A Formal Model of Forth Control Words in the Pi-Calculus

James Power
Department of Computer Science
National University of Ireland, Maynooth
Maynooth, Co. Kildare, Ireland

David Sinclair
School of Computer Applications
Dublin City University
Glasnevin, Dublin 9, Ireland

May 28, 2001

Abstract

In this paper we develop a formal specification of aspects of the Forth programming language. We describe the operation of the Forth compiler as it translates Forth control words, dealing in particular with the interpretation of immediate words during compilation. Our goal here is to provide a basis for the study of safety properties of embedded systems, many of which are constructed using Forth or Forth-like languages. To this end we construct a model of the Forth compiler in the π-calculus, and have simulated its execution by animating this model using the Pict *programming language.*

1 Introduction

In this paper we seek to contribute to the study of stack-based languages and architectures by providing a model of the control structures used in the Forth programming language. Stack-based machines have a long history in programming language implementation, but lately have achieved increase prominence due to the widespread use of the Java programming language, and its corresponding implementation architecture, the Java Virtual Machine (JVM) [8].

The Forth programming language is relatively old in the context of high-level programming languages, dating from around 1970. Its emphasis on high performance coupled with a small memory footprint has helped establish the language, particularly in relation to embedded microcontroller systems and similar industrial applications. Reflecting this, Forth has been standardised by both ANSI and the ISO [6].

In this section we give an overview of stack-based machines and point to some of the strengths and weaknesses of existing implementations of this approach. In particular we highlight the importance of formal models in establishing safety properties, particularly in relation to Forth's control structures. We also present an overview of the π-calculus, the formalism used later in the paper to model the interaction between the processes in the Forth compiler.

1.1 Stack-Based Machines

In the extreme case a stack-based language will insist on all programming activities being performed directly on the stack, including expression evaluation, local variable storage, parameter

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

passing and the return of result values from functions. More realistically, many such language will mask these basic operations with a friendlier syntax.

Much of the original motivation for the use of stack-based machines in programming language translation was as an abstract "machine code". Such languages were low-level enough to allow straightforward translation to a given assembly language for a specific architecture, but yet sufficiently high-level to allow the compiler-writer ignore many implementation details, most particularly the number and nature of the target architecture's registers.

The development and increasing popularity of the Java programming language and the JVM however have sparked renewed interest in the pragmatics surrounding stack-based language design. The Java language technology typically involves a Java compiler that translates Java programs into bytecode, along with an interpreter (the JVM) that executes this code. However, such systems still have difficulty in competing with Forth-based applications in terms of speed and memory efficiency [2], although the situation is improving.

Forth combines a distinctive postfix stack-based programming approach with a remarkably economical syntax. While fundamentally similar to the JVM architecture, Forth differs from the JVM in that it does not provide direct primitive support for classes and objects. However, Forth does provide a flexible yet structured approach to the implementation of flow of control that contrasts sharply with that of the JVM, and it is Forth's implementation of these structures that forms the focus of this paper.

Forth control structures are a unique combination of high-level structured concepts with the flexibility of low-level test-and-branch operation. Further, Forth is unusual in that it allows the programmer define new control words, providing for a variety of possible constructs. One disadvantage of this approach, however, is that these control words can increase the complexity of the code, and can cause unexpected side-effects if used incautiously.

Systems such as the JVM incur significant performance overheads by dealing with safety properties at run time, using a bytecode verifier. Indeed, one alternative to JVM bytecode, also designed to allow safe, mobile code, deliberately preserves high-level control structures for this reason [4]. However, in industrial critial applications, and particularly with embedded systems, run-time failures are unacceptable, and considerable emphasis is placed on testing and static verification of the software.

1.2 Formal Semantics and the π-Calculus

In the following sections we present a formal model of the Forth compiler, concentrating on the compilation of Forth control words. Providing formal definitions of programming languages is a well-established field, usually known as formal semantics (see e.g. [16] for a survey) and, in this context, our definition would most likely be classed as an *operational semantics* of Forth.

However, the structure of the Forth compiler lends itself to a particular form of specification. Specifically, the operation of Forth is most often described in terms of the Forth engine's concurrent interactions with the control, data and return stacks, with the total effect being the parallel composition of these interactions. Thus the words in a Forth program become events that trigger changes in the processes representing the Forth stacks, along with other internal structures such as the dictionary.

This contrasts with the *compositional* approach typically taken in denotational descriptions, such as in [14], and the *structural* approach taken in modern operational semantics, as in [5]. Our

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

model differs from both these approaches since, in each case, the definition is structured around the (abstract) syntax of the programming language being defined. This makes sense for high-level languages which express control using nested syntactic structures (such as if statements, while-loops etc.), but is not so appropriate to Forth, where a program is simply a sequential list of words, with no real nesting.

Thus we take a different approach, making the processes of the Forth system central to our model, and viewing the program text as a stream of events which affect these processes. In this, our semantics bears a similarity to previous specifications of aspects of Forth, such as [7] and particularly [15]. However, these specifications concentrated on the execution of Forth programs, whereas our specification concentrates on the actions of the Forth compiler.

The modelling formalism that we have chosen to use here is the π-calculus [9]. The π-calculus provides a primitive set of operations for describing the interactions between communicating processes, as well as allowing for the movement of communication channels between these processes. This formalism is not typically used to specify the formal semantics of programming languages ([13] is an exception, but even this concentrates on concurrency aspects of the language).

However, we believe that the π-calculus is particularly suited to providing an operational model of the web of interactions between various components of the Forth system. To model the Forth system it is necessary to allow for a number of interacting processes which may include either compile-time or run-time behaviour, as the system switches from compiler to interpreter mode. In addition, the use of immediate and postponed words within word definitions further enmesh these processes, making their presentation using standard structural approaches quite difficult. Also, the basic purpose of the Forth compiler, the definition of new words, finds a natural model in the π-calculus primitives for the creation and movement of names between processes, and the necessary reconfiguration of communications between these processes.

We will not give a full presentation of the calculus here; we seek only to introduce the notation used in the rest of this paper. The two main events that can occur are:

- $c(n)$, denoting the receipt of some message, hereby named n, along a channel c, and
- $\overline{c}\langle n\rangle$, denoting the sending of some existing name n out along the channel c

In the π-calculus channels are first-order objects; that is, channels can be sent and received along other channels. The silent event, τ denotes an internal action, hidden to other processes.

For our purposes, processes can then be described as:

- $e.P$, where e is an event and P is a process - the process P here is guarded by the event e
- $P_1 + P_2$ denoting nondeterministic choice between processes P_1 and P_2
- $P_1 \mid P_2$, which denotes the processes P_1 and P_2 being run in parallel
- new a (P) introducing (and binding) the new name a in process P
- $P_1; P_2$, which denotes the sequential composition of processes P_1 and P_2[1]

[1]If we assume that every process performs the action $\overline{done}\langle\rangle$ as its last action we can define sequential composition as a special case of parallel composition.
$P; Q \stackrel{\text{def}}{=}$ new $start$ $(\{start/done\}P \mid start.Q)$

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

Two processes running in parallel may use guards to synchronise; the basic reaction rule formalises the synchronisation in a manner similar to β-reduction in the lambda calculus:

$$(x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \quad \rightarrow \quad \{\frac{z}{y}\}P \mid Q$$

1.3 Notation

To increase the readability of our specification, we use some notational conveniences not primitive to the π-calculus:

- We use macro-like definitions to give names to processes; thus $P(n) \stackrel{\texttt{def}}{=} Q$ defines the macro P indexed by n, defined to be the same as process Q after suitable substitution for parameter n.
- Tuples are represented using square brackets; for example: $[x, y]$ is the pair consisting of x and y.
- We use $l_1{}^\wedge l_2$ to represent the concatenation of lists l_1 and l_2, and overload this notation to also apply to list elements. We use the constant Nil to represent the empty list, and the functions hd and tl to represent the usual list head and tail operations.

All of the specifications presented in the following sections have been type-checked and tested using the Pict system [11], a programming language based closely on the π-calculus. The specification was translated almost directly into Pict, which helped to check the consistency of the model, and, as an executable language, allowed us to simulate the actions of the compiler as it dealt with various configurations of source code. The full Pict source code is presented in [12].

2 Overview of the Model

The Forth programming language is characterised by the fact that it not only a stack-based language, but also a semi-compiled language. A Forth program consists of a series of words, which can be either Forth standard words or words defined by the user. These words are processed sequentially by the *Interpreter*. Each word has an entry in the *Dictionary* and the complied code associated with the word read by the *Interpreter* is retrieved from the *Dictionary* and executed. The effect of each word is to modify the state of one of the Forth stacks.

The Forth language has three stacks. The *Data Stack* is used to manipulate data. The *Return Stack* is used store the return addresses of the calling procedures[2]. The *Control Flow Stack* is used to implement control flow management. Forth provides a set of standard words that provide a more structured approach to control flow management than the basic low-level branches and labels implemented by many assembly languages.

In this paper we are interested in Forth's approach in providing structured control flow primitives, as we believe that this has a fundamental impact on verification and optimisation, as well as a possible influence the design of future intermediate representations. We will abstract the syntax of a Forth program to the primitives shown in figure 1.

[2]Forth allows the user to access this stack using the words >R and R>

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

POSTPONE	Postpone the execution of an immediate word when in compiler mode.
: *wordname*	In interpreter mode execute the user defined word. In compiler mode execute the user defined word if it is an immediate word, otherwise append its compiled code to the complied code of the word currently being complied.
;	Finish compiling the current word and mark it as a non-immediate word.
; IMMEDIATE	Finish compiling the current word and mark it as an immediate word. We will treat ; IMMEDIATE as one word.
CHANGE	We will use CHANGE as an abstraction of all Forth words that modify the data stack.
IF	This is an immediate word that marks the origin of a forward conditional branch.
AHEAD	This is an immediate word that marks the origin of a forward unconditional branch.
THEN	This is a immediate word that resolves an IF or AHEAD.
BEGIN	This is an immediate word that marks a backward destination.
AGAIN	This is an immediate word that resolves a backward unconditional branch.
UNTIL	This is a immediate word that resolves a backward conditional branch.
CS-ROLL	This is an immediate word that reorders the control stack.
CS-PICK	This is an immediate word that copies an item on the control stack.

Figure 1: *The abstracted Forth syntax, as used in this paper.* Here we choose to use an abstraction to represent the usual data stack manipulations, allowing us to concentrate on Forth's control words.

The user can extend the Forth dictionary by defining new words. When a : is encountered, a new word is being defined, and the Forth systems switches from *interpreter mode* into *compiler mode* until the ending ; or ; IMMEDIATE is encountered. The new Forth word is defined in terms of existing Forth words, which are either immediate words or non-immediate words. Immediate words are executed immediately by the compiler whereas non-immediate words have their compiled code appended to the compiled code of the Forth word being defined. It is only when the non-immediate word is subsequently executed by the *Interpreter* that the behaviour defined by the non-immediate word is executed. An immediate word may be POSTPONEd, causing it to be treated as a non-immediate word.

As well as primitives for defining new words, we have included the six immediate control words IF, THEN, BEGIN, AGAIN, UNTIL and AHEAD. These control the generation of branch and labelled instructions by maintaining all labels in the program on the control flow stack. Each of these instructions corresponds to either generating a new label or using an existing label from the control flow stack. The primitives CS-ROLL and CS-PICK are used to change the ordering on the control flow stack without generating any new labels.

Figure 2 gives an overview of our specification of the Forth programming system. It shows the processes that model each component of the Forth system and the channels that carry information between the processes, and should thus be used as a reference when reading the π-calculus specification in sections 3 and 4. Section 5 will describe the interactions between these processes as the Forth compiler goes through selection and iteration constructs.

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

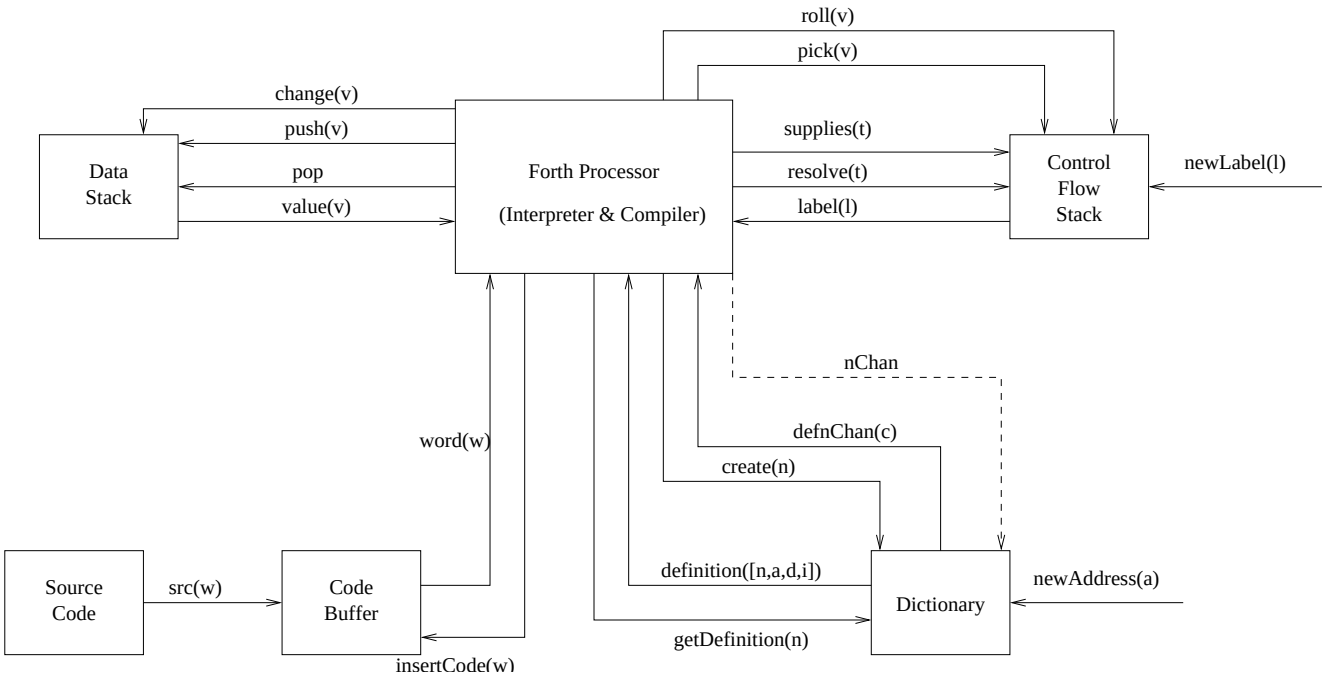

Figure 2: *Overview of the specification of the Forth system.* The processes and events depicted here are described by the specifications in sections 3 and 4.

$$FORTH \stackrel{\texttt{def}}{=} INTERP \mid DICTIONARY(d) \mid CBUF(src, Nil) \mid DATA(Nil, 0) \mid CFS(Nil) \mid genLabel(1) \mid genAddress(1)$$

Figure 3: *Specification of the main processes of the Forth system.* This is the starting point for the processing of some Forth program, available on the stream *src*, based on an initial dictionary d of pre-defined words.

---

3 Processing a Forth Program

Figure 3 is our top level specification of the Forth programming system. The system is the parallel composition of the *Interpreter/Compiler* process, the *Dictionary* process, the *Source Code* process, the *Source Code Buffer* process and the two main stack processes - the *Data Stack* and the *Control Flow Stack*.

In the remainder of this section we describe those processes within the Forth system that interact with the Forth source code, specifically the dictionary, code buffer and the data stack. We do not deal with Forth's return stack further here; its specification is similar to that of the data stack, and it does not play a significant role in the rest of our specification.

3.1 The Forth Dictionary

We start the specification with our description of the Forth dictionary, used to hold word definitions, and defined in figure 4 as the *Dictionary* process. The two primary processes of the

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

$$
\begin{array}{l}
DICTIONARY(w) \stackrel{\texttt{def}}{=} \\
\quad create(n).newAddress(a). \\
\quad\quad \text{new } nChan\ (DICTADD(nChan, [n, a, Nil, ord], w) \mid \overline{defnChan}\langle nChan\rangle) \\
\quad + \quad getDefinition(n).\overline{definition}\langle(find\ n\ w)\rangle.DICTIONARY(w)
\end{array}
$$

$$
\begin{array}{l}
DICTADD(nChan, [n, a, d, i], w) \stackrel{\texttt{def}}{=} \\
\quad nChan(s). \\
\quad\quad \text{if } s = done \text{ then } \quad DICTIONARY([n, a, d, i]^\wedge w) \\
\quad\quad \text{else if } s = doneimm \text{ then } \quad DICTIONARY([n, a, d, imm]^\wedge w) \\
\quad\quad\quad \text{else} \quad DICTADD(nChan, [n, a, d^\wedge s, i], w)
\end{array}
$$

$$
\begin{array}{l}
genAddress(n) \stackrel{\texttt{def}}{=} \\
\quad \overline{newAddress}\langle n\rangle.genAddress(n+1)
\end{array}
$$

Figure 4: *Specification of the Forth dictionary.* This specification describes the dictionary itself, along with the maintenance processes of accessing a dictionary entry, and adding to the dictionary. Here the function $find$ returns a dictionary entry for a given word n.

Dictionary are to add a new word to the *Dictionary* or report on an existing word. Reporting on an word n involves searching the dictionary and returning the address a of its compiled code, its definition d, and its immediacy status i. When adding a new word to the *Dictionary* a new address is generated along *newAddress* and a new channel *nChan* is created. This cannel can then be used by the the *Compiler* to supply words from the definition to the *Dictionary*. When the definition is completed then the word n is marked as either *imm* or *ord*, depending on whether the definition is an ordinary word terminated with ; or an immediate word terminated with ; IMMEDIATE, and the channel is released.

3.2 Source Code

The compilation and interpretation processes in Forth are both driven by the source code of the program being processed, in the form of a stream of Forth words. In our specification we introduce a buffer *CBUF* between the Forth source code stream *src* and the processor *INTERP*. Typically a word is read from the source code *src*, and, via *CBUF*, delivered to the interpreter *INTERP* via the *word* channel.

The introduction of the code buffer provides for modification of the input stream, which can occur when the compiler processes an immediate word, and is achieved using the *insertCode* channel. Figure 5 specifies this process.

3.3 The Data Stack

Figure 6 specifies the *Data Stack* process. Since the data stack is not the primary focus of this paper, we have abstracted considerably from the actual details of its operation. In particular, we model this stack as consisting of some specified size *size*, along with the stack contents, a list of *values*. Values can be added to the front of the list via the *push* channel or removed from the front of the list in response to a *pop* signal. The *change* channel allows us to abstract the

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

$$
\begin{aligned}
&CBUF(src, buf) \stackrel{\texttt{def}}{=} \\
&\qquad \text{if } buf = Nil \text{ then } \quad src(w).\overline{word}\langle w\rangle.CBUF(src, buf) \\
&\qquad\qquad\qquad \text{else } \quad \overline{word}\langle hd\ buf\rangle.CBUF(src, (tl\ buf)) \\
&\quad + \quad insertCode(w).CBUF(src, buf^{\wedge}w)
\end{aligned}
$$

Figure 5: *Specification of the Forth source code buffer.* Here the stream *src* contains the Forth program as a series of words, which are then buffered by CBUF before being fed to the program processor.

$$
\begin{aligned}
&DATA(values, size) \stackrel{\texttt{def}}{=} \\
&\qquad push(v).DATA([v^{\wedge}values, size + 1]) \\
&\quad + \quad pop().\overline{value}\langle hd\ values\rangle.DATA((tl\ values), size - 1) \\
&\quad + \quad change(n).DATA(Nil, size + n)
\end{aligned}
$$

Figure 6: *Specification of the Forth data stack.* We have abstracted considerably here from the full details of its operation.

---

behaviour of Forth words that manipulate the *Data Stack*. Forth words that increase the size of the *Data Stack*, such as DUP, PICK and DROP, can be abstracted as *change(1)* and *change(-1)*. Forth words that manipulate the *Data Stack* without changing its size, such as ROLL, can be abstracted as *change(0)*.

4 Control Structures

In this section we turn to the main focus of our paper, the formal specification of Forth control words. In particular we specify the compile-time activities of the Forth processor, mainly centered around label management via the control stack, and code generation. To describe the generated code we use a simple pseudo-assembly language consisting mainly of labels and jumps.

4.1 The Compiler

Figure 7 specifies the *Compiler* process. When the *Interpreter* reads a word definition, which starts : n, it creates an entry in the *Dictionary* for the word n and invokes the *Compiler* with the supplied channel *nChan*. The *Compiler* reads successive words from the code buffer *CBUF* until it reads either ; or ; IMMEDIATE, relays the appropriate signal to the Dictionary, and then returns to the top-level *Interpreter/Compiler* process.

If, during compilation, the *Compiler* reads the word POSTPONE it reads the next word m from the source code buffer, retrieves m's definition and sends that definition along channel *nChan* to the *Dictionary*. If the *Compiler* reads a numeric literal from the source code buffer, it sends some code along the channel *nChan* that will push this number onto the data stack when the word n is executed. If the *Compiler* reads any other word it enters the *COMP* process. The *COMP* process will be described after the *Control Flow Stack* is specified since it relies on the *Control Flow Stack* to generate the structured control flow.

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

$$
\begin{array}{l}
COMPILE(nChan) \stackrel{\texttt{def}}{=} \\
\quad word(w). \\
\qquad \text{if } w = \text{“POSTPONE”} \text{ then } \; word(m).\overline{nChan}\langle m\rangle.COMPILE(nChan) \\
\qquad\quad \text{else if } w = \text{“;”} \text{ then } \; \overline{nChan}\langle done\rangle.INTERP \\
\qquad \text{else if } w = \text{“; IMMEDIATE”} \text{ then } \; \overline{nChan}\langle doneimm\rangle.INTERP \\
\qquad\quad \text{else if } isNumber(w) \text{ then } \; \overline{nChan}\langle \texttt{"push w"}\rangle.COMPILE(nChan) \\
\qquad\qquad\quad \text{else } \; COMP(nChan, w); COMPILE(nChan)
\end{array}
$$

Figure 7: *Specification of the main actions of the Forth compiler.* Here we assume a boolean-valued function *isNumber* to test if a word is a numeric literal. The immediate words are dealt with by the process COMP, described later.

$$
\begin{array}{l}
CFS([l_0,t_0]^\wedge[l_1,t_1]^\wedge\cdots^\wedge[l_n,t_n]) \stackrel{\texttt{def}}{=} \\
\quad resolve(t_0).\overline{label}\langle l_0\rangle.CFS([l_1,t_1]^\wedge\cdots^\wedge[l_n,t_n]) \\
+ \quad supplies(t').newLabel(l').\overline{label}\langle l'\rangle.CFS([l',t']^\wedge[l_0,t_0]^\wedge[l_1,t_1]^\wedge\cdots^\wedge[l_n,t_n]) \\
+ \quad pick(k).CFS([l_k,t_k]^\wedge[l_0,t_0]^\wedge[l_1,t_1]^\wedge\cdots^\wedge[l_n,t_n]) \\
+ \quad roll(k).CFS([l_k,t_k]^\wedge[l_0,t_0]^\wedge[l_1,t_1]^\wedge\cdots[l_{k-1},t_{k-1}]^\wedge[l_{k+1},t_{k+1}]^\wedge\cdots^\wedge[l_n,t_n])
\end{array}
$$

$$
\begin{array}{l}
genLabel(n) \stackrel{\texttt{def}}{=} \\
\quad \overline{newLabel}\langle n\rangle.genLabel(n+1)
\end{array}
$$

Figure 8: *Specification of the Forth control stack.* This stack is used for label management, and maintained by the compiler as it processes immediate words.

4.2 The Control Flow Stack

The *Control Flow Stack* stores [*label*,*type*] pairs that are used by the control flow words IF, THEN, BEGIN, AGAIN, UNTIL and AHEAD, as described in figure 1. The valid label types are *orig* and *dest*. Pairs are added to the *Control Flow Stack* via the *supplies* channel, which also causes the creation of a new label along *newLabel*. A pair is removed from the *Control Flow Stack* when the value on the *resolve* channel matched the *type* of the pair on top of the *Control Flow Stack*. In each case, the *label* value of the pair is transmitted by the *Control Flow Stack* to the *Compiler* via the *label* channel. The *pick* and *roll* channels are used to modify the *Control Flow Stack* in response to CS-PICK and CS-ROLL instructions.

4.3 Compiling a New Word

Figure 9 specifies the processing of a word w, compiling it, and sending the results along *nChan*. If the word w is not one of the built-in control words, then the *Dictionary* is searched for the an entry for w. If w is a non-immediate word a subroutine call (represented as `"jsr"`) to the compiled code for w is generated and added to the n's entry in the *Dictionary*. If w is an immediate word its definition is retrieved from the *Dictionary* and inserted into the stream of words from the *Code Buffer* via the *insertCode* channel.

When the *COMP* process receives the immediate Forth words IF, THEN, BEGIN, AGAIN, UNTIL or AHEAD it performs the corresponding actions on the *Control Flow Stack*, and generates either

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

$$
\begin{array}{rl}
COMP(nChan, w) \stackrel{\texttt{def}}{=} & \\
\text{if } w = \textsc{if} \text{ then} & \overline{supplies}\langle orig\rangle.label(l).\overline{nChan}\langle\texttt{"ifzero l"}\rangle \\
\text{else if } w = \textsc{then} \text{ then} & \overline{resolve}\langle orig\rangle.label(l).\overline{nChan}\langle\texttt{"l:"}\rangle \\
\text{else if } w = \textsc{begin} \text{ then} & \overline{supplies}\langle dest\rangle.label(l).\overline{nChan}\langle\texttt{"l:"}\rangle \\
\text{else if } w = \textsc{again} \text{ then} & \overline{resolve}\langle dest\rangle.label(l).\overline{nChan}\langle\texttt{"goto l"}\rangle \\
\text{else if } w = \textsc{until} \text{ then} & \overline{resolve}\langle dest\rangle.label(l).\overline{nChan}\langle\texttt{"ifzero l"}\rangle \\
\text{else if } w = \textsc{ahead} \text{ then} & \overline{supplies}\langle orig\rangle.label(l).\overline{nChan}\langle\texttt{"goto l"}\rangle \\
\text{else} & \overline{getDefinition}\langle w\rangle.definition([w, a, d, i]). \\
& \quad \text{if } (i = ord) \text{ then } \overline{nChan}\langle\texttt{"jsr a"}\rangle \\
& \quad \text{else } \overline{insertCode}\langle d\rangle
\end{array}
$$

Figure 9: *Specification of the execution of Forth immediate words.* This specification should be considered in conjunction with that of the control-flow stack, since its action mainly involve accessing the labels on that stack.

$$
\begin{array}{l}
INTERP \stackrel{\texttt{def}}{=} \\
\quad word(w). \\
\quad \text{if } w = \text{":" then} \quad word(n).\overline{create}\langle n\rangle.defnChan(nChan).COMPILE(nChan) \\
\qquad\qquad \text{else} \quad \tau.INTERP
\end{array}
$$

Figure 10: *Specification of the Forth interpreter.* As before, we have abstracted away any details not dealing directly with word compilation.

---

a label, conditional jump or unconditional jump as appropriate.

For example, in the case of the conditional forward branch (IF) a request for a label of type *orig* is sent to the *Control Flow Stack* and the label l is then received along *label*. The conditional branch `"ifzero l"` is sent along the code channel. A subsequent THEN word issues a $\overline{resolve}\langle orig\rangle$ to the *Control Flow Stack* and receives a label l. The label `"l:"` will then be appended to the compiled code.

4.4 The Interpreter

The Forth *Interpreter* reads words from the source code, retrieves the compiled code associated with the Forth word from the *Dictionary*, executes the compiled code and modifies the *Data Stack*. We choose not to fully specify the actions of the interpreter here, except to note that when the *Interpreter* reads : *wordname* it issues a command to the *Dictionary* to create an entry for *wordname* and switches into *compiler mode*.

5 Some Examples of Compiling Forth Control Words

In this section we show how the standard if-else selection construct and the while looping construct found in most high-level programming languages, can be implemented in Forth. The approach here should be contrasted with high-level languages with built-in nested control structures, as well as with assembly languages that must rely on unstructured test and branch instructions.

Accepted for the 6th International Workshop on Formal Methods for Industrial Critical Systems
Paris, France, July 16-17, 2001

We have simulated the operation of the Forth compiler over these and other examples through a translation of the π-calculus specification into the Pict programming language.

5.1 The if-else Statement

Forth's built-in control words can already handle the usual if-statement selection, using IF to test a condition, and THEN to mark the end of the conditionally-executed block. As is not uncommon in Forth programming, we shall use the more intuitive word ENDIF as a synonym for THEN.

One of the unique aspects of Forth programming is the possibility of creating new control structures via the definition of additional control words. In particular, we can enhance the IF-ENDIF structure by adding an ELSE, mimicking the usual if-else statements in high-level languages.

The standard definition for ELSE is as follows:

```
: ELSE  ( l1 -- l2 / -- )
  POSTPONE AHEAD
  1 CS-ROLL
  POSTPONE THEN
; IMMEDIATE
```

To demonstrate the use of ELSE and its interaction with the built-in control words, we consider the following example Forth code, along with its (rough) equivalent in a C-like language, and its translation into a pseudo assembly language as described by our specification.

Forth Code	**C equivalent**	**Translation**

: example1
cond
IF $stats_1$
ELSE $stats_2$
ENDIF
;
	void example1()
	{
	if (*cond*)
	$stats_1$
	else
	$stats_2$
	}
		example1 =
		jsr *cond*
		ifzero L1
		jsr $stats_1$
		goto L2
		L1 :
		jsr $stats_2$
		L2 :

When this is executed, *cond* is evaluated, leaving a flag on the stack, and then either $stats_1$ or $stats_2$ is executed depending on whether the flag is true or false respectively.

To show that this definition does indeed produce a stack-safe program, in figure 11 we show the actions of the Forth compiler as it processes the definition of `example1`. We can see that when the word ELSE is executed, the IF has already created a conditional jump to some as yet undefined label `L1`. The postponed word AHEAD will create another jump, generating a new label `L2` and placing it on the stack. The CS-ROLL word changes the order of these labels, and thus it is the target if the IF jump that is resolved by the postponed THEN.

A number of other selection patterns can be constructed using Forth words; for example appendix A.3 of [6] defines four other Forth words, CASE, OF, ENDCASE and ENDOF , which provide a functionality similar to the `switch` statement in C++ or Java.

Event	Control Flow Stack	Output
word(:)		
word(EXAMPLE1)		
word(*cond*)		`"jsr` *cond*`"`
word(IF)	$[L1, orig]$	`"ifzero L1"`
word($stats_1$)	$[L1, orig]$	`"jsr` $stats_1$`"`
word(ELSE)	$[L1, orig]$	
word(`"ahead"`)	$[L1, orig][L2, orig]$	`"goto L2"`
word(`"1 csroll"`)	$[L2, orig][L1, orig]$	
word(`"then"`)	$[L2, orig]$	`"L1:"`
word($stats_2$)	$[L2, orig]$	`"jsr` $stats_2$`"`
word(THEN)		`"L2:"`

Figure 11: *Compiling the definition of program* `example1`.

5.2 The while Loop

To define the while loop in Forth, we follow [6] and define two extra control words, WHILE and REPEAT.

```
: WHILE  ( dest -- orig dest / flag -- )
    POSTPONE IF
    1 CS-ROLL
; IMMEDIATE


: REPEAT  ( orig dest -- / -- )
    POSTPONE AGAIN
    POSTPONE THEN
; IMMEDIATE
```

The semantics of these words can best be explained in the context of the following example:

Forth Code	**C equivalent**	**Translation**
		example2 =
: example2	void example2()	jsr *init*
init	{	L3 :
BEGIN *cond*	*init*	jsr *cond*
WHILE *stats*	while (*cond*)	ifzero L4
REPEAT	*stats*	jsr *stats*
;	}	goto L3
		L4 :

In this program, *init* is executed before the loop, the Forth code *cond* represents the evaluation of the condition, and the code *stats* represents the statements in the loop body. Thus, BEGIN

Event	Control Flow Stack	Output
$word$(:)		
$word$(EXAMPLE2)		
$word(init)$		`"jsr` *init*`"`
$word$(BEGIN)	$[L3, dest]$	`"L3:"`
$word(cond)$	$[L3, dest]$	`"jsr` *cond*`"`
$word$(WHILE)	$[L3, dest]$	
$word$(`"if"`)	$[L3, dest][L4, orig]$	`"ifzero L4"`
$word$(`"1 csroll"`)	$[L4, orig][L3, dest]$	
$word(stats)$	$[L4, orig][L3, dest]$	`"jsr` *stats*`"`
$word$(REPEAT)	$[L4, orig][L3, dest]$	
$word$(`"again"`)	$[L4, orig]$	`"goto L3"`
$word$(`"then"`)		`"L4:"`

Figure 12: *Compiling the definition of program* `example2`.

marks the start of the loop, WHILE must be a conditional-branch depending on the truth of *cond*, and REPEAT must represent both an unconditional branch back to the start of the loop, as well as mark the end of the loop.

To show that this combination of words does indeed represent the familiar while loop control structure, figure 12 shows the translation of the definition of `example2`. We can see from this example how WHILE uses the IF control word to test the condition, and how REPEAT generates both an unconditional jump back to the label pushed by BEGIN, as well as a target for the conditional jump generated by WHILE.

As mentioned earlier, Forth programmers are not restricted to a predefined set of control structures, but are free to devise new combinations of the control words to construct more exotic control patterns. Appendix A.3 of [6] gives some examples of Forth versions of common and not-so-common control patterns.

6 Conclusions and Further Work

In this paper we have presented a formal specification of aspects of the Forth programming language, in particular the processes involved in the compilation of Forth control words. We see the contribution of this work falling into three main categories:

- *The formal specification of aspects of the Forth system.* As a stack-based machine, Forth exhibits many features similar to other stack-based languages, a technology crucial to embedded systems. We hope that a formal study of Forth constructs can act as a foundation for the study and comparison of such stack-based languages, and contribute to the verification of their safety properties.

- *A formal explication of Forth control structures.* The compilation of Forth control words, being a mixture of syntactically simple, but nonetheless structured constructs, is unique to Forth. The study of such structures can contribute to the general field of research

surrounding the user of low-level control structures that facilitate the data-flow analysis process, central to, for example, bytecode verification in the JVM.

- *The use of the π-calculus in programming language specification.* As mentioned in section 1, this approach gives a different perspective to traditional compositional semantics. Such a specification is particularly suited to languages with little syntactic structure that have to interact with a number of different data structures or devices.

We intend to extend the specification to cover some of the aspects of the Forth system not covered above, most notably the data stack. We have already type-checked and simulated the operation of the specification using the Pict system. The next step is to translate our specification into a format suitable for a proof assistant such as Coq [3] or Isabelle [10], so that it could be used as a basis for the formal verification of safety properties of Forth programs.

References

[1] A.V. Aho and R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] M. Barr and B. Frank, *Java: Too Much for Your System?*, Embedded Systems Programming, May 1997.

[3] C. Cornes et al., *The Coq proof assistant reference manual*, Rapport Technique 177, INRIA, July 1995.

[4] M. Franz, Open Standards Beyond Java: On the Future of Mobile Code for the Internet, *Journal of Universal Computer Science*, 4:5, pp. 521-532, May 1998.

[5] Matthew Hennessy, *The Semantics of Programming Languages*, Wiley, 1990.

[6] ISO/IEC 15145:1997 *Information technology - Programming languages - Forth*, International Standards Organisation, 1997.

[7] Peter J. Knaggs, *Practical and Theoretical Aspects of Forth Software Development*, Ph.D. thesis, University of Teesside, March 1993.

[8] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1999.

[9] Robin Milner, *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, 1999.

[10] L.C. Paulson, *Isabelle: A Generic Theorem Prover*, Springer-Verlag LNCS 828, 1994.

[11] Benjamin C. Pierce and David N. Turner, *Pict: a programming language based on the π-calculus*, Technical Report, Computer Science Department, Indiana University, 1997.

[12] James Power and David Sinclair, *A Formal Model of Forth Control Words in the Pi-Calculus - and its animation in Pict*, Technical Report NUIM-CS-TR-2001-03, Dept. of Computer Science, National University of Ireland, Maynooth, February 2001.

[13] C. Röckl and D. Sangiorgo, *A π-calculus Process Semantics of Concurrent Idealised ALGOL*, Second International Conference on the Foundations of Software Science and Computation Structure, pp. 306-321, Amsterdam, The Netherlands, 22-28 March, 1999.

[14] David A. Schmidt, *Denotational Semantics: a methodology for language development*, Allyn and Bacon, 1986.

[15] Bill Stoddart, *An Event Calculus Model of the Forth Programming System*, The 12th euroFORTH conference on the FORTH programming language and FORTH processors, St. Petersburg, Russia, 4-6 October, 1996.

[16] David A. Watt, *Programming Language Syntax and Semantics*, Prentice-Hall, 1991.