Machine structure oriented control code logic

Citation for published version (APA):

Document status and date:
Published: 01/01/2007

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 01. Apr. 2019
Machine Structure Oriented Control Code Logic

J.A. Bergstra\textsuperscript{1,2} and C.A. Middelburg\textsuperscript{1,3}

\textsuperscript{1} Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
\textsuperscript{2} Department of Philosophy, Utrecht University,
P.O. Box 80126, 3508 TC Utrecht, the Netherlands
\textsuperscript{3} Computing Science Department, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
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 use the approach among other things to explain the well-known compiler fixed point.

Keywords: machine structure, control code, compiler fixed point.


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

\textsuperscript{*} This research was carried out as part of the Jacquard-project Symbiosis, which is funded by the Netherlands Organisation for Scientific Research (NWO).
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.

Machine structures are used as a basis of our approach. They are inspired by the machine functions introduced in [8] to provide a mathematical basis for the T-diagrams proposed in [6]. A machine structure offers a machine model at a very abstract level.

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, software patents, and software portability. Moreover, we find that control code production is in the end what software construction is about.

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.

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). After that, we sum up the effects of withdrawing a simplifying assumption concerning the representation of control codes made in the foregoing (Section 6). Following this, we use a synthetic execution architecture in the sense of [5] to give expression to the use of machine structures (Section 7). Finally, we make some concluding remarks and mention some options for future work (Section 8).

An in-depth treatment of quantitative IT portfolio management can be found in [13]. Software asset sourcing is an important part of IT sourcing, see e.g. [12, 7]. An extensive study of software patents and their implications on software engineering practices can be found in [3].
This paper is a major revision of [2]. It has been substantially rewritten. In particular, several important technical aspects have been significantly modified.

2 Machine Function 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 [8] by covering machines with several outputs. Machine structures can easily be defined without reference to machine functions. The introduction of machine functions is mainly for expository reasons.

2.1 Machine Functions

A machine function $f$ is actually a family of functions: it consists of a function $f_n$ for each natural number $n > 0$. Those functions map each finite sequence of bit sequences to either a bit sequence or $M$ or $D$. Here, $M$ stands for meaningless and $D$ stands for divergence. 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 $f$ and the machine modelled by it can be understood as follows:

- if $f_n(\langle x_1, \ldots, x_m \rangle)$ is a bit sequence, then the machine function $f$ models a machine that produces $f_n(\langle x_1, \ldots, x_m \rangle)$ as its $n$th output on it taking $x_1, \ldots, x_m$ as its inputs;
- if $f_n(\langle x_1, \ldots, x_m \rangle)$ is $M$, then the machine function $f$ models a machine that produces less than $n$ outputs on it taking $x_1, \ldots, x_m$ as its inputs;
- if $f_n(\langle x_1, \ldots, x_m \rangle)$ is $D$, then the machine function $f$ 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 finitely 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 first 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 $\mathbf{BS}$ for the set $\{0, 1\}^*$ of bit sequences.

We now define machine functions in a mathematically precise way. Let $\mathbf{BS} \subseteq \mathbf{BS}$. Then a machine function $f$ on $\mathbf{BS}$ is a family of functions

$$\{f_n : \mathbf{BS}^* \rightarrow (\mathbf{BS} \cup \{D, M\}) \mid n \in \mathbb{N}\}$$

satisfying the following rules:

$$\bigwedge_{n \in \mathbb{N}} \left( \bigwedge_{m \in \mathbb{N}} (f_n(\chi) = D \Rightarrow f_m(\chi) = D) \right),$$

$$\bigwedge_{n \in \mathbb{N}} \left( f_n(\chi) \neq D \Rightarrow \left( \bigvee_{m \in \mathbb{N}, m > n} f_m(\chi) = M \right) \right),$$

$$\bigwedge_{n \in \mathbb{N}} \left( f_n(\chi \rightarrow \chi') = M \Rightarrow f_n(\chi) = M \right),$$

$$\bigwedge_{n \in \mathbb{N}} \left( f_n(\chi \rightarrow \chi') = M \Rightarrow f_n(\chi) = M \right).$$

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

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

$$\mathbf{cf}_2(\langle x \rangle) = \langle \rangle \Rightarrow \mathbf{cf}_1(\langle x \rangle) \neq \langle \rangle,$$

$$\mathbf{df}_2(\langle x \rangle) = \langle \rangle \Rightarrow \mathbf{df}_1(\langle x \rangle) \neq \langle \rangle,$$

$$\mathbf{cf}_2(\langle x \rangle) = \langle \rangle \Rightarrow \mathbf{df}_1(\mathbf{cf}_3(\langle x \rangle)) = \mathbf{cf}_1(\langle x \rangle).$$

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 first 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 $\mathcal{M}$ consists of a set of bit sequences $BS$, functions $f_n$ that make up a machine function on $BS$, and a subset $E$ of $BS$. If $E$ is empty, then the machine structure $\mathcal{M}$ is essentially the same as the machine function contained in it. If $E$ is not empty, then the machine structure $\mathcal{M}$ is supposed to model a code controlled machine. In the case where $E$ is not empty, the connection between the machine structure $\mathcal{M}$ 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 $\mathcal{M}$ 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 $\mathcal{M}$;
- if $x \in E$, then the functions $f'_n$ that are defined by $f'_n(\langle y_1, \ldots, y_m \rangle) = f_n(\langle x, y_1, \ldots, y_m \rangle)$ make up a machine function on $BS$ modeling a machine that exhibits the same behaviour as the code controlled machine modelled by $\mathcal{M}$ 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 $BS \subseteq \mathcal{B}$,
- a unary function $f_n : BS^* \to (BS \cup \{D, M\})$, for each $n \in \mathbb{N}$,
- a unary relation $E \subseteq BS$,

where the family of functions $\{f_n : BS^* \to (BS \cup \{D, M\}) \mid n \in \mathbb{N}\}$ 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 $\mathfrak{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ 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

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

where the functions $f'_n$ are defined by

$$f'_n(y_1, \ldots, y_m) = f_n(\langle x, y_1, \ldots, y_m \rangle).$$

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

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

$$x \bullet_{\mathfrak{M}} y_1, \ldots, y_m \quad \text{for} \quad f_n(\langle x, y_1, \ldots, y_m \rangle).$$

Moreover, we will write

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

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

**Example 2.** Take a code controlled machine structure $\mathfrak{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$. 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 $n \in \mathbb{N}$:

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

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 $\mathcal{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ determines all by itself a machine model. For an execution, which takes a single step, an executable code $x \in E$, a sequence $(y_1, \ldots, y_m) \in BS^*$ of inputs and the machine function $\{f_n \mid n \in \mathbb{N}\}$ 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 first 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 justification 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 justification for the correspondence mentioned above can be found only if the machine function involved is asymmetric and moreover the first bit sequence in the argument of the function that yields the first 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 $\mathcal{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ 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 $\{f_n \mid n \in \mathbb{N}\}$ is asymmetric if there exist $x, y \in BS$ such that $f_1(x, y) \neq f_1(y, x)$. The first bit sequence in the argument of the function $f_1$ 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 $f_1(x_1, y) = z_1$ and $f_1(x_2, y) = z_2$ for all $y \in BS$. It is easily proved that the first bit sequence in the argument of the function $f_1$ overrules the second one only if the second bit sequence in the argument of the function $f_1$ does not overrule the first 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 first 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 first 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 fixed 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 fixed 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 \( M = (BS, \{ f_n \mid n \in \mathbb{N} \}, E) \) be a code controlled machine structure. Then a control code notation for \( M \) consists of a set \( CCN \subseteq BS \) and a function \( \psi: CCN \rightarrow E \). The members of \( CCN \) are called control codes for \( M \). The function \( \psi \) is called a machine structure projection.

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

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

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 \( M = (BS, \{ f_n \mid n \in \mathbb{N} \}, E) \) 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 \( M \). We trivially have \( |e|_E^M = |e|^M_\mathcal{E} \) 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 qualifies as a program notation, it is in fact immaterial how the concept of program is defined. 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 definition 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 $\mid_{PGN}: PGN \rightarrow MF$ which maps programs to the machine functions modelling their behaviour at the abstraction level of input-output relations.

In [4], a theory, called program algebra, is introduced in which a program is a finite or infinite 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 [5]. In the current paper, the definition of the concept of program from [4] could be used. However, we have not fixed a particular concept of program because we intend to abstract from the details involved in any such conceptual definition.

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 qualifies 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 qualifies as a program notation, then its elements are considered programs.

Let $\mathcal{M}$ be a code controlled machine structure, and let $(CCN, \psi)$ be a control code notation for $\mathcal{M}$. Then $(CCN, \psi)$ qualifies as a program notation if there exist an acknowledged program notation $PGN$ and a function $\phi: CCN \rightarrow PGN$ such that for all $c \in CCN$:

$$\mid \psi(c) \mid_{\mathcal{M}} = \mid \phi(c) \mid_{PGN} .$$

This definition implies that, in the case of a control code notation that qualifies 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 $\phi$ whose existence is demanded in the definition 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 $\phi$ is the opposite of a representation. It might be called a co-representation.

Suppose that $\mathcal{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ is a code controlled machine structure and $(E, 1_E)$ qualifies as a program notation. Then $\mathcal{M}$ models a code controlled machine whose executable codes constitute a control code notation that qualifies as a program notation. Therefore, it is appropriate to call $\mathcal{M}$ 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 $\mathcal{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then $\mathcal{M}$ is a proper code controlled machine structure if $(E', 1_{E'})$ qualifies as a program notation for some $E' \subseteq E$ only if $(E, 1_E)$ qualifies 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 qualifies 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 artificial intelligence.

Consider a neural network in hardware form, which is able to learn while working on a problem and thereby defining parameter values for many firing thresholds for artificial 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 find 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 first example shows that control codes can be produced without programming by means of artificial intelligence based techniques. The second example shows that the behaviour of machines applying artificial 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 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 qualifies 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 sufficient 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 $M$, disjoint sets $ACN_M$ and $SCN_M$ of control code notations for $M$ have been given. The members of $ACN_M$ and $SCN_M$ are called assembly code notations for $M$ and source code notations for $M$, respectively.

The following gives an idea of the grounds on which control code notations are classified 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 specific 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 specific to a machine.

A high-level programming language, such as Java [9] or C# [10], is considered a source code notation. The term high-level programming language suggests that it concerns a notation that qualifies as a program notation. However, as mentioned above, whether a source code notation qualifies 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 $M = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let $(CCN, \psi), (CCN', \psi')$ and $(CCN'', \psi'')$ be control code notations for $M$. Then we write $cc [CCN' \rightarrow CCN''] : CCN$ for

$$cc \in CCN \land \forall cc' \in CCN' \cdot (\exists cc'' \in CCN'' \cdot \psi(cc) \bullet cc' = cc'')$$

We say that $cc$ is in executable form if $CCN \subseteq E$, that $cc$ is in assembly form if $CCN \in ACN_M$, and that $cc$ is in source form if $CCN \in SCN_M$.

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 $M = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure, and let $(ACN, \psi)$ be a control code notation for $M$ that belongs to $ACN_M$. Suppose that $ass [ACN \rightarrow E] : E$ is an existing assembler for $ACN$. This assembler is in executable form. Suppose further that a new assembler $ass' [ACN \rightarrow E] : ACN$ 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.

$$\forall ac \in ACN \bullet ass \bullet\bullet ac \equiv_{beh} (ass \bullet\bullet ass') \bullet\bullet ac.$$  \hspace{1cm} (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,

$$ass'' \equiv_{beh} ass'''.$$  \hspace{1cm} (2)

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

$$ass \bullet\bullet ass' \equiv_{beh} (ass \bullet\bullet ass') \bullet\bullet ass',$$  \hspace{1cm} (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,

$$ass''' = ass'' \bullet\bullet ass'.$$  \hspace{1cm} (4)

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

$$ass'' \bullet\bullet ass' = ass''' \bullet\bullet ass',$$  \hspace{1cm} (5)

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

In theoretical computer science, correctness of a program is taken to mean that the program satisfies a mathematically precise specification of it. For the assembler $ass'$, $\forall ac \in ACN \bullet \psi(ass') \bullet\bullet ac = \psi(ac)$ would be an obvious mathematically precise specification. 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 specific 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 \( \mathcal{M} = (BS, \{ f_n \mid n \in \mathbb{N} \}, E) \) be a code controlled machine structure, let \((SCN, \psi_s)\) be a control code notation for \( \mathcal{M} \) that belongs to \( SCN_{\mathcal{M}} \), and let \((ACN, \psi_a)\) be a control code notation for \( \mathcal{M} \) that belongs to \( ACN_{\mathcal{M}} \). Suppose that \( com \colon SCN \rightarrow ACN \) is an existing compiler for \( SCN \) and \( ass \colon ACN \rightarrow E \) 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 \( com' \colon SCN \rightarrow ACN \) 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

\[
\forall sc \in SCN \bullet \nonumber \\
ass \bullet ((\ass \bullet com) \bullet sc) \equiv_{Ile} \ass \bullet ((\ass \bullet ((\ass \bullet com) \bullet com')) \bullet sc). \tag{6}
\]

Let \( com'' \) be the new compiler in assembly form obtained by compiling \( com' \) by means of \( com \), i.e. \( com'' = (ass \bullet com) \bullet com' \). 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. \( com''' = (ass \bullet com') \bullet com' = (ass \bullet ((ass \bullet com) \bullet com')) \bullet com'. \) If \( com' \) is correct, then \( com'' \) and \( com''' \) produce the same assembly codes. That is,

\[
ass \bullet com'' \equiv_{Ile} ass \bullet com'. \tag{7}
\]

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

\[
ass \bullet ((ass \bullet com) \bullet com') \equiv_{Ile} ass \bullet ((ass \bullet ((ass \bullet com) \bullet com')) \bullet com'), \tag{8}
\]

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,

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

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

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

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

The explanation of the compiler fixed point proceeds similar to the explanation of the assembler fixed 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 fixed point is essentially the same as the explanation of the assembler fixed 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 briefly 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 sufficient 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 classified as intermediate code notation. An intermediate code notation is a control code notation that resembles an assembly code notation, but it is not specific to any machine. Often, it is specific 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 specific 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 [14] 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 \( M = (BS, \{ f_n | n \in \mathbb{N} \}, E) \) be a code controlled machine structure, let \((SCN, \psi_s)\) be a control code notation for \( M \) that belongs to \( SCN_M \), let \((ICN, \psi_i)\) be a control code notation for \( M \) that belongs to \( ICN_M \), and let \((ACN, \psi_a)\) be a control code notation for \( M \) that belongs to \( ACN_M \). Suppose that \( com_{a(SCN \rightarrow ACN)} : ACN \) is an existing compiler for \( SCN \) and \( ass_{(ACN \rightarrow E)} : E \) is an existing assembler for \( ACN \). The compiler \( com_{a} \) lets \( M \) transform source codes into assembly codes. Suppose further that a new compiler \( com_i(SCN \rightarrow ICN) : ACN \) for \( SCN \) and a new interpreter \( int \in E \) for \( ICN \) are made available. The compiler \( com_i \) lets \( M \) transform source codes into intermediate codes.

The combination of \( com_i \) and \( int \) is considered correct if

\[
\forall sc \in SCN, (bs_1, \ldots, bs_m) \in BS^* \cdot \\
(ass \bullet_{2M} ((ass \bullet_{2M} com_a) \bullet_{2M} sc)) \bullet_{2M} bs_1, \ldots, bs_m = int \bullet_{2M} ((ass \bullet_{2M} com_i) \bullet_{2M} sc), bs_1, \ldots, bs_m .
\] (11)

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 [11].

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 $\mathcal{M} = (BS, \{f_n \mid n \in \mathbb{N}\}, E)$ be a code controlled machine structure. Then a control code notation for $\mathcal{M}$ 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 $\mathcal{M}$. The function $\rho$ is called the bs-representation function of $CCN$.

Let $CCN, \psi, \rho$ be a control code notation for a code controlled machine structure $(BS, \{f_n \mid n \in \mathbb{N}\}, E)$. 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 definition of the special notation $cc [CCN' \to CCN''] : CCN$ slightly.

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

$$cc \in CCN \land \forall cc' \in CCN' \cdot (\exists cc'' \in CCN'' \cdot \psi(cc) \bullet \rho'(cc') = \rho''(cc'')) .$$

The Explanation of the Assembler Fixed Point

In the explanation of the assembler fixed point given in Section 4.3, we have to replace the definitions of $ass''$ and $ass'''$ by $ass'' = ass \bullet \rho(ass')$ and $ass''' = (ass \bullet \rho(ass')) \bullet \rho(ass')$, assuming that $\rho$ 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

$$\forall ac \in ACN \cdot ass \bullet \rho(ac) \equiv_{beh}^{\mathcal{M}} (ass \bullet \rho(ass')) \bullet \rho(ac) .$$

Formula (3) must be replaced by

$$ass \bullet \rho(ass') \equiv_{beh}^{\mathcal{M}} (ass \bullet \rho(ass')) \bullet \rho(ass') .$$

Formula (4) must be replaced by

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

Formula (5) must be replaced by

$$ass'' \bullet \rho(ass') = ass''' \bullet \rho(ass') .$$
The Explanation of the Compiler Fixed Point

In the explanation of the compiler fixed point given in Section 4.4, we have to replace the definitions of \( \text{com}'' \) and \( \text{com}''' \) by 

\[
\text{com}'' = (\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(\text{com})
\]

and 

\[
\text{com}''' = (\text{ass} \cdot (\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(\text{com}')) \cdot \rho_s(\text{com}'),
\]

assuming that \( \rho_s \) is the bs-representation function of \( \text{SCN} \) and \( \rho_a \) is the bs-representation function of \( \text{ACN} \). Moreover, we have to adapt Formulas (6), (8), (9), and (10) slightly. Formula (6) must be replaced by

\[
\forall sc \in \text{SCN} \bullet 
\begin{align*}
\text{ass} \cdot ((\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(sc)) \\
&\equiv^{\text{beh}} \text{ass} \cdot ((\text{ass} \cdot (\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(\text{com}'))) \cdot \rho_s(sc).
\end{align*}
\]

Formula (8) must be replaced by

\[
\text{ass} \cdot ((\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(\text{com}')) \\
&\equiv^{\text{beh}} \text{ass} \cdot ((\text{ass} \cdot (\text{ass} \cdot \rho_a(\text{com})) \cdot \rho_s(\text{com}'))) \cdot \rho_s(\text{com}').
\]

Formula (9) must be replaced by

\[
\text{com}''' = (\text{ass} \cdot \text{com}''') \cdot \rho_s(\text{com}').
\]

Formula (10) must be replaced by

\[
(\text{ass} \cdot \text{com}''') \cdot \rho_s(\text{com}') = (\text{ass} \cdot \text{com}'''') \cdot \rho_s(\text{com}').
\]

The Correctness Criterion for Interpreters

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

\[
\forall sc \in \text{SCN}, \{bs_1, \ldots, bs_m\} \in \text{BS}^* \bullet 
\begin{align*}
(\text{ass} \cdot (\text{ass} \cdot \rho_a(\text{com}))) \cdot \rho_s(sc) \\
= \text{int} (\text{ass} \cdot \rho_a(\text{com})), \{bs_1, \ldots, bs_m\},
\end{align*}
\]

assuming that \( \rho_s \) is the bs-representation function of \( \text{SCN} \), \( \rho_l \) is the bs-representation function of \( \text{ICN} \), and \( \rho_a \) is the bs-representation function of \( \text{ACN} \).

7 Execution Architectures for Machine Structures

Synthetic execution architectures open up a way to give an operational explanation of code controlled machine structures.\(^5\) That is, synthetic execution architectures can provide a context in which expression is given to the use of

\(^5\) Analytic execution architectures, which are referred to in Section 3.2, and synthetic execution architectures are discussed in [5].
code controlled machine structures. In this section, we present such a synthetic execution architecture.

The synthetic execution architecture is a machine in which a code controlled machine structure is incorporated. The machine is operated by means of instructions that either yield a reply or diverge. The possible replies are T and F. In the instructions file names are used to refer to the bit sequences present in the machine. It is assumed that a set \( F_{Nm} \) of file names has been given. While designing the instruction set, we focussed on convenience of use rather than minimality.

Let \( \mathcal{M} = (BS, \{ f_n \mid n \in \mathbb{N} \}, E) \) be a code controlled machine structures. Then the machine in which \( \mathcal{M} \) is incorporated has the following instructions:

- for each \( f \in F_{Nm} \) and \( bs \in BS \), a set instruction \( \text{set} f \ bs \);
- for each \( f \in F_{Nm} \), a remove instruction \( \text{rm} f \);
- for each \( f_1, f_2 \in F_{Nm} \), a copy instruction \( \text{cp} f_1 f_2 \);
- for each \( f_1, f_2 \in F_{Nm} \), a move instruction \( \text{mv} f_1 f_2 \);
- for each \( f_1, f_2 \in F_{Nm} \), a concatenation instruction \( \text{cat} f_1 f_2 \);
- for each \( f_1, f_2 \in F_{Nm} \), a test on equality instruction \( \text{tsteq} f_1 f_2 \);
- for each \( f_1, f_2 \in F_{Nm} \), a test on difference instruction \( \text{tstne} f_1 f_2 \);
- for each \( f \in F_{Nm} \), a test on existence instruction \( \text{tstex} f \);
- for each \( f \in F_{Nm} \), a load instruction \( \text{load} f \);
- for each \( f_1, \ldots, f_m, f'_1, \ldots, f'_n \in F_{Nm} \), an execute instruction \( \text{exec} f_1 \ldots f_m > f'_1 \ldots f'_n \).

We say that a file name is in use if it has a bit sequence assigned. The 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 as follows:

- \( \text{set} f \ bs \): 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;
- \( \text{rm} f \): 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;
- \( \text{cp} f_1 f_2 \): 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;
- \( \text{mv} f_1 f_2 \): 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;
- \( \text{cat} f_1 f_2 \): 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;
- \( \text{tsteq} f_1 f_2 \): 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;
- \texttt{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;
- \texttt{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;
- \texttt{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;
- \texttt{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 \( x \cdotbs_1 \ldots bs_m \in BS \), then:
    * \( x \cdotbs_1 \ldots bs_m \) is assigned to \( f'_i \) for each \( i \) with \( 1 \leq i \leq n \) such that \( x \cdotbs_1 \ldots bs_m \neq M \),
    * \( f'_i \) is removed from the file names in use for each \( i \) with \( 1 \leq i \leq n \) such that \( x \cdotbs_1 \ldots bs_m = M \),
  and the reply is T;
  * if \( x \cdotbs_1 \ldots bs_m = M \), then nothing changes and the reply is F;
  * if \( x \cdotbs_1 \ldots bs_m = D \), then the machine does not halt;
  otherwise, nothing changes and the reply is F.

Note that there are three cases in which the instruction \texttt{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 file name among \( f_1, \ldots, f_m \) that is not in use; (c) there is no output produced.

The instructions of which the effect depends on the code controlled machine structure \( \mathcal{M} \) are the load and execute instructions only. These instructions are also the only essential instructions: all other instructions could be eliminated in favour of executable codes, assigned to known file names. However, we believe that elimination of the unessential instructions would not contribute to convenient use of the execution architecture. A distinction is made between loading and execution of executable codes in order not to mix two different aspects of code controlled machine structures. Moreover, the distinction allows for telling load-time errors from run-time errors.

8 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 expect to find applications among other things in the area of software asset sourcing.
References