APPLYING UNIFYING THEORIES OF PROGRAMMING TO REAL-TIME PROGRAMMING

Alvaro E. Arenas
Juan C. Bicarregui
E-Science Centre, CCLRC Rutherford Appleton Laboratory, Oxfordshire, UK

This paper introduces an approach to verify the correctness of the implementation of real-time languages. We apply the techniques presented in Hoare and He's Unifying Theories of Programming to reason about the correctness of compilers and schedulers for real-time languages, using high-level abstractions such as algebraic laws. In the compilation process, the existence of unique fixed-points is exploited to verify the implementation of crucial real-time operators such as asynchronous input, delay and timeout. It is developed an abstract model for scheduling real-time programs into a uniprocessor machine. The applicability of the model is shown by instancing it with two types of schedulers: a round-robin scheduler, employed when the participating parallel processes do not include deadline constraints, and a priority-based scheduler, used when each participating process is periodic and possesses an associated deadline.

Keywords: *unifying theories of programming, real-time programming, compiler specification, scheduling.*

1. Introduction

Safety-critical computer systems must be engineered to the highest quality in order to anticipate potential faults and to reduce the possibility of erroneous and unexpected behaviour. Correctness of system properties must then be guaranteed at all levels, from specification to low-level implementation into a target machine.

This paper presents a way of reasoning about correctness of compilers and schedulers using high-level abstractions such as algebraic laws. We apply the techniques presented in Hoare and He's *Unifying Theories of Programming* (Hoare and He Jifeng 1998) to model compilation of real-time programs and to reason about their scheduling. In our real-time language, processes communicate asynchronously via communication queues called shunts. A shunt can be seen as a directed channel with the capability of buffering messages. As the sender transmits a message, it stores it into the corresponding shunt and proceeds with the execution. When the receiver reads a shunt, it takes the oldest message deposited in it. However, if the shunt is empty, the receiver is blocked until a message arrives. The main advantage of this asynchronous mechanism is the loose coupling it provides between system parts: a sender is never blocked because a receiver is not ready to communicate. This communication scheme is adopted by several asynchronous models such as versions of CSP (Kumar and Pandya 1993) and SDL (Mitschele-Thiel 2001), or the Real-Time Specification for Java (Botella and Goslin 2000), among others.

The paper follows the next structure. Section 2 describes the source programming language and presents its main algebraic laws. Then, section 3 introduces the target language and develops some algebraic properties of machine-level programs. Section 4 formalises the compiling correctness relation that must hold between source and target code and illustrates the compilation of constructors for sequential programs. In section 5 we develop our abstract model of scheduling. We then illustrate in section 6 the application of the model to a round-robin scheduler. Following, section 7 presents the application to a fixed priority scheduler with pre-emption. Finally, section 8 gathers some concluding remarks and discusses possible extensions to our work.

2. The Real-Time Programming Language

Our real-time language is a small imperative language with real-time constructors such as deadline and delay, and primitives for asynchronous communication via communication queues called shunts. Its syntax is given by the following description:

$$\begin{array}{llllllll}
P & ::= & \perp & \mid & II & \mid & \bar{x} := \bar{e} & \mid \; s\,!\,e \\
 & \mid & s\,?\,x & \mid & \Delta d & \mid & [d]P & \mid \; P;P \\
 & \mid & P \sqcap P & \mid & P \trianglelefteq b \trianglerighteq P & \mid & \mathtt{while}(b,P) & \\
 & \mid & [P \rhd^s_d P] & \mid & P \parallel P & & &
\end{array}$$

where P stands for a process, $\bar{x}$ is a list of variables, x is a variable, s is a queue, $\bar{e}$ is a list of expressions, e is an expression, b is a Boolean expression, and d is a time expression.

The chaotic process $\perp$ defines an arbitrary behaviour, which is beyond control. The skip process II does nothing, terminating immediately. The multiple assignment $\bar{x} := \bar{e}$, where $\bar{x}$ is a list of distinct variables and $\bar{e}$ an equal-length list of expressions, evaluates the components of $\bar{e}$ and stores these results simultaneously into list $\bar{x}$, preserving the original ordering of the elements. We assume here that the evaluation of an expression always delivers a result and does not change the value of any variable, i.e. no *side-effect* is allowed. The output $s\,!\,e$ stores the value of the expression e into the output queue s, leaving all program variables unchanged. The input $s\,?\,x$ takes the oldest message from queue s and stores it into variable x. If the queue is empty, the process is blocked until a message arrives. We adopt the realistic premise that all communicating processes take time, the amount of time consumed by the instruction not being specified.

Composition $P;Q$ represents a process that executes P first and, at termination of P, starts with the execution of Q. It is assumed that there is no delay associated with the transfer of control from P to Q. Process $P \sqcap Q$ represents the non-deterministic choice between the participating processes. The conditional $P \trianglelefteq b \trianglerighteq Q$ represents a choice between alternatives P and Q in accordance with the value of Boolean expression b; it behaves like P if b is true, and like Q if b is false. It is assumed that some arbitrary time is spent in the evaluation of the guard b. The iteration $\mathtt{while}(b,P)$ executes process P while condition b is true, and terminates when the condition is false. It is also assumed that some time is spent in each iteration evaluating expression b.

The delay process Δd is guaranteed to wait for a minimum of d time units before terminating. The process $[d]P$ behaves as P and its execution does not take more than d time units. The timeout process $[P \rhd^s_d Q]$ monitors input queue s for d time units; if there is a message in s during that time, it executes process P, otherwise it executes process Q. Parallel composition $P \parallel Q$ describes the concurrent execution of processes P and Q. Each process has its own program state, which is inaccessible to its partner, and interacts with its partner and the external world via communication through shared queues.

In previous work (Arenas 2001), we have given a specification-oriented semantics to our language and derived its main algebraic laws. The semantic is constructed by following the predicative approach described in (Hoare and He Jifeng 1998), where a process is modelled as a predicate that describes all the *observations* that it can generate. Notation $P \equiv Q$ is used to denote that processes P and Q are semantically equivalent and proved that all derived laws are sound with respect to the model. Further, we use ordering relation $P \sqsubseteq Q$ to indicate that Q is at least as deterministic as P. It is defined in terms of non-deterministic choice as $P \sqsubseteq Q \ \widehat{=}\ (P \sqcap Q) \equiv P$.

Laws for primitive programs coincide with classical laws for imperative sequential programs and communicating processes.

Law 1 *Laws for Primitive Programs*
(1) $P;\ II \ \equiv\ II;\ P \ \equiv\ P$
(2) $\bot;\ P \ \equiv\ \bot$
(3) $x, y := e, y \ \equiv\ x := e$
(4) $x := e;\ x := f(x) \ \equiv\ x := f(e)$
(5) $x := e;\ s\,!f(x) \ \equiv\ x := e;\ s\,!f(e)$
(6) $s\,?y;\ x := y \ \equiv\ s\,?x;\ y := x$
(7) $x := e;\ s\,!f \ \equiv\ s\,!f;\ x := e$ *provided x is not free in f* .

Let us explain some of the above laws. Law 1(1) shows that II is unit of sequential composition. Law 1(2) expresses that once a process is out of control, its sequential composition with another process does not redeem the situation. In our formal model (Arenas 2001), an assignment may take time, but the amount of time consumed by the instruction is not specified; this allows us to derive law 1(4). Law 1(7) describes a special case of commutation between assignment and output.

The following laws describe some properties of the real-time operators.

Law 2 *Laws for Real-Time Operators*
(1) $\Delta d_1;\ \Delta d_2 \ \equiv\ \Delta(d_1 + d_2)$
(2) $[d_1]P \sqsubseteq [d_2]P$ *provided* $d_2 \leq d_1$
(3) $[P \rhd^s_1 [P \rhd^s_d Q]] \equiv [P \rhd^s_{d+1} Q]$
(4) $[[P \rhd^s_0 R] \rhd^s_d Q] \equiv [P \rhd^s_d Q]$

2.1. Some Auxiliary Processes

We introduce here some auxiliary processes useful in reasoning about process behaviour. The idle process Δ represents a process that may terminate at any arbitrary time without changing any variable or shunt. The conditional process $(P \lhd b \rhd Q)$ selects one alternative depending on the value of expression b; if b is true, it acts like process P, otherwise it behaves like Q. It differs from the conditional of our programming language in that it is assumed that the evaluation of b does not take time. The miracle program, denoted by $\top$, stands for a product that can never be used because its conditions of use are impossible to satisfy. The assumption of b, denoted by $b^\top$, can be regarded as a miracle test: it behaves like *II* if b is true; otherwise it behaves like $\top$. By contrast, the assertion of b, denoted by $b_{\perp}$, also behaves like *II* if b is true, otherwise it fails, behaving like $\perp$. The declaration $\texttt{var}\,x$ introduces new program variable x and permits x to be used in the portion of the program that follows it. The complementary operation, $\texttt{end}\,x$, terminates the region of permitted use for variable x. The next law shows some examples of the use of auxiliary processes.

Law 3 *Laws for Auxiliary Processes*
(1) $b^{\top};\ (P \lhd b \rhd Q)\ \equiv\ b^{\top};\ P$
(2) $x := e;\ (x = e)^{\top}\ \equiv\ x := e$
(3) $\Delta;\ s!e\ \equiv\ s!e\ \equiv\ s!e;\ \Delta$
(4) $(s \neq \langle\rangle)^{\top};\ [P \rhd^{s}_{d} Q]\ \equiv\ (s \neq \langle\rangle)^{\top};\ \Delta;\ P$
(5) $(\texttt{end}\,x;\ \texttt{var}\,x) \sqsubseteq II\ \equiv\ (\texttt{var}\,x;\ \texttt{end}\,x)$
(6) $\Delta;\ \texttt{end}\,x\ \sqsubseteq\ x := e;\ \texttt{end}\,x$
(7) *If x is not free in P then* $P;\ \texttt{var}\,x\ \equiv\ \texttt{var}\,x;\ P$ *and* $\texttt{end}\,x;\ P\ \equiv\ P;\ \texttt{end}\,x$.

Let X be the name of a recursive program we wish to construct, and $F(X)$ a function on the space of processes denoting the intended behaviour of the program. We can show that the space of processes forms a complete lattice (Arenas 2000). Notation $\mu X.F(X)$ stands for the least fixed point of function F and notation $\nu X.F(X)$ denotes the greatest fixed point of F. The following law illustrates the main properties of these operators.

Law 4 *Fixed Point Laws*
(1) $F(\mu X.F(X))\ \equiv\ \mu X.F(X)$
(2) $F(Y) \sqsubseteq Y\ \Rightarrow\ \mu X.F(X) \sqsubseteq Y$
(3) $F(\nu X.F(X))\ \equiv\ \nu X.F(X)$
(4) $F(Y) \sqsupseteq Y\ \Rightarrow\ \nu X.F(X) \sqsupseteq Y$.

The iteration $b * P$ can be defined as the least fixed point of the equation $\mu X.((P;\ X) \lhd b \rhd II)$. Typical laws for the loop include the following.

Law 5 *Loop Laws*
(1) $b^{\top};\ b * P\ \equiv\ b^{\top};\ P;\ b * P$
(2) $(\neg\, b)^{\top};\ b * P\ \equiv\ (\neg\, b)^{\top}$.

There is an interesting case in which the least and greatest fixed points coincide, as described below.

Theorem 1 *Unique Fixed Point*
Let $F(X)$ be a monotonic function on the space of processes. If it is guaranteed that there is a delay of at least one time unit before invoking the recursion, then the fixed point of F is unique.

3. The Target Language

Our target machine has a rather simple architecture, consisting of a store for instructions $m : Addr \rightarrow Instr$, modelled as a function from the set of addresses to the set of machine instructions; a program counter $pc : Addr$ that points to the current instruction; and a data stack $st : \mathrm{seq}\,.\mathbb{Z}$, used to hold temporary values. The target language is an *intermediate-representation* language close to, but more abstract than the final machine code. Following tradition, the machine instructions are represented by updates to machine components. These instructions are introduced in Table 1.

Let us explain some of the instructions. Instruction $\mathtt{LD}(x)$ has variable x as its operand; its execution pushes the value of x onto the evaluation stack, and increases the value of the program counter pc by 1. Symbol $+\!\!+$ denotes concatenation of sequences; $\mathtt{last}.st$ stands for the last element of sequence st; $\mathtt{front}.st$ corresponds to the sequence obtained from eliminating last element of st. Instruction $\mathtt{ST}(x)$ pops the value at the top of the stack into variable x, and then passes the control to the next instruction; the requirement of having at least one element in the stack is expressed as an initial assumption in the instruction. Instructions $\mathtt{EV}(e), \mathtt{EVB}(b)$ and $\mathtt{EVT}(d)$ are used to evaluate integer, Boolean and time expressions respectively; the instructions push the result of evaluating the expression onto the top of the stack and increment the program counter by one. When non-integer values are stored into the stack, they are translated into the appropriate representation by using a *representation function* R_T, of type $T \rightarrow \mathbb{Z}$ for each basic type T, as presented in (Müller-Olm 1997, Lermer and Fidge 2002). Arithmetic instructions are introduced by means of the ADD and SUB instructions; the operation $\mathtt{front2}.st$ obtains the front of $\mathtt{front}.st$; similarly, the operation $\mathtt{last2}.st$ obtains the last element of $\mathtt{front}.st$. Comparison of the last two elements of the evaluation stack is introduced by the instructions LE and LT. Instructions JP, JPF and JPT are used for unconditional and conditional jump respectively. The instruction DUP duplicates the value stored at the top of the evaluation stack st. The output instruction $\mathtt{OUT}(s)$ sends the value on top of the stack through shunt s, taking that value out of the stack. The input instruction $\mathtt{IN}(s)$ is executed only when shunt s is not empty; it inputs the oldest message from s and leaves it at the top of the stack. Instruction $\mathtt{TST}(s)$ tests whether there is a message in shunt s. Instruction $\mathtt{STM}(s)$ stores in top of the stack the time stamp of the oldest unread message of s. The TIM instruction reads the current time and places it on top of the stack; it is simply a specification that the hardware implementator must guarantee.

The target language is a distinguished subset of the modelling language. The assignment statements are "timed" assignments so that time passes while an instruction executes. Let $\mathcal{T} : Instr \rightarrow Time$ be a function denoting the duration of executing a machine instruction such that $\mathcal{T}(\mathtt{INSTR}) > 0$ for $\mathtt{INSTR} \in Instr$. Notation $\mathcal{T}$ is used later to define the execution time of blocks of machine code.

3.1. Execution of Target Programs

The execution of a target program is represented by the repetition of a set of machine instructions. In this part we formalise such concepts, borrowing some elements from (Hoare and He Jifeng 1998).

Definition 1 *Labelled Instruction*
Let $\texttt{INSTR} : Instr$ *be a machine instruction as defined in Table 1 and* $l : Addr$ *a machine location. Labelled instruction* $l : \texttt{INSTR}$ *expresses that instruction* $\texttt{INSTR}$ *is executed when the program counter has value l. It is defined as* $l : \texttt{INSTR} \quad \widehat{=} \quad (\texttt{INSTR} \lhd pc = l \rhd II)$.

Labelled instructions are used to model the possible actions during the execution of a target program. The fact that the executing mechanism can perform one of a set of actions according to the current value of the program counter can be modelled by a program of the form $l_1 : \texttt{INSTR}_1 [] \, l_2 : \texttt{INSTR}_2 [] \cdots [] \, l_n : \texttt{INSTR}_n$ where locations $l_1, \cdots, l_n$ are pairwise disjoint and operator $[]$ denotes the *assembly* of machine programs.

Definition 2 *Assembly and Continuation Set*
– Let C be a machine program consisting only of labelled instruction $l : \texttt{INSTR}$*. Then, C is an assembly program with continuation set* $L.C = \{l\}$*.*
– Let C and D be assembly programs with disjoint continuation sets L.C and L.D respectively. The assembly program $(C [] D)$ *and its continuation set are defined as follows:*

$$\begin{array}{lcl} C [] D & \widehat{=} & (C \lhd pc \in L.C \rhd D) \\ & & \quad \lhd (pc \in L.C \cup L.D) \rhd II \\ L.(C[]D) & \widehat{=} & L.C \cup L.D \,. \end{array}$$

The continuation of assembly C denotes its set of valid locations. The value of the program counter determines the instruction to be executed.

Law 6 *Program Counter and Assembly Program*
Let $C = (l_1 : \texttt{INSTR}_1 [] \, l_2 : \texttt{INSTR}_2 [] \cdots [] \, l_n : \texttt{INSTR}_n)$ *be an assembly program. Then*
$(pc = l_i \wedge l_i \in L.C)^\top ; \; C \quad \equiv \quad (pc = l_i \wedge l_i \in L.C)^\top ; \; \texttt{INSTR}_i$.

The execution of an assembly program is modelled as a loop which iterates the program as long as the program counter remains within the continuation set.

Definition 3 *Execution*
Let C be an assembly program. Execution of program C is defined as follows: $C^* \;\widehat{=}\; (pc \in L.C) * C$*. The evaluation of the guard in the loop does not consume time. All execution time overheads are accounted for in the machine instructions.*

4. Compiling Sequential Programs

This section specifies a compiler that translates a sequential program into a target program represented as an assembly of single machine instructions whose behaviour represents an improvement with respect to that of the original source program. We also derive the execution time of each target program generated by the compiler.

Definition 4 *Compilation*
The correct compilation of a program is represented by a predicate $\mathcal{C}(P, a, C, z)$ *where P is the source program; C is a machine program stored in the code memory m, consisting of an assembly of single machine instructions; a and z stand for the initial and final addresses of program C, respectively. Predicate* $\mathcal{C}(P, a, C, z)$ *is formally defined by the following*

refinement:

$$\begin{aligned}\mathcal{C}(P,a,C,z) \;\widehat{=}\; P \sqsubseteq\; & (\mathtt{var}\, pc, st;\; (pc = a)^{\top};\\ & C^{*};\; (pc = z)_{\perp};\; \mathtt{end}\, pc, st)\,.\end{aligned}$$

In above definition, the declaration $\mathtt{var}\, pc, st$ introduces the machine components. The assumption $(pc = a)^{\top}$ expresses that program counter pc should be positioned at location a at the beginning of execution of C. The assertion $(pc = z)_{\perp}$ states the obligation to terminate with program counter pc positioned at location z.

Notation $\mathcal{T}_{\mathcal{C}}(P)$ is used to denote the worst-case execution time of the machine code that compiler specification $\mathcal{C}$ associates to source program P.

The compiler is specified by defining predicate $\mathcal{C}$ recursively over the syntax of sequential source programs. Correctness of the compiling relation follows from the algebraic laws of the language. We omit the proof for the classical sequential operators, since it follows lines similar to those of the untimed case, and refer the reader to (Arenas 2000). We outline the proof for output, input and timeout operators.

Assignment $x := e$ is implemented by a piece of code that evaluates expression e and stores the result into the corresponding program-variable store. Note that the duration of an assignment was unspecified at source level, however the code implementing it has an exact duration equal to the addition of the duration of each participating machine instruction.

Theorem 2 *Assignment Compilation*
$\mathcal{C}\,(\, x := e,\; a, (a : \mathit{EV}(e)\, [\,]\, a^{+1} : \mathit{ST}(x)), a^{+2})\;.$
$\mathcal{T}_{\mathcal{C}}(x := e) \;=\; \mathcal{T}(\mathit{EV}) + \mathcal{T}(\mathit{ST})$

Notation l^{+i} : `INSTR` states that machine instruction `INSTR` is located at position $l + i$. For simplicity, we are assuming that the evaluations of the integer expressions all have the same duration. We can determine the duration of evaluating an expression by using techniques for simplifying expressions.

Skip is implemented as an empty segment of code. Obviously, the duration of the code implementing the skip is zero. Let us assume that II also denotes a machine program with an empty location set, i.e $L.\mathit{II} = \emptyset$.

Theorem 3 *Skip Compilation*
$\mathcal{C}\,(\, \mathit{II},\; a,\, \mathit{II},\, a\,)\;.$
$\mathcal{T}_{\mathcal{C}}(\mathit{II}) \;=\; 0\;.$

The output process is implemented by a piece of code that evaluates the expression to be transmitted and then sends the value to the corresponding shunt. The duration time of the implementation is equal to the addition of its constituent machine instructions.

Theorem 4 *Output Compilation*
$\mathcal{C}\,(\, s\,!\,e,\; a,\; (a : \mathit{EV}(e)\, [\,]\, a^{+1} : \mathit{OUT}(s)),\; a^{+2}\,)\;.$
$\mathcal{T}_{\mathcal{C}}(s\,!\,e) \;=\; \mathcal{T}(\mathit{EV}) + \mathcal{T}(\mathit{OUT})\,.$
Proof:

$\mathtt{var}\, ps, st;\ (pc = a)^{\top};$
$(a : \mathtt{EV}(e) \,[]\, a^{+1} : \mathtt{OUT}(s))^{*};\ (pc = a^{+2})_{\perp};\ \mathtt{end}\, pc, st$

$\equiv$ *Definition of execution, def. 3, and loop laws, law 5*
$\mathtt{var}\, pc, st;\ (pc = a)^{\top};\ \mathtt{EV}(e);\ \mathtt{OUT}(s);$
$(pc = a^{+2})_{\perp};\ \mathtt{end}\, pc, st$

$\equiv$ *Definition machine instructions, table 1*
$\mathtt{var}\, pc, st;\ (pc = a)^{\top};\ pc, st := pc + 1, st +\!\!+ \langle e \rangle;$
$(\# st \geq 1)^{\top};\ s\,!\,\mathrm{last}\ .st;\ pc, st := pc + 1, \mathrm{front}\ .st;$
$(pc = a^{+2})_{\perp};\ \mathtt{end}\, pc, st$

$\equiv$:=-! *substitution and commutation, law 1(5)(7),*
and := *void* $^{\top}$*, law 3(2)*
$\mathtt{var}\, pc, st;\ (pc = a)^{\top};\ s\,!\,e;\ pc, st := pc + 1, st +\!\!+ \langle e \rangle;$
$pc, st := pc + 1, \mathrm{front}\ .st;\ (pc = a^{+2})_{\perp};\ \mathtt{end}\, pc, st$

$\sqsupseteq$:= *combination, law 1(4), and* := *identity, law 1(3)*
$\mathtt{var}\, pc, st;\ (pc = a)^{\top};\ s\,!\,e;\ pc := a + 2;$
$(pc = a^{+2})_{\perp};\ \mathtt{end}\, pc, st$

$\sqsupseteq$:=-`end` *combination, law 3(6),*
and change scope, law 3(7)
$\mathtt{var}\, pc, st;\ \Delta;\ \mathtt{end}\, pc, st;\ s\,!\,e$

$\equiv$ *change scope and* `end`-`var` *inverse, law 3(7)(5),*
and Δ *output, law 3(3)*
$s\,!\,e$

□

The implementation of input instruction $s\,?\,x$ is split into two parts. The first one, code A below, tests if there exists a message in shunt s; if there is no message, it jumps back to execute code A again. If there is a message in s, the second part, code I below, does input the oldest message and finishes storing it into variable x. To determine the execution time of the implementation of an input is an infeasible problem, since the arrival of messages into a shunt depends on the environment's behaviour; however, we can estimate the execution time of the input implementation if we know that the shunt is not empty.

Theorem 5 *Input Compilation*
Let $A = (\, a : \mathtt{TST}(s) \,[]\, a^{+1} : \mathtt{JPT}(a)\,)$
and $I = (\, a^{+2} : \mathtt{IN}(s) \,[]\, a^{+3} : \mathtt{ST}(x)).$
Then $\mathcal{C}\,(\, s\,?\,x,\ a,\ (A \,[]\, I),\ a^{+4}\,)\ .$
If s is not empty, then
$\mathcal{T}_{\mathcal{C}}(s\,?\,x) \quad = \quad \mathcal{T}(\mathtt{TST}) + \mathcal{T}(\mathtt{JPT}) + \mathcal{T}(\mathtt{IN}) + \mathcal{T}(\mathtt{ST})\ .$

Proof: We use a novel strategy in which the uniqueness of the fixed point for recursive equations plays an important role. Let us start by defining a function F that portrays the execution of the target code.

Let $C = (A \,[]\, I)$, $START = \mathtt{var}\, pc, st;\ (pc = a)^{\top}$,
$END = (pc = a^{+4})_{\perp};\ \mathtt{end}\, pc, st$,
$END_0 = (pc = a \vee pc = a^{+4})_{\perp};\ \mathtt{end}\, pc, st$
and $F(X) = START;\ A^{*};\ (X \lhd pc = a \rhd I^{*});\ END_0$.

Function $F(X)$ starts by executing code A. Depending on the value of the program counter

at the end of the execution of A, it proceeds either to execute code I or to invoke parameter program X. As all instructions in A take time, we conclude then that function F is time-guarded for variable X. From theorem 1, it follows that F has a unique fixed point. Our strategy consists in proving first that $s\,?\,x$ is a pre-fixed point of F, i.e. $s\,?\,x \sqsubseteq F(s\,?\,x)$, concluding by the strongest fixed point law, law 4 (4), that $s\,?\,x \sqsubseteq \mu X \bullet F(X)$. Then we proceed by proving that $(START;\ C^*;\ END)$ is a post-fixed point of F, i.e.
$F(START;\ C^*;\ END) \sqsubseteq (START;\ C^*;\ END)$, concluding by the weakest fixed point law, law 4 (2), that $\mu X \bullet F(X) \sqsubseteq (START;\ C^*;\ END)$. The desired result follows from the transitivity of the refinement relation. Complete proof of this theorem is presented in (Arenas 2000).
$\Box$

The strategy employed in the implementation of the input program can be used to prove the implementation of constructors that require to wait for the occurrence of an event, namely delay and timeout.

The code implementing the delay program Δd is divided into two parts: codes S and T. Execution of code S determines the time when delay Δd should finish: it is equal to the addition of the current time to the value of time parameter d, leaving the result on top of the evaluation stack. Code T compares the current time with the value at the top of the stack, in order to determine whether the delay has expired.

Theorem 6 *Delay Compilation*
Let $S = (\,a : \mathtt{TIM} \,[]\, a^{+1} : \mathtt{EVT}(d) \,[]\, a^{+2} : \mathtt{ADD}\,)$
and $T = (\,a^{+3} : \mathtt{DUP} \,[]\, a^{+4} : \mathtt{TIM} \,[]$
$\qquad a^{+5} : \mathtt{LT} \,[]\, a^{+6} : \mathtt{JPT}(a^{+3})\,)\,.$
Then $\mathcal{C}\,(\,\Delta d,\ a,\ (S \,[]\, T),\ a^{+7}\,)\,.$
$\mathcal{T}_{\mathcal{C}}(\Delta d) \ =\ d + \mathcal{T}(S) + \mathcal{T}(T)\,.$

Let us now turn to the implementation of compound processes. Sequential composition can be compiled componentwise, having as target code the assembly of its components.

Theorem 7 *Sequential Composition Compilation*
Let $\mathcal{C}\,(P,\ a,\ C,\ h)\ ,\ \mathcal{C}\,(Q,\ h,\ D,\ z)$
and $(L.C \cap L.D) = \emptyset\,.$
Then $\mathcal{C}\,(P;Q,\ a,\ (C \,[]\, D),\ z)\ .$
$\mathcal{T}_{\mathcal{C}}(P;Q)\ =\ \mathcal{T}_{\mathcal{C}}(P) + \mathcal{T}_{\mathcal{C}}(Q)\ .$

The compilation of a timed conditional includes an initial piece of code that evaluates the corresponding guard and then, depending on the result of the evaluation, chooses one of the participating programs.

Theorem 8 *Conditional Compilation*
Let $B = (\,a : \mathtt{EVB}(b) \,[]\, a^{+1} : \mathtt{JPF}(h)\,)\ ,$
$\mathcal{C}\,(P,\ a^{+2},\ C,\ z)\ ,\ \mathcal{C}\,(Q,\ h,\ D,\ z)\,,$
$(L.C \cap L.D) = \emptyset$ *and* $(L.B \cap L.C \cap L.D) = \emptyset\ .$
Then $\mathcal{C}\,(P \trianglelefteq b \trianglerighteq Q,\ a,\ (C \,[]\, B \,[]\, D),\ z)\ .$
$\mathcal{T}_{\mathcal{C}}(P \trianglelefteq b \trianglerighteq Q)\ =\ \mathcal{T}(\mathtt{EVB}) + \mathcal{T}(\mathtt{JPF}) +$
$\qquad \max(\mathcal{T}_{\mathcal{C}}(P), \mathcal{T}_{\mathcal{C}}(Q))\ .$

The iteration program is implemented by a piece of machine code that evaluates the guard. In case the guard holds, the body of the program is executed. Once it has terminated, it jumps back to repeat the whole process. To determine the execution time of the iteration program, it is necessary to know the upper bound on the possible number of iterations.

Theorem 9 *Iteration Compilation*
Let $B = (\, a : \mathtt{EVB}(b) \,[]\, a^{+1} : \mathtt{JPF}(z) \,)$, $J = (\, j : \mathtt{JP}(a) \,)$
$\mathcal{C}\,(P,\ a^{+2},\ C,\ j)$ *and* $(L.B \cap L.J \cap L.C) = \emptyset$.
Then $\mathcal{C}\,(\mathtt{while}(b,P),\ a,\ (B \,[]\, C \,[]\, J),\ z)$.
Let T be the maximum number of iterations of the program $\mathtt{while}(b,P)$*. Then*
$\mathcal{T}_{\mathcal{C}}(\mathtt{while}(b,P)) \;=\; T * (\mathcal{T}(B) + \mathcal{T}(C) + \mathcal{T}(J)) + \mathcal{T}(B)$.

The timeout $[P\ \rhd^s_d\ Q]$ is implemented by a machine program that monitors shunt s for at most d time units. If a message arrives in that period of time, the program jumps to execute the code associated to program P. After d time units, if a message has not arrived on shunt s, the program jumps to execute the code associated to program Q.

Theorem 10 *Timeout Compilation*
Let $S = (\, a : \mathtt{TIM} \,[]\, a^{+1} : \mathtt{EVT}(d) \,[]\, a^{+2} : \mathtt{ADD} \,)$,
$E = (\, a^{+3} : \mathtt{TST}(s) \,[]\, a^{+4} : \mathtt{JPF}(a^{+10}) \,)$,
$T = (\, a^{+5} : \mathtt{DUP} \,[]\, a^{+6} : \mathtt{TIM} \,[]\, a^{+7} : \mathtt{LT} \,[]$
$a^{+8} : \mathtt{JPF}(h) \,)$,
$J = (\, a^{+9} : \mathtt{JP}(a^{+3}) \,)$,
$M = (\, a^{+10} : \mathtt{STM}(s) \,[]\, a^{+11} : \mathtt{LE} \,[]\, a^{+12} : \mathtt{JPF}(h) \,)$,
$\mathcal{C}\,(P,\ a^{+13},\ B,\ z)$, $\mathcal{C}\,(Q,\ h,\ D,\ z)$,
$(L.B \cap L.D) = \emptyset$,
$(L.S \cap L.E \cap L.T \cap L.J \cap L.M \cap L.B \cap L.D) = \emptyset$ *and*
$C = (S \,[]\, E \,[]\, T \,[]\, J \,[]\, M \,[]\, B \,[]\, D)$.
Then $\mathcal{C}\,([P\ \rhd^s_d\ Q],\ a,\ C,\ z)$.
$\mathcal{T}_{\mathcal{C}}([P\ \rhd^s_d\ Q]) \;=\; d + \mathcal{T}(S) + \mathcal{T}(E) + \mathcal{T}(T) +$
$\mathcal{T}(J) + \mathcal{T}(M) + \max(\mathcal{T}_{\mathcal{C}}(P), \mathcal{T}_{\mathcal{C}}(Q))$.

Proof: Let us first explain the implementation of $[P\ \rhd^s_d\ Q]$, assuming that $\mathcal{C}\,(P,\ a^{+13},\ B,\ z)$ and $\mathcal{C}\,(Q,\ h,\ D,\ z)$. Code S refers to the evaluation of the timeout parameter; it reads the current time, and then adds to it the value of parameter d, leaving the result at the top of the evaluation stack. Code E determines whether there exists messages in the shunt. In case there are no messages in the shunt, code T compares the current time with the value at the top of the stack, to determine if a timeout has occurred. In case of a timeout, the program jumps to location h to execute piece of code D. If there is no timeout, the program proceeds with the execution of code J, which simply jumps to repeat code E. If there is a message in shunt s, code M determines if it arrived before the timeout; to do so, it obtains the time stamp of the oldest unread message, and compares it with the timeout value that is stored at the top of the evaluation stack. In case of the stamp being less than the timeout, code M jumps to location a^{+13}, where it continues with the execution of B. In case of the stamp being greater than the timeout value, a timeout has happened (although some messages could have arrived after the timeout, in which case they are not considered), code M jumps then to location h, the initial location of D.

In the proof, we follow a strategy similar to the one used for proving the input instruction. It starts with the definition of a function F that mimics the execution of code C and then exploits the uniqueness of its fixed point to get the desired result.

$$
\begin{array}{ll}
\text{Let} & START = M;\ (pc = a)^{\top}\ , \\
& END = (pc = z)^{\top};\ \mathtt{end}\, M\ , \\
& G(X) = E^*;\ [\,(\,T^*;\ (J^*;\ X \lhd pc = a^{+9} \rhd D^*)\,) \\
& \qquad\qquad \lhd pc = a^{+6} \rhd \\
& \qquad (\,M^*;\ (B^* \lhd pc = a^{+13} \rhd D^*)\,)\,] \\
& F(X) = START;\ S^*;\ G(X);\ END\ .
\end{array}
$$

Invocation of X in $F(X)$ is preceded by instructions that take time. Then, by Theorem 1, it follows that F has a unique fixed point. The proof strategy consists in showing first that $[P \rhd^s_d Q] \sqsubseteq F([P \rhd^s_d Q])$. Such proof follows by induction on time parameter d, using law 2(3). Then, according to the strongest fixed point law, it follows that $[P \rhd^s_d Q] \sqsubseteq \mu X \bullet F(X)$. The second part consists in showing that $(START;\ C^*;\ END)$ is a post fixed point of F, $F(START;\ C^*;\ END) \sqsubseteq (START;\ C^*;\ END)$. By the weakest fixed point law, it follows that $\mu X \bullet F(X) \sqsubseteq (START;\ C^*;\ END)$. The result arises from transitivity of the refinement relation.
□

Our compilation process restricts the compilation of deadline to the case in which it is the outermost operator. Let notation $\mathcal{C}_D(P,\ a,\ C,\ z)$ stand for $([D]P \sqsubseteq (\mathtt{var}\, pc, st;\ (pc = a)^{\top};\ [D]C^*;\ (pc = z)_{\perp};\ \mathtt{end}\, pc, st))$. The following theorem illustrates the compilation of deadline.

Theorem 11 *Deadline Compilation*
Let $\mathcal{C}(P, a, C, z)$ *and* $\mathcal{T}_C(P) \leq D$.
Then $\mathcal{C}_D(P,\ a,\ C,\ z)$.

$\mathcal{T}_C([D]P) = \mathcal{T}_C(P)$.

We are following an approach similar to (Fidge, Hayes and Watson 1999) by considering the compilation of deadline as a sort of annotation on the target code, annotation that will be used in the later stage of scheduling analysis.

5. An Abstract Model for Scheduling

This section summarises the abstract model for scheduling presented previously in (Arenas 2002a).

Let $\langle a_i, C_i, z_i \rangle$ be a collection of target codes working on machine components pc_i and st_i, for $i = 1, \cdots, n$; and $C = ([D_1]C_1^* \parallel [D_2]C_2^* \parallel \cdots \parallel [D_n]C_n^*)$ be a parallel program where execution of each participating process C_i has an associated deadline D_i. In this section we define an abstract model in which the implementation of C into a uniprocessor machine is represented as transformation $\mathcal{S}(C)$ and derive conditions to guarantee the validity of such a transformation.

Let us assume that the continuation set of the participating processes in C is pairwise disjoint. Process C_i is executed when its program counter belongs to the valid set of locations, i.e. $pc_i \in L.C_i$, and the processor has been allocated to it by the scheduler. To represent allocation, integer variable $id : \mathbb{Z}$ is employed. Variable id has value i when the processor has been allocated to process C_i; it has a value different from i when C_i is positioned out of its continuation set. The above restrictions can be summarised in the predicate $\mathcal{I}$, which can be seen as an invariant on id:

$$\begin{aligned}\mathcal{I} \;\widehat{=}\; & \bigwedge_{i=1}^{n} (id = i \;\Rightarrow\; pc_i \in L.C_i) \;\;\wedge \\ & (id \notin [1,n] \;\Rightarrow\; \bigwedge_{i=1}^{n} pc_i \notin L.C_i)\end{aligned}$$

The effect of scheduling process C_i is then represented by a transformation $\mathcal{S}^i(C_i)$ in which each labelled instruction l : `INSTR` of C_i is transformed by strengthening its guard by the condition $(id = i)$ and by executing a piece of code, called $CHANGE_{\mathcal{S}}$, at the end of instruction `INSTR`. Code $CHANGE_{\mathcal{S}}$ performs the allocation of the processor according to the defined policy.

Definition 5 *Transformation* $\mathcal{S}^i$

$$\begin{aligned}\mathcal{S}^i(l : \mathit{INSTR}) \;&=\; (\mathit{INSTR};\; CHANGE_{\mathcal{S}}) \\ &\qquad \lhd\, pc_i = l \wedge id = i \,\rhd\, II \\ \mathcal{S}^i(C \,[]\, D) \;&=\; (\mathcal{S}^i(C) \,[]\, \mathcal{S}^i(D))\,.\end{aligned}$$

For simplicity, we assume that scheduling instructions, i.e. the instructions of code $CHANGE_{\mathcal{S}}$, are instantaneous. Their execution time is included in the duration of the associated machine instructions.

The implementation of program C into a uniprocessor machine is represented by a loop that iterates while the processor is allocated to any of the processes. It proves to be useful to include in the loop the case when the processor is idle because no process is active. Such a condition is represented by condition $IdleCond$, that has the property $IdleCond \Rightarrow id \notin [1,n]$. Action $IDLE$ is executed when $IdleCond$ holds; it is defined as $IDLE \;\widehat{=}\; ([1]\Delta \lhd IdleCond \rhd II)$. The loop implementing program C is then defined as:

$$\begin{aligned}\mathcal{S}(C) \;\;\widehat{=}\;\; & c * ((\mathop{[]}_{i=1}^{n} \mathcal{S}^i(C_i)) \,[]\, IDLE) \\ & \text{where } \; c \;=\; ((\bigvee_{i=1}^{n} id = i) \;\vee\; IdleCond)\,.\end{aligned}$$

To verify that $\mathcal{S}(C)$ correctly implements C requires proof that the timing and computational requirements of C are respected by $\mathcal{S}(C)$. Such requirements are represented in properties (1) and (2) below:

- Let $\mathcal{T}_{\mathcal{S}}(C_i)$ denote the time spent executing process C_i in $\mathcal{S}(C)$. If each process C_i^* in C has an associated deadline D_i, then we have the obligation to prove that implementation $\mathcal{S}(C)$ respects those timing constraints, i.e.

$$\mathcal{T}_{\mathcal{S}}(C_i) \;\;\leq\;\; D_i \qquad \text{for } i = 1, \cdots, n\,. \tag{1}$$

- Let $(\overset{n}{\underset{i=1}{\|}} C_i^*)$ denote the parallel execution of the target processes in C, without considering their timing constraints. The fact that the computational behaviour of C is respected by $\mathcal{S}(C)$ is represented as follows:
$$(\overset{n}{\underset{i=1}{\|}} C_i^*)_{+(id,\,\mathcal{I})} \quad \sqsubseteq \quad (\bigvee_{i=1}^{n} id = i) * (\overset{n}{\underset{i=1}{[]}} \mathcal{S}^i(C_i))\,. \tag{2}$$

Refinement requires that both programs have the same alphabet, hence we use notation $(\overset{n}{\underset{i=1}{\|}} C_i^*)_{+(id,\,\mathcal{I})}$ to indicate that alphabet of $(\overset{n}{\underset{i=1}{\|}} C_i^*)$ has been extended with variable *id*, and invariant $\mathcal{I}$ holds. In general, for program variable v and predicate $\mathcal{V}$ on v, alphabet extension $P_{+(v,\mathcal{V})}$ can be defined as $\mathcal{V}^{\top};\ P;\ \mathcal{V}_{\perp}$.

We close this part by showing that if the application of transformation $\mathcal{S}$ to each machine instruction of C_i is an improvement, then refinement (2) holds directly.

Theorem 12 *Computational Behaviour*
Let l : `INSTR` be an instruction of target process C_i. If the refinement $(\mathcal{S}^i(\mathtt{INSTR});\ \mathcal{I}^{\top}) \sqsupseteq$ `INSTR` holds then property (2) is valid.

Proof: Let $F(X) = ((\overset{n}{\underset{i=1}{[]}} \mathcal{S}^i(C_i));\ X \lhd (\bigvee_{i=1}^{n} id_i = i) \rhd II)$. The right hand side of property (2) is equivalent to $\mu X \bullet F(X)$. Since each instruction of C_i takes time, function F is time guarded for variable X and the fixed point is unique. We exploit this situation, and use the strategy of proving that $(\overset{n}{\underset{i=1}{\|}} C_i^*)_{+(id,\mathcal{I})}$ is a pre-fixed point of function F. Let us begin with the case when arbitrary process C_j is active and has been chosen by the scheduler to be executed. Let $ASS \mathrel{\widehat{=}} (pc_j = l \wedge id = j \wedge m[l] = \mathtt{INSTR})$. Then

$$F((\mathop{\|}_{i=1}^{n} C_i^*)_{+(id,\mathcal{I})})$$

$\equiv$ Definition of F

$$((\mathop{[]}_{i=1}^{n} \mathcal{S}^i(C_i));\ (\mathop{\|}_{i=1}^{n} C_i^*)_{+(id,\mathcal{I})}) \lhd (\bigvee_{i=1}^{n} id_i = i) \rhd II$$

$\equiv$ *ASS* and elimination of conditional, Law 1(3)

$$\mathcal{S}^j(C_j);\ (\mathop{\|}_{i=1}^{n} C_i^*)_{+(id,\mathcal{I})}$$

$\equiv$ Definition of assembly, and elimination of conditional, Law 1(3)

$$\mathcal{S}^j(\mathtt{INSTR});\ (\mathop{\|}_{i=1}^{n} C_i^*)_{+(id,\mathcal{I})}$$

$\sqsupseteq$ Definition of alphabet extension and assumption $(\mathcal{S}^i(\mathtt{INSTR});\ \mathcal{I}^\top) \sqsupseteq \mathtt{INSTR}$

$$\mathtt{INSTR};\ (\mathop{\|}_{i=1}^{n} C_i^*);\ \mathcal{I}_\perp$$

$\sqsupseteq$ Expansion law 1(7)

$$((\mathop{\|}_{i=1}^{j-1} C_i^*) \parallel (\mathtt{INSTR};\ C_j^*) \parallel (\mathop{\|}_{i=j+1}^{n} C_i^*));\ \mathcal{I}_\perp$$

$\equiv$ Unfolding the loop, Law 1(5)

$$((\mathop{\|}_{i=1}^{j-1} C_i^*) \parallel (C_j^*) \parallel (\mathop{\|}_{i=j+1}^{n} C_i^*));\ \mathcal{I}_\perp$$

$\equiv$ *ASS* implies predicate $\mathcal{I}$ and definition of alphabet extension

$$(\mathop{\|}_{i=1}^{n} C_i^*)_{+(\{id\},\mathcal{I})}$$

The case when all programs have finished their execution, i.e. $\bigwedge_{i=1}^{n} pc_i \notin L.C_i$, follows in a straightforward manner, since both sides of the refinement reduce to skip.
$\square$

6. Cyclic Scheduling

Let $C = (C_1^* \parallel C_2^* \parallel \cdots \parallel C_n^*)$ be the parallel program to be implemented, where each C_i represents a target process that is neither periodic nor has an associated deadline. Here we apply the technique presented previously to prove the implementation of C into a uniprocessor machine using a *round robin scheduler*. In this model, the execution in the single-processor machine is represented by an interleaving of the instructions executed by the participating processes. The interleaving is represented by executing piece of code $CHANGE_{\mathcal{S}}$ after a communication instruction (machine instructions `IN` and `OUT`) and after the last instruction of C_i. Code $CHANGE_{\mathcal{S}}$ is defined as follows.

Definition 6 *Code* $CHANGE_{\mathcal{S}}$

$$
\begin{aligned}
&\mathit{CHANGE}_{\mathcal{S}} \;\widehat{=}\\
&\quad (id \in [1,n])^{\top};\ \mathtt{var}\,A;\\
&\quad A := \{k \mid 1 \le k \le n+1 \ \wedge\\
&\qquad (\exists j \bullet j = (id+k) \ \mathsf{mod}\ (n+1) \wedge j \neq 0 \wedge pc_j \in L.C_j)\};\\
&\quad ((id := (id + \min(A)) \ \mathsf{mod}\ (n+1)) \lhd A \neq \emptyset \rhd id := -1);\\
&\quad \mathtt{end}\,A\ .
\end{aligned}
$$

Code $\mathit{CHANGE}_{\mathcal{S}}$ assumes initially that variable id has a value in the interval of 1 to n. It then stores in temporal variable A the set of distances between id and the identifier of other active processes, in such a way that the cyclic order is maintained. Finally, it updates id with the identifier of the closest active process.

For target program $\langle a, C_i, z\rangle$, the effect of scheduling process C_i is represented by a transformation $\mathcal{S}^i_z(C_i)$. We rewrite definition 5 in the following way.

Definition 7 *Transformation* $\mathcal{S}^i_z$

$$
\begin{aligned}
\mathcal{S}^i_z(l : \mathtt{INSTR}) &= (\mathtt{INSTR};\ \mathit{CHANGE}_{\mathcal{S}}) \lhd pc_i = l \wedge id = i \rhd II\\
&\qquad \textit{if } \mathtt{INSTR} \in \{\mathtt{IN}(s), \mathtt{OUT}(s)\} \ \textit{or}\\
&\qquad\ l : \mathtt{INSTR} \equiv (l : \mathtt{INSTR};\ (pc_i = z)_{\perp})\\
&= \mathtt{INSTR} \lhd pc_i = l \wedge id = i \rhd II \qquad \textit{otherwise}\\
\mathcal{S}^i_z(C \,[]\, D) &= (\mathcal{S}^i_z(C) \,[]\, \mathcal{S}^i_z(D))\ .
\end{aligned}
$$

Following the abstract model, implementation of C in the single-processor machine is denoted by $\mathcal{S}(C)$. Execution of C finishes when all processes C_i terminate their execution; thus, it is not necessary to model in $\mathcal{S}(C)$ the idle case in which the processor is not executing processes, and *IdleCond* is defined to be false. Implementation $\mathcal{S}(C)$ for the case of the cyclic scheduler is then defined as follows: $\mathcal{S}(C) \;\widehat{=}\; (\bigvee_{i=1}^{n} id = i) * (\,[]_{i=1}^{n}\ \mathcal{S}^i_{z_i}(C_i))\ .$

Correctness of $\mathcal{S}(C)$ requires verification that the timing and computational constraints hold, i.e. to prove that equation (1) and refinement (2) are valid. Equation (1) follows trivially, since there are not timing constraints associated with processes in C. To prove refinement (2), we utilise theorem 12 and show that the application of transformation $\mathcal{S}^i_z$ to a machine instruction is an improvement on the original instruction.

As all assignments in $\mathit{CHANGE}_{\mathcal{S}}$ are instantaneous, execution of $\mathit{CHANGE}_{\mathcal{S}}$ is a refinement of skip.

Lemma 1 *If* $id \in [1,n]$ *then* $(\mathit{CHANGE}_{\mathcal{S}};\ \mathcal{I}^{\top}) \ \sqsupseteq\ II$.

Proof of refinement (2) follows directly from the next theorem. Application of $\mathcal{S}^i_z$ to a machine instruction is an improvement on the original instruction.

Theorem 13 $\mathcal{S}^i_z(l : \mathtt{INSTR});\ \mathcal{I}^{\top} \ \sqsupseteq\ l : \mathtt{INSTR}$

Proof: The case for instructions that do not execute code $\mathit{CHANGE}_{\mathcal{S}}$ follows directly from the definition of transformation $\mathcal{S}^i_z$. We illustrate the case when $pc_i = l$, $id = i$, $m_i[l] = \mathtt{INSTR}$, and code $\mathit{CHANGE}_{\mathcal{S}}$ is executed after $\mathtt{INSTR}$.

$\mathcal{S}^i_z(l : \texttt{INSTR});\ \mathcal{I}^\top$

$\equiv$ Definition of $\mathcal{S}^z_i$, Def. 7

$((\texttt{INSTR};\ \mathit{CHANGE}_\mathcal{S}) \lhd pc_i = l \wedge id = i \rhd \mathit{II});\ \mathcal{I}^\top$

$\equiv$ Assumption $pc_i = l$ and $id = i$

$(\texttt{INSTR};\ \mathit{CHANGE}_\mathcal{S});\ \mathcal{I}^\top$

$\sqsupseteq$ Associativity of sequential composition and Lemma 1

$\texttt{INSTR};\ \mathit{II}$

$\equiv$ II unit of sequential composition, Law 1(1), assumption $pc_i = l$ and $id = i$

$\texttt{INSTR} \lhd pc_i = l \rhd \mathit{II}$

$\equiv$ Definition of labelled instruction

$l : \texttt{INSTR}$

$\square$

7. Fixed Priority Scheduling with Pre-emption

Let $C = (\overline{C_1} \parallel \overline{C_2} \parallel \cdots \parallel \overline{C_n})$ be a parallel program where each $\overline{C_i} = [D_i]C_i^*$ is a process with deadline D_i and period T_i. Here we show how to apply the technique presented in section to validate the implementation of C.

In general, in priority-based scheduling schemes processes are assigned priorities such that, at all times, the process with the highest priority is executing (if it is not delayed or otherwise suspended). A scheduling scheme will therefore involve a priority assignment algorithm and a schedulability test, i.e. a means of confirming that the temporal requirements of the system are satisfied (Burns and Wellings 1997). We have selected the *deadline monotonic priority ordering* (Leung and Whitehead 1982) as our method for priority assignment. In this ordering, the fixed priority of a process is inversely related to its deadline: if $D_i < D_j$ then process C_i has higher priority than C_j. As the parallel operator is commutative and associative, we reorganise processes in C such that if $i < j$ then C_i has higher priority than C_j. Regarding schedulability tests, we have chosen the *worst-case response time* analysis (**?**). The worst-case response time of process C_i, denoted by R_i, is defined to be the longest time between the invocation of C_i and its subsequent completion. We assume that the set of processes C_is in C has passed the worst-case response time test, i.e. $R_i \leq D_i$ for $i = 1, \cdots, n$, and use a model of cooperative scheduling (Burns and Wellings 1997), in which each machine instruction is considered an atomic action and pre-emption is deferred at the end of executing an instruction.

Code $\mathit{CHANGE}_\mathcal{S}$ states the scheduling policy that the processor is always executing the active process with the highest priority. In order to implement such a policy, the following elements are included:

- Integer variable *clock* represents the clock of the system. For simplicity, it is assumed that *clock* is equal to zero when the execution of the system begins.
- Integer variables inv_i and com_i representing the number of invocations and completions of each process C_i respectively. A natural requirement for the system is that each invocation of a process is completed before its next invocation, i.e.

$\bigwedge_{i=1}^{n} (inv_i \geq com_i \geq inv_i - 1)$. That requirement holds under the restriction that the deadline of executing a process is less than its period, i.e. $D_i < T_i$. Process C_i is active when condition $inv_i > com_i$ holds; further, it is the highest-priority active process if condition $(inv_i > com_i \wedge \bigwedge_{j=1}^{i-1} \neg\, (inv_j > com_j))$ is true.

- Following Liu and Joseph (Liu and Joseph 2001), to verify that the implementation of C satisfies its real-time constraints, timers Ta_i and Tc_i are included for each process C_i. Timer Ta_i records the time that has elapsed since the last invocation of C_i. Timer Tc_i records the time spent in executing proper instructions of C_i.
- Pre-emption cannot happen in the middle of the execution of a machine instruction; therefore, it is necessary to record those processes that were activated during the execution of an instruction, as well as the time of activation. These values will be used to update the corresponding variables once the execution of the instruction finishes. Auxiliary variable $newact_i$ is true if process C_i was released during the execution of a machine instruction, and t_i records the value of the clock at the moment of the activation.

As initial condition, it is assumed that the system starts execution at time zero, and at that time all participating processes are active. Condition *INIT* represents such situation.

$$\begin{aligned} INIT \;\widehat{=}\;\; & clock = 0 \wedge id = 1 \wedge \\ & \bigwedge_{i=1}^{n} (inv_i = 1 \wedge com_i = 0 \wedge pc_i = a_i \wedge \\ & \qquad \neg\, newact_i \wedge Ta_i = 0 \wedge Tc_i = 0)\,. \end{aligned}$$

Processes are released by the system according to their period. Condition *TRIGGER* represents the release of processes.

$$\begin{aligned} TRIGGER \;\widehat{=}\;\; & \bigwedge_{i=1}^{n} (clock \bmod T_i) = 0 \;\Rightarrow \\ & (inv_i, newact_i, Ta_i, Tc_i := inv_i + 1, false, 0, 0 \\ & \qquad \lhd \bigvee_{j=1}^{n} \neg\, (inv_j > com_j) \rhd \\ & \qquad\qquad inv_i, newact_i, t_i := inv_i + 1, true, clock)\,. \end{aligned}$$

If the processor was idle ($\bigvee_{j=1}^{n} \neg\, (inv_j > com_j)$), we activate the ready processes immediately; otherwise, the activation is deferred until the end of the current machine instruction.

Let us now define code $CHANGE_{\mathcal{S}}$, which is attached to each machine instruction and performs the actions associated with the scheduler: update the timers, activate new processes and achieve pre-emption.

Definition 8 *Code* $CHANGE_{\mathcal{S}}$
Code $CHANGE_{\mathcal{S}}(i, a, z, \texttt{INSTR})$ *where i denotes the identifier of a process C_i, a stands for the initial location of process C_i, z corresponds to the final location of C_i, and* `INSTR`

denotes a machine instruction of C_i, is defined as follows.

$$
\begin{array}{l}
\mathit{CHANGE}_{\mathcal{S}}(i, a, z, \texttt{INSTR}) \ \widehat{=} \\
\quad Tc_i := Tc_i + \mathcal{T}(\texttt{INSTR}); \quad \textit{(Update } Tc_i\textit{)} \\
\quad (\bigwedge_{j=1}^{n} \mathit{newact}_j \Rightarrow (\mathit{newact}_j, Ta_j, Tc_j := \mathit{false}, \mathit{clock} - t_j, 0)); \ \textit{(New processes)} \\
\quad (\bigwedge_{j=1}^{n} (\mathit{inv}_j > \mathit{com}_j \land \neg\, \mathit{newact}_j) \Rightarrow Ta_j := Ta_j + \mathcal{T}(\texttt{INSTR})); \quad \textit{(Update Ta)} \\
\quad (\mathit{com}_i, pc_i := \mathit{com}_i + 1, a \lhd pc_i = z \rhd \mathit{II}); \qquad \textit{(End of execution of } C_i\textit{)} \\
\quad \mathit{id} := \min\{j \in [1, n] \mid \mathit{inv}_j > \mathit{com}_j \land \bigwedge_{k=1}^{j-1} \neg\, (\mathit{inv}_k > \mathit{com}_k)\} \quad \textit{(Pre-emption)} \\
\qquad\qquad\qquad\qquad \lhd \bigvee_{j=1}^{n} \mathit{inv}_j > \mathit{com}_j \rhd \mathit{II}
\end{array}
$$

In above definition, timer Tc_i counts the time spent executing instructions of C_i, hence it is incremented by the duration of INSTR. In case a new process C_j was activated during the execution of INSTR, timer Ta_j should count the fraction of time its activation has been deferred. In case other processes C_j are active, timer Ta_j is incremented by the duration of instruction INSTR. If execution of C_i finishes, the counter com_i is incremented and the program counter is located to the initial position, so that it will be ready for the next invocation. Finally, variable *id* is updated with the identifier of the active process with the highest priority.

For target program $\langle a, C_i, z\rangle$, the effect of scheduling process C_i is represented by a transformation $\mathcal{S}^i_{(a,z)}$ in which the guard of each instruction l : INSTR of C_i is strengthened by the condition $(id = i)$, as is the case in the abstract model, and code $\mathit{CHANGE}_{\mathcal{S}}(i, a, z, \texttt{INSTR})$ is executed at the end of INSTR. We rewrite definition 5 as follows.

Definition 9 *Transformation* $\mathcal{S}^i_{(a,z)}$

$$
\begin{array}{lcl}
\mathcal{S}^i_{(a,z)}(l : \texttt{INSTR}) & = & (\texttt{INSTR}; \\
& & \quad \mathit{CHANGE}_{\mathcal{S}}(i, a, z, \texttt{INSTR})) \\
& & \qquad \lhd pc_i = l \land id = i \rhd \mathit{II} \\
\mathcal{S}^i_{(a,z)}(C \,[]\, D) & = & (\mathcal{S}^i_{(a,z)}(C) \,[]\, \mathcal{S}^i_{(a,z)}(D))\ .
\end{array}
$$

Following the abstract model, implementation of C in the single-processor machine is denoted by $\mathcal{S}(C)$. The processor is idle when no process is active; thus, *IdleCond* is defined as $id \notin [1, n]$. As a result, the guard of loop $\mathcal{S}(C)$ is reduced to true. Implementation $\mathcal{S}(C)$ for the case of the pre-emptive fixed-priority scheduler is defined as follows:

$$
\mathcal{S}(C) \quad \widehat{=} \quad \mathit{INIT}^{\top};\ \mathit{true} * ((\mathop{[]}_{i=1}^{n} \mathcal{S}^i_{(a_i,z_i)}(C_i)) \,[]\, \mathit{IDLE})\ .
$$

Correctness of $\mathcal{S}(C)$ requires verification that the timing constraints and the computational behaviour of C, equation (1) and refinement (2), are preserved.

7.1. Verifying Computational Behaviour

In order to prove that $\mathcal{S}(C)$ preserves the computational behaviour of C, we resort to theorem 12 and simply show that application of $\mathcal{S}^i_{(a,z)}$ to a machine instruction of C_i is an improvement on such an instruction. Let us define the auxiliary variables $\overline{v}$ introduced in the implementation of C and its associated invariant: $\overline{v} \;\widehat{=}\; \{inv, com, clock, Ta, Tc, newact, t\}$ and $\mathcal{V} \;\widehat{=}\; \bigwedge_{i=1}^{n} (inv_i \geq com_i \geq inv_i - 1)$.

The following theorem illustrates that any labelled instruction of process C_i is improved by the application of transformation $\mathcal{S}^i_{(a,z)}$.

Theorem 14 *Computational Behaviour*
$\mathcal{S}^i_{(a,z)}(l : \mathtt{INSTR});\ (\mathcal{I} \wedge \mathcal{V})^{\top} \;\sqsupseteq\; l : \mathtt{INSTR}$.

Proof: The proof follows lines similar to those of proof of theorem 13, since all instructions in $CHANGE_{\mathcal{S}}$ are instantaneous.
□

7.2. Verifying Timing Constraints

To verify timing constraints, we follow an approach presented previously (Liu and Joseph 2001), which relies on the value of timers Ta_i and Tc_i to determine the duration of process C_i. The time spent executing process C_i in $\mathcal{S}(C)$ corresponds to the value of timer Ta_i after executing last instruction of C_i; hence, proving equation (1) is equivalent to prove that timer Ta_i is less than the deadline associated to process C_i. As all processes share a common release, it can be shown from general scheduling theory that if all processes meet their first deadline then they will meet all future ones. As a result, we concentrate our attention to the case $com_i = 0$ and $inv_i > com_i$.

Assume that all processes with higher priorities than C_i, i.e. processes C_j for $j = 1, \cdots, i-1$, have met their deadline so far. In the worst case, the time spent on executing processes of higher priority than C_i is given by the formula: $Comphp_i \;\widehat{=}\; \sum_{j=1}^{i-1} com_j * \mathcal{T}(C_j) + \sum_{j=1}^{i-1} (inv_j - com_j) * Tc_j$, where $\mathcal{T}(C_i)$ denotes the worst-case execution time of process C_i^*.

The next lemma summarises the main properties of Ta_i and Tc_i.

Lemma 2 *Timer Properties*
(1) $Tc_i \;\leq\; \mathcal{T}(C_i)$.
(2) *If* $com_i = 0$ *and* $inv_i > com_i$ *then*
$Ta_i \;=\; Comphp_i + Tc_i$.
(3) *If* $com_i = 0$ *and* $inv_i > com_i$, *then*
$Comphp_i + \mathcal{T}(C_i) \;\leq\; R_i$

We close this section by proving property (1), i.e. the time spent executing arbitrary process C_i is less than its associated deadline.

Theorem 15 *Timing Constraints*
Let $C = (\overline{C_1} \parallel \overline{C_2} \parallel \cdots \parallel \overline{C_n})$ *be a parallel target program where each process* $\overline{C_i} = [D_i]C_i^*$ *has associated deadline* D_i *and period* T_i, *such that* $D_i < T_i$. *If the set of processes in C passes the worst-case response analysis, i.e.* $R_i \leq D_i$, *then the following property holds for implementation* $\mathcal{S}(C)$:
$inv_i > com_i \;\Rightarrow\; Ta_i \leq D_i \qquad$ *for* $i = 1, \cdots, n$.

Proof:

$$\begin{array}{ll}
& com_i = 0 \ \wedge \ inv_i > com_i \\
\Rightarrow & \textit{Lemma 2} \\
& (Ta_i \leq Comphp_i + \mathcal{T}(C_i)) \ \wedge \ (Comphp_i + \mathcal{T}(C_i) \leq R_i) \\
\Rightarrow & \textit{Transitivity of} \leq \textit{relation} \\
& Ta_i \ \leq \ R_i \\
\Rightarrow & \textit{Assumption } R_i \leq D_i \\
& Ta_i \ \leq \ D_i
\end{array}$$

□

8. Conclusions

Many authors have shown that unique fixed points arise naturally in real-time contexts when restricting the model to allow the progress of time (Davies and Schneider 1993). In this paper we have taken advantage of this characteristic to verify the implementation of a real-time language using the refinement-algebra approach to compilation.

Implementation of classical sequential constructors (such as assignment, sequential composition, conditional and iteration) has followed lines similar to those of the untimed case. The novelty in our work consisted in devising a strategy for proving the implementation of constructors that are required to wait for the occurrence of an event (input, delay and timeout), as presented in (Arenas 2002b).

The approach to prove correctness of compiling specification using algebraic laws was originally suggested by Hoare in (Hoare 1991, Hoare, He Jifeng and Sampaio 1993). Hoare's work was accomplished in the context of the ProCoS project (Bowen, Hoare, Langmaack, Olderog and Ravn 1996) and has inspired several investigations. Notable is the work of Müller-Olm (Müller-Olm 1997), that describes the design of a code generator translating the language of while programs — extended by communication statements and upper-bound timing — to the Inmos Transputer. Emphasis is put on modularity and abstraction of the proofs, which is achieved by constructing a hierarchy of increasingly more abstract views of the Transputer's behaviour, starting from bit-code level up to assembly levels with symbolic addressing. In (Hale 1994), a compilation is defined for a real-time sequential language with memory-mapped input and output commands. Both the source and target languages are modelled in the Interval Temporal Logic, and a set of algebraic laws are derived in a way similar to that presented here. The compilation process is simplified by representing the compilation of communication processes as a compilation of assignments to port variables.

Also influenced by Hoare's work, but using an alternative approach, Lermer and Fidge define compilation for real-time languages with asynchronous communication (Lermer and Fidge 2002). Their semantic model is based on the real-time refinement calculus of Hayes where communication is achieved by shared variables, and the language offers delay and deadline constructors. Our intermediate target language is very close to their target code, also modelled as a subset of the source language. The operation of composition of machine programs is achieved by means of an operation for merging loops, similar to our model of execution of machine programs. Although there are many similarities between the two studies, this approach does not define a compiling relation. Instead, a set of "compilation laws" are derived, where each law looks like a rule of the well-known refinement calculus of Morgan.

having a high-level of abstraction while reasoning about schedulers enables one to model different types of schedulers and to investigate their effect on different programs. The potential of our model for scheduling has been illustrated by instancing it with two types of schedulers: a round-robin scheduler and a fixed-priority scheduler. In both cases, we have derived and verified properties that guarantee the correctness of the implementation. In the case of the round-robin scheduler, we have verified that the implementation preserves the computational behaviour of the parallel program. In the case of the fixed-priority scheduler, we have verified that the implementation preserves the computational behaviour as well as the timing constraints of the parallel program.

There are some limitations in our scheduling work: the programs analysed have some restrictions on their structure such as not having deadlines (cyclic scheduler) or being periodic (priority-based scheduler). Intended future work includes studying more general forms of scheduling such as dynamic schedulers.

References

Arenas, A. E.: 2000, *Implementation of an Asynchronous Real-Time Programming Language*, PhD thesis, Oxford University Computing Laboratory.

Arenas, A. E.: 2001, A Specification-Oriented Semantics for Asynchronous Real-Time Programming, *Proceedings of CLEI'2001, XXVII Latin American Conference on Informatics*.

Arenas, A. E.: 2002a, An Abstract Model for Scheduling Real-Time Programs, *Formal Methods and Software Engineering*, Vol. 2495 of *Lecture Notes in Computer Science*, Springer.

Arenas, A. E.: 2002b, An Algebraic Approach for Compiling Real-Time Programs, *Electronic Notes in Theoretical Computer Science* **68**(5).

Botella, G. and Goslin, J.: 2000, The Real-Time Specification for Java, *IEEE Computer* .

Bowen, J. P., Hoare, C. A. R., Langmaack, H., Olderog, E.-R. and Ravn, A. P.: 1996, A ProCoS II Project Final Report: ESPRIT Basic Research Project 7071, *Bulletin of the European Association for Theoretical Computer Science (EATCS)* **59**, 76–99.

Burns, A. and Wellings, A. J.: 1997, *Real-Time Systems and Programming Languages*, Addison-Wesley.

Davies, J. and Schneider, S.: 1993, Recursion Induction for Real-Time Processes, *Formal Aspects of Computing* **5**(6), 530–553.

Fidge, C., Hayes, I. and Watson, G.: 1999, The Deadline Command, *Software* **146**(2), 104–111.

Hale, R. W. S.: 1994, Program Compilation, *Towards Verified Systems*, Vol. 2 of *Real-Time Safety Critical Systems*, Elsevier, pp. 131–146.

Hoare, C. A. R.: 1991, Refinement Algebra Proves Correctness of Compiling Specifications, *in* C. C. Morgan and J. C. P. Woodcock (eds), *3rd Refinement Workshop*, Workshops in Computing, Springer-Verlag, pp. 33–48.

Hoare, C. A. R. and He Jifeng: 1998, *Unifying Theories of Programming*, Prentice Hall Series in Computer Science.

Hoare, C. A. R., He Jifeng and Sampaio, A.: 1993, Normal Form Approach to Compiler Design, *Acta Informatica* **30**(8), 701–739.

Kumar, K. N. and Pandya, P. K.: 1993, ICSP and its Relationship with ACSP and CSP, *Foundations of Software Technology and Theoretical Computer Science*, Vol. 761 of *Lecture Notes in Computer Science*, Springer.

Lermer, K. and Fidge, C.: 2002, A Formal Model of Real-Time Program Compilation, *Theoretical Computer Science* **282**(1), 151–190.

Leung, J. Y. T. and Whitehead, J.: 1982, On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks, *Performance Evaluation North Holland* **2**, 237–250.

Liu, Z. and Joseph, M.: 2001, Verification, Refinement and Scheduling of Real-Time Programs, *Theoretical Computer Science* **253**(1).

Mitschele-Thiel, A.: 2001, *Systems Engineering with SDL*, John Wiley and Sons.

Müller-Olm, M.: 1997, *Modular Compiler Verification*, Vol. 1283 of *Lecture Notes in Computer Science*, Springer-Verlag.

Table 1: The Target Language

$\texttt{LD}(x)$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle x \rangle$
$\texttt{ST}(x)$	$\widehat{=}$	$(\#st \geq 1)^{\top};\ pc, st, x := pc + 1, \texttt{front}.st, \texttt{last}.st$
$\texttt{EV}(e)$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle e \rangle$
$\texttt{EVB}(b)$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle R_{\mathbb{B}}.b \rangle$
$\texttt{EVT}(d)$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle R_{Time}.d \rangle$
$\texttt{ADD}$	$\widehat{=}$	$(\#st \geq 2)^{\top};$ $pc, st := pc + 1, \texttt{front2}.st \mathbin{+\!\!+} \langle \texttt{last2}.st + \texttt{last}.st \rangle$
$\texttt{SUB}$	$\widehat{=}$	$(\#st \geq 2)^{\top};$ $pc, st := pc + 1, \texttt{front2}.st \mathbin{+\!\!+} \langle \texttt{last2}.st - \texttt{last}.st \rangle$
$\texttt{LE}$	$\widehat{=}$	$(\#st \geq 2)^{\top};$ $pc, st := pc + 1, \texttt{front2}.st \mathbin{+\!\!+}$ $\langle 1 \lhd \texttt{last}.st \leq \texttt{last2}.st \rhd 0 \rangle$
$\texttt{LT}$	$\widehat{=}$	$(\#st \geq 2)^{\top};$ $pc, st := pc + 1, \texttt{front2}.st \mathbin{+\!\!+}$ $\langle 1 \lhd \texttt{last}.st < \texttt{last2}.st \rhd 0 \rangle$
$\texttt{JP}(l)$	$\widehat{=}$	$pc := l$
$\texttt{JPF}(l)$	$\widehat{=}$	$(\#st \geq 1)^{\top};$ $pc, st := (l \lhd \texttt{last}.st = 0 \rhd pc + 1), \texttt{front}.st$
$\texttt{JPT}(l)$	$\widehat{=}$	$(\#st \geq 1)^{\top};$ $pc, st := (l \lhd \texttt{last}.st = 1 \rhd pc + 1), \texttt{front}.st$
$\texttt{DUP}$	$\widehat{=}$	$(\#st \geq 1)^{\top};\ pc, st := pc + 1, st \mathbin{+\!\!+} \langle last(st) \rangle$
$\texttt{OUT}(s)$	$\widehat{=}$	$(\#st \geq 1)^{\top};\ s\,!\,\texttt{last}.st;\ pc, st := pc + 1, \texttt{front}.st$
$\texttt{IN}(s)$	$\widehat{=}$	$(s \neq \langle\rangle)^{\top};\ \texttt{var}\,x;$ $s\,?\,x;\ pc, st := pc + 1, st \mathbin{+\!\!+} \langle x \rangle;\ \texttt{end}\,x$
$\texttt{TST}(s)$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle 1 \lhd s = \langle\rangle \rhd 0 \rangle$
$\texttt{STM}(s)$	$\widehat{=}$	$(s \neq \langle\rangle)^{\top};\ pc, st := pc + 1, st \mathbin{+\!\!+} \langle stamp(s) \rangle$
$\texttt{TIM}$	$\widehat{=}$	$pc, st := pc + 1, st \mathbin{+\!\!+} \langle t \rangle$ where $t \in [t_\alpha, t_\omega]$ and t_α, t_ω stand for the time when starts and finishes the execution of the instruction