Small Journal Name, 9, 275–294 (1992)

Predictability of a RTX2000-based Implementation

CHARLES ANDRÉ AND MARIE-AGNÈS PÉRALDI {andre,map}@unice.fr
Laboratoire Informatique, Signaux, Systèmes (I3S)
Université de Nice-Sophia Antipolis / CNRS
41 bd Napoléon III
F 06041 Nice cedex, France

Abstract. Any implementation should be proved to meet its specification. In this paper, we cope with the issue of checking the temporal correctness of a real-time program, implemented on a RTX2000 microcontroller. Our real-time programs are written in a high-level synchronous language (ESTEREL), and then, automatically translated into T-FORTH. Under these assumptions, the temporal behaviour of the generated RTX2000 program can be predicted, at compile-time. In this paper, we present algorithms able to compute realistic duration of system reactions. These algorithms use an abstract representation of the RTX2000 program based on a behavioural semantics given to T-FORTH.

Keywords: T-FORTH, RTX2000, ESTEREL, predictability, temporal correctness, behavioural semantics.

1. Introduction

Real-Time controllers are often used in critical environment, so they must perform their reactions to incoming requests without failure and on time. More generally, real-time applications require

- dependability
- predictability
- safety.

A modern approach to the design of real-time system must provide high-level "real-time" programming languages, safe and efficient compilers, and *verification tools*. Verification tools are necessary to prove that an implementation meets its specification. Both *the logical correctness* and *the temporal correctness* must be established. The semantics of the programming language and the quality of its compiler are determining factors in the logical correctness which is highly implementation-independent. On the contrary, the temporal correctness is implementation-dependent: the actual performances of the processor must be taken into account.

Recently, a book has been dedicated to *predictability* [1]. To guarantee predictable behaviour is a difficult task because it is a matter of both logical and temporal correctness. In this paper, we address the issue of predictability for a restricted class of real-time system: Our applications are *almost pure reactive systems* (mostly control, few data-processing) and our target processor is a *microcontroller*.

The microcontroller we use is the Harris'RTX2000 [2]. It is a fast stack-oriented processor whose programming language is FORTH (more precisely a dialect: T-FORTH). We adopt the *synchronous approach to real-time programming*. This new approach has been advocated in an issue of the "proceedings of the IEEE" [3]. The introductory paper [4] of this issue explains why the synchronous approach improves real-time programming. Facilities for proving the logical correctness of synchronous programs are presented, too. Among synchronous languages, we have chosen ESTEREL [5], an imperative synchronous language particularly suitable for programming event-driven applications. The implementation of an ESTEREL program on the processor is carried out by an automatic translator from ESTEREL to FORTH. Further details about the method of translation can be found in [6]. In this paper we restrict our attention to one of the requirement of real-time system: the predictability. So, we explain *how and why* it is possible *to guarantee the temporal behaviour of an implementation running on RTX2000 and programmed in ESTEREL*. The intermediate language (FORTH) plays a central role in the verification process.

Since ESTEREL has a formal semantics, the logical correctness of a program can be proved using tools of the ESTEREL environment (Auto, Xsimul [5]). We strive to preserve this correctness down to the target processor by an *automatic code generation*. In order to prove the correctness of the translation, the target language (i.e., T-FORTH) must be also endowed with a formal semantics. So, we propose *a behavioural semantics* for an *abstract FORTH machine*.

Besides the logical correctness, the temporal correctness must be checked, too. To ensure that a Real-time application acts in a predictable timely manner, the program must be analyzed, at compile-time, to determine whether or not it will meet all deadlines. This analysis is based on the proposed semantics. Algorithms are given that compute the duration of each transition (i.e., a reaction to a stimulus).

In order to explain these different aspects, we have organized this paper as follows:

- First, we give an overview of the method from the specification down to the verification of constraints. We present the steps of the compilation and we sketch the principles of our run-time system.
- Then, we point out the characteristics of our implementations. We explain *why* the predictability may result from a conjunction of several factors (software and hardware) and *how* to evaluate the duration.
- The third part is more formal. We introduce an abstract view of the RTX2000 processor and we provide T-FORTH with a behavioural semantics. The actual program is transformed into a semantically equivalent "lumped program". A symbolic execution is then performed, in order to determine realistic reaction time.
- Finally, results are presented and commented.

2. An overview of the method

2.1. Compilation line

We briefly present how to generate predictable code on the RTX2000. Fig. 1 shows the main steps of the method. The ESTEREL compiler yields a file including several tables (for inputs, outputs, actions, ...) and an automaton. The contents of this file, called the oc-file, is semantically equivalent to the source program. Dedicated software which works on the oc-file, allows to test the *logical correctness* of the program [5]. The oc-file is not executable: An *execution machine* must be built. Execution machines are described in [7]. The actual executable code is derived from the oc-file; the target language may be any sequential general purpose language. For RTX2000, we have chosen T-FORTH. The translator oc2forth [6] makes this generation. Because of the possible optimizations made by the T-FORTH assembler, we have to disassemble the object code. On the disassembled code we can evaluate the *actual performance of the implementation* by applying techniques explained in section 4.

Figure 1. Main step of the method

2.2. Execution on RTX2000

Details about the execution machine running on RTX2000 are available in [6].

To be executed, an ESTEREL program has to be nested in a so-called "execution machine" that controls the automaton. The oc2forth translator generates an execution machine, automatically. This execution machine is composed of application-dependent subroutines and some invariant run-time routines.

A reaction of the execution machine proceeds in successive steps:

1. the Acquisition phase,
2. the Activation phase,
3. the Cleaning phase.

During the acquisition phase the execution machine checks the presence and the value of signals and sensors. The physical state of the environment is reported and memorized in data structures manipulated by the automaton during its transition. Physical events are connected to physical input ports of the RTX2000 via the asic bus. A flag (boolean variable asserted iff the signal is present) and a value (if any) are associated with each input signal. Updating routines are provided by the ESTEREL compiler; but the effective calls are made by the execution machine (polling or interrupt).

The second phase is the activation of the automaton. This execution is supported by a special run-time routine called the "engine". Given the status of each signal, and the current state of the automaton, the engine determines *the* transition to fire (strictly deterministic behaviour). The firing of a transition is a sequence of actions, followed by a change of state. Instants of activations are chosen by the execution machine. The simplest solution is a periodic activation, but, in this case, two conditions must hold: 1) the period must be small with respect to the shortest time constant of the system to be controlled, 2) the period must be proven longer than the longest transition duration. These are typical timing constraints not to be violated.

The last phase is the cleaning phase. Contrary to its value, the presence of a signal is not a remanent information. So, at the end of a transition all the presence flags must be cleared. Dedicated run-time routines must be provided.

3. Characteristics of the implementation

Both the hardware and the software participate to the predictability of our implementation. The processor has been selected for its efficiency and predictable behaviour. The software has been tuned to preserve these qualities.

3.1. The processor

The RTX2000 [2] is a high performance and autonomous microcontroller dedicated to the design of embedded real-time applications. Because of the potential of the architecture (parallelism, Asic buses, hardwired instructions), outstanding performances can be achieved. As a risc architecture, the RTX2000 runs fast but it has neither pipeline nor cache memory so that its behaviour is predictable. Some interesting features deserve to be mentioned. A subroutine call takes 1 clock cycle, memory access takes 2 cycles and interrupt latency is 4 clock cycles. A clock cycle on the processor is 100 ns. RTX2000 is a stack-based architecture, the chip has a parameter stack for manipulation of data, memory access and multitasking management, and a return stack for subroutine calls. The environment for designing RTX2000 applications consists of a FORTH environment running on a PC, and a dedicated development system RTXDS providing interactive debugging.

3.2. The software

The source language is ESTEREL [5]. The code generation for RTX2000 is generated by a sequence of transformations. The last one (the disassembling) is appended for verification purpose. It is an essential phase for proving the correctness of our implementation. The main steps are:

1. a compilation (from ESTEREL to oc),
2. a translation (from oc to T-FORTH),
3. a compilation (from T-FORTH to RTX2000 code), and
4. a disassembling (from RTX2000 code to the code to be analyzed).

We have very little influence on code production during phases 3 and 4. Phase 1 is a bit more versatile (directives given to the ESTEREL compiler). On the contrary, phase 2 offers a great opportunity to produce fully mastered programs.

The oc code

Oc is the object code given by ESTEREL compilation. The oc code is made of a series of tables that contain information associated with the input and output signals defined in the ESTEREL program, and the automaton. The ESTEREL compiler generates purely sequential and linear code: the execution of a reaction in oc is free of iteration or recursion.

After the compilation phase, the oc code is translated into the actual code for the RTX2000 (T-FORTH code).

The T-Forth language

T-FORTH is a language based on the notion of WORDS. A WORD can be seen as a procedure. While a FORTH program is running, words are calling to each other. After lexical and syntactic analyses, input and output functions, tables and the automaton are translated into FORTH words and data structures. Besides this code which is application-dependent, there are some *management words* that control the execution of the automaton and the access to data. Management words are invariant code and they constitute the heart of the *execution machine*. The management word "`engine`" executes reactions (Acquisition, Activation and Cleaning) (cf.2.2). `engine` calls application-dependent words via the FORTH instruction "`EXECUTE`" which allows vectored executions.

In order to respect the synchrony hypothesis, the execution machine must run the automaton in an indivisible way. Moreover, real-time constraints require fast enough executions. Quoting Charles H. Moore [1], the FORTH language has been created for "providing the power, ease and flexibility" not found in traditional languages. T-FORTH can produce efficient code and is used in many real-time applications.

Since our T-FORTH programs are derived from ESTEREL programs, their control structure is very simple. The absence of recursion and iteration in the oc code is preserved by the translation. Some management words (e.g., initialization and polling) introduce loops but the number of iterations is known at compile-time. All those features meet classical demands for real-time programming.

The RTX2000 code

The T-FORTH compiler generates optimized RTX2000 code making the best of the parallelism of the architecture. Up to 3 T-FORTH instructions can be combined into one RTX2000 instruction leading to a more efficient code. Effective optimizations can be seen on the disassembled code.

The RTX2000 disassembled code

The listing of the disassembled program contains information pertinent to the evaluation of performance. A line associated with an instruction has a fixed format: an address, an opcode, and a comment field giving the mnemonic and the operand in a symbolic form. T-FORTH words used by the program are easily identified; the code of each word can be isolated as a block of instructions.

3.3. Evaluation of the duration

3.3.1. Duration of one execution

Because of the very nature of the RTX2000 architecture (cf. 3.1), we know the exact duration of every instruction of the processor. The programming unit in FORTH is the WORD. Given an initial state, the execution of a word is a fully deterministic sequence of instructions. The duration of this sequence is obtained by adding up the duration of elementary instructions. Loops do not raise problems because they are special "natural loops" [8] executed a fixed number of times. Thus, iterations have fully predictable behaviour. Since a reaction is a sequence of words, its duration can be evaluated, even if it may entail heavy computation.

3.3.2. Symbolic computation

A reaction is extremely sensitive to input data. Its duration may widely fluctuate according to the stimuli. To guarantee real-time behaviour, the worst case for any reaction must be predicted.

Therefore, we have to compute the duration of each reaction for all input scenarios. The complexity of this analysis is exponential in time. A more sensible approach is to give realistic bounds on duration for reactions. This can be carried out by a "symbolic computation of the program". Basically, we explore all control paths of the program, not taking account of actual data. This analysis is tractable at a reasonable cost because only few control structures are used in our programs (sequence, branching, conditional branching, word calls, fixed loops).

The main steps of the program analysis are:

1. By inspection of the disassembled code:
    - extract words,
    - detect loops and create a pseudo-word for each,
    - construct dependence graph between words (caller/callee)
2. Apply a topological sorting on the dependence graph
3. Compute the symbolic duration of words according to the topological order
4. Deduce bounds on duration of reactions.

The third step requires a precise and abstract representation of the RTX2000 instructions. Next section is devoted to a formal presentation of executions.

4. Semantics of RTX programs

RTX2000 has been designed to execute FORTH (T-FORTH) very efficiently. One, two, or even three elementary FORTH's instructions can be executed in a single clock cycle of the processor. For clarity, when a RTX2000's instruction is equivalent to a sequence of FORTH's instructions, the RTX2000's instruction is denoted by the associated sequence of FORTH's instructions (e.g., `R> DROP`).

4.1. The Abstract Machine

RTX2000 has only one memory space, but generally composed of ROMs and RAMs. At the abstract level, we consider that the abstract machine has two separate memories (the instruction memory and the data memory). All operations involve values on a stack. A program is an ordered set of instructions. An instruction is composed of one operation and possibly one operand. Each instruction is assigned a location in the program memory. A register (the *Program Counter*) contains the address of the instruction to be executed. Besides its two stacks and its program counter, RTX2000 has other internal registers. These registers do not play an essential role in our applications. No explicit reference to these registers will be made in what follows. So, we consider that these registers are parts of the data memory.

A stack-oriented abstract machine is a good approximation to the processor RTX2000.

Definition. The **Abstract Machine** is a 5-tuple:
$\mathcal{M}$=<$\mathcal{DM}$, $\mathcal{PS}$, $\mathcal{RS}$, $\mathcal{PM}$, pc> where

- $\mathcal{DM}$ is the Data Memory,
- $\mathcal{PM}$ is the Program Memory,
- $\mathcal{PS}$ is the Parameter Stack,
- $\mathcal{RS}$ is the Return Stack,
- pc is the Program Counter.

Although the size of items in $\mathcal{PS}$ and $\mathcal{RS}$ are different in RTX2000, we assume, at the abstract level that all the data are from a domain "**Value**" (including a least element $\perp$ standing for the undefined value). A value may be either a data for arithmetic or logical manipulation, or the address of an instruction.

In RTX2000, the addresses of instructions are even program memory addresses (0, 2, 4, ...). At the abstract level, addresses are denoted by α, β, γ, ... Let **Addr** be the associated domain, where $\perp$ is the undefined address. Let **Oper** be the domain of the operations ($\perp$ stands for the undefined operation).

Definition. The domain of the instructions "**Inst**" is such that
Inst= (**Addr**×**Oper**) ∪ (**Addr**×**Oper**×**Operand**)
where **Operand**=**Addr**∪**Literal**.

In fact, **Addr** and **Literal** are integers.

Definition. @ : **Inst** → **Addr** yields the address field of an instruction.

Definition. A **state** of the abstract machine $\mathcal{M}$ is a couple $< \delta, \eta >$ where

- $\delta =< \mu, \pi, \rho >$ is the actual data composed of
  - μ :$\mathcal{DM}$→ **Value** (contents of the Data Memory),
  - $\pi \in$ **Value*** (contents of the Parameter Stack),
  - $\rho \in$ **Value*** (contents of the Return Stack),
- $\eta \in$ **Addr** is the actual content of pc.

Definition. A **program** of $\mathcal{M}$ is a 3-tuple $\Pi =< \varpi, \eta_0, \mu_0 >$ where

- $\varpi : \mathcal{PM} \rightarrow$ **Inst** (contents of the Program Memory),
- $\eta_0 \in$ **Addr** is the entry point of the program,
- $\mu_0 : \mathcal{DM} \rightarrow$ **Value** (initial contents of the Data Memory).

The *initial state* of $\mathcal{M}$ for a program $\Pi =< \varpi, \eta_0, \mu_0 >$ is $s_0 =<< \mu_0, \epsilon, \epsilon >, \eta_0 >$, where ϵ stands for the empty stack.

4.2. Behaviour of the Abstract Machine

4.2.1. Computations

The state of $\mathcal{M}$ changes according to the given program Π. We suppose that the inputs are made by *polling*, i.e. by special "input instructions", and likewise, the outputs are generated by "output instructions". The abstract machine executes instructions in numerical sequences unless told to do otherwise by a jump statement. So, the execution of an instruction may change the *state* and may have different *continuations*. We adopt the following format for a transition:

$$\alpha \xrightarrow[\delta/\delta']{A} \beta \qquad \text{(transition rule)} \tag{1}$$

where $\delta =< \mu, \pi, \rho >$ is the data before the execution of the instruction A, the address of which is α. $\delta' =< \mu', \pi', \rho' >$ is the data after the execution and β is the address of the next instruction to be executed. In fact, Rule 1 expresses that starting with state $< \delta, \alpha >$, the execution of A leads to state $< \delta', \beta >$. Note that $\beta = \alpha + 2$ if the control is passed in sequence.

Since the program is written in a low-level language (an assembling language), the semantics of a program is easily derived from the semantics of each instruction using a *composition rule* ($\bullet$). Rewriting rules are convenient for expressing the composition.

$$\frac{\alpha \xrightarrow[\delta/\delta']{A} \beta \quad \beta \xrightarrow[\delta'/\delta'']{B} \gamma}{\alpha \xrightarrow[\delta/\delta'']{A \bullet B} \gamma} \qquad \text{(composition)} \tag{2}$$

$A \bullet B$ means A followed by B, or B is a continuation for A. Note that this composition rule is not as simple as the one used in programs without "goto" statements. The theoretical framework adequate to deal with jumps is the theory of continuations. This theory is beyond the scope of this paper. For our performance evaluation we have no necessity to rely on a full fledged semantical theory.

Given an initial state $s_0 = <\delta_0\,,\,\alpha_0>$, a *computation* from this state α_0, is the sequence of instructions obtained by iterative application of the composition rule:

Definition. $A_0 \bullet A_1 \bullet \cdots \bullet A_n \bullet \cdots$ is a **computation** from $<\delta_0\,,\,\alpha_0>$

$$\texttt{iff}\ \forall i : \alpha_i \xrightarrow[\delta_i/\delta_{i+1}]{A_i} \alpha_{i+1}.$$

Thanks to the composition rule it is sufficient to give the semantics of each elementary instruction.

4.2.2. Semantics of RTX2000's instructions

RTX2000's instructions fall into 15 classes. The relationship between instructions and their category depends on their stack and register effects. All instructions are 16 bits long with one exception for the literal value which requires 16 bits for the instruction and 16 bits for the literal value.

Most instructions are *data processing*, they may change μ, π or ρ, but they pass the control in sequence (i.e. $\beta = \alpha + 2$). Some are *control instructions*, they may change μ, π or ρ and they may alter the normal flow of control. We list the transition rules of such instructions (viz., `BRANCH`, `?DUP OBRANCH`, `OBRANCH`, `NEXT`, `EXIT`, `CALL`, `EXECUTE`). The semantics of other instructions could be easily deduced from the RTX2000's reference manual.

Notation 1 *When the uppermost elements of a stack are significant, they can be distinguished, e.g. $\pi.b.a$ where $\pi \in$ Value*, and $b, a \in$ Value. The rightmost item in each list is the top stack element, the item before is the content of the cell just beneath the top, and so on.*

Control Instructions:

- unconditional branch

$$\alpha \xrightarrow[\delta/\delta]{\texttt{BRANCH}\ \beta} \beta \tag{3}$$

- conditional branch

$$\alpha \xrightarrow[<\mu\,,\,\pi.0\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho>]{\texttt{?DUP OBRANCH}\ \beta} \beta \tag{4}$$

$$\texttt{if}\ b \neq 0\ \texttt{then}\ \alpha \xrightarrow[<\mu\,,\,\pi.b\,,\,\rho>/<\mu\,,\,\pi.b\,,\,\rho>]{\texttt{?DUP OBRANCH}\ \beta} \alpha + 2 \tag{5}$$

$$\alpha \xrightarrow[<\mu\,,\,\pi.0\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho>]{\texttt{0BRANCH}\ \beta} \beta \tag{6}$$

$$\texttt{if}\ b \neq 0\ \texttt{then}\ \alpha \xrightarrow[<\mu\,,\,\pi.b\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho>]{\texttt{0BRANCH}\ \beta} \alpha + 2 \tag{7}$$

- exit from FOR$\cdots$ NEXT loop

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.0>/<\mu\,,\,\pi\,,\,\rho>]{\texttt{NEXT}\ \beta} \alpha + 2 \tag{8}$$

$$\texttt{if}\ n \neq 0\ \texttt{then}\ \alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.n>/<\mu\,,\,\pi\,,\,\rho.n-1>]{\texttt{NEXT}\ \beta} \beta \tag{9}$$

- return from a word [2]

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.\beta>/<\mu'\,,\,\pi\,,\,\rho>]{\texttt{EXIT}^*} \beta \tag{10}$$

- call to a word

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho.\alpha+2>]{\texttt{CALL}\ \beta} \beta \tag{11}$$

- "vectored" call to a word

$$\alpha \xrightarrow[<\mu\,,\,\pi.\beta\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho.\alpha+2>]{\texttt{EXECUTE}} \beta \tag{12}$$

Remark: RTX2000 has a streamed instruction feature. Details about this mode are omitted because our programs do not make use of it.

The composition rule and the different rewriting rules allow to follow a program execution step by step. But this low level representation is too much detailed for effective performance evaluation and does not lend itself to symbolic execution. Because a program results from some high-level source program, there exist underlying structuration. In order to make use of this structuration, we introduce a new level: *"the lumped program level"*.

4.3. Restructuration of RTX2000 programs

4.3.1. The Words

The FORTH's words are the only entities of the original FORTH's program that can be easily identified in the disassembled code. The whole program is a word by itself. Most of the words have been defined by the programmer. Some words have been generated by the compiler (e.g: =,>, `do`, `loop`, ...). The code produced by the T-FORTH compiler is such that each word has one and only one entry address. A word has at least one `EXIT` instruction. All the instructions of a word are stored in contiguous program memory locations. They form a block with a unique starting point and which ends with an `EXIT` instruction. More formally:

Definition. The **word** is the programming unit. With each word W are associated:

- **instruction**(W) $\subset$ **Inst** (The set of the instructions in W)
- **address**(W) $\subset$ **Addr** (The set of the addresses of instructions in W)
- **entry**(W) $\in$ **address**(W) (The unique entry point of W)
- **exit**(W) $\subset$ **address**(W) (The non-empty set of exit points of W)

instruction(W), **address**(W), **entry**(W), and **exit**(W) are easily constructed by a simple inspection of the code. A *complete computation* of a word W is an execution from a call to W, up to one of its `EXIT` instructions.

Definition. $\alpha_0 \xrightarrow[\delta/\delta']{A_0 \bullet A_1 \bullet \cdots \bullet A_{n-1}} \alpha_n$ is a **complete computation** of W from $< \delta_0 \, , \, \alpha_0 >$ `iff`

- $A_0 \bullet A_1 \bullet \cdots \bullet A_{n-1}$ is a computation from $< \delta_0 \, , \, \alpha_0 >$,
- $(A_0 = \texttt{CALL}\ \beta) \wedge (\beta = \mathbf{entry}(W))$,
- $\alpha_{n-1} = @(A_{n-1}) \in \mathbf{exit}(W)$,
- $\forall i \in 0..n-2 : A_i \notin \mathbf{exit}(W)$.

Notation 2 `WORDS` *denotes the set of the words of the program at hand.*

Notation 3 *A complete computation of* W *from* $< \delta \, , \, \alpha >$ *is denoted by* $\alpha \xrightarrow[\delta/\delta']{[\![W]\!]} \beta$ *where* $< \delta' \, , \, \beta >$ *is the state after the execution of W.*

Generally, words are called by other words. Most words behave like ordinary subroutines: after their execution they pass the control back to the caller. In this case, the data memory and the parameter stack may change, but the return stack is not altered by the complete execution:

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho>/<\mu'\,,\,\pi'\,,\,\rho>]{[\![W]\!]} \alpha+2$$

All the words we have defined act like that. Some compiler-defined words alter the contents of $\mathcal{RS}$. Let's call these words "**special words**". `do`, `loop` and `+loop` are the only special words we have found in our code. The words `loop` and `+loop` have different return points, this specificity implies a special treatement for these words. Starting with the actual code generated by the T-FORTH compiler for those special words, and applying the composition rule, we have derived the following behaviours:

$$\alpha \xrightarrow[<\mu\,,\,\pi.l.i\,,\,\rho>/<\mu\,,\,\pi\,,\,\rho.l.i>]{[\![\texttt{do}]\!]} \alpha+2 \tag{13}$$

$$\texttt{if}\ l \neq i+1\ \texttt{then} \qquad \alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.l.i>/<\mu\,,\,\pi\,,\,\rho.l.i+1>]{[\![\texttt{loop}]\!]} \alpha+2 \tag{14}$$

$$\texttt{if}\ l = i+1\ \texttt{then} \qquad \alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.l.i>/<\mu\,,\,\pi\,,\,\rho.l.l>]{[\![\texttt{loop}]\!]} \alpha+4 \tag{15}$$

Words `do` and `loop` or `do` and `+loop` are used as paired bracketting words for iteration. It is unsafe for programmer to define his (her) own special words. We have excluded this possibility for our applications.

4.3.2. Loops

Our programs must run in "real-time", i.e. the duration of computations are strictly bounded in time. Only finite loops are allowed. T-FORTH provides two styles of definite loops: `DO`...`LOOP` and `FOR`...`NEXT` which is a more efficient fixed loop for RTX2000. Let's see how these T-FORTH's loops are implemented in RTX2000.

The `FOR`...`NEXT` loop is the simplest. It involves only elementary instructions. Let n be the initial value of the loop index, "`n FOR <body> NEXT`" is translated into:

	`LIT n`	(n is actually an integer)
α	`>R`	(pop the index from $\mathcal{PS}$ and push it onto $\mathcal{RS}$)
β	`<body>`	(code of the body of the loop)
γ	`NEXT` β	
$\gamma+2$	$\cdots$	

Assuming that there is no forced exit from the body, according to the semantics of `NEXT`, it is easy to establish that the `<body>` is executed (n+1) times.

The `DO-LOOP` loop needs two special words `do` and `loop`. Let `l` and `i` be two integers, "`l i DO <body> LOOP`" is implemented as:

	`LIT l`	(l: limit parameter is an integer)
	`LIT i`	(i: initial index parameter is an integer)
α	`CALL do`	
β	`<body>`	(body of the loop)
γ	`CALL loop`	
$\gamma+2$	`BRANCH` β	
$\gamma+4$	`R> DROP`	(discard the index parameter)
$\gamma+6$	`R> DROP`	(discard the limit parameter)
$\gamma+8$	...	

This case is less trivial than the previous one. Its semantics is deduced from the semantics of its components. Before entering the loop, two parameters l (limit) and i (index) are pushed onto $\mathcal{PS}$. Assuming no forced exit within the body, the number of iterations is $n = \texttt{if } l > i \texttt{ then } l - i \texttt{ else } 65536 + l - i$.

Because of the restrictions imposed on our loops, there are "*natural loops*" [8]: they have one entry point and one exit point. The instructions of the body are in contiguous program memory locations. The body is similar to a word. We introduce a new entity: the *pseudo-word* for bodies of loops.

Definition. For each body of a loop, a **pseudo-word** is defined. With each pseudo-word pW are associated:

- $\mathbf{instruction}(pW) \subset \mathbf{Inst}$ (The set of the instructions in pW)
- $\mathbf{address}(pW) \subset \mathbf{Addr}$ (The set of the addresses of instructions in pW)
- $\mathbf{entry}(pW) \in \mathbf{address}(pW)$ (The unique entry point of pW)
- $\mathbf{exit}(pW) \in \mathbf{address}(pW)$ (The unique end point of pW)

$\mathbf{instruction}(pW)$, $\mathbf{address}(pW)$, $\mathbf{entry}(pW)$, and $\mathbf{exit}(pW)$ are easily constructed by a simple inspection of the code. The exit of a pseudo-word is the address of the last instruction of the body. In an altered (loop-free) version of the program this instruction is substituted by an `EXIT` (see Table 1).

Notation 4 `PSEUDOWORDS` *denotes the set of the pseudo-words of the program at hand. Let* `GWORDS` = `WORDS` ∪ `PSEUDOWORDS`.

Extending the definition of complete computation to pseudo-words, we can express the behaviour of our loops as:

- `FOR–NEXT` loop:

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.n>/<\mu'\,,\,\pi'\,,\,\rho>]{[\![pW]\!]} \beta + 2 \ where\ \beta = \mathbf{exit}(pW)$$

- `DO–LOOP` loop:

$$\alpha \xrightarrow[<\mu\,,\,\pi\,,\,\rho.l.i>/<\mu'\,,\,\pi'\,,\,\rho.l.l>]{[\![pW]\!]} \beta+2 \ \mathit{where}\ \beta = \mathbf{exit}(pW)$$

4.3.3. Lumped programs

Words and pseudo-words allow a more abstract view of a program. Starting with the disassembled code, we derive a "lumped program" by a systematic translation.

Definition. The **lumped program** is derived from the disassembled code by the textual substitutions defined in Table 1.

Table 1. Program Lumping

Code			Condition
Disassembled	Alteration	Lumped	
α :CALL β		α :W_EXEC(W)	$\beta = \mathbf{entry}(W)$
α :EXECUTE		α :V_EXEC(L)	L = list of called words
β :<body> : γ :NEXT β	 EXIT	β :N_EXEC(pW, n) $\beta + 2$:BRANCH $\gamma + 2$:	$\beta = \mathbf{entry}(pW)$ n = initial index
β :<body> : γ :CALL loop $\gamma + 2$:BRANCH β	 EXIT	β :D_EXEC(pW, l, i) $\beta + 2$:BRANCH $\gamma + 4$: :	$\beta = \mathbf{entry}(pW)$ l = limit parameter i = initial index
α :OTHERS		α :OTHERS	catch all

A lumped program includes new instructions that are abstractions of piece of code:

- `W_EXEC` and `V_EXEC` are procedural abstractions,
- `N_EXEC` and `D_EXEC` stand for (bounded) iterative executions.

Definition. The domain of the instructions "**INST**" of the lumped program is such that

$$\mathbf{INST} = (\mathbf{Addr} \times \mathbf{OPER}) \cup (\mathbf{Addr} \times \mathbf{OPER} \times \mathbf{OPERAND})$$

where $\mathbf{OPER} = (\mathbf{Oper} \backslash \{\mathtt{CALL}, \mathtt{NEXT}, \mathtt{EXECUTE}\}) \cup \{\mathtt{W\_EXEC}, \mathtt{V\_EXEC}, \mathtt{N\_EXEC}, \mathtt{D\_EXEC}\}$
and, $\mathbf{OPERAND} =$
$\mathbf{Operand} \cup \mathtt{WORDS} \cup \mathtt{WORDS}^* \cup (\mathtt{PSEUDOWORDS} \times \mathbb{N}) \cup (\mathtt{PSEUDOWORDS} \times \mathbb{N} \times \mathbb{N})$

Notation 5 *Denote by* $\alpha \overset{A}{\twoheadrightarrow} \beta$ *the transition of the lumped program from instruction* $\alpha : A$ *to* β *(the address of the first instruction to be executed after* A*).*

Because of our lumping rules:

- $\beta = \alpha + 2$ for `W_EXEC`, `V_EXEC`, `N_EXEC`, `D_EXEC`
- otherwise, β is given by the (ordinary) transition rule.

We extend the notation to sequences of instructions of the lumped program, to words, and to pseudo-words as we did for ordinary computations (Sec 4.2.1).

Since iterations are replaced by `N_EXEC` or `D_EXEC`, lumped programs are loop-free. Moreover, each word is a "flat program" without subroutine call. Therefore, DAG (Directed Acyclic Graph) is a possible representation for the control flow of each word, in a lumped program.

4.4. Duration

4.4.1. Elementary instructions

RTX2000 has a full predictable behaviour. Each elementary instruction takes a given number of clock cycles to be executed: 1 or 2 cycles.

Definition. $\nu : \mathbf{Oper} \rightarrow \mathbb{N}$ yields the duration of an elementary instruction in clock cycle number.

4.4.2. Computations

ν is extended to computations as usually. The duration of an actual computation $A_0 \bullet A_1 \bullet \cdots \bullet A_{n-1}$ from the state $< \delta , \alpha >$ is deterministic and its value is the sum of the duration of each instruction.

Definition. $\nu : \mathbf{Inst}^* \rightarrow \mathbb{N}$ is such that:

$$\nu(A_0 \bullet A_1 \bullet \cdots \bullet A_{n-1}) = \sum_{k=0}^{n-1} \nu(A_k).$$

4.4.3. Symbolic computations

A symbolic computation from α is an execution of the lumped program, not taking actual data into account. Because a symbolic computation represents many actual sequences the duration is no longer an integer but a set of integer values. We characterize those values by the least interval of $\mathbb{N}$ that covers all the values.

Notation 6 *A duration of a symbolic computation is defined by a pair of integers (low and high integers) and denoted as [low, high].*

Definition. $\texttt{Dur} : \texttt{GWORDS} \times \mathbf{Addr} \rightarrow \mathbb{N} \times \mathbb{N}$ is the duration of a symbolic computation of a word or a pseudo-word from an address up to its termination.
For $W \in \texttt{GWORDS}$, $Dur(W, \alpha)$ is defined only for $\alpha \in \mathbf{address}(W)$.

Notation 7 $\forall W \in \texttt{GWORDS} : Dur([\![W]\!])$ *stands for* $Dur(W, \mathbf{entry}(W))$.

Notation 8 *We introduce operations over integer intervals:*

- $+ : [a, b] + [c, d] = [a + b, c + d]$,
- $* : n * [a, b] = [n * a, n * b]$ *where* $n \in \mathbb{N}$

The evaluation of $Dur([\![W]\!])$ is made by graph traversal of the DAG associated with W. The following recursive algorithm can be applied:

Recursive computation of `Dur`*:*

Let $U \in \texttt{GWORDS}$ and $\alpha : A$ be an instruction of U in the lumped program.
$\texttt{Dur}(U, \alpha) =$

`case of` A

$\texttt{EXIT}^* : [1, 1]$

$\texttt{W\_EXEC}(W) : [1, 1] + \texttt{Dur}([\![W]\!]) + \texttt{Dur}(U, \alpha + 2)$

$\texttt{V\_EXEC}(L) : [1, 1]+$
$[\min_{W \in L} \texttt{Dur}([\![W]\!]).low, \max_{W \in L} \texttt{Dur}([\![W]\!]).high] + \texttt{Dur}(U, \alpha + 2)$

$\texttt{N\_EXEC}(pW, n) : (n + 1) * \texttt{Dur}(pW) - [1, 1] + \texttt{Dur}(U, \alpha + 2)$

$\texttt{D\_EXEC}(pW, l, i) : (l - i) * \texttt{Dur}(pW) - [2, 2] + \texttt{Dur}(U, \alpha + 2)$

$\texttt{OBRANCH}\ \beta : [1, 1]+$
$[\min\{\texttt{Dur}(U, \beta).low, \texttt{Dur}(U, \alpha + 2).low\},$
$\max\{\texttt{Dur}(U, \beta).high, \texttt{Dur}(U, \alpha + 2).high\}]$

`otherwise`

$[\nu(A), \nu(A)] + \texttt{Dur}(U, \beta)$ where $\alpha \xrightarrow{A} \beta$

`end case`

It appears that $\texttt{Dur}([\![U]\!])$ needs the duration of all the words and pseudo-words used in the body of U. This dependence induces an order that facilitates the computation.

Definition. $\prec \subseteq$ `GWORDS` $\times$ `GWORDS` is a "callee-caller" dependence relation such that:

$$U \prec W \Longleftrightarrow U \textit{ is used by } W$$

U is used by W means either U is called by W (if U is a word), or the loop associated with U is included in the body of W (if U is a pseudo-word).

Given a program, (`GWORDS`, $\prec$) is a POset. The set `GWORDS` can be totally ordered:

Definition. Define a linear ordering on `GWORDS` $\sqsubset \subseteq$ `GWORDS` $\times$ `GWORDS` such that:

$$\forall U, W \in \texttt{GWORDS} : U \prec W \Longrightarrow U \sqsubset W$$

.

(`GWORDS`, $\sqsubset$) is easily derived from (`GWORDS`, $\prec$) by a "topological sort". When the duration of all the words and pseudo-words is computed according to the order $\sqsubset$, the previous algorithm can be programmed in an efficient way. Each piece of code is analyzed only once.

4.5. Results

Results about a full session of implementation on the RTX2000 can be found in [9]. This technical report describes a complete session, at a detailed level. Starting with an example, the different "mutations" of the initial ESTEREL program are illustrated, and quantitative results on the effective duration are given. As above-mentioned, results heavily depend on the chosen example: from a few units to hundreds of clock cycles for the duration of words. Quantitative results are of little importance for the reader. The interest of this analysis is to point out the "bottlenecks" in the run-time code. Naturally, during the perfecting phase of our method, optimization efforts have focused on those inefficient words. For instance, the `DO...LOOP` loops have been replaced by the `FOR...NEXT` loops, a faster iterative structure on the RTX2000.

A most important outcome of this temporal analysis is the evidence of the high price to pay for the input/output management. Solutions may vary from *polling* to fully *interrupt-driven* data acquisition. For polling, the acquisition time is proportional to the number of inputs, and can be statically determined (duration of the word `Polling`). For interrupts, the cost must integrate the duration of the interrupt-handling and the dynamics of the inputs.

Let `Polling` and `Handler` be words associated with the input management policy,
T the duration of the word associated with a given transition,
`Cleaning` the management word executed at the end of the transition,
Let n be the number of input signals.
The expression of the duration of a complete reaction can be expressed as follows:

In case of polling policy:

$$D_P = \texttt{Dur}(\llbracket\texttt{Polling}\rrbracket) + T + \texttt{Dur}(\llbracket\texttt{Cleaning}\rrbracket) \tag{16}$$

In case of interrupt handling:

Let Δ_k be the minimal duration between two successive occcurrences of input k, $\texttt{Handler}(k)$ the interrupt handler for k.

$$D_I = \frac{T + \texttt{Dur}(\llbracket\texttt{Cleaning}\rrbracket)}{1 - \sum_{k=1}^{k=n} \frac{\texttt{Dur}(\llbracket\texttt{Handler}(k)\rrbracket)}{\Delta_k}} \tag{17}$$

Though n does not appear explicitely in formula (16), $\texttt{Dur}(\llbracket\texttt{Polling}\rrbracket)$ is proportional to n. There is a trade-off between the speed and the complexity of the program. It is up to the designer to choose the best solution according to its knowledge on the environment.

5. Conclusion

We aimed at producing predictable code for a specific class of systems known as "real-time controllers". The solution advocated in this paper uses a synchronous approach enriched with an automatic code generation for an industrial microcontroller and a temporal performance analyzer.

Our method results from a conjunction of favourable factors:

1. High-level programming of the reactive behaviour of the application,
2. Efficient implementation on a fast and predictable processor,
3. Intermediate language suitable for performance analysis,
4. Automatic code translations,
5. Sound semantic foundation.

The first factor is the ESTEREL language which is dedicated to reactive system programming. This language is based on mathematical semantics. This formal background allows to prove *the logical correctness* of ESTEREL programs. Safety properties can be checked by the analysis of the automaton generated by the compiler and semantically equivalent to the source program. In order to preserve the logical correctness of a program, we have proposed automatic translations between the different intermediate codes. Of course, an automatic translation avoids the error-prone hand-coding process, but this is not sufficient to guarantee a property-preserving translation. Thus, our transformations rely on *formal models*: the processor is viewed as an abstract machine, and a behavioural semantics is given to its programming language.

The second factor is the processor itself. The RTX2000 has been specially designed for real-time controller implementations, and yet, its programming language (T-FORTH) may hinder its development. T-FORTH, which is a dialect of FORTH, is not a classical language and its reverse-Polish programming style may disconcert many of its potential users. Using a high-level language like ESTEREL and our sound translation techniques, eliminates this drawback. Nevertheless, T-FORTH is useful as an intermediate language (Factor 3). The structures underlying this language, once formalized, lend themselves to tractable performance evaluation of the code. We have explained why and how the duration of executions can be known at compile-time. This prediction of the temporal behaviour is necessary to establish the *temporal correctness* of the implementation.

The method presented in this paper is a realistic solution to the issue of implementing safe and predictable (small) real-time systems. The approach is both pragmatic and formal. It is an illustration of the fact that the design of modern controller is no longer a simple matter of "know-how", but a well-balanced mixture of high-level programming, code analysis and hardware.

Notes

1. Creator of FORTH.
2. `EXIT` may be combined with some data processing instruction that may modify μ. `EXIT*` denotes both simple and combined exit statements.

References

1. W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems.* Amsterdam: Kluwer Academic Publishers, 1991.
2. *RTX2000 Reference manual.* Harris Semiconductor Corporation, Melbourne (FL), 1988.
3. A. Benveniste and G. Berry., "Another look at real-time programming," *Proceeding of the IEEE*, vol. 79, pp. 1268–1336, September 1991.
4. A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceeding of the IEEE*, vol. 79, pp. 1270–1282, September 1991.
5. F. Boussinot and R. De Simone, "The Esterel language," *Proceeding of the IEEE*, vol. 79, pp. 1293–1304, September 1991.
6. C. André and M.-A. Péraldi, "From Esterel to Forth," (Kansas-City (USA)), SIGForth'92 FORTH Language Workshop, Mars 1992.
7. C. André and M.-A. Péraldi, "Effective implementation of Esterel programs," (Oulu (Finland)), pp. 262–267, 5^{th} Euromicro Workshop on Real-Time Systems, IEEE, June 1993.
8. A. Aho, R. Sethi, and J. Ullman, *Compilers Principles, Techniques and Tools.* Addison Wesley Publishing Company, 1988.
9. C. André and M.-A. Péraldi, "A comprehensive session of synchronous program implementation on RTX2000," Tech. Rep. TR 93–46, I3S, Sophia-Antipolis, France, July 1993.