turing machine tape as a *state machine*, i.e., an interacting data structure, is sufficiently powerful to denote all computable threads as well.

Then, an attempt is made to cover the most important phenomena regarding program execution in a context. Two kinds of execution architectures are defined: *analytic* architectures (AnArchs) and *synthetic* architectures (SynArchs). An AnArch serves to model how a program may be executed in a context by making explicit its interaction with other system components, in particular the so-called *reactors*. A reactor is a generalization of a state machine: it may also

* Corresponding author.
E-mail address: alban@science.uva.nl (A. Ponse).

1570-8683/$ – see front matter
doi:10.1016/j.jal.2005.10.013

have interaction with other parties than the control. A SynArch focuses on how a program may be a physical part of a context. The AnArch is useful for explanation, while a SynArch may play a role during construction. It is shown that all SynArchs admit a specification in a process algebra with abstraction and recursion operators.

In the remainder of the paper some special analytic execution architectures are discussed. First, we consider the Turing machine and introduce an enhanced version of the Turing machine tape (TMT) in the form of a state machine ETMT. Finite control is phrased in terms of programs executed in an analytical architecture providing only the ETMT. In this setting the halting problem takes the form of the nonexistence of certain programs, which is demonstrated in full detail.

Ignoring the (E)TMT, the halting problem reduces to its logical core: the inability to forecast termination behavior of programs that may use the results of forecasting. It is shown how an analytic architecture can be used to give a sound definition of a *forecasting reactor*, and it is demonstrated that a correct forecaster needs to escape from the two classical truth values. This brings the halting problem close to some logical paradoxes, in particular the Liar Paradox.

A *rational agent* is a reactor that has the objective to achieve certain goals by giving appropriate replies for specific requests. It is shown that in some cases also a rational agent needs to use more truth values than true and false. Combining rational agents and forecasting reactors provides a remarkable setting, in which the famous Newcomb Paradox [17] can be modelled. This paradox seems to prove that the very concept of a forecaster reliably forecasting a rational agent is utterly problematic. Nevertheless this is done all the time in stock market transactions, gaming technology and so on. Using the analytic architectures and some, perhaps exotic process algebra involving the constant 0 (see [3]), a formalization of one reactor forecasting another reactor is given. The Newcomb Paradox now shows up as follows: given a fixed execution architecture (viewed as a geometric structure with several components), its process semantics determines what a rational agent reactor should best reply in order to achieve a specific objective. The normal process semantics predicts one reply as being rational, whereas the semantics specifically tailored to forecasting predicts a different reply. But the normal semantics is so robust that it seems to take into account the possibility that one reactor predicts the behavior of another reactor just as well. The novelty of our approach is a precise formalization of the conditions required to run both executions of the Newcomb Paradox system. As a last and related example, we discuss the well-known Prisoner's Dilemma [18].

In the setting of program algebra one can consider execution architectures that take security matters into account, and establish an analysis in similar style. Based on the undecidability of virus detection as described by Cohen [15], a first result is recorded in [14].

The further content of this paper is divided into five parts: in Section 2 we formally introduce threads and state machines. In Section 3 we introduce the program algebra PGA. Then, in Section 4 we introduce execution architectures. In Section 5, we study the Turing machine as an example of an analytic architecture. Finally, in Section 6, we focus on forecasting reactors and rational agents in the setting of (analytic) execution architectures.

2. Threads and state machines

The behavior of programs under execution is modelled by *threads*. In this section we introduce thread algebra. Then we introduce state machines, devices that can be *used* by a thread in order to increase expressiveness.

2.1. Thread algebra

Basic thread algebra, or BTA for short, is intended for the description of sequential program behavior (see [9]; in [8] BTA is introduced as *basic polarized process algebra*). Based on a finite set A of *actions* it has the following constants and operators:

- the *termination* constant S,
- the *deadlock* or *inaction* constant D,
- for each $a \in A$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use *action prefixing* $a \circ P$ as an abbreviation for $P \trianglelefteq a \trianglerighteq P$ and take $\circ$ to bind strongest. Furthermore, for $n \in \mathbb{N}$ we define $a^n \circ P$ by $a^0 \circ P = P$ and $a^{n+1} \circ P = a \circ (a^n \circ P)$.

The operational intuition behind thread algebra is that each action represents a command which is to be processed by the execution environment of a thread. More specifically, an action is taken as a command for a service offered by the environment. The processing of a command may involve a change of state of this environment. At completion of the processing of the command, the service concerned produces a reply value true or false to the thread under execution. The thread $P \trianglelefteq a \trianglerighteq Q$ will then proceed as P if the processing of a yielded the reply true indicating successful processing, and it will proceed as Q if the processing of a yielded the reply false.

BTA can be equipped with a partial order and an *approximation operator*.

(1) $\sqsubseteq$ is the partial ordering on BTA generated by the clauses
   (a) for all $P \in \mathrm{BTA}$, $\mathsf{D} \sqsubseteq P$, and
   (b) for all $P_1, P_2, Q_1, Q_2 \in \mathrm{BTA}$, $a \in A$,

$$P_1 \sqsubseteq Q_1 \;\&\; P_2 \sqsubseteq Q_2 \Rightarrow P_1 \trianglelefteq a \trianglerighteq P_2 \sqsubseteq Q_1 \trianglelefteq a \trianglerighteq Q_2.$$

(2) $\pi : \mathbb{N} \times \mathrm{BTA} \to \mathrm{BTA}$ is the approximation operator determined by the equations
   (a) for all $P \in \mathrm{BTA}$, $\pi(0, P) = \mathsf{D}$,
   (b) for all $n \in \mathbb{N}$, $\pi(n+1, \mathsf{S}) = \mathsf{S}$, $\pi(n+1, \mathsf{D}) = \mathsf{D}$, and
   (c) for all $P, Q \in \mathrm{BTA}$, $n \in \mathbb{N}$,

$$\pi(n+1, P \trianglelefteq a \trianglerighteq Q) = \pi(n, P) \trianglelefteq a \trianglerighteq \pi(n, Q).$$

   We further write $\pi_n(P)$ instead of $\pi(n, P)$.

The operator π finitely approximates every thread in BTA. That is, for all $P \in \mathrm{BTA}$,

$$\exists n \in \mathbb{N}\ \pi_0(P) \sqsubseteq \pi_1(P) \sqsubseteq \cdots \sqsubseteq \pi_n(P) = \pi_{n+1}(P) = \cdots = P.$$

Threads can be finite or infinite. Following the metric theory of [2] as the basis of processes in [7], BTA has a completion BTA^∞ which comprises also infinite threads. Standard properties of the completion technique yield that we may take BTA^∞ as the cpo consisting of all so-called *projective* sequences. That is,

$$\mathrm{BTA}^\infty = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}\ (P_n \in \mathrm{BTA} \;\&\; \pi_n(P_{n+1}) = P_n)\}$$

with

$$(P_n)_{n\in\mathbb{N}} \sqsubseteq (Q_n)_{n\in\mathbb{N}} \quad \Leftrightarrow \quad \forall n \in \mathbb{N}\ P_n \sqsubseteq Q_n$$

and

$$(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}} \quad \Leftrightarrow \quad \forall n \in \mathbb{N}\ P_n = Q_n.$$

(For a detailed account of this construction see [4].)

Let $I = \{1, \ldots, n\}$ for some $n > 0$. A *finite linear recursive specification* over BTA is a set of equations

$$X_i = t_i(\overline{X})$$

for $i \in I$ with $\overline{X} = X_1, \ldots, X_n$ and all $t_i(\overline{X})$ of the form S, D, or $X_{i_l} \trianglelefteq a_i \trianglerighteq X_{i_r}$ for $i_l, i_r \in I$ and $a_i \in A$. In BTA^∞, finite linear recursive specifications represent continuous operators having as unique fixed points *regular* threads, i.e., threads which can only reach finitely many states.

Example 1. Let $n > 0$. The regular thread $a^n \circ \mathsf{D}$ is the fixed point for X_1 in the specification

$$\{X_i = a \circ X_{i+1} \mid i = 1, \ldots, n\} \cup \{X_{n+1} = \mathsf{D}\}.$$

The regular thread $a^n \circ \mathsf{S}$ is the fixed point for X_1 in

$$\{X_i = a \circ X_{i+1} \mid i = 1, \ldots, n\} \cup \{X_{n+1} = \mathsf{S}\}.$$

Both these threads are finite.

The infinite regular thread a^∞ is the fixed point for X_1 in the specification $\{X = a \circ X\}$ and corresponds to the projective sequence $(P_n)_{n\in\mathbb{N}}$ with $P_0 = \mathsf{D}$ and $P_{n+1} = a \circ P_n$.

Observe that e.g. $a^n \circ \mathsf{D} \sqsubseteq a^n \circ \mathsf{S}$, $a^n \circ \mathsf{D} \sqsubseteq a^\infty$ but $a^n \circ \mathsf{S} \not\sqsubseteq a^\infty$.

For the sake of simplicity, we shall often define regular threads by providing only one or more equations. For example, we say that $P = a \circ P$ defines a regular thread with name P (so $P = a^\infty$ in this case).

We end this section with the observation that for regular threads P and Q, $P \sqsubseteq Q$ is decidable. Because one can always take the disjoint union of two recursive specifications, it suffices to argue that $P_i \sqsubseteq P_j$ in

$$P_1 = t_1(\overline{P}), \ldots, P_n = t_n(\overline{P})$$

is decidable. This follows from the assertion

$$\forall i, j \leqslant n\ \pi_n(P_i) \sqsubseteq \pi_n(P_j) \quad \Leftrightarrow \quad P_i \sqsubseteq P_j, \tag{1}$$

where $\pi_l(P_k)$ is defined by $\pi_l(t_k(\overline{P}))$, because $\sqsubseteq$ is decidable for finite threads. Without loss of generality, assume $n > 1$. To prove (1), observe that $\Leftarrow$ follows by definition of regular threads. For the reverse, choose i, j and assume that $\pi_n(P_i) \sqsubseteq \pi_n(P_j)$. Suppose $P_i \not\sqsubseteq P_j$, then for some $k > n$, $\pi_k(P_i) \not\sqsubseteq \pi_k(P_j)$ while $\pi_{k-1}(P_i) \sqsubseteq \pi_{k-1}(P_j)$. So there exists a trace of length k from P_i of the form

$$P_i \xrightarrow{a_{\mathsf{true}}} P_{i'} \xrightarrow{b_{\mathsf{false}}} \cdots$$

that is not a trace of P_j, while by the assumption the first n actions are a trace of P_j. These n actions are connected by $n+1$ states, and because there are only n different states $P_1, \ldots, P_n$, a repetition occurs in this sequence of states. So the trace witnessing $\pi_k(P_i) \not\sqsubseteq \pi_k(P_j)$ can be made shorter, contradicting k's minimality and hence the supposition. Thus $P_i \sqsubseteq P_j$. As a consequence, also $P = Q$ (i.e., $P \sqsubseteq Q$ and $Q \sqsubseteq P$) is decidable for regular threads P and Q.

2.2. State machines

A *state machine* is a pair $\langle \Sigma, F \rangle$ consisting of a set Σ of so-called *co-actions* and a *reply function* F. This reply function is a mapping that gives for each finite sequence of co-actions from Σ a reply value true or false. State machines were introduced in [13].

Example 2. A *counter* is a state machine $C = \langle \Sigma, F \rangle$ with $\Sigma = \{inc, dec\}$ consisting of the increase and decrease co-actions and the reply function F which replies true to *inc*, and false to *dec* if and only if the counter is zero. The initial value of C is zero, and the counter has a value in each state. Below, we return to this example.

In order to provide a specific description of the interaction between a thread and a state machine, we will use for actions the general notation $c.a$ where c is the so-called *channel* or *focus*, and a is the co-action. For example, $c.inc$ is the action which increases a counter via channel c. This interaction is defined with help of the *use operator* $/$. For a state machine $\mathcal{S} = \langle \Sigma, F \rangle$, a finite thread P and a channel c, the defining rules for $P/_c\mathcal{S}$ (the thread P *using the state machine* $\mathcal{S}$ *via channel* c) are:

$$\mathsf{S}/_c\mathcal{S} = \mathsf{S},$$
$$\mathsf{D}/_c\mathcal{S} = \mathsf{D},$$
$$(P \trianglelefteq c'.a \trianglerighteq Q)/_c\mathcal{S} = (P/_c\mathcal{S}) \trianglelefteq c'.a \trianglerighteq (Q/_c\mathcal{S}) \quad \text{if } \mathsf{c}' \neq \mathsf{c},$$
$$(P \trianglelefteq c.a \trianglerighteq Q)/_c\mathcal{S} = P/_c\mathcal{S}' \quad \text{if } a \in \Sigma \text{ and } F(a) = \mathsf{true},$$
$$(P \trianglelefteq c.a \trianglerighteq Q)/_c\mathcal{S} = Q/_c\mathcal{S}' \quad \text{if } a \in \Sigma \text{ and } F(a) = \mathsf{false},$$
$$(P \trianglelefteq c.a \trianglerighteq Q)/_c\mathcal{S} = \mathsf{D} \quad \text{if } a \notin \Sigma,$$

where $\mathcal{S}' = \langle \Sigma, F' \rangle$ with $F'(\sigma) = F(a\sigma)$ for all co-action sequences $\sigma \in \Sigma^+$. The use operator is expanded to infinite threads P by stipulating

$$P/_c\mathcal{S} = \big(\pi_n(P)/_c\mathcal{S}\big)_{n\in\mathbb{N}}.$$

As a consequence, $P/_c\mathcal{S} = \mathsf{D}$ if for any n, $\pi_n(P)/_c\mathcal{S} = \mathsf{D}$. Finally, repeated applications of the use operator bind to the left, thus

$$P/_{c0}\mathcal{S}_0/_{c1}\mathcal{S}_1 = (P/_{c0}\mathcal{S}_0)/_{c1}\mathcal{S}_1.$$

We end this section with an example on the use of a state machine.

Example 3. Let $\{a, b, c.inc, c.dec\} \subseteq A$, where the last two actions refer to the counter C defined in Example 2. We write $C(n)$ for a counter with value $n \in \mathbb{N}$, so $C = C(0)$. By the defining equations for the use operator it follows that for any thread P,

$$(c.inc \circ P)/_c C(0) = P/_c C(1),$$

and $\forall n \in \mathbb{N}$, $(c.inc \circ P)/_c C(n) = P/_c C(n+1)$. Furthermore, it easily follows that

$$(P \trianglelefteq c.dec \trianglerighteq \mathsf{S})/_c C(n) = \begin{cases} \mathsf{S} & \text{if } n = 0, \\ P/_c C(n-1) & \text{otherwise.} \end{cases}$$

Now consider the regular thread Q defined by[1]

$$Q = (c.inc \circ Q) \trianglelefteq a \trianglerighteq R,$$
$$R = b \circ R \trianglelefteq c.dec \trianglerighteq \mathsf{S}.$$

Then

$$\begin{aligned} Q/_c C(0) &= ((c.inc \circ Q) \trianglelefteq a \trianglerighteq R)/_c C(0) \\ &= (Q/_c C(1)) \trianglelefteq a \trianglerighteq (R/_c C(0), \end{aligned}$$

and for all $n \in \mathbb{N}$, $Q/_c C(n) = (Q/_c C(n+1)) \trianglelefteq a \trianglerighteq (R/_c C(n))$. It is not hard to see that $Q/_c C(0)$ is an infinite thread with the property that for all n, a trace of $n+1$ a-actions produced by n positive and one negative reply on a is followed by $b^n \circ \mathsf{S}$. This yields an *irregular* thread: if $Q/_c C(0)$ were regular, it would be a fixed point of some finite linear recursive specification, say with k equations. But specifying a trace $b^k \circ \mathsf{S}$ already requires $k+1$ linear equations $X_1 = b \circ X_2, \ldots, X_k = b \circ X_{k+1}, X_{k+1} = \mathsf{S}$, which contradicts the assumption. So $Q/_c C(0)$ is not regular.

3. Programs and program algebra

In this section we introduce the program algebra PGA (see [8]) and discuss its relation with thread algebra. Furthermore, we show that each computable thread can be expressed by either a computable sequence of PGA-instructions, or by a regular sequence of PGA-instructions that uses a Turing machine tape as a state machine.

3.1. PGA, basics of program algebra

Given a thread algebra with actions in A, we now consider the actions as so-called *basic instructions*. The syntax of PGA has the following primitive instructions as constants:

Basic instruction $a \in A$. It is assumed that upon the execution of a basic instruction, the (executing) environment provides an answer true or false. However, in the case of a basic instruction, this answer is not used for program control. After execution of a basic instruction, the next instruction (if any) will be executed; if there is no next instruction, inaction will occur.

Positive/negative test instruction $\pm a$ *for* $a \in A$. A positive test instruction $+a$ executes like the basic instruction a. Upon false, the program skips its next instruction and continues with the instruction thereafter; upon true the program executes its next instruction. For a negative test instruction $-a$, this is reversed: upon true, the program skips its next instruction and continues with the instruction thereafter; upon false the program executes its next instruction. If there is no subsequent instruction to be executed, inaction occurs.

Termination instruction !. This instruction prescribes successful termination.

Jump instruction $\#k$ *(*$k \in \mathbb{N}$*).* This instruction prescribes execution of the program to jump k instructions forward; if there is no such instruction, inaction occurs. In the special case that $k = 0$, this prescribes a jump to the instruction itself and inaction occurs, in the case that $k = 1$ this jump acts as a *skip* and the next instruction is executed. In the case that the prescribed instruction is not available, inaction occurs.

[1] Note that a *linear* recursive specification of Q requires (at least) five equations.

Table 1
Axioms for PGA's instruction sequence congruence

$(X; Y); Z = X; (Y; Z)$	(PGA1)	$X^\omega;\ Y = X^\omega$	(PGA3)
$(X^n)^\omega = X^\omega$ for $n > 0$	(PGA2)	$(X; Y)^\omega = X; (Y; X)^\omega$	(PGA4)

Table 2
Equations for thread extraction on PGA

$\lvert!\rvert = \mathsf{S}$	$\lvert!; X\rvert = \mathsf{S}$	$\lvert\#k\rvert = \mathsf{D}$
$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert a; X\rvert = a \circ \lvert X\rvert$	$\lvert\#0; X\rvert = \mathsf{D}$
$\lvert{+}a\rvert = a \circ \mathsf{D}$	$\lvert{+}a; X\rvert = \lvert X\rvert \trianglelefteq a \trianglerighteq \lvert\#2; X\rvert$	$\lvert\#1; X\rvert = \lvert X\rvert$
$\lvert{-}a\rvert = a \circ \mathsf{D}$	$\lvert{-}a; X\rvert = \lvert\#2; X\rvert \trianglelefteq a \trianglerighteq \lvert X\rvert$	$\lvert\#k + 2; u\rvert = \mathsf{D}$
		$\lvert\#k + 2; u; X\rvert = \lvert\#k + 1; X\rvert$

PGA-terms are composed by means of *concatenation*, notation _; _, and *repetition*, notation $(\_)^\omega$. Instruction sequence congruence for PGA-terms is axiomatized by the axioms PGA1-4 in Table 1. Here PGA2 is an axiom-*scheme*: for each $n > 0$, $(X^n)^\omega = X^\omega$, where $X^1 = X$ and $X^{k+1} = X; X^k$. A closed PGA-term is often called a PGA-program.

From the axioms PGA1–4 one easily derives *unfolding*, i.e.,

$$X^\omega = X; X^\omega.$$

Furthermore, each PGA-program can be rewritten into an instruction equivalent *canonical form*, i.e., a closed term of the form X or $X; Y^\omega$ with X and Y not containing repetition. This also follows from the axioms in Table 1.

We use the abbreviation SPI for Sequence of Primitive Instructions. A SPI is also called a *program object*, or sometimes shortly, a *program*. PGA-programs represent a certain class of SPI's, called the *regular* SPI's. In particular, each regular SPI can be represented in PGA as a canonical form.

We will often use basic instructions in so-called *focus.method* notation, i.e., basic instructions of the form

$$f.m$$

where f is a focus (channel name) and m a method name. The m here is sometimes called a *service-instruction* because it refers to the use of some state machine, and is related with a co-action as defined in Section 2.2. Two examples of instructions in focus.method notation are $c.inc$ and $c.dec$, related with the actions controlling a counter discussed in Example 3. In the next section we will relate all basic and test instructions to the actions of a thread; this is called *thread extraction*.

3.2. Thread extraction: from PGA to thread algebra

The *thread extraction* operator $\lvert X\rvert$ assigns a thread to program object X. Thread extraction is defined by the thirteen equations in Table 2, where $a \in A$ and u is a primitive instruction.

Some examples:

$$\begin{aligned}
\lvert(\#0)^\omega\rvert &= \lvert\#0; (\#0)^\omega\rvert \\
&= \mathsf{D}, \\
\lvert{-}a; b; c\rvert &= \lvert\#2; b; c\rvert \trianglelefteq a \trianglerighteq \lvert b; c\rvert \\
&= \lvert\#1; c\rvert \trianglelefteq a \trianglerighteq b \circ \lvert c\rvert \\
&= \lvert c\rvert \trianglelefteq a \trianglerighteq b \circ c \circ \mathsf{D} \\
&= c \circ \mathsf{D} \trianglelefteq a \trianglerighteq b \circ c \circ \mathsf{D}.
\end{aligned}$$

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

$$\lvert(\#2; a)^\omega\rvert = \lvert\#2; a; (\#2; a)^\omega\rvert = \lvert\#1; (\#2; a)^\omega\rvert = \lvert(\#2; a)^\omega\rvert = \cdots.$$

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

It is also possible that thread extraction yields an infinite recursion, e.g.,

$$|a^\omega| = |a; a^\omega| = a \circ |a^\omega|$$

(in the previous section we denoted this thread by a^∞). If the behavior of X is infinite, it is regular and can be represented by a (linear) recursive specification, e.g.,

$$\begin{aligned} |(a; +b; \#3; -b; \#4)^\omega| = P \text{ in} \\ P = a \circ (P \trianglelefteq b \trianglerighteq Q), \\ Q = P \trianglelefteq b \trianglerighteq Q. \end{aligned}$$

It follows easily that any PGA-program defines a regular thread, and conversely, each regular thread can be defined in PGA: linear equations of the form $X = \mathsf{S}$ or $X = \mathsf{D}$ can be defined by instructions ! and #0, respectively, and a linear equation

$$X = Y \trianglelefteq a \trianglerighteq Z$$

can be associated with a triple $+a; \#k; \#l$. Connecting these program fragments in a repetition and instantiating the jump counters k and l with the appropriate values then yields a PGA-program that defines a solution for the first equation. A typical example:

$$\begin{aligned} &P_1 = P_2 \trianglelefteq a \trianglerighteq P_2, && (+a; \#2; \#1; \\ &P_2 = P_3 \trianglelefteq b \trianglerighteq P_1, \quad \mapsto && +b; \#2; \#2; \\ &P_3 = \mathsf{D}. && \#0)^\omega. \end{aligned}$$

3.3. Programming computable threads

A thread is *computable* if it can be represented by an identifier E_0 and two computable functions g, f in the following way ($k \in \mathbb{N}$):

$$E_k = \begin{cases} \mathsf{D} & \text{if } g(k) = 0, \\ \mathsf{S} & \text{if } g(k) = 1, \\ E_{\langle k+f(k),1\rangle} \trianglelefteq a_{g(k)} \trianglerighteq E_{\langle k+f(k),2\rangle} & \text{if } g(k) > 1. \end{cases}$$

Here we use the bijective pairing function $\langle \_, \_ \rangle$ defined by $\langle n, m\rangle = \frac{1}{2}((n+m)^2 + 3m + n)$. So $n < \langle n+i, 1\rangle < \langle n+i, 2\rangle$ for all $n, i \in \mathbb{N}$.

Theorem 4. *PGA's sequences of primitive instructions* (*SPI's*) *are universal*: *for each computable thread α there is a SPI with α as its behavior.*

Proof. Let E_0 be a computable thread as defined above. Then we define

$$\tilde{E}_k = \begin{cases} \#0; \#0; \#0 & \text{if } g(k) = 0, \\ !; !; ! & \text{if } g(k) = 1, \\ +a_{g(k)}; \#3(\langle k + f(k), 1\rangle - k - 1) + 2;\ \#3(\langle k + f(k), 2\rangle - k - 1) + 1 & \text{if } g(k) > 1. \end{cases}$$

It is easily seen that $E_0 = |\tilde{E}_0; \tilde{E}_1; \ldots|$ (and that $E_k = |\tilde{E}_k; \tilde{E}_{k+1}; \ldots|$). □

Furthermore, PGA's *repeating* sequences of instructions—the *regular* SPI's—are universal with the aid of a state machine TMT (Turing machine tape) if we restrict to a finite number of actions:

Theorem 5. *For each computable thread α there is a closed PGA-term X such that $|X|/_{\mathrm{tmt}}\ \mathrm{TMT} = \alpha$.*

This is a standard result in the setting of Turing machines (see, e.g., [19,20]), given the fact that finite control can be modelled in PGA.

4. Execution architectures

In this section we focus on programs in an execution architecture. We will use ACP-based process algebra to model so-called 'analytical architectures'. In Appendix A we shortly recall the process algebra used. Finally, we try to clarify the role of programs as binaries in machines.

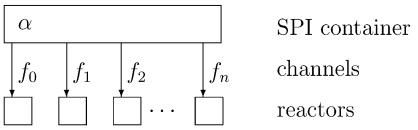
4.1. Analytic versus synthetic architectures

We consider the following types of execution architectures:

Analytic Architecture (AnArch): provides a hypothetical decomposition of a system into parts. An AnArch can serve as an explanation of a setting in a black box context (the system is seen as a blackbox, with the AnArch describing its internals for the sake of explanation). An AnArch will not be on the pathway to construction. In Section 4.2 we discuss this type of architecture in detail.

Synthetic Architecture (SynArch): an architecture (description of how a whole is composed from parts) providing information on the true (or proposed, intended) internal structure of a system. Often a compositional semantic technique is absent. Typically, a program in compiled form (a *binary*) can be a part of a SynArch. In Section 4.3 we discuss these architectures.

The proposed execution architecture for PGA is an AnArch consisting of several interacting components. First there is a component containing an instruction sequence. This component is called the SPI *container*; it is able to process one instruction at a time and issue the appropriate action. Next there are *reactors*. A reactor is a generalized state machine: it can not only process actions generated by the SPI container and issue replies, but may also engage in external communications. No attention is paid to the way in which a SPI may in fact be stored or generated. We visualize an AnArch in the following way:



Here α represents a SPI of which the basic instructions are in focus.method notation for foci/channels $f_0, \ldots, f_n$, thus of the form $f_i.a$ (see Section 3.1). These channels play a reserved role and are supposed not to be composed with other parts of the AnArch or any extension of it. As in the case of state machines (see Section 2.2), a reactor is unaware of the name of the SPI-channel that addresses it. We will not be very specific about the definition of a reactor, and impose only a few requirements. A first semantics for reactors comprises that of state machines: a reply function is defined that takes a history of requests as input (a sequence of co-actions) and upon the request via a focused action produces a reply true or false.

In Section 4.2 we introduce a process based semantics for AnArchs. A process is a mathematical entity in a space of processes (like a number being an element of a field). The design of the process space depends on the underlying theory of processes used. We will use ACP-based process algebra ([7]; in Appendix A we shortly recall the process algebra used), but many other process theories can be used instead.

The purpose of the use of processes is *specification*. Here, 'specification' is used in a fairly limited way: it must be compared with 'quantification' (stating numerical sizes) and 'qualification' (expressing objectives, goals, methods and reasons). Furthermore, specification stands for the *specification of behavior*. Specification need not be perfect (i.e., it may provide an approximation of a system rather than a perfect view, be it at some level of abstraction). Specification has no a priori place in some artifact construction life cycle, just as quantification or qualification.

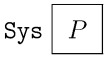
A process expression, e.g. $r_1(a)(s_2(\mathsf{true}) \cdot r_1(b) + s_3(\mathsf{false}) \cdot \delta)$ provides a text that represents a process (that is, a specification of behavior). In a similar way, a program expression

$$f_1.a; (+f_2.b; \#4, -f_3.c)^\omega$$

represents a SPI. However, there is a crucial difference: suppose process expression P denotes a specification of a system Sys, say $P = \mathtt{Sys}$, or at least, P is a reasonable approximation of Sys. Now it is *not* plausible to expect that

P (in any form) constitutes a part in any SynArch for Sys. On the other hand, if Sys is a system executing some program p, then it is plausible that a SynArch of Sys contains, perhaps in a transformed (compiled) form, p as a part.

Process expressions occur as parts of systems that analyze or simulate other systems. The following AnArch is perfectly acceptable:



Sys contains process expression P and behaves as P, thus Sys is a P-simulator. As a SynArch this makes little sense. Moreover, simulation is only one of many objectives supported by processes. Calculation and verification is another and probably more important one.

4.2. Compositional process specification for analytic architectures

We introduce an ACP-based process semantics for AnArchs over action alphabet

$$A = \{f.a \mid f \in F,\ a \in B\}$$

where F is a set of channels and B a set of co-actions. For this process semantics we will employ communication between send actions $s_f(a)$ and receive actions $r_f(a)$:

$$r_f(a)|s_f(a) = c_f(a) \quad \text{for } a \in B \cup \{\mathsf{true}, \mathsf{false}\}.$$

Furthermore, we use an action t to model the difference between thread termination and deadlock. So we assume that our set of process algebra actions includes

$$\big\{t, r_f(a), s_f(a), c_f(a) \mid f \in F,\ a \in B \cup \{\mathsf{true}, \mathsf{false}\}\big\}.$$

We use the notation $[\![P]\!]$ for process semantics of a thread P extracted from a SPI container:

$$[\![\mathsf{S}]\!] = t,$$
$$[\![\mathsf{D}]\!] = t^*\delta,$$
$$[\![P \trianglelefteq f.a \trianglerighteq Q]\!] = s_f(a)(r_f(\mathsf{true})[\![P]\!] + r_f(\mathsf{false})[\![Q]\!]).$$

Here x^*y is defined by $x^*y = x(x^*y) + y$ (see [5] or Appendix A). Taking δ for y and using the ACP-axiom $x + \delta = x$, it follows that $t^*\delta$ behaves as t^∞, i.e., an infinite sequence of t-actions.

We write $[\![R]\!]$ for the process algebraic semantics of a reactor R. We require that $[\![R]\!]$ uses the following actions for communication with a SPI container:

$$r_{serv}(a), \quad s_{serv}(\mathsf{true}), \quad s_{serv}(\mathsf{false}).$$

Furthermore, we require that any reactor R has the property that $r_{serv}(a)$ and $s_{serv}(b)$ ($b \in \{\mathsf{true}, \mathsf{false}\}$) occur in alternating order in each trace of $[\![R]\!]$. When connected to a SPI container via channel f, the renaming operator $\rho_{serv \mapsto f}$ (which renames the channel name *serv* to f) will be applied to $[\![R]\!]$.

Definition 6. Given a PGA-program X and reactors $R_0, \ldots, R_n$, a *concrete analytical architecture*, notation cpgaEA, is defined by

$$\text{cpgaEA}[X \mid f_0{:}R_0, f_1{:}R_1, .., f_n{:}R_n] = \partial_H([\![|X|]\!] \parallel \rho_{serv \mapsto f_0}([\![R_0]\!]) \parallel \rho_{serv \mapsto f_1}([\![R_1]\!]) \parallel .. \parallel \rho_{serv \mapsto f_n}([\![R_n]\!]))$$

with encapsulation set $H = \{r_i(a), s_i(a) \mid a \in B \cup \{\mathsf{true}, \mathsf{false}\},\ i = f_0, f_1, \ldots, f_n\}$.

An (*abstract*) *analytical architecture* pgaEA is defined by

$$\text{pgaEA}[X \mid f_0{:}R_0, f_1{:}R_1, .., f_n{:}R_n] = \tau_I(\text{cpgaEA}[X \mid f_0{:}R_0, f_1{:}R_1, \ldots, f_n{:}R_n])$$

with abstraction set $I = \{t, c_i(a) \mid a \in B \cup \{\mathsf{true}, \mathsf{false}\},\ i = f_0, f_1, \ldots, f_n\}$.

If the value of n is known or immaterial, we often write (c)pgaEA$[X \mid \overline{f_i{:}R_i}]$.

Notice that in common process semantics, $\tau^*\delta = \tau\delta$ (cf. [12]). We observe that cpgaEA$[X \mid f_0{:}R_0, f_1{:}R_1, \ldots, f_n{:}R_n]$ and its pgaEA-variant are computable if $|X|$ and all R_j are.

A reactor R is a state machine if $[\![R]\!]$ has only actions $r_{serv}(a)$, $s_{serv}(\mathsf{true})$ and $s_{serv}(\mathsf{false})$, i.e., no external events, only update of its memory state and computation of Boolean output. The results in [13] imply the following theorem.

Theorem 7. *Let $\overline{R} = R_0, R_1, \ldots, R_k, R_{k+1}, \ldots, R_n$ with $R_{k+1}, \ldots, R_n$ state machines. Furthermore, let*

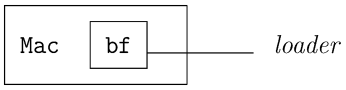
$$\alpha = |X|/_{f_{k+1}} R_{k+1} \ldots /_{f_n} R_n.$$

Then for $H = \{r_i(a), s_i(a) \mid a \in B \cup \{\mathsf{true}, \mathsf{false}\},\ i = f_0, f_1, \ldots, f_k\}$ and $I = \{t, c_i(a) \mid a \in B \cup \{\mathsf{true}, \mathsf{false}\}, i = f_0, f_1, \ldots, f_k\}$,

$$\text{pgaEA}[X \mid \overline{f_i{:}R_i}] = \tau_I(\partial_H([\![\alpha]\!] \between \rho_{serv\mapsto f_0}([\![R_0]\!]) \between \rho_{serv\mapsto f_1}([\![R_1]\!]) \between .. \between \rho_{serv\mapsto f_k}([\![R_k]\!]))).$$

4.3. Synthetic architectures for binaries

In this section we try to clarify the role of programs in machines. A binary is just a finite $\{0, 1\}$-sequence (i.e., a binary file) with an end-of-file marker `eof`. Consider the following SynArch:



This SynArch displays a machine `Mac` containing a binary file `bf`. It has a special port named *loader* used to enter `bf` in `Mac` bitwise.

Assuming that `Mac` is a classical piece of computing machinery, the process $[\![\texttt{Mac(bf)}]\!]$ specifying the behavior of `bf` in `Mac` will be a computable process. With ϵ denoting the empty bit-sequence, $[\![\texttt{Mac}]\!]$ can be specified in process algebra as follows:

$$\begin{aligned}
&[\![\texttt{Mac}]\!] = M_{loading}(\epsilon),\\
&M_{loading}(w) = r_{loader}(0) \cdot M_{loading}(w0)\\
&\qquad\qquad + r_{loader}(1) \cdot M_{loading}(w1)\\
&\qquad\qquad + r_{loader}(\texttt{eof}) \cdot [\![\texttt{Mac}(w)]\!].
\end{aligned}$$

It is reasonable to expect that $[\![\texttt{Mac(bf)}]\!]$ depends uniformly on `bf` (in the sense of computability theory, see e.g., [20]). Then, also $[\![\texttt{Mac}]\!]$ itself is a computable process, implying the following result.

Theorem 8. *The process $[\![\texttt{Mac}]\!]$ can be denoted modulo branching bisimulation equivalence in ACP extended with abstraction, binary Kleene star and pushdown operator, and this can be done using finitely many actions.*

This follows from a general expressiveness result, proven in detail in [11, Theorem 4.2.5].

Note 1. Let $\texttt{bf} = \texttt{bf}_0\texttt{bf}_1 \ldots \texttt{bf}_n\texttt{eof}$. For appropriate encapsulation set H and abstraction set I, we find

$$\tau \cdot [\![\texttt{Mac(bf)}]\!] = \tau_I \circ \partial_H(L(\texttt{bf}) \between [\![\texttt{Mac}]\!])$$

where $L(\texttt{bf}) = s_{loader}(\texttt{bf}_0) \cdots s_{loader}(\texttt{bf}_n) \cdot s_{loader}(\texttt{eof})$.

Let `pga2bin4m` be a mapping from PGA to bit sequences (`pga` *to* `bin4m`, where `bin4m` abbreviates "binaries for machine"). Then `pga2bin4m` is a *code generator mapping* if the following holds for all $X \in$ PGA:

$$\text{pgaEA}[X \mid \overline{f_i{:}R_i}] = [\![\texttt{Mac(pga2bin4m}(X))]\!].$$

That is: the analytic architecture pgaEA with its set of reactors *explains* (i.e., corresponds to) the synthetic architecture `Mac`. In practice one is happy if this works for all X with $\texttt{pga2bin4m}(X)$ having a size of less than k Mb (for some k). In this setting, the following jargon is useful:

(1) *Middle code* or *intermediate code*: a PGA-program as above.
(2) A machine (program) producing $\mathtt{pga2bin4m}(X)$ from X is a *code generator* or *compiler back end* for $\mathtt{Mac}$.
(3) The concept of a *machine code* can not be defined here: clearly, some $\mathtt{bf}$ are more useful than other $\mathtt{bf}$'s. But there is no obvious criterion regarding $[\![\mathtt{Mac(bf)}]\!]$ to select the binaries for $\mathtt{Mac}$ from arbitrary bit sequences.
(4) A higher program notation, say PGLZ, can be understood if a behavior preserving mapping

$$\mathtt{pglz2pga}$$

to PGA is known (such a mapping is called a projection) and a pgaEA such that

$$\mathrm{pgaEA}[\mathtt{pglz2pga}(X) \mid \overline{f_i{:}R_i}]$$

corresponds to the intended meaning of program X. A *compiler* is a system (or a program for a system) that allows to compute $\mathtt{pglz2pga}$ (or an optimized version of it that produces semantically equivalent behavior).

For a PGLZ-program X we then find

$$\mathrm{pgaEA}[\mathtt{pglz2pga}(X) \mid \overline{f_i{:}R_i}] = [\![\mathtt{Mac(pga2bin4m(pglz2pga}(X)))]\!],$$

and it is common practice to call $\mathtt{pga2bin4m(pglz2pga}(X))$ a *program*. This is one of the possible justifications for the qualification of a binary that is part of a SynArch as a program. We fix the nature of this qualification as follows:

Code generator mapping criterion: a binary $\mathtt{bf}$ is a program if it is in the range of a code generator mapping (in a setting that explains the behavior of $\mathtt{Mac(bf)}$ via an AnArch).

The qualification of $\mathtt{bf}$ as a program by this criterion seems to be at odds with the basis of PGA because PGA starts from the assumption that *a program is a sequence of instructions* (see [8]). However, if $\mathtt{pga2bin4m}$ is computable, it has a semi-computable inverse, say

$$\mathtt{bin4m2pga}$$

and $\mathtt{bf}$ qualifies as a program because $\mathtt{bin4m2pga(bf)}$ does. Of course, it is immaterial that $\mathtt{bin4m2pga}$ is *taken* to be an inverse of $\mathtt{pga2bin4m}$. What matters is: for all $\mathtt{bf}$ (or as many as one cares),

$$[\![\mathtt{Mac(bf)}]\!] = \mathrm{pgaEA}[\mathtt{bin4m2pga(bf)} \mid \overline{f_i{:}R_i}]$$
$$\big(= [\![\mathtt{Mac(pga2bin4m(bin4m2pga(bf)))}]\!]\big).$$

Thus, the code generator mapping criterion is consistent with the PGA-criterion for being a program.

Note 2. 1. Having a far more detailed SynArch at hand with $\mathtt{bf}$ as a part, one may find other justifications for qualifying $\mathtt{bf}$ as a program. However, we failed to develop such a story with any form of acceptable generality.

2. The projection $\mathtt{bin4m2pga}$ may be called a disassembler-projection (ignoring the complexity of loading). Then, if the qualification of $\mathtt{bf}$ as a program in $\mathtt{Mac(bf)}$ is justified by means of the code generator mapping criterion, a disassembler-projection semantics of $\mathtt{bf}$ is (implicitly) known/given.

3. The justification of the qualification of $\mathtt{bf}$ as a part of the SynArch $\mathtt{Mac(bf)}$ is an argument of a certain form: *qualification on the basis of a most plausible history*. If we see an object when it is a corpse, of course we see it if it was a living individual of some species that subsequently died. How else could the object have come into existence? If we see $\mathtt{bf}$ in $\mathtt{Mac}$ where $\mathtt{bf} = \mathtt{pga2bin4m}(X)$, that must be related to $\mathtt{bf}$'s history. How else would it have originated? I.e., $\mathtt{bf}$ is just another form or phase of X, like a corpse being another phase of a living body.

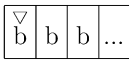
4. The middle code exists at the instruction sequence level in PGA. It is at the same time *target code for projection semantics*. Given a SynArch $\mathtt{Mac}(\ldots)$, its binaries are also called *object code*.

5. An analytic architecture for Turing machines

In this section we consider an enhanced version of the Turing machine tape and a PGA-based language for programming it. We prove the unsolvability of the halting problem, and show that this problem becomes decidable if we restrict our language sufficiently.

5.1. The Turing machine

The original reference to the Turing machine is [21]. A Turing machine M consists of a finite control and a tape, often visualized in the following style:



where b stands for "blank" (i.e., a blank square), and a head that can be used for reading and writing on that tape, depicted as ▽. Usually, the tape has a left end, and extends indefinitely to the right. The head can never fall off the tape (at the left side). The control is such that it distinguishes a halting state, after which control is terminated and the tape can be inspected for possible output. In a non-halting state, control prescribes some action to be undertaken and the next control state. Actions are either *read* or *write* a symbol in the square (where write means: replace the symbol that was already there, and write a blank means: *erase*), or *move* the head one tape square to the left (if possible) or right.

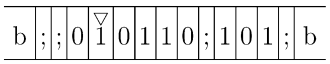
The *Church–Turing thesis* is the following principle (formulation taken from [19, p. 246]):

> *The Turing machine that halts on all inputs is a precise formal notion that corresponds to the naive notion of an "algorithm".*

Finally, the *halting problem* HP is the question whether or not a Turing machine M halts on input string w.

5.2. Enhanced Turing machine tape

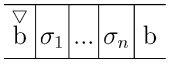
A Turing machine tape is seen as a state machine TMT. We consider an enhanced type of Turing machine tape over alphabet $\{0, 1, ;\}$ called ETMT to allow for more powerful programming. A state of the ETMT is for example



where the b stands for *blank*, the semi-colon serves as a separator, and the ▽ is the head. The left b represents an indefinite number of blanks to the left,[2] and the rightmost b signifies that the tape indefinitely extends to the right.

The empty tape (containing only blanks), thus

is denoted by ETMT($\hat{\text{b}}$ b) or simply by ETMT. The configuration of ETMT that contains sequence $\sigma = \sigma_1 \dots \sigma_n$ with the head at the left blank, i.e.,



is denoted by ETMT($\hat{\text{b}}\,\sigma$ b). In this ETMT-notation, the two occurring b's will be referred to as the *left blank* and the *right blank*, respectively.

We consider the following service-instructions for controlling the ETMT, where a *bit sequence* on the tape is a sequence of 0 or 1 occurrences of maximal length (so at both ends neighboring either a semicolon or a blank).

test:0	*write*:0	*mv*:*left*	*mv*:*begin*
test:1	*write*:1	*mv*:*right*	*dup*
test:*semicolon*	*write*:*semicolon*		
test:*b*	*write*:*b*		

These instructions have the following effect:

test:0 (or 1, *semicolon*, b) checks whether the head points to a 0 (or the other symbol indicated) and returns true if this is the case, and false otherwise.

[2] This does not increase the computational power of a Turing machine (see e.g., [19]).

write:0 (or 1, *semicolon*) writes the mentioned symbol at head position and returns true.

write:b writes a blank at head position if to the left or the right of the head there is a blank already and returns true, otherwise nothing changes and false is returned.

mv:*left* fails if the head points at a blank and to the left there is a blank as well, in this case it returns false and nothing happens; otherwise the head moves one square to the left and returns true.

mv:*right* works similar.

mv:*begin* places the head at the left blank if it is not already there, and returns true.

dup duplicates the leftmost bit sequence if any exists, and puts the result next to it, separated by a semicolon. Furthermore, the head moves to the left blank. Returns true if actual duplication has taken place, and false otherwise. Some examples:

$$\mathrm{ETMT}(\mathrm{b};;\hat{\mathrm{b}}) \xrightarrow{dup} \mathrm{ETMT}(\hat{\mathrm{b}};;\mathrm{b}) \quad \text{(returns false)},$$

$$\mathrm{ETMT}(\mathrm{b};;0\hat{\mathrm{b}}) \xrightarrow{dup} \mathrm{ETMT}(\hat{\mathrm{b}};;0;0\,\mathrm{b}) \quad \text{(returns true)},$$

$$\mathrm{ETMT}(\mathrm{b}\,01;1\hat{0}1;\mathrm{b}) \xrightarrow{dup} \mathrm{ETMT}(\hat{\mathrm{b}}\,01;01;101;\mathrm{b}) \quad \text{(returns true)}.$$

5.3. Programming the Turing machine

For programming the Turing machine we shall use the programming language PGLC [8], a language based on PGA. The only construct (operator) in PGLC is concatenation. Instead of the repetition operator $(\_)^\omega$ of PGA, PGLC contains backwards jumps $\backslash\#k$ for any $k \in \mathbb{N}$. For example, $+a;\#0;\backslash\#2$ is a PGLC-program that behaves as $(+a;\#0)^\omega$. Furthermore, termination is modelled implicitly in PGLC: a program terminates after a jump outside the program, or after the last instruction has been executed and that instruction was no jump into the program; the termination instruction ! is not present in PGLC.

Let p be some PGLC-program. We write $|p|_{pglc}$ for the thread defined by p. Thread extraction is defined using a projection function `pglc2pga` from PGLC-programs to PGA-terms:

$$\texttt{pglc2pga}(u_1;\ldots;u_k) = (\psi_1(u_1);\ldots;\psi_k(u_k);!;!)^\omega,$$

with the auxiliary operators ψ_j defined by

$$\psi_j(\#l) = \begin{cases} \#l & \text{if } j+l \leqslant k \\ ! & \text{otherwise,} \end{cases}$$

$$\psi_j(\backslash\#l) = \begin{cases} \#k+2-l & \text{if } 0 < l < j \\ \#0 & \text{if } l = 0 \\ ! & \text{otherwise,} \end{cases}$$

$$\psi_j(u) = u \quad \text{otherwise,}$$

and $|p|_{pglc} = |\texttt{pglc2pga}(p)|$.

For example, $|+a;\#2;\backslash\#2;+b|_{pglc} = |(+a;\#2;\#4;+b;!;!)^\omega| = P$ with

$$P = b \circ \mathsf{S} \trianglelefteq a \trianglerighteq P.$$

As basic instructions for PGLC we use the service-instructions defined for the ETMT in the previous section prefixed with the focus *etmt*. An example of a basic instruction is

etmt.*test*:0.

We consider execution of Turing machine programs in an AnArch using ETMT as a state machine. To enhance readability, we use an AnArch that takes as its first argument a PGLC-program (instead of a sequence of primitive PGA-instructions), notation

pglcEA[p | *etmt*:ETMT].

This AnArch is defined with help of the projection function `pglc2pga`:

pglcEA[p | *etmt*:ETMT] = pgaEA[$\texttt{pglc2pga}(p)$ | *etmt*:ETMT].

An example:

$$\begin{aligned}&\mathrm{pglcEA}[\mathit{etmt.dup};\ \mathit{etmt.mv}{:}\mathit{right} \mid \mathit{etmt}{:}\mathrm{ETMT}(\mathrm{b}\,01;\ \hat{1}01;\ \mathrm{b})]\\ &\quad\downarrow \tau\\ &\mathrm{pglcEA}[\mathit{etmt.mv}{:}\mathit{right} \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,01;\ 01;\ 101;\ \mathrm{b})]\\ &\quad\downarrow \tau\\ &\quad\surd \text{ with ETMT's configuration: } \mathrm{ETMT}(\mathrm{b}\,\hat{0}1;\ 01;\ 101;\ \mathrm{b}).\end{aligned}$$

Here each of the τ-steps ($\xrightarrow{\tau}$) comes from two (abstracted) communications triggered by the current program-fragment and the ETMT, and the symbol $\surd$ represents termination.

5.4. The Halting Problem

The *Halting Problem* (HP) can in this setting be modelled as follows: a PGLC-program p halts on the ETMT with initial configuration $\hat{\mathrm{b}}\,w\,\mathrm{b}$ (w a bit sequence), notation $(p, w) \in \mathrm{HP}$, if

$$\mathrm{pglcEA}[p \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,w\,\mathrm{b})] = \tau,$$

as opposed to $\tau^*\delta$ ($= \tau\delta$). After halting the tape can be inspected to obtain an output.

A program in PGLC is an ASCII character sequence (see, e.g., [1]), and therefore a sequence of bits. As an example, the character a has 97 as its decimal code, which is as a byte (sequence of 8 bits) 01100001, and the character ";" has 59 as its decimal code, which is as a byte 00111011. Given a PGLC-program p, we write $\overline{p}$ for its representation as a bit sequence.

Definition 9. Program $q \in \mathrm{PGLC}$ *solves* the question whether $(p, w) \in \mathrm{HP}$ if:

$$\mathrm{pglcEA}[q \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,\overline{p};\ w\,\mathrm{b})]$$

always halts, and after halting, the tape configuration is of the form

$$\begin{aligned}&\mathrm{ETMT}(\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}) \quad \text{if } \mathrm{pglcEA}[p \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,w\,\mathrm{b})] \text{ halts, thus } (p, w) \in \mathrm{HP},\\ &\mathrm{ETMT}(\hat{\mathrm{b}}\,1\rho\,\mathrm{b}) \quad \text{if } \mathrm{pglcEA}[p \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,w\,\mathrm{b})] \text{ halts not, i.e. } (p, w) \notin \mathrm{HP},\end{aligned}$$

for some string σ or ρ.

Theorem 10. *The halting problem is unsolvable by means of any program in* PGLC.

Proof. Suppose the contrary, i.e., a program $q \in \mathrm{PGLC}$ exists that solves HP. Consider the following program:

$$s = \mathit{etmt.dup};\ q;\ r$$

with r the program that tests whether a 0 or a 1 is at the leftmost non-blank square, after which respectively deadlock or termination follows:

$$r = \mathit{etmt.mv}{:}\mathit{right};\ -\mathit{etmt.test}{:}1;\ \#0;\ \mathit{etmt.mv}{:}\mathit{begin}.$$

Assume without loss of generality that the program q satisfies the following properties:

- it does not end with a test instruction (note that $|p|_{pglc} = |p;\ \#1|_{pglc}$),
- each forward jump in q does not exceed the number of subsequent instructions with more than 1,
- each backward jump in q does not exceed the number of preceding instructions.

As a consequence, execution of the program $q;\ r$ continues after q has been executed with the *first* instruction of r.

We show that both assumptions $(s, \bar{s}) \in \mathrm{HP}$ and $(s, \bar{s}) \notin \mathrm{HP}$ lead to a contradiction. Hence, s cannot exist, and thus q cannot exist.

pglcEA[r | *etmt*:ETMT($\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}$)]

$=$

pglcEA[*etmt.mv*:*right*; $-$*etmt.test*:1; #0; *etmt.mv*:*begin* | *etmt*:ETMT($\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}$)]

$\downarrow\tau$ (*etmt.mv*:*right*)

pglcEA[$-$*etmt.test*:1; #0; *etmt.mv*:*begin* | *etmt*:ETMT($\mathrm{b}\,\hat{0}\sigma\,\mathrm{b}$)]

$\downarrow\tau$ ($-$*etmt.test*:1)

pglcEA[#0; *etmt.mv*:*begin* | *etmt*:ETMT($\mathrm{b}\,\hat{0}\sigma\,\mathrm{b}$)].

Fig. 1. Last part of the behavior in the case that $(s,\bar{s}) \in$ HP in the proof of Theorem 10.

pglcEA[s | *etmt*:ETMT($\hat{\mathrm{b}}\,\bar{s}\,\mathrm{b}$)]

$\downarrow\tau$ (*etmt.dup*)

pglcEA[q; r | *etmt*:ETMT($\hat{\mathrm{b}}\,\bar{s}$; $\bar{s}\,\mathrm{b}$)]

$\downarrow\tau$ (by program q)

pglcEA[r | *etmt*:ETMT($\hat{\mathrm{b}}\,1\rho\,\mathrm{b}$)]

$=$

pglcEA[*etmt.mv*:*right*; $-$*etmt.test*:1; #0; *etmt.mv*:*begin* | *etmt*:ETMT($\hat{\mathrm{b}}\,1\rho\,\mathrm{b}$)]

$\downarrow\tau$ (*etmt.mv*:*right*)

pglcEA[$-$*etmt.test*:1; #0; *etmt.mv*:*begin* | *etmt*:ETMT($\mathrm{b}\,\hat{1}\rho\,\mathrm{b}$)]

$\downarrow\tau$ ($-$*etmt.test*:1)

pglcEA[*etmt.mv*:*begin* | *etmt*:ETMT($\mathrm{b}\,\hat{1}\rho\,\mathrm{b}$)]

$\downarrow\tau$ (*etmt.mv*:*begin*)

$\surd$ with ETMT's configuration: ETMT($\hat{\mathrm{b}}\,1\rho\,\mathrm{b}$).

Fig. 2. The case that $(s,\bar{s}) \notin$ HP in the proof of Theorem 10.

Assume that $(s,\bar{s}) \in$ HP. Then

pglcEA[s | *etmt*:ETMT($\hat{\mathrm{b}}\,\bar{s}\,\mathrm{b}$)]
$\downarrow\tau$ (*etmt.dup*)
pglcEA[q; r | *etmt*:ETMT($\hat{\mathrm{b}}\,\bar{s}$; $\bar{s}\,\mathrm{b}$)]
$\downarrow\tau$ (by program q)
pglcEA[r | *etmt*:ETMT($\hat{\mathrm{b}}\,0\sigma\,\mathrm{b}$)]

for some string σ. The remaining behavior is displayed in Fig. 1, and results in pglcEA[#0; *etmt.mv*:*begin* | *etmt*:ETMT($\mathrm{b}\,\hat{0}\sigma\,\mathrm{b}$)]. This last AnArch clearly represents deadlock because of the first instruction #0, and therefore $(s,\bar{s}) \notin$ HP. Contradiction.

Assume that $(s,\bar{s}) \notin$ HP. The behavior of pglcEA[s | *etmt*:ETMT($\hat{\mathrm{b}}\,\bar{s}\,\mathrm{b}$)] is displayed in Fig. 2. This behavior ends with termination, thus $(s,\bar{s}) \in$ HP. Contradiction.

So our supposition was definitely wrong and there is no program $q \in$ PGLC that solves the halting problem. □

It is an easy but boring task to program *mv*:*begin* and *dup* in terms of the other instructions, thus obtaining a stronger proof. As a theorem, the above one (Theorem 10) suffices. From that point of view there is nothing special about the (E)TMT or any of its versions. What we see is that:

(1) For a close relative of the TMT an impossibility result is obtained.
(2) Increasing the instruction set of the ETMT to a 'super' ETMT does not help. The proof goes exactly the same. Computability of these instructions is immaterial. What matters is that the halting problem (HP) is posed about all programs over the instruction set that may be used to program its solution.
(3) The Church–Turing thesis is not used because the result is phrased about PGLC programs, and not about 'algorithms' or 'computable methods'. Nevertheless, if it is considered convincing that an effective method can be performed by a certain Turing machine, then it is also obvious that it can be programmed in PGLC:
   - finite control can be modelled in the program;
   - additional instructions only strengthen the expressive power.

This situation changes if we restrict the set of basic instructions. In the proof above we used *etmt.dup*, *etmt.test*:1, *etmt.mv*:*right* and *etmt.mv*:*begin*. Let the language PGLC^- contain these as the *only* basic instructions. Note that *etmt.dup* is the only PGLC^--instruction that writes on the tape.

Theorem 11. *For the language* PGLC^-*, the halting problem is decidable.*

Proof. We show that halting can be decided for any program and initial tape configuration (the latter not necessarily a sequence of bits and the head pointing at the left blank).

So, let $\mathrm{pglcEA}[p \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,\sigma\,\mathrm{b})]$ be given with $p \in \mathrm{PGLC}^-$ and σ some string. If the tape contains no sequence of bits, each occurrence of *etmt.dup* in p can be replaced by *etmt.mv*:*begin* and the tape remains a fixed and finite structure (in the case that *etmt.dup* is used as a test instruction, the sign of *etmt.mv*:*begin* should be reversed). Execution now either yields a cyclic pattern, or stops at #0, or after the last instruction or a terminating jump. As there are finitely many combinations of current instruction and head position, halting is decidable.

In the other case the tape contains a sequence of bits. We write $\mathrm{ETMT}(\sigma, k)$ if the head points to the kth position of the sequence σ, and to the nearest blank if k is out of range. Transform p to a canonical form in PGA using the projection `pglc2pga` and the axioms in Table 1. If this canonical form contains no repetition we are done, otherwise we obtained a PGA-program $U; V^\omega$ with U and V containing no repetition. Halting on U is decidable: either one of the decisive instructions ! or #0 is to be executed. In the other case, execution enters the repeating part V^ω. So, we further consider $\mathrm{pgaEA}[W^\omega \mid \mathit{etmt}{:}\mathrm{ETMT}(\rho, k)]$ for some cyclic permutation W of V and tape configuration ρ with the head at position k. Now, either *etmt.dup* occurs at a reachable position in W^ω (i.e., occurs in $|W^\omega|$), or not. This is decidable. In the last case, the tape remains a fixed and finite structure, and iterating W yields a regular behavior, so halting is decidable. In the other case, the number of *etmt.mv*:*right*-instructions in W, say n, limits the number of positions that the head can shift to the right. Consider $\mathrm{pgaEA}[W^n; W^\omega \mid \mathit{etmt}{:}\mathrm{ETMT}(\rho', k')]$. Either halting can be decided on W^n, or the repeating part is entered, say X^ω (X a cyclic permutation of W). We may replace all *etmt.dup*-instructions in X^ω by *etmt.mv*:*begin* because these duplications concern an unreachable part at the right end of the tape. So, this case is reduced to the previous one, and halting is again decidable. □

Our objective is to position Turing's impossibility result regarding the assessment of halting properties of program execution as a result about programs rather than machines. The mere requirement that programs of a certain form can decide the halting behavior of all programs of that form leads to a contradiction. This contradiction can be found in the case of programs for a Turing machine tape (TMT). The argument is significantly simplified if an extended command set for a Turing machine tape is used (ETMT). But then the program notation may be reduced to those features (instructions) actually playing a role in the argument and the impossibility result remains but now in a setting where the underlying halting problem is in fact decidable.

Note 3. In the case of the language PGLC^-, one should take care in what way a program q solves the halting problem (cf. Definition 9). A leftmost 0 or 1 on the tape upon q's termination is not appropriate because apart from *etmt.dup*, the language contains no write-instructions. For PGLC^-, a possible criterion is the head position after termination of q:

$$\mathrm{pglcEA}[q \mid \mathit{etmt}{:}\mathrm{ETMT}(\hat{\mathrm{b}}\,\overline{p}; w\,\mathrm{b})]$$

always terminates, and the head is at the left blank if $(p, w) \in \mathrm{HP}$ and at the rightmost blank otherwise. This is consistent because in PGLC^- there is no means to remove bits from the tape and the initial configuration contains two non-empty bit sequences, so left and right blank can always be distinguished. As in the proof of Theorem 10, this yields the impossibility result for the question whether $(s, \bar{s}) \in \mathrm{HP}$ for $s = \mathit{etmt.dup}; q; +\mathit{etmt.mv}{:}\mathit{right}; \#0$.

We conclude that as a methodological fact about computer programming, the undecidability of the halting problem is an impossibility result which is quite independent of the computational power of the machine models to which it can be applied.

6. Forecasting reactors and rational agents

The halting problem can be investigated without the use of TMTs as a problem regarding the potential capabilities of a reactor serving as a computation forecasting device. In this section we show that restricting to true and false is problematic and introduce a third truth-value. Furthermore, we combine forecasters with 'rational agents', and provide a modeling of the Newcomb Paradox. Finally, we model the Prisoner's Dilemma as an analytic architecture.

6.1. Forecasting reactors

Forecasting is not an obvious concept, the idea that it is to be done by means of a machine even less. We will provide a 'clean' interpretation of forecasting and investigate its fate in the context of execution architectures for PGA. The use of an AnArch is justified because this story is meant at a conceptual level and is not part of any technical design.

In the previous section it was shown that restricting to true and false is problematic. Therefore we now consider the case that the evaluation of test instructions may yield not only true or false, but also the value M (*meaningless*):

$$|{+}a; X| = \begin{cases} a \circ |\#1; X| & \text{if } a\text{'s execution returns true,} \\ a \circ |\#2; X| & \text{if } a\text{'s execution returns false,} \\ a \circ |\#3; X| & \text{if } a\text{'s execution returns M,} \end{cases}$$

and

$$|{-}a; X| = \begin{cases} a \circ |\#2; X| & \text{if } a\text{'s execution returns true,} \\ a \circ |\#1; X| & \text{if } a\text{'s execution returns false,} \\ a \circ |\#3; X| & \text{if } a\text{'s execution returns M.} \end{cases}$$

More information on many-valued logics using true, false and M can be found in [6,10].

We will use *fcu* as the focus pointing to a forecasting unit FCU in the following way: $\mathit{fcu}.Q$ will ask the forecaster to reply about its opinion regarding the question Q. At this point the precise phrasing of the requirement on the FCU is essential. In $\mathrm{pgaEA}[\mathit{fcu}.Q; X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}]$ one expects FCU to reply true if Q is valid as an assertion on $\mathrm{pgaEA}[X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}]$, notation

$$\mathrm{pgaEA}[X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}] \ \mathbf{sat}\ Q.$$

More precisely, in

$$\mathrm{pgaEA}[{+}\mathit{fcu}.Q; u; X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}]$$

we expect that

- true is returned if $\mathrm{pgaEA}[u; X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}]\ \mathbf{sat}\ Q$,
- false is returned if $\mathrm{pgaEA}[X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}] \not\mathbf{sat}\ Q$,
- M is returned otherwise.

Moreover, in case that both true and false could be returned, preference is given to returning true.

Consider $Q = \mathit{halting}$: when $\mathrm{pgaEA}[X \mid \mathit{fcu}{:}\mathrm{FCU}, \overline{f_i{:}R_i}]\ \mathbf{sat}\ \mathit{halting}$ it will hold along all execution traces (irrespective of any context) that X halts. Then, if a reactor can engage in external communications, the possibility that it will must be taken into account. Moreover, we cannot exclude the possibility that a reactor falls into inaction as a result

of such an external communication. Therefore we assume the absence of reactors apart from FCU, and investigate to what extent FCU can be made to provide a useful forecast regarding the *halting*-question.

Theorem 12. *A forecasting reactor* FCU *needs the third truth value* M.

Proof. Consider

$$\mathrm{pgaEA}[+fcu.halting; \#0; !; ! \mid fcu{:}\mathrm{FCU}].$$

If true is replied then pgaEA[#0; !; ! | *fcu*:FCU] **sat** *halting* which is not true; if false is replied then pgaEA[!; !|*fcu*:FCU] **$\not$sat** *halting* which is also not true. Thus M should be replied, and

$$\mathrm{pgaEA}[+fcu.halting; \#0; !; ! \mid fcu{:}\mathrm{FCU}] \xrightarrow{\tau} \mathrm{pgaEA}[! \mid fcu{:}\mathrm{FCU}]$$

(which will halt by the way). □

We notice that the reactor FCU may use whatever inspection of other parts of an AnArch. However, it cannot correctly forecast the question with either true or false. Nevertheless, a lot is possible, e.g.:

$$\mathrm{pgaEA}[+fcu.halting; !; \#0 \mid fcu{:}\mathrm{FCU}]$$

generates reply true, and

$$\mathrm{pgaEA}[+fcu.halting; \#0; \#0 \mid fcu{:}\mathrm{FCU}]$$

generates reply false.

Theorem 13. *A best possible* FCU *for halting can be given for* PGA *with fcu.halting as its only basic instruction.*

Proof. Let X^{true} be obtained from X by replacing each occurrence of *fcu.halting* by *fcu*.true, the test that always yields true, and let X^{false} be defined similarly (replacement by *fcu*.false). Consider

$$\mathrm{pgaEA}[+fcu.halting; u; X \mid fcu{:}\mathrm{FCU}].$$

(1) In the case that $\mathrm{pgaEA}[u^{\mathsf{true}}; X^{\mathsf{true}} \mid fcu{:}\mathrm{FCU}]$ **sat** *halting*, we may define that $+fcu.halting$ returns true, and as a consequence

$$\mathrm{pgaEA}[+fcu.halting; u; X \mid fcu{:}\mathrm{FCU}] \textbf{ sat } halting.$$

(2) In the case that $\mathrm{pgaEA}[X^{\mathsf{false}} \mid fcu{:}\mathrm{FCU}]$ **$\not$sat** *halting*, we may define that $+fcu.halting$ returns false, and as a consequence

$$\mathrm{pgaEA}[+fcu.halting; u; X \mid fcu{:}\mathrm{FCU}] \textbf{ $\not$sat } halting.$$

(3) If none of the cases above applies, $+fcu.halting$ generates reply M.

E.g., pgaEA[$+fcu.halting$; #0; !; ! | *fcu*:FCU] will return M although it is going to be halting: by returning M it moves to an instruction from where halting is guaranteed indeed, while replying false would not produce a consistent answer.

It is easy to see that if this definition of replies given by FCU returns M, it cannot be replaced by either true or false. Hence, FCU is optimal. □

So, a halting forecaster can be built, but it cannot always provide a useful reply. On PGA one can decide whether a useful reply can be given. Given the fact that all practical computing takes place on finite state machines for which PGA is of course sufficient, we conclude this:

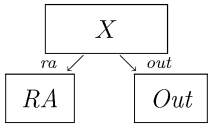
(1) All practical instances of *halting* are decidable, given a $\mathrm{pgaEA}[X \mid \overline{f_i{:}R_i}]$ with all R_i finite state.
(2) Nevertheless, a halting forecaster cannot work in a flawless fashion, although it can be 'optimal' (i.e., minimizing output M).

If all R_i's are finite state machines in pgaEA$[X \mid \overline{f_i{:}R_i}]$ **sat** *halting* (which is always the case 'in practice'), we find that for a particular AnArch fixing the R_i's the halting problem will be decidable, especially if the AnArch is tailored to fit the constraints of some realistic SynArch.

Of course, one can investigate forecasting *reactors*. Then the question is: what impossibility will one encounter? The obvious undecidability result critically depends on one of the reactors being infinite state or engaging in external communications. We return to forecasting reactors in Section 6.3.

6.2. Reactors formalizing rational agents

We consider a 'rational agent' *RA* with focus *ra*. The rational agent intends to achieve an objective and acts accordingly. Here is a simple example:

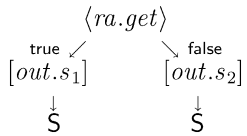


where *Out* has five states 0, 1, 2, 3, 4 and initially is in state 0. There are four instructions $s_1, \ldots, s_4$ which all succeed in each state of *Out*, with s_i leaving *Out* in state i for $i \in \{1, 2, 3, 4\}$.

The PGA-program X is as follows:

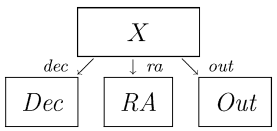
$$X = +ra.get; \#3; out.s_2 :!; out.s_1; !.$$

The thread $|X|$ can be visualized as follows:



The task of *RA* is to make the system terminate with a maximal content of *Out*. *RA* is aware of the contents of program X. In this case, it is clear that the reply false is optimal.

For a second example we add a decision agent *Dec* such that *RA* cannot know which decision *Dec* takes. The focus for *Dec* is *dec*. The instruction *dec.set* asks *Dec* to take a decision, which it will keep forever, and allows inspection via *dec.get*. An inspection not preceded by a *dec.set* returns M.



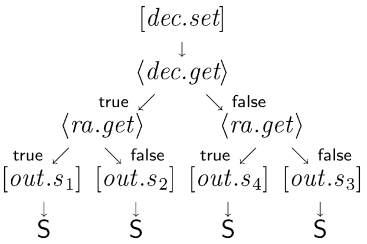
The model for *Dec* in concrete process algebra is:

$$\begin{aligned}
&[\![Dec]\!] = r(set)(t \cdot [\![Dec^{\mathsf{true}}]\!] + t \cdot [\![Dec^{\mathsf{false}}]\!]),\\
&[\![Dec^{\mathsf{true}}]\!] = (s(\mathsf{true}) \cdot r(get))^*\delta,\\
&[\![Dec^{\mathsf{false}}]\!] = (s(\mathsf{false}) \cdot r(get))^*\delta.
\end{aligned}$$

We consider the following PGA-program X':

$$\begin{aligned}
X' = {} & dec.set; +dec.get; \#2; \#7;\\
& +ra.get; \#3; out.s_2; !; out.s_1; !;\\
& +ra.get; \#3; out.s_3; !; out.s_4; !
\end{aligned}$$

or as a thread:



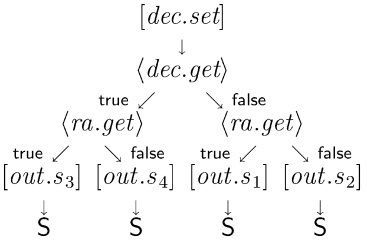
Now both replies true, false of *RA* are not optimal. If *RA* replies true, this leads to 1 after a positive decision of *Dec* (and false would have given 2), while false is not optimal after a negative decision of *Dec* (giving 3 rather than 4). Therefore it is plausible to return M, but that yields no maximum either (it leaves *Out* in state 0).

6.3. A Newcomb Paradox system

In this section we consider the following program, a small modification of the last program in the previous section (all $out.s_i$-instructions switched places):

$$\begin{aligned} X = {} & dec.set; +dec.get; \#2; \#7; \\ & +ra.get; \#3; out.s_4; !; out.s_3; !; \\ & +ra.get; \#3; out.s_2; !; out.s_1; ! \end{aligned}$$

with behavior



Now, quite independently of *Dec*'s action, it is plausible that *RA* replies false as its best reply. This answer is very robust and covers all possible/conceivable ways for which *RA* might work.

For the next example we introduce the property of pgaEA's that a reactor may be a *forecaster* of another one:

$$\mathrm{pgaEA}_{forecast:f>g}$$

is as pgaEA but with the additional constraint that the reactor focused by f forecasts the reactor focused by g. I.e., if $f.get$ returns true (false) then the next $g.get$ will reply true (false) as well.

Consider

$$\mathrm{pgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out].$$

If *RA* chooses to reply true, *Dec* must have replied true, yielding *Out* in state 3, and if *RA* replies false, *Dec* must have replied false and the yield is 2.

This is a version of Newcomb's Paradox, the original form of which has been made famous by Gardner [17, Chapters 13, 14].[3] The additional assumption of forecasting reverses the rational answer becoming true instead of false.[4] But the argument for false was completely compelling, making use of case-based reasoning regarding uncertainty

[3] A short description based on this source: Given two boxes, B1 which contains $1000 and B2 which contains either nothing or a million dollars, you may pick either B2 or both. However, at some time before the choice is made, an omniscient Being has predicted what your decision will be and filled B2 with a million dollars if he expects you to take it, or with nothing if he expects you to take both.

[4] I.e. (in terms of the previous footnote), pick only box B2 instead of both.

Table 3
Axioms for $F_{a>b>c}$, where a, b, c, e are actions

$F_{a>b>c}(e) = e,$	$F_{a>b>c}(x+y) = F_{a>b>c}(x) + F_{a>b>c}(y),$
$F'_{a>b>c}(e) = \begin{cases} 0 & \text{if } e = b, \\ e & \text{otherwise,} \end{cases}$	$F'_{a>b>c}(x+y) = F'_{a>b>c}(x) + F'_{a>b>c}(y),$
$F_{a>b>c}(\delta) = \delta,$	$F_{a>b>c}(e\cdot x) = \begin{cases} e\cdot F'_{a>b>c}(x) & \text{if } e = a, \\ e\cdot F_{a>b>c}(x) & \text{otherwise,} \end{cases}$
$F'_{a>b>c}(\delta) = \delta,$	$F'_{a>b>c}(e\cdot x) = \begin{cases} 0 & \text{if } e = b, \\ e\cdot x & \text{if } e = c, \\ e\cdot F'_{a>b>c}(x) & \text{otherwise.} \end{cases}$

about past events. The Newcomb Paradox then arises from the apparently illegal identification of the two following execution architectures:

$$\mathrm{pgaEA}[X \mid dec{:}Dec, ra{:}RA, out{:}Out] \quad \text{and}$$
$$\mathrm{pgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out].$$

In particular, the mistaken view that the second architecture somehow refines the first one by merely providing additional information leads to a contradiction. Thus: adding more information about the component *Dec*, the plausibility of *RA* giving the reply false in order to maximize the contents of *Out* at program termination is lost.

To formalize forecasting between reactors in process algebra, we use the constant 0 (see [3]):

$$x + 0 = x,$$
$$x \cdot 0 = 0 \quad \text{provided } x \neq \delta,$$
$$0 \cdot x = 0.$$

We write $F_{a>b>c}(X)$ for the following modification of process X: from the first a onwards, replace all b's by 0 until the first occurrence of c. The operator $F_{a>b>c}$ is axiomatized in Table 3. The auxiliary operator $F'_{a>b>c}$ models the situation in which the first action a is passed.

Forecasting can be formalized as follows:

$$\mathrm{cpgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out] =$$
$$F_{c_{dec}(\mathsf{true})>c_{ra}(\mathsf{false})>c_{dec}(\mathsf{true})}($$
$$F_{c_{dec}(\mathsf{false})>c_{ra}(\mathsf{true})>c_{dec}(\mathsf{false})}(\mathrm{cpgaEA}[X \mid dec{:}Dec, ra{:}RA, out{:}Out]))$$
$$\mathrm{pgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out] =$$
$$\tau_I(\mathrm{cpgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out]) \text{ for appropriate } I.$$

Some computation shows that indeed

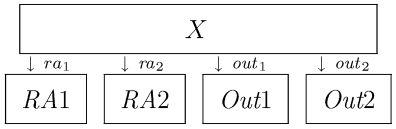
$$\mathrm{pgaEA}_{forecast:dec>ra}[X \mid dec{:}Dec, ra{:}RA, out{:}Out] = \tau(\tau\cdot out.s_3 + \tau\cdot out.s_2),$$

while $\mathrm{pgaEA}[X \mid dec{:}Dec, ra{:}RA, out{:}Out] = \tau(\tau\cdot out.s_4 + \tau\cdot out.s_2)$ if *RA* chooses false as its best reply.

6.4. Prisoner's Dilemma

A close relative of the above examples is the so-called Prisoner's Dilemma [18], which has become very well-known in game theory, psychology and economics.

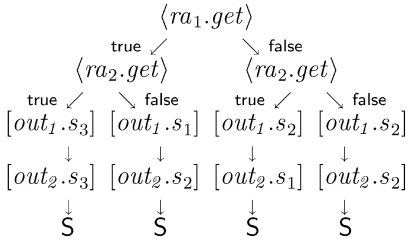
Consider the following situation:



Each rational agent has its own "out" and intends to maximize its own yield, irrespective of the other yield. We define the program X by

$$\begin{aligned} X = {} & +ra_1.get; \#2; \#9; \\ & +ra_2.get; \#4; out_1.s_1; out_2.s_2; !; out_1.s_3; out_2.s_3; !; \\ & +ra_2.get; \#4; out_1.s_2; out_2.s_2; !; out_1.s_2; out_2.s_1; ! \end{aligned}$$

so $|X|$ can be depicted by



This yields the following scenario's:

- if $RA1$ and $RA2$ both reply true, both yield the value 3,
- if $RA1$ and $RA2$ both reply false, each gets the value 2,
- if one replies true and the other false, the reply false gets 2 and the reply true yields 1.

As a consequence, in order to exclude the risk of yielding only 1 a sure strategy is to choose false. But in order to get 3, both RA's must trust one another and choose true, at the same time taking the risk to get only 1. Unable to communicate, the RA's may both go for certainty and reply false.

Note 4. The idea of forecasting does not apply to this example. Because of the particular way X is programmed, $RA1$ forecasting $RA2$, thus

$$\mathrm{pgaEA}_{forecast:ra1>ra2}[X \mid ra_1{:}RA1, ra_2{:}RA2, out_1{:}Out1, out_2{:}Out2],$$

excludes the last scenario, and hence denies the dilemma. Applying the reverse forecasting does not have any effect:

$$\begin{aligned} & \mathrm{pgaEA}_{forecast:ra2>ra1}[X \mid ra_1{:}RA1, ra_2{:}RA2, out_1{:}Out1, out_2{:}Out2] \\ & \quad = \mathrm{pgaEA}[X \mid ra_1{:}RA1, ra_2{:}RA2, out_1{:}Out1, out_2{:}Out2]. \end{aligned}$$

Note 5. A common application of the Prisoner's Dilemma is to assume that the reply true complies with the law and false opposes the law. If both comply with the law, both have significant advantage. Complying with the law while others don't is counterproductive, however.

Acknowledgement

We thank the referees for their valuable reviews of an earlier version of this paper.

Appendix A. ACP-based process algebra

In this appendix we shortly recall some process algebra. For more information and explanation we refer to [16].

The signature of ACP has a constant δ and constants for actions. Furthermore, ACP has binary operators $+$ (alternative composition), $\cdot$ (sequential composition), $\|$ (parallel composition, merge), $\lfloor\!\lfloor$ (left merge), and $|$ (communication merge). Finally, there is a unary renaming operator ∂_H (encapsulation) for every set H of actions, which renames the actions in H into δ. We use infix notation for all binary operators, and adopt the binding convention that $+$ binds weakest and $\cdot$ binds strongest. We suppress $\cdot$, writing xy for $x \cdot y$. Alternative and sequential composition are associative, and the first operator is also commutative and idempotent.

Parallel composition in ACP satisfies the law

$$x \parallel y = (x \between y + y \between x) + x \mid y,$$

where $\between$ is as $\parallel$ with the restriction that the first action must be one from the left argument, while $\mid$ has the restriction that the first action must be a communication. E.g., $a \between x = ax$ and $ax \between y = a(x \parallel y)$.

Encapsulation is a special case of renaming. The renaming operator ρ_f with $f : A \to A$ for action set A is defined by

$$\begin{aligned}
&\rho_f(a) = f(a) \quad (a \in A),\\
&\rho_f(\delta) = \delta,\\
&\rho_f(x + y) = \rho_f(x) + \rho_f(y),\\
&\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y).
\end{aligned}$$

(In the case of encapsulation, the range of f includes δ.)

Communication in ACP is predefined on the set of actions. For example, $a|b = c$ implies $a \parallel b = (ab + ba) + c$. Encapsulation can be used to enforce communication between different parallel components, e.g., $\partial_{\{a,b\}}(a \parallel b) = (\delta\delta + \delta\delta) + c = c$ (by the two laws $x + \delta = x$ and $\delta x = \delta$).

Kleene's binary star is in ACP-based process algebra defined by $x^* y = x(x^* y) + y$ (see [5]). Taking δ for y it follows that $t^*\delta$ behaves as t^∞, i.e., an infinite sequence of t-actions. The *pushdown* operator is defined by $x^\$ y = x(x^\$ y)(x^\$ y) + y$ (see [12]).

In common process semantics involving abstraction, the law $x\tau = x$ holds, and this identity is used in the paper. Furthermore, fairness implies that $\tau^*\delta = \tau\delta$ [5]. Finally, the unary abstraction operator τ_I with I a set of actions, renames the actions in I into τ (cf. encapsulation).

References

[1] ASCII table and description, http://www.lookuptables.com/.
[2] J.W. de Bakker, J.I. Zucker, Processes and the denotational semantics of concurrency, Information and Control 54 (1/2) (1982) 70–120.
[3] J.C.M. Baeten, J.A. Bergstra, Process algebra with a zero object, in: J.C.M. Baeten, J.W. Klop (Eds.), Proceedings CONCUR'90, Amsterdam, in: Lecture Notes in Computer Science, vol. 458, Springer-Verlag, 1990, pp. 83–98.
[4] J.A. Bergstra, I. Bethke, Polarized process algebra and program equivalence, in: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (Eds.), Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30–July 4, in: Lecture Notes in Computer Science, vol. 2719, Springer-Verlag, 2003, pp. 1–21.
[5] J.A. Bergstra, I. Bethke, A. Ponse, Process algebra with iteration and nesting, The Computer Journal 37 (4) (1994) 243–258.
[6] J.A. Bergstra, I. Bethke, P.H. Rodenburg, A propositional logic with 4 values: true, false, divergent and meaningless, Journal of Applied Non-Classical Logics 5 (1995) 199–217.
[7] J.A. Bergstra, J.W. Klop, Process algebra for synchronous communication, Information and Control 60 (1–3) (1984) 109–137.
[8] J.A. Bergstra, M.E. Loots, Program algebra for sequential code, Journal of Logic and Algebraic Programming 51 (2) (2002) 125–156.
[9] J.A. Bergstra, C.A. Middelburg, A thread algebra with multi-level strategic interleaving, in: S.B. Cooper, B. Loewe, L. Torenvliet (Eds.), CiE 2005, in: Lecture Notes in Computer Science, vol. 3526, Springer-Verlag, 2005, pp. 35–48.
[10] J.A. Bergstra, A. Ponse, Bochvar–McCarthy logic and process algebra, Notre Dame Journal of Formal Logic 39 (4) (1998) 464–484.
[11] J.A. Bergstra, A. Ponse, Register-machine based processes, Journal of the ACM 48 (6) (2001) 1207–1241.
[12] J.A. Bergstra, A. Ponse, Non-regular iterators in process algebra, Theoretical Computer Science 269 (1–2) (2001) 203–229.
[13] J.A. Bergstra, A. Ponse, Combining programs and state machines, Journal of Logic and Algebraic Programming 51 (2) (2002) 175–192.
[14] J.A. Bergstra, A. Ponse, A bypass of Cohen's impossibility result, in: P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, M. Bubak (Eds.), Advances in Grid Computing—EGC 2005, in: Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2005, pp. 1097–1106.
[15] F. Cohen, Computer viruses: theory and experiments, Computers & Security 6 (1) (1987) 22–35.
[16] W.J. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science, Springer-Verlag, 2000.
[17] M. Gardner, Knotted Doughnuts and Other Mathematical Entertainments, W.H. Freeman, New York, 1986.
[18] S.T. Kuhn, Prisoner's Dilemma, http://plato.stanford.edu/entries/prisoner-dilemma/, 2003 (including many references).
[19] H.R. Lewis, C.H. Papadimitriou, Elements of the Theory of Computation, second ed., Prentice-Hall, 1997.
[20] H. Rogers, Theory of Recursive Functions and Effective Computability, McGraw-Hill Book Co., 1967. Reprinted, MIT Press, 1987.
[21] A. Turing, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Ser. 2 42 (1937) 230–265. Corrections: Proceedings of the London Mathematical Society, Ser. 2 43 (1937) 544–546.