TRANSFORMATION METHODS FOR IMPLEMENTING CORRECT REAL-TIME PROGRAMS

M JOSEPH

Rapporteur: G M Megson

Implementing Real-Time Systems by Transformation

Abha Moitra*, Mathai Joseph**

August 25 1989

Abstract

A common method for designing a complex system is to start with a specification of its functional properties and then to refine this specification into units that may be directly implemented. By itself, such a method is inadequate for implementing a real-time program because, in addition to the functional properties defined in the specification, the implementation must satisfy real-time constraints. In this paper, we suggest that a real-time system can be designed in two stages. First, the functional specifications are used to refine the design into successively smaller units; during this refinement, it is merely assumed that the program is implemented with *sufficient* computing resources (i.e. processors of adequate speed, enough memory, etc.) for all the timing constraints to be met. In the next stage, this implementation is transformed into one for a specific system and it is determined if the real-time constraints can be satisfied. We show how such transformations can be done in general and then illustrate their use in producing implementations for systems with priorities for processes, pre-emptible executions, etc.

Keywords: real-time programming, implementation, transformation, correctness, feasibility

1 Introduction

A typical real-time system is linked to its external environment through sensors from which it receives data, and actuators to which it must send data. In many systems, the arrival of an input induces a timing constraint on the despatch of an output. Different kinds of real-time system are distinguished by the nature of this constraint; e.g. in a *hard real-time* system no input must be lost and each output must take place within a limited interval. Many such systems have *periodic* processing requirements: inputs appear at regular intervals, the data must then be processed and outputs sent periodically in response. The structure of the real-time software often mirrors this: low-level processes service the input devices periodically, and higher-level processes perform computations based on the inputs and send output to lower-level processes for despatch to the devices.

*General Electric CRD, Schenectady, N.Y., U.S.A.; part of this work done while at Odyssey Research Associates under U.S. Air Force RADC contract F30602-86-C-0115. **University of Warwick, U.K. (supported by Research Grant GR/D/73881 from the Science and Engineering Research Council)

A common method for designing any complex system is to decompose a concise higher level specification into more detailed lower level units, repeating the procedure until units which are sufficiently simple to be directly implemented are obtained. If the design is *compositional* [10], then provided that the implementation of these units meet their specifications, the implementation as a whole will meet the specification of the system. A strategy of this kind is used in many problem-solving procedures: for example in design by stepwise refinement, or in the refinement steps used in formal program design methods.

The specification of a real-time program will define its functional properties and the time constraints that must be satisfied by its implementation. If the design strategy described above is followed, both of these requirements will need to appear in the specification of smaller units. But while the functional properties may be decomposed using a standard method, there is no obvious way in which the timing constraints can similarly be decomposed. And the closer a design gets to an implementation, the harder it is to consider the functional and timing requirements of a design simultaneously. Moreover, both of these requirements will need to be examined afresh after every program modification.

1.1 Implementation by Transformation

As an alternative method, assume that a real-time system is designed in two stages. First, the functional specifications are used to refine the design into successively smaller units; during this refinement, no attempt is made to tailor the design to ensure that the timing properties will be satisfied. Instead, it is merely assumed that the program is implemented with *sufficient* computing resources (i.e. processors of adequate speed, enough memory, etc.) for all the timing constraints to be met. Let this be called the *maximum resource implementation*. If the functional specification and the design refinement are correct, then such an implementation must exist.

The next stage of the design is to transform the maximum resource implementation into an implementation on a specific system. Such transformations will not in general be unique and may not always be possible, for example if the system lacks the resources needed to satisfy the time constraints of the specification.

Assume that in the maximum resource implementation, each process is implemented on a separate processor [8]; normally, the processors will not be fully utilised. We may then determine if two processes can be combined to execute on a single processor so that each process still meets its timing constraints. This will allow the number of processors needed for the implementation to be reduced by one. The procedure can be repeated until the number of processors (or, in general, the size of any other resource) is reduced until it is equal to that available in the system, or until no further reduction is possible if the timing constraints for each process are to be satisfied. The method is in some ways similar to the implementation technique called "process inversion" which is used in Jackson System Development [4] though there no particular attention is paid to the execution times of processes. Note that the question of finding an *optimal* implementation (i.e. one using the fewest resources) is in general NP-complete. The procedure merely enables the amount of resource needed for a particular program to be reduced by combining processes to share resources. The choice of processes to be combined is left to the designer, so there may be cases where more reduction is possible using a different choice.

Processes can be combined by transforming their execution schedules for execution on one processor. Such transformations must preserve the sequential ordering of each process execution and the dependencies induced by communication between processes. In this paper, we define such transformations and show how they can be used to model implementations with different scheduling disciplines. The transformations are commutative and associative, and so can be applied in any order.

2 Program Model

Let the real-time program P consist of n processes $P_1, P_2, \dots, P_n$. Assume that the processes are *cyclic*, i.e. that each process contains a loop which is executed forever. Thus the structure of process P_i can be represented as a sequence of commands

$$P_i :: C_1, C_2, \dots, C_{j-1}, \{C_j, \dots, C_k\}^*$$

where $Init(P_i) = C_1, C_2, \dots, C_{j-1}$ is the initialisation section and $Cycle(P_i) = C_j, \dots, C_k$ is the loop. For convenience we will at times consider $Init(P_i)$ and $Cycle(P_i)$ to be sets in which each command C_i is unique.

In general, a command C will be executed after any one of several different commands have completed execution; similarly, execution of any one of several different commands may immediately follow the execution of C. For example, C may follow or precede the execution of a nondeterministic command containing several commands as alternatives. Let $Prec(C)$ be the set of sequential commands which are the *immediate* predecessors of C in all possible executions: in any execution, if C is not the first command in the program then exactly one of the commands in $Prec(C)$ will be executed immediately before C. Similarly, let $Post(C)$ be the set of sequential commands which are the *immediate* successors of C in all possible executions. Then the following invariant assertions *INV*1 can be made about $Init(P_i)$ and $Cycle(P_i)$:

$$\begin{aligned}
&\forall C \in Init(P_i),\ Prec(C) \subseteq Init(P_i)\\
&\exists C \in Prec(C_j),\ C \in Init(P_i)\\
&\exists C \in Prec(C_j),\ C \in Cycle(P_i)\\
&\forall C \in Cycle(P_i),\ C \neq C_j,\ Prec(C) \subseteq Cycle(P_i)\\
&\forall C \in Cycle(P_i),\ Post(C) \subseteq Cycle(P_i)\\
&\forall C' \in Prec(C),\ Prec(C') \cap Prec(C) = \emptyset
\end{aligned}$$

Assume initially that all communication between processes is synchronous and let $Match(C_1, C_2)$ be a predicate which is true if C_1 and C_2 are syntactically matching communications in different processes.

$Pred(C)$, the set of all commands that may immediately precede C, can be defined as

$$Pred(C) = \{C_1 \mid C_1 \in Prec(C_2) \wedge Match(C, C_2)\} \cup \{C_1 \mid C_1 \in Prec(C)\}$$

For a different communication mechanism, e.g. asynchronous communication, the definition of *Pred* can be changed appropriately.

3 System Model

Assume that the real-time system is connected to a number of input and output devices. For simplicity, let each device either send inputs to the system or receive outputs from the system. Assume that the real-time program is structured in levels such that processes at the same level do not communicate with each other.

The input and output devices are treated as processes at level $L0$. Processes at level $L1$ are device processes and there is a one-to-one 'connection' between level $L0$ and level $L1$ processes. Processes at level $L2$ are the servers. A level $L2$ process may be connected to more than one level $L1$ process; however, it never initiates any computation (i.e., it is *reactive*). Assume initially that there are just two levels in the program, $L1$ and $L2$: generalisation to more levels with the same constraints on process connections is straightforward.

In the class of programs we shall initially consider, there is one important constraint: each communication command must match syntactically with exactly one communication statement in another process. There is then no distinction between syntactic and semantic matching [1] but note that this still allows processes to have nondeterministic behaviours. The restriction enables programs to be statically analyzed.

The real-time constraints of the problem are specified as deadlines, or as timed input-output relations. Assume that all the input devices and some output devices are cyclic. (Any devices which receive outputs produced as responses to inputs and which can accept successive outputs with some minimum interval can also be considered cyclic in the worst-case). The deadlines for level $L1$ input processes and output processes come from the characteristics of their associated devices and can be converted into deadlines over the execution of the communication commands within these processes. The deadlines for level $L2$ processes can then be derived from these deadlines, and will once again translate into deadlines over communication commands.

Informally, the system's real-time requirement can then be defined briefly as (i) receive all inputs, (ii) send outputs at regular intervals to cyclic output devices, and (iii) send outputs within deadlines to acyclic output devices. The worst case load on the system from a single device arises when successive inputs from the device arrive with the minimum separation time. The worst case load on the system as a whole arises when all the input devices produce their individual worst case loads and the first inputs from all processes arrive simultaneously [6].

Example: In the following program, inputs from two devices are received by processes $P1$ and $P2$, and outputs are sent to a device by process $P3$. Processes $P1$, $P2$ and $P3$ are level 1 processes. Process $P4$ is a level 2 process which receives inputs from processes $P1$ and $P2$ and then produces an output for process $P3$. The deadline for process $P1$ is specified by requiring that the statement Receive(x) be executed every d_1 units. Similarly, d_2 and d_3 specify deadlines for processes $P2$ and $P3$. So in the program, each deadline is the maximum time between successive iterations of the loop.

```
P1 :: Init1;
      *[$d_1$ : Receive(x); P4 ! x ]

P2 :: Init2;
      *[$d_2$ : Receive(y); P4 ! y ]

P3 :: Init3;
      *[$d_3$ : P4 ? z; Send(z)]

P4 :: Init4;
      *[ P1 ? x → P2 ? y; P3 ! f(x,y)
         □
         P2 ? y → P1 ? x; P3 ! f(x,y)]
```

Definition : In a *maximum resource implementation,* execution of a program is never suspended for lack of computing resources.

Thus, for example, each process in a program will be executed on a separate processor and there will be sufficient memory to keep all the program's variables immediately accessible at all times during the execution. In a maximum resource implementation of this program, there will be four processors for the four processes P_1 to P_4 and we assume that such an implementation will meet the deadlines. Other implementations of this program which meet the deadlines may also be possible and so, for example it is possible to ask under what conditions the program can be implemented on fewer than four processors and still meet all its deadlines.

The rest of this paper develops a method by which such questions, and the broader issues of limited resource execution, can be addressed. We shall assume that the implementations to be considered have the properties defined in the following two policies.
Policy 1: All of the commands of a process are executed on a single processor.
Policy 2: A processor is never idle if it can execute a command from any of the processes that run on it.

4 Semantics of Limited Resource Execution

The execution time of a command depends on the availability of resources such as processors, memory, communication channels etc. There are bounds, *Lower*(C) and *Upper*(C) (which is the maximal requirement), to the resources needed for any command C. If the available resources exceed the upper bound, the execution time of C is the same as that with the maximal resource. Execution of a command becomes impossible if the available resource is smaller than the lower bound for the command.

The effect of limiting the resources for the execution of a command C can be described by associating each behaviour with a set of *wait intervals.* If I is a wait interval for a behaviour F, then, in the corresponding execution, one or more component commands of C are suspended during the interval I and wait intervals are included in the execution times of those components. If a sequential component S in C is suspended in an interval I, then the resources allocated to S for the interval I can be re-assigned to another command running concurrently with C.

Let us call a command C' the *functional equivalent* of C if they use the same variables and if each behaviour of C is a possible behaviour of C' (but possibly with different times). For example, if C is an assignment command then C' is the same assignment command but executed at different times due to the limited availability of resources. The semantics $\Gamma(C)$ of a command C is a set of quadruples $< R, C', F, W >$, where F is a behaviour of C, W is the set of (command,wait interval) pairs associated with F, R is a description of the resource availability and C' is a functional equivalent of C for which R is the upper bound of the resource requirement. The command C' is called an *R-representation* of C. The resource availability R may vary between $Lower(C)$ and $Upper(C)$ (it is assumed that there is a total ordering on the resource availability descriptions of C). The set of quadruples in $\Gamma(C)$ with R as the first component is the *R-semantics* of C.

If C is a primitive command, then $Lower(C)$ and $Upper(C)$ are the same, and C' is C. The set W may be nonempty even for a primitive command if its termination depends on cooperation from other commands.

4.1 Combinators and limited resources

Individual commands in a program are composed together using combinators. For example, in the sequential composition $C_1; C_2$ the combinator ';' composes the two commands C_1 and C_2 while in the parallel composition $C_1 \| C_2 \| \ldots \| C_n$, the n individual commands $C_1 \ldots C_n$ are composed using the parallel combinator $\|$.

Let $\diamond$ be an n-ary combinator and

$$C \triangleq C_1 \diamond \cdots \diamond C_n$$

The *R-semantics* of C is obtained by combining

$$< R_i, C'_i, F_i, W_i > \in \Gamma(C_i)$$

for $i \in [1, n]$, such that the F_i's are consistent with respect to $\diamond$ and an execution of C with R can be obtained by executing C_1 with R_1, C_2 with R_2, and so on.

Consider the case of the parallel composition combinator. A limited processor behaviour of a parallel command can be obtained from the limited processor behaviours of its constituent commands, noting that the constituent commands may contain further parallel sub-components. Let each constituent command be partitioned so that if a sub-component does not contain processes its variables are in the same element of the partition, while if it has nested processes each such process is assigned to a different element of the partition.

Given such a partitioning, let $\pi_1 \triangleq \{X_1, \ldots, X_p\}$ be the p-partition of a command c_1 and $\pi_2 \triangleq \{Y_1, \ldots, Y_q\}$ the q-partition of a command c_2. Then the k-combination of π_1 and

π_2, written $\pi_1 \oplus_k \pi_2$, is defined if $p + q \geq k$ and is a set of k-partitions, each of the form $\{Z_1, \ldots, Z_k\}$ where $Z_i \in \pi_1 \cup \pi_2$ or Z_i is the union of two or more elements of $\pi_1 \cup \pi_2$.

Obviously, if $p + q = k$, then for example $\pi_1 \oplus_k \pi_2 = \pi_1 \cup \pi_2$.

It is usual to take the specific case of the processors when considering resource limitations but the approach can readily be generalized to any resource whose requirement can be determined from the syntax of a command. Limited processor semantics of several other combinators have been defined [5] and further details are available elsewhere [3].

5 Resource Annotation

A command serves as a unit for the allocation of resources in an implementation. Execution of a command requires a set of computational resources (e.g. processors, memory etc.) and each command can be annotated with its resource requirements. Typically, a resource requirement is defined in terms of the size and the time for which it is required.

It is common practice to define resource requirements in relation to a specific architecture or system. Initially, we shall consider all resource requirements in terms of a *canonical* system S_0 with processor type P_0, memory M_0 etc. We shall assume that these requirements can be transformed into those for a particular target architecture. This will allow us to ignore the particular characteristics of a system during the resource annotation of each command. Then, during the transformation, the requirements for each command can be redefined to those for a particular system, retaining the possibility that the architecture may contain heterogeneous systems by using different transformations for the resources required by different components in the system.

Resource requirements tend to be dominated by the requirements for processors: a computation cannot proceed at all unless it has at least one processor for its execution. Given the necessary processors, the progress of a computation is dependent on the provision of its other resource requirements. These requirements are often not fixed, but will always have a minimum below which the computation cannot proceed. Assuming that it is possible to place upper bounds on loop iterations, communication delays and memory use, there will also be a maximum above which the computation cannot use further resources.

We will use the following notation to annotate the resource requirements of a command. $Length(C, r, s)$ is the time required to execute the command C when s units of the resource r are available. We assume that increasing the resource size never decreases the performance of the system. This is captured as the following axiom.

$$s1 \geq s2 \Rightarrow Length(C, r, s1) \leq Length(C, r, s2)$$

We will sometimes abbreviate $Length(C, r, s)$ to $Length(C)$, usually when the resource is the processor and the size is 1.

In a *maximum resource implementation*, if a process (and therefore any command within a process) is able to execute, it will be executed. That is, the execution of a process is never suspended for the lack of sufficient resources or for any interleaving introduced during implementation.

6 Feasible Schedules

The implementation of a program enables a command C_{hk} in the program to be executed on some processor. Let an executed command, or *e-command*, c_{ij} be the j-th program command to be executed by processor i. The execution of c_{ij} spans the non-empty interval $[c_{ij}/start, c_{ij}/end)$. Let $Length(c_{ij}) = c_{ij}/end - c_{ij}/start$. A *processor schedule* S_i is a sequence of e-commands $c_{i1}, \dots, c_{im}$ such that

$$\forall c_{ij} \in S_i,\ 1 \leq j < m,\ c_{ij}/end \leq c_{i(j+1)}/start$$

A *schedule SCH* is a sequence of processor schedules.

For any e-command c_{ij} in a schedule, let $F(c_{ij})$ be a function which returns the corresponding command in the *program* and $Process(c_{ij})$ be a function which returns the identifier of the process containing that command. For a cyclic execution, there may be several e-commands $c_{pi}, \dots, c_{pj}$ which map to the same program command $F(c_{pi})$.

If c_{ij} is an e-command in a schedule SCH, then $Pred(F(c_{ij}))$ is a set of commands *in the program* at least one of which will be executed immediately before every execution of c_{ij}: $Pred(F(c_{ij}))$ is a statically defined set. We can define a similar function $Prev(c_{ij})$ which is the set of *e-commands* which immediately precede the execution of c_{ij}: there are at most two e-commands in $Prev(c_{ij})$ and they are chosen dynamically.

Let $Trunc(S_i, j-1)$ return the sequence consisting of the first $j-1$ elements of S_i and $S/Process(c_{ij})$ be the restriction of the execution sequence S to those elements which are in $Process(c_{ij})$. Then

$$\begin{aligned} Prev(c_{ij}) = & \{Last(S') \mid S' = Trunc(S_i, j-1)/Process(c_{ij}) \wedge \#S' \geq 1\} \\ & \cup \{Last(S') \mid Match(c_{ij}, c_{hk}) \\ & \qquad \wedge S' = Trunc(S_h, k-1)/Process(c_{hk}) \\ & \qquad \wedge \#S' \geq 1\} \end{aligned}$$

where the predicate *Match* has been used with e-commands.

The dependencies between e-commands induce a relation which defines a *proper schedule* with the following invariant properties *INV*2:

1. $\forall S_i \in SCH,\ \forall c_{ij} \in S_i,\ Length(c_{ij}) = Length(F(c_{ij}))$.

2. $\forall S_i \in SCH,\ \forall c_{ij} \in S_i,\ Prev(c_{ij}) \neq \emptyset \Rightarrow c_{ij}/start \geq max\{Prev(c_{ij})/end\}$

3. For each process p_i,

$$\exists S_j \in SCH, \exists k, 1 \leq k \leq \#S_j : \mathcal{F}(Rest(S_j, k)/p_i) = (Cycle(p_i))^+$$

$\mathcal{F}$ is like F but operates on sequences, and the $^+$ denotes one or more cycles.

We have assumed that since the real-time system is cyclic, satisfaction of any real-time constraint can be verified by considering finite runs of such a system. The first execution of each process starts with the initialisation section and then continues with the execution of the cycle; subsequent executions of the cycle then take place normally. The worst case processing load on the system occurs when all the inputs arrive simultaneously and will repeat after a time equal to the least common multiple of the time periods of the inputs. The system must meet its deadlines for the worst case load, including any initialisation before cyclic processing.

A *feasible* schedule is a schedule which is *proper* and which always meets its deadlines. If d_{ij} is the deadline for command $F(c_{ij})$, then it is the deadline for each execution of c_{ij}.

$$\begin{aligned} Feasible(SCH) = {} & Proper(SCH) \\ & \wedge \forall S_i \in SCH,\ \forall c_{ij} \in S_i,\ \exists k > j : F(c_{ik}) = F(c_{ij}) \\ & \quad \Rightarrow \exists h > j : F(c_{ih}) = F(c_{ij}) \\ & \qquad \wedge\ c_{ih}/end - c_{ij}/end \leq d_{ij} \end{aligned}$$

7 Transformations

The annotated command description can be used to map a program to an implementation. If the commands are annotated with maximum resource requirements, this implementation is the *maximum resource implementation.* We shall assume that for the implementation on the canonical system, each process meets its deadlines. Practical systems will usually have fewer resources than the maximum needed and will execute at speeds lower than those of the canonical system. So any practical implementation of the program will be a *reduced* implementation, when compared with the maximum resource implementation. Unless the maximum resource implementation meets the system's deadline, no reduced implementation can meet the deadlines.

A reduced implementation is produced by transforming a feasible schedule into another feasible schedule for a system with fewer resources. In general, this would mean starting with the maximum resource implementation, since that can be defined directly from the program and the characteristics of the real-time environment. A transformation is *valid* if, given a feasible schedule, it produces a new feasible schedule.

In these transformations, we shall assume (i) that the speeds of the actual processors can be defined relative to that of the canonical processor, (ii) that the actual processors are identical and (iii) that before and after transformation, Policy 2 is followed, i.e. all the commands of a process will be executed on the same processor. Assumption (ii) is not essential but allows the presentation to be simplified. Assumption (iii) follows common practice as the overheads of executing a process on different processors can be substantial.

7.1 Transformations and target architectures

A valid transformation converts a feasible schedule into another feasible schedule. Notwithstanding the assumptions (i) - (iii), which are made largely for simplification, transformations should not be viewed as depending on either the components or the structure of the target architecture. Rather, transformations are like functions which operate on *schedules*: it is the choice of transformation which is dictated by the target architecture. For example, a transformation which combines two processor schedules into a single processor schedule can be applied to reduce the number of processors required in an implementation provided that, in doing so, an appropriate choice is made of the processes to be combined. Repeated use of such a transformation can then convert, for example, a multi-processor implementation to a single processor implementation.

In the transformations we shall describe, a common operation will be to merge two processor sequences into one sequence, for execution on a single processor. Typically, such a merge will require some shifting in time of the elements of one or both sequences, and the communication dependencies between schedules may then cause shifts of the elements of other sequences. There may be more than one way in which one processor sequence is merged into another, so the merge will in general require nondeterministic choices. Let each process have a unique identifier and for any schedule S_k let the set of identifiers for the processes in the schedule be $Ids(S_k) = \{x \mid \exists y \in S_k : Process(y) = x\}$. Then the merge of sequences S_i, S_j in a schedule $< S_1, \ldots, S_n >$ is represented by

$$SeqMerge(< S_1, \ldots, S_n >, Ids(S_i) \cup Ids(S_j))$$

Informally, $SeqMerge(< S_1, \ldots, S_n >, Ids(S_i) \cup Ids(S_j))$ is a set of schedules in each of which the e-commands executed on the original S_i and S_j are merged into a single sequence.

More formally, *SeqMerge* is defined as:

$SeqMerge(< S_1, \ldots, S_n >, Ids(S_i) \cup Ids(S_j))$
$\quad = \{SCH \mid SCH$ satisfies the following 5 conditions$\}$

1. $SCH = < S'_1, \ldots, S'_{i-1}, S'_{i+1}, \ldots, S'_{j-1}, S'_{j+1}, \ldots, S'_n, S'_{ij} >$
2. $Proper(SCH)$
3. $1 \leq k \leq n,\ k \neq i,\ k \neq j,\ ShiftedSeq(S'_k, S_k)$
   $\wedge\ Ids(S'_{ij}) = Ids(S_i) \cup Ids(S_j)$
   $\wedge\ \forall x \in Ids(S'_{ij}),\ ShiftedSeq(S'_{ij}/x, S_i/x) \vee ShiftedSeq(S'_{ij}/x, S_j/x)$
4. $ShiftedSch(SCH, SCH') \wedge SCH \neq SCH' \Rightarrow \neg Proper(SCH')$
5. SCH satisfies Policies 1 and 2

where

$ShiftedSeq(S_i, S_j) =$ if $\#S_i = \#S_j$
$\qquad \wedge \forall\, 1 \leq h \leq \#S_i,\ F(c_{ih}) = F(c_{jh})$
$\qquad \wedge\ Length(c_{ih}) = Length(c_{jh})$
$\qquad \wedge c_{ih}/start \geq c_{jh}/start$
$\quad$ then true
$\quad$ else false

$ShiftedSch(< S_1, .., S_n >, < S'_1, .., S'_n >) = \forall i, 1 \leq i \leq n,$
$\qquad ShiftedSeq(S_i, S'_i) = true$

Condition (4) specifies that each processor sequence is shifted no more than necessary, and condition (5) that no processor in the resulting schedule is idle if it can execute some command from any of the processes running on it.

For notational convenience we will also define *SeqMerge* on a set of schedules as follows.

$$\begin{aligned}&SeqMerge(SchSet, Ids(S_i) \cup Ids(S_j))\\ &= \{y \mid \exists x \in SchSet : \exists z_1, z_2 \in x : Ids(S_i) = Ids(z_1)\\ &\qquad \wedge Ids(S_j) = Ids(z_2) \wedge y \in SeqMerge(x, Ids(S_i) \cup Ids(S_j))\}\end{aligned}$$

7.2 Properties of transformations

Schedule transformations can be shown to be commutative and associative. Hence the order of application of transformations can be ignored and a particular reduced implementation can be obtained by choosing the appropriate transformations in any order.

Theorem 1 (Commutativity)*: If SC is the schedule* $< S_1, \ldots, S_n >$ *with* $n \geq 2$ *then*

$$SeqMerge(SC, Ids(S_i) \cup Ids(S_j)) = SeqMerge(SC, Ids(S_j) \cup Ids(S_i))$$

Proof: Follows trivially, from the definition of set union. □

Theorem 2 (Associativity)*: If SC is the schedule* $< S_1, \ldots, S_n >$ *with* $n \geq 3$ *then*

$$\begin{aligned}&SeqMerge(SeqMerge(SC, Ids(S_i) \cup Ids(S_j)), Ids(S_i) \cup Ids(S_j) \cup Ids(S_k)) =\\ &SeqMerge(SeqMerge(SC, Ids(S_j) \cup Ids(S_k)), Ids(S_i) \cup Ids(S_j) \cup Ids(S_k)) \qquad (1)\end{aligned}$$

Proof: Let the left hand side of eq. (1) be called *LHS*. We can rewrite *LHS* as follows.

$$\begin{aligned}&LHS\\ &= \{A \mid \exists T \in SeqMerge(SC, Ids(S_i) \cup Ids(S_j))\\ &\qquad : A \in SeqMerge(T, Ids(S_i) \cup Ids(S_j) \cup Ids(S_k))\}\end{aligned}$$

Define Q as follows.

$Q = \{P \mid P$ satisfies the following 5 conditions$\}$

1. $P = < S'_1, \ldots, S'_{i-1}, S'_{i+1}, \ldots, S'_{j-1}, S'_{j+1}, \ldots, S'_{k-1}, S'_{k+1}, \ldots, S'_n, S'_{ijk} >$
2. $Proper(P)$
3. $1 \leq h \leq n, h \neq i,\ h \neq j,\ h \neq k\ ShiftedSeq(S'_h, S_h)$
   $\wedge Ids(S'_{ijk}) = Ids(S_i) \cup Ids(S_j) \cup Ids(S_k)$
   $\wedge \forall x \in Ids(S_{ijk}), ShiftedSeq(S'_{ijk}/x, S_i/x) \vee ShiftedSeq(S_{ijk}/x, S_j/x)$
   $\vee ShiftedSeq(S_{ijk}/x, S_k/x)$
4. $ShiftedSch(P, P') \wedge P \neq P' \Rightarrow \neg Proper(P')$
5. P satisfies Policies 1 and 2

We will now show that $Q = LHS$.

Part 1. The elements of LHS satisfy the 5 properties satisfied by the elements of Q. Hence $LHS \subseteq Q$.

Part 2. Consider an arbitrary schedule R in Q. Let R' be a copy of the schedule R in which the elements belonging to the original S_k appear in a new schedule; the start and end times of all the e-commands remain the same. By construction it follows that R' is proper and satisfies Policy 1. We can obtain a new schedule R'' from R' by shifting the e-commands to the left to satisfy Policy 2. By this construction it follows that $R'' \in SeqMerge(SC, Ids(S_i) \cup Ids(S_j))$. It then follows that $R \in SeqMerge(R'', Ids(S_i) \cup Ids(S_j) \cup Ids(S_k))$. Hence $R \in LHS$.

So $Q = LHS$. A similar argument shows that the right hand side of eq. (1) = Q. □

8 Processor Reduction Without Pre-emption

Merging processor schedules allows the number of processors needed for an implementation to be reduced. The choice of processor and the properties of the new processor schedule are governed by Policies 1 and 2. By definition, the maximum resource implementation satisfies these policies. *SeqMerge* defines the *set* of schedules obtained by merging two sequences and in most practical cases it is only a subset of these schedules that will be of interest. In fact, different subsets are obtained by considering different restrictions on the ways in which sequences can be merged.

In this section we shall consider transformations which follow Policies 1 and 2 in reducing the number of processors, e.g. using a procedure such as *SchMerge*:

$$\begin{aligned} &SchMerge\,(< S_1, \ldots, S_n >) \\ &= \{< S_1 >\} \text{ if } n = 1 \\ &\quad \{T \mid \exists R \in SeqMerge(< S_1, S_2 >, Ids(S_1) \cup Ids(S_2)) \\ &\qquad : T \in SchMerge(R)\} \text{ otherwise} \end{aligned}$$

Theorem 3 *There is a feasible schedule S' of P on one processor under Policies 1 and 2* iff $\exists S'' \in SchMerge(S)$ *such that* $Feasible(S'')$.

Proof: Define Q as follows.

$Q = \{P \mid P \text{ satisfies the following 5 conditions}\}$
1. $P = < S_{12 \ldots n} >$
2. $Proper(P)$
3. $\forall x \in Ids(S_{12 \ldots n}),\ \exists\, i : ShiftedSeq(S_{12 \ldots n}/x, S_i/x)$
4. $ShiftedSch(P, P') \wedge P \neq P' \Rightarrow \neg Proper(P')$
5. P satisfies Policies 1 and 2

By an argument similar to that in proof of Theorem 2, $SchMerge(S) = Q$. Thus $SchMerge(S)$ generates all possible schedules for one process satisfying the above 5 properties. Since any feasible schedule on one processor satisfying Policies 1 and 2 must also satisfy these same 5 properties, the theorem follows. □

8.1 Reduction from n to m processors

The method used to reduce the number of processors to 1 can be generalised to reduce the processor requirement from n to m processors. However, here an initial choice must be made of the processes to be merged (and more generally of the processor schedules that are to be merged). The method has therefore to be used in conjunction with a strategy for choosing processor schedules to be merged. Many simple strategies can be used; however, the problem of finding an optimum strategy is NP-complete.

An obvious method is at each step to merge the schedules of the least loaded processors, and to repeat this until the number of processors has been sufficiently reduced. However, this might lead very soon to a schedule in which no further merging is possible when it is clear that with a different order of merging further progress could have been made. An improvement on this strategy would be at each step to merge processor schedules so that in the resulting schedule the remaining processors are least loaded. In the first case the processes can be chosen by simple inspection (in linear time) while in the second the complexity will be $O(n^2)$ for n processor schedules.

8.2 Priorities

So far, all processor schedule merging has been attempted assuming that each process runs to completion or until it is blocked, and that when two processor schedules are merged, the choice of which process is executed first is made arbitrarily. If processes are assigned priorities, a new strategy is possible: processes are still run to completion or until blocked but when two processor schedules are merged, the process with the highest priority is always executed before processes of lower priority. This will reduce the size of the set of schedules produced by *SeqMerge* at each step but it is still far from the commonly-known and more efficient method of pre-empting the execution of lower priority processes.

9 Processor Reduction With Pre-emption

When higher priority processes pre-empt lower priority processes, each process is run to completion, until it is blocked, or until it is pre-empted. The 'unit' of scheduling is then dynamically determined and this can convert an infeasible implementation into a feasible implementation in many cases. We consider two ways in which this may be done.

9.1 Arbitrary Pre-emption

Priorities are used for the pre-emptive allocation of the processor: i.e. if a process of lower priority is executing, it is suspended when it is necessary for a process of higher priority to be executed and then its execution is resumed. This is identical to the familiar pre-emptive priorities implemented in typical real-time processor hardware. While we had earlier discussed the execution of each process in terms of the execution of a sequence of its commands, it now becomes necessary to define the smallest non-pre-emptable unit of a process. In most cases, this is a single machine instruction (or even part of such an instruction) as it is at this level that the state of the process can be saved on pre-epmtion and later restored.

If a program is represented solely in terms of its machine instructions, many of the merging operations described here could become prohibitively expensive in terms of computation times. It would therefore be more useful for the program to be represented at more than one level of detail: the highest level of detail, or the largest unit of program command could be used wherever pre-emption is not necessary and the points of pre-emption could be defined more precisely by using the lowest level of representation.

Scheduling with pre-emptive priorities results in a number of switches between processes (context switches) and has some associated overhead. This can be modelled in the transformations by adding some 'cost' to each switch. However, as in practice, an apparently feasible schedule may be found to be infeasible when the overhead of context switching is taken into account.

9.2 Voluntary Pre-emption

It is often possible to reduce the overhead of context switching to a minimum by determining points in the execution when processes can voluntarily relinquish the processor *when needed* by other higher priority processes. This will typically have the effect of delaying any particular point of pre-emption but it can also reduce the number of context switches.

Merging schedules with points of voluntary pre-emption marked will further reduce the size of the resulting set of schedules. However, determining the best points for voluntary pre-emption is an issue that must be solved at both the language and the implementation level [7].

10 Discussion and Conclusions

The paper discusses techniques that can be used to develop a *method* for the design and implementation of verifiable real-time systems. It focusses on the problem of converting a correct 'abstract' implementation of a program into a correct 'concrete' implementation. There has been other work on language-based static program analysis for single processor implementations [2], and on pre-emptive scheduling methods for accommodating resource limitations [9]. Our concern has been to show that a variety of such techniques can be accommodated in a formal design method.

As part of a continuing programme of research into real-time system design at the University of Warwick, the work described here is to be extended and equipped with tools to support the transformations: this phase of the work is being supported by the Information Engineering Directorate Advanced Technology Programme.

11 Acknowledgement

We are grateful to Asis Goswami for suggesting improvements to the notation.

References

[1] K.R. Apt, N. Francez, and W.-P. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–384, 1980.

[2] A.D. Stoyenko E. Klingerman. Real-time euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, 1986.

[3] A. Goswami and M. Joseph. A semantic model for the specification of real-time processes. In *CONCURRENCY 88, Lecture Notes in Computer Science 335*, pages 292–306, Springer-Verlag, Heidelberg, 1988.

[4] M. Jackson. *System Development*. Prentice-Hall International, 1983.

[5] M. Joseph and A. Goswami. What's 'real' about real-time systems? In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 78–85, Huntsville, Alabama, 1988.

[6] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprocessing in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[7] A. Moitra. *Voluntary Preemption : A Tool in the Design of Hard Real-Time Systems.* Technical Report, Odyssey Research Associates, Ithaca, NY 14850, 1988.

[8] A. Salwicki and T. Müldner. On the algorithmic properties of concurrent programs. In *Lecture Notes in Computer Science 125*, pages 169–197, Springer-Verlag, Heidelberg, 1981.

[9] W. Zhao, K. Ramamritham, and J.A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(6):949–960, 1987.

[10] J. Zwiers, W.-P. de Roever, and P.E.M de Boas. Compositionality and concurrent networks: soundness and completeness of a proof system. In *Lecture Notes in Computer Science 194*, Springer-Verlag, Heidelberg, 1985.

DISCUSSION

Rapporteur: Dr G M Megson

The presentation described a transformational approach for converting a real time system specification into an implementation with limited physical resources. The central theme was to produce a feasible schedule of processes using a maximum resource requirement and then to move towards the limited resources by a sequence of transformations from one feasible schedule to another.

There were two main threads to the discussion comprising of comments on the validity of the transformation process and queries regarding the actual specification of the limited resources.

The first point regarding the transformation was the explicit mechanism through which the maximum resource implementation could be verified with the specification, and the way that the limited resource schedule could be validated with respect to timing requirements againist the max resource implementation. It was acknowledged that if the transformations were valid that an automatic method (e.g compiler) should be capable enforcing the rules and guaranteeing correctness so that no explicit validation by a designer would be required. However, explicit verification and validation was an integral part of defining and proving that transformations were in fact correct during development. A related point was raised as to the generality of such transformations, the argument being that each system would require different resource requirements in order to maintain the time deadlines. The danger being that new transformations would have to be developed for each new application making it impossible to develop automatic transformation methods. It was emphasized that the transformation approach was based on moving from feasible schedule to feasible schedule a method which made no explicit use of the problem structure. A possible drawback was that a large number of possible schedules (although this was finite) existed and that an optimal schedule involved the solution of an NP- complete problem.

The comments on the resource specification can be summarized in three points. The first problem being that the available resources needed to be specified and the transformation method mapped from a maximum resource solution to that available. If the specified limited resources were not sufficient to solve the problem and still meet all the timing constraints the method would be useless. In particular all the feasible schedules would have to be examined to find out that a particular transformation sequence did not produce a solution. It was suggested that it may be more appropriate to generate the minimum resources required after moving from one schedule to another. Related to this last point it was remarked that the way in which resources were quantified would have great bearing on the success of the transformation. For example if processing speed was used as a metric it may be possible to meet all the deadlines with a CRAY, but how many transputers would we need to achieve the same task. Thus the resource requirement depended not only on the computing power available but on how processors were interconnected - this would be difficult to quantify in general. As an extension to the problems quantifying physical components it was remarked that the approach assumed static load balancing in that processes once allocated to a processor did not move. The point being that dynamic load balancing may allow the physical resource requirements to be reduced even further. In particular a problem may not be feasible with a certain resource specification using static load balancing but possible with dynamic load balancing. It was concluded that the basic approach was a useful step towards achieving these more ambitious goals.