Formalizing SANE Virtual Processor in thread algebra ⋆

Thuy Duong Vu[1] and Chris Jesshope[2]
{tdvu,jesshope}@science.uva.nl

[1] Sectie Software Engineering,
[2] Computer Systems Architecture Group,
University of Amsterdam,
The Netherlands

Abstract. *The SANE Virtual Processor* (SVP) is a fine-grain, thread-based model of concurrent program composition developed and used at the University of Amsterdam as a basis for designing and programming many-core chips. Its design goal was to support dynamic concurrency and hence support self-adaptive systems within the AETHER collaborative European project. It provides an effective solution for programming chip multiprocessor systems [11, 12, 17]. In this paper, we take thread algebra [10], a semantics for recent object-oriented programming languages such as C# and Java, as a theoretical framework to the verification and evaluation of SVP. We show how a SVP program behavior can be determined in TA_{svp}, an extension of thread algebra with the features of SVP, and prove that SVP programs satisfy the determinism property, i.e. the programs always give the same result, a key property of the sequential paradigm that SVP will replace.
Keywords: SANE Virtual Processor, microthreading, thread algebra

1 Introduction

The SANE Virtual Processor (SVP) was defined as a concurrent programming model with two broad requirements. Firstly, that it provide a suitable substitute for the ubiquitous sequential model of program composition while retaining the latter's properties of safe composition of programs. The required properties in this case are freedom from deadlock under composition and determinism of results under whatever schedule is used to execute the equivalent concurrent program. The second broad requirement of the model is that it should have scalable implementations in silicon as many-core chips. The model is defined by a small number of actions used to create and asynchronously manage the execution of concurrent SVP programs. These actions capture concurrency, implicit communication and resource management, and using these abstractions

⋆ The work presented in this paper is supported by NWO (Netherlands Organisation for Scientific Research) in the "Foundations for Massively Parallel on-chip Architectures using Microthreading" project.

we aim to develop an understanding of self-adaptive computational systems in the AETHER collaborative European project (http://www.aether-ist.org/).

Programs are composed in SVP by executing a *create* action on a fragment of code (a microthread), which dynamically creates a parameterized family of thread contexts based on that fragment and which may all execute concurrently, together with the creating thread. Every thread in a family is identified by a unique index value in its context. Further actions are defined to manage infinite families of threads and the termination of families, both destructively and by preempting the concurrent program defined by a family. The create action is used in place of the sequential composition actions of looping (both for and while loops) and function invocation.

This paper tackles the first broad requirement. However, it should be emphasized that this is not a theoretical exercise. Implementations of all of the above actions have been evaluated using an emulated many-core processor in which these actions are implemented as instructions in the processors' instruction set. Silicon implementations have also been investigated. This paper represents therefore, the application of theory to a very practical situation. This model will provide the issues of scalability and code compatibility that will be required for future generations of many-core processor chips. For more detail on this aspect of the research, the interested reader is directed towards the prior work dating back some ten years [11, 18, 16, 12, 17].

We aim to give a formal proof for the determinism property of SVP programs. We will need to define formally the semantics and the memory model of SVP. We take thread algebra [10], a semantics for recent object-oriented programming languages such as C# and Java, as a theoretical framework to the verification and evaluation of SVP. In particular, we extend thread algebra with the features of SVP to TA_{svp} (thread algebra for SVP), and show that TA_{svp} indeed is a formal semantics of SVP. To interpret the memory model of SVP, we adapt the concept of a Maurer machine [9], an extension of a Maurer computer [19, 20], with the features of SVP. The reason to use Maurer computers is that they are closer to real computers than the well-known models such as register machines, multistack machines and Turing machines (see e.g. [14]). Threads in TA_{svp} can perform operations to transform states of a Maurer machine. The determinism property of SVP programs, i.e. concurrent SVP programs always give the same result as the result obtained when they are executed sequentially, therefore, can be proved as program behaviors and memory states are represented as threads in TA_{svp} and states of a Maurer machine.

Our work, like the previous works given in [9, 6–8, 24], is a part of a project investigating microthreading in a collaboration between the Computer Systems Architecture group and Sectie Software Engineering at the University of Amsterdam. We note that a denotational semantics and a structural operational semantics for TA_{svp} can be found in our other paper [24]. The other desired property of SVP programs, namely freedom from deadlock under composition, is also proven in that paper.

The structure of this paper as follows. Section 2 summaries the informal semantics of the SVP model. Section 3 defines TA_{svp} (thread algebra for SVP), a theoretical frame work for the verification and evaluation of SVP. Section 4 models the memory of SVP with the use of Maurer computers. Section 5 illustrates the programming language μTC (*microthreaded C*) [15], a realization of SVP, as a simple programming language $\mathcal{L}_{svp}$ and determines $\mathcal{L}_{svp}$ program behaviors in TA_{svp}. We also prove the determinism property of SVP programs in this section. The paper is ended with some concluding remarks in Section 6.

2 A summary of the SVP model

The SVP model provides five actions in order to create and manage concurrency. These actions replace those normally used to construct sequential programs, namely loops and function calls. Three of these actions are used to parallelize sequential programs and the other two are used for concurrency engineering, i.e. the self-adaptive aspects of the model.

The model is designed to capture the precise functionality of an equivalent sequential program, while relaxing as far as is practical the order of execution of the instructions. It captures a more relaxed partial order of instruction execution than the sequential program, although a more restricted partial order when compared to a dataflow representation of the same program. Because this paper is concerned with the determinacy of the results compared to the equivalent sequential program, only the parallelizing actions will be described. They include the *create, sync* and *break* actions.

The create action defines a *family* of threads based on a single fragment of code. The result of the create action is the creation of an ordered set of thread contexts defined by parameters to the action. These parameters define the code used, the size of the context required and the number of threads to be created. The latter is defined by a triple defining an index range and each thread has its context initialized with a unique value from this range. This provides an analogy to the limits defied within a loop in the sequential model. This family of threads is identified with a unique name so that it may be monitored and controlled by the other actions.

A thread's code may itself contain create actions and this provides for hierarchy in the composition of programs in the SVP model. Nested loops are just one example of a sequential construct that maps to nested creates. Function calls are another, as they are also translated into SVP create actions. A created function is a family with a singleton thread that executes concurrently and asynchronously with its creating thread. However, any thread creating a subordinate family of threads, be it a loop or a function equivalent, must wait for the entire family of threads to complete before any results the family has written to memory are fully defined. As a result, no two concurrent threads can read and write the same location in memory (nor both write to the same location). These constraints are enforced in the compilation of an SVP program. This model allows for significant concurrency without the cost associated with

dataflow models. Unlike dataflow however, this constraint on concurrency allows for expressive stored-variable semantics on memory locations rather than the single assignment model of memory used in dataflow.

A thread creating a subordinate family can detect its termination using an SVP *sync* action. This identifies the family by name so that multiple concurrent families can be created and synchronized from within a single thread. The termination of the family also guarantees that all memory locations written to the shared memory by its threads have completed. Note that no guarantees are made on shared memory latency as it is deemed to be asynchronous and distributed. The sync action provides a return code that specifies how a family was terminated and can provide a return value in the case of a *break* action.

The break action is provided to allow for the creation of dynamically bounded families of threads. In such circumstances, a semi-infinite range of index values is specified in the family's parameters and any thread in the family may terminate the creation of new contexts using the break action and return a scalar value (e.g. an index or pointer) back to the creating thread via the sync action. This construct is the SVP concurrent equivalent of a while loop in a sequential program. Because of lack of space and simplicity, in this first report of our approach, we will ignore the existence of sync and break actions.

Communication between threads is achieved by two mechanisms. The first is the bulk synchronization on memory described above, where read after writes to asynchronous shared memory are synchronized by a sync action between the family that writes to memory and the family that reads from memory. The second mechanism uses a fine-grain synchronizing memory that stores the context of scalar variables for each thread created. This acts like a stack in a conventional sequential machine. Each word in this memory contains synchronization bits that identify whether the word has been written to, enabling threads to block on a read and to be rescheduled when data is written. This allows synchronization on operations within a thread, for example loading data from asynchronous shared memory and subsequently using it. It also provides blocking reads on communication between concurrently executing threads. The latter enables dependencies to be defined between the creating thread and the first thread it creates, as well as between a created thread and its successor in index sequence. It enables dataflow-like scheduling of instructions within a concurrently executing family of threads. Note that the sequential equivalent of this communication is a scalar value assigned within a loop body. This dependency chain is captured by defining a *shared* variable in each thread's context, where each thread has read only access to its predecessor's shared variable (the first thread in a family has access to an initializing variable defined in the creating thread).

The exact relationship between a sequential program and its SVP equivalent is outside of the scope of this paper, however the goal is to program multicores chips using code compiled from sequential languages and the goal of this paper is to ensure that the sequential program's determinism is retained by this model. A realization of the SVP model has been developed at the University

of Amsterdam and called *microthreaded C* or μTC for short. More information, including program examples in μTC can be found in the following report [15].

3 Thread algebra for SVP (TA_{svp})

In this section, we introduce TA_{svp}, a theoretical framework for the verification and evaluation of SANE Virtual Processors.

3.1 Basic thread algebra for SANE virtual processors (BTA_{svp})

BTA_{svp} (*basic thread algebra for SVP*) is defined as a semantics for SVP sequential programs. It is based on *basic thread algebra* (BTA) [10], a semantics for sequential programming languages which was first introduced as *basic polarized process algebra* (BPPA) in [5].

We assume the existence of a fixed but arbitrary set $\mathcal{BA}$ of *basic actions* in BTA_{svp}. Each basic action $a \in \mathcal{BA}$ of a thread is taken as a command to the execution environment of the thread. This command is accepted or rejected depending on a boolean value $?a$ produced by the execution environment. If $?a = T$ (`true`) then the action a is *independent* and can be executed otherwise it is *blocked*. The execution environment cannot do anything with it. Here the term "independent" means that the execution of action a does not depend on any other actions. At completion of the processing of a basic action $a \in \mathcal{BA}$, the execution environment produces a reply value $\mathtt{y}_a$. This reply is either T or F (`false`) and is returned to the thread concerned.

BTA_{svp} (basic thread algebra for SVP) is the extension of BTA with the *independence* and *reply* [8] conditional operators. Hence, the set A of finite threads in BTA_{svp} is defined inductively with the operators in BTA and additionally with the following operators:

- *Successful termination*: $S \in A$ yields successful terminating behavior.
- *Unsuccessful termination* or *deadlock*: $D \in A$ represents inactive behavior.
- *Postconditional composition*: $- \trianglelefteq a \trianglerighteq -$ with $a \in \mathcal{BA}$. The thread $p \trianglelefteq a \trianglerighteq q$, where $p, q \in A$, first performs a and then proceeds with p if T was returned and with q otherwise. In case $p = q$ we abbreviate this thread by the *action prefix* operator: $a \circ -$. In particular, $a \circ p = p \trianglelefteq a \trianglerighteq p$.
- The *independence* conditional operator: $-\lhd ?a \rhd -$ with $a \in \mathcal{BA}$. The thread $p \lhd ?a \rhd q$ behaves as p is a is independent ($?a = T$) and it behaves as q otherwise.
- The *reply* conditional operator: $-\lhd \mathtt{y}_a \rhd -$ with $a \in \mathcal{BA}$. The thread $p \lhd \mathtt{y}_a \rhd q$ behaves as p if the execution of a returns a positive reply T, and it behaves as q otherwise. In fact, $p \trianglelefteq a \trianglerighteq q = a \circ (p \lhd \mathtt{y}_a \rhd q)$.

We note that the reply conditional operator is needed for defining synchronous cooperation of threads. This operator and the independence conditional operator originate from the *conditional* operator [2] defined for process algebra, where the second argument must be T or F. Table 1 represents the axioms for these

$x \lhd T \rhd y = x$	CO1
$x \lhd F \rhd y = y$	CO2
$x \lhd c \rhd x = x$	CO3
$(x \lhd c \rhd y) \lhd c \rhd z = x \lhd c \rhd z$	CO4
$x \lhd c \rhd (y \lhd c \rhd z) = x \lhd c \rhd z$	CO5
$(x \lhd c_1 \rhd y) \lhd c_2 \rhd z = (x \lhd c_2 \rhd z) \lhd c_1 \rhd (y \lhd c_2 \rhd z)$	CO6
$x \lhd c_1 \rhd (y \lhd c_2 \rhd z) = (x \lhd c_1 \rhd y) \lhd c_2 \rhd (x \lhd c_1 \rhd z)$	CO7
$x \trianglelefteq a \trianglerighteq y = a \circ (x \lhd \mathtt{y}_a \rhd y)$	RC

Table 1. Axioms for conditions.

$x \trianglelefteq \mathtt{tau} \trianglerighteq y = \mathtt{tau} \circ x$	IA1
$x \lhd ?\mathtt{tau} \rhd y = x$	IA2
$x \lhd \mathtt{y}_{\mathtt{tau}} \rhd y = x$	IA3

Table 2. Axioms for the concrete internal action.

operators. The constants S and D are similar to the termination ϵ and the deadlock δ used in other process algebras such as CCS [21] and ACP [4].

The *concrete internal action* $\mathtt{tau} \in \mathcal{BA}$ [10] plays a special role. Its execution will never change any state and always produces a positive reply. The axiom for this action is given in Table 2.

We write $p.q$ for threads $p, q \in A$ with each occurrence of S of p replaced by q. This thread executes p and q sequentially.

3.2 Approximation Induction Principle

An *infinite* thread in BTA_{svp} is represented by a *projective sequence* consisting of its finite approximations. These finite approximations are defined inductively by means of the approximation operators $\pi_n(-)$ of depth n of threads with $n \in \mathbb{N}$ whose axioms on finite threads are given as P0-P5 in Table 3. Note that axioms P4 and P5 makes use of the assumption that $\mathcal{BA}$ is finite.

A *projective sequence* is a sequence $(p_n)_{n\in\mathbb{N}}$ such that $\pi_n(p_{n+1}) = p_n$ for all $n \in \mathbb{N}$.

The *Approximation Induction Principle* (AIP) in Table 3 states that two threads are considered identical if their finite approximations at every depth are identical. We write A^∞ for the set of (finite and infinite) threads, and $\pi_n(p)$ for the projection at depth n of a thread $p \in A^\infty$. A^∞ is called a *projective limit model*. For infinite threads $p_1, \ldots, p_n \in A^\infty$, we define for all $n \in \mathbb{N}$ that

$$\pi_n(p_1 \ldots p_n) = \pi_n(\pi_n(p_1) \ldots \pi_n(p)).$$

3.3 The current thread persistence with blocking strategy in TA_{svp}

We now extend BTA_{svp} with the basic interleaving strategy that is used in SVP, called the *current thread persistence with blocking* and written as $\|_{ctpb}$.

$\pi_0(x)=D$	P0
$\pi_{n+1}(S)=S$	P1
$\pi_{n+1}(D)=D$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y)=\pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3
$\pi_{n+1}(x \lhd \mathtt{y}_a \rhd y)=\pi_{n+1}(x) \lhd \mathtt{y}_a \rhd \pi_{n+1}(y)$	P4
$\pi_{n+1}(x \lhd ?a \rhd y)=\pi_{n+1}(x) \lhd ?a \rhd \pi_{n+1}(y)$	P5
If $\pi_n(x) = \pi_n(y)$ for all $n \in \mathbb{N}$ then $x = y$	AIP

Table 3. Axioms for approximation operators and induction principle.

$x \trianglelefteq \mathtt{NT}(\langle (z_1)\rangle \frown \ldots \frown \langle z_n\rangle) \trianglerighteq y = \mathtt{NT}(\langle z_1\rangle \frown \ldots \frown \langle z_n\rangle) \circ x$	PerfectNT
$\pi_{n+1}(\mathtt{NT}(\langle z_1\rangle \frown \ldots \frown \langle z_k\rangle) \circ x) = \mathtt{NT}(\langle \pi_n(z_1)\rangle \frown \ldots \frown \langle \pi_n(z_k)\rangle) \circ \pi_n(x)$	PNT

Table 4. Axioms for approximation operators with thread creation.

Thread creation First of all, we will explain how a family of threads in SVP is created. We assume that there is no resource deadlock in thread creation. Hence thread creation considered here is a perfect forking.

Let $\langle\rangle$ denote the empty sequence, $\langle p\rangle$ the sequence having p as sole element, and $\alpha \frown \beta$ the concatenation of finite sequences α and β.

The creation of a family of threads in TA_{svp} is given by the *forking postconditional composition* operator $-\trianglelefteq \mathtt{NT}(\alpha) \trianglerighteq -$ where α is a sequence of threads. The thread $r = p \trianglelefteq \mathtt{NT}(\alpha) \trianglerighteq q$ for some threads p, q is called the *creating* thread of the threads in α. $\mathtt{NT}(\alpha)$ is considered as a *thread forking action*. Like a real action, its execution also produces a reply. Since we only deal with perfect forking in this paper, this reply is always T. The axioms for thread creation are given in Table 4. We note that thread creation has been considered for thread algebra in [10, 8] with perfect forking and imperfect forking (forking off a thread may be blocked and/or fail). Our axiom PNT coincides with the axiom for thread creation in [10, 8] in the case that the sequence of threads to be created is of length one.

In the thread creation $\mathtt{NT}(\langle p_1\rangle \frown \ldots \frown \langle p_n\rangle)$, we say that $p_1, \ldots, p_n$ are the threads in the same family, and are also in the same family with the creating thread. Moreover, p_i is a *predecessor* of p_j for all $1 \leq i < j \leq n$, and p_n is a predecessor of its creating thread. If $r \neq p_n$ is a predecessor of the creating thread then r is also a predecessor of $p_1, \ldots, p_n$.

In SVP, the blocking of a thread in a sequence of concurrent threads is allowed in a very restricted manner, depending only on its predecessors. In other words, a thread may only be waiting for some data produced by its predecessors. Hence, dependencies between threads in SVP can be represented as an acyclic graph, which in turn ensures freedom from *communication-deadlock* in the model SVP (see [24]).

The current thread persistence with blocking We assume the existence of a special action $\mathtt{swch} \in \mathcal{BA}$ to switch off the current thread to another thread in the sequence of concurrent threads. Like the concrete internal action $\mathtt{tau}$, the

$x \trianglelefteq \mathtt{swch} \trianglerighteq y = \mathtt{swch} \circ x$	SWCH1
$x \triangleleft ?\mathtt{swch} \triangleright y = x$	SWCH2
$x \triangleleft \mathtt{y}_{\mathtt{swch}} \triangleright y = x$	SWCH3

Table 5. Axioms for `swch`.

$\|_{ctpb}\ (\alpha) = \|^0_{ctpb}\ (\alpha)$	Ctpb0
$\|^{\mathtt{length}(\alpha)}_{ctpb}\ (\alpha) = D$	Ctpb1
$\|^k_{ctpb}\ (\langle\rangle) = S$	Ctpb2
$\|^k_{ctpb}\ (\langle S\rangle \frown \alpha) = \|^k_{ctpb}\ (\alpha)$	Ctpb3
$\|^k_{ctpb}\ (\langle D\rangle \frown \alpha) = D$	Ctpb4
$\|^k_{ctpb}\ (\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) =$ $(\|^0_{ctpb}\ (\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \|^0_{ctpb}\ (\langle y\rangle \frown \alpha)) \triangleleft ? a \triangleright (\mathtt{tau} \circ \|^{k+1}_{ctpb}\ (\alpha \frown \langle x \trianglelefteq a \trianglerighteq y\rangle))$	Ctpb5
$\|^k_{ctpb}\ (\langle x \trianglelefteq \mathtt{swch} \trianglerighteq y\rangle \frown \alpha) = \mathtt{tau} \circ \|^0_{ctpb}\ (\alpha \frown \langle x\rangle)$	Ctpb6
$\|^k_{ctpb}\ (\langle x \trianglelefteq \mathtt{NT}(\beta) \trianglerighteq y\rangle \frown \alpha) = \mathtt{tau} \circ \|^k_{ctpb}\ (\beta \frown \langle x\rangle \frown \alpha)$	Ctpb7
$\|^k_{ctpb}\ (\langle x \triangleleft ? a \triangleright y\rangle \frown \alpha) = \|^k_{ctpb}\ (\langle x\rangle \frown \alpha) \triangleleft ? a \triangleright \|^k_{ctpb}\ (\langle y\rangle \frown \alpha)$	Ctpb8

Table 6. Axioms for current thread persistence with blocking. Here $a \in \mathcal{BA}$.

execution of `swch` will never change any state and always produces a positive reply. The switching off may speed up processors in some cases. The axiom for the `swch` action is given in Table 5.

The axioms for current thread persistence with blocking $\|_{ctpb}$ are given in Table 6. Initially, $\|_{ctpb}\ (\alpha) = \|^0_{ctpb}$. The superscript k used in $\|^k_{ctpb}\ (\alpha)$ denotes the number of the blocked threads in α. If all the threads are blocked then communication-deadlock occurs. The composition of an empty sequence of threads will terminate successfully. If the first thread of the sequence is terminated then the execution proceeds with the subsequent threads. If the first thread is in deadlock then whole system is in deadlock. In the remaining case, the system will execute the actions of the first thread until there is a blocked action, or the action `swch`. The control flow then proceeds with the next thread in the sequence. The first thread meanwhile is put to the end of the sequence in a round-robin fashion. When creating a new family of threads or switching off to another thread, the action `tau` will arise as a residue to keep pace with other threads in the sequence. We note that the threads are supposed initially not to contain any guards.

The axioms in Table 6 are defined for finite threads only. For a sequence of arbitrary (finite or infinite) threads $\alpha = \langle p_1\rangle \frown \ldots \frown \langle p_m\rangle$, $\|_{ctpb}\ (\alpha)$ is determined by its projective sequence where $\pi_n(\|_{ctpb}\ (\alpha)) = \pi_n(\|_{ctpb}\ (\langle \pi_n(p_1)\rangle \frown \ldots \frown \langle \pi_n(p_m)\rangle))$.

3.4 Synchronous cooperation of threads in TA_{svp}

In this section, we extend TA_{svp} with a form of synchronous cooperation of threads in SVP.

$\mathtt{tau}\vert\xi$	$= \xi$
$(\xi\vert\xi')\vert\xi''$	$= \xi\vert(\xi'\vert\xi'')$
$(\xi\vert\xi')\vert\xi''$	$= (\xi'\vert\xi)\vert\xi''$
$x \lhd ?(\xi\vert\xi') \rhd y$	$= x \lhd ?\xi' \rhd y$
$x \lhd \mathtt{y}_{\xi\vert\xi'} \rhd y$	$= x \lhd \mathtt{y}_{\xi'} \rhd y$

Table 7. Conditions on the synchronization function. Here $\xi, \xi', \xi'' \in \mathcal{CA}$.

$\Vert_{scb}\ (\langle\rangle) = S$	Scb1
$\Vert_{scb}\ (\alpha \frown \langle S\rangle \frown \beta) = \Vert_{scb}\ (\alpha \frown \beta)$	Scb2
$\Vert_{scb}\ (\alpha \frown \langle D\rangle \frown \beta) = D$	Scb3
$\Vert_{scb}\ (\langle x_1 \trianglelefteq \xi_1 \trianglerighteq y_1\rangle \frown \ldots \frown \langle x_n \trianglelefteq \xi_n \trianglerighteq y_n\rangle) = \psi_0^\emptyset$	Scb4
$\Vert_{scb}\ (\alpha \frown \langle x \trianglelefteq \mathtt{NT}(\langle z_1\rangle \frown \ldots \frown \langle z_n\rangle) \trianglerighteq y\rangle \frown \beta) =$ $\Vert_{scb}\ (\alpha \frown \langle \mathtt{tau} \circ z_1\rangle \frown \ldots \frown \langle \mathtt{tau} \circ z_n\rangle \frown \langle \mathtt{tau} \circ x\rangle \frown \beta)$	Scb5
$\Vert_{scb}\ (\alpha \frown \langle x \lhd c \rhd y\rangle \frown \beta) = \Vert_{scb}\ (\alpha \frown \langle x\rangle \frown \beta) \lhd c \rhd \Vert_{scb}\ (\alpha \frown \langle y\rangle \frown \beta)$	Scb6

Table 8. Axioms for synchronous cooperation with blocking.

Atomic actions and concurrent actions Like [8], we assume a fixed but arbitrary set $\mathcal{AA}$ of *atomic actions* ($\mathtt{tau} \in \mathcal{AA}$), a fixed but arbitrary set $\mathcal{CA} \supseteq \mathcal{AA}$ of *concurrent actions*, and a fixed but arbitrary synchronization function $\vert : \mathcal{CA} \times \mathcal{CA} \to \mathcal{CA}$ satisfying that:

- $\mathtt{tau} \in \mathcal{AA}$;
- for an action $\xi \in \mathcal{CA}$ if and only if $\xi \in \mathcal{AA}$ or there exist ξ', ξ'' such that $\xi = \xi'\vert\xi''$;
- for an action $\xi \in \mathcal{CA}$ there is a boolean value $?\xi$ stating that ξ is independent or blocked.

Hence, each concurrent action can be reduced to one of the following form:

- a with $a \in \mathcal{AA}$;
- $a_1\vert\ldots\vert a_n$ with $a_1, \ldots a_n \in \mathcal{AA}$ for $n > 1$;

The axioms for concurrent actions are given in Table 7. We assume that the independence of a concurrent action and its reply depend only on its last atomic action. The set $\mathcal{BA}$ of basic actions is extended with this set $\mathcal{CA}$ of concurrent actions.

The synchronous cooperation with blocking strategy The synchronous cooperation of threads in SVP is dynamic. We intend to perform simultaneously the maximum number of independent actions from concurrent threads. This might speed up processors [23]. We call this interleaving strategy the *synchronous cooperation with blocking*, denoted by $\Vert_{scb}$.

The axioms for synchronous cooperation with blocking $\Vert_{scb}$ are given in Table 8. The threads are supposed initially not to contain any guards. In this strategy, the composition of an empty sequence of threads is a termination. If a

thread is in deadlock then the whole system is also in deadlock. If a thread is terminated then this thread is simply removed from the sequence. If all threads are blocked, deadlock will occur. If all threads are deadlock free, the synchronous cooperation strategy will execute simultaneously all and only independent threads. The indexes of these threads are contained in a set I. We note that the $\mathtt{tau}$ actions will arise when a family of thread is created in order to keep pace with other threads in the sequence. The auxiliary function ψ_i^I is defined by:

$$\begin{array}{l}\psi_i^I = \psi_{i+1}^{I\cup\{i+1\}} \lhd ?\xi_{i+1} \rhd \psi_{i+1}^I\\ \psi_n^\emptyset = D\\ \psi_n^I = |_{i\in I}\xi_i\circ \|_{scb}\ (\langle\chi_i^I(x_1 \trianglelefteq \xi_1 \trianglerighteq y_1)\rangle \frown \ldots \frown \langle\chi_n^I(x_n \trianglelefteq \xi_n \trianglerighteq y_n)\rangle) \quad (I \neq \emptyset)\end{array}$$

where $\chi_i^I(x \trianglelefteq \xi \trianglerighteq y) = \begin{cases} x \lhd \mathtt{y}_\xi \rhd y \text{ if } i \in I,\\ x \trianglelefteq \xi \trianglerighteq y \text{ otherwise.}\end{cases}$

For a sequence of arbitrary infinite threads $\alpha = \langle p_1\rangle \frown \ldots \frown \langle p_m\rangle$, $\|_{scb}\ (\alpha)$ is determined by its projective sequence where $\pi_n(\|_{scb}\ (\alpha)) = \pi_n(\|_{scb}\ (\langle\pi_n(p_1)\rangle \frown \ldots \frown \langle\pi_n(p_m)\rangle))$.

3.5 Basic terms and guarded recursive specifications in TA_{svp}

Basic terms We now denote $\mathcal{T}_{svp}$ as the set of all closed terms over the signature of TA_{svp}. The set $\mathcal{B}$ of *basic terms* is inductively defined by the following rules:

- $S, D \in \mathcal{B}$;
- if $p \in \mathcal{B}$ then $\mathtt{tau} \circ p \in \mathcal{B}$;
- if $p, q \in \mathcal{B}$ and $a \in \mathcal{BA}$ then $p \trianglelefteq a \trianglerighteq q \in \mathcal{B}$;
- if $p, q \in \mathcal{B}$ then $p \trianglelefteq \mathtt{swch} \trianglerighteq q \in \mathcal{B}$;
- if $p, r_1, \ldots, r_n \in \mathcal{B}$ then $\mathtt{NT}(\langle r_1\rangle \frown \ldots \frown \langle r_n\rangle) \circ p \in \mathcal{B}$;
- if $p, q \in \mathcal{B}$ and $a \in \mathcal{BA}$ then $p \lhd \mathtt{y}_a \rhd q \in \mathcal{B}$;
- if $p, q \in \mathcal{B}$ and $a \in \mathcal{BA}$ then $p \lhd ? a \rhd q \in \mathcal{B}$;

We write $\mathcal{B}^0$ for the set of all terms from $\mathcal{B}$ in which no subterm of the form $p \trianglelefteq \mathtt{NT}(\alpha) \trianglerighteq q$ occurs.

Lemma 1. *For all $p_1, \ldots, p_n \in \mathcal{B}$, there is a term $q \in \mathcal{B}^0$ such that $\|^k_{ctpb}\ (\langle p_1\rangle \frown \ldots \frown \langle p_n\rangle) = q$ is derivable from the axioms of TA_{svp}.*

Theorem 1. *(**Elimination**). For all $p \in \mathcal{T}_{svp}$, there is a term $q \in \mathcal{B}$ such that $p = q$ is derivable from the axioms of TA_{svp}.*

The proofs of Lemma 1 and Theorem 1 are given in [24].

Guarded recursive specifications in TA_{svp} We assume the existence of a fixed but arbitrary set of variables $\mathcal{X}$. Let $X \subseteq \mathcal{X}$. We write $\mathcal{T}_{svp}^X$ for the set of all terms from $\mathcal{T}_{svp}$ in which no other variables than the ones in X have free occurrences. The set $\mathcal{G}$ of *guarded* terms is defined inductively as follows:

- $S, D \in \mathcal{G}$;

$\langle x_i\|E\rangle = t_i(\langle x_1\|E\rangle, \ldots, \langle x_n\|E\rangle)$ $(i \in [1,n])$	RDP
If $y_i = t_i(y_1, \ldots, y_n)$ for $i \in [1,n]$ then $y_i = \langle x_i\|E\rangle$ $(i \in [1,n])$	RSP

Table 9. Axioms for the constants $\langle X|E\rangle$.

- if $\xi \in \mathcal{BA}$ and $t_1, t_2 \in \mathcal{T}_{svp}$ then $t_1 \trianglelefteq \xi \trianglerighteq t_2 \in \mathcal{G}$;
- if $t_1, t_2, t_3 \in \mathcal{T}_{svp}$ then $t_1 \trianglelefteq \mathtt{NT}(t_3) \trianglerighteq t_2 \in \mathcal{G}$;
- if $\xi \in \mathcal{BA}$ and $t_1, t_2 \in \mathcal{G}$ then $t_1 \lhd \mathtt{y}_\xi \rhd t_2 \in \mathcal{G}$;
- if $\xi \in \mathcal{BA}$ and $t_1, t_2 \in \mathcal{G}$ then $t_1 \lhd ?\xi \rhd t_2 \in \mathcal{G}$;
- if $t_1, \ldots, t_n \in \mathcal{G}$ then $\|_{ctpb}\ (\langle t_1\rangle \curvearrowright \ldots \curvearrowright \langle t_n\rangle) \in \mathcal{G}$.
- if $t_1, \ldots, t_n \in \mathcal{G}$ then $\|_{scb}\ (\langle t_1\rangle \curvearrowright \ldots \curvearrowright \langle t_n\rangle) \in \mathcal{G}$.

A *finite recursive specification* E is a finite set $\{x_i = t_i | i \in [1,n]\}$ of recursive equations where t_i, for all $1 \leq i \leq n$, are terms in $\mathcal{T}_{svp}^{\{x_1,\ldots,x_n\}}$. The finite recursive specification E is *guarded* if for all $1 \leq i \leq n$, t_i are guarded.

Theorem 2. *A guarded recursive specification determines a unique solution.*

The proof of the previous theorem can be obtained in the same line as the proof of Theorem 5 in [8]. If E is a guarded recursive specification and x a recursive variable in E, then $\langle x|E\rangle$ denotes the thread that has to be substituted for x in the solution for E. This thread is called *regular*. The axioms for guarded recursive specifications are given in Table 9, where RDP and RSP refer to *Recursive Definition Principle* and *Recursive Specification Principle* as in other process algebras (see e.g. [13]).

In [24], we have given both a denotational semantics [3] and a structural operational semantics [22, 1] for TA_{svp}, and shown that threads in TA_{svp} are communication-deadlock free. In Section 5.3, we will see how a SVP program behavior is represented as a thread in TA_{svp}. This means that SVP programs are communication-deadlock free, a desired property for the SVP model.

4 Memory model for SVP

This section interprets the memory of the SVP model with the use of Maurer computers [19].

4.1 Maurer computer

A Maurer computer has a memory and operations. The contents of all memory elements construct a state of the computer. This state can be transformed to another state when a certain operation is performed. We recall the definition of Maurer computers from [20, 9]. A *Maurer computer* C consists of the following components:

- a set M;
- a set B with $|B| \geq 2$;

- a set $\mathcal{S}$ of functions $s : M \to B$;
- a set $\mathcal{O}$ of functions $O : \mathcal{S} \to \mathcal{S}$;

and satisfies the following conditions:

- $s_1, s_2 \in \mathcal{S}$, $M' \subseteq M$ and $s_2 : M \to B$ is such that $s_3(x) = s_1(x)$ if $x \in M'$ and $s_3(x) = s_2(x)$ if $x \notin M'$, then $s_3 \in \mathcal{S}$;
- if $s_1, s_2 \in \mathcal{S}$ then the set $\{x \in M | s_1(x) \neq s_2(x)\}$ is finite.

M is called the *memory*; the elements of M are called the *locations*; B is called the *base set*; the elements of $\mathcal{S}$ are called the *states*; the elements of $\mathcal{O}$ are called the *operations*. The first condition is satisfied if C is *complete*, i.e. if $\mathcal{S}$ is the set of all functions $s : M \to B$, and the second condition is satisfied if C is *finite*, i.e. M and B are finite sets.

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, and $O \in \mathcal{O}$. Then the *input region* of O, written $\mathtt{IR}$, and the *output region* of O, written $\mathtt{OR}$, are the subsets of M defined as follows:

$$\begin{aligned}\mathtt{IR}(O) = \{x \in M \mid \exists s_1, s_2 \in \mathcal{S} : \forall z \in M : s_1(z) = s_2(z) \wedge \\ \exists y \in \mathtt{OR}(O) : O(s_1)(y) \neq O(s_2)(y)\}, \\ \mathtt{OR}(O) = \{y \in M \mid \exists s \in \mathcal{S} : s(y) \neq O(s)(y)\}\end{aligned}$$

$\mathtt{OR}(O)$ is the set of all memory elements (or locations) that are possibly affected by O; and $\mathtt{IR}(O)$ is the set of all memory elements that possibly affects elements of $\mathtt{OR}(O)$.

4.2 Maurer machines

Threads in TA_{svp} can be used to direct a Maurer machine [9, 8], an extension of a Maurer computer, in performing operations on its states. In this section, we define Maurer machines with the features of memory and synchronization of the SVP model.

We extend Maurer computers $(M, B, \mathcal{S}, \mathcal{O})$ with a set $\mathtt{Act}$, a function $?\_ : \mathtt{Act} \to M$ and a function $[\![\,\_\,]\!] : \mathtt{Act} \to (\mathcal{O} \times M)$ to obtain Maurer machines. For each $a \in \mathtt{Act}$, we write $m_a^?$ for the unique $m \in M$ such that $?a = m$. Furthermore, we write O_a and m_a for the unique $O \in \mathcal{O}$ and $m' \in M$, respectively, such that $[\![\, a \,]\!] = (O, m')$.

A *Maurer machine* is a tuple $H = (M, B, \mathcal{S}, \mathcal{O}, \mathtt{Act}, ?\_, [\![\,\_\,]\!])$ where $(M, B, \mathcal{S}, \mathcal{O})$ is a Maurer computer, and:

- $?\_ : \mathtt{Act} \to$ is such that for all $a \in \mathtt{Act}$, $s(m_a^?) \in \{T, F\}$.
- $[\![\,\_\,]\!] : \mathtt{Act} \to (\mathcal{O} \times M)$ is such that for all $a \in \mathtt{Act}$ and $s \in \mathcal{S}$, $s(m_a) \in \{T, F\}$; $O_a(s)$ is defined if $s(m_a^?) = T$ otherwise it is *undefined*, denoted by $\uparrow$;

The elements of $\mathtt{Act}$ are the *actions*, and $?\_$ is the *request function*, and $[\![\,\_\,]\!]$ is the *action interpretation function* of H.

We assume that $\mathtt{Act} = \mathcal{AA} \cup (\mathcal{BA} \setminus \mathcal{CA})$. Let $\sigma(p)$ denote the set of actions $a \in \mathtt{Act}$ occurring in a thread p. We define that

$$\mathtt{IR}(O_p) = \cup_{a \in \sigma(p)} \mathtt{IR}(O_a) \text{ and } \mathtt{OR}(O_p) = \cup_{a \in \sigma(p)} \mathtt{OR}(O_a)$$

The SVP model supports two kinds of memory namely *asynchronous shared memory* and *synchronizing memory*.

- *Asynchronous shared memory* supports bulk synchronization between families of threads and provides the permanent state of a computation. There are two simple rules that can be identified for writing deterministic programs, e.g.:
  - no two concurrently executing threads write to the same location in asynchronous shared memory;
  - no two concurrently executing threads read and write to the same location in asynchronous shared memory.
- *Synchronizing memory* supports communication and synchronization between threads in a family. A location in synchronizing memory accessed by a thread in a family is available in read-only form to the other threads in the family.

We then impose two restrictions on threads in TA_{svp} as follows:

- no two concurrently executing threads write to the same location in the memory, i.e. for two threads p and q that are in the same family or in different families executing concurrently, $\mathtt{OR}(O_p) \cap \mathtt{OR}(O_q) = \emptyset$.
- no two concurrently executing threads from different families of threads read and write to the same location in the memory, i.e. for two concurrent threads p and q that are not in the same family, $\mathtt{IR}(O_p) \cap \mathtt{OR}(O_q) = \emptyset$.

We say that an action a is *independent from* a thread p if a is not waiting for any data produced by p, and p is not waiting for the data produced by a either, i.e. $\mathtt{IR}(O_a) \cap \mathtt{OR}(O_p) = \emptyset$ and $\mathtt{IR}(O_p) \cap \mathtt{OR}(O_a)$.

The request function $?a$ is to request the execution of an action a. This action a can be executed if it is *independent*, acknowledged by $s(m_a^?) = T$. In case $s(m_a^?) = F$, a is *blocked* and cannot be executed.

As mentioned earlier, the dependency of a thread in the SVP model is allowed in a very restricted manner, depending only on its predecessors. In particular, a thread may only be waiting for a data produced by its predecessors. Hence, if thread p is not a predecessor of thread q then $\mathtt{IR}(p) \cap \mathtt{OR}(q) = \emptyset$.

We now say that an action a of a thread p in a sequence of concurrent threads is *independent* if it is not waiting for any data produced by the predecessors of p, i.e. for all predecessors q of p, $\mathtt{IR}(O_a) \cap \mathtt{OR}(O_q) = \emptyset$. This also means that a is independent from all predecessors of p.

The actions `tau` and `swch` are always independent and have no effect on the state space. That is, for all $s \in \mathcal{S}$, $s(m_{\mathtt{tau}}^?) = T$, $s(m_{\mathtt{swch}}^?) = T$, $O_{\mathtt{tau}}(s) = s$, $s(m_{\mathtt{tau}}) = T$, $O_{\mathtt{swch}}(s) = s$, and $s(m_{\mathtt{swch}}) = T$.

For a concurrent action $\xi \in \mathcal{CA}$, where $\xi = a_1 | \ldots | a_n$ with $n > 1$ and $a_i \in \mathcal{AA}$ for $i \in [1, n]$, we define that $s(m_\xi^?) = s(m_{a_n}^?)$ and $s(m_\xi) = s(m_{a_n})$ for all $s \in \mathcal{S}$. Furthermore, $O_\xi(s) = O_{a_n}(\ldots O_{a_1}(s)\ldots)$,

$$\mathtt{IR}(O_\xi) = \cup_{i\in[1,n]}\mathtt{IR}(O_{a_i}) \text{ and } \mathtt{OR}(O_\xi) = \cup_{i\in[1,n]}\mathtt{OR}(O_{a_i}).$$

$x\bullet\uparrow$	$=$	$\uparrow$	
$S\bullet s$	$=$	s	
$D\bullet s$	$=$	$\uparrow$	
$(x\lhd ?\xi\rhd y)\bullet s$	$=$	$x\bullet s$	if $s(m^?_\xi)=T$
$(x\lhd ?\xi\rhd y)\bullet s$	$=$	$y\bullet s$	if $s(m^?_\xi)=F$
$(x\trianglelefteq\xi\trianglerighteq y)\bullet s$	$=$	$x\bullet O_\xi(s)$	if $s(m^?_\xi)=T$ and $O_\xi(s)(m_\xi)=T$
$(x\trianglelefteq\xi\trianglerighteq y)\bullet s$	$=$	$x\bullet O_\xi(s)$	if $s(m^?_\xi)=T$ and $O_\xi(s)(m_\xi)=F$
$(x\trianglelefteq\xi\trianglerighteq y)\bullet s$	$=$	$\uparrow$	if $s(m^?_\xi)=F$
$(x\trianglelefteq \mathtt{NT}(\alpha)\trianglerighteq y)\bullet s$	$=$	$\uparrow$	
$(x\lhd \mathtt{y}_\xi\rhd y)\bullet s$	$=$	$x\bullet s$	if $s(m_\xi)=T$
$(x\lhd \mathtt{y}_\xi\rhd y)\bullet s$	$=$	$y\bullet s$	if $s(m_\xi)=F$
$\wedge_{n\geq 0}\pi_n(x)\bullet s=\uparrow$	$\Rightarrow$	$x\bullet s=\uparrow$	

Table 10. Axioms for the apply operator. Here $\xi\in\mathcal{BA}$.

In TA_{svp}, the simultaneously performing act $\xi=a_1|\ldots|a_n$ occurs only in the case that all actions a_i ($i\in[1,n]$) are independent. Since the dependency of an action depends only on the predecessors of the thread containing that action, there cannot be two actions a_i and a_j for $i,j\in[1,n]$ with $i\neq j$ such that they write to the same location in the memory, or they read and write to the same location in the memory. Therefore, $s(m^?_\xi)=T$ and $O_\xi(s)=O_{a_{i_n}}(\ldots O_{a_{i_1}}(s)\ldots)$ where $i_1..i_n$ is a permutation of $1..n$.

4.3 Applying threads in TA_{svp} to Maurer machines

The *apply* operator $\_\bullet\_$ [9, 8] allows threads to transform states of the Maurer machine H by means of it operations. Such state transformations produce either a state of H or the undefined state $\uparrow$.

Let $p\in\mathrm{TA}_{svp}$ and $s\in\mathcal{S}$, then $p\bullet s$ is the state that results if all actions performed by thread p are processed by the Maurer machine H from initial state H. The processing of an action $\xi\in\mathcal{BA}$ is allowed by the boolean value produced by H contained in memory element $m^?_\xi$. This processing amounts to a state change according to the operation O_ξ. In the resulting state, the reply produced by H is contained in memory element m_ξ. If p is S, then there will be no state change. If p is D, then the result is $\uparrow$. If the current action of p is a thread forking action, then the resulting is also $\uparrow$, since thread forking is carried into effect only if it is put in the context of concurrency. Table 10 represents axioms for the apply operator.

We say that $p\bullet s$ is *convergent* if $\exists n\in\mathbb{N}:\pi_n(p)\bullet s\neq\uparrow$. If $p\bullet s$ is convergent then the *length of the computation* of $p\bullet s$, written $\|p\bullet s\|$, is the least $n\in\mathbb{N}$ such that $\pi_n(s)\bullet S\neq\uparrow$.

Two threads p and q are *state transformer equivalent*, written $p\approx q$, if for all $s\in\mathcal{S}$, $p\bullet s=q\bullet s$.

Lemma 2. *If an action ξ is independent from a thread p then for all $s\in\mathcal{S}$, $p\bullet O_\xi(s)=O_\xi(p\bullet s)$.*

Proof. Straightforward.

5 TA$_{svp}$ as a formal semantics of SVP

This section intends to determine the behaviors of programs (or threads) in μTC, a programming language that realizes the model SVP [15], in the setting of TA$_{svp}$. In order to illustrate our approach, we construct a simple programming language $\mathcal{L}_{svp}$, a subset of the language μTC, with a least collection of primitive statements, but rich enough for important applications. We will show that $\mathcal{L}_{svp}$ threads have a desired property, the determinism property, i.e. the threads should always give the same result as the result obtained when they are executed sequentially.

5.1 The program notation $\mathcal{L}_{svp}$

We assume the existence of a set `Var` of *variables* ranged over $x, y, \ldots$. The program notation $\mathcal{L}_{svp}$ is generated from five kinds of constructs and two composition mechanisms. The constructs used in $\mathcal{L}_{svp}$ are as follows:

- assignment $x{=}e$;
- the constant `swch`, used to switch off the current thread to another thread in the sequence of concurrent threads;
- thread creation $\texttt{create}(X_1, \ldots, X_n)$;
- conditional statement $\texttt{if}(e)\{X\}\{Y\}$;
- while-loop statement $\texttt{while}(e)\{X\}$.

Here x is a variable and e stands for a boolean or an arithmetic expression, whose syntax we do not describe here. The semantics of conditional statements and while-loops is given as in other programming languages. Two composition mechanisms of $\mathcal{L}_{svp}$ are:

1. sequential composition $X; Y$; and
2. concurrent composition $\| \, (\langle X_1 \rangle \frown \ldots \frown \langle X_n \rangle)$ where $\| \in \{\|_{ctpb}, \|_{scb}\}$.

Let X, Y, X_i denote the programs (or threads) in $\mathcal{L}_{svp}$. Then

$$X, Y := x{=}e \mid \texttt{if}(e)\{X\}\{Y\} \mid \texttt{while}(e)\{X\} \mid \texttt{swch} \mid \texttt{create}(X_1, \ldots, X_n) \mid X; Y \mid \, \| \, (\langle X_1 \rangle \frown \ldots \frown \langle X_n \rangle).$$

Threads X and Y in the sequential composition $X; Y$ must be sequential. Furthermore, thread creation $\texttt{create}(X_1, \ldots, X_n)$ must be put in the context of concurrency initially. In μTC, the threads $X_1, \ldots, X_n$ in $\texttt{create}(X_1, \ldots, X_n)$ are identical. However, in the verification and evaluation of SVP, it is not necessary to impose this restriction to programs in $\mathcal{L}_{svp}$. Thus, the threads in a created family in $\mathcal{L}_{svp}$ can be different.

```
thread main()
{
    int *a;
    int fid, t=0, n=3;
    create(fid;0;n-1)
       {
       index int i;
       shared int s=t; /*initializes s in first thread only*/
       s = s + a[i];
       }
    sync(fid);
}
```

Table 11. Example of μTC programs.

5.2 Communication between threads with shared variables in $\mathcal{L}_{svp}$

A variable occurring in a thread can be a *local* variable (defined by the thread itself) or a *global* variable (defined by the creating threads of that thread). A local variable can be *shared*. Every local variable of a thread corresponds to a location in the memory. The set of the locations of all local variables of a thread is called the *working* memory of that thread. A thread can manipulate the values in its working memory, which is inaccessible to other threads from a different family. However, this working memory will be available in read-only form to the subthreads created by the thread itself, and to other threads in the same family if the variable is a shared variable. The communication between threads, as described in Section 2, happens via shared variables. In other words, shared variables define dependency between threads. This dependency depends only on the predecessors of a thread. In particular, if a shared variable s occurs as an input of an assignment or a conditional in a thread then its value is taken as the last value of s produced by the thread and its predecessors.

Example 1. A typical μTC program is given in Table 11. The thread in this table sums the values of array a. Here i for $i \in [0, n-1]$ are the indexes of the subthreads. The instruction $s = s + a[0]$ is never blocked, and the instruction $s = s + a[i]$ of thread i for $i > 0$ will be blocked until the value of s is produced by thread $i - 1$.

Without loss of generality, we can assume that all local variables of concurrently executing threads in $\mathcal{L}_{svp}$ are distinct. Furthermore, a shared variable of a thread does not occur in the predecessors of that thread. Since a thread can write only to its working memory, there cannot be an assignment $x = e$ occurring in that thread where x is not defined by the thread itself.

$\lvert\langle\rangle\rvert$	$= S$
$\lvert a\rvert$	$= a \circ S$
$\lvert a; X\rvert$	$= a \circ \lvert X\rvert$
$\lvert\mathtt{swch}; X\rvert$	$= \mathtt{swch} \circ \lvert X\rvert$,
$\lvert\mathtt{create}(X_1, \ldots, X_n); X\rvert$	$= \mathtt{NT}(X_1, \ldots, X_n) \circ \lvert X\rvert$
$\lvert\mathtt{if}(a)\{X\}\{Y\}; Z\rvert$	$= \lvert X; Z\rvert \trianglelefteq a \trianglerighteq \lvert Y; Z\rvert$
$\lvert\mathtt{if}(T)\{X\}\{Y\}; Z\rvert$	$= \lvert X; Z\rvert$
$\lvert\mathtt{if}(F)\{X\}\{Y\}; Z\rvert$	$= \lvert Y; Z\rvert$
$\lvert\mathtt{while}(a)\{X\}; Z\rvert$	$= \lvert X; \mathtt{while}(a)\{X\}; Z\rvert \trianglelefteq a \trianglerighteq \lvert Z\rvert$
$\lvert\mathtt{while}(T)\{X\}; Z\rvert$	$= \lvert X; \mathtt{while}(T)\{X\}\rvert$
$\lvert\mathtt{while}(F)\{X\}; Z\rvert$	$= \lvert Z\rvert$
$\lvert \parallel_{ctpb} (\langle X_1\rangle \curvearrowright \ldots \curvearrowright \langle X_n\rangle)\rvert$	$= \parallel_{ctpb} (\langle\lvert X_1\rvert\rangle \curvearrowright \ldots \curvearrowright \langle\lvert X_n\rvert\rangle)$
$\lvert \parallel_{scb} (\langle X_1\rangle \curvearrowright \ldots \curvearrowright \langle X_n\rangle)\rvert$	$= \parallel_{scb} (\langle\lvert X_1\rvert\rangle \curvearrowright \ldots \curvearrowright \langle\lvert X_n\rvert\rangle)$

Table 12. Axioms for thread extraction operation.

Example 2. The thread given in Example 1 can be formulated as program X in $\mathcal{L}_{svp}$ below.

$$\begin{aligned} X_0 &:= s_0 = t; s_0 = s_0 + a[0] \\ X_1 &:= s_1 = s_0 + a[1] \\ X_2 &:= s_2 = s_1 + a[2] \\ X &:= \parallel (t = 0; \mathtt{create}(X_0, X_1, X_2)) \end{aligned}$$

where $\parallel \in \{\parallel_{ctpb}, \parallel_{scb}\}$. We note that the sum of the array a is stored in s_2.

5.3 The thread extraction operation

We now consider instructions $x{=}e$ and e of $\mathcal{L}_{svp}$ as the actions in $\mathcal{BA}$, written as $[x = e]$ and $\langle e\rangle$. The behaviors of programs in $\mathcal{L}_{svp}$ are determined by means of the *thread extraction operation* $\lvert\_\rvert$, which assigns a thread in TA_{svp} to a program (or thread) in $\mathcal{L}_{svp}$. Table 12 represents axioms for thread extraction operation. If the behavior of a thread in $\mathcal{L}_{svp}$ cannot be computed according to the thread extraction operation, then it is identified with D. For instance, the behavior of a non-trivial loop in which no action occurs can be identified with D.

Example 3. The behavior of program X in Example 2 can be determined as a thread in TA_{svp} as follows.

$$\begin{aligned} \lvert X_0\rvert &= [s_0 = t] \circ [s_0 = s_0 + a[0]] \circ S \\ \lvert X_1\rvert &= [s_1 = s_0 + a[1]] \circ S \\ \lvert X_2\rvert &= [s_2 = s_1 + a[2]] \circ S \\ \lvert X\rvert &:= \parallel (\langle [t = 0] \circ \mathtt{NT}(\langle\lvert X_0\rvert\rangle \curvearrowright \langle\lvert X_1\rvert\rangle \curvearrowright \langle\lvert X_2\rvert\rangle) \circ S\rangle) \text{ with } \parallel \in \{\parallel_{ctpb}, \parallel_{scb}\}. \end{aligned}$$

Lemma 3. *Let X, Y be two sequential programs in $\mathcal{L}_{svp}$. Then $\lvert X; Y\rvert = \lvert X\rvert.\lvert Y\rvert$.*

Theorem 3. *The behaviors of programs in $\mathcal{L}_{svp}$ are regular threads in TA_{svp}.*

The proof of Lemma 3 is straightforward, and Theorem 3 can be proven by induction on the structure of the programs using Lemma 3.

5.4 Determinism

For each program $X \in \mathcal{L}_{svp}$, we write $\overline{X}$ for the sequential form of X, obtained from X by replacing any subterm of the forms $\mathtt{create}(X_1, \ldots, X_n)$, $\|_{ctpb}\ (\langle X_1\rangle \frown \ldots \frown \langle X_n\rangle)$ and $\|_{scb}\ (\langle X_1\rangle \frown \ldots \frown \langle X_n\rangle)$ with $X_1; \ldots; X_n$.

Theorem 4. *(***Determinism***). Let X be a program in $\mathcal{L}_{svp}$. Then $|X| \approx |\overline{X}|$.*

Proof. See Appendix A.

6 Concluding remarks

In this paper, we have given a formal proof for the determinism property of SANE Virtual Processors (SVP). We taken thread algebra (TA) [10] as a theoretical framework for the verification and evaluation of SVP. In particular, TA has been extended with the features of SVP to TA_{svp} (thread algebra for SVP). We have shown that TA_{svp} indeed is a formal semantics of SVP by assigning a thread in TA_{svp} to a program in $\mathcal{L}_{svp}$, a simple programming language that illustrates the realization μTC of SVP. We have interpreted the memory of SVP with the use of Maurer computers [19, 20], and considered the interaction between threads in TA_{svp} and Maurer machines. Finally, we have proven that $\mathcal{L}_{svp}$ programs always give the same result as the result obtained when they are executed sequentially. Our work together with the work presented in [24] show that the SVP model has the desired properties, namely determinism and freedom from deadlock.

Acknowledgments We thank Jan Bergstra for the fruitful discussions.

A Proofs

In this section, we provide a proof for Theorem 4. We will use some supporting results. The first result states that the execution of concurrent finite threads in TA_{svp} gives the same result as the result obtained when they are executed sequentially.

Lemma 4. *Let $p_1, \ldots, p_n$ be terms in B, and $\| \in \{\|_{ctpb}, \|_{scb}\}$. By Lemma 1 and Theorem 1, there are $q_i \in \mathcal{B}^0$ such that $\|\ (\langle p_i\rangle) = q_i$ for all $i \in [1, n]$. Then*

1. *If for all $1 \leq i < j \leq n$, p_j is not a predecessor of p_i, and $i_1..i_n$ is a permutation of $1..n$ then $\|\ (\langle p_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle) \approx q_1 \ldots q_n$.*
2. *$\|\ (\langle p_1 \ldots p_n\rangle) \approx q_1 \ldots q_n$.*

Proof. We prove (1) only. The proof of (2) can be obtained in the same way. We consider the case $\|{=}\|^k_{ctpb}$. The case $\|{=}\|_{scb}$ is similar. We note that in the interleaving strategy $\|^k_{ctpb}$, the case $k \geq n$ never happens since the threads in

TA_{svp} are communication-deadlock free (see [24]). Let $p = \|^k_{ctpb}\ (\langle p_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$. We prove by induction on $(L(p), n-k)$ where $L(p)$ is defined by

$$\begin{array}{ll} L(S) & = 1, \\ L(D) & = 1, \\ L(p \lhd c \rhd q) & = \max\{L(p), L(q)\} + 1, \\ L(p \trianglelefteq a \trianglerighteq q) & = \max\{L(p), L(q)\} + 1, \\ L(\mathtt{NT}(\langle r_1\rangle \frown \ldots \frown \langle r_n\rangle) \circ p) & = \max\{L(p), L(r_1), \ldots, L(r_n)\} + 1, \\ L(\|\ (\langle q_1\rangle \frown \ldots \frown \langle q_n\rangle)) & = \max\{L(q_1), \ldots, L(q_n)\} + 1. \end{array}$$

Let $q = q_1 \ldots q_n$. We show that $p \approx q$. Let $q' = q_1 \ldots q_{i_1 - 1}$ and $q'' = q_{i_1+1} \ldots q_n$. We consider the following possibilities:

- $p_{i_1} = S$. Then $p \approx \|^k_{ctpb}\ (\langle p_{i_2}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$. By the induction hypothesis, $p \approx q'.q''$. Since $p_{i_1} = S$, $p \approx q$.
- $p_{i_1} = D$. Then $p \approx q \approx D$.
- $p_{i_1} = \mathtt{tau} \circ p'_{i_1}$. Then $p = \mathtt{tau} \circ p'$ where $p' = \|^0_{ctpb}\ (\langle p'_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$. By the induction hypothesis, $p' \approx q'.q'_{i_1}.q''$ where $q_{i_1} = \mathtt{tau} \circ q'_{i_1}$. Since $\mathtt{tau}$ has no effect on the state space $\mathcal{S}$, $p \approx p' \approx q' \approx q$.
- $p_{i_1} = \mathtt{swch} \circ p'_{i_1}$. Similar to the previous case, $p \approx q$.
- $p_{i_1} = p'_{i_1} \lhd c \rhd p''_{i_1}$. Then $p = p' \lhd c \rhd p''$ where $p' = \|^k_{ctpb}\ (\langle p'_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$ and $p'' = \|^k_{ctpb}\ (\langle p''_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$. By the induction hypothesis, we also have $p \approx q$.
- $p_{i_1} = p'_{i_1} \trianglelefteq a \trianglerighteq p''_{i_1}$. Then $p = (p' \trianglelefteq a \trianglerighteq p'') \lhd ?a \rhd \mathtt{tau} \circ p'''$ where $p' = \|^0_{ctpb}\ (\langle p'_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$, $p'' = \|^0_{ctpb}\ (\langle p''_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$ and $p''' = \|^{k+1}_{ctpb}\ (\langle p_{i_2}\rangle \frown \ldots \frown \langle p_{i_n}\rangle \frown \langle p_{i_1}\rangle)$. Hence for all $s \in \mathcal{S}$,

$$p \bullet s = \begin{cases} p' \bullet O_a(s) \lhd O_a(s)(m_a) \rhd p'' \bullet O_a(s) \text{ if } s(m_a^?) = T \\ p''' \bullet s \text{ otherwise.} \end{cases}$$

  Let $q'_{i_1} = \|_{ctpb}\ (\langle p'_{i_1}\rangle)$ and $q''_{i_1} = \|_{ctpb}\ (\langle p''_{i_1}\rangle)$. By the induction hypothesis, $p' \approx q'.q'_{i_1}.q''$, $p'' \approx q'.q''_{i_1}.q''$, and $p''' \approx q$. If $s(m_a^?) = F$ then $p \bullet s = p''' \bullet s \approx q \bullet s$. If $s(m_a^?) = T$ then a is independent from all predecessors of p_{i_1}. This implies that a is independent from q'. Let $s' = q' \bullet s$. It follows from Lemma 2 that $q' \bullet O_a(s) = O_a(q' \bullet s) = O_a(s')$. Furthermore, $s'(m_a^?) = s(m_a^?) = T$. Therefore,

$$\begin{array}{l} p \bullet s = p' \bullet O_a(s) \lhd O_a(s)(m_a) \rhd p'' \bullet O_a(s) \\ \quad \approx (q'.q'_{i_1}.q'') \bullet O_a(s) \lhd O_a(s)(m_a) \rhd (q'.q''_{i_1}.q'') \bullet O_a(s) \\ \quad \approx (q'_{i_1}.q'') \bullet (q' \bullet O_a(s)) \lhd O_a(s)(m_a) \rhd (q''_{i_1}.q'') \bullet (q' \bullet O_a(s)) \\ \quad \approx q'_{i_1}.q'' \bullet (O_a(s')) \lhd O_a(s')(m_a) \rhd q''_{i_1}.q'' \bullet (O_a(s')) \\ \quad = (q'_{i_1} \trianglelefteq a \trianglerighteq q''_{i_1}.q'') \bullet s' \approx (q_{i_1}.q'') \bullet s' \approx q \bullet s. \end{array}$$

  This implies that $p \approx q$.
- $p_{i_1} = \mathtt{NT}(\langle r_1\rangle \frown \ldots \frown \langle r_m\rangle) \circ p'_{i_1}$. Then $p = \mathtt{tau} \circ \|^k_{ctpb}\ (\langle r_1\rangle \frown \ldots \frown \langle r_m\rangle \frown \langle p'_{i_1}\rangle \frown \ldots \frown \langle p_{i_n}\rangle)$. By the induction hypothesis,

$$p \approx q'.r'_1 \ldots r'_m.q'_{i_1}.q'' \approx q'.q_{i_1}.q'' \approx q.$$

The following lemma extends Lemma 4 with the case of infinite threads.

Lemma 5. *Let $p_1, \ldots, p_n$ be infinite threads, and $\| \in \{\|_{ctpb}, \|_{scb}\}$. Then*

1. *If for all $1 \leq i < j \leq n$, p_j is not a predecessor of p_i, and $i_1..i_n$ is a permutation of $1..n$ then $\| (\langle p_{i_1} \rangle \frown \ldots \frown \langle p_{i_n} \rangle) \approx \| (\langle p_1 \rangle) \ldots \| (\langle p_n \rangle)$.*
2. *$\| (\langle p_1 \ldots p_n \rangle) \approx \| (\langle p_1 \rangle) \ldots \| (\langle p_n \rangle)$.*

Proof. We prove (1) only. The proof of (2) is similar. Let $p = \| (\langle p_{i_1} \rangle \frown \ldots \frown \langle p_{i_n} \rangle)$ and $q = \| (\langle p_1 \rangle) \ldots \| (\langle p_n \rangle)$. Since $\pi_k(p) = \pi_k(\| (\langle \pi_k(p_{i_1}) \rangle \frown \ldots \frown \langle \pi_k(p_{i_n}) \rangle))$, there is $N_p \in \mathbb{N}$ such that $p \approx \| (\langle \pi_k(p_{i_1}) \rangle \frown \ldots \frown \langle \pi_k(p_{i_n}) \rangle)$ for all $k \geq N_p$. Furthermore, for all $i \in [1, n]$, there are $N_i \in \mathbb{N}$ such that $p_i \approx \pi_k(p_i)$ for all $k \geq N_i$. Let $N = \max\{N_p, N_1, \ldots, N_n\}$. It follows from Lemma 4 that

$$\begin{aligned} p &\approx \| (\langle \pi_N(p_{i_1}) \rangle \frown \ldots \frown \langle \pi_N(p_{i_n}) \rangle) \approx \| (\langle \pi_N(p_1) \rangle) \ldots \| (\langle \pi_N(p_n) \rangle) \\ &\approx \| (\langle p_1 \rangle) \ldots \| (\langle p_n \rangle) \approx q. \end{aligned}$$

Finally, we can prove our main result as follows.

Proof. (**The proof of Theorem 4**). If X is a sequential program then we are done. In the remaining case, we prove by induction on the structure of X.

- $X = \langle\rangle$. Then $\overline{X} = \langle\rangle$. Hence $|X| \approx |\overline{X}| \approx S$.
- $X = \| (\langle a \rangle)$. Then $\overline{X} = a$. Thus, $|X| \approx a \circ S = |\overline{X}|$.
- $X = \| (\langle \mathtt{if}(a)\{Y\}\{Z\} \rangle)$. Then $\overline{X} = \mathtt{if}(a)\{\overline{Y}\}\{\overline{Z}\}$. By the induction hypothesis, $|X| = \| (\langle |Y| \trianglelefteq a \trianglerighteq |Z| \rangle) \approx \| (\langle |Y| \rangle) \trianglelefteq a \trianglerighteq \| (\langle |Z| \rangle) \approx |\overline{Y}| \trianglelefteq a \trianglerighteq |\overline{Z}| = |\overline{X}|$.
- $X = \| (\langle \mathtt{while}(a)\{Y\} \rangle)$. Then $\overline{X} = \mathtt{while}(a)\{\overline{Y}\}$. Let $Z = \mathtt{while}(a)\{Y\}$. It follows from Lemma 3, Lemma 5 and the induction hypothesis that
  $|X| = \| (\langle |Y|.|Z| \trianglelefteq a \trianglerighteq S \rangle) \approx \| (\langle |Y|.|Z| \rangle) \trianglelefteq a \trianglerighteq S \approx \| (\langle |Y| \rangle).|X| \trianglelefteq a \trianglerighteq S \approx |\overline{Y}|.|X| \trianglelefteq a \trianglerighteq S$,
  $|\overline{X}| = |\overline{Y}|.|\overline{X}| \trianglelefteq a \trianglerighteq S$. This implies that $|X| \approx |\overline{X}|$.
- $X = \| (\langle \mathtt{create}(X_1, \ldots, X_n) \rangle)$. By Lemma 5 and the induction hypothesis, $|X| = \| (\langle |X_1| \frown \ldots \frown \langle |X_n| \rangle \rangle) \approx \| (\langle |X_1| \rangle) \ldots \| (\langle |X_n| \rangle) \approx |\overline{X_1}| \ldots |\overline{X_n}| = |\overline{X}|$.
- $X = \| (\langle X_1; \ldots; X_n \rangle)$. Similar to the previous case, we likewise get $|X| = \| (\langle |X_1| \ldots |X_n| \rangle) \approx \| (\langle |X_1| \rangle) \ldots \| (\langle |X_n| \rangle) \approx |\overline{X_1}| \ldots |\overline{X_n}| = |\overline{X}|$.
- $X = \| (\langle X_1 \frown \ldots \frown \langle X_n \rangle \rangle)$. Similarly, $|X| \approx \| (\langle |X_1| \rangle) \ldots \| (\langle |X_n| \rangle) \approx |\overline{X_1}| \ldots |\overline{X_n}| = |\overline{X}|$.

References

1. L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 197–222. Elsevier, 2001.
2. J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods*, volume F88 of *NATO ASI Series*, pages 1–21, 1992.

3. J.W. Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.
4. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Inform. and Control*, 60 (1-3):109–137, 1984.
5. J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *J. of Logic and Algebraic Programming*, 51:125–156, 2002.
6. J.A. Bergstra and C.A. Middelburg. Simulating turing machines on maurer machines. CS-Report 05-28, Department of mathematics and computer science, Technische Universiteit Eindhoven, 2005.
7. J.A. Bergstra and C.A. Middelburg. Maurer computers for pipelined instruction processing. CS-Report 06-12, Department of mathematics and computer science, Technische Universiteit Eindhoven, 2006.
8. J.A. Bergstra and C.A. Middelburg. Synchronous cooperation for explicit multithreading. CS-Report 06-29, Department of mathematics and computer science, Technische Universiteit Eindhoven, 2006.
9. J.A. Bergstra and C.A. Middelburg. Maurer computers with single-thread control. *Fundamenta Informaticae*, 2007. To appear.
10. J.A. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 2007. To appear. Preliminary version: Computer Science Report PRG0404: Sectie Software Engineering, University of Amsterdam.
11. A. Bolychevsky, C.R. Jesshope, and V. Muchnick. Dynamic sheduling in rics architectures. In *IEE Proceedings Computers and Digital Techniques*, volume 143(5), pages 309–317, 1996.
12. K. Bousias, N.M. Hasasneh, and C.R. Jesshope. Instruction-level parallelism through Microthreading-a scalable Approach to chip multiprocessors. *The Computer Journal*, 49 (2):211–233, 2006.
13. W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer, 2000.
14. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addition-Wesley, Reading, MA, second edition, 2001.
15. C.R. Jesshope. SVP and μTC-A dynamic model of concurrency and its implementation as a compiler target. `http://staff.science.uva.nl/~jesshope/Papers/uTC-paper.pdf`.
16. C.R. Jesshope. Multithreaded microprocessors evolution or revolution. In Sedukhin Omondo, editor, *ACSAC 2003: Advances in Computer Systems Architecture*, pages 21–45, 2003.
17. C.R. Jesshope. Microthreading a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16 (2):209–228, 2006.
18. C.R. Jesshope and B. Luo. Micro-threading: A new approach to future risc. In *ACAC 2000*, pages 31–41. IEEE Computer Society Press, 2000.
19. W.D. Maurer. A theory of computer instructions. *Journal of ACM*, 13(2):226–235, 1966.
20. W.D. Maurer. A theory of computer instructions. *Science of Computer Programming*, 55(1/2):1–19, 2006.
21. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
22. G. Plotkin. A structural approach to operational semantics. Aarhus DAIMI FN-19, Computing Science Department, 1981.
23. T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35 (1):29–63, 2003.
24. T.D. Vu and C.R. Jesshope. Thread algebra for SANE virtual processors. Available at `http://staff.science.uva.nl/~jesshope`.