COVER SHEET

A Theory for Execution-Time Derivation in Real-Time Programs

Karl Lermer [a,*], Colin J. Fidge [b] and Ian J. Hayes [a]

[a] *School of Information Technology and Electrical Engineering, The University of Queensland, Queensland 4072, Australia*

[b] *School of Software Engineering and Data Communications, Queensland University of Technology, Queensland 4001, Australia*

Abstract

We provide an abstract command language for real-time programs and outline how a partial correctness semantics can be used to compute execution times. The notions of a timed command, refinement of a timed command, the command traversal condition, and the worst-case and best-case execution time of a command are formally introduced and investigated with the help of an underlying weakest liberal precondition semantics. The central result is a theory for the computation of worst-case and best-case execution times from the underlying semantics based on supremum and infimum calculations. The framework is applied to the analysis of a message transmitter program and its implementation.

Key words: Real-time programming; Control-flow analysis; Execution-time derivation and prediction; Predicate transformer semantics; Partial correctness.

1 Introduction

State-of-the-art development of safety-critical systems must guarantee the implementation of the system's safety requirements to the highest possible assurance level. This especially applies to real-time systems where producing results between certain time bounds is crucial for correct and safe behaviour. The major advances in real-time programming have been in the areas of scheduling periodic processes [5,6] and worst-case execution-time analysis of program

* Corresponding author.
Email address: lermer@itee.uq.edu.au (Karl Lermer).

code [8,23]. Together these allow the design of systems as a set of communicating processes, where each process consists of a block of code which is repeatedly executed with a given (fixed) period. In order to successfully schedule such a set of processes it is necessary to know the worst-case execution time of the block of code associated with each process.

The requirements of a reactive real-time system are most succinctly specified in terms of a relation between inputs and outputs over all time. To bridge the gap between such high-level system requirements and real-time implementations that satisfy those requirements, a theory of real-time software development is required [19]. Figure 1 shows our approach to formal development and timing analysis of real-time systems. This figure outlines the development of a real-time system from an abstract specification into a high-level language program to a machine-code implementation in a specific environment. The separation of timing behaviour and functional behaviour [13] is a key feature of this framework that allows a more abstract approach to the program development process. Real-time program development is partitioned into

- a machine-independent phase that, given the specification of a real-time system, develops a machine-independent program, annotated with timing requirements, to meet the specification [14,19], and
- a machine-dependent phase, that checks that the compiled version of the program will meet all its timing requirements when it is executed on the particular target machine [24,25].

This partitioning makes it possible to perform abstract reasoning about timing requirements in the system design and high-level language program, and to separately prove that the requirements are satisfied by the compiled code.

Within this framework, timing analysis begins by annotating the program with the programmer's real-time requirements. This can be done by extending the program semantics with a variable to denote the current time [21] and by adding a 'deadline' statement to the programming language [13] which allows bounds on the current time to be expressed. We have also developed methods to derive sequential real-time programs from specifications [19] and perform timing analysis [15]. This paper focuses on using the program's semantics to derive execution times for real-time programs. In particular, we are interested in high-level program analysis where the derivation of execution times for annotated real-time programs is essential for the formulation of timing constraints. These constraints can be used in the subsequent timing analysis phase to verify that the compiled machine code has acceptable timing behaviour. This phase refers to the dashed ellipses in Figure 1. Numerous algorithms and tools for predicting worst-case execution times from both high-level and machine-level code already exist [35,34,28,11]. Our goal here is not to derive yet another such algorithm, but to elaborate the seman-

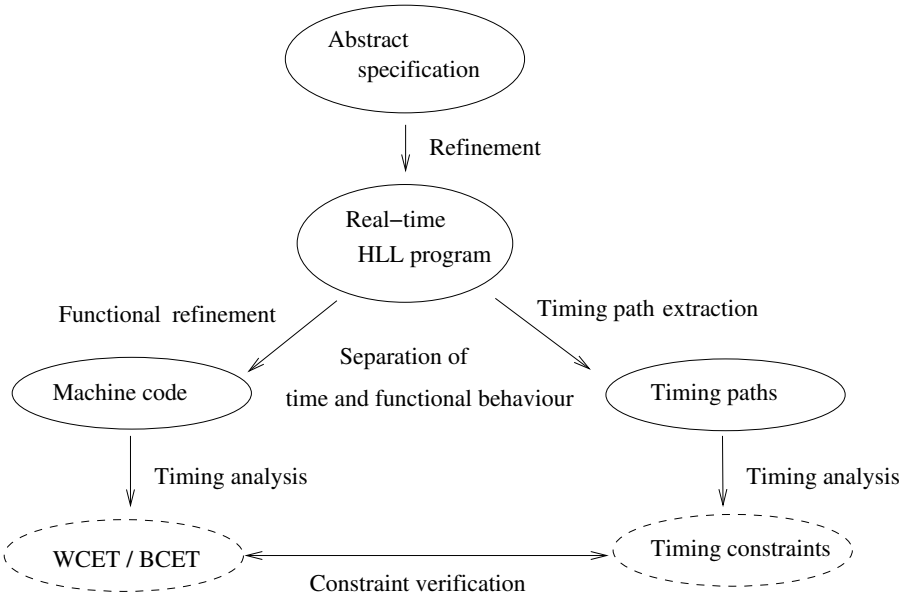

Fig. 1. Phases of real-time program development.

tic theory that underlies these implementations. We investigate the notions of timed commands and their associated execution times from a fundamental perspective, and thus provide a sound theoretical basis for the justification of the correctness of execution-time analysis techniques.

The remainder of this paper is structured as follows. Section 2 states the syntax and semantics for abstract commands. The notions of command and timed command are defined. Section 3 defines the entry and traversal conditions for commands. Section 4 defines the liberal refinement relation on the command language. In Section 5 the notions of worst-case and best-case execution times are introduced for timed commands. As a main result we present a method for symbolic computation of traversal conditions, and worst-case and best-case execution times. In Section 6 these techniques are applied to a 'message transmitter' example. Three appendices at the end of the paper contain important definitions and theorems.

1.1 Related work

In this paper we develop a semantic theory for deriving traversal conditions for, and execution-time expressions from, real-time program paths. Both of these goals have been explored previously, but from a pragmatic, rather than theoretical, perspective. Extraction and analysis of 'program paths', or 'partial programs', has long been a cornerstone of static program analysis [22]. For efficiency and ease of use, the aim is usually to devise algorithms that can be applied automatically and syntactically to a program or program path.

Path traversal conditions are important because they can be used to show that programmer-supplied assertions are consistent with the code, and that the code itself is internally consistent. For instance, Bergeretti and Carré devised algorithms that manipulate matrices representing information flow between the statements in a path in order to analyse programs for coding errors such as ineffective assignments [4]. Subsequently, a data-flow analysis algorithm was incorporated in the SPADE program analysis tool [7] and its successors. In particular, the SPARK Examiner tool uses weakest precondition semantics for program paths [3]. A particularly explicit tool for path traversal condition calculation is the Path Exploration Tool [16]. It allows the programmer to select a control-flow path through a program and a simple algorithm then calculates the path's traversal condition. A particularly important application of algorithms that calculate path traversal conditions is to identify 'dead', or 'infeasible', program paths, i.e., those that can never be followed at run time [7,8]. In previous work we described a semantic basis for such dead path analysis [17].

Analysing program paths to predict their worst-case (and sometimes best-case) execution times is a standard requirement of high-integrity real-time programming. The process is awkward because the actual execution time of the final system cannot be known until the program is compiled and the particular host processor and operating environment are chosen. The most accurate timing predictions are those made by analysing paths through the compiled assembler code [36], although even this is difficult for code that will run on pipelined and cached processors [28]. However, our interest here is also with high-level language program analysis. In this case the program is divided into its individual control-flow paths and formulas corresponding to each language construct in the path can be used to derive an execution-time expression for that path. The resulting expression may be symbolic, or it may be instantiated with specific execution times for each primitive component to produce a numerical result [35,33]. For instance, Chapman et al. present practical algorithms for partitioning high-level language programs into paths, eliminating dead paths by calculating their traversal conditions, and predicting the worst-case execution times of the remainder based on a cost function associated with each path component [8]. (Eliminating dead paths is particularly important in static timing analysis, because incorporating their execution times can severely distort the overall timing predictions [12,27].) Puschner and Schedl's approach to timing analysis uses graph theory to extract paths which are then assessed using a worst-case execution-time function [34]. Engblom and Ermedahl convert control-flow information to linear constraints to support subsequent execution-time analysis [11]. Park advocates an approach in which the programmer uses an explicit 'path language' to identify those paths to be analysed for their worst-case execution times [32].

Whichever technique is used, to be effective in practice, automatic algorithms

for identifying path traversal conditions and predicting worst-case execution times must work syntactically because they cannot afford the overheads of semantic theorem proving. Our goal below, therefore, is to provide the general semantics underlying such algorithms, thus providing a sound basis for justifying their correctness.

2 Syntax and semantics for timed commands

This section presents a general language for real-time programs on the basis of a partial correctness semantics. For some real-time programs their correct behaviour for some inputs may be to never terminate, for example, a loop waiting for an external signal will never terminate if the signal never arrives. Additionally, there are situations in which the signal may only appear transiently: in these cases the loop may or may not detect the signal and hence may or may not terminate; either behaviour is correct in this situation.

We would like to reason about the timing behaviour of such programs, for example, if the loop in the above example detects the signal and terminates, it may be required to respond to the signal within some deadline. For this reason *weakest liberal precondition (wlp)* semantics [10] are used in this paper. Recall that $wlp(S, R)$ is the weakest predicate characterising initial states from which predicate R is guaranteed to hold in the final state, provided that command S terminates. This can be contrasted with *weakest precondition (wp)* semantics in which $wp(S, R)$ characterises those initial states from which S is both guaranteed to terminate and establish R on termination. The following relationship holds between wp and wlp:

$$wp(S, R) \equiv wp(S, \mathit{true}) \wedge wlp(S, R),$$

where $wp(S, \mathit{true})$ characterises those states from which S is guaranteed to terminate. According to weakest precondition semantics a loop that may or may not terminate is indistinguishable from a loop that never terminates (both have a weakest precondition of *false* for any postcondition) but the two loops are distinguished by weakest liberal precondition semantics.

2.1 Basics

Let *Var* denote the universal set of variables that may take their values from a universal set of values, *Val*, which includes the booleans, $\mathbb{B}$. The set of all possible states, Σ, is then defined as the set of all functions from *Var* to *Val*. We assume that for every unprimed variable $x \in \mathit{Var}$ there is also a primed

copy $x' \in \mathit{Var}$. The set of predicates Pred is defined as the set of all functions from states to booleans. We define the set of *single-state predicates*, SPred, as the set of predicates that do not have any free primed variable. With the pointwise extension of the ordering $\Rightarrow$ and the operators $\neg$, $\wedge$ and $\vee$ from $\mathbb{B}$, Pred and SPred become complete lattices. For two predicates P_1 and P_2 the *entailment ordering* $\Rrightarrow$ on Pred is implication for every state and $\equiv$ is equivalence for every state. A *predicate transformer* is defined as a function from predicates to predicates that is monotonic with respect to the entailment ordering. Further details on the predicate space are given in Appendix A. The following definitions summarise the above.

$$\begin{aligned} \Sigma &\stackrel{\text{def}}{=} \mathit{Var} \rightarrow \mathit{Val} \\ \mathit{Pred} &\stackrel{\text{def}}{=} \Sigma \rightarrow \mathbb{B} \\ P_1 \Rrightarrow P_2 &\stackrel{\text{def}}{=} \forall \sigma (\sigma \in \Sigma \Rightarrow (P_1(\sigma) \Rightarrow P_2(\sigma))) \\ P_1 \equiv P_2 &\stackrel{\text{def}}{=} (P_1 \Rrightarrow P_2) \wedge (P_2 \Rrightarrow P_1) \end{aligned}$$

2.2 The command language

Before defining our real-time programming language, we first define a set of primitive constructs in terms of which the rest of the language can be defined. Then we add contexts which allow variable declarations and typing information to be represented. Finally, we define our real-time language in terms of these primitives by adding a current time variable, τ.

Our primitive program constructs consist of the specification statement [29], here denoted $x{:}^{nt}[Q]$ where the "nt" superscript denotes 'nontimed' to distinguish it from the timed version below; an assumption about the program state denoted $\{P\}$; nondeterministic choice over some set of commands indexed by set $\mathcal{A}$, $(\sqcap_{i \in \mathcal{A}} S_i)$; and sequential composition of commands, $(S_1 \,;\, S_2)$. Table 1 states the syntax and weakest liberal precondition semantics for the primitive commands. In the postcondition predicate R we do not allow references to primed variable names, i.e., $R \in \mathit{SPred}$. The specification statement updates the variables in its frame x according to the predicate $Q \in \mathit{Pred}$. Note that the frame x stands for a (possibly empty) set of variables that may be changed by this statement. The primed set of variables x' denotes their final values. The predicate Q may have free unprimed variables and only the free primed variables x' corresponding to those that occur in the frame of the specification statement. In the semantics, variables x are renamed x' in predicate R to match the final state naming convention used in predicate Q. The specification statement can describe the behaviour of assignment and other primitive programming statements [29]. For a predicate $P \in \mathit{SPred}$, statement $\{P\}$ allows an assumption about the program state to be included. Nondeter-

Table 1
Weakest liberal precondition semantics for primitive commands.

Command S	$wlp(S, R)$
$x{:}^{nt}\,[Q]$	$\forall x'(Q \Rightarrow R[x'/x])$
$\{P\}$	$P \Rightarrow R$
$(\sqcap_{i\in\mathcal{A}} S_i)$	$\wedge_{i\in\mathcal{A}}\, wlp(S_i, R)$
$S_1 \,;\, S_2$	$wlp(S_1, wlp(S_2, R))$

Table 2
Commands extended with a context predicate C.

Command S	Equivalent to command
$(x{:}^{nt}\,[Q])^C$	$x{:}^{nt}\,[C \wedge C[x'/x] \wedge Q]$
$\{P\}^C$	$\{P \wedge C\}$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$(\sqcap_{i\in\mathcal{A}} S_i^C)$
$(S_1 \,;\, S_2)^C$	$S_1^C \,;\, S_2^C$
$(S_1^{C_1})^{C_2}$	$S_1^{C_1\wedge C_2}$

Table 3
Real-time commands.

Command S	Equivalent command
$x{:}\,[Q]$	$x, \tau{:}^{nt}\,[Q \wedge \tau \leq \tau']$
$[Q]$	$\emptyset{:}\,[Q]$

ministic choice over commands can be used to model conditional and iterative behaviour [2].

We extend the commands with a 'context' predicate which is used to record invariants concerning the surrounding program. Such invariants are typically type declarations for variables of the form $v \in T$. A command S is always seen in a certain (possibly *true*) context $C \in \mathit{SPred}$, expressed as S^C. Table 2 defines commands with contexts in terms of the primitive commands. For a specification statement the context is assumed to hold in the initial state and in the final state. Contexts in this sense are similar to the invariants introduced by Morgan and Vickers [31]. An assumption occurring in a particular context always assumes that the context predicate holds. Contexts are carried through nondeterministic choice, nesting and sequential composition in obvious ways.

As well as the commands in Table 2, a variable declaration $(\mathbf{var}\, v \,:T \bullet S_1)^C$ extends statement S_1's context C by introducing the fact that new variable v has type T, and that the original context C holds in the presence of this newly-declared variable. The type T of a variable must be a nonempty subset of *Val*. The weakest liberal precondition of a variable block is defined as follows:

$$wlp((\mathbf{var}\, v \,:T \bullet S_1)^C, R) \;\equiv\; (\forall\, v(wlp(S_1^{v\in T\wedge C[w/v]}, R[w/v])))[v/w]$$
$$\text{where } w \in \mathit{Independ}(R) \text{ and } w \notin \mathit{Idf}((\mathbf{var}\, v \,:T \bullet S_1)^C)$$

A substitution with a fresh variable w is performed in context C and predicate R to exclude unwanted bindings to an externally declared variable with name v, if any. In the proviso, $\mathit{Independ}(R)$ is the set of variable identifiers that the predicate R does not depend on (see Appendix A for more details)

and *Idf*(S) denotes those variable identifiers that appear free in command S (see Table A.1 for a full definition).

Table 3 extends the language further with the abstract syntax and semantics of a *timed specification statement*, $x{:}[Q]$. A timed specification statement is defined with the help of a conventional specification statement and an additional variable τ that refers to the current time and ranges over the real numbers $\mathbb{R}$. The timed specification statement implicitly has the reserved current time variable τ in its frame, plus the additional condition $\tau \leq \tau'$, making sure that the time never goes backwards. The frame x of the timed specification statement denotes a possibly empty set of variables that are distinct from τ. The timed specification statement is the primitive construct for 'timed' commands [19].

Definition 2.1 (Timed command) *A* timed command *denotes a command S^C, with a context C such that $C \Rrightarrow \tau \in \mathbb{R}$, that is constructed from variable declarations and the program constructors in the first column of Table 2 with timed specification statements instead of nontimed ones.*

Finally, coercions $[Q]$ are defined as timed specification statements with an empty frame.

Other commands may be modelled using timed commands. For example, the **idle** command, that changes no variables but may consume time, and a (timed) conditional command may be modelled as follows,

$$\begin{aligned} \mathbf{idle} &\stackrel{\text{def}}{=} [\mathit{true}] \\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} &\stackrel{\text{def}}{=} ([B]; S_1; \mathbf{idle}) \sqcap ([\neg B]; S_2; \mathbf{idle}) \end{aligned} \tag{1}$$

where we have used the infix operator "$\sqcap$" as a shorthand for a choice over two alternatives. Notice how this decomposes the conditional into its two possible paths. In dealing with the timing analysis of such commands it is desirable to treat each path separately because they may have quite different timing characteristics. Note that an individual path, such as $([B]; S_1; \mathbf{idle})$, is miraculous (or infeasible) in initial states in which B does not hold and hence that path is not followed. Further commands are defined in Table 6 and Table 7.

3 Command entry and traversal conditions

The execution time of a command, S, may depend on the initial state (of the program's variables). However, we need to consider two exceptional cases: in some states, (a) the command executes forever, or (b) the command is

miraculous [9]. In both cases we are not interested in the execution time because (a) the command does not terminate or (b) the command corresponds to a path that is not followed. In the weakest liberal precondition semantics these alternatives are not distinguished. They both correspond to the states satisfying $wlp(S, \mathit{false})$. Recall that $wlp(S, R)$ defines those initial states from which, if S terminates, it does so in a state that satisfies R. Hence $wlp(S, \mathit{false})$ corresponds to those initial states from which S does not terminate, or if it does, the state satisfies *false*, i.e., it is miraculous from that initial state. For execution-time analysis we are interested in the initial states that may lead to nonmiraculous termination, which we refer to as the *entry condition*, $\mathcal{E}(S)$, of command S.

Definition 3.1 (Entry condition) *The* entry condition*, $\mathcal{E}(S)$, of a command, S, is defined as the predicate $\neg wlp(S, \mathit{false})$.*

If there are no states that satisfy a command's entry condition it is a *dead command* [17].

Definition 3.2 (Dead command) *A command S is* dead *iff $\mathcal{E}(S) \equiv \mathit{false}$.*

For a sequential composition, $(S_1; S_2)$, the calculation of its entry condition $\mathcal{E}(S_1; S_2)$ gives the following.

$$\neg wlp((S_1; S_2), \mathit{false}) \equiv \neg wlp(S_1, wlp(S_2, \mathit{false})) \equiv \neg wlp(S_1, \neg \mathcal{E}(S_2)) \quad (2)$$

This motivates a generalisation of the entry condition that defines the *traversal condition* for a command S as a restriction of the entry condition to those states from which S can possibly achieve a state satisfying postcondition R.

Definition 3.3 (Traversal condition) *The* traversal condition $\mathcal{E}(S, R)$ *of a command S, for $R \in \mathit{SPred}$, is defined as the predicate $\neg wlp(S, \neg R)$.*

Note that $\mathcal{E}(S) \equiv \mathcal{E}(S, \mathit{true})$, i.e., the entry condition is the traversal condition to terminate in any state. The entry condition for the sequential composition (2) can now be written as $\mathcal{E}(S_1, \mathcal{E}(S_2, \mathit{true}))$. Note that $\mathcal{E}(S, R)$ is monotonic with respect to entailment on R, i.e., if $R \Rrightarrow R'$ then $\mathcal{E}(S, R) \Rrightarrow \mathcal{E}(S, R')$.

Theorem 3.4 (Traversal condition) *Table 4 defines the traversal condition $\mathcal{E}(S, R)$ for commands S and predicates $R \in \mathit{SPred}$.*

The proof is by expanding the definition of the traversal condition.

For example, let $B \in \mathit{SPred}$ with $\tau \in \mathit{Independ}(B)$. By assuming that the entry condition for S_1^C is C, the entry condition for the first alternative of the conditional (1) in context $C \stackrel{\text{def}}{=} \tau \in \mathbb{R}$ can be calculated as follows.

Table 4
Computation of the traversal condition $\mathcal{E}(S, R)$.

Command S	$\mathcal{E}(S, R)$
$(x{:}^{nt}\,[Q])^C$	$\exists x'\,(C \wedge (C \wedge R)[x'/x] \wedge Q)$
$\{P\}^C$	$P \wedge C \wedge R$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$(\exists v\, \mathcal{E}(S_1^{v\in T \wedge C[w/v]}, R[w/v]))[v/w]$ with $w \notin \mathit{Idf}(S)$ and $w \in \mathit{Independ}(R)$
$(\sqcap_{i\in\mathcal{A}} S_i)^C$	$\vee_{i\in\mathcal{A}} \mathcal{E}(S_i^C, R)$
$(S_1^{C_1})^{C_2}$	$\mathcal{E}(S_1^{C_1\wedge C_2}, R)$
$(S_1\ ;\ S_2)^C$	$\mathcal{E}(S_1^C, \mathcal{E}(S_2^C, R))$

$$\begin{aligned}\mathcal{E}(([B]; S_1; \mathbf{idle})^C, \mathit{true}) &\equiv \mathcal{E}([B]^C, \mathcal{E}(S_1^C, \mathcal{E}(\mathbf{idle}^C, \mathit{true})))\\ &\equiv \mathcal{E}([B]^C, \mathcal{E}(S_1^C, C))\\ &\equiv \mathcal{E}([B]^C, C)\\ &\equiv B \wedge C\end{aligned}$$

Similarly, assuming $\mathcal{E}(S_2^C) \equiv C$, the entry condition for the second alternative is $\neg B \wedge C$, and the entry condition for the whole conditional is $(B \wedge C) \vee (\neg B \wedge C)$ $(\equiv C)$.

4 Liberal command refinement.

We define a liberal program refinement relation on the command language via the weakest liberal precondition semantics by interpreting a command as a predicate transformer on the subspace consisting of all predicates in *SPred*. This refinement relation ensures partial correctness in program development. All functions on predicates that have been used to define the semantics of commands in Table 2 are monotone functions and the restrictions on the program statements ensure that the underlying predicate transformers map predicates from *SPred* to predicates in *SPred*.

Definition 4.1 (Liberal refinement) *Let S_1, S_2 be two commands. We say that S_2* liberally refines *S_1 and write $S_1 \sqsubseteq_{wlp} S_2$ if $wlp(S_1, R) \Rrightarrow wlp(S_2, R)$ for all predicates $R \in SPred$. Liberal refinement equivalence of S_1 and S_2 is denoted by $S_1 \sqsupseteq\sqsubseteq_{wlp} S_2$.*

Note that liberal refinement is different from common refinement relations based on weakest preconditions used in total correctness approaches [1,30]. A refinement $S \sqsubseteq_{wlp} S'$ means that the predicate characterising the entry condition of command S', is stronger than the one for command S, and furthermore,

that all states that command S' can achieve are possible ones for command S. Every liberal refinement decreases nondeterminism and strengthens the entry condition. Unlike refinement based on weakest preconditions (total correctness), liberal refinement may increase the domain of nontermination.

5 Symbolic computation of execution times

This section focuses on timed commands. We show how worst-case and best-case execution times for timed commands can be defined and computed on the basis of the underlying weakest liberal precondition semantics.

5.1 Infimum and supremum

Our theory relies on the notions of infimum and supremum. As with predicates (which are defined to be functions from states to booleans) our definitions of supremum and infimum are also functions from the state. We use the notation $(\sup\{\mathcal{V}|B\bullet\theta\})(\sigma)$ to denote the supremum (least upper bound) of function $\theta \in \Sigma \to (\mathbb{R}\cup\{\infty,-\infty\})$ over all states σ' where predicate B holds that may differ from state σ only in the variables $\mathcal{V}$. The infimum is defined similarly. Note that $\sup\{\mathcal{V}|B \bullet \theta\}$ is of type $\Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$. The domain $\mathbb{R}$ must be extended with infinite values ∞ and $-\infty$ here to account for situations where no finite bound exists. In particular, the supremum (infimum) of the empty set is defined to be $-\infty$ (∞). We also use the notation $(\sup_{\theta\in\Omega}\theta)(\sigma)$ to denote the supremum of the set of expressions $\theta(\sigma)$, for θ ranging over a set of functions Ω, and a similar notation for the infimum. Appendix A contains the formal definition of these notations.

5.2 Defining time bounds

As a motivational example the assignment of value 1 to variable x taking between one and two time units can be specified as the timed specification statement

$$x\colon [x' = 1 \wedge 1 \leq \tau' - \tau \leq 2]. \tag{3}$$

This command has 'worst-case' execution time 2 and 'best-case' execution time 1. To formally derive the execution time of a timed command, S, we introduce a fresh variable, ξ, to stand for the termination time of the command.

Then $\mathcal{E}(S, \xi = \tau)$ is a predicate (in ξ and the initial state variables) representing those initial states from which S can possibly terminate at time ξ. The current time in the initial state is τ, and hence $\xi - \tau$ represents the execution time of S. To determine the worst-case (best-case) execution time, we take the supremum (infimum) over ξ of $\xi - \tau$ for states satisfying the $\mathcal{E}(S, \xi = \tau)$. Let $\mathcal{W}(S)$ and $\mathcal{B}(S)$ be the worst and best-case execution times of command S, respectively.

$$\begin{aligned} \mathcal{W}(S) &\stackrel{\text{def}}{=} \sup\{\xi \mid \mathcal{E}(S, \xi = \tau) \bullet \xi - \tau\} \\ \mathcal{B}(S) &\stackrel{\text{def}}{=} \inf\{\xi \mid \mathcal{E}(S, \xi = \tau) \bullet \xi - \tau\} \end{aligned} \tag{4}$$

For the example (3) above we get its worst-case execution time, $\mathcal{W}(S)$, as

$$\begin{aligned} & \sup\{\xi \mid \mathcal{E}(x{:}\,[x' = 1 \wedge 1 \leq \tau' - \tau \leq 2], \xi = \tau) \bullet \xi - \tau\} \\ =& \sup\{\xi \mid \exists x', \tau'(x' = 1 \wedge 1 \leq \tau' - \tau \leq 2 \wedge \xi = \tau') \bullet \xi - \tau\} \\ =& \sup\{\xi \mid 1 \leq \xi - \tau \leq 2 \bullet \xi - \tau\} \\ =& 2 \end{aligned}$$

Similarly, its best-case execution time can be calculated as 1. $\mathcal{W}(S)$ and $\mathcal{B}(S)$ are expressions on the initial state (including τ). To calculate the worst-case execution time over all initial states one needs to take the supremum of $\mathcal{W}(S)(\sigma)$ over all states σ. For the example (3) above $\mathcal{W}(S)$ is independent of the state and hence $\mathcal{W}(S)(\sigma)$ is 2 for all states.

If S is dead then $\mathcal{E}(S, \mathit{true}) \equiv \mathit{false}$, and because the traversal condition is monotone on *SPred*, $\mathcal{E}(S, \xi = \tau) \Rrightarrow \mathcal{E}(S, \mathit{true})$ and hence $\mathcal{E}(S, \xi = \tau) \equiv \mathit{false}$. Therefore the worst-case execution time for a dead command is given by

$$\sup\{\xi \mid \mathcal{E}(S, \xi = \tau) \bullet \xi - \tau\} = \sup\{\xi \mid \mathit{false} \bullet \xi - \tau\} \;=\; -\infty$$

The determination of the worst-case execution time of a sequential composition, $(S_1; S_2)$, is, unfortunately, not as simple as summing the respective worst-case execution times of the two statements. There are two reasons for this. Firstly, S_1 may terminate in a state that does not satisfy the entry condition for S_2. Hence when determining the worst-case execution time for S_1, we need to restrict our attention to initial states from which S_1 can achieve states satisfying the entry condition for S_2, that is, initial states satisfying $\mathcal{E}(S_1, \mathcal{E}(S_2))$. Secondly, the worst-case execution time of S_2 depends on its initial state. The predicate

$$\mathcal{E}(S_1, \mathcal{E}(S_2) \wedge \xi = \tau + \mathcal{W}(S_2))$$

characterises those initial states from which S_1 can reach a state σ which (a) satisfies the entry condition for S_2, and (b) in which ξ is equal to the worst-case execution time of S_2 from state σ plus the termination time of S_1, i.e., τ. Hence

$$\mathcal{W}(S_1; S_2) = \sup\{\xi | \mathcal{E}(S_1, \mathcal{E}(S_2) \wedge \xi = \tau + \mathcal{W}(S_2)) \bullet \xi - \tau\}. \tag{5}$$

Theorem 5.2 proves that this corresponds with our definition (4) of worst-case execution time.

5.3 Calculating time bounds

To allow systematic (symbolic) calculation of the worst-case execution time of a sequence (or path) of commands, we generalise function $\mathcal{W}(S_1)$ to function $\mathcal{W}(S_1, R, \theta)$, which represents the worst-case execution time of the path consisting of S_1 followed by some path S_2, where R is a predicate representing the entry condition for path S_2 and θ represents the worst-case execution time of path S_2. Note that θ is a function of the state. With this definition, the worst-case execution time in equation (5) can be written as $\mathcal{W}(S_1, \mathcal{E}(S_2), \mathcal{W}(S_2))$.

Definition 5.1 (Execution-time bounds) *Let S be a timed command, let $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$ and let $R \in SPred$ and let $\xi \in Var$ with $\xi \notin Idf(S)$ and $\xi \in Independ(R) \cap Independ(\theta)$. The best-case and worst-case execution time expressions $\mathcal{B}(S, R, \theta)$ and $\mathcal{W}(S, R, \theta)$ are of type $\Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$ and are defined by*

$$\mathcal{B}(S, R, \theta) \stackrel{\text{def}}{=} \inf\{\xi \mid \mathcal{E}(S, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\}, \text{ and}$$
$$\mathcal{W}(S, R, \theta) \stackrel{\text{def}}{=} \sup\{\xi \mid \mathcal{E}(S, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\}.$$

Note that $\mathcal{W}(S) = \mathcal{W}(S, \mathit{true}, 0)$ and $\mathcal{B}(S) = \mathcal{B}(S, \mathit{true}, 0)$, where "0" here represents the function that is zero in all states.

Table 5 provides a method for the symbolic computation of *worst-case* execution times of timed commands.

Theorem 5.2 (Execution-time bounds) *Let S be a timed command, $R \in SPred$ and $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$, then Table 5 defines $\mathcal{W}(S, R, \theta)$.*

Proof. We prove the assertion of the theorem by induction on the structure of a timed command S^C with context C. The proof for specification statements $x{:}\,[Q]$ is as follows.

Table 5
Computation of the worst-case execution time $\mathcal{W}(S, R, \theta)$.

Command S	$\mathcal{W}(S, R, \theta)$
$(x\colon [Q])^C$	$\sup\{\tau', x' \mid C \wedge (C \wedge R)[x', \tau'/x, \tau] \wedge Q \wedge \tau \leq \tau' \bullet$ $\theta[x', \tau'/x, \tau] + \tau' - \tau\}$
$\{P\}^C$	$\sup\{\xi \mid P \wedge C \wedge R \wedge \xi = \tau + \theta \bullet \xi - \tau\}$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$\sup\{v \mid \mathcal{W}(S_1^{v \in T \wedge C[w/v]}, R[w/v], \theta[w/v])\}[v/w]$ with $w \notin \mathit{Idf}(S)$ and $w \in \mathit{Independ}(R) \cap \mathit{Independ}(\theta)$
$(\sqcap_{i \in \mathcal{A}} S_i)^C$	$\sup_{i \in \mathcal{A}} \mathcal{W}(S_i^C, R, \theta)$
$(S_1^{C_1})^{C_2}$	$\mathcal{W}(S_1^{C_1 \wedge C_2}, R, \theta)$
$(S_1\ ;\ S_2)^C$	$\mathcal{W}(S_1^C, \mathcal{E}(S_2^C, R), \mathcal{W}(S_2^C, R, \theta))$

$$
\begin{aligned}
&\mathcal{W}((x\colon [Q])^C, R, \theta)\\
=&\sup\{\xi \mid \mathcal{E}((x\colon [Q])^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\}\\
=&\text{ “By Table 4”}\\
&\sup\{\xi \mid \exists \tau', x'\, (C \wedge (C \wedge R)[x', \tau'/x, \tau] \wedge Q \wedge \tau \leq \tau' \wedge\\
&\quad \xi = \tau' + \theta[x', \tau'/x, \tau] \bullet \xi - \tau\}\\
=&\text{ “By Theorem C.1”}\\
&\sup\{\tau', x' \mid C \wedge (C \wedge R)[x', \tau'/x, \tau] \wedge Q \wedge \tau \leq \tau' \bullet\\
&\quad \tau' + \theta[x', \tau'/x, \tau] - \tau\}
\end{aligned}
$$

The proof for assumptions $\{P\}$ is trivial. Next we show the induction step for variable declarations. We fix a variable w with $w \notin \mathit{Idf}((\mathbf{var}\, v\ : T \bullet S_1)^C)$ and $w \in \mathit{Independ}(R) \cap \mathit{Independ}(\theta)$.

$$
\begin{aligned}
&\mathcal{W}((\mathbf{var}\, v\ : T \bullet S_1)^C, R, \theta)\\
=&\sup\{\xi \mid \mathcal{E}((\mathbf{var}\, v\ : T \bullet S_1)^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\}\\
=&\text{ “By Table 4”}\\
&\sup\{\xi \mid \left(\exists v\, \mathcal{E}(S_1^{v \in T \wedge C[w/v]}, R[w/v] \wedge \xi = \tau + \theta[w/v])\right)[v/w] \bullet \xi - \tau\}\\
=&\text{ “By Theorem C.1”}\\
&\sup\{v \mid \sup\{\xi \mid (\mathcal{E}(S_1^{v \in T \wedge C[w/v]}, R[w/v] \wedge \xi = \tau + \theta[w/v])) \bullet\\
&\quad \xi - \tau\}\}[v/w]\\
=&\text{ “By induction hypothesis”}\\
&\sup\{v \mid \mathcal{W}(S_1^{v \in T \wedge C[w/v]}, R[w/v], \theta[w/v])\}[v/w]
\end{aligned}
$$

The induction step for nondeterministic choice $(\sqcap_{i \in \mathcal{A}} S_i)$ is as follows.

$$
\mathcal{W}((\sqcap_{i \in \mathcal{A}} S_i)^C, R, \theta) = \sup\{\xi \mid \mathcal{E}((\sqcap_{i \in \mathcal{A}} S_i)^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\}
$$

$$
\begin{aligned}
&= \text{“By Table 4”} \\
&\quad \sup\{\xi \mid \vee_{i\in\mathcal{A}}\, \mathcal{E}(S_i^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\} \\
&= \sup_{i\in\mathcal{A}} \sup\{\xi \mid \mathcal{E}(S_i^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\} \\
&= \text{“By induction hypothesis”} \\
&\quad \sup_{i\in\mathcal{A}} \mathcal{W}(S_i^C, R, \theta)
\end{aligned}
$$

It remains to show the induction step for sequential composition, $(S_1 \;;\; S_2)$. According to Theorem B.5 we can assume that there is a timed specification statement $x{:}\,[Q]$ which is liberal refinement equivalent to S_1^C.

$$
\begin{aligned}
&\quad \mathcal{W}(S_1^C, \mathcal{E}(S_2^C, R), \mathcal{W}(S_2^C, R, \theta)) \\
&= \text{“By Definition 5.1 of } \mathcal{W} \text{ on } S_1^C\text{”} \\
&\quad \sup\{\xi \mid \mathcal{E}(S_1^C, \mathcal{E}(S_2^C, R) \wedge \xi = \tau + \mathcal{W}(S_2^C, R, \theta)) \bullet \xi - \tau\} \\
&= \text{“By Definition 5.1 of } \mathcal{W} \text{ on } S_2^C\text{”} \\
&\quad \sup\{\xi \mid \mathcal{E}(S_1^C, \mathcal{E}(S_2^C, R) \wedge \\
&\qquad \xi = \tau + \sup\{\eta \mid \mathcal{E}(S_2^C, R \wedge \eta = \tau + \theta) \bullet \eta - \tau\}) \bullet \xi - \tau\} \\
&= \text{“By Table 4 assuming } S_1^C \text{ is equivalent to } x{:}\,[Q]\text{”} \\
&\quad \sup\{\xi \mid \exists \tau', x' (Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R)[\tau', x'/\tau, x] \wedge \\
&\qquad \xi = \tau' + \sup\{\eta \mid \mathcal{E}(S_2^C, R \wedge \eta = \tau + \theta)[\tau', x'/\tau, x] \bullet \eta - \tau'\}) \bullet \xi - \tau\} \\
&= \text{“By Theorem C.1; also cancelling out } \tau'\text{”} \\
&\quad \sup\{\xi, \tau', x' \mid Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R)[\tau', x'/\tau, x] \wedge \\
&\qquad \xi = \sup\{\eta \mid \mathcal{E}(S_2^C, R \wedge \eta = \tau + \theta)[\tau', x'/\tau, x] \bullet \eta\} \bullet \xi - \tau\} \\
&= \sup\{\tau', x' \mid Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R)[\tau', x'/\tau, x] \bullet \\
&\qquad \sup\{\xi \mid \mathcal{E}(S_2^C, R \wedge \xi = \tau + \theta)[\tau', x'/\tau, x] \bullet \xi\} - \tau\} \\
&= \sup\{\xi, \tau', x' \mid Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R)[\tau', x'/\tau, x] \wedge \\
&\qquad \mathcal{E}(S_2^C, R \wedge \xi = \tau + \theta)[\tau', x'/\tau, x] \bullet \xi - \tau\} \\
&= \text{“By monotonicity of the command traversal condition”} \\
&\quad \sup\{\xi, \tau', x' \mid Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R \wedge \xi = \tau + \theta)[\tau', x'/\tau, x] \bullet \xi - \tau\} \\
&= \text{“By Theorem C.1”} \\
&\quad \sup\{\xi \mid \exists \tau', x' (Q \wedge \tau \le \tau' \wedge \mathcal{E}(S_2^C, R \wedge \xi = \tau + \theta)[\tau', x'/\tau, x]) \bullet \xi - \tau\} \\
&= \text{“By Table 4 assuming } S_1^C \text{ is equivalent to } x{:}\,[Q]\text{”} \\
&\quad \sup\{\xi \mid \mathcal{E}(S_1^C, \mathcal{E}(S_2^C, R \wedge \xi = \tau + \theta)) \bullet \xi - \tau\} \\
&= \text{“By Table 4”} \\
&\quad \sup\{\xi \mid \mathcal{E}((S_1 \;;\; S_2)^C, R \wedge \xi = \tau + \theta) \bullet \xi - \tau\} \\
&= \text{“By Definition 5.1”} \\
&\quad \mathcal{W}((S_1 \;;\; S_2)^C, R, \theta)
\end{aligned}
$$

□

A similar theorem holds for the best-case execution time: by replacing the expression $\mathcal{W}(S, R, \theta)$ with $\mathcal{B}(S, R, \theta)$ and the *supremum* with the *infimum* in all expressions in Table 5 we obtain a method for the symbolic computation of best-case execution times.

A consequence of the definition of timed commands, worst-case and best-case execution times and the previous theorem are the following correspondences.

Theorem 5.3 (Achievable execution times) *For every timed command S and predicate $R \in SPred$, it is the case that*

- $\mathcal{B}(S, R, 0) \geq 0$,
- $\mathcal{E}(S, R) \equiv (\mathcal{B}(S, R, 0) \leq \mathcal{W}(S, R, 0))$,
- $\neg\mathcal{E}(S, R) \equiv \mathcal{B}(S, R, 0) = \infty$, *and*
- $\neg\mathcal{E}(S, R) \equiv \mathcal{W}(S, R, 0) = -\infty$.

In other words, no command takes a negative time, the best-case execution time never exceeds the worst-case one on states that fulfil the traversal condition, and the execution times are unsatisfiable on states that do not fulfil the traversal condition.

5.4 Refining time bounds

Any liberal refinement $S \sqsubseteq_{wlp} S'$ always tightens the worst-case and the best-case execution times of a timed command S in the sense that the worst-case execution time of S' is below the one for S and the best-case execution time of S' is above the one for S. This is stated in the following theorem which is derived from Definition 5.1 and the fact that $\mathcal{E}(S', R) \Rrightarrow \mathcal{E}(S, R)$ for any predicate $R \in SPred$.

Theorem 5.4 (Time-bound refinement) *Let S, S' be timed commands with $S \sqsubseteq_{wlp} S'$, let $R \in SPred$ and let θ be a function on Σ with values in $\mathbb{R} \cup \{-\infty, \infty\}$. Then, $\mathcal{B}(S, R, \theta) \leq \mathcal{B}(S', R, \theta)$ and $\mathcal{W}(S', R, \theta) \leq \mathcal{W}(S, R, \theta)$.*

This shows that program development with liberal refinement (partial correctness) preserves upper bounds for worst-case and lower bounds for best-case execution-time behaviour. This is not necessarily the case for refinement based on weakest preconditions (total correctness), since states from where the program may not terminate can then become states from where anything can happen.

Table 6
High-level language commands and equivalent modelling language commands.

Command S	Wlp-command equivalent
x `:=` E	$x{:}\,[x' = E]$
`deadline` D	$[\tau' = \tau \wedge \tau \leq D]$
`delay until` D	$[D \leq \tau' \leq max\{\tau, D\} + late]$
`while` G `do` S `end`	$(\sqcap_{i \in \mathbb{N}} S_i \ ; \ [\neg G])$
	where $S_0 \stackrel{\text{def}}{=} [\tau' = \tau]$ and $S_{i+1} \stackrel{\text{def}}{=} [G] \ ; \ S \ ; \ S_i$, for $i \in \mathbb{N}$

6 Example

This section applies our techniques to the timing analysis of a 'message transmitter' example. Timing constraints are extracted from the high-level program, and the compiled code is evaluated against those. This is performed by giving a common semantic model for the high-and low-level programs and investigating timing paths through the programs.

Table 6 defines the syntax and semantics of a set of language commands for a high-level real-time language with conventional program primitives and loops. The timing behaviour of a real-time program in this language can be specified with the help of the 'delay' and 'deadline' commands. The delay statement delays the program's execution until a specified time with possible maximal overrun $late > 0$, and the deadline command requires that the preceding statements finish before a certain time [13]. Furthermore, there is a (possibly time consuming) assignment to program variables. All commands are defined on the base of the language for timed commands. The definitions of the while loop have been taken from previous work [17] and re-expressed with the general nondeterministic choice command of Table 2. In the table, G denotes a predicate with no free primed variables, and D and E denote expressions without free primed variables. The expression E, time-valued expression D, and predicate G must all be idle-stable, that is, their values cannot change with just the passage of time. This means that they cannot refer to the current time variable, τ, or to the value of external inputs. The identifier x denotes a local variable.

6.1 The message transmitter

To illustrate the execution time computation techniques on a high-level real-time program, Figure 2 contains a simplified version of a program which displays a message of *size* characters, one character at a time, in a shared memory location, *out*, under the constraint that the first character must be displayed

```
var msg : array(0..size − 1) of nat;
    out : nat;
    n : nat;
⋮
{τ ≤ 30};
n := 0;
while n ≠ size do
  out := msg(n);
  deadline 50 + 35 ∗ n;
  delay until 56 + 35 ∗ n;
  n := n + 1
end ;
deadline 56 + 35 ∗ size
```

Fig. 2. High-level language transmitter program.

from (at least) time 50 to time 56, the second character must be displayed from time 85 to time 91, and so on [18]. A new character appears every 35 microseconds, and remains visible for at least 6 microseconds. The *delay until* statement forces the character to remain visible for the required time, and the *deadline* states that the program must have produced the character by the required time.

The challenge when verifying such a program is to show that the timing of each control-flow path ending at a deadline command ensures that the deadline will always be met. As an example of a timing path in this program, we investigate the control-flow path that starts with the assumption at the beginning of the program, enters the while-loop for the first time and ends with the first deadline command. We denote this path by *Path A*.

```
{τ ≤ 30};
n := 0;
[n ≠ size];
out := msg(n);
deadline 50 + 35 ∗ n
```

Note that in *Path A*, the coercion statement $[n \neq size]$ records the condition that has to be true for the loop to be entered [17].

This is, of course, only one of the many paths through such a program. In particular, to successfully prove the timing correctness of an embedded program it is usually necessary to prove that each process iterates quickly enough to process data at the correct rate. For instance, in this case we would also want to analyse the path that goes from the deadline of $50 + 35 * n$, around the 'loop', and back to itself, in order to prove that the task can achieve its

required periodicity. We do not go further into the issue of finding all relevant timed paths and the problem of extracting a minimal set of primitive paths that have to be analysed to derive a sufficient set of timing constraints [15,20].

The following predicate defines the program context C for $Path\ A$.

$$C \stackrel{\text{def}}{=} \tau \in \mathbb{R} \wedge out, n \in \mathbb{N} \wedge msg \in \{0, .., size - 1\} \to \mathbb{N}$$

In this context (which is implicit below) we compute the traversal condition of $Path\ A$ to achieve $\xi = \tau$ as follows, by calculating backwards up the path with the help of Table 4.

$$\begin{aligned}
&\mathcal{E}(\texttt{deadline}\ 50 + 35 * n, \xi = \tau) \\
\equiv\ &\tau \leq 50 + 35 * n \wedge \xi = \tau \\
&\mathcal{E}(out := msg(n), \tau \leq 50 + 35 * n \wedge \xi = \tau) \\
\equiv\ &\exists \tau', out' \, (\tau \leq \tau' \wedge out' = msg(n) \wedge \tau' \leq 50 + 35 * n \wedge \xi = \tau') \\
\equiv\ &\tau \leq \xi \leq 50 + 35 * n \\
&\mathcal{E}([n \neq size], \tau \leq \xi \leq 50 + 35 * n) \\
\equiv\ &\exists \tau' \, (\tau \leq \tau' \wedge n \neq size \wedge \tau' \leq \xi \leq 50 + 35 * n) \\
\equiv\ &n \neq size \wedge \tau \leq \xi \leq 50 + 35 * n \\
&\mathcal{E}(n := 0, n \neq size \wedge \tau \leq \xi \leq 50 + 35 * n) \\
\equiv\ &\exists \tau', n' \, (\tau \leq \tau' \wedge n' = 0 \wedge n' \neq size \wedge \tau' \leq \xi \leq 50 + 35 * n') \\
\equiv\ &\tau \leq \xi \leq 50 \wedge 0 \neq size \\
&\mathcal{E}(\{\tau \leq 30\}, \tau \leq \xi \leq 50 \wedge 0 \neq size) \\
\equiv\ &\tau \leq 30 \wedge \tau \leq \xi \leq 50 \wedge 0 \neq size \\
\equiv\ &\mathcal{E}(Path\ A, \xi = \tau)
\end{aligned}$$

This is exactly the condition we require to traverse this path with termination time ξ. The starting time τ is not later than 30 and the length of the message is not 0 (which is necessary to enter the loop at least once) and ξ must be not greater than 50. The worst-case execution time of $Path\ A$ is computed with the help of Definition 5.1 as follows.

$$\begin{aligned}
\mathcal{W}(Path\ A) &= \sup\{\xi \mid \mathcal{E}(Path\ A, \xi = \tau) \bullet \xi - \tau\} \\
&= \sup\{\xi \mid \tau \leq 30 \wedge \tau \leq \xi \leq 50 \wedge 0 \neq size \bullet \xi - \tau\} \\
&= \begin{cases} 50 - \tau & : \quad \tau \leq 30 \wedge 0 \neq size \\ -\infty & : \quad \text{otherwise} \end{cases}
\end{aligned}$$

To interpret this result, consider that the path is not permitted to start later than time 30. The computed expression $50 - \tau$ specifies the uppermost execution time of the path with respect to the starting time τ. In the worst

```
     --{τ ≤ 30}
20:  li $0, msg          $0 := msg base address
21:  li $1, size         $1 := size (length of msg)
22:  li $2, 0            $2 := 0 (loop counter n )
23:  li $3, 56           $3 := 56 (first delay time)
24:  li $4, 35           $4 := 35 (separation time)
25:  j 34                goto loop test
26:  add $5, $2, $0      $5 := n + msg base
27:  lb $6, ($5)         $6 := msg(n)
28:  sb $6, out          out := msg(n)
     -- deadline $3 − 6
29:  lt $7, clock        $7 := current time
30:  sub $7, $3, $7      $7 := delay time − current time
31:  bgtz $7, 29         delay if $7 > 0
32:  add $3, $3, $4      delay time := delay time + 35
33:  addi $2, $2, 1      increment n
34:  bne $1, $2, 26      goto loop body if n ≠ size
```

Fig. 3. MIPS-like assembler code of the message transmitter.

case, when the program starts at time 30, the path can take no more than 20 microseconds. Importantly, if we can prove that an implementation of this path never takes more than 20 time units, the deadline at the end will always be met for any anticipated starting time. However, if the path starts too late or *size* equals zero, then there is no timing obligation on the path.

6.2 Analysing the transmitter implementation

Figure 3 depicts an implementation of the transmitter program in MIPS-like assembler language. Here *msg* and *out* are symbolic constants representing the addresses allocated to these high-level language variables. Expression $i denotes register number i and function *clock* denotes the machine clock. The *clock* is modelled as an input to the program which keeps track of the actual time. However, since the instruction that samples the clock takes 1 microsecond the sampled time *clock* is slightly behind the actual time τ. The assembler

Table 7
Assembler commands and defining modelling language commands.

Assembler comd.	Equivalent modelling language command
lb \$i, a	$r, pc\colon [r' = r \oplus \{i \mapsto m(a)\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
lb \$i, (\$j)	$r, pc\colon [r' = r \oplus \{i \mapsto m(r(j))\} \land 1 \leq \tau' - \tau \leq 3 \land pc' = pc + 1]$
li \$i, z	$r, pc\colon [r' = r \oplus \{i \mapsto z\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
lt \$i, *clock*	$r, pc\colon [r' = r \oplus \{i \mapsto clock(\tau)\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
sb \$i, a	$pc, m\colon [m' = m \oplus \{a \mapsto r(i)\} \land \tau' = \tau + 2 \land pc' = pc + 1]$
j a	$pc\colon [\tau' = \tau + 1 \land pc' = a]$
bgtz \$i, a	$pc\colon [(r(i) \leq 0 \Rightarrow pc' = pc + 1) \land (r(i) > 0 \Rightarrow pc' = a) \land$ $1 \leq \tau' - \tau \leq 2]$
bne \$i, \$j, a	$pc\colon [(r(i) \neq r(j) \Rightarrow pc' = a) \land (r(i) = r(j) \Rightarrow pc' = pc + 1) \land$ $1 \leq \tau' - \tau \leq 2]$
sub \$i, \$j, \$k	$r, pc\colon [r' = r \oplus \{i \mapsto r(j) - r(k)\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
add \$i, \$j, \$k	$r, pc\colon [r' = r \oplus \{i \mapsto r(j) + r(k)\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
addi \$i, \$j, z	$r, pc\colon [r' = r \oplus \{i \mapsto r(j) + z\} \land \tau' = \tau + 1 \land pc' = pc + 1]$
$l\colon S$	$[pc = l \land \tau = \tau'] \,;\, S$

Address a, registers \$i, \$j, \$k, integer z and program counter address l.

commands are defined in Table 7 on the base of our modelling language. We assume an abstract machine with a register function r on the set of registers $Reg \stackrel{\text{def}}{=} \{0, ..., 31\}$ and a memory function m on the set of addresses (natural numbers) $Addr$, both with values in the set of machine representations (integers) Int. Every assembler command S has a machine address l in $Addr$ denoted by $l\colon S$. The program counter pc ranges over these addresses. In addition, execution times have been given to each of the commands. For simplicity, we have not included any caching or pipelining effects. The following defines the context of the assembler program in Figure 3.

$$C \stackrel{\text{def}}{=} \tau \in \mathbb{R} \land msg, out, pc \in Addr \land m \in Addr \to Int \land r \in Reg \to Int \land clock \in \mathbb{R} \to Int \land Addr \subseteq Int \land \tau - 1 \leq clock(\tau) \leq \tau$$

On the base of the timing behaviour of the assembler program we can now compare the high-level timing constraints of the transmitter with the actual execution times of its implementation. The following control-flow path through the assembler code corresponds to $Path\,A$ of the high-level program. We call this program path $Path\,B$.

$$
\begin{array}{ll}
 & \texttt{--}\{\tau \leq 30\} \\
20: & \textbf{li}\,\$0, msg \\
21: & \textbf{li}\,\$1, size \\
22: & \textbf{li}\,\$2, 0 \\
23: & \textbf{li}\,\$3, 56 \\
24: & \textbf{li}\,\$4, 35 \\
25: & \textbf{j}\,34 \\
34: & \textbf{bne}\,\$1, \$2, 26 \\
26: & \textbf{add}\,\$5, \$2, \$0 \\
27: & \textbf{lb}\,\$6, (\$5) \\
28: & \textbf{sb}\,\$6, out \\
 & \texttt{-- deadline } \$3 - 6
\end{array}
$$

The computation of the traversal condition of $Path\,B$ is outlined below. Any command with machine address i is referenced under the name S_i. The equivalences hold under context C.

$$
\begin{aligned}
 & \mathcal{E}(S_{28}, \xi = \tau) \\
\equiv\, & \exists m', pc', \tau' (pc = 28 \wedge m' = m \oplus \{out \mapsto r(6)\} \wedge \tau' = \tau + 2 \wedge \\
 & \quad pc' = 29 \wedge \xi = \tau') \\
\equiv\, & pc = 28 \wedge \xi = \tau + 2 \\
 & \vdots \\
 & \mathcal{E}(S_{20}, pc = 21 \wedge m(size) \neq 0 \wedge 10 \leq \xi - \tau \leq 13) \\
\equiv\, & pc = 20 \wedge m(size) \neq 0 \wedge 11 \leq \xi - \tau \leq 14 \\
\equiv\, & \mathcal{E}(Path\,B)
\end{aligned}
$$

This tells us that in order to traverse $Path\,B$ the message must have some content and the program counter must point to the start of the path at address 20. The worst-case execution time of $Path\,B$ is then computed as follows.

$$
\begin{aligned}
\mathcal{W}(Path\,B) &= \sup\{\xi \mid pc = 20 \wedge m(size) \neq 0 \wedge 11 \leq \xi - \tau \leq 14 \bullet \xi - \tau\} \\
&= \begin{cases} 14 & : \quad pc = 20 \wedge m(size) \neq 0 \\ -\infty & : \quad \text{otherwise} \end{cases}
\end{aligned}
$$

This tells us that the worst-case execution time of $Path\,B$ is 14 microseconds, in those cases where the path's behaviour is well-defined. Since this is below the timing constraint 20 that we computed for the functionally equivalent $Path\,A$, we can conclude that $Path\,B$ will always meet the deadline of the high-level program. The implementation's timing behaviour is correct for this control-flow path.

7 Conclusion

We have defined a semantics for timed program commands and used it to define calculations for traversal conditions and execution-time bounds. We have shown how this approach can be used as a basis for timing analysis of high-level language real-time programs, where semantic extraction of the 'specified' worst-case and best-case execution times is needed to support timing verification of the machine-dependent implementation. We have shown that the techniques can be likewise applied in low-level program analysis with machine languages, where commands may have machine-specific timing bounds defined. In this context, our formal approach can be then used for the derivation of 'actual' worst-case and best-case execution times.

The expressive power of our model in terms of nondeterminism and nontermination needs some discussion. Our partial correctness semantics does not allow us to distinguish program states from which either a terminating or nonterminating behaviour is possible from states where only termination is possible. This lack of expressiveness is irrelevant for the computation of worst-case and best-case execution times, since their derivation is relevant only for terminating program executions: in a nondeterministic choice between nontermination and termination our model disregards nontermination, and permits the computation of execution times for the terminating alternative alone. Total correctness semantics would only permit the computation of execution times for 'strictly' terminating programs.

The symbolic computation of execution times that we describe in Theorem 5.2 looks tedious and repetitive on paper, but can be automated with techniques from Linear Programming. This approach permits the approximation of execution times for programs in arbitrary programming languages. A prototype tool using this approach has been implemented in Prolog on the base of linear programming techniques. The tool approximates the supremum (infimum) expressions of Table 5 by transforming the predicates into 'linear' predicates and translating the supremum (infimum) computations over the reals into linear optimisation problems. As a result of this, we obtain upper and lower bounds for execution times for arbitrary timed commands in our language [26].

Generally, though, the model is independent of the particular target machine and applies to any programming language with a predicate transformer semantics. In particular, the model provides a sound basis for verification of practical algorithms and tools for worst-case and best-case execution-time prediction.

Acknowledgements This research was conducted as part of ARC Large Grant A49937045, *Effective Real-Time Program Analysis*, and this article was

prepared as part of ARC Discovery Grant DP0209722, *Derivation and Timing Analysis of Concurrent Real-Time Software.*

A The predicate space

Predicates in the space *Pred* may have the entire set *Var* as free variables. Such a predicate would not permit variable substitution with variables in the variable universe *Var* in the usual sense. To avoid pathological cases of this kind we need to make sure that predicates cannot have more than a certain number of free variables. For a given cardinal number γ we therefore introduce the set $Pred_\gamma$ which consists of all predicates P in *Pred* such that the cardinality of the set of all free variables of P is not greater than γ. If γ is a limit ordinal, then $Pred_\gamma$ is closed under infimum (conjunction) and supremum (disjunction) on sets of predicates from $Pred_\gamma$, i.e., for predicates P_i from $Pred_\gamma$, with index i ranging over an index set $\mathcal{A}$ of cardinality not greater than γ, it is the case that $\wedge_{i\in\mathcal{A}} P_i$ and $\vee_{i\in\mathcal{A}} P_i$ are predicates in $Pred_\gamma$. This in turn ensures that the definition of nondeterministic choice is well defined.

For a state $\sigma \in \Sigma$, a subset of variables $\mathcal{V} \subseteq Var$ and a mapping $\mu \in \mathcal{V} \to Val$ we define overriding of σ by μ, denoted $\sigma \oplus \mu$, by $(\sigma \oplus \mu)(v) \stackrel{\text{def}}{=} \sigma(v)$ if $v \in Var \setminus \mathcal{V}$ and by $(\sigma \oplus \mu)(v) \stackrel{\text{def}}{=} \mu(v)$ if $v \in \mathcal{V}$. A function $f \in \Sigma \to Val$ may be *independent* of a variable v meaning that for every state $\sigma \in \Sigma$ and value $r \in Val$ the equality $f(\sigma) = f(\sigma \oplus \{v \mapsto r\})$ holds, where $\{v \mapsto r\}$ denotes the mapping of a variable v to a value r. The set of all independent variables of f is denoted by $Independ(f)$. Variable v is called a *free variable* of f if $v \in Var \setminus Independ(f)$.

The term algebra is defined as usual from the set of variables *Var* and the set of function symbols *Fun* which represent total functions of a certain arity over the value space *Val*. Substitution of a term t for a variable v in a predicate or function P on Σ is defined by $P[t/v](\sigma) \stackrel{\text{def}}{=} P(\sigma \oplus \{v \mapsto \sigma(t)\})$ where the evaluation of a state σ on a term means the canonical extension of σ onto the term algebra. By $P[y', z'/y, z]$ we denote the simultaneous substitution of y' and z' for y and z, whereas $P[y'/y][z'/z]$ denotes the substitution of y' for y followed by the substitution of z' for z.

The function *Idf* gives the set of identifiers used in a command; it is defined in Table A.1.

Definition A.1 (Supremum) *Let $\mathcal{V} \subseteq Var$, $B \in Pred$ and $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$. For $\sigma \in \Sigma$, let the expression $(\sup\{\mathcal{V} | B \bullet \theta\})(\sigma)$ denote the supremum of the set*

Table A.1
Sets of identifiers occuring in commands.

Command S	Identifiers $Idf(S)$
$(x{:}^{nt}\,[Q])^C$	$(Var \setminus Independ(Q)) \cup (Var \setminus Independ(C)) \cup \{x, x'\}$
$\{P\}^C$	$(Var \setminus Independ(P)) \cup (Var \setminus Independ(C))$
$(\mathbf{var}\, v\ : T \bullet S_1)^C$	$Idf(S_1^{v \in T \wedge C}) \cup \{v, v'\}$
$(\sqcap_{i \in \mathcal{A}} S_i)^C$	$\cup_{i \in \mathcal{A}} Idf(S_i^C)$
$(S_1^{C_1})^{C_2}$	$Idf(S_1^{C_1 \wedge C_2})$
$(S_1\ ;\ S_2)^C$	$Idf(S_1^C) \cup Idf(S_2^C)$

$$\{\mu \in \mathcal{V} \to Val \,|\, B(\sigma \oplus \mu) \bullet \theta(\sigma \oplus \mu)\}$$

in $\mathbb{R} \cup \{\infty, -\infty\}$. *In addition, for a set* Ω *of functions of the same type as* θ *above we denote by* $(\sup_{\theta \in \Omega} \theta)(\sigma)$ *the supremum of the set* $\{\theta \in \Omega \,|\, true \bullet \theta(\sigma)\}$.

B Basic Lemmas and Theorems

Lemma B.1 *Let* x *and* y *be disjoint sets of variables and let* Q *be a predicate that can only have free primed variables in* x'. *Then, the liberal refinement equivalence* $x{:}^{nt}\,[Q] \sqsupseteq_{wlp} x, y{:}^{nt}\,[Q \wedge y = y']$ *holds.*

Proof. Let $R \in SPred$. Then the following holds.

$$\begin{aligned} wlp(x{:}^{nt}\,[Q], R) \equiv\ & \forall x'(Q \Rightarrow R[x'/x]) \\ & \text{“Since } y' \text{ does not occur free in } Q \text{ and } R\text{”} \\ & (\forall x', y'((Q \wedge y = y') \Rightarrow R[x', y'/x, y]) \\ \equiv\ & wlp(x, y{:}^{nt}\,[Q \wedge y = y'], R) \end{aligned}$$

□

Lemma B.2 *Let* v *and* x *denote disjoint sets of variables, then*

$$(\mathbf{var}\, v\ : T \bullet x, v{:}^{nt}\,[Q])^C \sqsupseteq_{wlp} x{:}^{nt}\,[C \wedge C[x'/x] \wedge \exists v, v'(v, v' \in T \wedge Q)].$$

Proof. Let $R \in SPred$ and let $w \notin Idf((\mathbf{var}\, v\ : T \bullet x, v{:}^{nt}\,[Q])^C)$ and $w \in Independ(R)$. Then the following equivalences hold.

$$
\begin{aligned}
& wlp((\mathbf{var}\, v\ : T \bullet x, v{:}^{nt}\,[Q])^{C}, R) \\
\equiv\ & (\forall v(wlp(x, v{:}^{nt}\,[Q]^{v\in T\wedge C[w/v]}, R[w/v])))[v/w] \\
\equiv\ & (\forall v(\forall x', v'((Q \wedge v, v' \in T \wedge C[w/v] \wedge C[w/v][x'/x]) \Rightarrow \\
& \quad R[w/v][x'/x])))[v/w] \\
\equiv\ & (\forall x', v, v'((Q \wedge v, v' \in T \wedge C[w/v] \wedge C[w/v][x'/x]) \Rightarrow \\
& \quad R[w/v][x'/x]))[v/w] \\
\equiv\ & (\forall x'((C[w/v] \wedge C[w/v][x'/x] \wedge \exists v, v'(Q \wedge v, v' \in T)) \Rightarrow \\
& \quad R[w/v][x'/x]))[v/w] \\
\equiv\ & \forall x'((C \wedge C[x'/x] \wedge \exists v, v'(Q \wedge v, v' \in T)) \Rightarrow R[x'/x]) \\
\equiv\ & wlp(x{:}^{nt}\,[C \wedge C[x'/x] \wedge \exists v, v'(v, v' \in T \wedge Q)], R)
\end{aligned}
$$

□

Lemma B.3 $(x{:}^{nt}\,[Q]\ ;\ x{:}^{nt}\,[W]) \sqsupseteq_{wlp} x{:}^{nt}\,[\exists v(Q[v/x'] \wedge W[v/x])]$.

Proof. Let $R \in \mathit{SPred}$. Then the following holds.

$$
\begin{aligned}
wlp((x{:}^{nt}\,[Q]\ ;\ x{:}^{nt}\,[W]), R) \equiv\ & wlp(x{:}^{nt}\,[Q], wlp(x{:}^{nt}\,[W], R)) \\
\equiv\ & \forall x'(Q \Rightarrow (\forall x'(W \Rightarrow R[x'/x]))[x'/x]) \\
\equiv\ & \text{“Let } v \notin \mathit{Idf}(x{:}^{nt}\,[Q]) \cup \mathit{Idf}(x{:}^{nt}\,[W]) \\
& \qquad \text{and } v \in \mathit{Independ}(R)\text{”} \\
& \quad \forall v(Q[v/x'] \Rightarrow (\forall x'(W \Rightarrow R[x'/x]))[v/x]) \\
\equiv\ & \forall v(Q[v/x'] \Rightarrow (\forall x'(W[v/x] \Rightarrow R[x'/x]))) \\
\equiv\ & \forall x', v((Q[v/x'] \wedge W[v/x]) \Rightarrow R[x'/x]) \\
\equiv\ & \forall x'(\exists v(Q[v/x'] \wedge W[v/x]) \Rightarrow R[x'/x]) \\
\equiv\ & wlp(x{:}^{nt}\,[\exists v(Q[v/x'] \wedge W[v/x])], R)
\end{aligned}
$$

□

Lemma B.4 $(\sqcap_{i\in\mathcal{A}} x{:}^{nt}\,[Q_i]) \sqsupseteq_{wlp} x{:}^{nt}\,[\vee_{i\in\mathcal{A}} Q_i]$.

Proof. Let $R \in \mathit{SPred}$. Then the following holds.

$$
\begin{aligned}
wlp(x{:}^{nt}\,[\vee_{i\in\mathcal{A}} Q_i], R) \equiv\ & \forall x'((\vee_{j\in\mathcal{A}} Q_i) \Rightarrow R[x'/x])) \\
\equiv\ & \forall x'(\wedge_{i\in\mathcal{A}}(Q_i \Rightarrow R[x'/x])) \\
\equiv\ & \wedge_{i\in\mathcal{A}}(\forall x'(Q_i \Rightarrow R[x'/x])) \\
\equiv\ & wlp((\sqcap_{i\in\mathcal{A}} x{:}^{nt}\,[Q_i]), R)
\end{aligned}
$$

□

Theorem B.5 (Command reduction) *Let S be a (timed) command, then there is a (timed) specification statement S' such that $S \sqsupseteq_{wlp} S'$ holds.*

Proof. This follows from the definition of (timed) commands and Lemmas B.1, B.2, B.3 and B.4. □

C Efficient execution-time calculation

For a set of variables $\mathcal{V} = \{v_1, ..., v_n\}$, $B \in \mathit{Pred}$, and $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$, we abbreviate $\exists v_1, ..., v_n\, B$ by $\exists \mathcal{V}\, B$, and $\{v_1, ..., v_n \mid B \bullet \theta\}$ by $\{\mathcal{V} \mid B \bullet \theta\}$.

Theorem C.1 *Let $\mathcal{V} \subseteq \mathit{Var}$ and $\mathcal{W} \subseteq \mathit{Var}$ be disjoint, $B \in \mathit{Pred}$ and $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$. If $\mathcal{W} \subseteq \mathit{Independ}(\theta)$ then the following equality holds.*

$$\sup\{\mathcal{V} \cup \mathcal{W} \mid B \bullet \theta\} = \sup\{\mathcal{V} \mid (\exists \mathcal{W}\, B) \bullet \theta\}$$

Proof. Let $\sigma \in \Sigma$. Then the following equalities can be easily checked.

$$\begin{aligned}(\sup\{\mathcal{V} \cup \mathcal{W} \mid B \bullet \theta\})(\sigma) &= \mathrm{supremum}\{\mu \in \mathcal{V} \cup \mathcal{W} \to \mathit{Val} \mid \\ &\qquad B(\sigma \oplus \mu) = \mathit{true} \bullet \theta(\sigma \oplus \mu)\} \\ &= \text{“Since } \mathcal{W} \subseteq \mathit{Independ}(\theta)\text{”} \\ &\quad \mathrm{supremum}\{\mu' \in \mathcal{V} \to \mathit{Val} \mid \\ &\qquad (\exists \mathcal{W}\, B)(\sigma \oplus \mu') = \mathit{true} \bullet \theta(\sigma \oplus \mu')\} \\ &= (\sup\{\mathcal{V} \mid (\exists \mathcal{W}\, B) \bullet \theta\})(\sigma)\end{aligned}$$

□

Theorem C.2 *Let $\mathcal{V} \subseteq \mathit{Var}$, $A, B \in \mathit{Pred}$ and $\theta \in \Sigma \to (\mathbb{R} \cup \{\infty, -\infty\})$. If $\mathcal{V} \subseteq \mathit{Independ}(B)$ then the following entailment relation holds.*

$$B \Rrightarrow (\sup\{\mathcal{V} \mid A \wedge B \bullet \theta\} = \sup\{\mathcal{V} \mid A \bullet \theta\})$$

Proof. Let $\sigma \in \Sigma$ and assume $B(\sigma)$.

$$\begin{aligned}(\sup\{\mathcal{V} \mid A \wedge B \bullet \theta\})(\sigma) &= \mathrm{supremum}\{\mu \in \mathcal{V} \to \mathit{Val} \mid A(\sigma \oplus \mu) \wedge \\ &\qquad B(\sigma \oplus \mu) \bullet \theta(\sigma \oplus \mu)\} \\ &= \text{“Since } B(\sigma) \text{ holds and } \mathcal{V} \subseteq \mathit{Independ}(B)\text{”} \\ &\quad \mathrm{supremum}\{\mu \in \mathcal{V} \to \mathit{Val} \mid A(\sigma \oplus \mu) \wedge \mathit{true} \bullet \\ &\qquad \theta(\sigma \oplus \mu)\} \\ &= (\sup\{\mathcal{V} \mid A \bullet \theta\})(\sigma)\end{aligned}$$

□

References

[1] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Springer-Verlag, 1998.

[2] R.-J. Back, J. von Wright, Reasoning algebraically about loops, Acta Informatica 36 (4) (1999) 295–334.

[3] J. Barnes, High Integrity Ada: The Spark Approach, Addison-Wesley, 1997.

[4] J.-F. Bergeretti, B. A. Carré, Information-flow and data-flow analysis of while-programs, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 37–61.

[5] A. Burns, A. Wellings, Real-Time Systems and Programming Languages, 2nd Edition, Addison-Wesley, 1997.

[6] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Kluwer, 1997.

[7] B. Carré, Program analysis and verification, in: C. T. Sennett (Ed.), High-Integrity Software, Plenum Press, 1989, Ch. 8, pp. 176–197.

[8] R. Chapman, A. Burns, A. Wellings, Combining static worst-case timing analysis and program proof, Real-Time Systems 11 (1996) 145–171.

[9] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

[10] E. W. Dijkstra, C. S. Scholten, Predicate Calculus and Program Semantics, Springer-Verlag, 1990.

[11] J. Engblom, A. Ermedahl, Modeling complex flows for worst-case execution-time analysis, in: Proceedings of the 21st IEEE Real-Time Systems Symposium, IEEE Computer Society, 2000, pp. 163–174.

[12] A. Ermedahl, J. Gustafsson, Deriving annotations for tight calculation of execution time, in: C. Lengauer, M. Griebel, S. Gorlatch (Eds.), Euro-Par'97: Parallel Processing, Vol. 1300 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 1298–1307.

[13] C. J. Fidge, I. J. Hayes, G. Watson, The deadline command, IEE Proceedings—Software 146 (2) (1999) 104–111.

[14] C. J. Fidge, M. Utting, P. Kearney, I. J. Hayes, Integrating real-time scheduling theory and program refinement, in: M.-C. Gaudel, J. Woodcock (Eds.), FME'96: Industrial Benefit and Advances in Formal Methods, Vol. 1051 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 327–346.

[15] S. Grundon, I. J. Hayes, C. J. Fidge, Timing constraint analysis, in: C. McDonald (Ed.), Computer Science '98: Proc. 21st Australasian Computer Science Conference, Springer-Verlag, 1998, pp. 575–586.

[16] E. L. Gunter, D. Peled, Path exploration tool, in: W. R. Cleaveland (Ed.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS'99), Vol. 1579 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 405–419.

[17] I. J. Hayes, C. J. Fidge, K. Lermer, Semantic characterisation of dead control-flow paths, IEE Proceedings—Software 148 (6) (2001) 175–186.

[18] I. J. Hayes, M. Utting, Coercing real-time refinement: A transmitter, in: D. J. Duke, A. S. Evans (Eds.), BCS-FACS Northern Formal Methods Workshop, 1996, Electronic Workshops in Computing, Springer-Verlag, 1997, `http://www.ewic.org.uk/ewic/`.

[19] I. J. Hayes, M. Utting, A sequential real-time refinement calculus, Acta Informatica 37 (2001) 385–448.

[20] I. J. Hayes, Programs as paths: An approach to timing constraint analysis, in: J. S. Dong, J. Woodcock (Eds.), Formal Methods and Software Engineering: Proceedings 5th International Conference on Formal Engineering Methods, ICFEM 2003, Vol. 2885 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 1–15.

[21] E. C. R. Hehner, Termination is timing, in: J. L. A. van de Snepscheut (Ed.), Mathematics of Program Construction, Vol. 375 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 36–47.

[22] S. Horwitz, T. Reps, The use of program dependence graphs in software engineering, in: Proceedings of the Fourteenth International Conference on Software Engineering (ICSE'92), ACM Press, 1992, pp. 392–411.

[23] Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, C. S. Kim, Worst case timing analysis of RISC processors: R3000/R3010 case study, in: Proc. 16th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1995, pp. 308–319.

[24] K. Lermer, C. J. Fidge, A methodology for compilation of high-integrity real-time programs, in: C. Lengauer, M. Griebel, S. Gorlatch (Eds.), Euro-Par'97: Parallel Processing, Vol. 1300 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 1274–1281.

[25] K. Lermer, C. J. Fidge, A formal model of real-time program compilation, Theoretical Computer Science 282 (1) (2002) 151–190.

[26] K. Lermer, C. J. Fidge, I. J. Hayes, Linear approximation of execution-time constraints, Formal Aspects of Computing 15 (4) (2003) 319–348.

[27] Y.-T. Li, S. Malik, A. Wolfe, Efficient microarchitecture modeling and path analysis for real-time software, in: Proc. 16th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 1995, pp. 298–307.

[28] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, C. S. Kim, An accurate worst case timing analysis for

RISC processors, IEEE Transactions on Software Engineering 21 (7) (1995) 593–604.

[29] C. Morgan, The specification statement, ACM Transactions on Programming Languages and Systems 10 (3) (1988) 403–419.

[30] C. Morgan, Programming from Specifications, 2nd Edition, Prentice-Hall, 1994.

[31] C. Morgan, T. Vickers, Types and invariants in the refinement calculus, Science of Computer Programming 14 (1990) 281–304.

[32] C. Y. Park, Predicting program execution times by analyzing static and dynamic program paths, Real-Time Systems 5 (1993) 31–62.

[33] P. Puschner, C. Koza, Calculating the maximum execution time of real-time programs, Journal of Real-Time Systems 1 (2) (1989) 159–176.

[34] P. P. Puschner, A. V. Schedl, Computing maximum task execution times: A graph-based approach, Real-Time Systems 13 (1) (1997) 67–91.

[35] A. C. Shaw, Reasoning about time in higher-level language software, IEEE Transactions on Software Engineering 15 (7) (1989) 875–889.

[36] H. Theiling, C. Ferdinand, R. Wilhelm, Fast and precise WCET prediction by separated cache and path analyses, The International Journal of Time-Critical Computing Systems 18 (2/3) (2000) 157–179.