



UvA-DARE (Digital Academic Repository)

On the Formalization of the Notion of an Algorithm

Middelburg, Cornelis A.

DOI

[10.1007/978-3-031-66673-5_2](https://doi.org/10.1007/978-3-031-66673-5_2)

Publication date

2024

Document Version

Final published version

Published in

The Practice of Formal Methods

License

Article 25fa Dutch Copyright Act (<https://www.openaccess.nl/en/policies/open-access-in-dutch-copyright-law-taverne-amendment>)

[Link to publication](#)

Citation for published version (APA):

Middelburg, C. A. (2024). On the Formalization of the Notion of an Algorithm. In A. Cavalcanti, & J. Baxter (Eds.), *The Practice of Formal Methods: Essays in Honour of Cliff Jones* (Vol. II, pp. 23-44). (Lecture Notes in Computer Science; Vol. 14781). Springer.
https://doi.org/10.1007/978-3-031-66673-5_2

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.



On the Formalization of the Notion of an Algorithm

Cornelis A. Middelburg^(✉) 

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 900, 1098 XH Amsterdam, The Netherlands
C.A.Middelburg@uva.nl

Abstract. The starting point of this paper is a collection of properties of an algorithm that have been distilled from the informal descriptions of what an algorithm is that are given in standard works from the mathematical and computer science literature. Based on that, the notion of a proto-algorithm is introduced. The thought is that algorithms are equivalence classes of proto-algorithms under some equivalence relation. Three equivalence relations are defined. Two of them give bounds between which an appropriate equivalence relation must lie. The third lies in between these two and is likely an appropriate equivalence relation. A sound method is presented to prove, using an imperative process algebra based on ACP, that this equivalence relation holds between two proto-algorithms.

Keywords: Proto-algorithm · Algorithmic equivalence · Computational equivalence · Imperative process algebra · Algorithm process

1 Introduction

Cliff Jones' Vienna Development Method (VDM) [14] is one of the most impactful program development methods based on formal specification and verified design. In several of his illuminating explanations of data reification and operation decomposition in VDM, Cliff uses the term algorithm for what is expressed by the result of the data reification and operation decomposition steps of a verified design. In those explanations, he can rightly rely on an intuitive understanding of what an algorithm is. However, I find it unsatisfactory that there is still no proper formalization of this central notion of computer science. This paper concerns a quest for a formalization of the notion of an algorithm.

In many works from the mathematical and computer science literature, including standard works such as [15, 16, 19, 24], the notion of an algorithm is informally characterized by properties that are considered the most important ones of an algorithm. Most of those characterizations agree with each other and indicate that an algorithm is considered to express a pattern of behaviour by which all instances of a computational problem can be solved. A remark like “Formally, an algorithm is a Turing machine” is often made in the works concerned if additionally Turing machines are rigorously defined.

However, the viewpoint that the formal notion of a Turing machine is a formalization of the intuitive notion of an algorithm is unsatisfactory in at least two ways: (a) a Turing machine expresses primarily a way in which a computational problem-solving pattern of behaviour can be generated and (b) a Turing machine restricts the data involved in such a pattern of behaviour to strings over some finite set of symbols. There are not many alternative formalizations of the notion of an algorithm that are regularly cited. To the best of my knowledge, the main exceptions are the ones that can be found in [12, 21]. In both papers, a notion of an algorithm is formally defined that does not depend on a particular machine model such as the Turing machine model.

In [21], an algorithm is defined as a fairly complex set-theoretic object. The definition has its origins in the idea that, if a partial function is defined recursively by a system of equations, that system of equations induces an algorithm. An algorithm according to this definition fails to have many properties that are generally considered to belong to the most important ones of an algorithm.

In [12], an algorithm is defined as an object that satisfies certain postulates. The postulates concerned appear to be devised with the purpose that Gurevich's abstract state machines would satisfy them. However, this definition covers objects that have almost all properties that are generally considered to belong to the most important ones of an algorithm as well as more abstract objects that have almost none of those properties.

What is mentioned above about the formalizations of the notion of an algorithm in [12, 21] makes them unsatisfactory as well. This state of affairs motivated me to start a quest for a formalization of the notion of an algorithm that is more satisfactory than the existing ones. One possibility is to investigate whether this can be done by adapting the postulates from [12] or adding postulates to them. Another possibility is to investigate whether a constructive definition can be given. This is what will be done in this paper. In addition, the connection between the resulting objects and the processes considered in the imperative process algebra presented in [20] will be investigated.

In [3], I made a first attempt to give a constructive definition. A main drawback of the approach followed there is that the data involved in an algorithm is restricted to bit strings. The idea was that this restriction could be discarded without much effort. This turned out not to be the case. Therefore, I follow a rather different approach in this paper.

2 The Informal Notion of an Algorithm

What is an algorithm? A brief answer to this question usually goes something like this: an algorithm is a procedure for solving a computational problem in a finite number of steps. This is a reasonable answer. A difficulty is that it is common to describe a computational problem informally as a problem that can be solved using an algorithm. For this reason, first a description of a computational problem that does not refer to the notion of an algorithm must be given:

A computational problem is a problem where, given an input value that belongs to a certain set, an output value that is in a certain relation to the given input value must be found if it exists. The input values that belong to the certain set are also called the instances of the problem and an output value that is in the certain relation to the given input value is also called a solution for the instance concerned.

The existing viewpoints on what an algorithm is indicate that something like the following properties are essential for an algorithm:

- an algorithm is a finite expression of a pattern of behaviour by which all instances of a computational problem can be solved;
- the pattern of behaviour expressed by an algorithm is made up of discrete steps, each of which consists of performing an elementary operation or inspecting an elementary condition unless it is the initial step or a final step;
- the pattern of behaviour expressed by an algorithm is such that there is one possible step immediately following a step that consists of performing an operation;
- the pattern of behaviour expressed by an algorithm is such that there is one possible step immediately following a step that consists of inspecting a condition for each outcome of the inspection;
- the pattern of behaviour expressed by an algorithm is such that the initial step consists of inputting an input value of the problem concerned;
- the pattern of behaviour expressed by an algorithm is such that, for each input value of the problem concerned for which a correct output value exists, a final step is reached after a finite number of steps and that final step consists of outputting a correct output value for that input value;
- the steps involved in the pattern of behaviour expressed by an algorithm are precisely and unambiguously defined and can be performed exactly in a finite amount of time.

These properties give an intuitive characterization of the notion of an algorithm and form the starting point for the formalization of this notion in upcoming sections. They have been distilled from the descriptions of what an algorithm is that are given in standard works from the mathematical and computer science literature such as [15, 16, 19, 24]. They can also be found elsewhere in the mathematical and computer science literature and even in the philosophical literature on algorithms, see e.g. [13, 22].

Usually it is also mentioned in some detail how an algorithm is generally expressed. However, usually it is mentioned at most in passing that an algorithm expresses a pattern of behaviour. Following [6], this point is central here. The reason for this is that, in order to formalize the notion of an algorithm well, it is more important to know what an algorithm expresses than how an algorithm is expressed.

Recently, discussions about the notion of an algorithm take also place in the social sciences. This leads to viewpoints on algorithms that are useless in mathematics and computer science. For example, in [26] is proposed to view algorithms

as ‘heterogeneous and diffuse sociotechnical systems’. Such viewpoints preclude formalization and are therefore disregarded.

It should be noted that the characterization of the notion of an algorithm given by the above-mentioned properties of an algorithm reflects a rather operational view of what an algorithm is. In a more abstract view of what an algorithm is, an algorithm expresses a collection of patterns of behaviour that are equivalent in some well-defined way. We will come back to this at the end of Sect. 3.

3 Proto-Algorithms

In this section, the notion of an proto-algorithm is introduced. The thought is that algorithms are equivalence classes of proto-algorithms under an appropriate equivalence relation. An equivalence relation that is likely an appropriate one is introduced in Sect. 4.

The notion of a proto-algorithm will be defined in terms of three auxiliary notions. The definition of one of these auxiliary notions is based on the well-known notion of a rooted labeled directed graph. However, the definitions of this notion given in the mathematical and computer science literature vary. Therefore, the definition that is used in this paper is given first.

Definition. A rooted labeled directed graph G is a sextuple (V, E, L_v, L_e, l, r) , where:

- V is a non-empty finite set, whose members are called the vertices of G ;
- E is a subset of $V \times V$, whose members are called the edges of G ;
- L_v is a countable set, whose members are called the vertex labels of G ;
- L_e is a countable set, whose members are called the edge labels of G ;
- l is a partial function from $V \cup E$ to $L_v \cup L_e$ such that
 - for all $v \in V$ for which $l(v)$ is defined, $l(v) \in L_v$ and
 - for all $e \in E$ for which $l(e)$ is defined, $l(e) \in L_e$,
 called the labeling function of G ;
- $r \in V$, called the root of G .

The additional graph theoretical notions defined below are also used in this paper.

Definition. Let $G = (V, E, L_v, L_e, l, r)$ be a rooted labeled directed graph. Then a cycle in G is a sequence $v_1 \dots v_{n+1} \in V^*$ such that, for all $i \in \{1, \dots, n\}$, $(v_i, v_{i+1}) \in E$, $\text{card}(\{v_1, \dots, v_n\}) = n$, and $v_1 = v_{n+1}$. Let, moreover, $v \in V$. Then the indegree of v , written $\text{indegree}(v)$, is $\text{card}(\{v' \mid (v', v) \in E\})$ and the outdegree of v , written $\text{outdegree}(v)$, is $\text{card}(\{v' \mid (v, v') \in E\})$.

We proceed with defining the three auxiliary notions, starting with the notion of an alphabet. This notion concerns the symbols used to refer to the operations and conditions involved in the steps of which the pattern of behaviour expressed by an algorithm is made up.

Definition. An alphabet Σ is a couple (F, P) , where:

- F is a countable set, whose members are called the function symbols of Σ ;
- P is a countable set, whose members are called the predicate symbols of Σ ;
- F and P are disjoint sets and $\text{ini}, \text{fin} \in F$.

We write \tilde{F} , where F is the set of function symbols of an alphabet, for the set $F \setminus \{\text{ini}, \text{fin}\}$.

The function symbols and predicate symbols of an alphabet refer to the operations and conditions, respectively, involved in the steps of which the pattern of behaviour expressed by an algorithm is made up. The function symbols ini and fin refer to inputting an input value and outputting an output value, respectively.

We are now ready to define the notions of a Σ -algorithm graph and a Σ -interpretation. They concern the pattern of behaviour expressed by an algorithm.

Definition. Let $\Sigma = (F, P)$ be an alphabet. Then a Σ -algorithm graph G is a rooted labeled directed graph (V, E, L_v, L_e, l, r) such that

- $L_v = F \cup P$;
- $L_e = \{0, 1\}$;
- for all $v \in V$:
 - $l(v) = \text{ini}$ iff $v = r$;
 - if $l(v) = \text{ini}$, then $\text{indegree}(v) = 0$, $\text{outdegree}(v) = 1$, and, for the unique $v' \in V$ for which $(v, v') \in E$, $l((v, v'))$ is undefined;
 - if $l(v) = \text{fin}$, then $\text{indegree}(v) > 0$ and $\text{outdegree}(v) = 0$;
 - if $l(v) \in \tilde{F}$, then $\text{indegree}(v) > 0$, $\text{outdegree}(v) = 1$, and, for the unique $v' \in V$ for which $(v, v') \in E$, $l((v, v'))$ is undefined;
 - if $l(v) \in P$, then $\text{indegree}(v) > 0$, $\text{outdegree}(v) = 2$, and, for the unique $v' \in V$ and $v'' \in V$ with $v' \neq v''$ for which $(v, v') \in E$ and $(v, v'') \in E$, $l((v, v'))$ is defined, $l((v, v''))$ is defined, and $l((v, v')) \neq l((v, v''))$;
- if $v_1 \dots v_{n+1}$ is a cycle in G , then, for some $v \in \{v_1, \dots, v_n\}$, $l(v) \in F$.

Σ -algorithm graphs are somewhat reminiscent of program schemes as defined, for example, in [27].

In the above definition, the condition on cycles in a Σ algorithm graph excludes infinitely many consecutive steps, each of which consists of inspecting a condition.

In the above definition, the conditions regarding the vertices of a Σ -algorithm graph correspond to the essential properties of an algorithm mentioned in Sect. 2 that concern its structure. Adding an interpretation of the symbols of the alphabet Σ to a Σ -algorithm graph yields something that has all of the mentioned essential properties of an algorithm.

Definition. Let $\Sigma = (F, P)$ be an alphabet. Then a Σ -interpretation \mathcal{I} is a quadruple $(D, D_{\text{in}}, D_{\text{out}}, I)$, where:

- D is a set, called the main domain of \mathcal{I} ;
- D_{in} is a set, called the input domain of \mathcal{I} ;
- D_{out} is a set, called the output domain of \mathcal{I} ;

- I is a total function from $F \cup P$ to the set of all total computable functions from D_{in} to D , D to D_{out} , D to D or D to $\{0, 1\}$ such that:
 - $I(\text{ini})$ is a function from D_{in} to D ;
 - $I(\text{fin})$ is a function from D to D_{out} ;
 - for all $f \in \tilde{F}$, $I(f)$ is a function from D to D ;
 - for all $p \in P$, $I(p)$ is a function from D to $\{0, 1\}$;
- there does not exist a $D' \subset D$ such that:
 - for all $d \in D_{\text{in}}$, $I(\text{ini})(d) \in D'$;
 - for all $f \in \tilde{F}$, for all $d \in D'$, $I(f)(d) \in D'$.

In the above definition, the minimality condition on D is not essential, but this condition facilitates establishing a connection between proto-algorithms and the processes considered in the imperative process algebra $\text{BPA}_{\delta\epsilon}\text{-I}$ (see Sect. 6).

The pattern of behavior expressed by an algorithm can completely be represented by the combination of an alphabet Σ , a Σ -algorithm graph G , and a Σ -interpretation \mathcal{I} . This brings us to defining the notion of a proto-algorithm.

Definition. A proto-algorithm A is a triple (Σ, G, \mathcal{I}) , where:

- Σ is an alphabet, called the alphabet of A ;
- G is a Σ -algorithm graph, called the algorithm graph of A ;
- \mathcal{I} is a Σ -interpretation, called the interpretation of A .

Let $A = (\Sigma, G, \mathcal{I})$ be a proto-algorithm, where $\Sigma = (F, P)$, $G = (V, E, L_v, L_e, l, r)$, and $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$. Then the intuition is that A is something that goes through states, where states are elements of the set $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$. The elements of D_{in} , $V \times D$, and D_{out} are called input states, internal states, and output states, respectively. A goes from one state to the next state by making a step, it starts in an input state, and it stops in an output state. The state that A is in determines what the step to the next state consists of and what the next state is as follows:

- if A is in input state d , then the step to the next state consists of applying function $I(\text{ini})$ to d and the next state is the unique internal state (v', d') such that $(r, v') \in E$, and $I(\text{ini})(d) = d'$;
- if A is in internal state (v, d) and $l(v) \in \tilde{F}$, then the step to the next state consists of applying function $I(l(v))$ to d and the next state is the unique internal state (v', d') such that $(v, v') \in E$, and $I(l(v))(d) = d'$;
- if A is in internal state (v, d) and $l(v) \in P$, then the step to the next state consists of applying function $I(l(v))$ to d and the next state is the unique internal state (v', d) such that $(v, v') \in E$, and $I(l(v))(d) = l((v, v'))$;
- if A is in internal state (v, d) and $l(v) = \text{fin}$, then the step to the next state consists of applying function $I(\text{fin})$ to d and the next state is the unique output state d' such that $I(\text{fin})(d) = d'$.

This informal explanation of how the state that A is in determines what the next state is, is formalized by the algorithmic step function δ_A^a defined in Sect. 4.

The term proto-algorithm has been chosen instead of the term algorithm because proto-algorithms are considered too concrete to be called algorithms. For example, from a mathematical point of view, it is natural to consider the behavioral patterns expressed by isomorphic proto-algorithms to be the same. Isomorphism of proto-algorithms is defined as expected.

Definition. Let $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma', G', \mathcal{I}')$ be proto-algorithms, where $\Sigma = (F, P)$, $\Sigma' = (F', P')$, $G = (V, E, L_v, L_e, l, r)$, $G' = (V', E', L'_v, L'_e, l', r')$, $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$, and $\mathcal{I}' = (D', D'_{\text{in}}, D'_{\text{out}}, I')$. Then A and A' are isomorphic, written $A \cong A'$, if there exist bijections $\beta_f: F \rightarrow F'$, $\beta_p: P \rightarrow P'$, $\beta_v: V \rightarrow V'$, $\beta_d: D \rightarrow D'$, $\beta_i: D_{\text{in}} \rightarrow D'_{\text{in}}$, $\beta_o: D_{\text{out}} \rightarrow D'_{\text{out}}$, and $\beta_b: \{0, 1\} \rightarrow \{0, 1\}$ such that:

- $\beta_f(\text{ini}) = \text{ini}$ and $\beta_f(\text{fin}) = \text{fin}$;
- for all $v, v' \in V$, $(v, v') \in E$ iff $(\beta_v(v), \beta_v(v')) \in E'$;
- for all $v \in V$ with $l(v) \in F$, $\beta_f(l(v)) = l'(\beta_v(v))$;
- for all $v \in V$ with $l(v) \in P$, $\beta_p(l(v)) = l'(\beta_v(v))$;
- for all $(v, v') \in E$ with $l((v, v'))$ is defined, $\beta_b(l((v, v'))) = l'((\beta_v(v), \beta_v(v')))$;
- for all $d \in D_{\text{in}}$, $\beta_d(I(\text{ini})(d)) = I'(\text{ini})(\beta_i(d))$;
- for all $d \in D$, $\beta_o(I(\text{fin})(d)) = I'(\text{fin})(\beta_d(d))$;
- for all $d \in D$ and $f \in \tilde{F}$, $\beta_d(I(f)(d)) = I'(\beta_f(f))(\beta_d(d))$;
- for all $d \in D$ and $p \in P$, $\beta_b(I(p)(d)) = I'(\beta_p(p))(\beta_d(d))$.

Proto-algorithms may also be considered too concrete in a way not covered by isomorphism of proto-algorithms. This issue is addressed in Sect. 4 and leads there to the introduction of two other equivalence relations. Although it is intuitive clear what isomorphism of proto-algorithms is, its precise definition is not easy to memorize. The equivalence relations that are given in Sect. 4 may be easier to memorize.

A proto-algorithm could also be defined as a quadruple $(D, D_{\text{in}}, D_{\text{out}}, \overline{G})$ where \overline{G} is a graph that differs from a Σ -algorithm graph in that its vertex labels are computable functions from D_{in} to D , D to D_{out} , D to D or D to $\{0, 1\}$ instead of function and predicate symbols from Σ . I consider the definition of a proto-algorithm given earlier more insightful because it isolates as much as possible the operations to be performed and the conditions to be inspected from its structure.

4 Algorithmic and Computational Equivalence

In Sect. 3, the intuition was given that a proto-algorithm A is something that goes through states. It was informally explained how the state that it is in determines what the next state is. The algorithmic step function δ_A^a that is defined below formalizes this. The computational step function δ_A^c that is also defined below is like the algorithmic step function δ_A^a , but conceals the steps that consist of inspecting conditions.

Definition. Let $A = (\Sigma, G, \mathcal{I})$ be a proto-algorithm, where $\Sigma = (F, P)$, $G = (V, E, L_v, L_e, l, r)$, and $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$. Then the algorithmic step function δ_A^{a} induced by A is the unary total function on the set $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$ defined by:

$$\begin{aligned} \delta_A^{\text{a}}(d) &= (v', d') \text{ if } d \in D_{\text{in}}, (r, v') \in E, \text{ and } I(\text{ini})(d) = d'; \\ \delta_A^{\text{a}}((v, d)) &= (v', d') \text{ if } l(v) = o, o \in \tilde{F}, (v, v') \in E, \text{ and } I(o)(d) = d'; \\ \delta_A^{\text{a}}((v, d)) &= (v', d) \text{ if } l(v) = p, p \in P, (v, v') \in E, \text{ and } I(p)(d) = l((v, v')); \\ \delta_A^{\text{a}}((v, d)) &= d' \text{ if } l(v) = \text{fin} \text{ and } I(\text{fin})(d) = d'; \\ \delta_A^{\text{a}}(d) &= d \text{ if } d \in D_{\text{out}}; \end{aligned}$$

and the computational step function δ_A^{c} induced by A is the unary total function on the set $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$ defined by:

$$\begin{aligned} \delta_A^{\text{c}}(d) &= (v', d') \text{ if } d \in D_{\text{in}}, (r, v') \in E, \text{ and } I(\text{ini})(d) = d'; \\ \delta_A^{\text{c}}((v, d)) &= (v', d') \text{ if } l(v) = o, o \in \tilde{F}, (v, v') \in E, \text{ and } I(o)(d) = d'; \\ \delta_A^{\text{c}}((v, d)) &= \delta_A^{\text{c}}((v', d)) \text{ if } l(v) = p, p \in P, (v, v') \in E, \text{ and } I(p)(d) = l((v, v')); \\ \delta_A^{\text{c}}((v, d)) &= d' \text{ if } l(v) = \text{fin} \text{ and } I(\text{fin})(d) = d'; \\ \delta_A^{\text{c}}(d) &= d \text{ if } d \in D_{\text{out}}. \end{aligned}$$

If a proto-algorithm A' can mimic a proto-algorithm A step-by-step, then we say that A is algorithmically simulated by A' . If the steps that consist of inspecting conditions are ignored, then we say that A is computationally simulated by A' . Algorithmic and computational simulation can be formally defined using the step functions defined above.

Definition. Let $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma', G', \mathcal{I}')$ be two proto-algorithms, where $G = (V, E, L_v, L_e, l, r)$, $G' = (V', E', L'_v, L'_e, l', r')$, $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$, and $\mathcal{I}' = (D', D'_{\text{in}}, D'_{\text{out}}, I')$. Then an algorithmic simulation of A by A' is a set $R \subseteq (D_{\text{in}} \times D'_{\text{in}}) \cup ((V \times D) \times (V' \times D')) \cup (D_{\text{out}} \times D'_{\text{out}})$ such that:

$$\begin{aligned} \text{if } d \in D_{\text{in}}, & \text{ then there exists a unique } d' \in D'_{\text{in}} \text{ such that } (d, d') \in R; \\ \text{if } d' \in D'_{\text{out}}, & \text{ then there exists a unique } d \in D_{\text{out}} \text{ such that } (d, d') \in R; \\ \text{if } (d, d') \in R, & \text{ then } (\delta_A^{\text{a}}(d), \delta_{A'}^{\text{a}}(d')) \in R; \end{aligned}$$

and a computational simulation of A by A' is a set $R \subseteq (D_{\text{in}} \times D'_{\text{in}}) \cup ((V \times D) \times (V' \times D')) \cup (D_{\text{out}} \times D'_{\text{out}})$ such that:

$$\begin{aligned} \text{if } d \in D_{\text{in}}, & \text{ then there exists a unique } d' \in D'_{\text{in}} \text{ such that } (d, d') \in R; \\ \text{if } d' \in D'_{\text{out}}, & \text{ then there exists a unique } d \in D_{\text{out}} \text{ such that } (d, d') \in R; \\ \text{if } (d, d') \in R, & \text{ then } (\delta_A^{\text{c}}(d), \delta_{A'}^{\text{c}}(d')) \in R. \end{aligned}$$

A is algorithmically simulated by A' , written $A \sqsubseteq_{\text{a}} A'$, if there exists an algorithmic simulation of A by A' .

A is computationally simulated by A' , written $A \sqsubseteq_{\text{c}} A'$, if there exists a computational simulation of A by A' .

A is algorithmically equivalent to A' , written $A \equiv_{\text{a}} A'$, if $A \sqsubseteq_{\text{a}} A'$ and $A' \sqsubseteq_{\text{a}} A$.

A is computationally equivalent to A' , written $A \equiv_{\text{c}} A'$, if $A \sqsubseteq_{\text{c}} A'$ and $A' \sqsubseteq_{\text{c}} A$.

The following theorem tells us how isomorphism, algorithmic equivalence, and computational equivalence are related.

Theorem 1. *Let A and A' be proto-algorithms. Then:*

$$(1) \quad A \cong A' \text{ only if } A \equiv_a A' \qquad (2) \quad A \equiv_a A' \text{ only if } A \equiv_c A'.$$

Proof. Let $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma', G', \mathcal{I}')$ be proto-algorithms, where $\Sigma = (F, P)$, $\Sigma' = (F', P')$, $G = (V, E, L_v, L_e, l, r)$, $G' = (V', E', L'_v, L'_e, l', r')$, $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$, and $\mathcal{I}' = (D', D'_{\text{in}}, D'_{\text{out}}, I')$.

Part 1. Let $\beta_v, \beta_d, \beta_i$, and β_o be as in the definition of \cong , and let β be the bijection from $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$ to $D'_{\text{in}} \cup (V' \times D') \cup D'_{\text{out}}$ defined by: $\beta(d) = \beta_i(d)$ if $d \in D_{\text{in}}$, $\beta((v, d)) = (\beta_v(v), \beta_d(d))$, and $\beta(d) = \beta_o(d)$ if $d \in D_{\text{out}}$. It is easy to show that, for all $d \in D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$, $\beta(\delta_A^a(d)) = \delta_{A'}^a(\beta(d))$. It immediately follows that the set $\{(\delta_A^{a^n}(d), \beta(\delta_{A'}^{a^n}(d))) \mid d \in D_{\text{in}} \wedge n \in \mathbb{N}\}$ is an algorithmic simulation of A by A' .¹ Hence, $A \sqsubseteq_a A'$. The proof of $A' \sqsubseteq_a A$ is done in the same way.

Part 2. Because $A \equiv_a A'$, there exists an algorithmic simulation of A by A' . Let R be an algorithmic simulation of A by A' . Then it is easy to show that, for all $(d, d') \in R$, $(\delta_A^c(d), \delta_{A'}^c(d')) \in R$. It immediately follows that R is also a computational simulation of A by A' . Hence, $A \sqsubseteq_c A'$. The proof of $A' \sqsubseteq_c A$ is done in the same way. \square

We do not have that $A \cong A'$ if $A \equiv_a A'$. The following example illustrates this. Take proto-algorithms $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma, G', \mathcal{I})$ where:

- G contains edges (v_1, v'_1) , (v_1, v'') , (v_2, v'_2) , and (v'_2, v'') where the vertices v'_1 and v'_2 are labeled by the same function symbol;
- G' is obtained from G by replacing the edge (v_2, v'_2) by (v_2, v'_1) and removing the edge (v'_2, v'') .

Clearly, A and A' are algorithmically equivalent, but not isomorphic.

We also do not have $A \equiv_a A'$ if $A \equiv_c A'$. The following example illustrates this. Take proto-algorithms $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma, G', \mathcal{I})$ where:

- G contains a cycle in which only one vertex occurs that is labeled by a predicate symbol p and the outgoing edge of this vertex that is not part of the cycle is labeled by 1;
- G' is obtained from G by adding immediately before the cycle a copy of the cycle in which the predicate symbol p is replaced by a predicate symbol p' whose interpretation yields 1 whenever the interpretation of p yields 1.

It is easy to see that A and A' are computationally equivalent, but not algorithmically equivalent.

¹ The notation $\delta_A^{a^n}(d)$, where $n \in \mathbb{N}$, is used for the n -fold application of δ_A^a to d , i.e. $\delta_A^{a^0}(d) = d$ and $\delta_A^{a^{n+1}}(d) = \delta_A^a(\delta_A^{a^n}(d))$.

The definition of algorithmic equivalence suggests that the patterns of behaviour expressed by algorithmically equivalent proto-algorithms must be considered the same. This suggests in turn that algorithms are equivalence classes of proto-algorithms under algorithmic equivalence.

If two proto-algorithms are computationally equivalent, then, for each input value, they lead to the same sequence of operations being performed. The point of view should not be taken that the patterns of behaviour expressed by computationally equivalent proto-algorithms are the same: the steps that consist of inspecting a condition are treated as if they do not belong to the patterns of behaviour.

The relevance of the computational equivalence relation is that any equivalence relation that captures the sameness of the patterns of behaviour expressed by proto-algorithms to a higher degree than the algorithmic equivalence relation must be finer than the computational equivalence relation.

Definition. Let $A = (\Sigma, G, \mathcal{I})$ be a proto-algorithm, where $\Sigma = (F, P)$, $G = (V, E, L_v, L_e, l, r)$, and $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$. Then the function \widehat{A} computed by A is the partial function from D_{in} to D_{out} defined by $\widehat{A}(d) = \delta_A^{a*}(d)$, where δ_A^{a*} is the least-defined unary partial function on $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$ satisfying

$$\begin{aligned} \delta_A^{a*}(d) &= \delta_A^a(\delta_A^a(d)) \text{ if } \delta_A^a(d) \in V \times D; \\ \delta_A^{a*}(d) &= \delta_A^a(d) \quad \text{if } \delta_A^a(d) \in D_{\text{out}}. \end{aligned}$$

Let, moreover, $d \in D_{\text{in}}$ be such that $\widehat{A}(d)$ is defined. Then the number of algorithmic steps to compute $\widehat{A}(d)$ by A , written $\#_{\text{astep}}(A, d)$, is the smallest $n \in \mathbb{N}$ such that $\delta_A^{a^n}(d) = \widehat{A}(d)$.

The following theorem tells us that, if a proto-algorithm A is simulated by a proto-algorithm A' , then (a) the function computed by A' models the function computed by A (in the sense of e.g. [14]) and (b) for each input value for which A eventually outputs an output value, A' does so in the same number of algorithmic steps.

Theorem 2. Let $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma', G', \mathcal{I}')$ be proto-algorithms, where $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$ and $\mathcal{I}' = (D', D'_{\text{in}}, D'_{\text{out}}, I')$. Then $A \sqsubseteq_a A'$ only if there exist total functions $\gamma_i : D_{\text{in}} \rightarrow D'_{\text{in}}$ and $\gamma_o : D'_{\text{out}} \rightarrow D_{\text{out}}$ such that:

- (1) for all $d \in D_{\text{in}}$, $\widehat{A}(d)$ is defined only if $\widehat{A}'(\gamma_i(d))$ is defined;
- (2) for all $d \in D_{\text{in}}$ and $d' \in D_{\text{out}}$, $\widehat{A}(d) = d'$ only if $\gamma_o(\widehat{A}'(\gamma_i(d))) = d'$;
- (3) for all $d \in D_{\text{in}}$ such that $\widehat{A}(d)$ is defined, $\#_{\text{astep}}(A, d) = \#_{\text{astep}}(A', \gamma_i(d))$.

Proof. Because $A \sqsubseteq_a A'$, there exists an algorithmic simulation of A by A' . Let R be an algorithmic simulation of A by A' , let γ_i be the unique function from D_{in} to D'_{in} such that, for all $d \in D_{\text{in}}$, $(d, \gamma_i(d)) \in R$, and let γ_o be the unique function from D'_{out} to D_{out} such that, for all $d' \in D'_{\text{out}}$, $(\gamma_o(d'), d') \in R$. From the definition of an algorithmic simulation, it follows immediately that, for all $d \in D_{\text{in}}$, for all $n \in \mathbb{N}$, $(\delta_A^{a^n}(d), \delta_{A'}^{a^n}(\gamma_i(d))) \in R$. From this result and the definition of an algorithmic simulation, it follows immediately that, for all $d \in D_{\text{in}}$, for all $n \in \mathbb{N}$:

- (a) $\delta_A^{an}(d) \in D_{\text{out}}$ iff $\delta_{A'}^{an}(\gamma_i(d)) \in D'_{\text{out}}$;
 (b) for all $d' \in D_{\text{out}}$, $\delta_A^{an}(d) = d'$ iff there exists a $d'' \in D'_{\text{out}}$ such that $\delta_{A'}^{an}(\gamma_i(d)) = d''$ and $\gamma_o(d'') = d'$.

By the definition of the function computed by a proto-algorithm, we have that $\widehat{A}(d)$ is defined iff there exists an $n \in \mathbb{N}$ such that $\delta_A^{an}(d) \in D_{\text{out}}$ and that $\widehat{A'}(\gamma_i(d))$ is defined iff there exists an $n \in \mathbb{N}$ such that $\delta_{A'}^{an}(\gamma_i(d)) \in D'_{\text{out}}$. From this and (a), (1) follows immediately.

By the definition of the function computed by a proto-algorithm, we have that $\widehat{A}(d) = d'$ iff there exists an $n \in \mathbb{N}$ such that $\delta_A^{an}(d) = d'$ and that $\widehat{A'}(\gamma_i(d)) = d''$ iff there exists an $n \in \mathbb{N}$ such that $\delta_{A'}^{an}(\gamma_i(d)) = d''$. From this and (b), (2) follows immediately.

By the definition of $\#_{\text{astep}}$ and (a), (3) also follows immediately. \square

It is easy to see that, for all $d \in D_{\text{in}}$, $\widehat{A}(d) = \delta_A^{c*}(d)$, where δ_A^{c*} is the least-defined unary partial function on $D_{\text{in}} \cup (V \times D) \cup D_{\text{out}}$ satisfying

$$\begin{aligned} \delta_A^{c*}(d) &= \delta_A^{c*}(\delta_A^c(d)) \text{ if } \delta_A^c(d) \in V \times D; \\ \delta_A^{c*}(d) &= \delta_A^c(d) \quad \text{if } \delta_A^c(d) \in D_{\text{out}}. \end{aligned}$$

This means that Theorem 2 goes through as far as (1) and (2) are concerned if algorithmic simulation is replaced by computational simulation. It follows immediately from the example of computationally equivalent proto-algorithms given earlier that (3) does not go through if algorithmic simulation is replaced by computational simulation.

5 The Imperative Process Algebra $\text{BPA}_{\delta\epsilon}$ -I

In Sect. 6, a connection is made between proto-algorithms and the processes that are considered in the imperative process algebra $\text{BPA}_{\delta\epsilon}$ -I. In this section, a short survey of $\text{BPA}_{\delta\epsilon}$ -I and recursion in the setting of $\text{BPA}_{\delta\epsilon}$ -I is given. The constants and operators of the algebraic theory $\text{BPA}_{\delta\epsilon}$ -I and the additional constants of its extension with recursion are discussed. The axioms of $\text{BPA}_{\delta\epsilon}$ -I are given in the Appendix. $\text{BPA}_{\delta\epsilon}$ -I is a subtheory of $\text{ACP}_{\epsilon}^{\tau}$ -I. In [20], a comprehensive treatment of $\text{ACP}_{\epsilon}^{\tau}$ -I can be found. The axioms of $\text{BPA}_{\delta\epsilon}$ -I are the axioms of $\text{ACP}_{\epsilon}^{\tau}$ -I in which only constants and operators of $\text{BPA}_{\delta\epsilon}$ -I occur. The additional axioms of the extension of $\text{BPA}_{\delta\epsilon}$ -I with recursion are simply the additional axioms of the extension of $\text{ACP}_{\epsilon}^{\tau}$ -I with recursion.

5.1 BPA with Inaction and Empty Process

First, a short survey of $\text{BPA}_{\delta\epsilon}$ is given. $\text{BPA}_{\delta\epsilon}$ is the version of BPA with inaction and empty process constants that was first presented in [1, Section 2.2]. In Sect. 5.2, $\text{BPA}_{\delta\epsilon}$ -I will be introduced as an extension of $\text{BPA}_{\delta\epsilon}$.

In $\text{BPA}_{\delta\epsilon}$, it is assumed that a fixed but arbitrary finite set \mathbf{A} of *basic actions*, with $\delta, \epsilon \notin \mathbf{A}$, has been given. Basic actions are taken as atomic processes.

The algebraic theory $\text{BPA}_{\delta\epsilon}$ has one sort: the sort \mathbf{P} of *processes*. This sort is made explicit to anticipate the need for many-sortedness later on. The algebraic theory $\text{BPA}_{\delta\epsilon}$ has the following constants and operators to build terms of sort \mathbf{P} :

- a *basic action* constant $a : \mathbf{P}$ for each $a \in A$;
- an *inaction* constant $\delta : \mathbf{P}$;
- an *empty process* constant $\epsilon : \mathbf{P}$;
- a binary *alternative composition* or *choice* operator $+$: $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$;
- a binary *sequential composition* operator \cdot : $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$.

It is assumed that there is a countably infinite set \mathcal{X} of variables of sort \mathbf{P} , which contains x , y and z . Terms are built as usual. Infix notation is used for the operators $+$ and \cdot . The following precedence convention are used to reduce the need for parentheses: the operator \cdot binds stronger than the operator $+$.

The constants a ($a \in A$), ϵ , and δ can be explained as follows: (a) a denotes the process that first performs the action a and then terminates successfully, (b) ϵ denotes the process that terminates successfully without performing any action, and (c) δ denotes the process that cannot do anything, it cannot even terminate successfully.

Let t and t' be closed $\text{BPA}_{\delta\epsilon}$ terms. Then the operators $+$ and \cdot can be explained as follows: (a) $t + t'$ denotes the process that behaves as the process denoted by t or as the process denoted by t' , where the choice between the two is resolved at the instant that one of them does something, and (b) $t \cdot t'$ denotes the process that first behaves as the process denoted by t and following successful termination of that process behaves as the process denoted by t' .

5.2 Imperative $\text{BPA}_{\delta\epsilon}$

$\text{BPA}_{\delta\epsilon}\text{-I}$, imperative $\text{BPA}_{\delta\epsilon}$, extends $\text{BPA}_{\delta\epsilon}$ with features to change data involved in a process in the course of the process and to proceed at certain stages of a process in a way that depends on the changing data.

In $\text{BPA}_{\delta\epsilon}\text{-I}$, it is assumed that the following has been given with respect to data:

- a many-sorted signature $\Sigma_{\mathcal{D}}$ that includes:
 - a sort \mathbf{D} of *data* and a sort \mathbf{B} of *bits*;
 - constants of sort \mathbf{D} and/or operators with result sort \mathbf{D} ;
 - constants 0 and 1 of sort \mathbf{B} and operators with result sort \mathbf{B} ;
- a minimal algebra \mathcal{D} of signature $\Sigma_{\mathcal{D}}$ in which the carrier of sort \mathbf{B} has cardinality 2 and the equation $0 = 1$ does not hold.

We write \mathbb{D} for the set of all closed terms over the signature $\Sigma_{\mathcal{D}}$ of sort \mathbf{D} .

In $\text{BPA}_{\delta\epsilon}\text{-I}$, it is moreover assumed that a finite or countably infinite set \mathcal{V} of *flexible variables* has been given. A flexible variable is a variable whose value may change in the course of a process.²

² The term flexible variable is used for this kind of variables in e.g. [17, 25].

A *flexible variable valuation* is a total function from \mathcal{V} to \mathbb{D} . We write $\mathcal{V}Val$ for the set of all flexible variable valuations.

Flexible variable valuations provide closed terms from \mathbb{D} that denote the members of \mathfrak{D} 's carrier of sort \mathbf{D} assigned to flexible variables when a $BPA_{\delta\epsilon}$ -I term of sort \mathbf{D} is evaluated. Because \mathfrak{D} is a minimal algebra, each member of \mathfrak{D} 's carrier of sort \mathbf{D} can be represented by a term from \mathbb{D} . We write d , where d is a member of \mathfrak{D} 's carrier of sort \mathbf{D} , for a fixed but arbitrary term from \mathbb{D} representing d when it is clear from the context that a term from \mathbb{D} is expected.

$BPA_{\delta\epsilon}$ -I has the following sorts: the sorts included in $\Sigma_{\mathfrak{D}}$, the sort \mathbf{C} of *conditions*, and the sort \mathbf{P} of *processes*.

For each sort s included in $\Sigma_{\mathfrak{D}}$ other than \mathbf{D} , $BPA_{\delta\epsilon}$ -I has only the constants and operators included in $\Sigma_{\mathfrak{D}}$ to build terms of sort s .

$BPA_{\delta\epsilon}$ -I has, in addition to the constants and operators included in $\Sigma_{\mathfrak{D}}$ to build terms of sorts \mathbf{D} , the following constants to build terms of sort \mathbf{D} :

- for each $v \in \mathcal{V}$, the *flexible variable* constant $v : \mathbf{D}$.

We write \mathcal{D} for the set of all closed $BPA_{\delta\epsilon}$ -I terms of sort \mathbf{D} .

$BPA_{\delta\epsilon}$ -I has the following constants and operators to build terms of sort \mathbf{C} :

- a binary *equality* operator $= : \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{C}$;
- a binary *equality* operator $= : \mathbf{D} \times \mathbf{D} \rightarrow \mathbf{C}$;³
- a *truth* constant $\mathbf{t} : \mathbf{C}$;
- a *falsity* constant $\mathbf{f} : \mathbf{C}$;
- a unary *negation* operator $\neg : \mathbf{C} \rightarrow \mathbf{C}$;
- a binary *conjunction* operator $\wedge : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$;
- a binary *disjunction* operator $\vee : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$;
- a binary *implication* operator $\Rightarrow : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$;

We write \mathcal{C} for the set of all closed $BPA_{\delta\epsilon}$ -I terms of sort \mathbf{C} .

$BPA_{\delta\epsilon}$ -I has, in addition to the constants and operators of $BPA_{\delta\epsilon}$, the following operators to build terms of sort \mathbf{P} :

- a unary *assignment action* operator $:=_v : \mathbf{D} \rightarrow \mathbf{P}$ for each $v \in \mathcal{V}$;
- a binary *guarded command* operator $:\rightarrow : \mathbf{C} \times \mathbf{P} \rightarrow \mathbf{P}$;
- a unary *evaluation* operator $V_\rho : \mathbf{P} \rightarrow \mathbf{P}$ for each $\rho \in \mathcal{V}Val$.

We write \mathcal{P} for the set of all closed $BPA_{\delta\epsilon}$ -I terms of sort \mathbf{P} .

It is assumed that there are countably infinite sets of variables of sort \mathbf{D} and \mathbf{C} and that the sets of variables of sort \mathbf{D} , \mathbf{C} , and \mathbf{P} are mutually disjoint and disjoint from \mathcal{V} .

The same notational conventions are used as before. Infix notation is also used for the additional binary operators. Moreover, the notation $[v := e]$, where $v \in \mathcal{V}$ and e is a $BPA_{\delta\epsilon}$ -I term of sort \mathbf{D} , is used for the term $:=_v(e)$.

Each term from \mathcal{C} can be taken as a formula of a first-order language with equality of \mathfrak{D} by taking the flexible variable constants as variables of sort \mathbf{D} . The

³ The overloading of $=$ can be trivially resolved if $\Sigma_{\mathfrak{D}}$ is without overloaded symbols.

flexible variable constants are implicitly taken as variables of sort \mathbf{D} wherever the context asks for a formula. In this way, each term from \mathcal{C} can be interpreted in \mathfrak{D} as a formula.

The notation $\phi \Leftrightarrow \psi$, where ϕ and ψ are $\text{BPA}_{\delta\epsilon}$ -I terms of sort \mathbf{C} , is used for the term $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$. The axioms of $\text{BPA}_{\delta\epsilon}$ -I include an equation $\phi = \psi$ for each two terms ϕ and ψ from \mathcal{C} for which the formula $\phi \Leftrightarrow \psi$ holds in \mathfrak{D} .

Let e be a term from \mathcal{D} , ϕ be a term from \mathcal{C} , and t be a term from \mathcal{P} . Then the additional operators to build terms of sort \mathbf{P} can be explained as follows:

- the term $[v := e]$ denotes the process that first performs the assignment action $[v := e]$, whose intended effect is the assignment of the result of evaluating e to flexible variable v , and then terminates successfully;
- the term $\phi \rightarrow t$ denotes the process that behaves as the process denoted by t if condition ϕ holds and as δ otherwise;
- the term $V_\rho(t)$ denotes the process that behaves as the process denoted by t , except that each subterm of t that belongs to \mathcal{D} is evaluated using flexible variable valuation ρ updated according to the assignment actions that have taken place at the point where the subterm is encountered.

Below will be referred to the subset \mathcal{A} of \mathcal{P} that consists of the terms from \mathcal{P} that denote the processes that are considered to be atomic.

\mathcal{A} is defined as follows: $\mathcal{A} = \mathbf{A} \cup \{[v := e] \mid v \in \mathcal{V} \wedge e \in \mathcal{D}\}$.

5.3 $\text{BPA}_{\delta\epsilon}$ -I with Recursion

In this section, recursion in the setting of $\text{BPA}_{\delta\epsilon}$ -I is treated. A closed $\text{BPA}_{\delta\epsilon}$ -I term of sort \mathbf{P} denotes a process with a finite upper bound to the number of actions that it can perform. Recursion allows the description of processes without a finite upper bound to the number of actions that it can perform.

A *recursive specification* over $\text{BPA}_{\delta\epsilon}$ -I is a set $\{X = t_X \mid X \in V\}$ of *recursion equations*, where V is a subset of \mathcal{X} and each t_X is a $\text{BPA}_{\delta\epsilon}$ -I term of sort \mathbf{P} in which only variables from V occur. We write $\text{vars}(S)$, where S is a recursive specification over $\text{BPA}_{\delta\epsilon}$ -I, for the set of all variables that occur in S .

A *solution* of a recursive specification S over $\text{BPA}_{\delta\epsilon}$ -I in some model of $\text{BPA}_{\delta\epsilon}$ -I is a set $\{p_X \mid X \in \text{vars}(S)\}$ of elements of the carrier of sort \mathbf{P} in that model such that each equation in S holds if, for all $X \in \text{vars}(S)$, X is assigned p_X . If $\{p_X \mid X \in \text{vars}(S)\}$ is a solution of a recursive specification S , then, for each $X \in \text{vars}(S)$, p_X is called the *X-component* of that solution of S . Each recursive specification over $\text{BPA}_{\delta\epsilon}$ -I that has a unique solution in the model of $\text{BPA}_{\delta\epsilon}$ -I given in [20] can be rewritten to a recursive specification in which the right-hand sides of equations are linear $\text{BPA}_{\delta\epsilon}$ -I terms.

The set \mathcal{L} of *linear $\text{BPA}_{\delta\epsilon}$ -I terms* is inductively defined by the following rules:

- $\delta \in \mathcal{L}$;
- if $\phi \in \mathcal{C}$, then $\phi \rightarrow \epsilon \in \mathcal{L}$;
- if $\phi \in \mathcal{C}$, $\alpha \in \mathcal{A}$, and $X \in \mathcal{X}$, then $\phi \rightarrow \alpha \cdot X \in \mathcal{L}$;
- if $t, t' \in \mathcal{L} \setminus \{\delta\}$, then $t + t' \in \mathcal{L}$.

A *linear recursive specification* over $\text{BPA}_{\delta\epsilon}\text{-I}$ is a recursive specification $\{X = t_X \mid X \in V\}$ over $\text{BPA}_{\delta\epsilon}\text{-I}$ where each $t_X \in \mathcal{L}$.

$\text{BPA}_{\delta\epsilon}\text{-I}$ is extended with recursion by adding constants for solutions of linear recursive specifications over $\text{BPA}_{\delta\epsilon}\text{-I}$ and axioms concerning these additional constants. For each linear recursive specification S over $\text{BPA}_{\delta\epsilon}\text{-I}$ and each $X \in \text{vars}(S)$, a constant $\langle X|S \rangle$ of sort \mathbf{P} is added to the constants of $\text{BPA}_{\delta\epsilon}\text{-I}$ and axioms postulating that $\langle X|S \rangle$ stands for the X -component of the unique solution of S are added to the axioms of $\text{BPA}_{\delta\epsilon}\text{-I}$. We write $\text{BPA}_{\delta\epsilon}\text{-I+REC}$ for the resulting theory.

We write \mathcal{P}_{rec} for the set of all closed $\text{BPA}_{\delta\epsilon}\text{-I+REC}$ terms of sort \mathbf{P} . We write $\vdash t = t'$, where t and t' are $\text{BPA}_{\delta\epsilon}\text{-I+REC}$ terms of sort \mathbf{P} , to indicate that the equation $t = t'$ is derivable from the axioms of $\text{BPA}_{\delta\epsilon}\text{-I+REC}$.

6 Algorithm Processes

In this section, a connection is made between proto-algorithms and the processes considered in the imperative process algebra $\text{BPA}_{\delta\epsilon}\text{-I}$. It is assumed that $\mathfrak{m} \in \mathcal{V}$.

Definition. Let $\Sigma = (F, P)$ be an alphabet. Then a Σ -algorithm process is a constant $\langle X|S \rangle$ of $\text{BPA}_{\delta\epsilon}\text{-I}$, where S is finite, $X_\epsilon \in \text{vars}(S)$, and for each $Y \in \text{vars}(S)$:

– the recursion equation for Y in S has one of the following forms:

- (1) $Y = \mathbf{t} \rightarrow [\mathfrak{m} := \text{ini}(\mathfrak{m})] \cdot Z$,
- (2) $Y = \mathbf{t} \rightarrow [\mathfrak{m} := o(\mathfrak{m})] \cdot Z$,
- (3) $Y = (p(\mathfrak{m}) = 1) \rightarrow [\mathfrak{m} := \mathfrak{m}] \cdot Z + (p(\mathfrak{m}) = 0) \rightarrow [\mathfrak{m} := \mathfrak{m}] \cdot Z'$,
- (4) $Y = \mathbf{t} \rightarrow [\mathfrak{m} := \text{fin}(\mathfrak{m})] \cdot X_\epsilon$,
- (5) $Y = \mathbf{t} \rightarrow \epsilon$,

where $o \in \tilde{F}$, $p \in P$, and $Z, Z' \in \text{vars}(S) \setminus \{X_\epsilon\}$;

- the recursion equation for Y in S is of the form (1) iff $Y \equiv X$;
- the recursion equation for Y in S is of the form (5) iff $Y \equiv X_\epsilon$.

We write AlgoGraph_Σ and $\text{AlgoProcess}_\Sigma$, where Σ is an alphabet, for the set of all Σ -algorithm graphs and the set of all Σ -algorithm processes, respectively.

Definition. Let $\Sigma = (F, P)$ be an alphabet. Then the graph-to-process function g2p_Σ is a total function from AlgoGraph_Σ to $\text{AlgoProcess}_\Sigma$ such that, for each Σ -algorithm graph $G = (V, E, L_v, L_e, l, r)$, $\text{g2p}_\Sigma(G) = \langle X|S \rangle$, where $\langle X|S \rangle$ is a Σ -algorithm process such that:

- $X = \mathbf{t} \rightarrow [\mathfrak{m} := \text{ini}(\mathfrak{m})] \cdot X_{v'} \in S$ iff $(r, v') \in E$;
- $X_v = \mathbf{t} \rightarrow [\mathfrak{m} := o(\mathfrak{m})] \cdot X_{v'} \in S$ iff $v \in V$, $l(v) = o$, $o \in \tilde{F}$, $(v, v') \in E$;
- $X_v = p(\mathfrak{m}) = 1 \rightarrow [\mathfrak{m} := \mathfrak{m}] \cdot X_{v'} + p(\mathfrak{m}) = 0 \rightarrow [\mathfrak{m} := \mathfrak{m}] \cdot X_{v''} \in S$
iff $v \in V$, $l(v) = p$, $p \in P$, $(v, v'), (v, v'') \in E$, $l((v, v')) = 1$, $l((v, v'')) = 0$;
- $X_v = \mathbf{t} \rightarrow [\mathfrak{m} := \text{fin}(\mathfrak{m})] \cdot X_\epsilon \in S$ iff $v \in V$, $l(v) = \text{fin}$;
- $X_\epsilon = \mathbf{t} \rightarrow \epsilon$;

where, for all $v \in V$, $X_v \in \mathcal{X}$ and, for all $v' \in V$, $X_v = X_{v'}$ only if $v = v'$.

The function g2p_Σ is uniquely defined up to renaming of variables.

The following theorem tells us that the function g2p_Σ is a bijection from AlgoGraph_Σ to $\text{AlgoProcess}_\Sigma$ up to isomorphism of algorithm graphs and renaming of variables in algorithm processes.

Theorem 3. *Let Σ be an alphabet. Then, for all $\langle X|S \rangle \in \text{AlgoProcess}_\Sigma$, there exists a unique $G \in \text{AlgoGraph}_\Sigma$ up to \cong such that $\langle X|S \rangle$ and $\text{g2p}_\Sigma(G)$ are identical up to consistent renaming of variables.*

Proof. Let $\Sigma = (F, P)$ be an alphabet, and let $\langle X|S \rangle \in \text{AlgoProcess}_\Sigma$. Then we construct a $G = (V, E, L_v, L_e, l, r) \in \text{AlgoGraph}_\Sigma$ from $\langle X|S \rangle$ as follows:

- $V = \text{vars}(S) \setminus \{X_\epsilon\}$;
- E is the set of all $(Y, Z) \in V \times V$ for which there exists an equation in S such that Y is its left-hand side and Z occurs in its right-hand side;
- $L_v = F \cup P$;
- $L_e = \{0, 1\}$;
- l is defined as follows:
 - $l(X) = \text{ini}$;
 - $l(Y) = o$ if $Y = t \rightarrow [m := o(m)] \cdot Z \in S$ for some $Z \in \text{vars}(S)$;
 - $l(Y) = p$ if $Y = (p(m) = 1) \rightarrow [m := m] \cdot Z + (p(m) = 0) \rightarrow [m := m] \cdot Z' \in S$ for some $Z, Z' \in \text{vars}(S)$;
 - $l(Y) = \text{fin}$ if $Y = t \rightarrow [m := \text{fin}(m)] \cdot X_\epsilon \in S$;
 - $l((Y, Z))$ is undefined if $Y = t \rightarrow [m := o(m)] \cdot Z \in S$ for some $o \in F$;
 - $l((Y, Z)) = 1$ if $Y = (p(m) = 1) \rightarrow [m := m] \cdot Z + (p(m) = 0) \rightarrow [m := m] \cdot Z' \in S$ for some $p \in P$ and $Z' \in \text{vars}(S)$;
 - $l((Y, Z)) = 0$ if $Y = (p(m) = 1) \rightarrow [m := m] \cdot Z' + (p(m) = 0) \rightarrow [m := m] \cdot Z \in S$ for some $p \in P$ and $Z' \in \text{vars}(S)$;
- $r = X$.

It is easy to see that $\text{g2p}_\Sigma(G)$ and $\langle X|S \rangle$ are identical up to consistent renaming of variables and that, for all $G' \in \text{AlgoGraph}_\Sigma$, $\text{g2p}_\Sigma(G')$ and $\langle X|S \rangle$ are identical up to consistent renaming of variables only if $G' \cong G$. \square

It is easy to obtain the signature $\Sigma_{\mathfrak{D}}$ and the minimal algebra \mathfrak{D} of signature $\Sigma_{\mathfrak{D}}$ for a given alphabet Σ and a given Σ -interpretation $(D, D_{\text{in}}, D_{\text{out}}, I)$ after the following issues have been addressed: (a) $D \cup D_{\text{in}} \cup D_{\text{out}}$ must be taken as \mathfrak{D} 's carrier of sort \mathbf{D} and consequently the interpretation of each symbol from Σ must be extended to $D \cup D_{\text{in}} \cup D_{\text{out}}$ and (b) each member of D_{in} must be representable by a closed term of sort \mathbf{D} . Any extension of the functions concerned may be chosen here because we comply with the convention to use each of them only if it is known that the value to which it is applied belongs to its original domain. For simplicity, we take all members of D_{in} as constants of sort \mathbf{D} .

Below, we write $[m \mapsto d]$, where d is a member of \mathfrak{D} 's carrier of sort \mathbf{D} , for a fixed but arbitrary $\rho \in \mathcal{V}\text{Val}$ such that $\rho(m) = d$.

The graph-to-process function g2p_Σ allows to characterize the algorithmic step function of a proto-algorithm $A = (\Sigma, G, T)$ in $\text{BPA}_{\delta\epsilon}\text{-I+REC}$.

Lemma 1. *Let $A = (\Sigma, G, \mathcal{I})$ be a proto-algorithm, where $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$ and $G = (V, E, L_v, L_e, l, r) \in \text{AlgoGraph}_\Sigma$, let $\langle X|S \rangle \in \text{AlgoProcess}_\Sigma$ be such that $\langle X|S \rangle = \text{g2p}_\Sigma(G)$. Then, for all $v, v_1, v_2 \in V$, $d, d_1, d_2 \in D$, $d_{\text{in}} \in D_{\text{in}}$, and $d_{\text{out}} \in D_{\text{out}}$:*

$$\begin{aligned} \delta_A^a(d_{\text{in}}) &= (v, d) \quad \text{iff } \vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S \rangle) = [m := d] \cdot \mathbf{V}_{[m \mapsto d]}(\langle X_v|S \rangle), \\ \delta_A^a((v_1, d_1)) &= (v_2, d_2) \quad \text{iff } \vdash \mathbf{V}_{[m \mapsto d_1]}(\langle X_{v_1}|S \rangle) = [m := d_2] \cdot \mathbf{V}_{[m \mapsto d_2]}(\langle X_{v_2}|S \rangle), \\ \delta_A^a((v, d)) &= d_{\text{out}} \quad \text{iff } \vdash \mathbf{V}_{[m \mapsto d]}(\langle X_v|S \rangle) = [m := d_{\text{out}}] \cdot \mathbf{V}_{[m \mapsto d_{\text{out}}]}(\langle X_\epsilon|S \rangle). \end{aligned}$$

Proof. This follows easily from the definition of the algorithmic step function δ_A^a , the definition of the graph-to-process function g2p_Σ , and the axioms of $\text{BPA}_{\delta_\epsilon\text{-I+REC}}$. \square

There exists a sound method for proving algorithmic equivalence of two proto-algorithms $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma, G', \mathcal{I})$ based on the graph-to-process function g2p_Σ .

Theorem 4. *Let $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma, G', \mathcal{I})$ be proto-algorithms, where $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$. Then $A \equiv_a A'$ if, for all $d \in D_{\text{in}}$, $\vdash \mathbf{V}_{[m \mapsto d]}(\text{g2p}_\Sigma(G)) = \mathbf{V}_{[m \mapsto d]}(\text{g2p}_\Sigma(G'))$.*

Proof. Suppose that $G = (V, E, L_v, L_e, l, r)$ and $G' = (V', E', L_v, L_e, l', r')$. Let $\langle X|S \rangle, \langle X'|S' \rangle \in \text{AlgoProcess}_\Sigma$ and let $d_{\text{in}} \in D_{\text{in}}$.

Starting from $\mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S \rangle)$, either there exists an $n \in \mathbb{N}$, such that, for some $v_1, \dots, v_{n+1} \in V$, $d_1, \dots, d_{n+1} \in D$, and $d_{\text{out}} \in D_{\text{out}}$:

$$\begin{aligned} \vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S \rangle) &= [m := d_1] \cdot \mathbf{V}_{[m \mapsto d_1]}(\langle X_{v_1}|S \rangle), \\ \vdash \mathbf{V}_{[m \mapsto d_i]}(\langle X_{v_i}|S \rangle) &= [m := d_{i+1}] \cdot \mathbf{V}_{[m \mapsto d_{i+1}]}(\langle X_{v_{i+1}}|S \rangle) \\ &\quad \text{for each } i \in \{1, \dots, n\}, \\ \vdash \mathbf{V}_{[m \mapsto d_{n+1}]}(\langle X_{v_{n+1}}|S \rangle) &= [m := d_{\text{out}}] \cdot \mathbf{V}_{[m \mapsto d_{\text{out}}]}(\langle X_\epsilon|S \rangle) \end{aligned}$$

or, for some $v_1, v_2, \dots \in V$ and $d_1, d_2, \dots \in D$:

$$\begin{aligned} \vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S \rangle) &= [m := d_1] \cdot \mathbf{V}_{[m \mapsto d_1]}(\langle X_{v_1}|S \rangle), \\ \vdash \mathbf{V}_{[m \mapsto d_i]}(\langle X_{v_i}|S \rangle) &= [m := d_{i+1}] \cdot \mathbf{V}_{[m \mapsto d_{i+1}]}(\langle X_{v_{i+1}}|S \rangle) \\ &\quad \text{for each } i \in \mathbb{N}. \end{aligned}$$

From this, using $\vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\text{g2p}_\Sigma(G)) = \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\text{g2p}_\Sigma(G'))$ and the fact that $\vdash \alpha \cdot t = \alpha' \cdot t'$ (where $\alpha, \alpha' \in \mathcal{A}$ and $t, t' \in \mathcal{P}_{\text{rec}}$) only if $\vdash \alpha = \alpha'$ and $\vdash t = t'$, it follows by an inductive argument that (for $i \in \{1, \dots, n\}$ or $i \in \mathbb{N}$):

$$\begin{aligned} \vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S \rangle) &= [m := d_1] \cdot \mathbf{V}_{[m \mapsto d_1]}(\langle X_{v_1}|S \rangle) \text{ only if} \\ \vdash \mathbf{V}_{[m \mapsto d_{\text{in}}]}(\langle X|S' \rangle) &= [m := d_1] \cdot \mathbf{V}_{[m \mapsto d_1]}(\langle X_{v'_1}|S' \rangle) \text{ for some } v'_1 \in V', \\ \vdash \mathbf{V}_{[m \mapsto d_i]}(\langle X_{v_i}|S \rangle) &= [m := d_{i+1}] \cdot \mathbf{V}_{[m \mapsto d_{i+1}]}(\langle X_{v_{i+1}}|S \rangle) \text{ only if} \\ \vdash \mathbf{V}_{[m \mapsto d_i]}(\langle X_{v'_i}|S' \rangle) &= [m := d_{i+1}] \cdot \mathbf{V}_{[m \mapsto d_{i+1}]}(\langle X_{v'_{i+1}}|S' \rangle) \text{ for some } v'_i, v'_{i+1} \in V', \\ \vdash \mathbf{V}_{[m \mapsto d_{n+1}]}(\langle X_{v_{n+1}}|S \rangle) &= [m := d_{\text{out}}] \cdot \mathbf{V}_{[m \mapsto d_{\text{out}}]}(\langle X_\epsilon|S \rangle) \text{ only if} \\ \vdash \mathbf{V}_{[m \mapsto d_{n+1}]}(\langle X_{v'_{n+1}}|S' \rangle) &= [m := d_{\text{out}}] \cdot \mathbf{V}_{[m \mapsto d_{\text{out}}]}(\langle X_\epsilon|S' \rangle) \text{ for some } v'_{n+1} \in V'. \end{aligned}$$

From this, using Lemma 1, it directly follows that (for $i \in \{1, \dots, n\}$ or $i \in \mathbb{N}$):

$$\begin{aligned} \delta_A^a(d_{\text{in}}) = (v_1, d_1) \text{ only if } \delta_{A'}^a(d_{\text{in}}) = (v'_1, d_1) \text{ for some } v'_1 \in V', \\ \delta_A^a((v_i, d_i)) = (v_{i+1}, d_{i+1}) \text{ only if} \\ \delta_{A'}^a((v'_i, d_i)) = (v'_{i+1}, d_{i+1}) \text{ for some } v'_i, v'_{i+1} \in V', \\ \delta_A^a((v_{n+1}, d_{n+1})) = d_{\text{out}} \text{ only if} \\ \delta_{A'}^a((v'_{n+1}, d_{n+1})) = d_{\text{out}} \text{ for some } v'_{n+1} \in V'. \end{aligned}$$

This means that there exists an algorithmic simulation of A by A' . In the same way, we can show that there exists an algorithmic simulation of A' by A . Hence, $A \equiv_a A'$. \square

We do not have that $A \equiv_a A'$ only if, for all $d \in D_{\text{in}}$, $\vdash V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G)) = V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G'))$. The following example illustrates this. Take proto-algorithms $A = (\Sigma, G, \mathcal{I})$ and $A' = (\Sigma, G', \mathcal{I})$, where $\Sigma = (F, P)$, $G = (V, E, L_v, L_e, l, r)$, $\mathcal{I} = (D, D_{\text{in}}, D_{\text{out}}, I)$, and G' is obtained from G by interchanging the labels of two vertices $v, v' \in V$ for which $(v, v') \in E$, $\text{indegree}(v') = 1$, $l(v), l(v') \in F$ and, for all $d \in D$, $I(l(v))(I(l(v'))(d)) = I(l(v'))(I(l(v))(d))$. This means that two steps, the latter of which is always immediately preceded by the first, and that consist of performing an operation, where the operations in question are independent, are interchanged. It is easy to see that A and A' are algorithmically equivalent. However, because of a different order of certain assignment actions in $\text{g2p}_\Sigma(G)$ and $\text{g2p}_\Sigma(G')$, we do not have that $\vdash V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G)) = V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G'))$.

We also do not have that $A \cong A'$ if, for all $d \in D_{\text{in}}$, $\vdash V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G)) = V_{[m \rightarrow d]}(\text{g2p}_\Sigma(G'))$. This is illustrated by the same example as the one used to illustrate that we do not have that $A \cong A'$ if $A \equiv_a A'$.

7 Discussion of Generalizations

The notion of a proto-algorithm introduced in this paper is based on the classical informal notion of an algorithm. Several generalizations of that notion have been proposed, e.g. the notion of a non-deterministic algorithm, the notion of a parallel algorithm, and the notion of an interactive algorithm.

The generalization of the notion of a proto-algorithm to a notion of a non-deterministic proto-algorithm is easy: weaken, in the definition of a Σ -algorithm graph, the outdegree of vertices labeled with a function symbol other than fin to greater than zero. In the case of a non-deterministic proto-algorithm, most definitions involving one or more proto-algorithms and the definition of a Σ -algorithm process need an obvious adaptation. However, the definition of algorithmic equivalence needs an adaptation that is not obvious at first sight: two non-deterministic proto-algorithms A and A' are algorithmically equivalent if there exist an algorithmic simulation R of A by A' and an algorithmic simulation R' of A' by A such that $R' = R^{-1}$. The condition $R' = R^{-1}$ is necessary to guarantee that A and A' have the same choice structure. With this adaptation, Theorem 4 goes through for non-deterministic proto-algorithms.

The generalization of the notion of a proto-algorithm to a notion of a parallel proto-algorithm is not so easy. The main reason for this is that there is no consensus about what properties are essential for a parallel algorithm. A parallel algorithm is usually informally described by a sentence like “A parallel algorithm is an algorithm in which more than one step can take place simultaneously”. The term parallel algorithm was introduced after the first studies on parallelization of ‘classical’ algorithms (see e.g. [7]). One of the earliest uses of the term in the computer science literature was in [23]. In that paper, a formalization of the notion of a parallel algorithm is given that does not depend on a particular machine model. However, the formalization is far from covering everything that is currently considered a parallel algorithm.

Since the introduction of the first models of parallel computation, it is common practice to identify parallel algorithms with the abstract machines considered in a particular model of parallel computation. Many adjustments of early models based on random access machines have been proposed, in particular of those introduced in [5, 8, 9, 11]. The resulting wide variety of proposed models of parallel computation does not make it easier to come up with a formal notion of a parallel algorithm that encompasses everything considered a parallel algorithm. It therefore seems useful to start with distinguishing different types of parallel algorithms and generalizing the notion of a proto-algorithm to a notion of a parallel algorithm per type of parallel algorithms.

The generalization of the notion of a proto-algorithm to a notion of an interactive proto-algorithm is not so easy too. As with parallel proto-algorithms, the main reason for this is that there is no consensus on which properties are essential for an interactive algorithm. An interactive algorithm is usually informally described by a sentence like “An interactive algorithm is an algorithm that can interact with the environment in which it takes place”. In [4], a specific view on the nature of interactive algorithms is discussed in detail, culminating in a characterization of interactive algorithms by a number of postulates. This view is the only one found in the computer science literature so far. Some of its details are based on choices whose impact on the generality of the characterization is not clear.

Recently, several models of interactive computation have been proposed. They are based on variants of Turing machines, to wit interactive Turing machines [18], persistent Turing machines [10], and reactive Turing machines [2]. These models are closely related. In [2], it is established that reactive Turing machines are at least as expressive as persistent Turing machines. Moreover, it is established in that paper that the behaviour of a reactive Turing machine can be defined by a recursive specification in a process algebra closely related to ACP_ϵ^τ [1, Section 5.3], an extension of $BPA_{\delta\epsilon}$ that includes parallel composition.

8 Concluding Remarks

I have reported on a quest for a satisfactory formalization of the notion of an algorithm. I have introduced the notion of a proto-algorithm. Algorithms

are expected to be equivalence classes of proto-algorithms under an appropriate equivalence relation. I have defined three equivalence relations on proto-algorithms. Two of them give bounds between which an appropriate equivalence relation must lie. The third one, called algorithmic equivalence, lies in between these two and is likely an appropriate one. I have also presented a sound method for proving algorithmic equivalence of two proto-algorithms using the imperative process algebra $\text{BPA}_{\delta\epsilon}\text{-I+REC}$.

The notion of a proto-algorithm defined in this paper does not depend on any particular machine model or algorithmic language, and also has most properties that are generally considered to belong to the important ones of an algorithm. This makes it neither too concrete nor too abstract to be a appropriate basis for investigating what exactly an algorithm is in the setting of emerging types of computation, such as interactive computation.

Due to the connection between proto-algorithms and processes that is expressed by Theorem 4, $\text{ACP}_{\epsilon}^{\tau}\text{-I+REC}$ [20], an extension of $\text{BPA}_{\delta\epsilon}\text{-I+REC}$ that includes among other things parallel composition, is potentially a suitable tool to find out how to generalize the notion of a proto-algorithm to the different types of parallel algorithms.

Early in my career I did research, development and consultancy. Later, the emphasis increasingly shifted to what suits me best, namely research. I cannot formally prove it, but I am convinced that the conversations I had with Cliff Jones in the 1980s and early 1990s about the work we were doing are largely responsible for that. They made me realize that research suits me better than development and consultancy. This paper shows that it suits me so well that I am still doing research long after my retirement.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

Appendix

The axioms of $\text{BPA}_{\delta\epsilon}\text{-I+REC}$ are presented in Table 1. In this table, t stands for an arbitrary term from \mathcal{P} , ϕ and ψ stand for arbitrary terms from \mathcal{C} , e stands for an arbitrary term from \mathcal{D} , a stands for an arbitrary basic action from \mathbf{A} , v stands for an arbitrary flexible variable from \mathcal{V} , ρ stands for an arbitrary flexible variable valuation from $\mathcal{V}\mathcal{V}al$, X stands for an arbitrary variable from \mathcal{X} , S stands for an arbitrary linear recursive specification over $\text{BPA}_{\delta\epsilon}\text{-I}$. The notation $\langle t|S \rangle$ is used in axiom RDP for t with, for all $X \in \text{vars}(S)$, all occurrences of X in t replaced by $\langle X|S \rangle$. The homomorphic extensions of a flexible variable valuation ρ from \mathcal{V} to \mathcal{D} and \mathcal{C} are denoted in axioms V3 and V5 by ρ as well. The notation $\rho\{e/v\}$ is used in axiom V3 for the flexible variable valuation ρ' defined by $\rho'(v') = \rho(v')$ if $v' \neq v$ and $\rho'(v) = e$.

Table 1. Axioms of $BPA_{\delta\epsilon}\text{-I+REC}$

$x + y = y + x$	A1	$\delta \cdot x = \delta$	A7
$(x + y) + z = x + (y + z)$	A2	$x \cdot \epsilon = x$	A8
$x + x = x$	A3	$\epsilon \cdot x = x$	A9
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4		
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$\langle X S \rangle = \langle t S \rangle$	if $X = t \in S$ RDP
$x + \delta = x$	A6	$S \Rightarrow X = \langle X S \rangle$	if $X \in \text{vars}(S)$ RSP
$t : \rightarrow x = x$	GC1	$V_\rho(\epsilon) = \epsilon$	V1
$f : \rightarrow x = \delta$	GC2	$V_\rho(a \cdot x) = a \cdot V_\rho(x)$	V2
$\phi : \rightarrow \delta = \delta$	GC3	$V_\rho([v := e] \cdot x) = [v := \rho(e)] \cdot V_{\rho\{\rho(e)/v\}}(x)$	V3
$\phi : \rightarrow (x + y) = \phi : \rightarrow x + \phi : \rightarrow y$	GC4	$V_\rho(x + y) = V_\rho(x) + V_\rho(y)$	V4
$\phi : \rightarrow x \cdot y = (\phi : \rightarrow x) \cdot y$	GC5	$V_\rho(\phi : \rightarrow y) = \rho(\phi) : \rightarrow V_\rho(x)$	V5
$\phi : \rightarrow (\psi : \rightarrow x) = (\phi \wedge \psi) : \rightarrow x$	GC6	$e = e'$	if $\mathfrak{D} \models e = e'$ IMP1
$(\phi \vee \psi) : \rightarrow x = \phi : \rightarrow x + \psi : \rightarrow x$	GC7	$\phi = \psi$	if $\mathfrak{D} \models \phi \Leftrightarrow \psi$ IMP2

References

- Baeten, J.C.M., Weijland, W.P.: Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990). <https://doi.org/10.1017/CBO9780511624193>
- Baeten, J.C., Luttkik, B., van Tilburg, P.: Reactive Turing machines. *Inf. Comput.* **231**, 143–166 (2013). <https://doi.org/10.1016/j.ic.2013.08.010>
- Bergstra, J.A., Middelburg, C.A.: On algorithmic equivalence of instruction sequences for computing bit string functions. *Fund. Inform.* **138**(4), 411–434 (2015). <https://doi.org/10.3233/FI-2015-1219>
- Blass, A., Gurevich, Y.: Ordinary interactive small-step algorithms, I. *ACM Trans. Comput. Log.* **7**(2), 363–419 (2006). <https://doi.org/10.1145/1131313.1131320>
- Cole, R., Zajicek, O.: The APRAM: incorporating asynchrony into the PRAM model. In: SPAA 1989, pp. 169–178. ACM Press (1989). <https://doi.org/10.1145/72935.72954>
- Dijkstra, E.W.: A Short Introduction to the Art of Programming, EWD, vol. 316. Technische Hogeschool Eindhoven (1971)
- Estrin, G., Turn, R.: Automatic assignment of computations in a variable structure computer system. *IEEE Trans. Electron. Comput.* **EC-12**(6), 755–773 (1963). <https://doi.org/10.1109/PGEC.1963.263559>
- Fortune, S., Wyllie, J.: Parallelism in random access machines. In: STOC 1978, pp. 114–118. ACM Press (1978). <https://doi.org/10.1145/800133.804339>
- Gibbons, P.B.: A more practical PRAM model. In: SPAA 1989, pp. 158–168. ACM Press (1989). <https://doi.org/10.1145/72935.72953>
- Goldin, D.Q., Smolka, S.A., Attie, P.C., Sonderegger, E.L.: Turing machines, transition systems, and interaction. *Inf. Comput.* **194**(2), 101–128 (2004). <https://doi.org/10.1016/j.ic.2004.07.002>
- Goldschlager, L.M.: A unified approach to models of synchronous parallel machines. In: STOC 1978, pp. 89–94. ACM Press (1978). <https://doi.org/10.1145/800133.804336>
- Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.* **1**(1), 77–111 (2000). <https://doi.org/10.1145/343369.343384>

13. Hill, R.K.: What an algorithm is. *Philos. Technol.* **29**(1), 35–59 (2016). <https://doi.org/10.1007/s13347-014-0184-5>
14. Jones, C.B.: *Systematic Software Development Using VDM*, 2nd edn. Prentice-Hall, Hoboken (1990)
15. Kleene, S.C.: *Mathematical Logic*. Wiley, New York (1967)
16. Knuth, D.E.: *The Art of Computer Programming: The Fundamental Algorithms*, 3rd edn. Addison Wesley Longman, Redwood City (1997)
17. Lamport, L.: The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **16**(3), 872–923 (1994). <https://doi.org/10.1145/177492.177726>
18. van Leeuwen, J., Wiedermann, J.: Beyond the turing limit: evolving interactive systems. In: Pacholski, L., Ružička, P. (eds.) *SOFSEM 2001*. LNCS, vol. 2234, pp. 90–109. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45627-9_8
19. Mal'cev, A.I.: *Algorithm and Recursive Functions*. Wolters-Noordhoff, Groningen (1970)
20. Middelburg, C.A.: Imperative process algebra with abstraction. *Sci. Ann. Comput. Sci.* **32**(1), 137–179 (2022). <https://doi.org/10.7561/SACS.2022.1.137>
21. Moschovakis, Y.N., Paschalis, V.: Elementary algorithms and their implementations. In: Cooper, S.B., Löwe, B., Sorbi, A. (eds.) *New Computational Paradigms*, pp. 87–118. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68546-5_5
22. Papayannopoulos, P.: On algorithms, effective procedures, and their definitions. *Philos. Math.* **31**(3), 291–329 (2023). <https://doi.org/10.1093/phimat/nkad011>
23. Reiter, R.: Scheduling parallel computations. *J. ACM* **15**(4), 590–599 (1968). <https://doi.org/10.1145/321479.321485>
24. Rogers, H.: *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York (1967)
25. Schneider, F.B.: *On Concurrent Programming*. Graduate Texts in Computer Science. Springer, New York (1997). <https://doi.org/10.1007/978-1-4612-1830-2>
26. Seaver, N.: Algorithms as culture: some tactics for the ethnography of algorithmic systems. *Big Data Soc.* **4**(2), 1–12 (2017). <https://doi.org/10.1177/2053951717738104>
27. Weyuker, E.J.: Modifications of the program scheme model. *J. Comput. Syst. Sci.* **18**(3), 281–293 (1979). [https://doi.org/10.1016/0022-0000\(79\)90036-9](https://doi.org/10.1016/0022-0000(79)90036-9)