



## UvA-DARE (Digital Academic Repository)

### Imperative Process Algebra and Models of Parallel Computation

Middelburg, C.A.

**DOI**

[10.1007/s00224-024-10164-0](https://doi.org/10.1007/s00224-024-10164-0)

**Publication date**

2024

**Document Version**

Final published version

**Published in**

Theory of Computing Systems

**License**

CC BY

[Link to publication](#)

**Citation for published version (APA):**

Middelburg, C. A. (2024). Imperative Process Algebra and Models of Parallel Computation. *Theory of Computing Systems*, 68(3), 529-570. <https://doi.org/10.1007/s00224-024-10164-0>

**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.



# Imperative Process Algebra and Models of Parallel Computation

Cornelis A. Middelburg<sup>1</sup> 

Accepted: 18 January 2024 / Published online: 14 March 2024  
© The Author(s) 2024

## Abstract

Studies of issues related to computability and computational complexity involve the use of a model of computation. Central in such a model are computational processes. Processes of this kind can be described using an imperative process algebra based on ACP (Algebra of Communicating Processes). In this paper, it is investigated whether the imperative process algebra concerned can play a role in the field of models of computation. It is demonstrated that the process algebra is suitable to describe in a mathematically precise way models of computation corresponding to existing models based on sequential, asynchronous parallel, and synchronous parallel random access machines as well as time and work complexity measures for those models.

**Keywords** Imperative process algebra · Computational process · Parallel random access machine · Parallel time complexity · Parallel computation thesis

## 1 Introduction

Central in a model of computation are computational processes, i.e. processes that solve a computational problem. A computational process is applied to a data environment that consists of data organized and accessible in a specific way. Well-known examples of data environments are the tapes found in Turing machines and the memories found in random access machines. The application of a computational process to a data environment yields another data environment. The data environment to which the process is applied represents an instance of the computational problem that is solved by the process and the data environment yielded by the application represents the solution of that instance. A computational process is divided into simple steps, each of

---

✉ Cornelis A. Middelburg  
C.A.Middelburg@uva.nl; <https://staff.fnwi.uva.nl/c.a.middelburg/>

<sup>1</sup> Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 900, 1098 XH Amsterdam, The Netherlands

which depends on and has an impact on only a small portion of the data environment to which the process is applied.

A basic assumption in this paper is that a model of computation is fully characterized by: (a) a set of possible computational processes, (b) for each possible computational process, a set of possible data environments, and (c) the effect of applying such processes to such environments. The set of possible computational processes is usually given indirectly, mostly by means of abstract machines that have a built-in program that belongs to a set of possible programs which is such that the possible computational processes are exactly the processes that are produced by those machines when they execute their built-in program. The abstract machines with their built-in programs emphasize the mechanical nature of the possible computational processes. However, in this way, details of how possible computational processes are produced become part of the model of computation. To the best of my knowledge, all definitions that have been given with respect to a model of computation and all results that have been proved with respect to a model of computation can do without reference to such details.

In [29], an extension of ACP (Algebra of Communicating Processes) [5] is presented whose additional features include assignment actions to change data in the course of a process, guarded commands to proceed at certain stages of a process in a way that depends on changing data, and data parameterized actions to communicate data between processes. The extension concerned is called  $ACP_{\epsilon}^{\tau}$ -I.

The term imperative process algebra was coined in [13] for process algebras like  $ACP_{\epsilon}^{\tau}$ -I. Some examples of other imperative process algebras are VPLA [22], IPAL [13],  $CSP_{\sigma}$  [11], AWN [15], and the nameless process algebra proposed in [9]. In [29], it is discussed what qualities of  $ACP_{\epsilon}^{\tau}$ -I distinguish it from imperative process algebras such as the above-mentioned ones, how its distinguishing qualities are achieved, and what its relevance is to the verification of properties of processes carried out by contemporary computer-based systems. Moreover, that paper goes into one of the application areas of  $ACP_{\epsilon}^{\tau}$ -I, namely the area of information-flow security analysis.

The current paper studies whether  $ACP_{\epsilon}^{\tau}$ -I can play a role in the field of models of computation. The idea of this study originates from the experience that definitions of models of computation and results about them in the scientific literature tend to lack preciseness, in particular if it concerns models of parallel computation. The study takes for granted the basic assumption about the characterization of models of computation mentioned above. Moreover, it focuses on models of computation that are intended for investigation of issues related to computability and computational complexity. It does not consider models of computation geared to computation as it takes place on concrete computers or computer networks of a certain kind. Such models are left for follow-up studies. Outcomes of this study are among other things mathematically precise definitions of models of computation corresponding to models based on sequential random access machines, asynchronous parallel random access machines, and synchronous parallel random access machines.

This paper is organized as follows. First, a survey of the imperative process algebra  $ACP_{\epsilon}^{\tau}$ -I and its extension with recursion is given (Section 2). Next, it is explained in this process algebra what it means that a given process computes a given function (Section 3). After that, a version of the sequential random access machine model of computation is described in the setting introduced in the previous two sections

(Section 4). Following that, an asynchronous parallel random access machine model of computation and a synchronous parallel random access machine model of computation are described in that setting as well (Sections 5 and 6, respectively). Then, complexity measures for the models of computation presented in the previous three sections are introduced (Section 7). Thereafter, the question whether the presented synchronous parallel random access machine model of computation is a reasonable model of computation is treated (Section 8). Finally, some concluding remarks are made (Section 9).

Section 2 is an abridged version of [29]. Portions of Sections 2–4 of that paper have been copied verbatim or slightly modified.

## 2 The Imperative Process Algebra $ACP_\epsilon^\tau$ -I

The imperative process algebra  $ACP_\epsilon^\tau$ -I is an extension of  $ACP_\epsilon^\tau$ , the version of ACP with empty process and silent step constants that was first presented in [4, Section 5.3]. In this section, first a survey of  $ACP_\epsilon^\tau$  is given and then  $ACP_\epsilon^\tau$ -I is introduced as an extension of  $ACP_\epsilon^\tau$ . Moreover, recursion in the setting of  $ACP_\epsilon^\tau$ -I is treated and some relevant results about  $ACP_\epsilon^\tau$ -I with recursion are presented.

### 2.1 ACP with Empty Process and Silent Step

The survey of  $ACP_\epsilon^\tau$  given in this section is kept brief. A more comprehensive treatment of this process algebra can be found in [4, Section 5.3].

In  $ACP_\epsilon^\tau$ , it is assumed that a fixed but arbitrary finite set  $A$  of *basic actions*, with  $\tau, \delta, \epsilon \notin A$ , and a fixed but arbitrary commutative and associative *communication function*  $\gamma : (A \cup \{\tau, \delta\}) \times (A \cup \{\tau, \delta\}) \rightarrow (A \cup \{\tau, \delta\})$ , such that  $\gamma(\tau, a) = \delta$  and  $\gamma(\delta, a) = \delta$  for all  $a \in A \cup \{\tau, \delta\}$ , have been given. Basic actions are taken as atomic processes. The function  $\gamma$  is regarded to give the result of simultaneously performing any two basic actions for which this is possible, and to be  $\delta$  otherwise. Henceforth, we write  $A_\tau$  for  $A \cup \{\tau\}$  and  $A_{\tau\delta}$  for  $A \cup \{\tau, \delta\}$ .

The algebraic theory  $ACP_\epsilon^\tau$  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  $ACP_\epsilon^\tau$  has the following constants and operators to build terms of sort  $\mathbf{P}$ :

- a *basic action* constant  $a : \mathbf{P}$  for each  $a \in A$ ;
- a *silent step* constant  $\tau : \mathbf{P}$ ;
- 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  $\cdot$  :  $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$ ;
- a binary *parallel composition* or *merge* operator  $\parallel$  :  $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$ ;
- a binary *left merge* operator  $\parallel\!\!\!|$  :  $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$ ;
- a binary *communication merge* operator  $|$  :  $\mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$ ;
- a unary *encapsulation* operator  $\partial_H$  :  $\mathbf{P} \rightarrow \mathbf{P}$  for each  $H \subseteq A$  and for  $H = A_\tau$ ;
- a unary *abstraction* operator  $\tau_I$  :  $\mathbf{P} \rightarrow \mathbf{P}$  for each  $I \subseteq A$ .

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 binary operators. The following precedence conventions are used to reduce the need for parentheses: the operator  $\cdot$  binds stronger than all other binary operators and the operator  $+$  binds weaker than all other binary operators.

The constants  $a$  ( $a \in A$ ),  $\tau$ ,  $\epsilon$ , and  $\delta$  can be explained as follows:

- $a$  denotes the process that first performs the observable action  $a$  and then terminates successfully;
- $\tau$  denotes the process that first performs the unobservable action  $\tau$  and then terminates successfully;
- $\epsilon$  denotes the process that terminates successfully without performing any action;
- $\delta$  denotes the process that cannot do anything, it cannot even terminate successfully.

Let  $t$  and  $t'$  be closed  $\text{ACP}_\epsilon^\tau$  terms denoting processes  $p$  and  $p'$ , respectively. Then the operators  $+$ ,  $\cdot$ ,  $\parallel$ ,  $\partial_H$  ( $H \subseteq A$  or  $H = A_\tau$ ), and  $\tau_I$  ( $I \subseteq A$ ) can be explained as follows:

- $t + t'$  denotes the process that behaves as either  $p$  or  $p'$ ;
- $t \cdot t'$  denotes the process that behaves as  $p$  and  $p'$  in sequence;
- $t \parallel t'$  denotes the process that behaves as  $p$  and  $p'$  in parallel;
- $\partial_H(t)$  denotes the process that behaves as  $p$ , except that actions from  $H$  are blocked from being performed;
- $\tau_I(t)$  denotes the process that behaves as  $p$ , except that actions from  $I$  are turned into the unobservable action  $\tau$ .

Here “behaves as  $p$  and  $p'$  in parallel” means that (a) each time an action is performed, either a next action of  $p$  is performed or a next action of  $p'$  is performed or a next action of  $p$  and a next action of  $p'$  are performed synchronously and (b) successful termination may take place at any time that both  $p$  and  $p'$  can terminate successfully.

The operators  $\parallel$  and  $|$  are of an auxiliary nature. They make a finite axiomatization of  $\text{ACP}_\epsilon^\tau$  possible.

The axioms of  $\text{ACP}_\epsilon^\tau$  are presented in Table 1. In this table,  $a$ ,  $b$ , and  $\alpha$  stand for arbitrary members of  $A_{\tau\delta}$ ,  $H$  stands for an arbitrary subset of  $A$  or the set  $A_\tau$ , and  $I$  stands for an arbitrary subset of  $A$ . So, CM3, CM7, D0–D4, T0–T4, and BE are actually axiom schemas. In this paper, axiom schemas are usually referred to as axioms.

The term  $\partial_{A_\tau}(x) \cdot \partial_{A_\tau}(y)$  occurring in axiom CM1E is needed to handle successful termination in the presence of  $\epsilon$ : it stands for the process that behaves the same as  $\epsilon$  if both  $x$  and  $y$  stand for a process that has the option to behave the same as  $\epsilon$  and it stands for the process that behaves the same as  $\delta$  otherwise. In [4, Section 5.3], the symbol  $\surd$  is used instead of  $\partial_{A_\tau}$ .

Notice that the equation  $\alpha \cdot \delta = \alpha$  would be derivable from the axioms of  $\text{ACP}_\epsilon^\tau$  if operators  $\partial_H$  where  $H \neq A_\tau$  and  $\tau \in H$  were added to  $\text{ACP}_\epsilon^\tau$ .

The notation  $\sum_{i=1}^n t_i$ , where  $n \geq 1$ , is used for right-nested alternative compositions. For each  $n \in \mathbb{N}^+$ ,<sup>1</sup> the term  $\sum_{i=1}^n t_i$  is defined by induction on  $n$  as follows:  $\sum_{i=1}^1 t_i = t_1$  and  $\sum_{i=1}^{n+1} t_i = t_1 + \sum_{i=1}^n t_{i+1}$ . In addition, the convention is used

<sup>1</sup> We write  $\mathbb{N}^+$  for the set  $\{n \in \mathbb{N} \mid n \geq 1\}$  of positive natural numbers.

**Table 1** Axioms of  $ACP_{\epsilon}^{\tau}$

$x + y = y + x$		A1
$(x + y) + z = x + (y + z)$		A2
$x + x = x$		A3
$(x + y) \cdot z = x \cdot z + y \cdot z$		A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		A5
$x + \delta = x$		A6
$\delta \cdot x = \delta$		A7
$x \cdot \epsilon = x$		A8
$\epsilon \cdot x = x$		A9
$x \parallel y = x \parallel y + y \parallel x + x   y + \partial_{A_{\tau}}(x) \cdot \partial_{A_{\tau}}(y)$		CM1E
$\epsilon \parallel x = \delta$		CM2E
$\alpha \cdot x \parallel y = \alpha \cdot (x \parallel y)$		CM3
$(x + y) \parallel z = x \parallel z + y \parallel z$		CM4
$\epsilon   x = \delta$		CM5E
$x   \epsilon = \delta$		CM6E
$a \cdot x   b \cdot y = \gamma(a, b) \cdot (x \parallel y)$		CM7
$(x + y)   z = x   z + y   z$		CM8
$x   (y + z) = x   y + x   z$		CM9
$\partial_H(\epsilon) = \epsilon$		D0
$\partial_H(\alpha) = \alpha$	if $\alpha \notin H$	D1
$\partial_H(\alpha) = \delta$	if $\alpha \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$		D3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$		D4
$\tau_I(\epsilon) = \epsilon$		T0
$\tau_I(\alpha) = \alpha$	if $\alpha \notin I$	T1
$\tau_I(\alpha) = \tau$	if $\alpha \in I$	T2
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$		T3
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$		T4
$\alpha \cdot (\tau \cdot (x + y) + x) = \alpha \cdot (x + y)$		BE

that  $\sum_{i=1}^0 t_i = \delta$ . Moreover, we write  $\partial_a$  and  $\tau_a$ , where  $a \in A$ , for  $\partial_{\{a\}}$  and  $\tau_{\{a\}}$ , respectively.

### 2.2 Imperative $ACP_{\epsilon}^{\tau}$ -I

This section concerns  $ACP_{\epsilon}^{\tau}$ -I, imperative  $ACP_{\epsilon}^{\tau}$ .  $ACP_{\epsilon}^{\tau}$ -I extends  $ACP_{\epsilon}^{\tau}$  with features to communicate data between processes, 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. A more comprehensive treatment of this imperative process algebra than that given in this section can be found in [29], the paper in which  $ACP_{\epsilon}^{\tau}$ -I was first presented.

In  $\text{ACP}_\epsilon^{\tau}$ -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 the signature  $\Sigma_{\mathcal{D}}$  in which the carrier of sort  $\mathbf{B}$  has cardinality 2 and the equation  $0 = 1$  does not hold.

The sort  $\mathbf{B}$  is assumed to be given in order to make it possible for operations to serve as predicates.

In  $\text{ACP}_\epsilon^{\tau}$ -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. The term flexible variable is used for this kind of variables in e.g. [26, 35].

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

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

Flexible variable valuations are intended to provide the data values — which are members of  $\mathcal{D}$ 's carrier of sort  $\mathbf{D}$  — assigned to flexible variables when an  $\text{ACP}_\epsilon^{\tau}$ -I term of sort  $\mathbf{D}$  is evaluated. To fit better in an algebraic setting, they provide closed terms from  $\mathbb{D}$  that denote those data values instead.

Because  $\mathcal{D}$  is a minimal algebra, for each sort  $S$  that is included in  $\Sigma_{\mathcal{D}}$ , each member of  $\mathcal{D}$ 's carrier of sort  $S$  can be represented by a closed term over  $\Sigma_{\mathcal{D}}$  that is of sort  $S$ .

In the rest of this paper, for each sort  $S$  that is included in  $\Sigma_{\mathcal{D}}$ , let  $ct_S$  be a function from  $\mathcal{D}$ 's carrier of sort  $S$  to the set of all closed terms over  $\Sigma_{\mathcal{D}}$  that are of sort  $S$  such that, for each member  $d$  of  $\mathcal{D}$ 's carrier of sort  $S$ , the term  $ct_S(d)$  represents  $d$ . We write  $d$ , where  $d$  is a member of  $\mathcal{D}$ 's carrier of sort  $S$ , for  $ct_S(d)$  if it is clear from the context that  $d$  stands for a closed term of sort  $S$  representing  $d$ .

Flexible variable valuations are used in Sections 4–6 to represent the data environments referred to in Section 1.

Let  $V \subseteq \mathcal{V}$ . Then a *V-indexed data environment* is a function from  $V$  to  $\mathcal{D}$ 's carrier of sort  $\mathbf{D}$ . Let  $\mu$  be a  $V$ -indexed data environment and  $\rho$  be a flexible variable valuation. Then  $\rho$  *represents*  $\mu$  if  $\rho(v) = ct_{\mathbf{D}}(\mu(v))$  for all  $v \in V$ .

Below, the sorts, constants and operators of  $\text{ACP}_\epsilon^{\tau}$ -I are introduced. The operators of  $\text{ACP}_\epsilon^{\tau}$ -I include two variable-binding operators. The formation rules for  $\text{ACP}_\epsilon^{\tau}$ -I terms are the usual ones for the many-sorted case (see e.g. [34, 38]) and in addition the following rule:

- if  $O$  is a variable-binding operator  $O : S_1 \times \dots \times S_n \rightarrow S$  that binds a variable of sort  $S'$ ,  $t_1, \dots, t_n$  are terms of sorts  $S_1, \dots, S_n$ , respectively, and  $X$  is a variable of sort  $S'$ , then  $OX(t_1, \dots, t_n)$  is a term of sort  $S$ .

An extensive formal treatment of the phenomenon of variable-binding operators can be found in [33].

$\text{ACP}_\epsilon^{\tau}$ -I has the following sorts: the sorts included in  $\Sigma_{\mathcal{D}}$ , the sort  $\mathbf{C}$  of *conditions*, and the sort  $\mathbf{P}$  of *processes*.

For each sort  $S$  included in  $\Sigma_{\mathcal{D}}$  other than  $\mathbf{D}$ ,  $\text{ACP}_{\epsilon}^{\tau}$ -I has only the constants and operators included in  $\Sigma_{\mathcal{D}}$  to build terms of sort  $S$ .

$\text{ACP}_{\epsilon}^{\tau}$ -I has, in addition to the constants and operators included in  $\Sigma_{\mathcal{D}}$  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  $\text{ACP}_{\epsilon}^{\tau}$ -I terms of sort  $\mathbf{D}$ .

$\text{ACP}_{\epsilon}^{\tau}$ -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  $= : \mathbf{D} \times \mathbf{D} \rightarrow \mathbf{C}$ ;<sup>2</sup>
- 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  $\Rightarrow : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$ ;
- a unary variable-binding *universal quantification* operator  $\forall : \mathbf{C} \rightarrow \mathbf{C}$  that binds a variable of sort  $\mathbf{D}$ ;
- a unary variable-binding *existential quantification* operator  $\exists : \mathbf{C} \rightarrow \mathbf{C}$  that binds a variable of sort  $\mathbf{D}$ .

We write  $\mathcal{C}$  for the set of all closed  $\text{ACP}_{\epsilon}^{\tau}$ -I terms of sort  $\mathbf{C}$ .

$\text{ACP}_{\epsilon}^{\tau}$ -I has, in addition to the constants and operators of  $\text{ACP}_{\epsilon}^{\tau}$ , the following operators to build terms of sort  $\mathbf{P}$ :

- an  $n$ -ary *data parameterized action* operator  $a : \mathbf{D}^n \rightarrow \mathbf{P}$  for each  $a \in \mathbf{A}$ , for each  $n \in \mathbb{N}$ ;
- 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  $\mathbf{V}_{\rho} : \mathbf{P} \rightarrow \mathbf{P}$  for each  $\rho \in \mathcal{V}\mathcal{A}\mathcal{L}$ .

We write  $\mathcal{P}$  for the set of all closed  $\text{ACP}_{\epsilon}^{\tau}$ -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  $\text{ACP}_{\epsilon}^{\tau}$ -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  $\mathcal{D}$  by taking the flexible variable constants as variables of sort  $\mathbf{D}$ . The 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  $\mathcal{D}$  as a formula.

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

<sup>2</sup> The overloading of  $=$  can be trivially resolved if  $\Sigma_{\mathcal{D}}$  is without overloaded symbols.

Let  $a$  be a basic action from  $\mathbf{A}$ ,  $e_1, \dots, e_n$ , and  $e$  be terms from  $\mathcal{D}$ ,  $\phi$  be a term from  $\mathcal{C}$ , and  $t$  be a term from  $\mathcal{P}$  denoting a process  $p$ . Then the additional operators to build terms of sort  $\mathbf{P}$  can be explained as follows:

- the term  $a(e_1, \dots, e_n)$  denotes the process that first performs the data parameterized action  $a(e_1, \dots, e_n)$  and then terminates successfully;
- 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  $p$  if condition  $\phi$  holds and as  $\delta$  otherwise;
- the term  $V_\rho(t)$  denotes the process that behaves as  $p$ , except that each subterm of  $t$  that belongs to  $\mathcal{D}$  is evaluated using flexible variable valuation  $\rho$  updated according to the assignment actions that have taken place at the point where the subterm is encountered.

Evaluation operators are a variant of state operators (see e.g. [3]).

The following closed  $\text{ACP}_\epsilon^\tau$ -I term is reminiscent of a program that computes the difference  $d$  between two integers  $i$  and  $j$  by subtracting the smaller one from the larger one ( $i, j, d \in \mathcal{V}$ ):<sup>3</sup>

$$[d := i] \cdot ((d \geq j = 1) \rightarrow [d := d - j] + (d \geq j = 0) \rightarrow [d := j - d]) .$$

That is, the final value of  $d$  is the absolute value of the result of subtracting the initial value of  $i$  from the initial value of  $j$ . Let  $\rho$  be a flexible variable valuation such that  $\rho(i) = 11$  and  $\rho(j) = 3$ . Then the following equation can be derived from the axioms of  $\text{ACP}_\epsilon^\tau$ -I given below:

$$V_\rho([d := i] \cdot ((d \geq j = 1) \rightarrow [d := d - j] + (d \geq j = 0) \rightarrow [d := j - d])) = [d := 11] \cdot [d := 8] .$$

This equation shows that in the case where the initial values of  $i$  and  $j$  are 11 and 3 the final value of  $d$  is 8, which is the absolute value of the result of subtracting 11 from 3.

A flexible variable valuation  $\rho$  can be extended homomorphically from flexible variables to  $\text{ACP}_\epsilon^\tau$ -I terms of sort  $\mathbf{D}$  and  $\text{ACP}_\epsilon^\tau$ -I terms of sort  $\mathbf{C}$ . Below, these extensions are denoted by  $\rho$  as well. Moreover, we write  $\rho[v \mapsto e]$  for the flexible variable valuation  $\rho'$  defined by  $\rho'(v') = \rho(v')$  if  $v' \neq v$  and  $\rho'(v) = e$ .

The subsets  $\mathcal{A}^{\text{par}}$ ,  $\mathcal{A}^{\text{ass}}$ , and  $\mathcal{A}$  of  $\mathcal{P}$  referred to below are defined as follows:

$$\begin{aligned} \mathcal{A}^{\text{par}} &= \bigcup_{n \in \mathbb{N}^+} \{a(e_1, \dots, e_n) \mid a \in \mathbf{A} \wedge e_1, \dots, e_n \in \mathcal{D}\} , \\ \mathcal{A}^{\text{ass}} &= \{[v := e] \mid v \in \mathcal{V} \wedge e \in \mathcal{D}\} , \\ \mathcal{A} &= \mathbf{A} \cup \mathcal{A}^{\text{par}} \cup \mathcal{A}^{\text{ass}} . \end{aligned}$$

<sup>3</sup> Here and in the next example, the carrier of  $\mathbf{D}$  is assumed to be the set of all integers. Moreover, the usual integer constants, operators on integers, and predicates on integers are assumed (where operators with result sort  $\mathbf{B}$  serve as predicates).

The elements of  $\mathcal{A}$  are the terms from  $\mathcal{P}$  that denote the processes that are considered to be atomic. Henceforth, we write  $\mathcal{A}_\tau$  for  $\mathcal{A} \cup \{\tau\}$ ,  $\mathcal{A}_\delta$  for  $\mathcal{A} \cup \{\delta\}$ , and  $\mathcal{A}_{\tau\delta}$  for  $\mathcal{A} \cup \{\tau, \delta\}$ .

The axioms of  $ACP_\epsilon^T$ -I are the axioms presented in Tables 1 and 2, where  $\alpha$  stands for an arbitrary term from  $\mathcal{A}_{\tau\delta}$ ,  $H$  stands for an arbitrary subset of  $\mathcal{A}$  or the set  $\mathcal{A}_\tau$ ,  $I$  stands for an arbitrary subset of  $\mathcal{A}$ ,  $e, e_1, e_2, \dots$  and  $e', e'_1, e'_2, \dots$  stand for arbitrary terms from  $\mathcal{D}$ ,  $\phi$  and  $\psi$  stand for arbitrary terms from  $\mathcal{C}$ ,  $v$  stands for an arbitrary flexible variable from  $\mathcal{V}$ , and  $\rho$  stands for an arbitrary flexible variable valuation from  $\mathcal{V}al$ . Moreover,  $a, b$ , and  $c$  stand for arbitrary members of  $\mathcal{A}_{\tau\delta}$  in Table 1 and for arbitrary members of  $\mathcal{A}$  in Table 2.

**Table 2** Additional axioms of  $ACP_\epsilon^T$ -I

$e = e'$	$\text{if } \mathcal{D} \models e = e'$	IMP1
$\phi = \psi$	$\text{if } \mathcal{D} \models \phi \Leftrightarrow \psi$	IMP2
$\mathbf{t} : \rightarrow x = x$		GC1
$\mathbf{f} : \rightarrow x = \delta$		GC2
$\phi : \rightarrow \delta = \delta$		GC3
$\phi : \rightarrow (x + y) = \phi : \rightarrow x + \phi : \rightarrow y$		GC4
$\phi : \rightarrow x \cdot y = (\phi : \rightarrow x) \cdot y$		GC5
$\phi : \rightarrow (\psi : \rightarrow x) = (\phi \wedge \psi) : \rightarrow x$		GC6
$(\phi \vee \psi) : \rightarrow x = \phi : \rightarrow x + \psi : \rightarrow x$		GC7
$(\phi : \rightarrow x) \parallel y = \phi : \rightarrow (x \parallel y)$		GC8
$(\phi : \rightarrow x)   y = \phi : \rightarrow (x   y)$		GC9
$x   (\phi : \rightarrow y) = \phi : \rightarrow (x   y)$		GC10
$\partial_H(\phi : \rightarrow x) = \phi : \rightarrow \partial_H(x)$		GC11
$\tau_I(\phi : \rightarrow x) = \phi : \rightarrow \tau_I(x)$		GC12
$V_\rho(\epsilon) = \epsilon$		V0
$V_\rho(\alpha \cdot x) = \alpha \cdot V_\rho(x)$	$\text{if } \alpha \notin \mathcal{A}^{\text{par}} \cup \mathcal{A}^{\text{ass}}$	V1
$V_\rho(a(e_1, \dots, e_n) \cdot x) = a(\rho(e_1), \dots, \rho(e_n)) \cdot V_\rho(x)$		V2
$V_\rho([v := e] \cdot x) = [v := \rho(e)] \cdot V_{\rho[v \mapsto \rho(e)]}(x)$		V3
$V_\rho(x + y) = V_\rho(x) + V_\rho(y)$		V4
$V_\rho(\phi : \rightarrow y) = \rho(\phi) : \rightarrow V_\rho(x)$		V5
$a(e_1, \dots, e_n) \cdot x   b(e'_1, \dots, e'_n) \cdot y =$ $(e_1 = e'_1 \wedge \dots \wedge e_n = e'_n) : \rightarrow c(e_1, \dots, e_n) \cdot (x \parallel y)$	$\text{if } \gamma(a, b) = c$	CM7Da
$a(e_1, \dots, e_n) \cdot x   b(e'_1, \dots, e'_m) \cdot y = \delta$	$\text{if } \gamma(a, b) = \delta \text{ or } n \neq m$	CM7Db
$a(e_1, \dots, e_n) \cdot x   \alpha \cdot y = \delta$	$\text{if } \alpha \notin \mathcal{A}^{\text{par}}$	CM7Dc
$\alpha \cdot x   a(e_1, \dots, e_n) \cdot y = \delta$	$\text{if } \alpha \notin \mathcal{A}^{\text{par}}$	CM7Dd
$[v := e] \cdot x   \alpha \cdot y = \delta$		CM7De
$\alpha \cdot x   [v := e] \cdot y = \delta$		CM7Df
$\alpha \cdot (\phi : \rightarrow \tau \cdot (x + y) + \phi : \rightarrow x) = \alpha \cdot (\phi : \rightarrow (x + y))$		BED

### 2.3 $ACP_{\epsilon}^{\tau}$ -I with Recursion

In this section, recursion in the setting of  $ACP_{\epsilon}^{\tau}$ -I is treated. A closed  $ACP_{\epsilon}^{\tau}$ -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  $ACP_{\epsilon}^{\tau}$ -I is a set  $\{X_i = t_i \mid i \in I\}$ , where  $I$  is a set, each  $X_i$  is a variable from  $\mathcal{X}$ , each  $t_i$  is a  $ACP_{\epsilon}^{\tau}$ -I term of sort  $\mathbf{P}$  in which only variables from  $\{X_i \mid i \in I\}$  occur, and  $X_i \neq X_j$  for all  $i, j \in I$  with  $i \neq j$ . We write  $\text{vars}(E)$ , where  $E$  is a recursive specification over  $ACP_{\epsilon}^{\tau}$ -I, for the set of all variables that occur in  $E$ . Let  $E$  be a recursive specification and let  $X \in \text{vars}(E)$ . Then there exists a unique equation in  $E$  whose left-hand side is  $X$ . This equation is called the *recursion equation for  $X$  in  $E$* .

Below, guarded linear recursive specifications over  $ACP_{\epsilon}^{\tau}$ -I are introduced. The set  $\mathcal{L}$  of *linear  $ACP_{\epsilon}^{\tau}$ -I terms* is inductively defined by the following rules:

1.  $\delta \in \mathcal{L}$ ;
2. if  $\phi \in \mathcal{C}$ , then  $\phi \rightarrow \epsilon \in \mathcal{L}$ ;
3. if  $\phi \in \mathcal{C}$ ,  $\alpha \in \mathcal{A}_{\tau}$ , and  $X \in \mathcal{X}$ , then  $\phi \rightarrow \alpha \cdot X \in \mathcal{L}$ ;
4. if  $t, t' \in \mathcal{L} \setminus \{\delta\}$ , then  $t + t' \in \mathcal{L}$ .

Let  $t \in \mathcal{L}$ . Then we refer to the subterms of  $t$  that have the form  $\phi \rightarrow \epsilon$  or the form  $\phi \rightarrow \alpha \cdot X$  as the *summands of  $t$* .

Let  $X$  be a variable from  $\mathcal{X}$  and let  $t$  be an  $ACP_{\epsilon}^{\tau}$ -I term in which  $X$  occurs. Then an occurrence of  $X$  in  $t$  is *guarded* if  $t$  has a subterm of the form  $\alpha \cdot t'$  where  $\alpha \in \mathcal{A}$  and  $t'$  contains this occurrence of  $X$ .

An occurrence of a variable  $X$  in a linear  $ACP_{\epsilon}^{\tau}$ -I term may be not guarded because a linear  $ACP_{\epsilon}^{\tau}$ -I term may have summands of the form  $\phi \rightarrow \tau \cdot X$ .

A *guarded linear recursive specification* over  $ACP_{\epsilon}^{\tau}$ -I is a recursive specification  $\{X_i = t_i \mid i \in I\}$  over  $ACP_{\epsilon}^{\tau}$ -I where each  $t_i$  is a linear  $ACP_{\epsilon}^{\tau}$ -I term and there does not exist an infinite sequence  $i_0 i_1 \dots$  over  $I$  such that, for each  $k \in \mathbb{N}$ , there is an occurrence of  $X_{i_{k+1}}$  in  $t_{i_k}$  that is not guarded.

A solution of a guarded linear recursive specification  $E$  over  $ACP_{\epsilon}^{\tau}$ -I in some model of  $ACP_{\epsilon}^{\tau}$ -I is a set  $\{p_X \mid X \in \text{vars}(E)\}$  of elements of the carrier of sort  $\mathbf{P}$  in that model such that each equation in  $E$  holds if, for all  $X \in \text{vars}(E)$ ,  $X$  is assigned  $p_X$ . A guarded linear recursive specification has a unique solution under the equivalence defined in [29] for  $ACP_{\epsilon}^{\tau}$ -I extended with guarded linear recursion. If  $\{p_X \mid X \in \text{vars}(E)\}$  is the unique solution of a guarded linear recursive specification  $E$ , then, for each  $X \in \text{vars}(E)$ ,  $p_X$  is called the  *$X$ -component* of the unique solution of  $E$ .

$ACP_{\epsilon}^{\tau}$ -I is extended with guarded linear recursion by adding constants for solutions of guarded linear recursive specifications over  $ACP_{\epsilon}^{\tau}$ -I and axioms concerning these additional constants. For each guarded linear recursive specification  $E$  over  $ACP_{\epsilon}^{\tau}$ -I and each  $X \in \text{vars}(E)$ , a constant  $\langle X|E \rangle$  of sort  $\mathbf{P}$ , that stands for the  $X$ -component of the unique solution of  $E$ , is added to the constants of  $ACP_{\epsilon}^{\tau}$ -I. The equation RDP and the conditional equation RSP given in Table 3 are added to the axioms of  $ACP_{\epsilon}^{\tau}$ -I. In this table,  $X$  stands for an arbitrary variable from  $\mathcal{X}$ ,  $t$  stands for an arbitrary  $ACP_{\epsilon}^{\tau}$ -I

**Table 3** Axioms for guarded linear recursion

$\langle X E \rangle = \langle t E \rangle$	if $X = t \in E$	RDP
$E \Rightarrow X = \langle X E \rangle$	if $X \in \text{vars}(E)$	RSP

term of sort  $\mathbf{P}$ ,  $E$  stands for an arbitrary guarded linear recursive specification over  $\text{ACP}_\epsilon^\tau\text{-I}$ , and the notation  $\langle t|E \rangle$  is used for  $t$  with, for all  $X \in \text{vars}(E)$ , all occurrences of  $X$  in  $t$  replaced by  $\langle X|E \rangle$ . Side conditions restrict what  $X$ ,  $t$  and  $E$  stand for.

We write  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$  for the resulting theory. Furthermore, we write  $\mathcal{P}_{\text{rec}}$  for the set of all closed  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$  terms of sort  $\mathbf{P}$ .

RDP and RSP together postulate that guarded linear recursive specifications over  $\text{ACP}_\epsilon^\tau\text{-I}$  have unique solutions.

Because RSP introduces conditional equations in  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$ , it is understood that conditional equational logic is used in deriving equations from the axioms of  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$ . A complete inference system for conditional equational logic can for example be found in [4, 18].

The following closed  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$  term is reminiscent of a program that computes by repeated subtraction the quotient  $q$  and remainder  $r$  of dividing a non-negative integer  $i$  by a positive integer  $j$  ( $i, j, q, r \in \mathcal{V}$ ):

$$[q := 0] \cdot [r := i] \cdot \langle Q|E \rangle ,$$

where  $E$  is the guarded linear recursive specification that consists of the following two equations ( $Q, R \in \mathcal{X}$ ):

$$\begin{aligned} Q &= (r \geq j = 1) \rightarrow [q := q + 1] \cdot R + (r \geq j = 0) \rightarrow \epsilon , \\ R &= \mathbf{t} \rightarrow [r := r - j] \cdot Q . \end{aligned}$$

Let  $\rho$  be a flexible variable valuation such that  $\rho(i) = 11$  and  $\rho(j) = 3$ . Then the following equation can be derived from the axioms of  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$ :

$$\begin{aligned} &V_\rho([q := 0] \cdot [r := i] \cdot \langle Q|E \rangle) \\ &= [q := 0] \cdot [r := 11] \cdot [q := 1] \cdot [r := 8] \cdot [q := 2] \cdot [r := 5] \cdot [q := 3] \cdot [r := 2] . \end{aligned}$$

This equation shows that in the case where the initial values of  $i$  and  $j$  are 11 and 3 the final values of  $q$  and  $r$  are 3 and 2, which are the quotient and remainder of dividing 11 by 3.

In [29], an equational axiom schema CFAR (Cluster Fair Abstraction Rule) is added to  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$ . CFAR expresses that every cluster of  $\tau$  actions will be exited sooner or later. This is a fairness assumption made in the verification of many properties concerning the external behaviour of systems. We write  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC} + \text{CFAR}$  for the theory  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$  extended with CFAR.

We write  $T \vdash t = t'$ , where  $T$  is  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC}$  or  $\text{ACP}_\epsilon^\tau\text{-I} + \text{REC} + \text{CFAR}$ , to indicate that the equation  $t = t'$  is derivable from the axioms of  $T$  using a complete inference system for conditional equational logic.

## 2.4 Results about $ACP_{\epsilon}^{\tau}$ -I with Recursion

In [29], a structural operational semantics of  $ACP_{\epsilon}^{\tau}$ -I+REC is presented and an equivalence relation  $\Leftrightarrow_{rb}$  on  $\mathcal{P}_{rec}$  based on this structural operational semantics is defined. This equivalence relation reflects the idea that two processes are equivalent if they can simulate each other insofar as their observable potentials to make transitions by performing actions and to terminate successfully are concerned, taking into account the assignments of data values to flexible variables under which the potentials are available.

Soundness and semi-completeness results for the axiom system of  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR with respect to  $\Leftrightarrow_{rb}$  are proved in [29]. The axiom system of  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR is incomplete with respect to  $\Leftrightarrow_{rb}$  for equations between terms from  $\mathcal{P}_{rec}$  and there is no straightforward way to rectify this. However, the axiom system of  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR is complete with respect to  $\Leftrightarrow_{rb}$  for equations between abstraction-free terms from  $\mathcal{P}_{rec}$  and for equations between bool-conditional terms from  $\mathcal{P}_{rec}$ .

A term  $t \in \mathcal{P}_{rec}$  is called *abstraction-free* if no abstraction operator occurs in  $t$ . A term  $t \in \mathcal{P}_{rec}$  is called *bool-conditional* if, for each  $\phi \in \mathcal{C}$  that occurs in  $t$ ,  $\mathcal{D} \models \phi \Leftrightarrow t$  or  $\mathcal{D} \models \phi \Leftrightarrow f$ .

The following auxiliary results about abstraction-free terms and bool-conditional terms are proved in [29]:

- for all abstraction-free  $t \in \mathcal{P}_{rec}$ , there exists a guarded linear recursive specification  $E$  and  $X \in \text{vars}(E)$  such that  $ACP_{\epsilon}^{\tau}$ -I+REC  $\vdash t = \langle X|E \rangle$ ;
- for all bool-conditional  $t \in \mathcal{P}_{rec}$ , there exists a guarded linear recursive specification  $E$  and  $X \in \text{vars}(E)$  such that  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR  $\vdash t = \langle X|E \rangle$ .

The soundness and semi-completeness results referred to above are as follows:

- for all terms  $t, t' \in \mathcal{P}_{rec}$ ,  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR  $\vdash t = t'$  only if  $t \Leftrightarrow_{rb} t'$ ;
- for all abstraction-free  $t, t' \in \mathcal{P}_{rec}$ ,  $ACP_{\epsilon}^{\tau}$ -I+REC  $\vdash t = t'$  if  $t \Leftrightarrow_{rb} t'$ ;
- for all bool-conditional  $t, t' \in \mathcal{P}_{rec}$ ,  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR  $\vdash t = t'$  if  $t \Leftrightarrow_{rb} t'$ .

For a better understanding of the evaluation operators, some results about these rather unfamiliar operators are given below.

The following lemma tells us that a closed term of the form  $V_{\rho}(t)$  equals a bool-conditional closed term.

**Lemma 1** *For all  $t \in \mathcal{P}_{rec}$  and  $\rho \in \mathcal{V}\mathcal{A}\mathcal{l}$ , there exists a bool-conditional  $t' \in \mathcal{P}_{rec}$  such that  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR  $\vdash V_{\rho}(t) = t'$ .*

**Proof** This is straightforwardly proved by induction on the length of  $t$ , case distinction on the structure of  $t$ , and in the case of the constants for solutions of guarded linear recursive specifications additionally by induction on the structure of the right-hand side of a recursion equation.  $\square$

As a corollary of Lemma 1 and the soundness and completeness result mentioned above, we have:

for all  $t, t' \in \mathcal{P}_{rec}$  and  $\rho \in \mathcal{V}\mathcal{A}\mathcal{l}$ ,  $ACP_{\epsilon}^{\tau}$ -I+REC+CFAR  $\vdash V_{\rho}(t) = V_{\rho}(t')$  iff  $V_{\rho}(t) \Leftrightarrow_{rb} V_{\rho}(t')$ .

Below, an elimination theorem for closed terms of the form  $V_\rho(t)$  is presented. In preparation, the subsets  $\mathcal{B}$  and  $\mathcal{B}_{cf}$  of  $\mathcal{P}$  are introduced.

The set  $\mathcal{B}$  of *basic*  $ACP_\epsilon^r$ -I terms is inductively defined by the following rules:

1.  $\delta \in \mathcal{B}$ ;
2. if  $\phi \in \mathcal{C}$ , then  $\phi \mapsto \epsilon \in \mathcal{B}$ ;
3. if  $\phi \in \mathcal{C}$ ,  $\alpha \in \mathcal{A}_\tau$ , and  $t \in \mathcal{B}$ , then  $\phi \mapsto \alpha \cdot t \in \mathcal{B}$ ;
4. if  $t, t' \in \mathcal{B} \setminus \{\delta\}$ , then  $t + t' \in \mathcal{B}$ .

The set  $\mathcal{B}_{cf}$  of *condition-free basic*  $ACP_\epsilon^r$ -I terms is inductively defined by the following rules:

1.  $\delta \in \mathcal{B}_{cf}$ ;
2.  $\epsilon \in \mathcal{B}_{cf}$ ;
3. if  $\alpha \in \mathcal{A}_\tau$ , and  $t \in \mathcal{B}_{cf}$ , then  $\alpha \cdot t \in \mathcal{B}_{cf}$ ;
4. if  $t, t' \in \mathcal{B}_{cf} \setminus \{\delta\}$ , then  $t + t' \in \mathcal{B}_{cf}$ .

**Lemma 2** For all *bool-conditional*  $t \in \mathcal{P}$ , there exists a *bool-conditional*  $t' \in \mathcal{B}$  such that  $ACP_\epsilon^r$ -I  $\vdash t = t'$ .

**Proof** This is straightforwardly proved by induction on the length of  $t$  and case distinction on the structure of  $t$ . □

**Lemma 3** For all *bool-conditional*  $t \in \mathcal{B}$ , there exists a  $t' \in \mathcal{B}_{cf}$  such that  $ACP_\epsilon^r$ -I  $\vdash t = t'$ .

**Proof** This is easily proved by induction on the structure of  $t$ . □

A term  $t \in \mathcal{P}_{rec}$  is called a *finite-process term* if there exists a term  $t' \in \mathcal{P}$  such that  $ACP_\epsilon^r$ -I+REC+CFAR  $\vdash t = t'$ .

The following theorem tells us that a finite-process term of the form  $V_\rho(t)$  equals a condition-free basic term.

**Theorem 1** For all  $t \in \mathcal{P}_{rec}$  and  $\rho \in \mathcal{V}\mathcal{A}l$  for which  $V_\rho(t)$  is a *finite-process term*, there exists a  $t' \in \mathcal{B}_{cf}$  such that  $ACP_\epsilon^r$ -I+REC+CFAR  $\vdash V_\rho(t) = t'$ .

**Proof** This follows immediately from Lemmas 1, 2, and 3. □

The terms from  $\mathcal{B}_{cf}$  are reminiscent of computation trees. In Section 3, use is made of the fact that each finite-process term of the form  $V_\rho(t)$  equals such a term.

Not every term from  $\mathcal{B}_{cf}$  corresponds to a computation tree of which each path represents a computation that eventually halts, not even when it concerns a computation tree with a single path.

A term  $t \in \mathcal{P}_{rec}$  is called a *terminating-process term* if there exists a term  $t' \in \mathcal{B}_{cf}$  such that  $ACP_\epsilon^r$ -I+REC+CFAR  $\vdash t = t'$  and  $t'$  can be formed by applying only the formation rules 2, 3, and 4 of  $\mathcal{B}_{cf}$ .

**Table 4** Axioms for the projection operators

$\pi_n(\epsilon) = \epsilon$	PR1
$\pi_0(\alpha \cdot x) = \epsilon$	PR2
$\pi_{n+1}(\alpha \cdot x) = \alpha \cdot \pi_n(x)$	PR3
$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	PR4
$\pi_n(\phi : \rightarrow x) = \phi : \rightarrow \pi_n(x)$	PR5
$\pi_n(\tau \cdot x) = \tau \cdot \pi_n(x)$	PR6

### 2.5 Extensions

This section concerns two extensions of  $ACP_\epsilon^T$ -I that are relevant to this paper, namely an extension with projection and an extension with action renaming. It is not unusual to come across these extensions in applications of ACP-style process algebras. The first extension is treated here because projections can be used to determine the maximum number of actions that a finite process can perform. The second extension is treated here because action renaming enables to easily define the synchronous variant of the parallel composition operator of  $ACP_\epsilon^T$ -I needed later in this paper.

$ACP_\epsilon^T$ -I,  $ACP_\epsilon^T$ -I+REC, and  $ACP_\epsilon^T$ -I+REC+CFAR can be extended with projection by adding, for each  $n \in \mathbb{N}$ , a unary *projection* operator  $\pi_n : \mathbf{P} \rightarrow \mathbf{P}$  to the operators of  $T$  and adding the axioms given in Table 4 to the axioms of  $T$ . In this table,  $n$  stands for an arbitrary natural number,  $\alpha$  stands for an arbitrary term from  $\mathcal{A}_\delta$ , and  $\phi$  stands for an arbitrary term from  $\mathcal{C}$ .

Let  $t$  be a closed term of the extended theory. Then the projection operator  $\pi_n$  can be explained as follows:  $\pi_n(t)$  denotes the process that behaves the same as the process denoted by  $t$  except that it terminates successfully after  $n$  actions have been performed.

Let  $T$  be  $ACP_\epsilon^T$ -I,  $ACP_\epsilon^T$ -I+REC or  $ACP_\epsilon^T$ -I+REC+CFAR. Then we write  $T$ +PR for  $T$  extended with the projection operators  $\pi_n$  and the axioms PR1–PR6 from Table 4.

$ACP_\epsilon^T$ -I,  $ACP_\epsilon^T$ -I+REC, and  $ACP_\epsilon^T$ -I+REC+CFAR can be extended with action renaming by adding, for each function  $f : \mathcal{A} \rightarrow \mathcal{A}$  such that  $f(\alpha) = \alpha$  for all  $\alpha \in \mathcal{A}^{ass}$ , a unary *action renaming* operator  $\rho_f : \mathbf{P} \rightarrow \mathbf{P}$  to the operators of  $T$  and adding the axioms given in Table 5 to the axioms of  $T$ . In this table,  $f$  stands for an

**Table 5** Axioms for the action renaming operators

$\rho_f(\epsilon) = \epsilon$	RN1
$\rho_f(\delta) = \delta$	RN2
$\rho_f(\alpha) = f(\alpha)$	RN3
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	RN4
$\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y)$	RN5
$\rho_f(\phi : \rightarrow x) = \phi : \rightarrow \rho_f(y)$	RN6
$\rho_f(\tau) = \tau$	RN7

arbitrary function  $f : \mathcal{A} \rightarrow \mathcal{A}$  such that  $f(\alpha) = \alpha$  for all  $\alpha \in \mathcal{A}^{\text{ass}}$ ,  $\alpha$  stands for an arbitrary term from  $\mathcal{A}$ , and  $\phi$  stands for an arbitrary term from  $\mathcal{C}$ .

Let  $t$  be a closed term of the extended theory. Then the action renaming operator  $\rho_f$  can be explained as follows:  $\rho_f(t)$  denotes the process that behaves the same as the process denoted by  $t$  except that, where the latter process performs an action  $\alpha$ , the former process performs the action  $f(\alpha)$ .

Let  $T$  be  $\text{ACP}_\epsilon^{\tau}\text{-I}$ ,  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC}$ ,  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC} + \text{CFAR}$  or  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC} + \text{CFAR} + \text{PR}$ . Then we write  $T + \text{RN}$  for  $T$  extended with the action renaming operators  $\rho_f$  and the axioms RN1–RN7 from Table 5.

### 3 Computation and the RAM Conditions

In order to investigate whether  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC}$  can play a role in the field of models of computation, it has to be explained in the setting of  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC}$  what it means that a given process computes a given function. This requires that assumptions about  $\mathfrak{D}$  have to be made. The assumptions concerned are given in this section. They are based on the idea that the data environment of a computational process consists of one or more RAM (Random Access Machine) memories. Because the assumptions amount to conditions to be satisfied by  $\mathfrak{D}$ , they are called the RAM conditions on  $\mathfrak{D}$ . It is also made precise in this section what it means, in the setting of  $\text{ACP}_\epsilon^{\tau}\text{-I} + \text{REC}$  where  $\mathfrak{D}$  satisfies the RAM conditions, that a given process computes a given partial function from  $(\{0, 1\}^*)^n$  to  $\{0, 1\}^*$  ( $n \in \mathbb{N}$ ).

#### 3.1 The RAM Conditions

The memory of a RAM consists of a countably infinite number of registers which are numbered by natural numbers. Each register is capable of containing a bit string of arbitrary length. The contents of the registers constitute the state of the memory of the RAM. The execution of an instruction by the RAM amounts to carrying out an operation on its memory state that changes the content of at most one register or to testing a property of its memory state. The RAM conditions are presented in this section using the notions of a RAM memory state, a RAM operation, and a RAM property.

A *RAM memory state* is a function  $\sigma : \mathbb{N} \rightarrow \{0, 1\}^*$  that satisfies the condition that there exists an  $i \in \mathbb{N}$  such that, for all  $j \in \mathbb{N}$ ,  $\sigma(i + j) = \lambda$ .<sup>4</sup> We write  $\Sigma_{\text{ram}}$  for the set of all RAM memory states.

Let  $\sigma$  be a RAM memory state. Then, for all  $i \in \mathbb{N}$ ,  $\sigma(i)$  is the content of the register with number  $i$  in memory state  $\sigma$ . The condition on  $\sigma$  expresses that the part of the memory that is actually in use remains finite.

<sup>4</sup> We write  $\lambda$  for the empty bit string.

The *input region* and *output region* of a function  $o : \Sigma_{\text{ram}} \rightarrow \Sigma_{\text{ram}}$ , written  $IR(o)$  and  $OR(o)$ , respectively, are the subsets of  $\mathbb{N}$  defined as follows:

$$\begin{aligned} OR(o) &= \{i \in \mathbb{N} \mid \exists \sigma \in \Sigma_{\text{ram}} \bullet \sigma(i) \neq o(\sigma)(i)\}, \\ IR(o) &= \{i \in \mathbb{N} \mid \exists \sigma_1, \sigma_2 \in \Sigma_{\text{ram}} \bullet (\forall j \in \mathbb{N} \setminus \{i\} \bullet \sigma_1(j) = \sigma_2(j) \wedge \\ &\quad \exists j \in OR(o) \bullet o(\sigma_1)(j) \neq o(\sigma_2)(j))\}. \end{aligned}$$

Let  $o : \Sigma_{\text{ram}} \rightarrow \Sigma_{\text{ram}}$ . Then  $OR(o)$  consists of the numbers of all registers that can be affected by  $o$ ; and  $IR(o)$  consists of the numbers of all registers that can affect the registers whose numbers are in  $OR(o)$  under  $o$ .

A *basic RAM operation* is a function  $o : \Sigma_{\text{ram}} \rightarrow \Sigma_{\text{ram}}$  that satisfies the condition that  $IR(o)$  is finite and  $OR(o)$  has cardinality 0 or 1. We write  $O_{\text{ram}}$  for the set of all basic RAM operations.

Let  $o$  be a basic RAM operation and  $\sigma$  be a RAM memory state. Then carrying out  $o$  on a RAM memory in state  $\sigma$  changes the state of the RAM memory into  $o(\sigma)$ . The condition on  $o$  expresses that the content of at most one register can be affected and that, if there is such a register, only a finite number of registers can affect it.

The following theorem states that each basic RAM operation transforms states of a RAM memory that coincide on its input region to states that coincide on its output region.

**Theorem 2** *Let  $\sigma_1, \sigma_2 \in \Sigma_{\text{ram}}$  and  $o \in O_{\text{ram}}$ . Then  $\sigma_1 \upharpoonright IR(o) = \sigma_2 \upharpoonright IR(o)$  implies  $o(\sigma_1) \upharpoonright OR(o) = o(\sigma_2) \upharpoonright OR(o)$ .<sup>5</sup>*

**Proof** It is easy to see that the 4-tuple  $(\mathbb{N}, \{0, 1\}^*, \Sigma_{\text{ram}}, O_{\text{ram}})$  is a computer according to Definition 3.1 from [28]. From this and Theorem 3.1 from [28], the theorem follows immediately.  $\square$

The *input region* of a function  $p : \Sigma_{\text{ram}} \rightarrow \{0, 1\}$ , written  $IR(p)$  is the subset of  $\mathbb{N}$  defined as follows:

$$IR(p) = \{i \in \mathbb{N} \mid \exists \sigma_1, \sigma_2 \in \Sigma_{\text{ram}} \bullet (\forall j \in \mathbb{N} \setminus \{i\} \bullet \sigma_1(j) = \sigma_2(j) \wedge p(\sigma_1) \neq p(\sigma_2))\}.$$

Let  $p : \Sigma_{\text{ram}} \rightarrow \{0, 1\}$ . Then  $IR(p)$  consists of the numbers of all registers that can affect what the value of  $p$  is.

A *basic RAM property* is a function  $p : \Sigma_{\text{ram}} \rightarrow \{0, 1\}$  that satisfies the condition that  $IR(p)$  is finite. We write  $P_{\text{ram}}$  for the set of all basic RAM properties.

Let  $p$  be a basic RAM property and  $\sigma$  be a RAM memory state. Then testing the property  $p$  on a RAM memory in state  $\sigma$  yields the value  $p(\sigma)$  and does not change the state of the RAM memory. The condition on  $p$  expresses that only a finite number of registers can affect what this value is. We say that  $p$  *holds in  $\sigma$*  if  $p(\sigma) = 1$ .

The following theorem states that each basic RAM property holds in some state of a RAM memory if and only if it holds in all states of the RAM memory that coincide with that state on its input region.

<sup>5</sup> We use the notation  $f \upharpoonright D$ , where  $f$  is a function and  $D \subseteq \text{dom}(f)$ , for the function  $g$  with  $\text{dom}(g) = D$  such that for all  $d \in \text{dom}(g)$ ,  $g(d) = f(d)$ .

**Theorem 3** *Let  $\sigma_1, \sigma_2 \in \Sigma_{\text{ram}}$  and  $p \in P_{\text{ram}}$ . Then  $\sigma_1 \upharpoonright IR(p) = \sigma_2 \upharpoonright IR(p)$  implies  $p(\sigma_1) = p(\sigma_2)$ .*

**Proof** Let  $\Sigma'_{\text{ram}}$  be the set of all functions  $\sigma : \mathbb{N} \cup \{-1\} \rightarrow \{0, 1\}^*$  that satisfy the condition that there exists an  $i \in \mathbb{N}$  such that, for all  $j \in \mathbb{N}$ ,  $\sigma(i + j) = \lambda$ , let  $O'_{\text{ram}}$  be the set of all functions  $o : \Sigma'_{\text{ram}} \rightarrow \Sigma'_{\text{ram}}$ , let  $\sigma'_1, \sigma'_2 \in \Sigma'_{\text{ram}}$  be such that  $\sigma'_1 \upharpoonright \mathbb{N} = \sigma_1$  and  $\sigma'_2 \upharpoonright \mathbb{N} = \sigma_2$ , and let  $o \in O'_{\text{ram}}$  be such that, for all  $\sigma \in \Sigma'_{\text{ram}}$ ,  $o(\sigma) \upharpoonright \mathbb{N} = \sigma \upharpoonright \mathbb{N}$  and  $o(\sigma)(-1) = p(\sigma)$ . Then  $\sigma_1 \upharpoonright IR(p) = \sigma_2 \upharpoonright IR(p)$  implies  $p(\sigma_1) = p(\sigma_2)$  iff  $\sigma'_1 \upharpoonright IR(o) = \sigma'_2 \upharpoonright IR(o)$  implies  $o(\sigma'_1) = o(\sigma'_2)$ . Because of this and the fact that  $(\mathbb{N} \cup \{-1\}, \{0, 1\}^*, \Sigma'_{\text{ram}}, O'_{\text{ram}})$  is also a computer according to Definition 3.1 from [28], this theorem now follows immediately from Theorem 3.1 from [28].  $\square$

With basic RAM operations only computational processes can be considered whose data environment consists of one RAM memory. Below,  $n$ -RAM operations are introduced to remove this restriction. They are defined such that the basic RAM operations are exactly the 1-RAM operations.

An  $n$ -RAM operation ( $n \in \mathbb{N}^+$ ) is a function  $o : \Sigma_{\text{ram}}^n \rightarrow \Sigma_{\text{ram}}$  that satisfies the condition that there exist a basic RAM operation  $o'$  and a  $k \in \mathbb{N}$  with  $1 \leq k \leq n$  such that, for all  $\sigma_1, \dots, \sigma_n \in \Sigma_{\text{ram}}$ ,  $o'(\beta(\sigma_1, \dots, \sigma_n)) = \beta(\sigma_1, \dots, \sigma_{k-1}, o(\sigma_1, \dots, \sigma_n), \sigma_{k+1}, \dots, \sigma_n)$ , where  $\beta : \Sigma_{\text{ram}}^n \rightarrow \Sigma_{\text{ram}}$  is the unique function such that  $\beta(\sigma_1, \dots, \sigma_n)(n \cdot i + k - 1) = \sigma_k(i)$  for all  $i \in \mathbb{N}$  and  $k \in \mathbb{N}$  with  $1 \leq k \leq n$ . We write  $O_{n\text{-ram}}$ , where  $n \in \mathbb{N}^+$ , for the set of all  $n$ -RAM operations.

The function  $\zeta : \{k \in \mathbb{N}^+ \mid k \leq n\} \times \mathbb{N} \rightarrow \mathbb{N}$  defined by  $\zeta(k, i) = n \cdot i + k - 1$  is a bijection. From this it follows that the basic RAM operation  $o'$  and the  $k \in \mathbb{N}$  referred to in the above definition are unique if they exist.

The operations from  $\bigcup_{n \geq 1} O_{n\text{-ram}}$  are referred to as *RAM operations*.

In a similar way as  $n$ -RAM operations,  $n$ -RAM properties are defined. The basic RAM properties are exactly the 1-RAM properties.

An  $n$ -RAM property ( $n \in \mathbb{N}^+$ ) is a function  $p : \Sigma_{\text{ram}}^n \rightarrow \{0, 1\}$  that satisfies the condition that there exists a basic RAM property  $p'$  such that, for all  $\sigma_1, \dots, \sigma_n \in \Sigma_{\text{ram}}$ ,  $p'(\beta(\sigma_1, \dots, \sigma_n)) = p(\sigma_1, \dots, \sigma_n)$ , where  $\beta : \Sigma_{\text{ram}}^n \rightarrow \Sigma_{\text{ram}}$  is defined as above. We write  $P_{n\text{-ram}}$ , where  $n \in \mathbb{N}^+$ , for the set of all  $n$ -RAM properties.

The properties from  $\bigcup_{n \geq 1} P_{n\text{-ram}}$  are referred to as *RAM properties*.

The RAM conditions on  $\mathfrak{D}$  are:

1. the signature  $\Sigma_{\mathfrak{D}}$  of  $\mathfrak{D}$  includes (in addition to what is stated in Section 2.2):
  - a sort **BS** of *bit strings* and a sort **N** of *natural numbers*;
  - constants  $\lambda, 0, 1 : \mathbf{BS}$  and a binary operator  $\frown : \mathbf{BS} \times \mathbf{BS} \rightarrow \mathbf{BS}$ ;
  - constants  $0, 1 : \mathbf{N}$  and a binary operator  $+$  :  $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ ;
  - a constant  $\sigma_\lambda : \mathbf{D}$  and a ternary operator  $\oplus : \mathbf{D} \times \mathbf{N} \times \mathbf{BS} \rightarrow \mathbf{D}$ ;
2. the sorts, constants, and operators mentioned under 1 are interpreted in  $\mathfrak{D}$  as follows:
  - the sort **BS** is interpreted as the set  $\{0, 1\}^*$ , the sort **N** is interpreted as the set  $\mathbb{N}$ , and the sort **D** is interpreted as the set  $\Sigma_{\text{ram}}$ ;
  - the constant  $\lambda : \mathbf{BS}$  is interpreted as the empty bit string, the constants  $0, 1 : \mathbf{BS}$  are interpreted as the bit strings with the bit 0 and 1, respectively, as sole element,

and the operator  $\frown: \mathbf{BS} \times \mathbf{BS} \rightarrow \mathbf{BS}$  is interpreted as the concatenation operation on  $\{0, 1\}^*$ ;

- the constants  $0, 1: \mathbb{N}$  are interpreted as the natural numbers 0 and 1, respectively, and the operator  $+: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  is interpreted as the addition operation on  $\mathbb{N}$ ;
- the constant  $\sigma_\lambda: \mathbf{D}$  is interpreted as the unique  $\sigma \in \Sigma_{\text{ram}}$  such that  $\sigma(i) = \lambda$  for all  $i \in \mathbb{N}$  and the operator  $\oplus: \mathbf{D} \times \mathbb{N} \times \mathbf{BS} \rightarrow \mathbf{D}$  is interpreted as the *override* operation defined by  $\oplus(\sigma, i, w)(i) = w$  and, for all  $j \in \mathbb{N}$  with  $i \neq j$ ,  $\oplus(\sigma, i, w)(j) = \sigma(j)$ ;

3. the signature  $\Sigma_{\mathcal{D}}$  of  $\mathcal{D}$  is restricted as follows:

- for each operator from  $\Sigma_{\mathcal{D}}$ , the sort of its result is  $\mathbf{D}$  only if the sort of each of its arguments is  $\mathbf{D}$  or the operator is  $\oplus$ ;
- for each operator from  $\Sigma_{\mathcal{D}}$ , the sort of its result is  $\mathbf{B}$  only if the sort of each of its arguments is  $\mathbf{D}$ ;

4. the interpretation of the operators mentioned under 3 is restricted as follows:

- each operator with result sort  $\mathbf{D}$  other than  $\oplus$  is interpreted as a RAM operation;
- each operator with result sort  $\mathbf{B}$  is interpreted as a RAM property.

The notation  $\sigma\{i \mapsto w\}$ , where  $\sigma \in \Sigma_{\text{ram}}$ ,  $i \in \mathbb{N}$ , and  $w \in \{0, 1\}^*$ , is used for the term  $\oplus(\sigma, i, w)$ .

The RAM conditions make it possible to explain what it means that a given process computes a given partial function from  $(\{0, 1\}^*)^n$  to  $\{0, 1\}^*$  ( $n \in \mathbb{N}$ ). Moreover, the RAM conditions are nonrestrictive: presumably they allow to deal with all proposed versions of the RAM model of computation as well as all proposed models of parallel computation that are based on a version of the RAM model and the idea that the data environment of a computational process consists of one or more RAM memories.

### 3.2 Computing Partial Functions from $(\{0, 1\}^*)^n$ to $\{0, 1\}^*$

Below, we make precise in the setting of  $\text{ACP}_\epsilon^\tau\text{-I+REC+CFAR}$ , where  $\mathcal{D}$  is assumed to satisfy the RAM conditions, what it means that a given process computes a given partial function from  $(\{0, 1\}^*)^n$  to  $\{0, 1\}^*$  ( $n \in \mathbb{N}$ ).

In the rest of this paper,  $\mathcal{D}$  is assumed to satisfy the RAM conditions. Moreover, it is assumed that  $\mathfrak{m} \in \mathcal{V}$ .

Henceforth, the notation  $\rho_{w_1, \dots, w_n}$ , where  $w_1, \dots, w_n \in \{0, 1\}^*$ , is used for the unique  $\rho \in \mathcal{V}\text{al}$  such that  $\rho(\mathfrak{m}) = \sigma_\lambda\{1 \mapsto w_1\} \dots \{n \mapsto w_n\}$  and  $\rho(v) = \sigma_\lambda$  for all  $v \in \mathcal{V} \setminus \{\mathfrak{m}\}$ .

If  $t \in \mathcal{P}_{\text{rec}}$  is a finite-process term, then there is a finite upper bound to the number of actions that the process denoted by  $t$  can perform.

The *depth* of a finite-process term  $t \in \mathcal{P}_{\text{rec}}$ , written  $\text{depth}(t)$ , is defined as follows:  $\text{depth}(t) = \min\{n \in \mathbb{N} \mid \text{ACP}_\epsilon^\tau\text{-I+REC+CFAR+PR} \vdash \pi_n(t) = t\}$ . This means that  $\text{depth}(t)$  is the maximum number of actions other than  $\tau$  that the process denoted by  $t$  can perform.

Let  $n \in \mathbb{N}$ , let  $F : (\{0, 1\}^*)^n \rightarrow \{0, 1\}^*$ ,<sup>6</sup> and let  $W : \mathbb{N} \rightarrow \mathbb{N}$ . Then, roughly speaking, a process  $p$  computes  $F$  in  $W$  steps if:

- for all  $w_1, \dots, w_n \in \{0, 1\}^*$  such that  $F(w_1, \dots, w_n)$  is defined:
  - if  $p$  is started in a RAM memory state  $\sigma$  in which  $\sigma(1) = w_1, \dots, \sigma(n) = w_n$ , then:
    - it terminates successfully in a state  $\sigma'$  in which  $\sigma'(0) = F(w_1, \dots, w_n)$ ;
    - the total number of actions that it performs is not greater than  $W(l)$ , where  $l$  is the sum of the lengths of  $w_1, \dots, w_n$ ;
- for all  $w_1, \dots, w_n \in \{0, 1\}^*$  such that  $F(w_1, \dots, w_n)$  is undefined:
  - if  $p$  is started in a RAM memory state  $\sigma$  in which  $\sigma(1) = w_1, \dots, \sigma(n) = w_n$ , then:
    - it does not terminate successfully.

Below a precise definition in the setting of  $ACP_\epsilon^\tau\text{-I} + \text{REC} + \text{CFAR}$  is given. In that definition, the equation  $V_{\rho_{w_1, \dots, w_n}}(t) = V_{\rho_{w_1, \dots, w_n}}(t \cdot (m = \sigma' : \rightarrow \epsilon))$  expresses that the RAM memory state at successful termination satisfies  $\sigma'(0) = F(w_1, \dots, w_n)$ . This is the case because  $m = \sigma' : \rightarrow \epsilon$  equals  $\epsilon$  if the latter equation is satisfied and  $\delta$  otherwise.

Let  $t \in \mathcal{P}_{\text{rec}}$ , let  $n \in \mathbb{N}$ , let  $F : (\{0, 1\}^*)^n \rightarrow \{0, 1\}^*$ , and let  $W : \mathbb{N} \rightarrow \mathbb{N}$ . Then  $t$  computes  $F$  in  $W$  steps if:

- for all  $w_1, \dots, w_n \in \{0, 1\}^*$  such that  $F(w_1, \dots, w_n)$  is defined, there exists a  $\sigma' \in \Sigma_{\text{ram}}$  with  $\sigma'(0) = F(w_1, \dots, w_n)$  such that:<sup>7</sup>

$$\begin{aligned}
 &V_{\rho_{w_1, \dots, w_n}}(t) \text{ is a terminating-process term,} \\
 &ACP_\epsilon^\tau\text{-I} + \text{REC} + \text{CFAR} \vdash V_{\rho_{w_1, \dots, w_n}}(t) = V_{\rho_{w_1, \dots, w_n}}(t \cdot (m = \sigma' : \rightarrow \epsilon)), \\
 &\text{depth}(V_{\rho_{w_1, \dots, w_n}}(t)) \leq W(\ell(w_1) + \dots + \ell(w_n));
 \end{aligned}$$

- for all  $w_1, \dots, w_n \in \{0, 1\}^*$  such that  $F(w_1, \dots, w_n)$  is undefined:

$$V_{\rho_{w_1, \dots, w_n}}(t) \text{ is not a terminating-process term.}$$

We say that  $t$  computes  $F$  if there exists a  $W : \mathbb{N} \rightarrow \mathbb{N}$  such that  $t$  computes  $F$  in  $W$  steps, we say that  $F$  is a computable function if there exists a  $t \in \mathcal{P}$  such that  $t$  computes  $F$ , and we say that  $t$  is a computational process if there exists a  $F : (\{0, 1\}^*)^n \rightarrow \{0, 1\}^*$  such that  $t$  computes  $F$ .

We write  $\mathcal{CP}_{\text{rec}}$  for the set  $\{t \in \mathcal{P}_{\text{rec}} \mid t \text{ is a computational process}\}$ .

With the above definition, we can establish whether a process of the kind considered in the current setting computes a given partial function from  $(\{0, 1\}^*)^n$  to  $\{0, 1\}^*$  ( $n \in \mathbb{N}$ ) by equational reasoning using the axioms of  $ACP_\epsilon^\tau\text{-I} + \text{REC} + \text{CFAR}$ . This setting is more general than the setting provided by any known version of the RAM

<sup>6</sup> We write  $f : A \rightarrow B$ , where  $A$  and  $B$  are sets, to indicate that  $f$  is a partial function from  $A$  to  $B$ .

<sup>7</sup> We write  $\ell(u)$ , where  $u$  is a sequence, for the length of  $u$ .

model of computation. It is not suitable as a model of computation itself. However, various known models of computation can be defined by fixing which RAM operations and which RAM properties belong to  $\mathcal{D}$  and by restricting the computational processes to the ones of a certain form. To the best of my knowledge, the models of computation that can be dealt with in this way include all proposed versions of the RAM model as well as all proposed models of parallel computation that are based on a version of the RAM model and the idea that the data environment of a computational process consists of one or more RAM memories.

Whatever model of computation is obtained by fixing the RAM operations and the RAM properties and by restricting the computational processes to the ones of a certain form, it is an idealization of a real computer because it offers an unbounded number of registers that can contain a bit string of arbitrary length instead of a bounded number of registers that can only contain a bit string of a fixed length.

## 4 The RAMP Model of Computation

The setting introduced in the previous sections is used in this section to describe a version of the RAM model of computation. Because it focuses on the processes that are produced by RAMs when they execute their built-in program, the version of the RAM model of computation described in this section is called the RAMP (Random Access Machine Process) model of computation.

First, the operators are introduced that represent the RAM operations and the RAM properties that belong to  $\mathcal{D}$  in the case of the RAMP model of computation. Next, the interpretation of those operators as a RAM operation or a RAM property is given. Finally, the RAMP model of computation is described.

### 4.1 Operators for the RAMP Model

In this section, the operators that are relevant to the RAMP model of computation are introduced.

In the case of the RAMP model of computation, the set of operators from  $\Sigma_{\mathcal{D}}$  that are interpreted in  $\mathcal{D}$  as RAM operations or RAM properties is the set  $\mathcal{O}_{\text{RAMP}}$  defined as follows:

$$\begin{aligned} \mathcal{O}_{\text{RAMP}} = & \{ \text{binop}:s_1:s_2:d \mid \text{binop} \in \text{Binop} \wedge s_1, s_2 \in \text{Src} \wedge d \in \text{Dst} \} \\ & \cup \{ \text{unop}:s_1:d \mid \text{unop} \in \text{Unop} \wedge s_1 \in \text{Src} \wedge d \in \text{Dst} \} \\ & \cup \{ \text{cmpop}:s_1:s_2 \mid \text{cmpop} \in \text{Cmpop} \wedge s_1, s_2 \in \text{Src} \} , \end{aligned}$$

where

$$\begin{aligned} \text{Binop} &= \{ \text{add, sub, and, or} \} , \\ \text{Unop} &= \{ \text{not, shl, shr, mov} \} , \\ \text{Cmpop} &= \{ \text{eq, gt, beq} \} \end{aligned}$$

and

$$\begin{aligned} Src &= \{\#i \mid i \in \mathbb{N}\} \cup \mathbb{N} \cup \{@i \mid i \in \mathbb{N}\}, \\ Dst &= \mathbb{N} \cup \{@i \mid i \in \mathbb{N}\}. \end{aligned}$$

We write  $\mathcal{O}_{\text{RAMP}}^p$  for the set  $\{cmpop:s_1:s_2 \mid cmpop \in Cmpop \wedge s_1, s_2 \in Src\}$  and  $\mathcal{O}_{\text{RAMP}}^o$  for the set  $\mathcal{O}_{\text{RAMP}} \setminus \mathcal{O}_{\text{RAMP}}^p$ .

The operators from  $\mathcal{O}_{\text{RAMP}}^o$  are the operators that are interpreted in  $\mathcal{D}$  as basic RAM operations and the operators from  $\mathcal{O}_{\text{RAMP}}^p$  are the operators that are interpreted in  $\mathcal{D}$  as basic RAM properties.

The following is a preliminary explanation of the operators from  $\mathcal{O}_{\text{RAMP}}$ :

- carrying out the operation denoted by an operator of the form  $binop:s_1:s_2:d$  on a RAM memory in some state boils down to carrying out the binary operation named  $binop$  on the values that  $s_1$  and  $s_2$  stand for in that state and then changing the content of the register that  $d$  stands for into the result of this;
- carrying out the operation denoted by an operator of the form  $unop:s_1:d$  on a RAM memory in some state boils down to carrying out the unary operation named  $unop$  on the value that  $s$  stands for in that state and then changing the content of the register that  $d$  stands for into the result of this;
- carrying out the operation denoted by an operator of the form  $cmpop:s_1:s_2$  on a RAM memory in some state boils down to carrying out the binary operation named  $cmpop$  on the values that  $s_1$  and  $s_2$  stand for in that state.

The value that  $s_i$  ( $i = 1, 2$ ) stands for is as follows:

- *immediate*: it stands for the shortest bit string representing the natural number  $i$  if it is of the form  $\#i$ ;
- *direct addressing*: it stands for the content of the register with number  $i$  if it is of the form  $i$ ;
- *indirect addressing*: it stands for the content of the register whose number is represented by the content of the register with number  $i$  if it is of the form  $@i$ ;

and the register that  $d$  stands for is as follows:

- *direct addressing*: it stands for the register with number  $i$  if it is of the form  $i$ ;
- *indirect addressing*: it stands for the register whose number is represented by the content of the register with number  $i$  if it is of the form  $@i$ .

The following kinds of operations and relations on bit strings are covered by the operators from  $\mathcal{O}_{\text{RAMP}}$ : *arithmetic* operations (add, sub), *logical* operations (and, or, not), *bit-shift* operations (shl, shr), *data-transfer* operations (mov), *arithmetic* relations (eq, gt), and the bit-wise equality relation (beq). The arithmetic operations on bit strings are operations that model arithmetic operations on natural numbers with respect to their binary representation by bit strings, the logical operations on bit strings are bitwise logical operations, and the data transfer operation on bit strings is the identity operation on bit strings (which is carried out when copying bit strings). The arithmetic relations on bit strings are relations that model arithmetic relations on natural numbers with respect to their binary representation by bit strings.

## 4.2 Interpretation of the Operators for the RAMP Model

The interpretation of the operators from  $\mathcal{O}_{\text{RAMP}}$  in  $\mathfrak{D}$  is defined in this section.

We start with defining auxiliary functions for conversion between natural numbers and bit strings and evaluation of the elements of  $\text{Src}$  and  $\text{Dst}$ .

We write  $\dot{-}$  for proper subtraction of natural numbers. We write  $\div$  for zero-totalized Euclidean division of natural numbers, i.e. Euclidean division made total by imposing that division by zero yields zero (like in meadows, see e.g. [6, 7]). We use juxtaposition for concatenation of bit strings.

The *natural to bit string* function  $\mathbf{b} : \mathbb{N} \rightarrow \{0, 1\}^*$  is recursively defined as follows:

$$\mathbf{b}(n) = n \text{ if } n \leq 1 \quad \text{and} \quad \mathbf{b}(n) = (n \bmod 2)\mathbf{b}(n \div 2) \text{ if } n > 1$$

and the *bit string to natural* function  $\mathbf{n} : \{0, 1\}^* \rightarrow \mathbb{N}$  is recursively defined as follows:

$$\mathbf{n}(\lambda) = 0 \quad \text{and} \quad \mathbf{n}(bw) = b + 2 \cdot \mathbf{n}(w).$$

These definitions tell us that, when viewed as the binary representation of a natural number, the first bit of a bit string is considered the least significant bit. Results of applying  $\mathbf{b}$  have no leading zeros, but the operand of  $\mathbf{n}$  may have leading zeros.

Thus, we have that  $\mathbf{n}(\mathbf{b}(n)) = n$  and  $\mathbf{b}(\mathbf{n}(w)) = w'$ , where  $w'$  is  $w$  without leading zeros, if  $w \neq \lambda$ .

For each  $\sigma \in \Sigma_{\text{ram}}$ , the *src-valuation in  $\sigma$*  function  $\mathbf{v}_\sigma : \text{Src} \rightarrow \{0, 1\}^*$  is defined as follows:

$$\mathbf{v}_\sigma(\#i) = \mathbf{b}(i), \mathbf{v}_\sigma(i) = \sigma(i), \text{ and } \mathbf{v}_\sigma(@i) = \sigma(\mathbf{n}(\sigma(i)))$$

and, for each  $\sigma \in \Sigma_{\text{ram}}$ , the *dst-valuation in  $\sigma$*  function  $\mathbf{r}_\sigma : \text{Dst} \rightarrow \mathbb{N}$  is defined as follows:

$$\mathbf{r}_\sigma(i) = i \text{ and } \mathbf{r}_\sigma(@i) = \mathbf{n}(\sigma(i)).$$

We continue with defining the operations on bit strings that the operation names from  $\text{Binop} \cup \text{Unop}$  refer to.

We define the operations on bit strings that the operation names *add* and *sub* refer to as follows:

$$\begin{aligned} + : \{0, 1\}^* \times \{0, 1\}^* &\rightarrow \{0, 1\}^* : w_1 + w_2 = \mathbf{b}(\mathbf{n}(w_1) + \mathbf{n}(w_2)); \\ \dot{-} : \{0, 1\}^* \times \{0, 1\}^* &\rightarrow \{0, 1\}^* : w_1 \dot{-} w_2 = \mathbf{b}(\mathbf{n}(w_1) \dot{-} \mathbf{n}(w_2)). \end{aligned}$$

These definitions tell us that, although the operands of the operations  $+$  and  $\dot{-}$  may have leading zeros, results of applying these operations have no leading zeros.

We define the operations on bit strings that the operation names *and*, *or*, and *not* refer to recursively as follows:

$$\begin{aligned} \wedge : \{0, 1\}^* \times \{0, 1\}^* &\rightarrow \{0, 1\}^* : \lambda \wedge \lambda = \lambda, \lambda \wedge (bw) = 0(\lambda \wedge w), \\ &(bw) \wedge \lambda = 0(w \wedge \lambda), (b_1 w_1) \wedge (b_2 w_2) = (b_1 \wedge b_2)(w_1 \wedge w_2); \\ \vee : \{0, 1\}^* \times \{0, 1\}^* &\rightarrow \{0, 1\}^* : \lambda \vee \lambda = \lambda, \lambda \vee (bw) = b(\lambda \vee w), \\ &(bw) \vee \lambda = b(w \vee \lambda), (b_1 w_1) \vee (b_2 w_2) = (b_1 \vee b_2)(w_1 \vee w_2); \\ \neg : \{0, 1\}^* &\rightarrow \{0, 1\}^* : \neg \lambda = \lambda, \neg(bw) = (\neg b)(\neg w). \end{aligned}$$

These definitions tell us that, if the operands of the operations  $\wedge$  and  $\vee$  do not have the same length, sufficient leading zeros are assumed to exist. Moreover, results of applying these operations and results of applying  $\neg$  can have leading zeros.

We define the operations on bit strings that the operation names  $\text{shl}$  and  $\text{shr}$  refer to as follows:

$$\begin{aligned} \ll : \{0, 1\}^* &\rightarrow \{0, 1\}^* : \ll \lambda = \lambda, \ll (bw) = 0bw; \\ \gg : \{0, 1\}^* &\rightarrow \{0, 1\}^* : \gg \lambda = \lambda, \gg (bw) = w. \end{aligned}$$

These definitions tell us that results of applying the operations  $\ll$  and  $\gg$  can have leading zeros. We have that  $n(\ll w) = n(w) \cdot 2$  and  $n(\gg w) = n(w) \div 2$ .

Now, we are ready to define the interpretation of the operators from  $\mathcal{O}_{\text{RAMP}}$  in  $\mathfrak{D}$ . For each  $o \in \mathcal{O}_{\text{RAMP}}$ , the interpretation of  $o$  in  $\mathfrak{D}$ , written  $\llbracket o \rrbracket$ , is defined as follows:

$$\begin{aligned} \llbracket \text{add}:s_1:s_2:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto v_\sigma(s_1) + v_\sigma(s_2)\}; \\ \llbracket \text{sub}:s_1:s_2:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto v_\sigma(s_1) \dot{-} v_\sigma(s_2)\}; \\ \llbracket \text{and}:s_1:s_2:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto v_\sigma(s_1) \wedge v_\sigma(s_2)\}; \\ \llbracket \text{or}:s_1:s_2:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto v_\sigma(s_1) \vee v_\sigma(s_2)\}; \\ \llbracket \text{not}:s_1:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto \neg v_\sigma(s_1)\}; \\ \llbracket \text{shl}:s_1:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto \ll v_\sigma(s_1)\}; \\ \llbracket \text{shr}:s_1:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto \gg v_\sigma(s_1)\}; \\ \llbracket \text{mov}:s_1:d \rrbracket(\sigma) &= \sigma \{r_\sigma(d) \mapsto v_\sigma(s_1)\}; \\ \llbracket \text{eq}:s_1:s_2 \rrbracket(\sigma) &= \begin{cases} 1 & \text{if } n(v_\sigma(s_1)) = n(v_\sigma(s_2)), \\ 0 & \text{otherwise;} \end{cases} \\ \llbracket \text{gt}:s_1:s_2 \rrbracket(\sigma) &= \begin{cases} 1 & \text{if } n(v_\sigma(s_1)) > n(v_\sigma(s_2)), \\ 0 & \text{otherwise;} \end{cases} \\ \llbracket \text{beq}:s_1:s_2 \rrbracket(\sigma) &= \begin{cases} 1 & \text{if } v_\sigma(s_1) = v_\sigma(s_2), \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Clearly, the interpretation of each operator from  $\mathcal{O}_{\text{RAMP}}^o$  is a basic RAM operation and the interpretation of each operator from  $\mathcal{O}_{\text{RAMP}}^p$  is a basic RAM property.

### 4.3 RAMP Terms and the RAMP Model

Below, the RAMP model of computation is characterized in the setting introduced in Sections 2 and 3. However, first the notion of a RAMP term is defined. This notion is introduced to make precise what the set of possible computational processes is in the case of the RAMP model of computation.

In this section,  $\mathfrak{D}$  is fixed as follows:

- $\Sigma_{\mathfrak{D}}$  is the smallest signature including (a) all sorts, constants, and operators required by the assumptions made about  $\mathfrak{D}$  in  $\text{ACP}_{\epsilon}^T$ -I or the RAM conditions on  $\mathfrak{D}$  and (b) all operators from  $\mathcal{O}_{\text{RAMP}}$ ;

- all sorts, constants, and operators mentioned under (a) are interpreted in  $\mathcal{D}$  as required by the assumptions made about  $\mathcal{D}$  in  $\text{ACP}_\epsilon^t\text{-I}$  or the RAM conditions on  $\mathcal{D}$ ;
- all operators mentioned under (b) are interpreted in  $\mathcal{D}$  as defined at the end of Section 4.2.

Moreover, it is assumed that  $m \in \mathcal{V}$ .

A RAM process term, called a RAMP term for short, is a term from  $\mathcal{P}_{\text{rec}}$  that is of the form  $(X|E)$ , where, for each  $Y \in \text{vars}(E)$ , the recursion equation for  $Y$  in  $E$  has one of the following forms:

$$\begin{aligned}
 Y &= t \rightarrow [m := o(m)] \cdot Z, \\
 Y &= (p(m) = 1) \rightarrow [m := m] \cdot Z + (p(m) = 0) \rightarrow [m := m] \cdot Z', \\
 Y &= t \rightarrow \epsilon,
 \end{aligned}$$

where  $o \in \mathcal{O}_{\text{RAMP}}^o$ ,  $p \in \mathcal{O}_{\text{RAMP}}^p$ , and  $Z, Z' \in \text{vars}(E)$ . We write  $\mathcal{P}_{\text{RAMP}}$  for the set of all RAMP terms, and we write  $\mathcal{CP}_{\text{RAMP}}$  for  $\mathcal{P}_{\text{RAMP}} \cap \mathcal{CP}_{\text{rec}}$ .

In the definition of a RAMP term above,  $E$  would not be a guarded linear recursive specification over  $\text{ACP}_\epsilon^t\text{-I}$  if the (ineffective) assignment action  $[m := m]$  had been omitted from the summands of recursion equations of the second form. Because we regard both performing a RAM operation and testing a RAM property as a step of a computational process, the presence of this action has the technical advantage that counting of steps becomes counting of actions.

A process that can be denoted by a RAMP term is called a RAM process or a RAMP for short. So, a RAMP is a process that is definable by a guarded linear recursive specification over  $\text{ACP}_\epsilon^t\text{-I}$  of the kind described above.

As mentioned in Section 1, a basic assumption in this paper is that a model of computation is fully characterized by: (a) a set of possible computational processes, (b) for each possible computational process, a set of possible data environments, and (c) the effect of applying such processes to such environments.

$\mathcal{D}$  as fixed above and  $\mathcal{CP}_{\text{RAMP}}$  induce the RAMP model of computation:

- the set of possible computational processes is the set of all processes that can be denoted by a term from  $\mathcal{CP}_{\text{RAMP}}$ ;
- for each possible computational process, the set of possible data environments is the set of all  $\{m\}$ -indexed data environments;
- the effect of applying the process denoted by a  $t \in \mathcal{CP}_{\text{RAMP}}$  to a  $\{m\}$ -indexed data environment  $\mu$  is  $V_\rho(t)$ , where  $\rho$  is a flexible variable valuation that represents  $\mu$ .

The RAMP model of computation described above is intended to be essentially the same as the standard RAM model of computation extended with logical instructions and bit-shift instructions. The RAMs from that model are referred to as the BBRAMs (Basic Binary RAMs). There is a strong resemblance between  $\mathcal{O}_{\text{RAMP}}$  and the set  $\mathcal{I}_{\text{BBRAM}}$  of instructions from which the built-in programs of the BBRAMs can be constructed. Because the concrete syntax of the instructions does not matter,  $\mathcal{I}_{\text{BBRAM}}$  can be defined as follows:

$$\mathcal{I}_{\text{BBRAM}} = (\mathcal{O}_{\text{RAMP}}^o) \cup \{\text{jmp}:p:i \mid p \in \mathcal{O}_{\text{RAMP}}^p \wedge i \in \mathbb{N}^+\} \cup \{\text{halt}\}.$$

A *BBRAM program* is a non-empty sequence  $C$  from  $\mathcal{I}_{\text{BBRAM}}^*$  in which instructions of the form  $\text{jmp}:p:i$  with  $i > \ell(C)$  do not occur. We write  $\mathcal{IS}_{\text{BBRAM}}$  for the set of all BBRAM programs.

The execution of an instruction  $o$  from  $\mathcal{O}_{\text{RAMP}}^o$  by a BBRAM causes the state of its memory to change according to  $\llbracket o \rrbracket$ . The execution of an instruction of the form  $\text{jmp}:p:i$  or the instruction *halt* by a BBRAM has no effect on the state of its memory. After execution of an instruction by a BBRAM, the BBRAM proceeds to the execution of the next instruction from its built-in program except when the instruction is of the form  $\text{jmp}:p:i$  and  $\llbracket p \rrbracket = 1$  or when the instruction is *halt*. In the case that the instruction is of the form  $\text{jmp}:p:i$  and  $\llbracket p \rrbracket = 1$ , the execution proceeds to the  $i$ th instruction of the program. In the case that the instruction is *halt*, the execution terminates successfully.

The processes that are produced by the BBRAMs when they execute their built-in program are given by a function  $\mathcal{M}:\mathcal{IS}_{\text{BBRAM}} \rightarrow \mathcal{P}_{\text{RAMP}}$  that is defined up to consistent renaming of variables as follows:  $\mathcal{M}(c_1 \dots c_n) = \langle X_1|E \rangle$ , where  $E$  consists of, for each  $i \in \mathbb{N}$  with  $1 \leq i \leq n$ , an equation

$$\begin{aligned} X_i &= \mathbf{t} \rightarrow [\mathbf{m} := c_i(\mathbf{m})] \cdot X_{i+1} && \text{if } c_i \in \mathcal{O}_{\text{RAMP}}^o, \\ X_i &= (p(\mathbf{m}) = 1) \rightarrow [\mathbf{m} := \mathbf{m}] \cdot X_j + (p(\mathbf{m}) = 0) \rightarrow [\mathbf{m} := \mathbf{m}] \cdot X_{i+1} && \text{if } c_i \equiv \text{jmp}:p:j, \\ X_i &= \mathbf{t} \rightarrow \epsilon && \text{if } c_i \equiv \text{halt}, \end{aligned}$$

where  $X_1, \dots, X_n$  are different variable from  $\mathcal{X}$ .

Let  $C \in \mathcal{IS}_{\text{BBRAM}}$ . Then  $\mathcal{M}(C)$  denotes the process that is produced by the BBRAM whose built-in program is  $C$  when it executes its built-in program.

The definition of  $\mathcal{M}$  is in accordance with the descriptions of various versions of the RAM model of computation in the literature on this subject (see e.g. [1, 12, 21, 32]). However, to the best of my knowledge, none of these descriptions is precise and complete enough to allow of a proof of this.

The RAMPs are exactly the processes that can be produced by the BBRAMs when they execute their built-in program.

**Theorem 4** *For each constant  $\langle X|E \rangle \in \mathcal{P}_{\text{rec}}$ ,  $\langle X|E \rangle \in \mathcal{P}_{\text{RAMP}}$  iff there exists a  $C \in \mathcal{IS}_{\text{BBRAM}}$  such that  $\langle X|E \rangle$  and  $\mathcal{M}(C)$  are identical up to consistent renaming of variables.*

**Proof** It is easy to see that (a) for all  $C \in \mathcal{IS}_{\text{BBRAM}}$ ,  $\mathcal{M}(C) \in \mathcal{P}_{\text{RAMP}}$  and (b)  $\mathcal{M}$  is an bijection up to consistent renaming of variables. From this, the theorem follows immediately. □

Notice that, if  $\langle X|E \rangle$  and  $\langle X'|E' \rangle$  are identical up to consistent renaming of variables, then the equation  $\langle X|E \rangle = \langle X'|E' \rangle$  is derivable from RDP and RSP.

The following theorem is a result concerning the computational power of RAMPs.

**Theorem 5** *For each  $F : (\{0, 1\}^*)^n \rightarrow \{0, 1\}^*$ , there exists a  $t \in \mathcal{P}_{\text{RAMP}}$  such that  $t$  computes  $F$  iff  $F$  is Turing-computable.*

**Proof** By Theorem 4, it is sufficient to show that each BBRAM is Turing equivalent to a Turing machine. The BBRAM model of computation is essentially the same as the BRAM model of computation from [14] extended with bit-shift instructions. It follows directly from simulation results mentioned in [14] (part (3) of Theorem 2.4, part (3) of Theorem 2.5, and part (2) of Theorem 2.6) that each BRAM can be simulated by a Turing machine and vice versa. Because each Turing machine can be simulated by a BRAM, we immediately have that each Turing machine can be simulated by a BBRAM. It is easy to see that the bit-shift instructions can be simulated by a Turing machine. From this and the fact that each BRAM can be simulated by a Turing machine, it follows that each BBRAM can be simulated by a Turing machine as well. Hence, each BBRAM is Turing equivalent to a Turing machine.  $\square$

Henceforth, we write  $POLY$  for  $\{f \mid f: \mathbb{N} \rightarrow \mathbb{N} \wedge f \text{ is a polynomial function}\}$ . The following theorem tells us that the decision problems that belong to the complexity class  $\mathbf{P}$  are exactly the decision problems that can be solved by means of a RAMP in polynomially many steps.

**Theorem 6** *For each  $F: \{0, 1\}^* \rightarrow \{0, 1\}$ , there exist a  $t \in \mathcal{P}_{RAMP}$  and a  $W \in POLY$  such that  $t$  computes  $F$  in  $W$  steps iff  $F \in \mathbf{P}$ .*

**Proof** By Theorem 4, it is sufficient to show that time complexity on BBRAMs under the uniform time measure, i.e. the number of steps, and time complexity on multi-tape Turing machines are polynomially related. The BBRAM is essentially the same as the BRAM model of computation from [14] extended with bit-shift instructions. It follows directly from simulation results mentioned in [14] (part (3) of Theorem 2.4, part (3) of Theorem 2.5, and part (2) of Theorem 2.6) that time complexity on BRAMs under the uniform time measure and time complexity on multi-tape Turing machines are polynomially related. It is easy to see that the bit-shift instructions can be simulated by a multi-tape Turing machine in linear time. Hence, the time complexities remain polynomially related if the BRAM model is extended with the bit-shift instructions.  $\square$

## 5 The APRAMP Model of Computation

The setting introduced in Sections 2 and 3 is used in this section to describe an asynchronous parallel RAM model of computation. Because it focuses on the processes that are produced by asynchronous parallel RAMs when they execute their built-in programs, the parallel RAM model of computation described in this section is called the APRAMP (Asynchronous Parallel Random Access Machine Process) model of computation. In this model of computation, a computational process is the parallel composition of a number of processes that each has its own private RAM memory. However, together they also have a shared RAM memory for synchronization and communication.

First, the operators are introduced that represent the RAM operations and the RAM properties that belong to  $\mathcal{D}$  in the case of the APRAMP model of computation. Next, the interpretation of those operators as a RAM operation or a RAM property is given. Finally, the APRAMP model of computation is described.

In the case of the APRAMP model of computation, the set of operators from  $\Sigma_{\mathcal{D}}$  that are interpreted in  $\mathcal{D}$  as RAM operations or RAM properties is the set  $\mathcal{O}_{\text{PRAMP}}$  defined as follows:

$$\mathcal{O}_{\text{PRAMP}} = \mathcal{O}_{\text{RAMP}} \cup \{\text{ini:}\#i \mid i \in \mathbb{N}^+\} \\ \cup \{\text{loa:}@i:d \mid i \in \mathbb{N} \wedge d \in \text{Dst}\} \cup \{\text{sto:s:}@i \mid s \in \text{Src} \wedge i \in \mathbb{N}\},$$

where *Src* and *Dst* are as defined in Section 4.1.

In operators of the forms *binop:s<sub>1</sub>:s<sub>2</sub>:d*, *unop:s<sub>1</sub>:d*, and *cmpop:s<sub>1</sub>:s<sub>2</sub>* from  $\mathcal{O}_{\text{RAMP}}$ , *s<sub>1</sub>*, *s<sub>2</sub>*, and *d* refer to the private RAM memory. In operators of the form *loa:@i:d* and *sto:s:@i* from  $\mathcal{O}_{\text{PRAMP}} \setminus \mathcal{O}_{\text{RAMP}}$ , *s* and *d* refer to the private RAM memory too. The operators of the form *loa:@i:d* differ from the operators of the form *mov:@i:d* in that *@i* stands for the content of the register from the shared RAM memory whose number is represented by the content of the register with number *i* from the private memory. The operators of the form *sto:s:@i* differ from the operators of the form *mov:s:@i* in that *@i* stands for the register from the shared RAM memory whose number is represented by the content of the register with number *i* from the private memory. The operators of the form *ini:#i* initialize the registers from the private memory as follows: the content of the register with number 0 becomes the shortest bit string that represents the natural number *i* and the content of all other registers becomes the empty bit string.

Now, we are ready to define the interpretation of the operators from  $\mathcal{O}_{\text{PRAMP}}$  in  $\mathcal{D}$ . For each  $o \in \mathcal{O}_{\text{PRAMP}}$ , the interpretation of  $o$  in  $\mathcal{D}$ , written  $\llbracket o \rrbracket$ , is as defined in Section 4.2 for operators from  $\mathcal{O}_{\text{RAMP}}$  and as defined below for the additional operators:

$$\llbracket \text{ini:}\#i \rrbracket(\sigma_p) = \sigma_\lambda \{0 \mapsto \mathbf{b}(i)\}; \\ \llbracket \text{loa:}@i:d \rrbracket(\sigma_p, \sigma_s) = \sigma_p \{r_{\sigma_p}(d) \mapsto \sigma_s(\mathbf{n}(\sigma_p(i)))\}; \\ \llbracket \text{sto:s:}@i \rrbracket(\sigma_p, \sigma_s) = \sigma_s \{\mathbf{n}(\sigma_p(i)) \mapsto \mathbf{v}_{\sigma_p}(s)\}.$$

Here,  $\sigma_p$  should be thought of as a private-memory state and  $\sigma_s$  should be thought of as a shared-memory state.

Clearly, the interpretation of each operator of the form *ini:#i* is a 1-RAM operation and the interpretation of each operator of the form *loa:@i:d* or *sto:s:@i* is a 2-RAM operation.

Below, the APRAMP model of computation is characterized in the setting introduced in Sections 2 and 3. However, first the notion of an APRAMP term is defined. This notion is introduced to make precise what the set of possible computational processes is in the case of the APRAMP model of computation.

In this section,  $\mathcal{D}$  is fixed as follows:

- $\Sigma_{\mathcal{D}}$  is the smallest signature including (a) all sorts, constants, and operators required by the assumptions made about  $\mathcal{D}$  in  $\text{ACP}_\epsilon^\tau\text{-I}$  or the RAM conditions on  $\mathcal{D}$  and (b) all operators from  $\mathcal{O}_{\text{PRAMP}}$ ;
- all sorts, constants, and operators mentioned under (a) are interpreted in  $\mathcal{D}$  as required by the assumptions made about  $\mathcal{D}$  in  $\text{ACP}_\epsilon^\tau\text{-I}$  or the RAM conditions on  $\mathcal{D}$ ;
- all operators mentioned under (b) are interpreted in  $\mathcal{D}$  as defined above.

Moreover, it is assumed that  $m \in \mathcal{V}$  and, for all  $i \in \mathbb{N}^+$ ,  $m_i \in \mathcal{V}$ . We write  $\mathcal{V}_n^m$ , where  $n \in \mathbb{N}^+$ , for the set  $\{m\} \cup \{m_i \mid i \in \mathbb{N}^+ \wedge i \leq n\}$ .

An  $n$ -APRAM process term ( $n \in \mathbb{N}^+$ ), called an  $n$ -APRAMP term for short, is a term from  $\mathcal{P}_{\text{rec}}$  that is of the form  $\langle X_1 | E_1 \rangle \parallel \dots \parallel \langle X_n | E_n \rangle$ , where, for each  $i \in \mathbb{N}^+$  with  $i \leq n$ :

- for each  $X \in \text{vars}(E_i)$ , the recursion equation for  $X$  in  $E_i$  has one of the following forms:

- (1)  $X = t \rightarrow [m_i := \text{ini}:\#i(m_i)] \cdot Y$ ,
- (2)  $X = t \rightarrow [m_i := \text{loa}:@j:d(m_i, m)] \cdot Y$ ,
- (3)  $X = t \rightarrow [m_i := \text{sto}:s:@j(m_i, m)] \cdot Y$ ,
- (4)  $X = t \rightarrow [m_i := o(m_i)] \cdot Y$ ,
- (5)  $X = (p(m_i) = 1) \rightarrow [m_i := m_i] \cdot Y + (p(m_i) = 0) \rightarrow [m_i := m_i] \cdot Y'$ ,
- (6)  $X = t \rightarrow \epsilon$ ,

where  $o \in \mathcal{O}_{\text{RAM}}^o$ ,  $p \in \mathcal{O}_{\text{RAM}}^p$ , and  $Y, Y' \in \text{vars}(E_i)$ ;

- for each  $X \in \text{vars}(E_i)$ , the recursion equation for  $X$  in  $E_i$  is of the form (1) iff  $X \equiv X_i$ .

We write  $\mathcal{P}_{\text{APRAMP}}$  for the set of all terms  $t \in \mathcal{P}_{\text{rec}}$  such that  $t$  is an  $n$ -APRAMP term for some  $n \in \mathbb{N}^+$ , and we write  $\mathcal{CP}_{\text{APRAMP}}$  for  $\mathcal{P}_{\text{APRAMP}} \cap \mathcal{CP}_{\text{rec}}$ . Moreover, we write  $\text{deg}(t)$ , where  $t \in \mathcal{P}_{\text{APRAMP}}$ , for the unique  $n \in \mathbb{N}^+$  such that  $t$  is an  $n$ -APRAMP term.

The terms from  $\mathcal{P}_{\text{APRAMP}}$  are referred to as *APRAMP terms*.

A process that can be denoted by an APRAMP term is called an *APRAM process* or an *APRAMP* for short. So, an APRAMP is a parallel composition of processes that are definable by a guarded linear recursive specification over  $\text{ACP}_\epsilon^r\text{-I}$  of the kind described above. Each of those parallel processes starts with an initialization step in which the number of its private memory is made available in the register with number 0 from its private memory.

It follows from the auxiliary result about abstraction-free terms mentioned in Section 2.4 that, for all  $t \in \mathcal{P}_{\text{APRAMP}}$ , there exists a guarded linear recursive specification  $E$  and  $X \in \text{vars}(E)$  such that  $\text{ACP}_\epsilon^r\text{-I+REC} \vdash t = \langle X | E \rangle$ .

As mentioned before, a basic assumption in this paper is that a model of computation is fully characterized by: (a) a set of possible computational processes, (b) for each possible computational process, a set of possible data environments, and (c) the effect of applying such processes to such environments.

$\mathcal{D}$  as fixed above and  $\mathcal{CP}_{\text{APRAMP}}$  induce the APRAMP model of computation:

- the set of possible computational processes is the set of all processes that can be denoted by a term from  $\mathcal{CP}_{\text{APRAMP}}$ ;
- for each possible computational process  $p$ , the set of possible data environments is the set of all  $\mathcal{V}_{\text{deg}(t)}^m$ -indexed data environments, where  $t$  is a term from  $\mathcal{CP}_{\text{APRAMP}}$  denoting  $p$ ;
- the effect of applying the process denoted by a  $t \in \mathcal{CP}_{\text{APRAMP}}$  to a  $\mathcal{V}_{\text{deg}(t)}^m$ -indexed data environment  $\mu$  is  $V_\rho(t)$ , where  $\rho$  is a flexible variable valuation that represents  $\mu$ .

The APRAMP model of computation described above is intended to be close to the asynchronous parallel RAM model of computation sketched in [10, 25, 30]. However,

the time complexity measure for this model that will be introduced in Section 7 is quite different from the ones proposed in those papers. The APRAMP model described above is considered less close to the asynchronous parallel RAM model sketched in [17] because the latter provides special instructions for synchronization.

The APRAMPs can be looked upon as the processes that can be produced by a collection of BBRAMs with an extended instruction set when they execute their built-in program asynchronously in parallel.

The BBRAMs with the extended instruction set are referred to as the SMBRAMs (Shared Memory Binary RAMs). There is a strong resemblance between  $\mathcal{O}_{\text{PRAMP}}$  and the set  $\mathcal{I}_{\text{SMBRAM}}$  of instructions from which the built-in programs of the SMBRAMs can be constructed. Because the concrete syntax of the instructions does not matter,  $\mathcal{I}_{\text{SMBRAM}}$  can be defined as follows:

$$\mathcal{I}_{\text{SMBRAM}} = (\mathcal{O}_{\text{PRAMP}} \setminus \mathcal{O}_{\text{RAMP}}^{\text{p}}) \cup \{\text{jmp}:p:i \mid p \in \mathcal{O}_{\text{RAMP}}^{\text{p}} \wedge i \in \mathbb{N}^+\} \cup \{\text{halt}\} .$$

An *SMBRAM program* is a non-empty sequence  $C$  from  $\mathcal{I}_{\text{SMBRAM}}^*$  in which instructions of the form  $\text{jmp}:p:i$  with  $i > \ell(C)$  do not occur. We write  $\mathcal{I}_{\text{SMBRAM}}$  for the set of all SMBRAM programs.

For the SMBRAMs whose private memory has number  $i$  ( $i \in \mathbb{N}^+$ ), the processes that are produced when they execute their built-in program are given by a function  $\mathcal{M}_i : \mathcal{I}_{\text{SMBRAM}} \rightarrow \mathcal{P}_{\text{APRAMP}}$  that is defined up to consistent renaming of variables as follows:  $\mathcal{M}_i(c_1 \dots c_n) = \langle X_i \mid E_i \rangle$ , where  $E_i$  consists of the equation

$$X_i = \text{t} : \rightarrow [m_i := \text{ini}:\#i(m_i)] \cdot Y_1$$

and, for each  $j \in \mathbb{N}$  with  $1 \leq j \leq n$ , an equation

$$\begin{aligned} Y_j &= \text{t} : \rightarrow [m_i := c_j(m_i, m)] \cdot Y_{j+1} && \text{if } c_j \in \text{Load}, \\ Y_j &= \text{t} : \rightarrow [m := c_j(m_i, m)] \cdot Y_{j+1} && \text{if } c_j \in \text{Store}, \\ Y_j &= \text{t} : \rightarrow [m_i := c_j(m_i)] \cdot Y_{j+1} && \text{if } c_j \in \mathcal{O}_{\text{RAMP}}^{\text{o}}, \\ Y_j &= (p(m_i) = 1) : \rightarrow [m_i := m_i] \cdot Y_{j'} + (p(m_i) = 0) : \rightarrow [m_i := m_i] \cdot Y_{j+1} && \text{if } c_j \equiv \text{jmp}:p:j', \\ Y_j &= \text{t} : \rightarrow \epsilon && \text{if } c_j \equiv \text{halt}, \end{aligned}$$

where  $\text{Load} = \{\text{loa}:@i:d \mid i \in \mathbb{N} \wedge d \in \text{Dst}\}$ ,  $\text{Store} = \{\text{sto}:s:@i \mid s \in \text{Src} \wedge i \in \mathbb{N}\}$ , and  $Y_1, \dots, Y_n$  are different variables from  $\mathcal{X} \setminus \{X_i\}$ .

The APRAMPs are exactly the processes that can be produced by a collection of SMBRAMs when they execute their built-in program asynchronously in parallel.

**Theorem 7** *Let  $n \in \mathbb{N}^+$ . For all constants  $\langle X_1 \mid E_1 \rangle, \dots, \langle X_n \mid E_n \rangle \in \mathcal{P}_{\text{rec}}$ ,  $\langle X_1 \mid E_1 \rangle \parallel \dots \parallel \langle X_n \mid E_n \rangle \in \mathcal{P}_{\text{APRAMP}}$  iff there exist  $C_1, \dots, C_n \in \mathcal{I}_{\text{SMBRAM}}$  such that  $\langle X_1 \mid E_1 \rangle \parallel \dots \parallel \langle X_n \mid E_n \rangle$  and  $\mathcal{M}_1(C_1) \parallel \dots \parallel \mathcal{M}_n(C_n)$  are identical up to consistent renaming of variables.*

**Proof** Let  $i \in \mathbb{N}^+$  be such that  $i \leq n$ . It is easy to see that (a) for all  $C \in \mathcal{IS}_{\text{SMBRAM}}$ ,  $\mathcal{M}_i(C) \in \mathcal{P}_{\text{APRAMP}}$  and (b)  $\mathcal{M}_i$  is a bijection up to consistent renaming of variables. From this, it follows immediately that there exists a  $C \in \mathcal{IS}_{\text{SMBRAM}}$  such that  $\langle X_i | E_i \rangle$  and  $\mathcal{M}_i(C)$  are identical up to consistent renaming of variables. From this, the theorem follows immediately.  $\square$

## 6 The SPRAMP Model of Computation

In the asynchronous parallel RAM model of computation presented in Section 5, the parallel processes that make up a computational process do not automatically synchronize after each computational step. In this section, we describe a parallel RAM model of computation where the parallel processes that make up a computational process automatically synchronize after each computational step.

### 6.1 Synchronization of Parallel Processes

For the purpose of synchronizing parallel processes, a special instance of the synchronization merge operator of CSP [23] is defined in terms of the operators of  $\text{ACP}_\epsilon^\tau$ -I+RN. It is assumed that  $\text{sync}, \widetilde{\text{sync}} \in A$  and  $\gamma$  is such that  $\gamma(\text{sync}, \text{sync}) = \widetilde{\text{sync}}$ ,  $\gamma(\text{sync}, a) = \delta$  for all  $a \in A \setminus \{\text{sync}\}$ , and  $\gamma(\widetilde{\text{sync}}, a) = \delta$  for all  $a \in A$ . The special instance of the synchronization merge operator,  $\parallel_{\text{sync}}$ , is defined as follows:

$$t \parallel_{\text{sync}} t' = \rho_f(\partial_{\{\text{sync}\}}(\rho_f(t) \parallel \rho_f(t'))),$$

where the renaming function  $f$  is defined by  $f(\widetilde{\text{sync}}) = \text{sync}$  and  $f(\alpha) = \alpha$  if  $\alpha \in A \setminus \{\widetilde{\text{sync}}\}$ .

The process denoted by  $t_1 \parallel_{\text{sync}} \dots \parallel_{\text{sync}} t_n$  behaves as the  $n$  processes denoted by  $t_1, \dots, t_n$  in parallel, but with the restriction that the special synchronization action  $\text{sync}$  can only be performed simultaneously by all  $n$  processes.

Because of the use of an action renaming operator in the definition of  $\parallel_{\text{sync}}$ , it is assumed in Sections 6.2 and 7.3 that:

- from Section 2.4,  $\mathcal{P}_{\text{rec}}$  stands for the set of all closed  $\text{ACP}_\epsilon^\tau$ -I+REC+RN terms of sort  $\mathbf{P}$ ;
- from Section 2.4, all occurrences of  $\text{ACP}_\epsilon^\tau$ -I+REC,  $\text{ACP}_\epsilon^\tau$ -I+REC+CFAR, and  $\text{ACP}_\epsilon^\tau$ -I+REC+CFAR+PR have been replaced by  $\text{ACP}_\epsilon^\tau$ -I+REC+RN,  $\text{ACP}_\epsilon^\tau$ -I+REC+CFAR+RN, and  $\text{ACP}_\epsilon^\tau$ -I+REC+CFAR+PR+RN, respectively.

What is defined by the definitions given from Section 3 is the same before and after these changes. Moreover, all results given from Section 2.4, including the soundness and semi-completeness results, go through after these changes.

### 6.2 SPRAMP Terms and the SPRAMP Model

The parallel RAM model of computation described in this section is called the SPRAMP (Synchronous Parallel Random Access Machine Process) model of computation.

The operators that represent the RAM operations and the RAM properties that belong to  $\mathcal{D}$  in the case of the SPRAMP model of computation are the same as in the case of the APRAMP model of computation. The interpretation of those operators as a RAM operation or a RAM property is also the same as in the case of the APRAMP model of computation. Moreover,  $\mathcal{D}$  is fixed as in Section 5.

Below, the SPRAMP model of computation is characterized. However, first the notion of an  $n$ -SPRAMP term is defined.

Like in Section 5, it is assumed that  $\mathbf{m} \in \mathcal{V}$  and, for all  $i \in \mathbb{N}^+$ ,  $\mathbf{m}_i \in \mathcal{V}$ . Again, we write  $\mathcal{V}_n^{\mathbf{m}}$ , where  $n \in \mathbb{N}^+$ , for the set  $\{\mathbf{m}\} \cup \{\mathbf{m}_i \mid i \in \mathbb{N}^+ \wedge i \leq n\}$ .

An  $n$ -SPRAMP process term ( $n \in \mathbb{N}^+$ ), called an  $n$ -SPRAMP term for short, is a term from  $\mathcal{P}_{\text{rec}}$  that is of the form  $\langle X_1 | E_1 \rangle \parallel_{\text{sync}} \dots \parallel_{\text{sync}} \langle X_n | E_n \rangle$ , where, for each  $i \in \mathbb{N}^+$  with  $i \leq n$ :

- for each  $X \in \text{vars}(E_i)$ , the recursion equation for  $X$  in  $E_i$  has one of the following forms:

- (1)  $X = \mathbf{t} \rightarrow \text{sync} \cdot Y$ ,
- (2)  $X = \mathbf{t} \rightarrow [\mathbf{m}_i := \text{ini}:\#i(\mathbf{m}_i)] \cdot Y$ ,
- (3)  $X = \mathbf{t} \rightarrow [\mathbf{m}_i := \text{loa}:@j:d(\mathbf{m}_i, \mathbf{m})] \cdot Y$ ,
- (4)  $X = \mathbf{t} \rightarrow [\mathbf{m} := \text{sto}:s:@j(\mathbf{m}_i, \mathbf{m})] \cdot Y$ ,
- (5)  $X = \mathbf{t} \rightarrow [\mathbf{m}_i := o(\mathbf{m}_i)] \cdot Y$ ,
- (6)  $X = (p(\mathbf{m}_i) = 1) \rightarrow [\mathbf{m}_i := \mathbf{m}_i] \cdot Y + (p(\mathbf{m}_i) = 0) \rightarrow [\mathbf{m}_i := \mathbf{m}_i] \cdot Y'$ ,
- (7)  $X = \mathbf{t} \rightarrow \epsilon$ ,

where  $o \in \mathcal{O}_{\text{RAMP}}^o$ ,  $p \in \mathcal{O}_{\text{RAMP}}^p$ , and  $Y, Y' \in \text{vars}(E_i)$ ;

- for each  $X, Y \in \text{vars}(E_i)$  with  $Y$  occurring in the right-hand side of the recursion equation for  $X$  in  $E_i$ , the recursion equation for  $X$  in  $E_i$  is of the form (1) iff the recursion equation for  $Y$  in  $E_i$  is not of the form (1);
- for each  $X \in \text{vars}(E_i)$ , the recursion equation for  $X$  in  $E_i$  is of the form (2) iff  $X \equiv X_i$ .

We write  $\mathcal{P}_{\text{SPRAMP}}$  for the set of all terms  $t \in \mathcal{P}_{\text{rec}}$  such that  $t$  is an  $n$ -SPRAMP term for some  $n \in \mathbb{N}^+$ , and we write  $\mathcal{CP}_{\text{SPRAMP}}$  for  $\mathcal{P}_{\text{SPRAMP}} \cap \mathcal{CP}_{\text{rec}}$ . Moreover, we write  $\text{deg}(t)$ , where  $t \in \mathcal{P}_{\text{SPRAMP}}$ , for the unique  $n \in \mathbb{N}^+$  such that  $t$  is an  $n$ -SPRAMP term.

The terms from  $\mathcal{P}_{\text{SPRAMP}}$  are referred to as *SPRAMP terms*.

A process that can be denoted by an SPRAMP term is called an *SPRAMP process* or an *SPRAMP* for short. So, an SPRAMP is a synchronous parallel composition of processes that are definable by a guarded linear recursive specification over  $\text{ACP}_{\epsilon}^{\tau}$ -I of the kind described above. Each of those parallel processes starts with an initialization step in which the number of its private memory is made available in the register with number 0 from its private memory.

It follows from the auxiliary result about abstraction-free terms mentioned in Section 2.4 that, for all  $t \in \mathcal{P}_{\text{APRAMP}}$ , there exists a guarded linear recursive specification  $E$  and  $X \in \text{vars}(E)$  such that  $\text{ACP}_\epsilon^r\text{-I+REC+RN} \vdash t = \langle X|E \rangle$ .

$\mathcal{D}$  as fixed above and  $\mathcal{CP}_{\text{SPRAMP}}$  induce the SPRAMP model of computation:

- the set of possible computational processes is the set of all processes that can be denoted by a term from  $\mathcal{CP}_{\text{SPRAMP}}$ ;
- for each possible computational process  $p$ , the set of possible data environments is the set of all  $\mathcal{V}_{\text{deg}(t)}^m$ -indexed data environments, where  $t$  is a term from  $\mathcal{CP}_{\text{SPRAMP}}$  denoting  $p$ ;
- the effect of applying the process denoted by a  $t \in \mathcal{CP}_{\text{SPRAMP}}$  to a  $\mathcal{V}_{\text{deg}(t)}^m$ -indexed data environment  $\mu$  is  $\mathcal{V}_\rho(t)$ , where  $\rho$  is a flexible variable valuation that represents  $\mu$ .

The SPRAMP model of computation described above is intended to be close to the synchronous parallel RAM model of computation sketched in [36].<sup>8</sup> However, that model is a PRIORITY CRCW model whereas the SPRAMP model is essentially an ARBITRARY CRCW model. Roughly speaking, this means that, in the case that two or more parallel processes attempt to change the content of the same register at the same time, there is a difference in how the process that succeeds in its attempt is chosen (see also Section 8). Moreover, in the model sketched in [36], the built-in programs of the RAMs that make up a PRAM must be the same whereas the parallel processes that make up an SPRAMP may be different.

The SPRAMPs can be looked upon as the processes that can be produced by a collection of SMBRAMs when they execute their built-in program synchronously in parallel.

For the SMBRAMs whose private memory has number  $i$  ( $i \in \mathbb{N}^+$ ), the processes that are produced when they execute their built-in program are now given by a function  $\mathcal{M}_i^{\text{sync}} : \mathcal{IS}_{\text{SMBRAM}} \rightarrow \mathcal{P}_{\text{SPRAMP}}$  that is defined up to consistent renaming of variables as follows:  $\mathcal{M}_i^{\text{sync}}(c_1 \dots c_n) = \langle X_i|E_i \rangle$ , where  $E_i$  consists of the equation

$$X_i = t \rightarrow [m_i := \text{ini}:\#i(m_i)] \cdot Y_1$$

and, for each  $j \in \mathbb{N}$  with  $1 \leq j \leq n$ , an equation

$$\begin{aligned} Y_{2j-1} &= t \rightarrow \text{sync} \cdot Y_{2j} \\ Y_{2j} &= t \rightarrow [m_i := c_j(m_i, m)] \cdot Y_{2j+1} && \text{if } c_j \in \text{Load}, \\ Y_{2j} &= t \rightarrow [m := c_j(m_i, m)] \cdot Y_{2j+1} && \text{if } c_j \in \text{Store}, \\ Y_{2j} &= t \rightarrow [m_i := c_j(m_i)] \cdot Y_{2j+1} && \text{if } c_j \in \mathcal{O}_{\text{RAMP}}^0, \\ Y_{2j} &= (p(m_i) = 1) \rightarrow [m_i := m_i] \cdot Y_{2j'-1} + (p(m_i) = 0) \rightarrow [m_i := m_i] \cdot Y_{2j+1} && \text{if } c_j \equiv \text{jmp}:p:j', \\ Y_{2j} &= t \rightarrow \epsilon && \text{if } c_j \equiv \text{halt}, \end{aligned}$$

<sup>8</sup> The model sketched in [36] is known as the PRAM model and is similar to, among others, the models sketched in [16, 19, 25, 27, 37].

where  $Load = \{loa:@i:d \mid i \in \mathbb{N} \wedge d \in Dst\}$ ,  $Store = \{sto:s:@i \mid s \in Src \wedge i \in \mathbb{N}\}$ , and  $Y_1, \dots, Y_{2n}$  are different variables from  $\mathcal{X} \setminus \{X_i\}$ .

The SPRAMPs are exactly the processes that can be produced by a collection of SMBRAMs when they execute their built-in program synchronously in parallel.

**Theorem 8** *Let  $n \in \mathbb{N}^+$ . For all constants  $\langle X_1|E_1 \rangle, \dots, \langle X_n|E_n \rangle \in \mathcal{P}_{rec}$ ,  $\langle X_1|E_1 \rangle \parallel_{sync} \dots \parallel_{sync} \langle X_n|E_n \rangle \in \mathcal{P}_{SPRAMP}$  iff there exist  $C_1, \dots, C_n \in \mathcal{IS}_{SMBRAM}$  such that  $\langle X_1|E_1 \rangle \parallel_{sync} \dots \parallel_{sync} \langle X_n|E_n \rangle$  and  $\mathcal{M}_1^{sync}(C_1) \parallel_{sync} \dots \parallel_{sync} \mathcal{M}_n^{sync}(C_n)$  are identical up to consistent renaming of variables.*

**Proof** Let  $i \in \mathbb{N}^+$  be such that  $i \leq n$ . It is easy to see that (a) for all  $C \in \mathcal{IS}_{SMBRAM}$ ,  $\mathcal{M}_i^{sync}(C) \in \mathcal{P}_{SPRAMP}$  and (b)  $\mathcal{M}_i^{sync}$  is a bijection up to consistent renaming of variables. From this, it follows immediately that there exists a  $C \in \mathcal{IS}_{SMBRAM}$  such that  $\langle X_i|E_i \rangle$  and  $\mathcal{M}_i^{sync}(C)$  are identical up to consistent renaming of variables. From this, the theorem follows immediately.  $\square$

The first synchronous parallel RAM models of computation, e.g. the models proposed in [16, 19, 36], are older than the first asynchronous parallel RAM models of computation, e.g. the models proposed in [10, 25, 30]. It appears that the synchronous parallel RAM models have been primarily devised to be used in the area of computational complexity and that the asynchronous parallel RAM models have been primarily devised because the synchronous models were considered of restricted value in the area of algorithm efficiency.

## 7 Time and Work Complexity Measures

This section concerns complexity measures for the models of computation presented in Sections 4–6. Before the complexity measures in question are introduced, it is made precise in the current setting what a complexity measure is and what the complexity of a computable function from  $(\{0, 1\}^n)^*$  to  $\{0, 1\}^*$  under a given complexity measure is.

Let  $CP \subseteq \mathcal{CP}_{rec}$ . Then a *complexity measure* for  $CP$  is a partial function  $M : CP \times \bigcup_{m \in \mathbb{N}} (\{0, 1\}^m)^* \rightarrow \mathbb{N}$  such that, for all  $t \in CP$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^m)^*$ ,  $M(t, (w_1, \dots, w_n))$  is defined iff  $V_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

This notion of a complexity measure bears little resemblance to Blum’s notion of a complexity measure [8], but it is in accordance with Blum’s notion.

Let  $CP \subseteq \mathcal{CP}_{rec}$  and let  $M$  be a complexity measure for  $CP$ . Let  $n \in \mathbb{N}$  and let  $F : (\{0, 1\}^n)^* \rightarrow \{0, 1\}^*$  be a computable function. Let  $V : \mathbb{N} \rightarrow \mathbb{N}$ . Then  $F$  is of complexity  $V$  under the complexity measure  $M$  if there exists a  $t \in CP$  such that:

- $t$  computes  $F$ ;
- for all  $w_1, \dots, w_n \in \{0, 1\}^*$  such that  $F(w_1, \dots, w_n)$  is defined:

$$M(t, (w_1, \dots, w_n)) \leq V(\ell(w_1) + \dots + \ell(w_n)) .$$

## 7.1 The RAMP Model of Computation

Below, a time complexity measure and a work complexity measure for the RAMP model of computation are introduced.

The sequential uniform time measure yields, for a given RAMP and a given data environment, the maximum number of steps that can be performed by the given RAMP before eventually halting in the case where the initial data environment is the given data environment.

The *sequential uniform time measure* is the complexity measure  $M_{\text{SUT}}$  for  $\mathcal{CP}_{\text{RAMP}}$  defined by

$$M_{\text{SUT}}(t, (w_1, \dots, w_n)) = \text{depth}(V_{\rho_{w_1, \dots, w_n}}(t))$$

for all  $t \in \mathcal{CP}_{\text{RAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $V_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

The sequential uniform time measure is essentially the same as the uniform time complexity measure for the standard RAM model of computation (see e.g [1]).

It is an idealized time measure: the simplifying assumption is made that a RAMP performs one step per time unit.

The maximum number of steps that can be performed by a given RAMP can also be looked upon as the maximum amount of work. This makes the sequential uniform time measure a very plausible work measure as well.

The *sequential work measure* is the complexity measure  $M_{\text{SW}}$  for  $\mathcal{CP}_{\text{RAMP}}$  defined by

$$M_{\text{SW}}(t, (w_1, \dots, w_n)) = M_{\text{SUT}}(t, (w_1, \dots, w_n))$$

for all  $t \in \mathcal{CP}_{\text{RAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $V_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

In the sequential case, it is in accordance with our intuition that the uniform time complexity measure coincides with the work complexity measure. In the parallel case, this is not in accordance with our intuition: it is to be expected that the introduction of parallelism results in a reduction of the amount of time needed but not in a reduction of the amount of work needed.

The following connection between the complexity measure  $M_{\text{SUT}}$  and the notion of “computability in  $W$  steps” from Section 3.2 is a corollary of the definitions involved.

**Corollary 1** *For each computable function  $F : (\{0, 1\}^*)^n \rightarrow \{0, 1\}^*$  and function  $V : \mathbb{N} \rightarrow \mathbb{N}$ ,  $F$  is of complexity  $V$  under the complexity measure  $M_{\text{SUT}}$  iff there exists a  $t \in \mathcal{P}_{\text{RAMP}}$  such that  $t$  computes  $F$  in  $V$  steps.*

## 7.2 The APRAMP Model of Computation

Below, a time complexity measure and a work complexity measure for the APRAMP model of computation are introduced.

The asynchronous parallel uniform time measure yields, for a given APRAMP and a given data environment, the maximum over all parallel processes that make up the given APRAMP of the maximum number of steps that can be performed before

eventually halting in the case where the initial data environment is the given data environment.

The *asynchronous parallel uniform time measure* is the complexity measure  $M_{\text{APUT}}$  for  $\mathcal{CP}_{\text{APRAMP}}$  defined by

$$M_{\text{APUT}}(t, (w_1, \dots, w_n)) = \max\{\text{depth}(\tau_{\overline{H}_i}(V_{\rho_{w_1, \dots, w_n}}(t))) \mid 1 \leq i \leq \text{deg}(t)\},$$

where  $\overline{H}_i$  is the set of all  $\alpha \in \mathcal{A}$  in which  $m_i$  does not occur, for all  $t \in \mathcal{CP}_{\text{APRAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $V_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

In this definition,  $\tau_{\overline{H}_i}$  turns steps of the process denoted by  $V_{\rho_{w_1, \dots, w_n}}(t)$  that are not performed by the parallel process whose private memory is referred to by  $m_i$  into silent steps. Because *depth* does not count silent steps,  $\text{depth}(\tau_{\overline{H}_i}(V_{\rho_{w_1, \dots, w_n}}(t)))$  is the maximum number of steps that the parallel process whose private memory is referred to by  $m_i$  can perform.

Because it yields the maximum number of steps that can be performed by one of the parallel processes that make up a given APRAMP, the asynchronous parallel uniform time measure differs from the asynchronous parallel work measure.

The *asynchronous parallel work measure* is the complexity measure  $M_{\text{APW}}$  for  $\mathcal{CP}_{\text{APRAMP}}$  defined by

$$M_{\text{APW}}(t, (w_1, \dots, w_n)) = \text{depth}(V_{\rho_{w_1, \dots, w_n}}(t))$$

for all  $t \in \mathcal{CP}_{\text{APRAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $V_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

The sequential work measure and the asynchronous parallel work measure are such that comparison of complexities under these measures have some meaning: both concern the maximum number of steps that can be performed by a computational process.

Like all complexity measures introduced in this section, the asynchronous parallel uniform time measure introduced above is a worst-case complexity measure. It is quite different from the parallel time complexity measures that have been proposed for the asynchronous parallel RAM model of computation sketched in [10, 25, 30]. The round complexity measure is proposed as parallel time complexity measure in [10, 25] and an expected time complexity measure is proposed as parallel time complexity measure in [30]. Neither of those measures is a worst-case complexity measure: the round complexity measure removes certain cases from consideration and the expected time complexity measure is an average-case complexity measure.

It appears that the round complexity measure and the expected time complexity measure are more important to analysis of the efficiency of parallel algorithms whereas the asynchronous parallel time complexity measure introduced above is more important to analysis of the complexity of computational problems that are amenable to solution by a parallel algorithm. After all, the area of computational complexity is mostly concerned with worst-case complexity.

In [30], the asynchronous parallel uniform time measure introduced above is explicitly rejected. Consider the case where there exists an interleaving of the parallel

processes that make up an APRAMP that is close to performing all steps of each of the processes uninterrupted by steps of the others. Then the interleaving concerned is not ruled out by synchronization (through the shared memory) and may even be enforced by synchronization. So it may be likely or unlikely to occur. Seen in that light, it is surprising why it is stated in [30] that such an interleaving has “very low probability, yielding a sequential measure”.

### 7.3 The SPRAMP Model of Computation

Below, a time complexity measure and a work complexity measure for the SPRAMP model of computation are introduced.

The time complexity measure introduced below is essentially the same as the uniform time complexity measure that goes with the synchronous parallel RAM model of computation sketched in [36] and similar models.

The synchronous parallel uniform time measure yields, for a given SPRAMP and a given data environment, the maximum number of synchronization steps that can be performed by the given SPRAMP before eventually halting in the case where the initial data environment is the given data environment.

The *synchronous parallel uniform time measure* is the complexity measure  $M_{\text{SPUT}}$  for  $\mathcal{CP}_{\text{SPRAMP}}$  defined by

$$M_{\text{SPUT}}(t, (w_1, \dots, w_n)) = \text{depth}(\tau_{\overline{\text{sync}}}(\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t))),$$

where  $\overline{\text{sync}} = \mathcal{A} \setminus \{\text{sync}\}$ , for all  $t \in \mathcal{CP}_{\text{SPRAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

In this definition,  $\tau_{\overline{\text{sync}}}$  turns all steps of the process denoted by  $\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)$  other than synchronization steps, i.e. all computational steps, into silent steps. Because *depth* does not count silent steps,  $\text{depth}(\tau_{\overline{\text{sync}}}(\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)))$  is the maximum number of synchronization steps that can be performed by the process denoted by  $\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)$  before eventually halting.

Because the parallel processes that make up a given SPRAMP synchronize after each computational step, the time between two consecutive synchronization steps can be considered one time unit. Therefore, the synchronous parallel uniform time measure is a plausible time measure. Clearly, the maximum number of synchronization steps that can be performed by the given SPRAMP and the maximum number of computational steps that can be performed by the given SPRAMP are separate numbers. So the synchronous parallel uniform time measure differs from the synchronous parallel work measure.

The *synchronous parallel work measure* is the complexity measure  $M_{\text{SPW}}$  for  $\mathcal{CP}_{\text{SPRAMP}}$  defined by

$$M_{\text{SPW}}(t, (w_1, \dots, w_n)) = \text{depth}(\tau_{\text{sync}}(\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)))$$

for all  $t \in \mathcal{CP}_{\text{SPRAMP}}$  and  $(w_1, \dots, w_n) \in \bigcup_{m \in \mathbb{N}} (\{0, 1\}^*)^m$  such that  $\mathcal{V}_{\rho_{w_1, \dots, w_n}}(t)$  is a terminating-process term.

The sequential work measure and the synchronous parallel work measure are such that comparison of complexities under these measures have some meaning: both concern the maximum number of computational steps that can be performed by a computational process.

Take an SPRAMP and the APRAMP which is the SPRAMP without the automatic synchronization after each computational step. Assume that at any stage the next step to be taken by any of the parallel processes that make up the APRAMP does not depend on the steps that have been taken by the other parallel processes. Then the synchronous parallel time measure  $M_{\text{SPUT}}$  yields for the SPRAMP the same result as the asynchronous parallel time measure  $M_{\text{APUT}}$  yields for the APRAMP.

## 8 SPRAMPs and the Parallel Computation Thesis

The SPRAMP model of computation is a simple model based on an idealization of existing shared memory parallel machines that abstracts from synchronization overhead. The synchronous parallel uniform time measure introduced for this model is a simple, hardware independent, and worst-case complexity measure.

The question is whether the SPRAMP model of computation is a reasonable model of parallel computation. A model of parallel computation is generally considered reasonable if the parallel computation thesis holds. In the current setting, this thesis can be phrased as follows: the *parallel computation thesis* holds for a model of computation if, for each computable partial function from  $(\{0, 1\}^*)^n$  to  $\{0, 1\}^*$  ( $n \in \mathbb{N}$ ), its complexity under the time complexity measure for that model is polynomially related to its complexity under the space complexity measure for the multi-tape Turing machine model of computation.

Before we answer the question whether the SPRAMP model of computation is a reasonable model of parallel computation, we go into a classification of synchronous parallel RAMs. This classification is used later on in answering the question. Following [24], PRAM is used as a common name for a synchronous parallel RAM regardless of its classification.

First of all, there are PRAMs whose constituent RAMs may execute different programs and PRAMs whose constituent RAMs must execute the same program. The former PRAMs are classified as MIMD (Multiple Instruction, Multiple Data) and the latter PRAMs are classified as SIMD (Single Instruction, Multiple Data).

In [24, Section 2.1], PRAMs are classified according to their restrictions on shared memory access as EREW (Exclusive-Read Exclusive-Write), CREW (Concurrent-Read Exclusive-Write) or CRCW (Concurrent-Read Concurrent-Write). CRCW PRAMs are further classified according to their way of resolving write conflicts as COMMON, where all values attempted to be written concurrently into the same shared register must be identical, ARBITRARY, where one of the values attempted to be written concurrently into the same shared register is chosen arbitrarily, or PRIORITY, where the RAMs making up the PRAM are numbered and, from all values attempted to be written concurrently into the same shared register, the one attempted to be written by the RAM with the lowest number is chosen.

An SPRAMP is a process that can be produced by a MIMD ARBITRARY CRCW PRAM with SMBRAMs (see Section 5) as constituent RAMs.

Below, the next two lemmas about the above classifications of PRAMs will be used to show that the parallel computation thesis holds for the SPRAMP model of computation.

**Lemma 4** *Assuming a fixed instruction set:*

1. *MIMD PRIORITY CRCW PRAMs can be simulated by MIMD ARBITRARY CRCW PRAMs with the same number of RAMs and with the parallel time increased by a factor of  $O(\log(p))$ , where  $p$  is the number of RAMs;*
2. *MIMD ARBITRARY CRCW PRAMs can be simulated by MIMD PRIORITY CRCW PRAMs with the same number of RAMs and the same parallel time.*

**Proof** Assume a fixed instruction set.

Part 1. It is shown in [24, Section 3.1] that MIMD PRIORITY CRCW PRAMs can be simulated by MIMD EREW PRAMs with the same number of RAMs and with the parallel time increased by only a factor of  $O(\log(p))$ , where  $p$  is the number of RAMs. It follows directly from the definitions concerned that MIMD EREW PRAMs can be simulated by MIMD ARBITRARY CRCW PRAMs with the same number of RAMs and the same parallel time (the programs involved can be executed directly). Hence, MIMD PRIORITY CRCW PRAMs can be simulated by MIMD ARBITRARY CRCW PRAMs with the same number of RAMs and with the parallel time increased by a factor of  $O(\log(p))$ , where  $p$  is the number of RAMs.

Part 2. It follows directly from the definitions concerned that MIMD ARBITRARY CRCW PRAMs can be simulated by MIMD PRIORITY CRCW PRAMs with the same number of RAMs and the same parallel time (the programs involved can be executed directly).  $\square$

**Lemma 5** *Assuming a fixed instruction set:*

1. *SIMD PRIORITY CRCW PRAMs can be simulated by MIMD PRIORITY CRCW PRAMs with the same number of RAMs and with the same parallel time;*
2. *MIMD PRIORITY CRCW PRAMs can be simulated by SIMD PRIORITY CRCW PRAMs with the same number of RAMs and with the parallel time increased by a constant factor.*

**Proof** Assume a fixed instruction set.

Part 1. This follows directly from the definitions concerned (the programs involved can be executed directly).

Part 2. This is a special case of Theorem 3 from [39].  $\square$

The next theorem expresses that the parallel computation thesis holds for the SPRAMP model of computation.

**Theorem 9** *Let  $F : (\{0, 1\}^*)^m \rightarrow \{0, 1\}^*$  for some  $m \in \mathbb{N}$  be a computable function and let  $T, S : \mathbb{N} \rightarrow \mathbb{N}$ . Then:*

- *if  $F$  is of complexity  $T(n)$  under the synchronous parallel time complexity measure  $M_{\text{SPUT}}$  for the SPRAMP model of computation, then there exists a  $k \in \mathbb{N}$  such that  $F$*

is of complexity  $O(T(n)^k)$  under the space complexity measure for the multi-tape Turing machine model of computation;

- if  $F$  is of complexity  $S(n)$  under the space complexity measure for the multi-tape Turing machine model of computation, then there exists a  $k \in \mathbb{N}$  such that  $F$  is of complexity  $O(S(n)^k)$  under the synchronous parallel time complexity measure  $M_{\text{SPUT}}$  for the SPRAMP model of computation provided that  $S(n) \geq \log(n)$  for all  $n \in \mathbb{N}$ .

**Proof** In [19], SIMDAGs are introduced. SIMDAGs are SIMD PRIORITY CRCW PRAMs with a subset of the instruction set of SMBRAMs as instruction set. Because  $DSPACE(S(n)) \subseteq NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$ , the variant of the current theorem for the SIMDAG model of computation follows immediately from Theorems 2.1 and 2.2 from [19] under a constructibility assumption for  $S(n)$ . However, the proofs of those theorems go through with the instruction set of SMBRAMs because none of the SMBRAM instructions builds bit strings that are more than  $O(T(n))$  bits long in  $T(n)$  time. Moreover, if we take forking variants of SIMDAGs with the instruction set of SMBRAMs (resembling the P-RAMs from [16]), the constructibility assumption for  $S(n)$  is not needed. This can be shown in the same way as in the proof of Lemma 1a from [16].

In the rest of this proof, we write E-SIMDAG for a SIMDAG with the instruction set of SMBRAMs and forking E-SIMDAG for a forking variant of an E-SIMDAG.

The variant of the current theorem for the forking E-SIMDAG model of computation follows directly from the above-mentioned facts.

Now forking E-SIMDAGs can be simulated by E-SIMDAGs with  $O(p)$  number of SMBRAMs and with the parallel time increased by a factor of  $O(\log(p))$ , where  $p$  is the number of SMBRAMs used by the forking E-SIMDAG concerned. This is proved as in the proof of Lemma 2.1 from [20]. The other way round, E-SIMDAGs can be simulated by forking E-SIMDAGs with eventually the same number of SMBRAMs and with the parallel time increased by  $O(\log(p))$ , where  $p$  is the number of SMBRAMs of the E-SIMDAG concerned. This is easy to see: before the programs of the  $p$  SMBRAMs involved can be executed directly, the  $p$  SMBRAMs must be created by forking and this can be done in  $O(\log(p))$  time. It follows immediately from these simulation results that time complexities on forking E-SIMDAGs are polynomially related to time complexities on E-SIMDAGs.

The variant of the current theorem for the E-SIMDAG model of computation follows directly from the variant of the current theorem for the forking E-SIMDAG model of computation and the above-mentioned polynomial relationship. From this, the fact that E-SIMDAGs are actually SIMD PRIORITY CRCW PRAMs that are composed of SMBRAMs, Lemmas 5, 4, and Theorem 8, the current theorem now follows directly.  $\square$

## 9 Concluding Remarks

In this paper, it has been studied whether the imperative process algebra  $ACP_{\epsilon}^{\tau}$ -I can play a role in the field of models of computation.

Models of computation corresponding to models based on sequential random access machines, asynchronous parallel random access machines, synchronous parallel random access machines, and complexity measures for those models have been described in a direct and mathematically precise way in the setting of  $ACP_{\epsilon}^c$ -I. Central in the models described are the computational processes considered instead of the abstract machines that produce those processes when they execute their built-in program.

The work presented in this paper pertains to formalizing models of computation. Little work has been done in this area. Three notable exceptions are [2, 31, 40]. Those papers are concerned with formalization in a theorem prover (HOL4, Isabelle/HOL, Matita) and focusses on some version of the Turing machine model of computation. This makes it impracticable to compare the work presented in those papers with the work presented in this paper.

Whereas it is usual in versions of the RAM model of computation that bit strings are represented by natural numbers, here natural numbers are represented by bit strings. Moreover, the choice has been made to represent the natural number 0 by the bit string 0 and to adopt the empty bit string as the register content that indicates that a register is (as yet) unused.

**Author Contributions** The single author wrote and reviewed the manuscript.

## Declarations

**Competing interests** The authors declare no competing interests.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms. Addison-Wesley, Reading, MA (1974)
2. Asperti, A., Ricciotti, W.: A formalization of multi-tape Turing machines. *Theoret. Comput. Sci.* **603**, 23–42 (2015). <https://doi.org/10.1016/j.tcs.2015.07.013>
3. Baeten, J.C.M., Bergstra, J.A.: Global renaming operators in concrete process algebra. *Inf. Control* **78**(3), 205–245 (1988). [https://doi.org/10.1016/0890-5401\(88\)90027-2](https://doi.org/10.1016/0890-5401(88)90027-2)
4. 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>
5. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Inf. Control* **60**(1–3), 109–137 (1984). [https://doi.org/10.1016/S0019-9958\(84\)80025-X](https://doi.org/10.1016/S0019-9958(84)80025-X)
6. Bergstra, J.A., Middelburg, C.A.: Inversive meadows and divisive meadows. *J. Appl. Log.* **9**(3), 203–220 (2011). <https://doi.org/10.1016/j.jal.2011.03.001>
7. Bergstra, J.A., Tucker, J.V.: The rational numbers as an abstract data type. *J. ACM* **54**(2), 7 (2007). <https://doi.org/10.1145/1219092.1219095>

8. Blum, M.: A machine-independent theory of the complexity of recursive functions. *J. ACM* **14**(2), 322–336 (1967). <https://doi.org/10.1145/321386.321395>
9. Bouwman, M.S., Luttik, S.P., Schols, W.R.M., Willemse, T.A.C.: A process algebra with global variables. *Electronic Proceedings in Theoretical Computer Science* **322**, 33–50 (2020). <https://doi.org/10.4204/EPTCS.322.5>
10. Cole, R., Zajicek, O.: The APRAM: Incorporating asynchrony into the PRAM model. In: SPAA '89. pp. 169–178 ACM, New York (1989). <https://doi.org/10.1145/72935.72954>
11. Colvin, R., Hayes, I.J.: CSP with hierarchical state. In: Leuschel, M., Wehrheim, H. (eds) IFM 2009. *Lecture Notes in Computer Science*, vol. 5423. pp 118–135. Springer, Berlin (2009). [https://doi.org/10.1007/978-3-642-00255-7\\_9](https://doi.org/10.1007/978-3-642-00255-7_9)
12. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. *J Comput. Syst. Sci.* **7**(4), 354–375 (1973). <https://doi.org/10.1145/800152.804898>
13. De Nicola, R., Pugliese, R. (1997) Testing semantics of asynchronous distributed programs. In: Dam, M. (ed) LOMAPS 1996. *Lecture Notes in Computer Science*, vol. 1192. pp 320–344. Springer, Berlin. [https://doi.org/10.1007/3-540-62503-8\\_15](https://doi.org/10.1007/3-540-62503-8_15)
14. van Emde Boas, P.: Machine models and simulations. In: van Leeuwen, J. (ed.) *Handbook of theoretical computer science* vol. A. pp 2–66. Elsevier, Amsterdam (1990) <https://doi.org/10.1016/B978-0-444-88071-0.50006-0>
15. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed) ESOP 2012. *Lecture Notes in Computer Science*, vol 7211. pp 295–315. Springer, Berlin (2012) [https://doi.org/10.1007/978-3-642-28869-2\\_15](https://doi.org/10.1007/978-3-642-28869-2_15)
16. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: STOC '78. pp 114–118. ACM, New York, (1978) <https://doi.org/10.1145/800133.804339>
17. Gibbons, P.B.: A more practical PRAM model. In: SPAA '89. pp 158–168. ACM, New York (1989) <https://doi.org/10.1145/72935.72953>
18. Goguen, J.A.: Theorem Proving and Algebra (2021) [arXiv:2101.02690](https://arxiv.org/abs/2101.02690)
19. Goldschlager, L.M.: A universal interconnection pattern for parallel computers. *J. ACM* **29**(4), 1073–1086 (1982). <https://doi.org/10.1145/322344.322353>
20. Goodrich, M.T.: Intersecting line segments in parallel with an output-sensitive number of processors. In: SPAA '89. pp 127–137. ACM, New York (1989). <https://doi.org/10.1145/72935.72950>
21. Hartmanis, J., Simon, J.: On the power of multiplication in random access machines. In: SWAT '74, pp. 13–23. IEEE, New York (1974). <https://doi.org/10.1109/SWAT.1974.20>
22. Hennessy, M., Ingólfsdóttir, A.: Communicating processes with value-passing and assignments. *Formal Aspects Comput.* **5**(5), 432–466 (1993). <https://doi.org/10.1007/BF01212486>
23. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (1985)
24. Karp, R.M., Ramachandran, V.: Parallel algorithms for shared-memory machines. In: van Leeuwen, J. (ed) *Handbook of Theoretical Computer Science* vol. A. pp 870–941. Elsevier, Amsterdam (1990). <https://doi.org/10.1016/B978-0-444-88071-0.50022-9>
25. Kruskal, C.P.: Rudolph L, Snir M: A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.* **71**(1), 95–132 (1990). [https://doi.org/10.1016/0304-3975\(90\)90192-K](https://doi.org/10.1016/0304-3975(90)90192-K)
26. Lamport, L.: The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **16**(3), 872–923 (1994). <https://doi.org/10.1145/177492.177726>
27. Mak, L.: Parallelism always helps. *SIAM Journal of Computing* **26**(1), 153–172 (1997). <https://doi.org/10.1137/S0097539794265402>
28. Maurer, W.D.: A theory of computer instructions. *Sci. Comput. Program.* **60**, 244–273 (2006). <https://doi.org/10.1016/j.scico.2005.09.001>
29. 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>
30. Nishimura, N.: A model for asynchronous shared memory parallel computation. *SIAM J Comput.* **23**(6), 1122–1147 (1994). <https://doi.org/10.1137/S0097539791219670>
31. Norrish, M.: Mechanised computability theory. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds) ITP 2011. *Lecture Notes in Computer Science*, vol. 6898. pp 297–311. Springer, Berlin (2011). [https://doi.org/10.1007/978-3-642-22863-6\\_22](https://doi.org/10.1007/978-3-642-22863-6_22)
32. Papadimitriou, C.H.: *Computational Complexity*. Addison-Wesley, Reading, MA (1994)
33. Pigozzi, D., Salibra, A.: The abstract variable-binding calculus. *Stud. Logica.* **55**(1), 129–179 (1995). <https://doi.org/10.1007/BF01053036>

34. Sannella, D., Tarlecki, A. : Algebraic preliminaries. In: Astesiano, E., Kreowski, H.-J., Krieg-Brückner, B. (eds) Algebraic Foundations of Systems Specification. pp 13–30. Springer, Berlin (1999). [https://doi.org/10.1007/978-3-642-59851-7\\_2](https://doi.org/10.1007/978-3-642-59851-7_2)
35. Schneider, F.B. : On Concurrent Programming. Graduate Texts in Computer Science. Springer, Berlin (1997). <https://doi.org/10.1007/978-1-4612-1830-2>
36. Stockmeyer, L.J.: Vishkin U: Simulation of parallel random access machines by circuits. *SIAM Journal of Computing* **13**(2), 409–422 (1984). <https://doi.org/10.1137/0213027>
37. Trahan, J.L., Vedantham, S.: Analysis of PRAM instruction sets from a log cost perspective. *Int. J. Found. Comput. Sci.* **5**(3–4), 231–246 (1994). <https://doi.org/10.1142/S0129054194000128>
38. Wirsing, M. : Algebraic specification. In: van Leeuwen, J. (ed) Handbook of Theoretical Computer Science vol. B. pp 675–788. Elsevier, Amsterdam (1990). <https://doi.org/10.1016/B978-0-444-88074-1.50018-4>
39. Wloka, M.G. : Parallel VLSI synthesis. PhD Thesis, Department of Computer Science, Brown University, Providence, RI (1991)
40. Xu, J., Zhang, X., Urban, C. : Mechanising Turing machines and computability theory in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds) ITP 2013. Lecture Notes in Computer Science, vol. 7998. pp 147–162. Springer, Berlin (2013) [https://doi.org/10.1007/978-3-642-39634-2\\_13](https://doi.org/10.1007/978-3-642-39634-2_13)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.