Machine Structure Oriented Control Code Logic

(Extended Version)

J.A. Bergstra
C.A. Middelburg

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

Machine Structure Oriented Control Code Logic⋆

(Extended Version)

J.A. Bergstra[1,2], C.A. Middelburg[1]

[1] Programming Research Group, University of Amsterdam,
   P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
[2] Department of Philosophy, Utrecht University,
   P.O. Box 80126, 3508 TC Utrecht, the Netherlands
   e-mail: `J.A.Bergstra@uva.nl, C.A.Middelburg@uva.nl`

Abstract   Control code is a concept that is closely related to a frequently occurring practitioner's view on what is a program: code that is capable of controlling the behaviour of some machine. We present a logical approach to explain issues concerning control codes that are independent of the details of the behaviours that are controlled. Using this approach, such issues can be explained at a very abstract level. We illustrate this among other things by means of an example about the production of a new compiler from an existing one. The approach is based on abstract machine models, called machine structures. We introduce a model of execution environments for the executable codes of machine structures and use it to go into portability of control codes.

Note   This paper is a substantially extended version of [7].

1 Introduction

In theoretical computer science, the meaning of programs usually plays a prominent part in the explanation of many issues concerning programs. Moreover, what is taken for the meaning of programs is mathematical by nature. On the other hand, it is customary that practitioners do not fall back on the mathematical meaning of programs in case explanation of issues concerning programs is needed. More often than not, they phrase their explanations from the viewpoint that a program is code

that is capable of controlling the behaviour of some machine. Both theorists and practitioners tend to ignore the existence of this contrast. In order to break through this, we as theorists make in this paper an attempt to map out the way in which practitioners explain issues concerning programs.

We informally define control code as code that is capable of controlling the behaviour of some machine. We believe that there are control codes that fail to qualify as programs. For that reason, we make the distinction between control codes and programs. However, there are issues concerning programs that can be explained at the level of control codes by considering them as control codes that qualify as programs. Relative to a fixed machine, the machine-dependent concept of control code that qualifies as program is more abstract than the machine-independent concept of program: control code that qualifies as program is just representative (on the fixed machine) of behaviour associated with a program that is not known. This might be an important motive to explain issues concerning programs at the level of control codes.

To simplify matters, we consider in this paper non-interactive behaviour only. We consider this simplification desirable to start with. Henceforth, control codes are implicitly assumed to control non-interactive behaviour only and the behaviours associated with programs are implicitly assumed to be non-interactive.

Our attempt to map out the way in which practitioners explain issues concerning programs yields a logical approach to explain issues concerning control codes that are independent of the details of the behaviours that are controlled. Machine structures are used as a basis of the approach. They are inspired by the machine functions introduced in [12] to provide a mathematical basis for the T-diagrams proposed in [10]. A machine structure offers a machine model at a very abstract level.

Mapping out the way in which practitioners explain issues concerning programs, being a matter of applied mathematics, turns out to follow a line of itself. This means among other things that steps made in this paper cannot always be motivated directly from the practice that we map out. This is an instance of a general problem of applied mathematics that unfortunately we cannot get round. The general problem is that the design of a mathematical theory does not follow imperatively from the problems of the application area concerned.

We believe that the presented approach is useful because in various areas frequently no distinction is made between programs and control codes and interest is primarily in issues concerning control codes that are independent of the details of the behaviours that are controlled. Some examples of such areas are IT portfolio management, software asset sourcing, and software patents.[1] Moreover, we find that control code production is in the end what software construction is about.

Machine structures in themselves are not always sufficient to explain issues concerning control codes that are independent of the details of the behaviours that are controlled. If systems that provide execution environments for the executable

---

[1] An in-depth treatment of quantitative IT portfolio management can be found in [20]. Software asset sourcing is an important part of IT sourcing, see e.g. [18,11]. An extensive study of software patents and their implications on software engineering practices can be found in [4].

codes of machine structures are involved, then more is needed. We introduce an execution architecture for machine structures as a model of such systems and explain portability of control codes using this execution architecture. An extension of basic thread algebra, introduced in [5] under the name basic polarized process algebra, is used to describe processes that operate upon the execution architecture. The reason to use basic thread algebra is that it has been designed as an algebra of processes that interact with machines of the kind to which also the execution architecture belongs. It is quite awkward to describe processes of that kind using a general process algebra such as ACP [13], CCS [19] or CSP [16].

This paper is organized as follows. First, we introduce machine structures (Section 2). Next, we introduce control code notations and program notations (Section 3). Then, we present our approach to explain issues concerning control codes by means of examples about the production of a new assembler using an existing one and the production of a new compiler using an existing one (Section 4). We also use this approach to explain the relation between compilers and interpreters (Section 5). Following this, we sum up the effects of withdrawing a simplifying assumption concerning the representation of control codes made in the foregoing (Section 6). After that, we outline an execution architecture for machine structures (Section 7). Then, we review the extension of basic thread algebra that covers the effects of applying threads to services (Section 8). Following this, we formalize the execution architecture for machine structures and de£ne the family of services determined by it (Section 9). After that, we explain portability of control codes using thread algebra and the execution architecture services (Section 10). Finally, we make some concluding remarks (Section 11).

Up to Section 7, this paper is a major revision of [2]. It has been substantially rewritten so as to streamline the material. Several important technical aspects have been signi£cantly modi£ed.

2 Machine Functions and Machine Structures

In this section, machine structures are introduced. Machine structures are the basis for our approach to explain issues concerning control codes. They offer models of machines at a very abstract level and cover non-interactive machine behaviour only.

First, we introduce the notion of machine function introduced in [2]. It generalizes the notion of machine function introduced in [12] by covering machines with several outputs. Machine structures can easily be de£ned without reference to machine functions. The introduction of machine functions is mainly for expository reasons.

2.1 Machine Functions

A machine function μ is actually a family of functions: it consists of a function μ_n for each natural number $n > 0$. Those functions map each £nite sequence of bit sequences to either a bit sequence or M or D. Here, M stands for meaningless

and D stands for divergent. A machine function is supposed to model a machine that takes several bit sequences as its inputs and produces several bit sequences as its outputs unless it does not halt on the inputs. Let $x_1, \ldots, x_m$ be bit sequences. Then the connection between the machine function μ and the machine modelled by it can be understood as follows:[2]

 – if $\mu_n(\langle x_1, \ldots, x_m \rangle)$ is a bit sequence, then the machine function μ models a machine that produces $\mu_n(\langle x_1, \ldots, x_m \rangle)$ as its nth output on it taking $x_1, \ldots,$ x_m as its inputs;
 – if $\mu_n(\langle x_1, \ldots, x_m \rangle)$ is M, then the machine function μ models a machine that produces less than n outputs on it taking $x_1, \ldots, x_m$ as its inputs;
 – if $\mu_n(\langle x_1, \ldots, x_m \rangle)$ is D, then the machine function μ models a machine that does not produce any output on it taking $x_1, \ldots, x_m$ as its inputs because it does not halt on the inputs.

Concerning the machine modelled by a machine function, we assume the following:

 – if it does not halt, then no output gets produced;
 – if it does halt, then only £nitely many outputs are produced;
 – if it does not halt, then this cannot be prevented by providing more inputs;
 – if it does halt, then the number of outputs cannot be increased by providing less inputs.

The intuitions behind the £rst two assumptions are obvious. The intuition behind the third assumption is that, with respect to not halting, a machine does not use more inputs than it needs. The intuition behind the last assumption is that, with respect to producing outputs, a machine does not use more inputs than it needs.

Henceforth, we write $\mathcal{BS}$ for the set $\{0,1\}^*$ of *bit sequences*. It is assumed that $M \notin \mathcal{BS}$ and $D \notin \mathcal{BS}$.

We now de£ne machine functions in a mathematically precise way.

Let $BS \subseteq \mathcal{BS}$. Then a *machine function* μ on BS is a family of functions

$$\{\mu_n : BS^* \to (BS \cup \{D, M\}) \mid n \in \mathbb{N}\}$$

satisfying the following rules:

$$\bigwedge_{n \in \mathbb{N}} \left(\bigwedge_{m \in \mathbb{N}} (\mu_n(\chi) = D \Rightarrow \mu_m(\chi) = D) \right),$$
$$\bigwedge_{n \in \mathbb{N}} \left(\mu_n(\chi) \neq D \Rightarrow \left(\bigvee_{m \in \mathbb{N}, m > n} \mu_m(\chi) = M \right) \right),$$
$$\bigwedge_{n \in \mathbb{N}} \left(\bigwedge_{m \in \mathbb{N}, m > n} (\mu_n(\chi) = M \Rightarrow \mu_m(\chi) = M) \right),$$
$$\bigwedge_{n \in \mathbb{N}} (\mu_n(\chi) = D \Rightarrow \mu_n(\chi \frown \chi') = D),$$
$$\bigwedge_{n \in \mathbb{N}} (\mu_n(\chi \frown \chi') = M \Rightarrow \mu_n(\chi) = M).$$

We write $\mathcal{MF}$ for the set of all machine functions.

---

[2] We write $\langle \, \rangle$ for the empty sequence, $\langle x \rangle$ for the sequence having x as sole element, and $\chi \frown \chi'$ for the concatenation of £nite sequences χ and χ'. We use $\langle x_1, \ldots, x_n \rangle$ as a shorthand for $\langle x_1 \rangle \frown \ldots \frown \langle x_n \rangle$. We write X^* for the set of all £nite sequences with elements from set X.

Example 1 Take a high-level programming language PL and an assembly language AL. Consider a machine function cf, which models a machine dedicated to compiling PL programs, and a machine function df, which models a machine dedicated to disassembling executable codes. Suppose that the compiling machine takes a bit sequence representing a PL program as its only input and produces a bit sequence representing an AL version of the PL program as its £rst output, a bit sequence representing a listing of error messages as its second output, and an executable code for the PL program as its third output. Moreover, suppose that the disassembling machine takes an executable code as its only input and producing a bit sequence representing an AL version of the executable code as its £rst output and a bit sequence representing a listing of error messages as its second output. The relevant properties of the machines modelled by cf and df that may now be formulated include:

$$cf_2(\langle x \rangle) = \langle \, \rangle \ \Rightarrow \ cf_1(\langle x \rangle) \neq \langle \, \rangle \ ,$$
$$df_2(\langle x \rangle) = \langle \, \rangle \ \Rightarrow \ df_1(\langle x \rangle) \neq \langle \, \rangle \ ,$$
$$cf_2(\langle x \rangle) = \langle \, \rangle \ \Rightarrow \ df_1(cf_3(\langle x \rangle)) = cf_1(\langle x \rangle) \ .$$

These formulas express that executable code is produced by the compiling machine unless errors are found, disassembly succeeds unless errors are found, and disassembly is the inverse of assembly.

Machines such as the compiling machine and the disassembling machine are special purpose machines. They are restricted to exhibit a particular type of behaviour. Computers are general purpose machines that can exhibit different types of behaviour at different times. This is possible because computers are code controlled machines. A code controlled machine takes one special input that controls its behaviour. In general, not all bit sequences that a code controlled machine can take as its inputs are capable of controlling the behaviour of that machine. The bit sequences that are capable of controlling its behaviour are known as its executable codes. Note that executable code is a machine-dependent concept.

Machine functions can be used to model code controlled machines as well. We will use the phrase code controlled machine function for machine functions that are used to model a code controlled machine. We will use the convention that the £rst bit sequence in the argument of the functions that make up a code controlled machine function corresponds to the special input that controls the behaviour of the machine modelled. Because, in general, not all bit sequences that a code controlled machine can take as its inputs are executable codes, more than just a machine function is needed to model a code controlled machine. That is why we introduce machine structures.

2.2 Machine Structures

A machine structure $\mathfrak{M}$ consists a set of bit sequences BS, functions μ_n that make up a machine function on BS, and a subset E of BS. If E is empty, then the

machine structure $\mathfrak{M}$ is essentially the same as the machine function contained in it. If E is not empty, then the machine structure $\mathfrak{M}$ is supposed to model a code controlled machine. In the case where E is not empty, the connection between the machine structure $\mathfrak{M}$ and the code controlled machine modelled by it can be understood as follows:

– BS is the set of all bit sequences that the code controlled machine modelled by $\mathfrak{M}$ can take as its inputs;
– if $x \in E$, then the bit sequence x belongs to the executable codes of the code controlled machine modelled by $\mathfrak{M}$;
– if $x \in E$, then the functions μ'_n that are defined by $\mu'_n(\langle y_1, \ldots, y_m \rangle) = \mu_n(\langle x, y_1, \ldots, y_m \rangle)$ make up a machine function on BS modeling a machine that exhibits the same behaviour as the code controlled machine modelled by $\mathfrak{M}$ exhibits under control of the executable code x.

The assumptions made about the machine modelled by a machine structure are the same as the assumptions made before about the machine modelled by a machine function. It is tempting to add the following assumption:

– if the special input meant to control its behaviour does not belong to its executable codes, then the machine halts without having produced any output.

We refrain from adding this assumption because it is to be expected that: (a) we can do without it in explaining issues concerning control codes; (b) it does not hold good for all machines that we may encounter. Moreover, in case we would incorporate this assumption in the notion of machine structure, it would not supersede the notion of machine function.

We now define machine structures in a mathematically precise way.

A *machine structure* $\mathfrak{M}$ is a structure composed of

– a set $BS \subseteq \mathcal{BS}$,
– a unary function $\mu_n : BS^* \rightarrow (BS \cup \{\mathsf{D}, \mathsf{M}\})$, for each $n \in \mathbb{N}$,
– a unary relation $E \subseteq BS$,

where the family of functions $\{\mu_n : BS^* \rightarrow (BS \cup \{\mathsf{D}, \mathsf{M}\}) \mid n \in \mathbb{N}\}$ is a machine function on BS. We say that $\mathfrak{M}$ is a *code controlled machine structure* if $E \neq \emptyset$, and we say that $\mathfrak{M}$ is a *dedicated machine structure* if $E = \emptyset$.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let $x \in E$. Then the *meaning* of x with respect to $\mathfrak{M}$, written $|x|^{\mathfrak{M}}$, is the machine function

$$\{\mu'_n : BS^* \rightarrow (BS \cup \{\mathsf{D}, \mathsf{M}\}) \mid n \in \mathbb{N}\} \, ,$$

where the functions μ'_n are defined by

$$\mu'_n(\langle y_1, \ldots, y_m \rangle) = \mu_n(\langle x, y_1, \ldots, y_m \rangle) \, .$$

Moreover, let $x', x'' \in E$. Then x' is *behaviourally equivalent* to x'' on $\mathfrak{M}$, written $x' \equiv^{\mathfrak{M}}_{\mathrm{beh}} x''$, if $|x'|^{\mathfrak{M}} = |x''|^{\mathfrak{M}}$.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then we will write

$$x \bullet\!\bullet_{\mathfrak{M}}^n y_1, \ldots, y_m \quad \text{for} \quad \mu_n(\langle x, y_1, \ldots, y_m \rangle) \ .$$

Moreover, we will write

$$x \bullet\!\bullet_{\mathfrak{M}} y_1, \ldots, y_m \quad \text{for} \quad x \bullet\!\bullet_{\mathfrak{M}}^1 y_1, \ldots, y_m \ .$$

We will also omit $\mathfrak{M}$ if the machine structure is clear from the context.

Example 2 Take a code controlled machine structure $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$. Consider again the machine functions cf and df from Example 1. These machine functions model a machine dedicated to compiling programs in some high-level programming language PL and a machine dedicated to disassembling executable codes, respectively. Let $e_{cf}, e_{df} \in E$ be such that

$$|e_{cf}|^{\mathfrak{M}} = cf \quad \text{and} \quad |e_{df}|^{\mathfrak{M}} = df \ .$$

Then e_{cf} and e_{df} are executable codes that control the behaviour of the code controlled machine modelled by $\mathfrak{M}$ such that this machine behaves the same as the dedicated machine modelled by cf and the dedicated machine modelled by df, respectively. This implies that for all $x \in BS$ and $n \in \mathbb{N}$:

$$e_{cf} \bullet\!\bullet_{\mathfrak{M}}^n x = cf_n(\langle x \rangle) \quad \text{and} \quad e_{df} \bullet\!\bullet_{\mathfrak{M}}^n x = df_n(\langle x \rangle) \ .$$

Note that for cf there may be an $e'_{cf} \in E$ with $e'_{cf} \neq e_{cf}$ such that $|e'_{cf}|^{\mathfrak{M}} = cf$, and likewise for df.

A code controlled machine structure $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ determines all by itself a machine model. For an execution, which takes a single step, an executable code $x \in E$, a sequence $\langle y_1, \ldots, y_m \rangle \in BS^*$ of inputs and the machine function $\{\mu_n \mid n \in \mathbb{N}\}$ are needed. The executable code is not integrated in the machine in any way. In particular, it is not stored in the machine. As nothing is known about any storage mechanism involved, due to the abstract nature of machine structures, it is not plausible to classify the model as a stored code machine model.

2.3 Identifying the Input that Controls Machine Behaviour

It is a matter of convention that the £rst bit sequence in the argument of the functions that make up the machine function of a code controlled machine structure corresponds to the special input that controls the behaviour of the machine modelled. The issue is whether a justi£cation for this correspondence can be found in properties of the code controlled machine structure. This amounts to identifying the input that controls the behaviour of the machine modelled.

Take the simple case where always two inputs are needed to produce any output and always one output is produced. Then a justi£cation for the correspondence

mentioned above can be found only if the machine function involved is asymmetric and moreover the £rst bit sequence in the argument of the function that yields the £rst output overrules the second bit sequence. Here, by overruling is meant being more in control.

In this simple case, the criteria of asymmetry and overruling can easily be made more precise. Suppose that $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ is a code controlled machine structure that models a machine that needs always two inputs to produce any output and produces always one output. Then the machine function $\{\mu_n \mid n \in \mathbb{N}\}$ is asymmetric if there exist $x, y \in BS$ such that $\mu_1(x, y) \neq \mu_1(y, x)$. The £rst bit sequence in the argument of the function μ_1 overrules the second one if there exist $x_1, x_2 \in E$ and $z_1, z_2 \in BS$ with $z_1 \neq z_2$ such that $\mu_1(x_1, y) = z_1$ and $\mu_1(x_2, y) = z_2$ for all $y \in BS$. It is easily proved that the £rst bit sequence in the argument of the function μ_1 overrules the second one only if the second bit sequence in the argument of the function μ_1 does not overrule the £rst one.

The criterion of overruling becomes more interesting if more than two inputs may be needed to produce any output, because this is usually the case with general-purpose machines. For example, on a general-purpose machine, the £rst input may be an executable code for an interpreter of intermediate codes produced by a compiler for some high-level programming language PL, the second input may be a bit sequence representing the intermediate code for a PL program, and one or more subsequent inputs may be bit sequences representing data needed by that program. In this example, the £rst input overrules the second input and subsequent inputs present and in addition the second input overrules the third input and subsequent inputs present.

3 Control Code Notations and Program Notations

In this section, we introduce the concepts of control code notation and program notation and discuss the differences between these concepts.

3.1 Control Code Notations

The intuition is that, for a £xed code controlled machine, control codes are objects (usually texts) representing executable codes of that code controlled machine. The principal examples of control codes are the executable codes themselves. Note that, like the concept of executable code, the concept of control code is machine-dependent. A control code notation for a £xed code controlled machine is a collection of objects together with a function which maps each of the objects from that collection to a particular executable code of the code controlled machine.

In order to make a code controlled machine transform members of one control code notation into members of another control code notation, like in compiling and assembling, control codes that are not bit sequences must be represented by bit sequences. To simplify matters, we will assume that all control code notations are collections of bit sequences. Assuming this amounts to identifying control codes

with the bit sequences representing them. In Section 6, we will withdraw this assumption.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then a *control code notation* for $\mathfrak{M}$ consists of a set $CCN \subseteq BS$ and a function $\psi : CCN \to E$. The members of CCN are called *control codes* for $\mathfrak{M}$. The function ψ is called a *machine structure projection*.

Let (CCN, ψ) be a control code notation for a code controlled machine structure $(BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$. Then we assume that $\psi(c) = c$ for all $c \in CCN \cap E$.

Let $\mathfrak{M}$ be a code controlled machine structure, let (CCN, ψ) be a control code notation for $\mathfrak{M}$, and let $c \in CCN$. Then the *meaning* of c with respect to $\mathfrak{M}$, written $|c|_{CCN}^{\mathfrak{M}}$, is $|\psi(c)|^{\mathfrak{M}}$.

Control codes, like executable codes, are given a meaning related to one code controlled machine structure. The executable codes of a code controlled machine structure themselves make up a control code notation for that machine structure. Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let 1_E be the identity function on E. Then $(E, 1_E)$ is a control code notation for $\mathfrak{M}$. We trivially have $|e|_E^{\mathfrak{M}} = |e|^{\mathfrak{M}}$ for all $e \in E$. Henceforth, we loosely write E for the control code notation $(E, 1_E)$.

3.2 Program Notations

To investigate the conditions under which it is appropriate to say that a control code notation quali£es as a program notation, it is in fact immaterial how the concept of program is de£ned. However, it is at least convenient to make the assumption that, whatever the program notation, there is a hypothetical machine model by means of which the intended behaviour of programs from the program notation can be explained at a level that is suited to our purpose. We believe that this assumption is realistic.

Let some theory of programming be given that offers a reliable de£nition of the concept of program. Then an *acknowledged program notation* is a set PGN of programs. It is assumed that there is a well-understood hypothetical machine model by means of which the intended behaviour of programs from PGN can be explained at a level that allows for the input-output relation of programs from PGN, i.e. the kind of behaviour modelled by machine functions, to be derived. It is also assumed that this hypothetical machine model determines a function $|_-|_{PGN} : PGN \to \mathcal{MF}$ which maps programs to the machine functions modelling their behaviour at the abstraction level of input-output relations.

In [5], a theory, called program algebra, is introduced in which a program is a £nite or in£nite sequence of instructions. Moreover, the intended behaviour of instruction sequences is explained at the level of input-output relations by means of a hypothetical machine model which involves processing of one instruction at a time, where some machine changes its state and produces a reply in case the instruction is not a jump instruction. This hypothetical machine model is an analytic execution architecture in the sense of [9]. In the current paper, the de£nition of the concept of program from [5] could be used. However, we have not £xed a particu-

lar concept of program because we intend to abstract from the details involved in any such conceptual de£nition.

Note that programs, unlike control codes, are given a meaning using a hypothetical machine model. This means that the given meaning is not related to some code controlled machine structure.

3.3 Control Code Notations Qualifying as Program Notations

The intuition is that a control code notation for a code controlled machine quali£es as a program notation if there exist an acknowledged program notation and a function from the control code notation to the program notation that maps each control code to a program such that, at the level of input-output relations, the machine behaviour under control of the control code coincides with the behaviour that is associated with the corresponding program. If a control code notation quali£es as a program notation, then its elements are considered programs.

Let $\mathfrak{M}$ be a code controlled machine structure, and let (CCN, ψ) be a control code notation for $\mathfrak{M}$. Then (CCN, ψ) *quali£es* as a program notation if there exist an acknowledged program notation PGN and a function $\phi : CCN \to PGN$ such that for all $c \in CCN$:

$$|\psi(c)|^{\mathfrak{M}} = |\phi(c)|_{PGN} \ .$$

This de£nition implies that, in the case of a control code notation that quali£es as a program notation, control codes can be given a meaning using a hypothetical machine model. Control code by itself is just representative of machine behaviour without any indication that it originates from a program with which it is possible to explain the behaviour by means of a well-understood hypothetical machine model. The function ϕ whose existence is demanded in the de£nition is suggestive of reverse engineering: by its existence, control codes look to be implementations of programs on a code controlled machine. We might say that the reason for classifying a control code notation in the ones that qualify as a program notation lies in the possibility of reverse engineering. The function ϕ is the opposite of a representation. It might be called a co-representation.

Suppose that $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ is a code controlled machine structure and $(E, 1_E)$ quali£es as a program notation. Then $\mathfrak{M}$ models a code controlled machine whose executable codes constitute a control code notation that quali£es as a program notation. Therefore, it is appropriate to call $\mathfrak{M}$ a program controlled machine structure. A program controlled machine structure is a code controlled machine structure, but there is additional information which is considered to make it more easily understood from the tradition of computer programming: each executable code can be taken for a program and the intended behaviour of that program can be explained by means of a well-understood hypothetical machine model. It is plausible that, for any code controlled machine structure modeling a real machine, there is additional information which is considered to make it more easily understood from some tradition or another.

We take the view that a code controlled machine structure having both executable codes that can be considered programs and executable codes that cannot

be considered programs are improper. Therefore, we introduce the notion of proper code controlled machine structure.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then $\mathfrak{M}$ is a *proper* code controlled machine structure if $(E', 1_{E'})$ quali£es as a program notation for some $E' \subseteq E$ only if $(E, 1_E)$ quali£es as a program notation.

3.4 Control Code Notations Not Qualifying as Program Notations

The question arises whether all control code notations qualify as program notations. If that were true, then the conceptual distinction between control code notations and program notations is small. If a control code notation quali£es as a program notation, then all control codes concerned can be considered the result of implementing a program on a code controlled machine. This indicates that counterexamples to the hypothesis that all control code notations qualify as program notations will concern control codes that do not originate from programming. We give two counterexamples where control codes arise from arti£cial intelligence.

Consider a neural network in hardware form, which is able to learn while working on a problem and thereby de£ning parameter values for many £ring thresholds for arti£cial neurons. The parameter values for a particular problem may serve as input for a machine that needs to address that problem. These problem dependent parameter inputs can be considered control codes by all means. However, there is no conceivable theory of programming according to which these problem dependent parameter inputs can be considered programs. The feature of neural networks that is important here is their ability to acquire control code by another process than programming.

Consider a purely hardware made robot that processes geographical data loaded into it to £nd a target location. The loaded geographical data constitute the only software that determines the behaviour of the robot. Therefore, the loaded geographical data constitute control code. However, there is no conceivable theory of programming according to which such control codes can be considered programs. They are certainly acquired by another process than programming.

In the case of control code notations that qualify as program notations, the control codes are usually produced by programming followed by compiling or assembling. The examples illustrate different forms of control code production that involve neither programming nor compiling or assembling. The £rst example shows that control codes can be produced without programming by means of arti£cial intelligence based techniques. The second example shows that the behaviour of machines applying arti£cial intelligence based techniques can be controlled by control codes that are produced without programming.

4 Assemblers and Compilers

In the production of control code, practitioners often distinguish two kinds of control codes in addition to executable codes: assembly codes and source codes. An

assembler is a control code corresponding to an executable code of a code controlled machine that controls the behaviour of that code controlled machine such that it transforms assembly codes into executable codes and a compiler is a control code corresponding to an executable code of a code controlled machine that controls the behaviour of that code controlled machine such that it that transforms source codes into assembly codes or executable codes.

In this section, we consider the issue of producing a new assembler for some assembly code notation using an existing one and the similar issue of producing a new compiler for some source code notation using an existing one. Whether an assembly code notation or a source code notation quali£es as a program notation is not relevant to these issues.

4.1 Assembly Code Notations and Source Code Notations

At the level of control codes for machine structures, the control code notations that are to be considered assembly code notations and the control code notations that are to be considered source code notations cannot be characterized. The level is too abstract. It happens to be suf£cient for many issues concerning assemblers and compilers, including the ones considered in this section, to simply assume that some collection of control code notations comprises the assembly code notations and some other collection of control code notations comprises the source code notations.

Henceforth, we assume that, for each machine structure $\mathfrak{M}$, disjoint sets $\mathcal{ACN}_\mathfrak{M}$ and $\mathcal{SCN}_\mathfrak{M}$ of control code notations for $\mathfrak{M}$ have been given. The members of $\mathcal{ACN}_\mathfrak{M}$ and $\mathcal{SCN}_\mathfrak{M}$ are called *assembly code notations* for $\mathfrak{M}$ and *source code notations* for $\mathfrak{M}$, respectively.

The following gives an idea of the grounds on which control code notations are classi£ed as assembly code notation or source code notation. Assembly code is control code that is very close to executable code. This means that there is a direct translation of assembly codes into executable codes. An assembly code notation is speci£c to a machine. Source code is control code that is not very close to executable code. The translation of source code into executable code is more involved than the translation of assembly code into executable code. Usually, a source code notation is not speci£c to a machine.

A high-level programming language, such as Java [14] or C# [15], is considered a source code notation. The term high-level programming language suggests that it concerns a notation that quali£es as a program notation. However, as mentioned above, whether a source code notation quali£es as a program notation is not relevant to the issues considered in this section.

4.2 Control Code Notations Involved in Assemblers and Compilers

Three control code notations are involved in an assembler or compiler: it lets a code controlled machine transform members of one control code notation into members of another control code notation and it is itself a member of some control code

notation. We introduce a special notation to describe this aspect of assemblers and compilers succinctly.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let (CCN, ψ), (CCN', ψ') and (CCN'', ψ'') be control code notations for $\mathfrak{M}$. Then we write $cc\,[CCN' \to CCN'']:CCN$ for

$$cc \in CCN \wedge \forall cc' \in CCN' \bullet (\exists cc'' \in CCN'' \bullet \psi(cc) \bullet\bullet_{\mathfrak{M}} cc' = cc'')\,.$$

We say that cc is *in executable form* if $CCN \subseteq E$, that cc is *in assembly form* if $CCN \in \mathcal{ACN}_{\mathfrak{M}}$, and that cc is *in source form* if $CCN \in \mathcal{SCN}_{\mathfrak{M}}$.

4.3 The Assembler Fixed Point

In this subsection, we consider the issue of producing a new assembler for some assembly code notation using an existing one.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let (ACN, ψ) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{ACN}_{\mathfrak{M}}$. Suppose that $ass\,[ACN \to E]:E$ is an existing assembler for ACN. This assembler is in executable form. Suppose further that a new assembler $ass'\,[ACN \to E]:ACN$ for ACN is made available. This new assembler is not in executable form. It needs to be assembled by means of the existing assembler. The new assembler is considered correct if behaviourally equivalent executable codes are produced by the existing assembler and the one obtained by assembling the new assembler by means of the existing assembler, i.e.

$$\forall ac \in ACN \bullet ass \bullet\bullet ac \equiv^{\mathfrak{M}}_{\text{beh}} (ass \bullet\bullet ass') \bullet\bullet ac\,. \tag{1}$$

Let ass'' be the new assembler in executable form obtained by assembling ass' by means of ass, i.e. $ass'' = ass \bullet\bullet ass'$. Now, ass' could be assembled by means of ass'' instead of ass. In case ass'' produces more compact executable codes than ass, this would result in a new assembler in executable form that is more compact. Let ass''' be the new assembler in executable form obtained by assembling ass' by means of ass'', i.e. $ass''' = ass'' \bullet\bullet ass' = (ass \bullet\bullet ass') \bullet\bullet ass'$. If ass' is correct, then ass'' and ass''' produce the same executable codes. That is,

$$ass'' \equiv^{\mathfrak{M}}_{\text{beh}} ass'''\,. \tag{2}$$

This is easy to see: rewriting in terms of ass and ass' yields

$$ass \bullet\bullet ass' \equiv^{\mathfrak{M}}_{\text{beh}} (ass \bullet\bullet ass') \bullet\bullet ass'\,, \tag{3}$$

which follows immediately from (1).

Now, ass' could be assembled by means of ass''' instead of ass''. However, if ass' is correct, this would result in ass''' again. That is,

$$ass''' = ass''' \bullet\bullet ass'\,. \tag{4}$$

This is easy to see as well: rewriting the left-hand side in terms of ass' and ass'' yields

$$ass'' \bullet\bullet ass' = ass''' \bullet\bullet ass' \; , \tag{5}$$

which follows immediately from (2). The phenomenon expresses by equation (4) is called the assembler £xed point.

In theoretical computer science, correctness of a program is taken to mean that the program satis£es a mathematically precise speci£cation of it. For the assembler ass', $\forall ac \in ACN \bullet \psi(ass') \bullet\bullet ac = \psi(ac)$ would be an obvious mathematically precise speci£cation. More often than not, practitioners have a more empirical view on the correctness of a program that is a new program serving as a replacement for an old one on a speci£c machine: correctness of the new program is taken to mean that the old program and the new program give rise to the same behaviour on that machine. The correctness criterion for new assemblers given above, as well as the correctness criterion for new compilers given below, is based on this empirical view.

4.4 The Compiler Fixed Point

In this subsection, we consider the issue of producing a new compiler for some source code notation using an existing one. Compilers may produce assembly code, executable code or both. We deal with the case where compilers produce assembly code only. The reason for this choice will be explained at the end this subsection.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, let (SCN, ψ_s) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{SCN}_{\mathfrak{M}}$, and let (ACN, ψ_a) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{ACN}_{\mathfrak{M}}$. Suppose that $com\,[SCN{\rightarrow}ACN]{:}ACN$ is an existing compiler for SCN and $ass\,[ACN{\rightarrow}E]{:}E$ is an existing assembler for ACN. The existing compiler is in assembly form. However, a compiler in executable form can always be obtained from a compiler in assembly form by means of the existing assembler. Suppose further that a new compiler $com'\,[SCN \rightarrow ACN]:SCN$ for SCN is made available. This new compiler is not in assembly form. It needs to be compiled by means of the existing compiler. The new compiler is considered correct if

$$
\begin{aligned}
&\forall sc \in SCN \bullet \\
&ass \bullet\bullet ((ass \bullet\bullet com) \bullet\bullet sc) \\
&\quad \equiv^{\mathfrak{M}}_{\mathrm{beh}} ass \bullet\bullet ((ass \bullet\bullet ((ass \bullet\bullet com) \bullet\bullet com')) \bullet\bullet sc) \; .
\end{aligned} \tag{6}
$$

Let com'' be the new compiler in assembly form obtained by compiling com' by means of com, i.e. $com'' = (ass \bullet\bullet com) \bullet\bullet com'$. Now, com' could be compiled by means of com'' instead of com. In case com'' produces more compact assembly codes than com, this would result in a new compiler in assembly form that is more compact. Let com''' be the new compiler in assembly form obtained by compiling com' by means of com'', i.e. $com''' = (ass \bullet\bullet com'') \bullet\bullet com' =$

$(ass \bullet\bullet ((ass \bullet\bullet com) \bullet\bullet com')) \bullet\bullet com'$. If com' is correct, then com'' and com''' produce the same assembly codes. That is,

$$ass \bullet\bullet com'' \equiv^{\mathfrak{M}}_{\mathrm{beh}} ass \bullet\bullet com''' . \tag{7}$$

This is easy to see: rewriting in terms of ass, com and com' yields

$$\begin{aligned} &ass \bullet\bullet ((ass \bullet\bullet com) \bullet\bullet com') \\ &\equiv^{\mathfrak{M}}_{\mathrm{beh}} ass \bullet\bullet ((ass \bullet\bullet ((ass \bullet\bullet com) \bullet\bullet com')) \bullet\bullet com') , \end{aligned} \tag{8}$$

which follows immediately from (6).

Now, com' could be compiled by means of com''' instead of com''. However, if com' is correct, this would result in com''' again. That is,

$$com''' = (ass \bullet\bullet com''') \bullet\bullet com' . \tag{9}$$

This is easy to see as well: rewriting the left-hand side in terms of ass, com' and com'' yields

$$(ass \bullet\bullet com'') \bullet\bullet com' = (ass \bullet\bullet com''') \bullet\bullet com' , \tag{10}$$

which follows immediately from (7). The phenomenon expresses by equation (9) is called the compiler £xed point. It is a non-trivial insight among practitioners involved in matters such as software con£guration and system administration.

The explanation of the compiler £xed point proceeds similar to the explanation of the assembler £xed point in Section 4.3, but it is more complicated. The complication vanishes if compilers that produce executable code are considered. In that case, due to the very abstract level at which the issues are considered, the explanation of the compiler £xed point is essentially the same as the explanation of the assembler £xed point.

5 Intermediate Code Notations and Interpreters

Sometimes, practitioners distinguish additional kinds of control codes. Intermediate code is a frequently used generic name for those additional kinds of control codes. Source code is often implemented by producing executable code for some code controlled machine by means of a compiler or a compiler and an assembler. Sometimes, source code is implemented by means of a compiler and an interpreter. In that case, the compiler used produces intermediate code. The interpreter is a control code corresponding to an executable code of a code controlled machine that makes that code controlled machine behave as if it is another code controlled machine controlled by an intermediate code.

In this section, we brie¤y consider the issue of the correctness of such a combination of a compiler and an interpreter.

5.1 Intermediate Code Notations

At the level of control codes for machine structures, like the control code notations that are to be considered assembly code notations and the control code notations that are to be considered source code notations, the control code notations that are to be considered intermediate code notations of some kind cannot be characterized. It happens to be suf£cient for many issues concerning compilers and interpreters, including the one considered in this section, to simply assume that some collection of control code notations comprises the intermediate code notations of interest.

Henceforth, we assume that, for each machine structure $\mathfrak{M}$, a set $\mathcal{ICN}_{\mathfrak{M}}$ of control code notations for $\mathfrak{M}$ has been given. The members of $\mathcal{ICN}_{\mathfrak{M}}$ are called *intermediate code notations* for $\mathfrak{M}$.

The following gives an idea of the grounds on which control code notations are classi£ed as intermediate code notation. An intermediate code notation is a control code notation that resembles an assembly code notation, but it is not speci£c to any machine. Often, it is speci£c to a source code notation or a family of source code notations.

An intermediate code notation comes into play if source code is implemented by means of a compiler and an interpreter. However, compilers for intermediate code notations are found where interpretation is largely eliminated in favour of just-in-time compilation, see e.g. [1], which is material to contemporary programming languages such as Java and C#.

In the case where an intermediate code notation is speci£c to a family of source code notations, it is a common intermediate code notation for the source code notations concerned. The Common Intermediate Language from the .NET Framework [21] is an example of a common intermediate code notation.

5.2 Interpreters

Interpreters are quite different from assemblers and compilers. An assembler for an assembly code notation makes a code controlled machine transform members of the assembly code notation into executable codes and a compiler for a source code notation makes a code controlled machine transform members of the source code notation into members of an assembly code notation or executable codes, whereas an interpreter for an intermediate code notation makes a code controlled machine behave as if it is a code controlled machine for which the members of the intermediate code notation serve as executable codes.

We consider the correctness of an interpreter combined with a compiler going with it. The correctness criterion given below is in the spirit of the empirical view on correctness discussed at the end of Section 4.3.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, let (SCN, ψ_{s}) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{SCN}_{\mathfrak{M}}$, let (ICN, ψ_{i}) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{ICN}_{\mathfrak{M}}$, and let (ACN, ψ_{a}) be a control code notation for $\mathfrak{M}$ that belongs to $\mathcal{ACN}_{\mathfrak{M}}$. Suppose that $com_{\mathrm{a}}\,[SCN \rightarrow ACN] : ACN$ is an existing compiler for SCN and $ass\,[ACN \rightarrow E] : E$ is an

existing assembler for ACN. The compiler com_a lets $\mathfrak{M}$ transform source codes into assembly codes. Suppose further that a new compiler $com_i\,[SCN \to ICN]$: ACN for SCN and a new interpreter $int \in E$ for ICN are made available. The compiler com_i lets $\mathfrak{M}$ transform source codes into intermediate codes.

The combination of com_i and int is considered correct if

$$\forall sc \in SCN, \langle bs_1, \ldots, bs_m \rangle \in BS^* \bullet$$
$$(ass \bullet\bullet_{\mathfrak{M}} ((ass \bullet\bullet_{\mathfrak{M}} com_a) \bullet\bullet_{\mathfrak{M}} sc)) \bullet\bullet_{\mathfrak{M}} bs_1, \ldots, bs_m \qquad (11)$$
$$= int \bullet\bullet_{\mathfrak{M}} ((ass \bullet\bullet_{\mathfrak{M}} com_i) \bullet\bullet_{\mathfrak{M}} sc), bs_1, \ldots, bs_m \ .$$

While being controlled by an interpreter, the behaviour of a code controlled machine can be looked upon as another code controlled machine of which the executable codes are the intermediate codes involved. The latter machine might appropriately be called a virtual machine. By means of interpreters, the same virtual machine can be obtained on different machines. Thus, all machine-dependencies are taken care of by interpreters. A well-known virtual machine is the Java Virtual Machine [17].

6 The Bit Sequence Representation of Control Codes

In order to make a code controlled machine transform members of one control code notation into members of another control code notation, like in assembling and compiling, control codes that are not bit sequences must be represented by bit sequences. To simplify matters, we assumed up to now that all control code notations are collections of bit sequences. In this section, we present the adaptations needed in the preceding sections when withdrawing this assumption. It happens that the changes are small.

The Concept of Control Code Notation

First of all, we have to adapt the concept of control code notation slightly.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then a *control code notation* for $\mathfrak{M}$ consists of a set CCN, a function $\psi : CCN \to E$, and a function $\rho : CCN \to BS$. For all $c \in CCN$, $\rho(c)$ is called the *bit sequence representation* of c on $\mathfrak{M}$. The function ρ is called the *bs-representation function* of CCN.

Let (CCN, ψ, ρ) be a control code notation for a code controlled machine structure $(BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$. Then we assume that $\psi(c) = c$ for all $c \in CCN \cap E$, $\rho(c') = c'$ for all $c' \in CCN \cap BS$, and $\rho(c'') = c''$ for all $c'' \in CCN$ with $\rho(c'') \in E$.

The Special Notation $cc\,[CCN' \to CCN'']:CCN$

We have to change the de£nition of the special notation $cc\,[CCN' \to CCN'']:CCN$ slightly.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let (CCN, ψ, ρ), (CCN', ψ', ρ') and (CCN'', ψ'', ρ'') be control code notations for $\mathfrak{M}$. Then we write $cc\,[CCN' \to CCN''] : CCN$ for

$$cc \in CCN \wedge \forall cc' \in CCN' \bullet (\exists cc'' \in CCN'' \bullet \psi(cc) \bullet\!\bullet_{\mathfrak{M}} \rho'(cc') = \rho''(cc'')) \ .$$

The Explanation of the Assembler Fixed Point

In the explanation of the assembler £xed point given in Section 4.3, we have to replace the de£nitions of ass'' and ass''' by $ass'' = ass \bullet\!\bullet \rho(ass')$ and $ass''' = (ass \bullet\!\bullet \rho(ass')) \bullet\!\bullet \rho(ass')$, assuming that ρ is the bs-representation function of ACN. Moreover, we have to adapt Formulas (1), (3), (4), and (5) slightly. Formula (1) must be replaced by

$$\forall ac \in ACN \bullet ass \bullet\!\bullet \rho(ac) \equiv^{\mathfrak{M}}_{\text{beh}} (ass \bullet\!\bullet \rho(ass')) \bullet\!\bullet \rho(ac) \ .$$

Formula (3) must be replaced by

$$ass \bullet\!\bullet \rho(ass') \equiv^{\mathfrak{M}}_{\text{beh}} (ass \bullet\!\bullet \rho(ass')) \bullet\!\bullet \rho(ass') \ .$$

Formula (4) must be replaced by

$$ass''' = ass''' \bullet\!\bullet \rho(ass') \ .$$

Formula (5) must be replaced by

$$ass'' \bullet\!\bullet \rho(ass') = ass''' \bullet\!\bullet \rho(ass') \ .$$

The Explanation of the Compiler Fixed Point

In the explanation of the compiler £xed point given in Section 4.4, we have to replace the de£nitions of com'' and com''' by $com'' = (ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(com')$ and $com''' = (ass \bullet\!\bullet ((ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(com'))) \bullet\!\bullet \rho_{\text{s}}(com')$, assuming that ρ_{s} is the bs-representation function of SCN and ρ_{a} is the bs-representation function of ACN. Moreover, we have to adapt Formulas (6), (8), (9), and (10) slightly. Formula (6) must be replaced by

$$\forall sc \in SCN \bullet$$
$$ass \bullet\!\bullet ((ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(sc))$$
$$\equiv^{\mathfrak{M}}_{\text{beh}} ass \bullet\!\bullet ((ass \bullet\!\bullet ((ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(com'))) \bullet\!\bullet \rho_{\text{s}}(sc)) \ .$$

Formula (8) must be replaced by

$$ass \bullet\!\bullet ((ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(com'))$$
$$\equiv^{\mathfrak{M}}_{\text{beh}} ass \bullet\!\bullet ((ass \bullet\!\bullet ((ass \bullet\!\bullet \rho_{\text{a}}(com)) \bullet\!\bullet \rho_{\text{s}}(com'))) \bullet\!\bullet \rho_{\text{s}}(com')) \ .$$

Formula (9) must be replaced by

$$com''' = (ass \bullet\bullet com''') \bullet\bullet \rho_s(com') \ .$$

Formula (10) must be replaced by

$$(ass \bullet\bullet com'') \bullet\bullet \rho_s(com') = (ass \bullet\bullet com''') \bullet\bullet \rho_s(com') \ .$$

The Correctness Criterion for Interpreters

The correctness criterion for interpreters given in Section 5.2, i.e. Formula (11), must be replaced by

$$\forall sc \in SCN, \langle bs_1, \ldots, bs_m \rangle \in BS^* \ \bullet$$
$$(ass \bullet\bullet_{\mathfrak{M}} ((ass \bullet\bullet_{\mathfrak{M}} \rho_a(com_a)) \bullet\bullet_{\mathfrak{M}} \rho_s(sc))) \bullet\bullet_{\mathfrak{M}} bs_1, \ldots, bs_m$$
$$= int \bullet\bullet_{\mathfrak{M}} ((ass \bullet\bullet_{\mathfrak{M}} \rho_a(com_i)) \bullet\bullet_{\mathfrak{M}} \rho_s(sc)), bs_1, \ldots, bs_m \ ,$$

assuming that ρ_s is the bs-representation function of SCN, ρ_i is the bs-representation function of ICN, and ρ_a is the bs-representation function of ACN.

7 An Execution Architecture for Machine Structures

In this section, we outline a synthetic execution architecture for code controlled machine structures.[3]

The synthetic execution architecture for code controlled machine structures, which is parametrized by a code controlled machine structure $\mathfrak{M}$, is an abstract model of a system that provides an execution environment for the executable codes of $\mathfrak{M}$. It can be looked upon as a machine. This machine is operated by means of instructions that either yield a reply or diverge. The possible replies are T and F. File names are used in the instructions to refer to the bit sequences present in the machine. It is assumed that a countably infinite set $\mathcal{FN}m$ of *file names* has been given. While designing the instruction set, we focussed on convenience of use rather than minimality.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then the instruction set consists of the following instructions:

- for each $f \in \mathcal{FN}m$ and $bs \in BS$, a *set* instruction set:f:bs;
- for each $f \in \mathcal{FN}m$, a *remove* instruction rm:f;
- for each $f_1, f_2 \in \mathcal{FN}m$, a *copy* instruction cp:f_1:f_2;
- for each $f_1, f_2 \in \mathcal{FN}m$, a *move* instruction mv:f_1:f_2;
- for each $f_1, f_2 \in \mathcal{FN}m$, a *concatenation* instruction cat:f_1:f_2;
- for each $f_1, f_2 \in \mathcal{FN}m$, a *test on equality* instruction tsteq:f_1:f_2;

---

[3] Analytic execution architectures, which are referred to in Section 3.2, and synthetic execution architectures are discussed in [9].

- for each $f_1, f_2 \in \mathcal{FN}m$, a *test on difference* instruction $\mathsf{tstne}{:}f_1{:}f_2$;
- for each $f \in \mathcal{FN}m$, a *test on existence* instruction $\mathsf{tstex}{:}f$;
- for each $f \in \mathcal{FN}m$, a *load* instruction $\mathsf{load}{:}f$;
- for each $f_1, \ldots, f_m \in \mathcal{FN}m$ and $f'_1, \ldots, f'_n \in \mathcal{FN}m$, an *execute* instruction $\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f'_1{:}\ldots{:}f'_n$.

We write I for this instruction set.

We say that a file name is in use if it has a bit sequence assigned. A state of the machine comprises the file names that are in use, the bit sequences assigned to those file names, a flag indicating whether there is a loaded executable code, and the loaded executable code if there is one.

The instructions can be explained in terms of the effect that they have and the reply that they yield as follows:

- $\mathsf{set}{:}f{:}bs$: the file name f is added to the file names in use if it is not in use, the bit sequence bs is assigned to f, and the reply is T;
- $\mathsf{rm}{:}f$: if the file name f is in use, then it is removed from the file names in use and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{cp}{:}f_1{:}f_2$: if the file name f_1 is in use, then the file name f_2 is added to the file names in use if it is not in use, the bit sequence assigned to f_1 is assigned to f_2, and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{mv}{:}f_1{:}f_2$: if the file name f_1 is in use, then the file name f_2 is added to the file names in use if it is not in use, the bit sequence assigned to f_1 is assigned to f_2, f_1 is removed from the file names in use, and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{cat}{:}f_1{:}f_2$: if the file names f_1 and f_2 are in use, then the concatenation of the bit sequence assigned to f_2 and the bit sequence assigned to f_1 is assigned to f_2 and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{tsteq}{:}f_1{:}f_2$: if the file names f_1 and f_2 are in use and the bit sequence assigned to f_1 equals the bit sequence assigned to f_2, then nothing changes and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{tstne}{:}f_1{:}f_2$: if the file names f_1 and f_2 are in use and the bit sequence assigned to f_1 does not equal the bit sequence assigned to f_2, then nothing changes and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{tstex}{:}f$: if the file name f is in use, then nothing changes and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{load}{:}f$: if the file name f is in use and the bit sequence assigned to f is a member of E, then the bit sequence assigned to f is loaded and the reply is T; otherwise, nothing changes and the reply is F;
- $\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f'_1{:}\ldots{:}f'_n$: if the file names $f_1, \ldots, f_m$ have bit sequences assigned, say $bs_1, \ldots, bs_m$, and there is a loaded executable code, say x, then:
  - if $x \bullet\bullet^1_{\mathfrak{M}} bs_1, \ldots, bs_m \in BS$, then:
    - $x \bullet\bullet^i_{\mathfrak{M}} bs_1, \ldots, bs_m$ is assigned to f'_i for each i with $1 \leq i \leq n$ such that $x \bullet\bullet^i_{\mathfrak{M}} bs_1, \ldots, bs_m \in BS$,
    - f'_i is removed from the file names in use for each i with $1 \leq i \leq n$ such that $x \bullet\bullet^i_{\mathfrak{M}} bs_1, \ldots, bs_m = \mathsf{M}$,

  and the reply is T;

- if $x \bullet\bullet^1_{\mathfrak{M}} bs_1, \ldots, bs_m = \mathsf{M}$, then nothing changes and the reply is F;
- if $x \bullet\bullet^1_{\mathfrak{M}} bs_1, \ldots, bs_m = \mathsf{D}$, then the machine does not halt;

otherwise, nothing changes and the reply is F.

Note that there are three cases in which the instruction $\mathrm{exec}{:}f_1{:}\ldots{:}f_m{>}f'_1{:}\ldots{:}f'_n$ yields the reply F: (a) there is no loaded executable code; (b) there is some £le name among $f_1, \ldots, f_m$ that is not in use; (c) there is no output produced, although the machine halts.

The instructions of which the effect depends on the code controlled machine structure $\mathfrak{M}$ are the load and execute instructions only. All other instructions could be eliminated in favour of executable codes, assigned to known £le names. However, we believe that elimination of these instructions would not contribute to a useful execution architecture. The distinction made between loading and execution of executable codes allows for telling load-time errors from run-time errors.

8 Thread Algebra

In Section 10, BTA (Basic Thread Algebra) extended with thread-to-service application is used to describe processes that operate upon the execution architecture outlined above. BTA is introduced in [5] under the name BPPA (Basic Polarized Process Algebra). In this section, we review BTA, including guarded recursion and the approximation induction principle, and extend it with thread-to-service application.

8.1 Basic Thread Algebra

BTA is a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there is a £xed but arbitrary set of *basic actions* $\mathcal{A}$. The intuition is that each basic action performed by a thread is taken as a command to be processed by a service provided by the execution environment of the thread. The processing of a command may involve a change of state of the service concerned. At completion of the processing of the command, the service produces a reply value. This reply is either T or F and is returned to the thread concerned.

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with additional sorts in Section 8.2.

The algebraic theory BTA has one sort: the sort $\mathbf{T}$ of *threads*. BTA has the following constants and operators:

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A}$, the binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Table 1 Axioms for guarded recursion

$\langle X|E\rangle = \langle t_X|E\rangle$ if $X = t_X \in E$ RDP	
$E \Rightarrow X = \langle X|E\rangle$ if $X \in \mathrm{V}(E)$ RSP	

Terms of sort $\mathbf{T}$ are built as usual. Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where p is a term of sort $\mathbf{T}$, abbreviates $p \trianglelefteq a \trianglerighteq p$.

Let p and q be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A}$. Then $p \trianglelefteq a \trianglerighteq q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply) and proceed as q if the processing of a leads to the reply F (called a negative reply).

Each closed term of sort $\mathbf{T}$ from the language of BTA denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Guarded recursive specifications give rise to infinite threads.

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort $\mathbf{T}$ and each t_X is a term of sort $\mathbf{T}$ that has the form D, S or $t \trianglelefteq a \trianglerighteq t'$. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [3].

We extend BTA with guarded recursion by adding constants for solutions of guarded recursive specifications and axioms concerning these additional constants. For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we add a constant of sort $\mathbf{T}$ standing for the unique solution of E for X to the constants of BTA. The constant standing for the unique solution of E for X is denoted by $\langle X|E\rangle$. Moreover, we add the axioms for guarded recursion given in Table 1 to BTA, where we write $\langle t_X|E\rangle$ for t_X with, for all $Y \in \mathrm{V}(E)$, all occurrences of Y in t_X replaced by $\langle Y|E\rangle$. In this table, X, t_X and E stand for an arbitrary variable of sort $\mathbf{T}$, an arbitrary term of sort $\mathbf{T}$ from the language of BTA, and an arbitrary guarded recursive specification over BTA, respectively. Side conditions are added to restrict the variables, terms and guarded recursive specifications for which X, t_X and E stand. The equations $\langle X|E\rangle = \langle t_X|E\rangle$ for a fixed E express that the constants $\langle X|E\rangle$ make up a solution of E. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one.

We will write BTA+REC for BTA extended with the constants for solutions of guarded recursive specifications and axioms RDP and RSP.

In [6], we show that the processes considered in BTA+REC can be viewed as processes that are definable over ACP [13].

Closed terms of sort $\mathbf{T}$ from the language of BTA+REC that denote the same infinite thread cannot always be proved equal by means of the axioms of BTA+REC. We introduce the approximation induction principle to remedy this. The approximation induction principle, AIP in short, is based on the view that two threads are identical if their approximations up to any finite depth are identical.

Table 2 Approximation induction principle

$$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \;\Rightarrow\; x = y \quad \text{AIP}$$

Table 3 Axioms for projection operators

$\pi_0(x) = \mathsf{D}$	P0
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

The approximation up to depth n of a thread is obtained by cutting it off after performing a sequence of actions of length n.

AIP is the infinitary conditional equation given in Table 2. Here, following [5], approximation of depth n is phrased in terms of a unary *projection* operator π_n. The axioms for the projection operators are given in Table 3. In this table, a stands for an arbitrary member of $\mathcal{A}$.

8.2 Applying Threads to Services

We extend BTA+REC to a theory that covers the effects of applying threads to services.

It is assumed that there is a fixed but arbitrary set of *foci* $\mathcal{F}$ and a fixed but arbitrary set of *methods* $\mathcal{M}$. For the set of basic actions $\mathcal{A}$, we take the set $FM = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Each focus plays the role of a name of a service provided by the execution environment that can be requested to process a command. Each method plays the role of a command proper. Performing a basic action $f.m$ is taken as making a request to the service named f to process the command m.

We introduce a second sort: the sort $\mathbf{S}$ of *services*. However, we will not introduce constants and operators to build terms of this sort. $\mathbf{S}$ is a parameter of theories with thread-to-service application. $\mathbf{S}$ is considered to stand for the set of all services. It is assumed that each service can be represented by a function $H : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{D}\}$ with the property that $H(\gamma) = \mathsf{D} \;\Rightarrow\; H(\gamma \frown \langle m \rangle) = \mathsf{D}$ for all $\gamma \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This function is called the *reply* function of the service. Given a reply function H and a method $m \in \mathcal{M}$, the *derived* reply function of H after processing m, written $\frac{\partial}{\partial m} H$, is defined by $\frac{\partial}{\partial m} H(\gamma) = H(\langle m \rangle \frown \gamma)$.

The connection between a reply function H and the service represented by it can be understood as follows:

- if $H(\langle m \rangle) = \mathsf{T}$, the request to process command m is accepted by the service, the reply is positive and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{F}$, the request to process command m is accepted by the service, the reply is negative and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{D}$, either the processing of command m by the service does not halt or the processing of a previous command by the service did not halt.

Table 4 Axioms for apply

$x \bullet_f \uparrow = \uparrow$		TSA0
$\mathsf{S} \bullet_f H = H$		TSA1
$\mathsf{D} \bullet_f H = \uparrow$		TSA2
$(x \unlhd g.m \unrhd y) \bullet_f H = \uparrow$	if $f \neq g$	TSA3
$(x \unlhd f.m \unrhd y) \bullet_f H = x \bullet_f \frac{\partial}{\partial m} H$	if $H(\langle m \rangle) = \mathsf{T}$	TSA4
$(x \unlhd f.m \unrhd y) \bullet_f H = y \bullet_f \frac{\partial}{\partial m} H$	if $H(\langle m \rangle) = \mathsf{F}$	TSA5
$(x \unlhd f.m \unrhd y) \bullet_f H = \uparrow$	if $H(\langle m \rangle) = \mathsf{D}$	TSA6
$(\bigwedge_{n \geq 0} \pi_n(x) \bullet_f H = \uparrow) \Rightarrow x \bullet_f H = \uparrow$		TSA7

Henceforth, we will identify a reply function with the service represented by it.

It is assumed that there is an *undefined service* $\uparrow$ with the property that $\uparrow(\gamma) = \mathsf{D}$ for all $\gamma \in \mathcal{M}^+$.

For each $f \in \mathcal{F}$, we introduce the binary *apply* operator $\_ \bullet_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. Intuitively, $p \bullet_f H$ is the service that evolves from H on processing all basic actions performed by thread p that are of the form $f.m$ by H. When a basic action $f.m$ performed by thread p is processed by H, p proceeds on the basis of the reply value produced.

The axioms for the apply operators are given in Table 4. In this table, f and g stand for arbitrary foci from $\mathcal{F}$ and m stands for an arbitrary method from $\mathcal{M}$. The axioms show that $p \bullet_f H$ does not equal $\uparrow$ only if thread p performs no other basic actions than ones of the form $f.m$ and eventually terminates successfully.

Let p be a closed term of sort $\mathbf{T}$ from the language of BTA+REC and H be a closed term of sort $\mathbf{S}$. Then p *converges* from H on f if there exists an $n \in \mathbb{N}$ such that $\pi_n(p) \bullet_f H \neq \uparrow$. Notice that axiom TSA7 can be read as follows: if x does not converge from H on f, then $x \bullet_f H$ equals $\uparrow$.

The extension of BTA introduced above originates from [8]. In the remainder of this paper, we will use just one focus. We have introduced the general case here because the use of several foci might be needed on further elaboration of the work presented in this paper.

9 The Execution Architecture Services

In order to be able to use the extension of BTA presented above to describe processes that operate upon the execution architecture for code controlled machine structures outlined in Section 7, we have to associate a service with each state of the execution architecture. In this section, we first formalize the execution architecture for code controlled machine structures and then associate a service with each of its states.

9.1 The Execution Architecture Formalized

The execution architecture for code controlled machine structures consists of an instruction set, a state set, an effect function, and a yield function. The effect and yield functions give, for each instruction u and state s, the state and reply, respectively, that result from processing u in state s.

It is assumed that $\mathsf{s_D} \notin \left(\bigcup_{F \in \mathcal{P}_{\mathrm{fin}}(\mathcal{FN}m)} (F \to \mathcal{BS}) \right) \times (\mathcal{BS} \cup \{\mathsf{M}\})$. Here, $\mathsf{s_D}$ stands for a state of divergence.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then the *execution architecture* for $\mathfrak{M}$ consists of

– the instruction set I de£ned in Section 7;
– the state set S de£ned by

$$S = \left(\left(\bigcup_{F \in \mathcal{P}_{\mathrm{fin}}(\mathcal{FN}m)} (F \to BS) \right) \times (E \cup \{\mathsf{M}\}) \right) \cup \{\mathsf{s_D}\} \; ;$$

– the effect function $\mathit{eff} : I \times S \to S$ de£ned in Table 5;
– the yield function $\mathit{yld} : I \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{D}\}$ de£ned in Table 6.

We use the following notation for functions: $[\,]$ for the empty function; $[d \mapsto r]$ for the function f with $\mathrm{dom}(f) = \{d\}$ such that $f(d) = r$; $f \oplus g$ for the function h with $\mathrm{dom}(h) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$ such that for all $d \in \mathrm{dom}(h)$, $h(d) = f(d)$ if $d \notin \mathrm{dom}(g)$ and $h(d) = g(d)$ otherwise; and $f \triangleleft D$ for the function g with $\mathrm{dom}(g) = \mathrm{dom}(f) \setminus D$ such that for all $d \in \mathrm{dom}(g)$, $g(d) = f(d)$.

Let $(\sigma, x) \in S$, and let $f \in \mathcal{FN}m$. Then f is in use if $f \in \mathrm{dom}(\sigma)$, and there is a loaded executable code if $x \neq \mathsf{M}$. If f is in use, then $\sigma(f)$ is the bit sequence assigned to f. If there is a loaded executable code, then x is the loaded executable code.

Execute instructions can diverge. When an instruction diverges, a situation arises in which no reply can be produced and no further instructions can be processed. This is modelled by eff producing $\mathsf{s_D}$ and yld producing D.

9.2 The Family of Execution Architecture Services

Each state of the execution architecture for code controlled machine structures can be looked upon as a service by assuming that $I \subseteq \mathcal{M}$ and extending the functions eff and yld from I to $\mathcal{M}$ by stipulating that $\mathit{eff}(m, s) = \mathsf{s_D}$ and $\mathit{yld}(m, s) = \mathsf{D}$ for all $m \in \mathcal{M} \setminus I$ and $s \in S$.

We de£ne, for each $s \in S$, a cumulative effect function $\mathit{ceff}_s : \mathcal{M}^* \to S$ in terms of s and eff as follows:

$$\mathit{ceff}_s(\langle \rangle) = s$$
$$\mathit{ceff}_s(\gamma \frown \langle m \rangle) = \mathit{eff}(m, \mathit{ceff}_s(\gamma)) \; .$$

Table 5 Effect function for an execution architecture ($u \in I$)

$eff(\mathsf{set}{:}f{:}bs, (\sigma, x)) = (\sigma \oplus [f \mapsto bs], x)$

$eff(\mathsf{rm}{:}f, (\sigma, x)) = (\sigma \vartriangleleft \{f\}, x)$

$eff(\mathsf{cp}{:}f_1{:}f_2, (\sigma, x)) = (\sigma \oplus [f_2 \mapsto \sigma(f_1)], x)$ if $f_1 \in \mathrm{dom}(\sigma)$

$eff(\mathsf{cp}{:}f_1{:}f_2, (\sigma, x)) = (\sigma, x)$ if $f_1 \notin \mathrm{dom}(\sigma)$

$eff(\mathsf{mv}{:}f_1{:}f_2, (\sigma, x)) = ((\sigma \oplus [f_2 \mapsto \sigma(f_1)]) \vartriangleleft \{f_1\}, x)$ if $f_1 \in \mathrm{dom}(\sigma)$

$eff(\mathsf{mv}{:}f_1{:}f_2, (\sigma, x)) = (\sigma, x)$ if $f_1 \notin \mathrm{dom}(\sigma)$

$eff(\mathsf{cat}{:}f_1{:}f_2, (\sigma, x)) = (\sigma \oplus [f_2 \mapsto \sigma(f_2) \frown \sigma(f_1)], x)$ if $f_1 \in \mathrm{dom}(\sigma) \wedge f_2 \in \mathrm{dom}(\sigma)$

$eff(\mathsf{cat}{:}f_1{:}f_2, (\sigma, x)) = (\sigma, x)$ if $f_1 \notin \mathrm{dom}(\sigma) \vee f_2 \notin \mathrm{dom}(\sigma)$

$eff(\mathsf{tsteq}{:}f_1{:}f_2, (\sigma, x)) = (\sigma, x)$

$eff(\mathsf{tstne}{:}f_1{:}f_2, (\sigma, x)) = (\sigma, x)$

$eff(\mathsf{tstex}{:}f, (\sigma, x)) = (\sigma, x)$

$eff(\mathsf{load}{:}f, (\sigma, x)) = (\sigma, \sigma(f))$ if $f \in \mathrm{dom}(\sigma) \wedge \sigma(f) \in E$

$eff(\mathsf{load}{:}f, (\sigma, x)) = (\sigma, x)$ if $f \notin \mathrm{dom}(\sigma) \vee \sigma(f) \notin E$

$eff(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = ((\ldots(\sigma \oplus \sigma_1')\ldots \oplus \sigma_n'), x)$

     where $\sigma_i' = [f_i' \mapsto x \bullet\!\bullet_{\mathfrak{M}}^{i} \sigma(f_1), \ldots, \sigma(f_m)]$ if $x \bullet\!\bullet_{\mathfrak{M}}^{i} \sigma(f_1), \ldots, \sigma(f_m) \in BS$

            $\sigma_i' = [\,]$ if $x \bullet\!\bullet_{\mathfrak{M}}^{i} \sigma(f_1), \ldots, \sigma(f_m) = \mathsf{M}$

    if $x \in E \wedge f_1 \in \mathrm{dom}(\sigma) \wedge \ldots \wedge f_m \in \mathrm{dom}(\sigma) \wedge x \bullet\!\bullet_{\mathfrak{M}}^{1} \sigma(f_1), \ldots, \sigma(f_m) \in BS$

$eff(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = (\sigma, x)$

    if $x \notin E \vee f_1 \notin \mathrm{dom}(\sigma) \vee \ldots \vee f_m \notin \mathrm{dom}(\sigma) \vee x \bullet\!\bullet_{\mathfrak{M}}^{1} \sigma(f_1), \ldots, \sigma(f_m) = \mathsf{M}$

$eff(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = \mathsf{s_D}$

    if $x \notin E \vee f_1 \notin \mathrm{dom}(\sigma) \vee \ldots \vee f_m \notin \mathrm{dom}(\sigma) \vee x \bullet\!\bullet_{\mathfrak{M}}^{1} \sigma(f_1), \ldots, \sigma(f_m) = \mathsf{D}$

$eff(u, \mathsf{s_D}) = \mathsf{s_D}$

We define, for each $s \in S$, an *execution architecture service* $H_s{:}\mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{D}\}$ in terms of $ceff_s$ and yld as follows:

$$H_s(\gamma \frown \langle m \rangle) = yld(m, ceff_s(\gamma)) \ .$$

For each $s \in S$, H_s is a service indeed: $H_s(\gamma) = \mathsf{D} \ \Rightarrow \ H_s(\gamma \frown \langle m \rangle) = \mathsf{D}$ for all $\gamma \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This follows from the following property of the execution architecture for code controlled machine structures:

$$\exists s \in S \bullet \forall i \in I \bullet$$
$$(yld(i, s) = \mathsf{D} \wedge \forall s' \in S \bullet (yld(i, s') = \mathsf{D} \ \Rightarrow \ eff(i, s') = s)) \ .$$

The witnessing state of this property is $\mathsf{s_D}$. This state is connected with the undefined service $\uparrow$ as follows: $H_{\mathsf{s_D}} = \uparrow$.

It is worth mentioning that $H_s(\langle m \rangle) = yld(m, s)$ and $\frac{\partial}{\partial m} H_s = H_{eff(m,s)}$.

We write $\mathcal{EAS}^{\mathfrak{M}}$ for the family of services $\{H_s \mid s \in S\}$.

Table 6 Yield function for an execution architecture ($u \in I$)

$yld(\mathsf{set}{:}f{:}bs, (\sigma, x)) = \mathsf{T}$

$yld(\mathsf{rm}{:}f, (\sigma, x)) = \mathsf{T}$ if $f \in \mathrm{dom}(\sigma)$

$yld(\mathsf{rm}{:}f, (\sigma, x)) = \mathsf{F}$ if $f \notin \mathrm{dom}(\sigma)$

$yld(\mathsf{cp}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{T}$ if $f_1 \in \mathrm{dom}(\sigma)$

$yld(\mathsf{cp}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{F}$ if $f_1 \notin \mathrm{dom}(\sigma)$

$yld(\mathsf{mv}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{T}$ if $f_1 \in \mathrm{dom}(\sigma)$

$yld(\mathsf{mv}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{F}$ if $f_1 \notin \mathrm{dom}(\sigma)$

$yld(\mathsf{cat}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{T}$ if $f_1 \in \mathrm{dom}(\sigma) \wedge f_2 \in \mathrm{dom}(\sigma)$

$yld(\mathsf{cat}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{F}$ if $f_1 \notin \mathrm{dom}(\sigma) \vee f_2 \notin \mathrm{dom}(\sigma)$

$yld(\mathsf{tsteq}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{T}$ if $f_1 \in \mathrm{dom}(\sigma) \wedge f_2 \in \mathrm{dom}(\sigma) \wedge \sigma(f_1) = \sigma(f_2)$

$yld(\mathsf{tsteq}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{F}$ if $f_1 \notin \mathrm{dom}(\sigma) \vee f_2 \notin \mathrm{dom}(\sigma) \vee \sigma(f_1) \neq \sigma(f_2)$

$yld(\mathsf{tstne}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{T}$ if $f_1 \in \mathrm{dom}(\sigma) \wedge f_2 \in \mathrm{dom}(\sigma) \wedge \sigma(f_1) \neq \sigma(f_2)$

$yld(\mathsf{tstne}{:}f_1{:}f_2, (\sigma, x)) = \mathsf{F}$ if $f_1 \notin \mathrm{dom}(\sigma) \vee f_2 \notin \mathrm{dom}(\sigma) \vee \sigma(f_1) = \sigma(f_2)$

$yld(\mathsf{tstex}{:}f, (\sigma, x)) = \mathsf{T}$ if $f \in \mathrm{dom}(\sigma)$

$yld(\mathsf{tstex}{:}f, (\sigma, x)) = \mathsf{F}$ if $f \notin \mathrm{dom}(\sigma)$

$yld(\mathsf{load}{:}f, (\sigma, x)) = \mathsf{T}$ if $f \in \mathrm{dom}(\sigma) \wedge \sigma(f) \in E$

$yld(\mathsf{load}{:}f, (\sigma, x)) = \mathsf{F}$ if $f \notin \mathrm{dom}(\sigma) \vee \sigma(f) \notin E$

$yld(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = \mathsf{T}$

     if $x \in E \wedge f_1 \in \mathrm{dom}(\sigma) \wedge \ldots \wedge f_m \in \mathrm{dom}(\sigma) \wedge x \bullet\!\bullet^1_{\mathfrak{M}} \sigma(f_1), \ldots, \sigma(f_m) \in BS$

$yld(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = \mathsf{F}$

     if $x \notin E \vee f_1 \notin \mathrm{dom}(\sigma) \vee \ldots \vee f_m \notin \mathrm{dom}(\sigma) \vee x \bullet\!\bullet^1_{\mathfrak{M}} \sigma(f_1), \ldots, \sigma(f_m) = \mathsf{M}$

$yld(\mathsf{exec}{:}f_1{:}\ldots{:}f_m{>}f_1'{:}\ldots{:}f_n', (\sigma, x)) = \mathsf{D}$

     if $x \notin E \vee f_1 \notin \mathrm{dom}(\sigma) \vee \ldots \vee f_m \notin \mathrm{dom}(\sigma) \vee x \bullet\!\bullet^1_{\mathfrak{M}} \sigma(f_1), \ldots, \sigma(f_m) = \mathsf{D}$

$yld(u, \mathsf{s_D}) = \mathsf{D}$

10 Control Codes and Execution Architecture Services

In this section, we make precise what it means that a control code is installed on an execution architecture service and what it means that a control code is portable from one execution architecture service to another execution architecture service.

10.1 Installed Control Codes

The intuition is that a control code is installed on an execution architecture service if some £le name has assigned an executable version of the control code or some £le name has assigned an interpretable version of the control code and an appropriate interpreter is installed on the execution architecture service as well.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, let (CCN, ψ) be a control code notation for $\mathfrak{M}$, let $c \in CCN$, and let $EAS = H_{(\sigma, x)} \in \mathcal{EAS}^{\mathfrak{M}}$. Then c is *installed* on EAS if there exist $f_0, \ldots, f_l \in \mathcal{FN}m$ with $\sigma(f_0) \in E$ such that

$$\forall \langle bs_1, \ldots, bs_m \rangle \in BS^* \bullet$$

$$\bigwedge_{n \in \mathbb{N}} \psi(c) \bullet\bullet^n_{\mathfrak{M}} bs_1, \ldots, bs_m = \sigma(f_0) \bullet\bullet^n_{\mathfrak{M}} \sigma(f_1), \ldots, \sigma(f_l), bs_1, \ldots, bs_m \ .$$

A control code is pre-installed on an execution architecture service if the execution architecture service can be expanded to one on which it is installed, using only control codes and data already assigned to £le names. Thread algebra is brought into play to make precise what it means that an execution architecture service can be expanded to another execution architecture service.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, let $EAS = H_{(\sigma, x)} \in \mathcal{EAS}^{\mathfrak{M}}$, and let $EAS' = H_{(\sigma', x')} \in \mathcal{EAS}^{\mathfrak{M}}$. Then EAS is *expansible* to EAS' if:

– $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$ and $\sigma(f) = \sigma'(f)$ for all $f \in \mathrm{dom}(\sigma)$;
– there exists a thread p without basic actions of the form ea.set:f:bs such that $EAS' = p \bullet_{\mathsf{ea}} EAS$.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, let (CCN, ψ) be a control code notation for $\mathfrak{M}$, let $c \in CCN$, and let $EAS \in \mathcal{EAS}^{\mathfrak{M}}$. Then c is *pre-installed* on EAS if

– c is not *installed* on EAS;
– there exists a $EAS' \in \mathcal{EAS}^{\mathfrak{M}}$ such that EAS is expansible to EAS' and c is *installed* on EAS'.

Example 3 Take an assembly code notation ACN and a source code notation SCN. Consider an execution architecture service EAS on which £le name f_1 has assigned an executable version of an assembler for ACN, £le name f_2 has assigned an ACN version of a compiler for SCN, and £le name f_3 has nothing assigned. Suppose that no £le name has assigned an executable version of the compiler. Then the compiler is not installed on EAS. However, the compiler is pre-installed on EAS because it is installed on the expanded execution architecture service $(\mathsf{ea.load}{:}f_1 \circ \mathsf{ea.exec}{:}f_2{>}f_3) \bullet_{\mathsf{ea}} EAS$.

10.2 Portable Control Codes

We take portability of control code to mean portability from a service de£ned by the execution architecture for one machine structure to a service de£ned by the execution architecture for another machine structure.

Transportability is considered a property of all bit sequences, i.e. each bit sequence can be transported between any two services de£ned by execution architectures for machine structures. Therefore, it is assumed that every bit sequence assigned to a £le name on a service can be assigned to a £le name on another service by means of an instruction of the form set:f:bs.

A prerequisite for portability of a control code from a service de£ned by the execution architecture for one machine structure to a service de£ned by the execution architecture for another machine structure is that, for all inputs covered by the former machine structure, the outputs produced under control of the control code coincide for the two machine structures concerned. Moreover, it must be possible to expand the service from which the control code originates such that the control code is pre-installed on the other service after some bit sequences assigned to £le names on the expanded service are assigned to £le names on the other service.

Let $\mathfrak{M} = (BS, \{\mu_n \mid n \in \mathbb{N}\}, E)$ and $\mathfrak{M}' = (BS', \{\mu'_n \mid n \in \mathbb{N}\}, E')$ be code controlled machine structures, let (CCN, ψ) and (CCN, ψ') be control code notations for $\mathfrak{M}$ and $\mathfrak{M}'$, respectively, let $c \in CCN$, and let $EAS_0 = H_{(\sigma_0, x_0)} \in \mathcal{EAS}^{\mathfrak{M}}$ and $EAS'_0 = H'_{(\sigma'_0, x'_0)} \in \mathcal{EAS}^{\mathfrak{M}'}$. Then c is *portable* from EAS_0 to EAS'_0 if

- $\forall \langle bs_1, \ldots, bs_m \rangle \in BS^* \bullet$
  $(\psi(c) \bullet\bullet^1_{\mathfrak{M}} bs_1, \ldots, bs_m \neq \mathsf{D}$
  $\Rightarrow \bigwedge_{n \in \mathbb{N}} \psi(c) \bullet\bullet^n_{\mathfrak{M}} bs_1, \ldots, bs_m = \psi'(c) \bullet\bullet^n_{\mathfrak{M}} bs_1, \ldots, bs_m)$.
- there exists a $EAS_1 = H_{(\sigma_1, x_1)} \in \mathcal{EAS}^{\mathfrak{M}}$ such that
  - EAS_0 is expansible to EAS_1,
  - there exist $f_1, \ldots, f_l \in \mathrm{dom}(\sigma_1) \setminus \mathrm{dom}(\sigma'_0)$ such that c is pre-installed on $(\mathsf{ea.set}{:}f_1{:}\sigma_1(f_1) \circ \ldots \circ \mathsf{ea.set}{:}f_l{:}\sigma_1(f_l)) \bullet_{\mathsf{ea}} EAS'_0$.

Because we assume that the set $\mathcal{FN}m$ of £le names is countably in£nite, this de£nition does not imply that the bit sequences to be transported have to be assigned to the same £le names at both sides.

Example 4 Take a source code notation SCN and an assembly code notation ACN. Consider an execution architecture service EAS on which £le name f_1 has assigned an executable version of a compiler for SCN that produces assembly codes from ACN, £le name f_2 has assigned a source code from SCN, and £le name f_3 has nothing assigned. Moreover, consider another execution architecture service EAS' on which £le name f_1 has assigned an executable version of an assembler for ACN, and £le name f_3 has nothing assigned. Suppose that the above-mentioned prerequisite for portability of the source code is ful-£lled. Then the source code is portable from EAS to EAS' because it is pre-installed on $\mathsf{ea.set}{:}f_3{:}bs \bullet_{\mathsf{ea}} EAS'$ where bs is the bit sequence assigned to f_3 on $(\mathsf{ea.load}{:}f_1 \circ \mathsf{ea.exec}{:}f_2{>}f_3) \bullet_{\mathsf{ea}} EAS$.

11 Conclusions

We have presented a logical approach to explain issues concerning control codes that are independent of the details of the behaviours that are controlled at a very abstract level. We have illustrated the approach by means of examples which demonstrate that there are non-trivial issues that can be explained at this level. In the explanations given, we have consciously been guided by empirical viewpoints usually taken by practitioners rather than theoretical viewpoints. The issues that have

been considered are well understood for quite a time. Application of the approach to issues that are not yet well understood is left for future work. We think among other things of applications in the areas of software asset sourcing, which is an important part of IT sourcing, and software patents.

We have based the approach on abstract machine models, called machine structures. If systems that provide execution environments for the executable codes of machine structures are involved in the issues to be explained, then more is needed. We have introduced an execution architecture for machine structures as a model of such systems and have explained portability of control codes using this execution architecture and an extension of basic thread algebra. The execution architecture for machine structures, as well as the extension of basic thread algebra, may form part of a setting in which the different kinds of processes that are often transferred when sourcing software assets, in particular software exploitation, can be described and discussed.

References

1. J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
2. J. A. Bergstra. Machine function based control code algebras. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO 2003*, volume 3188 of *Lecture Notes in Computer Science*, pages 17–41. Springer-Verlag, 2004.
3. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
4. J. A. Bergstra and P. Klint. About "trivial" software patents: The IsNot case. *Science of Computer Programming*, 64:264–285, 2006.
5. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
6. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
7. J. A. Bergstra and C. A. Middelburg. Machine structure oriented control code logic. Computer Science Report 07-10, Department of Mathematics and Computer Science, Eindhoven University of Technology, April 2007.
8. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
9. J. A. Bergstra and A. Ponse. Execution architectures for program algebra. *Journal of Applied Logic*, 5:170–192, 2007.
10. H. Bratman. An alternate form of the UNCOL diagram. *Communications of the ACM*, 4(3):142, 1961.
11. G. Delen. *Decision- en Controlfactoren voor IT-Sourcing*. Van Haren Publishing, Zaltbommel, Netherlands, 2005. In Dutch.
12. J. Earley and H. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.
13. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Speci£cation*. Addison-Wesley, Reading, MA, second edition, 2000.

15. A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Speci£cation*. Addison-Wesley, Reading, MA, 2003.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Speci£cation*. Addison-Wesley, Reading, MA, 1996.
18. L. Loh and N. Venkatraman. Diffusion of information technology outsourcing, in¤uence sources and the Kodak effect. *Information Systems Research*, 4:334–358, 1992.
19. R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989.
20. C. Verhoef. Quantitative IT portfolio management. *Science of Computer Programming*, 45(1):1–96, 2002.
21. D. Watkins, M. Hammond, and B. Abrams. *Programming in the .NET Environment*. Addison-Wesley, Reading, MA, 2003.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0703] J.A. Bergstra and C.A. Middelburg, *On the Operating Unit Size of Load/Store Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures,* Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory,* Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital,* Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation,* Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads,* Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading,* Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises,* Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields,* Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops,* Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration),* Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction,* Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures,* Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF,* Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics,* Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA,* Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads,* Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result,* Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services,* Programming Research Group - University of Amsterdam, 2004.

[PRG0404]   J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving,* Programming Research Group - University of Amsterdam, 2004.

[PRG0403]   B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot,* Programming Research Group - University of Amsterdam, 2004.

[PRG0402]   J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax,* Programming Research Group - University of Amsterdam, 2004.

[PRG0401]   B. Diertens, *Molecular Scripting Primitives,* Programming Research Group - University of Amsterdam, 2004.

[PRG0302]   B. Diertens, *A Toolset for PGA,* Programming Research Group - University of Amsterdam, 2003.

[PRG0301]   J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs,* Programming Research Group - University of Amsterdam, 2003.

[PRG0201]   I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences,* Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series