Understanding Concurrent Programs using Program Transformations

E. J. Younger
Centre for Software Maintenance Ltd
Unit 1P, Mountjoy Research Centre
Durham, DH1 3SW

M. P. Ward
Computer Science Department
Science Labs, South Rd
Durham DH1 3LE

Abstract

Reverse engineering of concurrent real-time programs with timing constraints is a particularly challenging research area, because the functional behaviour of a program, and the non-functional timing requirements, are implicit and can be very difficult to discover. In this paper we present a significant advance in this area, which is achieved by modelling real-time concurrent programs in the wide spectrum language WSL. We show how a sequential program with interrupts can be modelled in WSL, and the method is then extended to model more general concurrent programs. We show how a program modelled in this way may subsequently be "inverse engineered" by the use of formal program transformations, to discover a specification for the program. (We use the term "inverse engineering" to mean "reverse engineering achieved by formal program transformations").

1 Introduction

This paper describes extensions to the inverse engineering techniques developed at the University of Durham, to enable these to be applied to programs with interrupts and concurrency. An example of the use of the methods to derive a specification for a simple concurrent system is given.

The paper is organised as follows. In sections 2 and 3 we give a brief introduction to the WSL language and transformation theory. Then in Section 4 we show how to model a real-time interrupt-driven program in the WSL language. In Section 5 we show how the principles may be extended to model more general concurrent programs. Finally, we use this information to derive a specification of the example program in Section 6.

2 The Language WSL

In this section we give a brief introduction to the language WSL [2,9,14] the "Wide Spectrum Language", used in Ward's program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. For brevity we will only define the language constructs used in this paper.

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification. In [10,11,15] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [12,16] program transformations are used in reverse engineering and program comprehension tasks, including the derivation of concise specifications from program source code.

2.1 Sequences

Sequences are denoted by angled brackets: $s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i \,..\, j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i \,..\, j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i \,..]$ as an abbreviation for $s[i \,..\, \ell(s)]$. $\mathit{reverse}(s) = \langle a_n, a_{n-1}, \ldots, a_2, a_1 \rangle$, $\mathit{head}(s)$ is the same as $s[1]$ and $\mathit{tail}(s)$ is $s[2 \,..]$.

The concatenation of sequences s_1 and s_2 is denoted $s_1 \mathbin{+\!\!+} s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

2.2 Specification Statement

The statement $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.\mathbf{Q}$ assigns new values $x'_1, \ldots, x'_n$ to the variables $x_1, \ldots, x_n$ such that the formula $\mathbf{Q}$ is true. If there are no values which satisfy $\mathbf{Q}$ then the statement aborts (does not terminate). For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2.x')$ halves x if it is even and aborts if x is odd. If the sequence contains one variable then the sequence brackets may be omitted, for example: $x := x'.(x = 2.x')$. The assignment $x := x'.(x' = t)$ where x' does not occur in t is abbreviated to $x := t$.

2.3 Unbounded Loops

Statements of the form $\underline{\mathbf{do}}$ $\mathbf{S}$ $\underline{\mathbf{od}}$, where $\mathbf{S}$ is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form $\underline{\mathbf{exit}}(n)$ (where n is an integer, *not* a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow $\underline{\mathbf{exit}}$s which leave a block or a loop other than an unbounded loop. This type of structure is described in [6,8].

3 Program Refinement and Transformation

The WSL language includes both specification constructs, such as the general assignment, and programming constructs. One aim of our program transformation work is to develop programs by refining a specification, expressed in first order logic and set theory, into an efficient algorithm. This is similar to the "refinement calculus" approach of Morgan et al [4,7]; however, our wide spectrum language has been extended to include general action systems and loops with multiple exits. These extensions are essential for our second, and equally important aim, which is to use program transformations for reverse engineering from programs to specifications.

Refinement is defined in terms of the denotational semantics of the language: the semantics of a program $\mathbf{S}$ is a function which maps from an initial state to a final set of states. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. For programs $\mathbf{S}_1$ and $\mathbf{S}_2$ we say $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write $\mathbf{S}_1 \leq \mathbf{S}_2$, if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leq \mathbf{S}_2$ and $\mathbf{S}_2 \leq \mathbf{S}_1$ then we say $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ and write $\mathbf{S}_1 \approx \mathbf{S}_2$. Equivalence is thus defined in terms of the external "black box" behaviour of the program. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [9] and [14] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations. We use the term *abstraction* to denote the opposite of refinement: for example the "most abstract" program is the non-terminating program **abort**, since any program is a refinement of **abort**.

A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [9] and [14] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations.

4 Modelling Interrupt-Driven Programs in WSL

WSL has no notations for parallel execution or interrupts. We chose not to add such notations to the language, since this would complicate the semantics enormously and render virtually all our transformations invalid. Consider, for example, the simple transformation:

$$x := 1;\ \underline{\mathbf{if}}\ x = 1\ \underline{\mathbf{then}}\ y := 0\ \underline{\mathbf{fi}}\ \approx\ x := 1;\ y := 0$$

which is trivial to prove correct in WSL. However, this transformation is not universally valid if interrupts or parallel execution are possible, since an interrupting program could change the value of x between the assignment and the test. Instead, our approach is to *model* the interrupts in WSL by inserting a procedure call at all the points where the program could be interrupted. This procedure tests if an interrupt did actually occur, and if so it executes the interrupt routine, otherwise it does nothing. Although this increases the program size somewhat, the resulting program is written in pure WSL and all our transformations can be applied to it.

One of our aims in transforming the resulting WSL program is to move the interrupt calls through the body of the program, and collect them together and merge them as far as possible. The body of the main program would then be essentially sequences of statements from the original program, separated by the processing of any interrupts which occurred during their execution.

In order to model time within WSL we add a variable *time* to the program which is incremented appropriately whenever an operation is carried out which takes some time. We can then reason about the response times of the program by observing the initial and final values of this variable. We can also model the times when interrupts occur by providing an input sequence consisting of *pairs* of values $\langle t, c\rangle$ where t the time at which the interrupt occurs and c is the character. Naturally we should insist that the sequence of t values be monotonically increasing. Such a sequence can model input from an external device, a concurrent process, or even a hardware register. We make this explicit in our model of the program: the array (or equivalently, sequence) *input* consists of pairs of times and characters to represent the inputs, and is sorted by times. The interrupt routine tests the *time* variable against the time value associated with the first element of the input sequence to see if that interrupt is now "due". If so, then it removes a pair from the head of the sequence and processes the result. The program is modelled as follows:

$$\begin{array}{lll} \mathbf{S}_1; & \leadsto & \mathbf{S}_1; \\ \mathbf{S}_2; & & \mathit{interrupt}(\mathit{time});\ \mathit{time} := \mathit{time} + 1; \\ \text{etc} \ldots & & \mathbf{S}_2; \\ & & \mathit{interrupt}(\mathit{time});\ \mathit{time} := \mathit{time} + 1; \\ & & \text{etc} \ldots \end{array}$$

If we assume a discrete model of time, i.e. that the value of *time* is an integer, and we assume that time is incremented by one between each potential interrupt, then the test for validity of a call to the interrupt routine is simply:

$$\mathit{interrupt}(\mathit{time}_0) \quad \leadsto \quad \underline{\mathbf{if}}\ (\mathit{time}_0 = \mathit{input}[1][1])\ \underline{\mathbf{then}}\ \mathit{process\_interrupt}\ \underline{\mathbf{fi}}$$

where *process_interrupt* corresponds to the original interrupt service routine. Note that $\mathit{input}[1]$ is the first element of the input sequence: this element is a pair of values (a time and a character), so $\mathit{input}[1][1]$ is the first element of the pair, i.e. the time of the first interrupt.

However, a better model, which does not require a discrete model of time, and which allows different "atomic" (i.e. non-interruptable) operations to take different amounts of time, is the following:

$$\mathit{interrupt}(\mathit{time}_0) \leadsto \underline{\mathbf{while}}\ (\mathit{time}_0 \geqslant \mathit{input}[1][1])\ \underline{\mathbf{do}}\ \mathit{process\_interrupt}\ \underline{\mathbf{od}}$$

This revised model of the interrupt routine allows more than one interrupt to occur between atomic operations, and has the advantage that a call to *interrupt* can be merged with a second call which immediately follows it:

$$\mathit{interrupt}(t_1);\ \mathit{interrupt}(t_2) \ \approx \ \mathit{interrupt}(t_2)$$

provided $t_2 \geqslant t_1$. This follows from the transformation:

$$\begin{array}{l} \underline{\mathbf{while}}\ \mathbf{B}_1\ \underline{\mathbf{do}}\ \mathbf{S}\ \underline{\mathbf{od}}; \\ \underline{\mathbf{while}}\ \mathbf{B}_2\ \underline{\mathbf{do}}\ \mathbf{S}\ \underline{\mathbf{od}} \end{array} \ \approx \ \underline{\mathbf{while}}\ \mathbf{B}_2\ \underline{\mathbf{do}}\ \mathbf{S}\ \underline{\mathbf{od}}$$

provided $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$. This transformation is proved in [9]. If $t_2 \geqslant t_1$ then we have $(t_1 \geqslant \mathit{input}[1][1]) \Rightarrow (t_2 \geqslant \mathit{input}[1][1])$, and we can merge the two **while** loops and hence the two procedures.

Thus, once we have moved a set of interrupt procedure calls to the same place, we can merge them into one statement equivalent to "process all outstanding interrupts", which is much closer to a specification level statement than is a series of calls to the same procedure.

Note that the addition of interrupt calls is defining the "interruptable points" in the program, or equivalently, the "atomic operations". The increments to *time* define the processing time for each atomic operation. For real programming languages, e.g. Coral, the atomic operations may well be machine code instructions, rather than high-level language statements, and it is at the machine code level that the model needs to be constructed, for it to accurately reflect the real program. This will inevitably lead to a large and complex WSL program; however automatic restructuring and simplifying transformations can eliminate much of the complexity before the maintainer even has to look at the program.

5 Concurrency

Interrupts may be regarded as a special type of concurrent processing on a single processor. When an interrupt occurs, the "main" program is suspended and the interrupt routine is executed in its entirety, possible changing the state of the main program in the process. Execution of the main program then resumes. It is the fact that the interrupt routine is executed in its entirity that makes interrupts a special case from the point of view of modelling in WSL; we are able to insert a copy of the interrupt routine wherever an interrupt occurs, and hence the effect of the interrupt is deterministic.

The analogy with a single-processor multitasking system is obvious: here the running program executes until it is suspended by the operating system. Other tasks are then (partially) executed, and may change the state of the original program; eventually execution of this program is resumed. From the perspective of the original task, this looks like a call to a procedure which executes sequences of instructions from the operating system and the other tasks in the system, and subsequently returns. The analogy also applies to more general forms of concurrency, including fully-parallel multiprocessor systems. In principle, the state of any program or task in such a system may be changed, between one atomic operation and the next, by other concurrently executing tasks. Again this could be modelled by procedure calls between each pair of atomic operations, which perform the appropriate processing and change the state accordingly.

5.1 Rely and Guarantee conditions

Rely and guarantee conditions were introduced to augment the pre- and postconditions of VDM when developing parallel systems [5]. They provide a means to specify the interaction between a program and its execution environment (concurrent tasks). A guarantee condition is a condition on the state shared by the program and its environment, which the program will at all times preserve. Similarly, a rely condition is a condition on the state of the program which any interference from the environment will preserve. As an example, consider the abstraction:

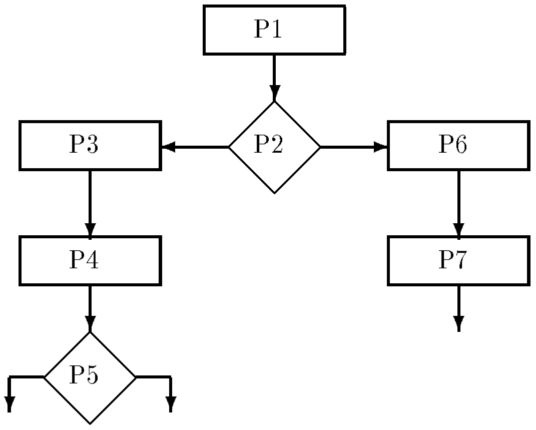
$$\begin{array}{lcl} x := 1; & \geq & x := x'.(x' \geqslant 1); \\ \underline{\textbf{if}}\ x \geqslant 1\ \underline{\textbf{then}}\ y := 0\ \underline{\textbf{fi}} & & y := 0 \end{array}$$

In pure WSL this is trivially correct, but if we allowed interference from the environment to change the value of x between the assignment and the test, the transformation is not necessarily valid. However, if we have a rely condition which specifies $x \geqslant \overleftarrow{x}$ then this transformation will be valid. (Here $\overleftarrow{x}$ and x represent respectively the initial and final values of x)

In the following section we use rely and guarantee conditions to model the effects of concurrency in pure WSL. Again this means that our program transformations are applicable since the resulting program is purely sequential.

5.2 Modelling Concurrency in WSL

Consider a sequential program **T** which can be represented by a flow chart. Any program can be restructured into this form. For example:



where each of the Pn represents an atomic instruction or an atomic test. Suppose we model Pn by Pn; $next := n{+}1$; $time := time{+}1$; , with the obvious extension for branch instructions. Now suppose **T** is a task in a concurrent system. Consider first a system with only two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$. We rewrite $\mathbf{T}_2$ as a procedure as follows;

```
proc T_2(steps) ≡
    while (steps > 0) do
        steps := steps − 1;
        if next = 1 → P1; next := 2
        □ next = 2 → if P2
                        then next := 3
                         else next := 6 fi
        ...
        □ next = 0 → skip fi od;
```

This procedure executes a sequence of *steps* instructions from $\mathbf{T}_2$, beginning from the last instruction executed in the last invocation of the procedure. If *steps*=0 then the procedure call returns immediately.

If we then model $\mathbf{T}_1$ by

$$\begin{array}{ll} \mathbf{T}_1 \leadsto & steps := steps'.(steps' \in \mathbb{N}^0); \\ & \mathbf{T}'_2(steps); \\ & \mathrm{S1};\ time := time + 1; \\ & steps := steps'.(steps' \in \mathbb{N}^0); \\ & \mathbf{T}'_2(steps); \\ & \mathrm{S2};\ time := time + 1; \\ & \ldots \end{array}$$

then we have modelled the semantics of the original system. By interposing calls to $\mathbf{T}'_2$ between the atomic instructions of $\mathbf{T}_1$ we have modelled the execution of $\mathbf{T}_2$ and its interference with $\mathbf{T}_1$.

This is a complete model of the system, even though it has the same surface structure as $\mathbf{T}_1$. It is non-deterministic due to the presence of the non-deterministic assignment statements in $\mathbf{T}_1$, which cause values to be assigned to the variable *steps*. This reflects the fact that we cannot know *a priori* how many instructions from $\mathbf{T}_2$ will be executed between Pn and P$n+1$ in $\mathbf{T}_1$. In order to proceed, we want to replace:

$$\mathit{steps} := \mathit{steps}'.(\mathit{steps}' \in \mathbb{N}^0);\ \mathbf{T}_2'(\mathit{steps});$$

with an abstraction, which specifies $\mathbf{T}_2'$ for any value of *steps* and any initial value of *next*. This will be the strongest condition preserved by the execution of *any* sequence of instructions from $\mathbf{T}_2$, and corresponds to a guarantee condition for $\mathbf{T}_2$. This specification is recoverable from $\mathbf{T}_2$ in isolation.

We can repeat this process, interchanging the roles of $\mathbf{T}_1$ and $\mathbf{T}_2$. We then will have two models, one based on $\mathbf{T}_1$ with interference from $\mathbf{T}_2$ inserted, the other based on $\mathbf{T}_2$ with interference from $\mathbf{T}_1$. However these are guaranteed to be equivalent, since they both capture the logic of the entire system.

Now suppose we have more than two tasks $\mathbf{T}_1 \ldots \mathbf{T}_n$. Let $\mathbf{T}_2 \ldots \mathbf{T}_n$ be rewritten as procedures as above. Then we can write

```
T₁ ⇝ $\mathit{steps} := \mathit{steps}'.(\mathit{steps}' \in \mathbb{N}^0)$;
     $V(\mathit{steps})$;
     S1; $\mathit{time} := \mathit{time} + 1$;
     $\mathit{steps} := \mathit{steps}'.(\mathit{steps}' \in \mathbb{N}^0)$;
     $V(\mathit{steps})$;
     ...
```

where

```
proc $V(\mathit{steps})$ ≡
    for $k := 1$ to $\mathit{steps}$ step 1 do
        if true → $\mathbf{T}_2'(1)$
        □ true → $\mathbf{T}_3'(1)$
        ...
        □ true → $\mathbf{T}_n'(1)$ fi od end
```

and each $\mathbf{T}_j'$ is derived from the corresponding $\mathbf{T}_j$ as before.

Procedure V causes *steps* instructions to be executed; these are chosen non-deterministically from the remaining tasks $\mathbf{T}_2, \ldots, \mathbf{T}_n$ by the **<u>if</u>** statement, which calls one of the corresponding procedures with parameter one, resulting in the execution of one instruction from the task.

The above is a complete specification for the system when taken together with the definitions of the procedures $\mathbf{T}_j'$. However it is also non-deterministic due to the assignment statements; as before we need to replace the assignments and procedure calls by abstractions.

Consider the procedure V. Here the loop body implements a non-deterministic choice from the set of procedures $\mathbf{T}_j'$, where $j = 2 \ldots n$. One of the procedures is selected non-deterministically, and a single instruction from the corresponding task is executed. A total of *steps* atomic instructions from the set of tasks is executed; individual instructions or sequences of instructions may be executed from any of the tasks.

We can abstract $\mathbf{T}_j'(1)$ to a non-deterministic choice over all sequences of instructions in $\mathbf{T}_j$ — in so doing we abstract away the fact that the instructions are executed in a specific order. This is equivalent to removing the guards from the **<u>if</u>** statement in the procedures $\mathbf{T}_j'$, which we can always do as an abstraction step. We can therefore write

$$\mathbf{T}_j'(1) \ \geq\ \prod_i \mathrm{P}_j^i$$

where P_j^i are the instructions in the task $\mathbf{T}_j$, and $\prod_i$ indicates a non-deterministic choice. This is the strongest specification which is an abstraction of every instruction in $\mathbf{T}_j$. In fact this specification may be very large for a task which is even moderately complex, making it difficult to use in practice. To reason about its effect on $\mathbf{T}_1$ however we are only concerned with the elements of the specification which affect the state of $\mathbf{T}_1$; we may therefore if necessary abstract away other details of the $\mathbf{T}_j$'s and so simplify the specifications of the $\mathbf{T}_j'$s.

Having found a specification for each of the $\mathbf{T}_j'$, we can use these to find the specification for the loop body of the procedure V. This is a non-deterministic choice from the set of $\mathbf{T}_j'(1)$'s, whose specifications are given above. The procedure V executes the loop body *steps* times, where the value of *steps* is not determinable, representing as it does the number of instructions executed between two sequential instructions in $\mathbf{T}_1$. The strongest specification for V which we can find is therefore the strongest abstraction which holds for any value of *steps*, i.e. any number of concatenations of the loop body. This will in general be an abstraction of the specification for the loop body.

To summarise, we have the following abstraction/refinement relations:

$$\mathbf{T}_j'(1) \ \geq\ \prod_i \mathrm{P}_j^i$$

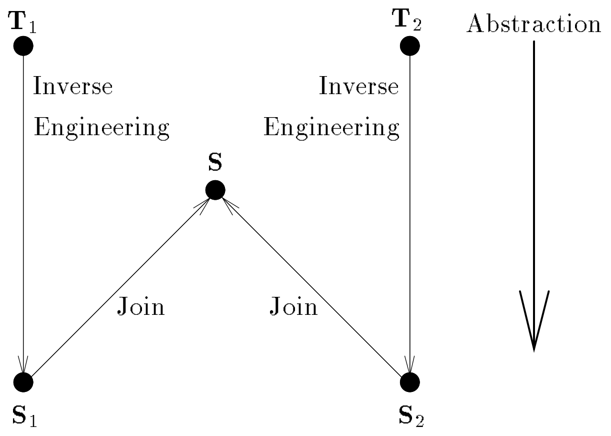
$$\mathrm{L} \ \approx\ \prod_{j>1} \mathbf{T}_j'(1) \ \geq\ \prod_{j>1}(\prod_i \mathrm{P}_j^i) \ \approx\ \mathbf{S}_L$$

$$\mathbf{S}_L \ \geq\ \mathbf{S}_V$$

where L is the body of the loop, $\mathbf{S}_L$ is a specification for the loop body, and $\mathbf{S}_V$ is a specification for the procedure V. The crucial factor in this method is that this specification is derived by analysis of the individual tasks in isolation, without the need to take account of the interaction between them. It can therefore be found using methods developed for purely sequential systems.

$\mathbf{S}_V$ therefore specifies the effects of the remaining tasks on the state of $\mathbf{T}_1$. It is in effect a rely condition for $\mathbf{T}_1$. By substituting this specification for the calls to procedure V in our model of $\mathbf{T}_1$, we can inverse engineer $\mathbf{T}_1$ to recover its specification, $\mathbf{S}_1$, which includes the effects of the other tasks upon it. Since we may have abstracted away those parts of the specification which do not affect the state of $\mathbf{T}_1$, in order to simplify the specifications of the $\mathbf{T}'_j(1)$'s, we no longer have a complete specification for the entire system. Repeating this exercise for the remaining tasks allows us to recover a set of specifications $\mathbf{S}_j$ for the tasks, each of which include the effects of the other tasks upon their state.

The specification $\mathbf{S}$ for the complete system must incorporate all the properties of the set of $\mathbf{S}_j$'s. This is expressed in WSL by the **join** operation. The **join** of a set of specifications is the most abstract specification which refines each member of the set. Thus the specification $\mathbf{S}$ is more refined than the specifications $\mathbf{S}_j$, unless the specifications $\mathbf{S}_j$ are equivalent; this arises since, in combining the specifications for the tasks, we are restoring information originally abstracted away in the derivation of the individual $\mathbf{S}_j$'s: $\mathbf{S}_i$ will include information abstracted away in the derivation of $\mathbf{S}_k$ (for any i and k).



Although $\mathbf{S}$ will be more refined than the set of $\mathbf{S}_j$'s, it may still be too abstract to be useful if too much abstraction is performed in deriving the $\mathbf{S}_j$'s. This will be a problem particularly if the objective is to validate a system against an existing specification, or to re-implement an existing system. If too much information is lost in deriving the $\mathbf{S}_j$'s then the specification $\mathbf{S}$ may be more abstract than the specification against which we wish to validate, or more abstract than the actual system requirements specification.

5.3 Summary of method

In order to model a concurrent system in WSL we proceed as follows:

For each task, we derive a specification for the conditions on its state which are guaranteed to be preserved by the other tasks in the system. To achieve this, we consider each of the remaining tasks in isolation, and derive for each of these a specification of the conditions on the state of the first task which are guaranteed to be preserved by the execution of any instruction from the second task. The resulting specifications, derived from all the remaining tasks, are then combined by a non-deterministic choice to give us a specification of the conditions on the state of the first task, guaranteed to be preserved by the execution of any instruction from any other task in the system. The full specification for the effect of the remaining tasks on the first is then given by any number of concatenations of this specification.

We substitute this specification between each atomic operation in the task: this gives us an abstraction of the task itself including the effect of interference from the rest of the system. We may then inverse engineer this model of the task using transformations, to find a specification for the task incorporating the effects of interference from the other tasks.

We can repeat this process for each of the tasks in the system. Having derived a specification for each of them, we can derive the specification for the complete system by combining the individual specifications using the WSL **join** operation.

6 Example of method

In this section we give an example of the use of the method to inverse engineer a simple concurrent system. This consists of only two tasks sharing a single processor under the control of a scheduler, though we ignore the details of the scheduler. One task receives characters from an input stream, and writes these into a buffer; the second process takes characters from this buffer and writes them to the standard output.

```
$\mathbf{T}_1 = \underline{\mathbf{do}}\ \underline{\mathbf{if}}\ empty(input)\ \underline{\mathbf{then}}\ \underline{\mathbf{exit}}(1)\ \underline{\mathbf{fi}};$
          $\underline{\mathbf{while}}\ (time < input[1][1])\ \underline{\mathbf{do}}$
                $suspend1\ \underline{\mathbf{od}};$
          $buf := buf \mathbin{+\!\!+} \langle input[1][2] \rangle;$
          $input := tail(input)\ \underline{\mathbf{od}}$

$\mathbf{T}_2 = \underline{\mathbf{do}}\ \underline{\mathbf{if}}\ empty(input)\ \wedge\ empty(buf)$
            $\underline{\mathbf{then}}\ \underline{\mathbf{exit}}(1)\ \underline{\mathbf{fi}};$
          $\underline{\mathbf{while}}\ (empty(buf))\ \underline{\mathbf{do}}$
                $suspend2\ \underline{\mathbf{od}};$
          $std\_out := std\_out \mathbin{+\!\!+} \langle buf[1] \rangle;$
          $buf := tail(buf)\ \underline{\mathbf{od}}$
```

The input consists of a sequence of pairs: a character and its arrival time. If there is no character waiting to be read, i.e. the arrival time of the next character in the input has not yet been reached, $\mathbf{T}_1$ suspends itself by means of the system call *suspend1*, which results in $\mathbf{T}_2$ being reactivated. Similarly, if the buffer is empty, $\mathbf{T}_2$ suspends itself by the system call *suspend2*. If there is no more input to be received, then $\mathbf{T}_1$ terminates; similarly if there is no further input and no characters left in the buffer $\mathbf{T}_2$ terminates.

Execution of the two tasks alternates on a timeslice basis: at regular intervals, the executing task is halted and the second task is restarted. We can model this using a procedure: we define

```
$\underline{\mathbf{proc}}\ timeslice1\ \equiv$
      $\underline{\mathbf{if}}\ (time - last \geqslant P_t)$
        $\underline{\mathbf{then}}\ last := time;\ suspend1\ \underline{\mathbf{fi}}.$
```

where P_t is the timeslice period, and an equivalent procedure *timeslice2* for $\mathbf{T}_2$. This procedure tests the time since the last context switch, and if it is greater than or equal to the timeslice period suspends the active task and reactivates the halted one via the system call *suspend1*. To incorporate the effect of interference from the other task we insert this procedure call between each statement in the task, together with an assignment which increments *time*. (For the purposes of this example we treat each WSL statement as an atomic instruction). The result, following the necessary restructuring of the **while** loops to accommodate these additions, is

```
$\hat{\mathbf{T}}_1 \approx \underline{\mathbf{do}}\ \underline{\mathbf{if}}\ empty(input)\ \underline{\mathbf{then}}\ \underline{\mathbf{exit}}(1)\ \underline{\mathbf{fi}};$
        $\underline{\mathbf{do}}\ \underline{\mathbf{if}}\ (time \geqslant input[1][1])$
              $\underline{\mathbf{then}}\ timeslice1;\ time := time + 1;$
                    $\underline{\mathbf{exit}}(1)$
               $\underline{\mathbf{else}}\ timeslice1;\ time := time + 1\ \underline{\mathbf{fi}};$
           $suspend1\ \underline{\mathbf{od}};$
        $buf := buf \mathbin{+\!\!+} \langle input[1][2] \rangle;$
        $timeslice1;\ time := time + 1;$
        $input := tail(input);$
        $timeslice1;\ time := time + 1\ \underline{\mathbf{od}}$

$\hat{\mathbf{T}}_2 \approx \underline{\mathbf{do}}\ \underline{\mathbf{if}}\ empty(input)\ \wedge\ empty(buf)$
          $\underline{\mathbf{then}}\ \underline{\mathbf{exit}}(1)\ \underline{\mathbf{fi}};$
        $\underline{\mathbf{do}}\ \underline{\mathbf{if}}\ \neg empty(buf)$
              $\underline{\mathbf{then}}\ timeslice2;\ time := time + 1;$
                    $\underline{\mathbf{exit}}(1)$
               $\underline{\mathbf{else}}\ timeslice2;\ time := time + 1\ \underline{\mathbf{fi}};$
           $suspend2\ \underline{\mathbf{od}};$
        $std\_out := std\_out \mathbin{+\!\!+} \langle buf[1] \rangle;$
        $timeslice2;\ time := time + 1;$
        $buf := tail(buf);$
        $timeslice2;\ time := time + 1\ \underline{\mathbf{od}}$
```

In order to proceed, we now need definitions for the procedures *suspend1* and *suspend2*. These specify the effect of executing the other task for one timeslice period, and must refine the conditions which are preserved by execution of any sequence of statements from the tasks, i.e. any number of whole or partial executions of the respective loop bodies. We can recover these conditions, and define abstractions of *suspend1* and *suspend2* which preserve the conditions but are otherwise unrestricted. We denote these by *G1* and *G2* respectively. We then know that $G1 \leq suspend1$ and $G2 \leq suspend2$.

During its timeslice, a task executes without interference from the other task. We therefore recover *G1* and *G2* from $\mathbf{T}_1$ and $\mathbf{T}_2$. As these are small in this example, we can do this by inspection. Consider first $\mathbf{T}_2$. We are interested only in how it affects the variables in $\mathbf{T}_1$: in fact the only variables which are changed by $\mathbf{T}_2$ are *std_out* and *buf*, (and also *time*, though this is not shown in the definition of $\mathbf{T}_2$ given above). *std_out* is not accessed or updated in $\mathbf{T}_1$, so we can disregard it. Since at least one statement in $\mathbf{T}_2$ will be executed, *time* will be incremented. Also, we can see that zero or more characters will be removed from the head of *buf*, up to a maximum of $\ell(input)$, in which case the buffer is emptied and $\mathbf{T}_2$ is then suspended. Using these facts we can write:

$$\begin{aligned} G1 = \langle time, buf\rangle := \langle time', buf'\rangle.&\\ (time' > time&\\ \wedge\ \exists j, 0 \leqslant j \leqslant \ell(buf).\ buf' = buf[j\,..])& \end{aligned}$$

Clearly, $G1;\ G1 \approx G1$. As $G1 \leq suspend1$ we can also write

$$\begin{aligned} &timeslice1\\ &\quad\geq \underline{\textbf{if}}\ (time - last \geqslant P_t)\\ &\qquad \underline{\textbf{then}}\ last := time;\ G1\ \underline{\textbf{fi}}\\ &\quad\geq \langle time, buf\rangle := \langle time', buf'\rangle.\\ &\qquad (time' > time\\ &\qquad\quad \wedge\ \exists j, 0 \leqslant j \leqslant \ell(buf).\ buf' = buf[j\,..])\\ &\quad\geq G1 \end{aligned}$$

Having found $G1$, we are able to inverse engineer $\hat{\mathbf{T}}_1$:

```
$\hat{\mathbf{T}}_1 \ \geq$
do if $empty(input)$ then exit(1) fi;
   do if $(time \geqslant input[1][1])$
         then $G1;\ time := time + 1;$
              exit(1)
         else $G1;\ time := time + 1$ fi;
      $G1$ od;
   $buf := buf \mathbin{+\!\!+} \langle input[1][2]\rangle;$
   $G1;\ time := time + 1;$
   $input := tail(input);$
   $G1;\ time := time + 1$ od;
```

Replacing $G1$ by its definition, transforming the loop to a **while** and simplifying gives:

```
while $\neg empty(input)$ do
      $\langle time, buf\rangle := \langle time', buf'\rangle.$
         $(time' > input[1][1]$
            $\wedge\ \exists j, 0 \leqslant j \leqslant \ell(buf).$
                $buf' = buf[j\,..] \mathbin{+\!\!+} input[1][2]);$
      $input := tail(input)$ od;
```

This loop can be replaced by a single assignment:

$$\begin{aligned} &\hat{\mathbf{T}}_1 \ \geq\\ &\langle time, buf\rangle := \langle time', buf'\rangle.\\ &\quad (time' > input[\ell(input)][1]\\ &\qquad \wedge\ \exists j, 0 \leqslant j \leqslant \ell(buf) + \ell(input).\\ &\qquad\qquad buf' = (buf \mathbin{+\!\!+} \pi_2 * input)[j\,..]);\\ &input := \langle\rangle\\ &= \mathbf{S}_1 \end{aligned}$$

Here $\pi_2 * input$ is the sequence of "second elements" from the sequence of pairs *input*. $\mathbf{S}_1$ is a specification for $\mathbf{T}_1$ with the interference from $\mathbf{T}_2$ included. All the input characters are read and so the input is emptied; the time at which the task terminates is some time after the last character is received; on termination, the buffer consists of the initial contents of the buffer with all the input characters appended to the end, and an unspecified number of characters (less than or equal to the number of characters in the buffer) removed from the head. This quantity cannot be determined solely from $\hat{\mathbf{T}}_1$ as it depends on the number of iterations of $\mathbf{T}_2$ which are executed before $\mathbf{T}_1$ terminates, and this is unknown to $\mathbf{T}_1$.

We now inverse engineer $\hat{\mathbf{T}}_2$ by the same method to find a specification $\mathbf{S}_2$. Inspection of $\mathbf{T}_1$ provides us with a suitable abstraction. Define:

$$\begin{aligned} G2 = \langle buf, time\rangle := \langle buf', time'\rangle.&\\ (time' > time&\\ \wedge\ \exists j, 0 \leqslant j \leqslant \ell(input).&\\ buf' = buf \mathbin{+\!\!+} \textstyle\sum_{i=0}^{j} input[i][2])& \end{aligned}$$

We have: $G2;\ G2 \approx G2$ and $timeslice2 \geq G2$.

This is an abstraction of *suspend2*. Substituting this in $\hat{\mathbf{T}}_2$, we can inverse engineer to find:

$$\begin{aligned} &\hat{\mathbf{T}}_2 \ \geq\\ &\langle std\_out, buf, time\rangle := \langle std\_out', buf', time'\rangle.\\ &\quad \big(time' > time\\ &\qquad \wedge\ std\_out' \mathbin{+\!\!+} buf' \mathbin{+\!\!+} \pi_2 * input'\\ &\qquad\qquad = std\_out \mathbin{+\!\!+} buf \mathbin{+\!\!+} \pi_2 * input\\ &\qquad \wedge\ buf' = \langle\rangle\ \wedge\ input' = \langle\rangle\big)\\ &\approx\\ &\langle std\_out, buf, time\rangle := \langle std\_out', buf', time'\rangle.\\ &\quad \big(std\_out' := std\_out \mathbin{+\!\!+} buf \mathbin{+\!\!+} \pi_2 * input\\ &\qquad \wedge\ time' > time\big)\\ &\approx \mathbf{S}_2 \end{aligned}$$

The specification for the complete system is then given by:

$$\mathbf{S} \ =_{\text{DF}}\ \underline{\textbf{join}}\ \mathbf{S}_1 \sqcup \mathbf{S}_2\ \underline{\textbf{nioj}}$$

For two specifications, the **join** operation reduces to and-ing the conditions in the specification statements. Therefore:

$$\begin{aligned} &\mathbf{S} \approx\\ &\langle std\_out, buf, time\rangle := \langle std\_out', buf', time'\rangle.\\ &\quad \big(time' > input[\ell(input)][1]\\ &\qquad \wedge\ std\_out' = std\_out \mathbin{+\!\!+} buf \mathbin{+\!\!+} \pi_2 * input\\ &\qquad \wedge\ buf' = \langle\rangle\ \wedge\ input' = \langle\rangle\big) \end{aligned}$$

This specification tells us that, for any initial values of *buf*, *input* and *std_out*, the system terminates in a state in which *buf* and *input* are empty, and *std_out* consists of its initial value with the initial contents of *buf* and the sequence of characters from *input* appended, in the order in which they occurred in the input. Additionally, the time at which the system terminates is greater than the time of arrival of the last character. In this example, the specification $\mathbf{S}_2$ is almost a complete specification for the system, reflecting the fact that very little high level information was abstracted away in deriving a specification for the interference of $\mathbf{T}_1$ with the state of $\mathbf{T}_2$.

7 Conclusions

This study shows that by using an appropriate models we can represent interrupt-driven and concurrent programs within the (purely sequential) WSL language. With such models, the inverse engineering techniques of [12,14] can be applied to extract the specification of the original program. Program transformations are sufficiently powerful to cope with these, often complex, models. Although a fairly large number of transformations are required to deal with these models, results from case studies indicate that these are used in a systematic way: this suggests that much of the work can be automated by a tool such as the Maintainer's Assistant [2,17] and this is currently being investigated under a SMART II (Small Firms Merit Award for Research and Technology) project at the Centre for Software Maintenance Ltd., and as part of a three-year SERC project at the University of Durham.

Acknowledgements

The research described in this paper has been funded partly by a Department of Trade and Industry SMART award to the Centre for Software Maintenance Ltd., and partly by SERC (The Science and Engineering Research Council) project "A Proof Theory for Program Refinement and Equivalence: Extensions".

References

[1] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[2] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[4] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

[5] C. B. Jones, "Specification and Design of (Parallel) Systems," in *Proc. IFIP 1983*, R. E. A. Mason, ed., North-Holland, Amsterdam, 1983, 321–332.

[6] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 (1974), 261–301.

[7] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[8] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[9] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[10] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990.

[11] M. Ward, "A Recursion Removal Theorem," Springer-Verlag, Proceedings of the 5th Refinement Workshop, London, 8th–11th January, New York–Heidelberg–Berlin, 1992.

[12] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (1993), 101–122.

[13] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," forthcoming, 1993.

[14] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., Apr., 1991.

[15] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., May, 1992.

[16] M. Ward & K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (May, 1993).

[17] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (Oct., 1989).