Mechanistic behavior of single-pass instruction sequences

Bergstra, J.A.; van der Zwaag, M.B.

Publication date
2008

Document Version
Submitted manuscript

Mechanistic Behavior of Single-Pass Instruction Sequences

Jan A. Bergstra
Mark B. van der Zwaag

Section Software Engineering, Informatics Institute, University of Amsterdam

Abstract

Earlier work on program and thread algebra detailed the functional, observable behavior of programs under execution. In this article we add the modeling of unobservable, *mechanistic* processing, in particular processing due to jump instructions. We model mechanistic processing preceding some further behavior as a *delay* of that behavior; we borrow a unary delay operator from discrete time process algebra. We define a mechanistic improvement ordering on threads and observe that some threads do not have an optimal implementation.

1 Introduction

We recall from [3] the notion of an instruction sequence and its (functional) thread extraction. Processing cost constitutes a non-functional aspect of instruction sequences. Below we will define a version of thread extraction that takes the cost of jump instructions into account. The simplest intuition for this cost is processing time, under the assumption that the instruction is used as a machine code which is not further compiled before processing on a suitable machine. The presence of a jump induces a delay which is independent of the size of the jump. We call the thread extracted from an instruction sequence while taking cost of jumps into account its mechanistic behavior. The mechanistic behavior reflects some non-functional properties that arise from the execution mechanism. Given the definition of a mechanistic behavior we can define when an implementation of a thread improves another implementation and when an implementation is (locally) optimal or globally optimal. After clarifying the definitions with a number of examples it is shown that some implementable threads have no optimal implementations: each implementation can be improved.

Mechanistic behavior is an essential ingredient for a theory of instruction sequences. Indeed compilation and code generation can often be viewed as steps

This work has been carried out within and funded by the NWO project "Thread Algebra for Strategic Interleaving".

transforming an instruction sequence into a functionally equivalent one with an improved mechanistic behavior. This paper provides an approach to the quantification of non-functional aspects of instruction sequences from first principles which might eventually enable a useful analysis of compilation methods as well as a principled investigation and explanation of what constitutes the best possible design of a set of instructions for writing machine programs.

Following [3] we use single pass instruction sequences for two reasons: it gives rise to a convenient algebra of instruction sequences and it allows to simplify the definition of thread extraction to a bare minimum. We refer to [4] for further introductory information on instruction sequences and threads.

2 Instruction Sequences

Program algebra (PGA, for ProGram Algebra, see [3]) provides a framework for the understanding of imperative sequential programming. Starting point is the perception of a program as an expression of an *instruction sequence* — a possibly infinite sequence

$$u_1; u_2; u_3; \ldots$$

of *primitive* instructions u_i.

Given a set $\mathcal{A}$ of *basic* instructions, the primitive instructions of PGA are the following:

Basic void instruction. All elements of $\mathcal{A}$, written, typically, as $a, b, \ldots$, can be used as *basic void instructions*. These are regarded as indivisible units and execute in finite time. The associated behavior may modify a state.

Termination instruction. The termination instruction ! yields successful termination of the execution. It does not modify a state, and it does not return a boolean value.

Basic test instruction. A basic instruction $a \in \mathcal{A}$ is viewed as a request to the environment, and it is assumed that upon its execution a boolean value (`true` or `false`) is returned that may be used for subsequent program control. For each element a of $\mathcal{A}$ there is a *positive test* instruction $+a$ and a *negative test* instruction $-a$. When a positive test is executed, the state is affected according to a, and in case `true` is returned, the remaining sequence of actions is performed. If there are no remaining instructions, inaction occurs. In the case that `false` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs. Execution of a negative test is the same, except that the roles of `true` and `false` are interchanged.

Forward jump instruction. For any natural number k, the instruction $\#k$ denotes a jump of length k and k is called the counter of this instruction. If $k = 0$, this jump is to the instruction itself and inaction occurs (one can say that $\#0$ *defines* divergence, which is a particular form of inaction).

If $k = 1$, the instruction skips itself, and execution proceeds with the subsequent instruction if available, otherwise inaction occurs. If $k > 1$, the instruction $\#k$ skips itself and the subsequent $k - 1$ instructions. If there are not that many instructions left in the remaining part of the program, inaction occurs.

In PGA, a *program* is an expression (in a programming language) that represents an instruction sequence. In [3], a hierarchy of programming languages is built, with languages containing constructs of increasing complexity such as labels and goto's, conditionals and while-loops, etc. Programs in these languages can always be projected to an expression at a basic level where it can be mapped to an instruction sequence directly. More specifically, at this basic level PGA allows to build programs from the primitive instructions listed above by means of two composition operators.

First, we have instruction sequence concatenation, written $X; Y$ for instruction sequences X and Y. Concatenation is supposed to be associative, so that its parentheses are usually omitted.

The second PGA operator is repetition, written X^ω, representing the infinite concatenation $X; X; X; \ldots$. Repetition unfolds in the following way: $X^\omega = X; X^\omega$, and if X is an infinite instruction sequence already, we will use $X^\omega = X$.

Below we will restrict attention to instruction sequences that can be written in PGA notation. This means that instruction sequences will be either finite or eventually periodic.

3 Functional and Mechanistic Behaviors

The *execution* of an instruction sequence is single-pass: the instructions are visited in order and are dropped after having been executed. Execution of a basic instruction is interpreted as a request to the execution environment: the environment processes the request and replies with a Boolean value. This has led to the modeling of the functional behavior of instruction sequences as threads, that is, as elements of Basic Thread Algebra (BTA). An interpretation mapping $|\_|$ from instruction sequences to threads is given in Section 4. This interpretation is called *thread extraction*.

Based on a set A of *actions*, which will be used to interpret basic instructions, BTA has the following constants and operators:

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

We use *action prefixing* $a \circ P$ as an abbreviation for $P \unlhd a \unrhd P$ and take $\circ$ to bind strongest. Furthermore, for $n \geq 1$ we define $a^n \circ P$ by $a^1 \circ P = a \circ P$ and $a^{n+1} \circ P = a \circ (a^n \circ P)$.

The operational intuition is that each action represents a command which is to be processed by the execution environment of the thread. The processing of a command may involve a change of state of this environment. At completion of the processing of the command, the environment produces a reply value `true` or `false`. The thread $P \unlhd a \unrhd Q$ proceeds as P if the processing of a yields `true`, and it proceeds as Q if the processing of a yields `false`.

Every thread in BTA is finite in the sense that there is a finite upper bound to the number of consecutive actions it can perform. BTA has a completion which comprises also the infinite threads. We interpret instruction sequences, which may be infinite, in this completion.[1]

Mechanistic Threads. To model non-functional aspects of the behavior of instruction sequences (in particular the processing of jumps), we extend BTA with a unary *delay operator* σ taken from relative discrete time process algebra [1, 2]. Write BTA_σ for this extension. Notation: define $\sigma^0(P) = P$ and $\sigma^{n+1}(P) = \sigma(\sigma^n(P))$.[2]

We call the elements of BTA *functional* threads, and the elements of BTA_σ *mechanistic* threads. For example,

$$\sigma(a \circ \mathsf{S})$$

is a mechanistic thread defining the functional behavior $a \circ \mathsf{S}$ preceded by one delay.

Observe that $\mathrm{BTA} \subset \mathrm{BTA}_\sigma$, so any functional thread is also a mechanistic thread. To take this further, we define the functional behavior of a mechanistic thread as the thread that is obtained if we remove all delays. The *functional abstraction operator* $\mathit{fa}(\_)$ does just this:

$$\begin{aligned}
\mathit{fa}(\mathsf{S}) &= \mathsf{S}\\
\mathit{fa}(\mathsf{D}) &= \mathsf{D}\\
\mathit{fa}(P \unlhd a \unrhd Q) &= \mathit{fa}(P) \unlhd a \unrhd \mathit{fa}(Q)\\
\mathit{fa}(\sigma(P)) &= \mathit{fa}(P)
\end{aligned}$$

So, for any mechanistic thread $P \in \mathrm{BTA}_\sigma$, the functional thread $\mathit{fa}(P) \in \mathrm{BTA}$ stands for the functional behavior of P. Two mechanistic threads $P, Q \in \mathrm{BTA}_\sigma$

[1] We omit the mathematical details of this construction because these are not essential for an understanding of the paper. In [3] the completion has been worked out in terms of projective limits, but other constructions are possible as well. The formalization of infinite objects on the basis of finite ones can be done in different ways and the development here is not specific for a particular choice of that formalization.

[2] In relative discrete time process algebra, the delay operator defers the contained behavior to the next time slice. It is assumed that time progresses in slices of equal length, and that the execution of actions does not take time: actions are executed within a time slice. In our sequential setting case such assumptions are not needed. In fact an 'effort' or 'cost' interpretation is just as valid as a 'time' interpretation. But in any interpretation we do assume that the size of one delay is fixed, and that delays can be added: $\sigma^{n+1}(P)$ always puts a strictly larger delay on P than $\sigma^n(P)$ unless $P = \mathsf{D}$.

are *functionally equivalent*, notation $P \sim_f Q$, if they have the same functional behavior: we define

$$P \sim_f Q \quad \text{iff} \quad fa(P) = fa(Q).$$

Mechanistic Improvement. The obvious question is now how to compare distinct mechanistic threads that are functionally equivalent.

For example, we find that

$$\sigma^{452}(P) \sim_f \sigma^3(P),$$
$$\sigma(P) \trianglelefteq a \trianglerighteq Q \sim_f P \trianglelefteq a \trianglerighteq \sigma(Q).$$

In the first case, the right-hand side $\sigma^3(P)$ yields the same functional behavior as the left-hand side using far fewer delays preceding the execution of P. We call $\sigma^3(P)$ a *mechanistic improvement* of $\sigma^{452}(P)$ (an ordering is defined formally below). A mechanistic improvement is viewed as a more efficient way to obtain a (desired) functional behavior. The threads in the second example above cannot be compared in this way: the delays that are visible here in their respective true and false branches occur under different circumstances (execution histories).

The mechanistic improvement ordering $\sqsubseteq_\sigma$ is defined as follows.

$$P \sqsubseteq_\sigma P,$$
$$P \sqsubseteq_\sigma \sigma(P),$$
$$\sigma(\mathsf{D}) \sqsubseteq_\sigma \mathsf{D},$$

and

$$P \sqsubseteq_\sigma P',\ Q \sqsubseteq_\sigma Q' \quad \text{imply} \quad P \trianglelefteq a \trianglerighteq Q \sqsubseteq_\sigma P' \trianglelefteq a \trianglerighteq Q'.$$

If $P \sqsubseteq_\sigma Q$, we say that P is a mechanistic improvement of Q. Further write $P \sqsubset_\sigma Q$ if P is a strict mechanistic improvement of Q (so $P \sqsubseteq_\sigma Q$ and $P \neq Q$).

An obvious observation is that mechanistic improvements yield the same functional behavior:

$$P \sqsubseteq_\sigma Q \quad \text{implies} \quad P \sim_f Q.$$

As seen in the example above, functionally equivalent mechanistic threads need not be comparable by the mechanistic improvement ordering:

$$P \sim_f Q \quad \text{does not imply} \quad (P \sqsubseteq_\sigma Q \text{ or } Q \sqsubseteq_\sigma P).$$

4 Thread Extraction

The *thread extraction operator* $|\_|$ assigns a possibly infinite BTA thread to a PGA instruction sequence. The resulting thread models the functional behavior of the sequence: basic instructions are interpreted as (observable) actions, while the interpretation of jump instructions is made part of the extraction.

Thread extraction is defined by the following equations (where a ranges over basic instructions, u over primitive instructions, and X over non-empty sequences):

$$\begin{aligned}
|X| &= |X;\#0| \qquad \text{if } X \text{ is finite}\\
|!;X| &= \mathsf{S}\\
|a;X| &= a \circ |X|\\
|{+a};u;X| &= |u;X| \trianglelefteq a \trianglerighteq |X|\\
|{-a};u;X| &= |X| \trianglelefteq a \trianglerighteq |u;X|\\
|\#0;X| &= \mathsf{D}\\
|\#1;X| &= |X|\\
|\#k+2;u;X| &= |\#k+1;X|
\end{aligned}$$

Observe that we interpret basic instructions as actions; we use the same symbol to denote both the instruction and its interpretation.

The functional interpretation of instruction sequences defined above abstracts from the (mechanistic) processing of jump instructions. For example, the sequences

$$\#1;\#1;a;!$$

and

$$a;!$$

yield the same functional behavior, namely, the thread $a \circ \mathsf{S}$. Still, execution of the first sequence may require more time and effort because of the processing of the jump instructions. We define an alternative thread extraction operator that takes the mechanistic aspect of behavior into account. We assume that the processing of a jump instruction $\#k$, irrespective of the value of k, results in one *delay* of the subsequent behavior.

The mechanistic thread extraction $|\_|^\sigma$ is defined by the following equations:

$$\begin{aligned}
|X|^\sigma &= |X;\#0|^\sigma \qquad \text{if } X \text{ is finite}\\
|!;X|^\sigma &= \mathsf{S}\\
|a;X|^\sigma &= a \circ |X|^\sigma\\
|{+a};u;X|^\sigma &= |u;X|^\sigma \trianglelefteq a \trianglerighteq |X|^\sigma\\
|{-a};u;X|^\sigma &= |X|^\sigma \trianglelefteq a \trianglerighteq |u;X|^\sigma\\
|\#0;X|^\sigma &= \mathsf{D}\\
|\#1;X|^\sigma &= \sigma(|X|^\sigma)\\
|\#k+2;u;X|^\sigma &= |\#k+1;X|^\sigma
\end{aligned}$$

Fact 1. The functional thread extraction $|X|$ of sequence X equals the functional abstraction of the mechanistic interpretation of X:

$$|X| = fa(|X|^\sigma)$$

for all sequences X.

Example 1. For both $(\#1;a)^\omega$ and $(\#2;\#1;a)^\omega$, mechanistic thread extraction yields the thread P defined recursively by $P = \sigma(a \circ P)$.

The mechanistic interpretation of $(\#1;\#1;a)^\omega$ yields $P = \sigma^2(a \circ P)$, and for $(\#2;a)^\omega$ it yields $P = \sigma(P)$.

5 Implementations

Definition 1 (Mechanistic pre-extraction)**.** Instruction sequence X is a *mechanistic pre-extraction* of thread P, if

$$P = |X|^\sigma.$$

A mechanistic pre-extraction of a thread P is a particular implementation of the behavior P (in fact, it is a particularly efficient implementation, see Fact 3 below), where an implementation is defined as follows.

Definition 2 (Implementation)**.** Instruction sequence X is an *implementation* of thread P, if

$$P \sqsubseteq_\sigma |X|^\sigma.$$

Fact 2. Not all (implementable) threads have a mechanistic pre-extraction. For example, consider

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

It is not difficult to see that P does not have a mechanistic pre-extraction: any implementation will contain at least one jump instruction leading to extractions containing delays not present in P. The sequence

$$X = +a;\#3;c;!;b;!$$

with

$$|X|^\sigma = \sigma(b \circ \mathsf{S}) \trianglelefteq a \trianglerighteq c \circ \mathsf{S}$$

is an implementation of P.

We compare implementations of a thread by their respective mechanistic extractions. That is, if X and Y are implementations of P, then we say that X is a mechanistic improvement of Y if $|X|^\sigma$ is a mechanistic improvement of $|Y|^\sigma$, that is, if $|X|^\sigma \sqsubseteq_\sigma |Y|^\sigma$.

Definition 3 (Optimal implementation)**.** Instruction sequence X is an *optimal implementation* of thread P, if

$$P \sqsubseteq_\sigma |X|^\sigma,$$

and for no other instruction sequence Y that implements P we have $|Y|^\sigma \sqsubset_\sigma |X|^\sigma$.

Definition 4 (Globally optimal implementation)**.** Instruction sequence X is a *globally optimal implementation* of thread P, if

$$P \sqsubseteq_\sigma |X|^\sigma,$$

and for each other instruction sequence Y that implements P we have $|X|^\sigma \sqsubseteq_\sigma |Y|^\sigma$.

Fact 3. If a thread P has a mechanistic pre-extraction X, that is, $|X|^\sigma = P$, then this X is a globally optimal implementation of P.

Example 2. We find that the sequences

$$X = \#1; \#1; a; ! \quad \text{and} \quad Y = \#1; a; !$$

yield the respective mechanistic extractions $\sigma^2(a \circ \mathsf{S})$ and $\sigma(a \circ \mathsf{S})$. Both X and Y are implementations of $P = a \circ \mathsf{S}$, and Y is a mechanistic improvement of X. Observe that the mechanistic pre-extraction $Z = a; !$ of P further improves Y, and that Z is a globally optimal implementation of P.

Example 3. Consider thread P defined by $P = P \trianglelefteq a \trianglerighteq b \circ \mathsf{S}$. Then $X = (+a; \#3; b; !)^\omega$ is an optimal implementation of P which is not globally optimal as it is not a mechanistic improvement of implementation $Y = -a; \#3; X$ of P.

This is worked out as follows. Find that $|X|^\sigma = Q$ defined by

$$Q = \sigma(Q) \trianglelefteq a \trianglerighteq b \circ \mathsf{S},$$

and that $|Y|^\sigma = Q'$, where

$$Q' = Q \trianglelefteq a \trianglerighteq \sigma(b \circ \mathsf{S}).$$

Notice that neither $Q \sqsubseteq_\sigma Q'$ nor $Q' \sqsubseteq_\sigma Q$.

Fact 4. If a thread has implementations it need not have a globally optimal implementation. For example, consider again the thread

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

from the example of Fact 2. Both

$$X = +a; \#3; c; !; b; ! \quad \text{and} \quad Y = -a; \#3; b; !; c; !$$

are implementations of P but they are not comparable:

$$|X|^\sigma = \sigma(b \circ \mathsf{S}) \trianglelefteq a \trianglerighteq c \circ \mathsf{S}$$

and

$$|Y|^\sigma = b \circ \mathsf{S} \trianglelefteq a \trianglerighteq \sigma(c \circ \mathsf{S}),$$

so that neither $|X|^\sigma \sqsubseteq_\sigma |Y|^\sigma$ nor $|Y|^\sigma \sqsubseteq_\sigma |X|^\sigma$. Furthermore, neither sequence can be improved, as seen in the example of Fact 2, so both are optimal implementations.

Fact 5. Every regular (that is, finite-state, see [4]) thread has an implementation.

Proof: For any regular thread P exists a sequence X with $|X| = P$ (see [4]). For this X it holds that $P \sqsubseteq_\sigma |X|^\sigma$.

6 Optimization of Implementations

Optimization of an implementation concerns the following question: given an implementation can we find an improved implementation of the same behavior? We restrict to two observations, where the second one requires a bit more argumentation:

1. Implementations are improved by the unchaining of jumps.
2. Some implementable threads do not have an optimal implementation.

We start with the first observation. If an instruction sequence contains jumps to jump instructions, we speak of *chained* jumps. For example, consider sequence

$$X = \#2; a; \#1; b; !$$

where the first instruction is a jump to the third instruction, which is a jump to the fourth instruction. Unchaining of jumps in this case simply means that we jump to the target location of the latter jump directly. This gives

$$X' = \#3; a; \#1; b; !.$$

Notice that X' is a mechanistic improvement of X. In [3] so-called structural congruence equations are used to capture various cases of jump chaining. Importantly, it is always possible to derive a sequence without chained jumps, and this unchaining leads to the mechanistic improvement of sequences. As a consequence we find this:

> Any sequence X can be improved to a sequence X', i.e., with
>
> $$|X'|^\sigma \sqsubseteq_\sigma |X|^\sigma,$$
>
> such that $|X'|^\sigma$ does not contain multiple consecutive delays, that is, $|X'|^\sigma$ does not have residuals of the form $\sigma^2(Q)$.

Proof idea: a multiple consecutive delay can only result from a jump to a jump instruction, which can always be unchained (leading to larger jumps).

We turn to our second observation. Consider thread P defined by

$$P = P \trianglelefteq a \trianglerighteq Q \quad \text{with} \quad Q = Q \trianglelefteq b \trianglerighteq \mathsf{S}.$$

We demonstrate that each implementation of P can be improved. Stated differently:

Fact 6. Thread P has no optimal implementation.

To begin with we consider P's implementation

$$X = (+a; \#4; +b; \#4; !)^\omega.$$

This X is not an optimal implementation of P because it is improved by

$$Y = (+a; \#6; -b; !; +b; \#4; !)^\omega.$$

To find an improvement of Y one duplicates the repeating part of Y:

$$Y = (+a; \#6; -b; !; +b; \#4; !; +a; \#6; -b; !; +b; \#4; !)^\omega.$$

Now consider

$$Z = (+a; \#8; -b; !; -b; !; -b; !; \#6; !; +a; \#6; -b; !; +b; \#4; !)^\omega$$

and notice that Z improves Y.

Now the proof consists of an extensive case distinction leading to the conclusion that a rewrite similar to the transformations from X to Y and from Y to Z is possible for any implementation X' of P. In particular the following facts can be obtained each with simple arguments most of which we leave to the reader.

1. An implementation X can be assumed to have been written in such a form that, (i) no jump leads to a termination instruction, (ii) no jump leads to another jump (no chained jumps), each instruction is accessible (that is there exists a run which executes that instruction), (iii) each occurrence of a and b is within either a positive or a negative test.

2. If X contains consecutive instructions u and v at least one of these is either a termination instruction or a jump.

3. Every occurrence of b is either in a subsequence $-b; !; u$ with u either $+b$ or $-b$, or in a subsequence $+b; \#k; !$ for some $k > 1$. To see this first notice that after a positive reply on b another b must be performed. Further notice that a subsequence of the form $-b; !; \#k$ can be rewritten as $+b; \#k+1; !$, as there are no chained jumps which make use of the jump in $-b; !; \#k$.

4. There is at least one occurrence of b in a subsequence of the form $+b; \#k; !$. (Otherwise the instruction sequence cannot be written as a finite PGA expression). In addition it can be assumed that this occurrence is contained in the repeating part of X.

5. Using the fact that $(X)^\omega = (X; X)^\omega$ it can be ensured that in this subsequence $+b; \#k; !$ the jump $\#k$ leads to an instruction u containing b and moreover such that u occurs within the repeating part subsequent to the fragment $+b; \#k; !$ that we consider. Moreover it can be ensured that after execution of u with a positive reply the next execution of b is in instruction v which is also included in the repeating part of the expression at a higher position.

6. Assuming $+b; \#k; !$, u and v as in the previous item, two cases are now distinguished: $u = +b$ and $u = -b$. In the first case u is followed by $\#l; !$ for some positive l with $\#l$ leading to v, while in the second case u is followed by $!; v$ or by $!; \#m$ with $\#m$ leading to v.

   In the first case an improvement of X is found by replacing the subsequence $+b; \#k; !$ by $-b; !; +b; \#k'; !$ with jump $\#k'$ leading to v. In this case all jumps that 'fly over' the modified part need to be increased by 2. The second case has two subcases: if u is followed by $!; \#m$ an improvement is found by replacing $+b; \#k; !$ by $-b; !; +b; \#k'; !$ again with jump $\#k'$ leading to v, and while appropriately increasing other jumps in the instruction sequence.

7. We are left with the remaining case that u is followed by $!; v$. Now observe that if we consider subsequent instructions it cannot be an indefinite repetition of $-b; !$ and at some stage a either a positive test followed by a jump $(+b)$ or a jump following termination must occur. We consider one such case only, the other variations being dealt with similarly. Let u be the start of a subsequence $-b; !; -b; !; -b; !; +b; \#n; !$. Then an improvement is found by expanding $+b; \#k; !$ to $-b; !; -b; !; -b; !; -b; !; +b; \#n'; !$, with n' chosen in such a way that it leads to v, and increasing all jumps that 'jump over' the expanded part of the instruction sequence by 6.

7 Concluding Remarks

Mechanistic thread extraction preserves some information concerning the computational mechanisms invoked by an instruction sequence. Our result that the thread $|(+a; \#4; +b; \#4; !)^\omega|$ has no optimal implementation suggests that improvements are possible for each implementation. Such improvements give rise to instruction sequences with increasingly longer repeating parts. This implies a decrease in code compactness which, somehow, will eventually lead to slower computations. Balancing code compactness versus improved implementation cannot be done in the absence of numerical data on implementation technologies and for that reason no attempt is made to do so here.

We have defined and studied mechanistic thread extraction for the simplest of program notations in the program algebra family as presented in [3]. For each new instruction one may provide a mechanistic thread extraction policy. In defining mechanistic behavior of instruction sequences there are several degrees of freedom. For instance one might insist that an absolute jump requires a single unit of time, whereas a relative jump takes two, in view of the fact that performing a relative jump requires some arithmetic involving the program counter.

Such decisions are to some extent arbitrary and it should be expected that in specific cases the most useful definition of mechanistic behavior of a PGA instruction sequence may differ from what we have defined above. For instance, jumps with a counter exceeding some large value, e.g., 100000, may be assigned a larger cost than small jumps. This modification already may invalidate the result that some finite state threads have no optimal implementation (for the particular definition of mechanistic thread extraction as given above).

Many instructions can be analyzed in mechanistic terms, we mention for instance: backward jumps, absolute jumps, goto's, indirect jumps (of various kind), returning jumps, calls to a service, calls to a blocking service, instructions that cause thread creation or thread migration, calls to another instruction sequence and unit instructions.

Mechanistic behavior has a focus on the numbers of steps needed for an immediate interpretation of an instruction sequence. This is by no means the only conceivable cost factor. Modeling energy consumption may be just as important and if basic actions are measured concerning their cost of execution the avoidance of expensive (with respect to time or energy or risk of failure) actions in favor of cheap ones may be more important than the minimization of the number of jumps.

References

[1] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing* 8:188–208, 1996.

[2] J.C.M. Baeten and C.M. Middelburg, *Process algebra with timing*, Springer Monographs in Theoretical Computer Science, 1998

[3] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.

[4] A. Ponse and M.B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al. (editors), *Logical Approaches to Computational Barriers*, *Proceedings CiE 2006*, Springer-Verlag, LNCS 3988:445–458, 2006.