Acta Informatica 37, 385–448 (2001)

A sequential real-time refinement calculus

[1] Department of Computer Science and Electrical Engineering, The University of Queensland, Brisbane, 4072, Australia (e-mail: ianh@csee.uq.edu.au)
[2] Department of Computer Science, School of Computing and Mathematical Sciences, The University of Waikato, Private Bag 3105, Hamilton, New Zealand (e-mail: marku@cs.waikato.ac.nz)

Received: 27 September 1997 / 13 June 2000

Abstract. We present a comprehensive refinement calculus for the development of sequential, real-time programs from real-time specifications. A specification may include not only execution time limits, but also requirements on the behaviour of outputs over the duration of the execution of the program.

The approach allows refinement steps that separate timing constraints and functional requirements. New rules are provided for handling timing constraints, but the refinement of components implementing functional requirements is essentially the same as in the standard refinement calculus.

The product of the refinement process is a program in the target programming language extended with timing deadline directives. The extended language is a machine-independent, real-time programming language. To provide valid machine code for a particular model of machine, the machine code produced by a compiler must be analysed to guarantee that it meets the specified timing deadlines.

Contents

1 Introduction

Our goal is to provide a method for the stepwise refinement of sequential, real-time programs from real-time specifications. We follow the refinement calculus approach [1, 2, 18, 19] of devising a wide-spectrum language that encompasses both real-time programs and real-time specifications, and the spectrum in between. Of course time is the all important additional dimension in our language. The motivation for our work comes from the real-time refinement calculus of Mahony [12], which allows not only the specification of real-time systems, but the refinement of a specification into a set of truly parallel processes. In this paper we address the task of refining an individual process to sequential code, using the foundations developed by Utting and Fidge [22].

The remainder of this section overviews the real-time, wide-spectrum language and introduces the mechanism for dealing with real-time constraints in the target code. Sect. 2 gives the details of the wide-spectrum language and a comprehensive set of refinement laws. Sect. 3 summarises the extended real-time programming language. Sect. 4 addresses the issue of

feasibility in the real-time calculus. Sect. 5 gives an example specification, timing analysis and refinement for a message receiver example.

1.1 Wide-spectrum language overview

A key change from the standard refinement calculus is our treatment of variables; they are modelled as functions from time to their value at that time. This allows a real-time specification command to constrain not only the final value of variables, but also their values over time. We also introduce a number of time-related commands, such as a delay command and a deadline command. The deadline command is novel to our approach and allows timing constraints to be expressed abstractly in an extended programming language. A separate program analysis is required to guarantee that the deadlines are met by the machine code generated for the program by a compiler.

Inputs, outputs and local variables. In a real-time system there are both external inputs, whose value may vary independently of the direct control of the program, and outputs, whose value over time is to be controlled by the program. Of course, by controlling an output, the program may indirectly control a related input. In addition to inputs and outputs, there are also variables local to the program.

In order to capture the time-varying nature of variables, they are modelled as functions from time to their value at that time [12, 14, 15]. For example, a variable representing the level of water in a mine, and a boolean variable to control a mine pump, are modelled as

$$\begin{array}{l} \mathit{level} : \mathit{Time} \to \mathit{Depth} \\ \mathit{pump} : \mathit{Time} \to \mathbf{boolean} \end{array} \tag{1}$$

where the types *Time* and *Depth* are modelled by the nonnegative real numbers ($\mathbf{real}_+$). To distinguish between *Time* and *Depth* we make use of dimensions of measurement. *Time* has dimension T (for time), and *Depth* has dimension L (for length).

$$\begin{array}{l} \mathit{Time} \mathrel{\widehat{=}} \mathbf{real}_+ \odot \mathsf{T} \\ \mathit{Depth} \mathrel{\widehat{=}} \mathbf{real}_+ \odot \mathsf{L} \end{array}$$

The operator '$\odot$' takes a magnitude and a dimension and forms a dimensioned type [7].

Although our underlying model treats variables as functions of time, it is convenient to abbreviate the declaration of a variable by just giving the type of the range of the variable; the type of the domain is always *Time*. Additionally, we distinguish between inputs, outputs and local variables

via the keywords **input**, **output** and **var**, respectively. In all cases the variables are implicitly functions of time. For example, instead of (1) we write the following.

input *level* : *Depth*
output *pump* : **boolean**

For many purposes outputs and local variables are treated similarly because their values are under the direct control of the program. We refer to outputs and local variables collectively as *program variables*. Because external inputs vary independently of the direct control of a program, it cannot be assumed that they will remain stable just because the program does not modify them. Hence we treat external inputs quite differently to program variables. For example, the guard of a loop cannot access an external input. That ensures that the evaluation of the guard is independent of the order of access to variables and how long it takes to evaluate the guard. Such restrictions greatly simplify reasoning about programs. Access to external inputs is restricted to a special read command.

Constants do not vary over time. Hence their definition is standard. For example, limits on the water level (in metres) in the mine and the minimum rate of change in the level (in metres of depth per second) may be defined as follows.

$$\textbf{const}\ \mathit{limit} \mathrel{\widehat{=}} 1\,\mathsf{m}\,;\ \mathit{minlevel} \mathrel{\widehat{=}} 0.1\,\mathsf{m}\,;\ \mathit{minpumprate} \mathrel{\widehat{=}} 0.01\,\mathsf{m\,s}^{-1}$$

Assumptions. In the standard refinement calculus, assumptions allow statements about the state of the program at the point at which they occur. In the real-time refinement calculus assumptions are generalised to allow properties about variables over time, as well as at the point at which they appear within the program. For example, the following assumption states that at any time, if the pump has been turned on for one second, then the water level in the mine is decreasing (its rate of change is negative) at a rate of at least *minpumprate*.

$$\star \left\{ \begin{array}{l} \forall\, t : \mathit{Time} \bullet \\ \mathit{pump}(\!|\, [\!|\, t - 1\,\mathsf{s} \ldots t \,]\!] \,|\!) = \{\mathit{true}\} \Rightarrow (\underline{\mathsf{s}}\,\mathit{level})(t) < -\mathit{minpumprate} \end{array} \right\} \quad (2)$$

where $[\!|\, t - 1\,\mathsf{s} \ldots t \,]\!]$ is the closed interval of real numbers from t minus one second to t, inclusive; $\mathit{pump}(\!|\, S \,|\!)$ is the set of all values of $\mathit{pump}(x)$ for $x \in S$; and $\underline{\mathsf{s}}\,\mathit{level}$ is the derivative of *level* with respect to time. In this paper we prefix constructs in the real-time language by a '$\star$' to differentiate them from the corresponding constructs of the standard refinement calculus.

Assumptions allow one to state properties about the behaviour of variables in the environment. That restricts the range of behaviours that a program has to cope with.

Specifications. A real-time specification command, $\star\vec{x}\colon [P,\ R]$, consists of a set of variables, $\vec{x}$, called the *frame*, that may be modified by the command; a predicate, P, giving the *assumptions* the command may make about the environment; and a predicate, R, giving the *effect* the command is to achieve. A command can modify only program variables, and not external inputs. Hence the frame $\vec{x}$ must be completely contained within the program variables. The special variable τ, of type *Time*, is introduced to refer to the *current* time. Within the assumptions, τ refers to the start time of the command, and within the effect, τ_0 and τ refer to the start and finish times, respectively; the assumptions may not refer to τ_0. For example, in the context of the assumption (2) the following is a specification of one activity of a mine pump controller.

$$\star pump\colon \left[level(\tau) > limit,\ \begin{array}{l} level(\tau) \leq limit \wedge \\ (\forall\, t : [\![\tau_0 \mathinner{..} \tau]\!] \bullet level(t) \geq minlevel) \end{array} \right] \quad (3)$$

In this case, no time bounds are specified for the execution time, but examples later in the paper show how this can easily be done.

The predicates used as assumptions and effects may refer to both external and local variables of the program. Such variables are modelled as functions of time. To reference the value of variable v at time t one simply indexes v by t, i.e., $v(t)$. Because many references to variables, especially program variables, within the assumptions refer to values of the variables at the start time of the command, we introduce a convention that an unindexed occurrence of a variable, v, in a context where an indexed occurrence is expected, stands for $v(\tau)$, i.e., the value of v at the start time of the command. For example, the assumption in (3) may be abbreviated to $level > limit$.

A similar convention is used within the effect where an unindexed occurrence of a variable v also stands for $v(\tau)$, where in this case τ is the finish time of the command. For example, the first conjunct in the effect of (3) can be abbreviated to $level \leq limit$. Within the effect there may also be references to the initial values of variables. Hence we introduce the convention that v_0 stands for $v(\tau_0)$, i.e., the value of v at the start time of the command. This convention is consistent with that used by Morgan [18].

As well as constraining the value of variables at termination time, the effect may constrain the value of variables over time. For example, the second conjunct in the effect of (3) requires that the water level be maintained above a minimum level for the entire duration of the command. The assumptions may also include predicates over the values of variables over time, typically stating assumptions about the properties of external inputs.

The effect of a specification command defines the possible range of behaviours required. In achieving that effect only variables in the frame may be modified, and it may be assumed that the environment satisfies the assumptions of the specification command.

1.2 Real-time constraints

The program development process begins with a specification and refines it step-by-step until a program written in the target programming language is reached. Real-time specifications can include timing constraints. Hence it is essential that the refinement process handles timing constraints.

A fundamental problem with developing real-time programs is that the actual timing performance of a program is determined by both the compiler for the language and the particular model of machine on which the generated code executes. No analysis of just the higher-level program can take into account low-level aspects such as register allocation and code optimisation within a compiler, or instruction pipelining and cache memories within processors, which together can affect the timing characteristics of a program by an order of magnitude.

By definition, specification commands achieve their stated effect, including any real-time constraints. However, the executable subset of the wide-spectrum language does not include general specification commands. The objective of our real-time refinement process is to derive program code that satisfies a given specification, including any real-time constraints. There is an obvious gap between a specification command, which by definition meets its real-time requirements, and program code, which may or may not meet the real-time requirements depending on the compiler and target machine.

To tackle this incompatibility we introduce an intermediate programming language that extends the target programming language to allow real-time constraints to be expressed within program code. The program development process is then split into two phases: refining a real-time specification to code in the extended programming language; and performing a timing analysis on the code generated by a compiler to ensure that it meets all the real-time constraints. The extended programming language enjoys the advantage of being both machine and compiler independent, which is one of the most significant advantages claimed for high-level programming languages in general. This allows the formal refinement process to be treated in a machine/compiler independent fashion.

However, there is a catch: there is no guarantee that one can compile a program in the extended language so as to guarantee all the real-time constraints are met. The compiled program must be analysed to determine whether it meets all the timing constraints. If the program passes the timing analysis phase, it is a valid implementation, but if it fails, it should be rejected. Logically the timing analysis phase can be viewed as part of a compiler for the extended real-time programming language. Note that we refer to three different levels of language in this paper:

– the *target programming language*, which is a standard programming language that does not include timing deadlines;

– the *extended programming language* that includes timing deadlines; and
– the *wide-spectrum language*, which includes all languages features, including specification commands.

The extended programming language. To allow timing constraints to be expressed we make a minimal extension to an existing programming language. The main extension is a novel command of the form, $\mathbf{deadline}\, D$, which on 'execution' guarantees to terminate by time D. This command allows timing constraints to be incorporated into programs, but the deadline command can not be directly implemented by a compiler. Consider the following fragment of a program, in which d is a program variable of type time.

$$\star\{\tau \leq d\}; \tag{4}$$
$$\mathbf{delay\ until}\, d; \tag{5}$$
$$\text{"commands which do not update } d\text{"}; \tag{6}$$
$$\mathbf{deadline}\, d + 1\,\mathsf{ms} \tag{7}$$

The assumption (4) states that the code is assumed to begin execution by time d. The delay command (5) guarantees to terminate after time d. The deadline directive (7) adds the constraint that the time at which it completes is before time d plus one millisecond.

To determine whether the above fragment meets its deadline, one needs to analyse the machine code generated for the delay command (5) and the other commands (6). The delay command is guaranteed to terminate after time d, but it may overrun past time d. The maximum time by which it may overrun is referred to as its *lateness*. If the lateness of the delay plus the maximum execution time of the machine code generated for the other commands (6) is guaranteed to be less than or equal to one millisecond, then the deadline is guaranteed to be met for all executions for which the initial assumption (4) holds.

The deadline command can be viewed as a directive to the compiler for the programming language. If the compiler can determine by analysing the machine code generated for (5) and (6) that they will always take less than one millisecond to execute on the target machine, then the compiler will successfully generate machine code for them. In that case the deadline is guaranteed to be met, and hence it can be discarded. On the other hand, if the compiler cannot guarantee that (5) and (6) will always take less than one millisecond, then it cannot guarantee to satisfy (7). In that case the compiler should reject the program because it is unable to guarantee its timing correctness on the given target machine; the compiler can also supply details indicating which deadline cannot be met, and by what margin it is missed. Note that a deadline directive defines a hard deadline that must be

met. It is not acceptable to generate code that tests whether the deadline was reached in time and raises an exception if it was missed. Timing analysis is examined in more detail in Sect. 3 and in the example in Sect. 5.2.

The combination of a delay until command and a deadline directive allows one to add quite detailed timing constraints to a program, and thus provides an effective real-time programming language. Given the extended programming language, our goal is to develop programs from specifications that include timing requirements, using a process of stepwise refinement. For the purposes of this paper we have chosen Dijkstra's guarded command language [3] as our base programming language, because that has been used as the basis for other refinement calculus work. However, the real-time extensions introduced here could equally well be applied to other programming languages, such as Ada. Interestingly, the nondeterministic language constructs found in Dijkstra's language are not problematical with the approach taken here.

1.3 Related work

We concentrate our comparison with related work on approaches that develop real-time programs from abstract specifications. The two approaches we consider are Scholefield's Temporal Agent Model (TAM) [20, 21], and Hooman's assertional specification and verification of real-time programs [9]. Both these approaches also support concurrency, but because our focus in this paper is just sequential programs, we ignore the concurrent constructs in this comparison.

All the methods introduce some form of time variable that allows reference to the start and finish times of commands, and that can be used to specify timing constraints and relationships. The two main features that distinguish our work are the addition of the deadline command, and the use of timed traces for inputs, outputs and local variables.

TAM provides a real-time refinement calculus. If we compare the sequential aspects of TAM with our approach, the main difference is in the treatment of deadlines. In TAM, deadlines are specified as a constraint on the execution time allowed for a command. This restricts deadlines to the command structure of the TAM language. In comparison, we express deadlines via a separate **deadline** command. This allows more flexibility in specifying timing constraints. In addition to being able to specify constraints that match the structure of the language, one can also specify constraints on execution paths that cross the boundaries of language constructs, e.g., a path that begins before an alternation command and ends within one branch of the alternation, or a path from a point within the body of a loop back around to a point within the body of the loop on its next iteration.

A consequence of the TAM approach to deadlines is that it is necessary to specify a constant representing the evaluation time for all guards of an alternation, even though in practice the guard evaluation time is different for different branches of an alternation. In our approach there is no need for such a constant: guard evaluation is just considered to be part of the execution time of each path on which the guard appears. The real constraints are the overall constraint on each path. There is no necessity to have an additional, more restrictive, constraint on the guard evaluation.

Another difference is in the treatment of inputs and outputs. TAM provides shunts for communication between processes and for communication with the environment. Our approach treats inputs and outputs as traces over time. One of the main application areas we see for our work is in the specification and refinement of systems with continuous variables, such as the *level* of water in the mine shaft. In order to be able to give a top-level specification of such systems, we need to use timed traces. Within this paper we define a simple **read** command, that samples an input, but because we use timed traces, more complex input commands, such as analog-to-digital conversion can be handled within the same framework.

Hooman's work [9] on assertional specification and verification extends the use of traditional Hoare logic triples to real-time systems. The real-time interface of the program with the environment can be specified using primitives denoting the timing of observable events. The interpretation of triples has been adapted to require the postcondition to hold for terminating and non-terminating computations.

Each of the atomic commands in Hooman's language has an associated constant representing its execution time, and the compound commands, such as **if-then-else** have constants representing the time to evaluate the guard. For example, Hooman [9, page 126] introduces a constant, T_a, representing the execution time of an assignment command. An obvious problem is that not all assignments take the same amount of time, but further, given a single assignment command, its execution time may vary greatly (due to the effects of pipelines or caches) depending upon the preceding commands on the path. Timing constraints on program components must be broken down into timing constraints on the individual commands. The overall approach is similar to that used in TAM and suffers in the same ways in comparison to the use of a **deadline** command to specify timing constraints on paths. Hooman also provides non-terminating loops, which we have not attempted within this paper.

The seemingly small addition of the **deadline** command in our work has had a significant impact on the whole development method, and importantly, has allowed developments to treat real-time constraints in a more realistic and practical manner than in the other approaches.

- During the refinement process, **deadline** commands can be used to separate out timing constraints, leaving behind a requirement to be met that does not explicitly contain timing constraints. The standard refinement calculus can be used to develop such components.
- The timing constraints are on execution paths through the program and are not necessarily constrained to the phrase structure of the programming language. This allows more realistic timing constraints to be devised.
- A timing constraint is on a whole execution path rather than each command in the path, and hence is less restrictive in terms of allowable implementations.

A **deadline** command could be added to the standard refinement calculus extended with a time variable, τ, but not using timed traces for variables. Such an extension would provide the benefits listed above. However, it would not allow the top-level specification of systems involving continuous inputs within the same framework.

Sect. 2 contains a comprehensive presentation of definitions of real-time programming constructs and associated refinement laws. A reader wishing to get an idea of how the calculus is used may prefer to skip directly to Sections 3 and 5 and refer back to Sect. 2 as needed.

2 Semantics and laws for language constructs

Because we treat variables as functions from time to their value at that time, we cannot directly use the standard refinement calculus [1, 2, 18, 19] for real-time refinement. However, Utting and Fidge [22, 23] introduced a way to encode the real-time refinement calculus as a standard refinement calculus that changes only time. In the encoding, variables in the real-time calculus are explicitly declared as functions of time in the standard calculus; a special standard variable, τ, (not itself a function of time) is introduced to stand for time; and all the constructs in the programming language are replaced with their real-time equivalents. The encoding should not be seen as a way of implementing the real-time language, but rather as a device for showing the relationship between the standard and real-time calculi. It also allows the reuse of the existing standard theory to build the new real-time theory.

In the encoding, program variables are created with their value initially unconstrained over all time, except that the value is of the declared type. The 'execution' of commands in the language may constrain the values of variables. At the specification level, arbitrary constraints are allowed, but at the level of executable code, the primitive commands of the language can only constrain the value of program variables during the time period

over which they execute. Without this constraint it would not be feasible to implement such constructs [23]. (See Sect. 4 for more details.)

2.1 Environment

An environment defines finite and disjoint sets of inputs, outputs and local variables. Identifier names for variables are chosen from the set *Ident*, and $\mathbb{F}\ Ident$ stands for the set of all finite subsets of *Ident*.

$$\begin{array}{|l} \textit{Env} \\ \hline in, out, local : \mathbb{F}\ Ident \\ \hline in \cap out = out \cap local = local \cap in = \varnothing \end{array}$$

Such an environment could be extended to include the type of each variable, but because the typing aspects are not novel, we have chosen to elide them in this paper. It is useful to be able to determine the set of all variables in an environment, $var(\rho)$, and the set of program variables: variables that are either outputs or local variables (but not inputs), $pvar(\rho)$.

$$\begin{array}{|l} var : Env \rightarrow \mathbb{F}\ Ident \\ pvar : Env \rightarrow \mathbb{F}\ Ident \\ \hline var(\rho) = \rho.in \cup \rho.out \cup \rho.local \\ pvar(\rho) = \rho.out \cup \rho.local \end{array}$$

As an abbreviation we write $\hat{\rho}$ for $pvar(\rho)$.

2.2 Predicate transformers and refinement

As for the standard refinement calculus, the semantics are given in terms of predicate transformers, *PTran*, which are monotonic functions from predicates (effects) to predicates (assumptions):

$$\begin{aligned} PTran \mathrel{\widehat=} \{ & PT : Pred \rightarrow Pred \mid \\ & \forall P, Q : Pred \bullet (P \Rrightarrow Q) \Rightarrow (PT(P) \Rrightarrow PT(Q))\} \end{aligned}$$

where '$\Rrightarrow$' is the universal quantification over all states of the corresponding implication, as used by Morgan [18]. However, we need to take into account the environment of the command being defined. Hence we define a meaning function, $\mathcal{M}$, that given an environment and a command, defines a predicate transformer,

$$\mathcal{M} : Env \rightarrow (Command \nrightarrow PTran)$$

where *Command* is the set of syntactic commands. We write the environment parameter to $\mathcal{M}$ as a subscript. Refinement is defined with respect to an environment, ρ,

$$C \sqsubseteq_\rho C' \mathrel{\widehat{=}} \mathcal{M}_\rho(C) \sqsubseteq \mathcal{M}_\rho(C') \tag{8}$$

where '$\sqsubseteq$' is standard refinement:

$$S \sqsubseteq S' \mathrel{\widehat{=}} (\forall G : Pred \bullet S(G) \Rrightarrow S'(G)) \tag{9}$$

Refinement equivalence is defined by

$$C \sqsupseteq_\rho C' \mathrel{\widehat{=}} (C \sqsubseteq_\rho C' \wedge C' \sqsubseteq_\rho C) \tag{10}$$

2.3 Unindexed variables in predicates and expressions

The assumption and effect predicates in a specification may include unindexed references to variables using the conventions outlined in Sect. 1.1. In an effect, unindexed variables of the form v stand for $v(\tau)$, and unindexed variables of the form v_0 stand for $v(\tau_0)$. We introduce the notation $R@(\tau_0, \tau)$ to stand for the predicate R with every unindexed occurrence of a variable, v, replaced by $v(\tau)$ and every unindexed occurrence of v_0 is replaced by $v(\tau_0)$. In fact, we generalise this notation so that τ_0 and τ are just parameters. We make use of the same conventions for expressions.

Definition 1 (at times) *Given a predicate or expression, R, $R @ (x, y)$ is defined to be R with all unindexed occurrences of the form v_0 replaced by $v(x)$ and all unindexed occurrences of v replaced by $v(y)$, and as well, all occurrences of τ_0 and τ in R are replaced by x and y, respectively.*

When the '@' operator is used in the form $R@(\tau_0, \tau)$, the replacements of τ_0 and τ have no effect. Note that R may contain explicit indexed references to variables at times other than τ; these are not affected by the '@' operator. The operator '@' has a lower precedence than all the normal logical operators, but a higher precedence than '$\equiv$' and '$\Rrightarrow$'. For predicates, such as assumptions, that do not contain any zero-subscripted variables, we use the notation $P@x$.

Definition 2 (at time) *Given a predicate or expression, P, $P @ x$ is defined to be P with all unindexed occurrences of a variable v replaced by $v(x)$, and any occurrences of τ replaced by x.*

If there are no occurrences of τ_0 or zero-subscripted variables in P then $P @ (\tau_0, \tau) \equiv P @ \tau$. The operator '@' distributes over logical operators. For example, the following identities hold.

$$(P \wedge Q @ x) \equiv (P @ x) \wedge (Q @ x)$$
$$(P \vee Q @ x) \equiv (P @ x) \vee (Q @ x)$$
$$(\neg P @ x) \equiv \neg (P @ x)$$

Aside: One interesting approach to proving properties of predicates using the conventions outlined above, is to prove entailment between predicates treating unindexed variables as variables of their range type. This allows simpler proofs of properties that do not involve explicit time indices. For example, from

$$x = y \Rrightarrow x - y = 0$$

we may deduce that

$$x = y \mathbin{@} \tau \Rrightarrow x - y = 0 \mathbin{@} \tau$$

also holds. We note that this allows one to treat proof obligations that do not involve time indices in a manner very close to the proof obligations in Morgan's calculus [18]. This lifting property is not essential to the calculus presented in this paper; it is treated in more detail in [6].

In the remainder of this section we introduce the constructs in the wide-spectrum language along with related refinement laws. On the first reading the proofs of laws may be skipped over unless the reader is uncertain about a given law.

2.4 Specifications

A real-time specification command can be defined in terms of a standard specification command using the conventions introduced by Utting and Fidge [22]. The equivalent standard specification command allows time to increase, and insists that all program variables that are not in the frame remain stable (unchanged) for the duration of the command.

Definition 3 (stable) *Given a variable, v, and a set of times, S,*

$$\mathit{stable}(v, S) \mathrel{\widehat{=}} (\forall\, t, u : S \bullet v(t) = v(u))$$

Within this paper we allow a set of variables to be used as the first parameter to *stable*, with the meaning that every variable in the set is stable over the set of times. The notation, $\vec{x}$, stands for a vector of variables; we allow a vector to be used where a set is expected, with the meaning that it stands for the set of variables contained in the vector.

The assumptions of a specification command determine the range of possible values of variables over time, as well as the start time of the command. The effect further constrains the values of variables over time, as well as constraining the finish time of the command.

Definition 4 (specification) *A specification command,* $\star\vec{x}\colon [P,\ R]$*, is well formed in an environment,* ρ*, provided*

- *the frame, $\vec{x}$, is a vector of program variables, $\vec{x} \subseteq \hat{\rho}$ (remember $\hat{\rho}$ is an abbreviation for the program variables of ρ),*
- *P is a predicate involving the variables in the environment plus τ, and*
- *R is a predicate involving variables in the environment plus τ_0, τ and zero-subscripted versions of variables in the environment.*

The meaning of a well-formed specification command is given by the following

$$\mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R]) \mathrel{\widehat{=}} \\ \tau\colon [P \mathbin{@} \tau,\ R \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\!| \tau_0 \ldots \tau |\!])]$$

where $\hat{\rho} \setminus \vec{x}$ stands for the set $\hat{\rho}$ with any elements in the frame $\vec{x}$ removed.

Note that τ, unlike other variables, is not itself a function of time. The frame of the real-time command, $\vec{x}$, does not appear in the frame of the equivalent standard command. Instead, those program variables that are not in the frame are constrained to be stable for its duration, and the program variables in the frame are only constrained by the effect of the specification, *R*. The use of traces, and their exclusion from the frame, leads to interesting *feasibility* issues, which are discussed in Sect. 4. In the assumption and effect of a specification command it is permissible to include both explicitly indexed references and unindexed references to the same variable.

The definitions of many of the remaining commands can be given in terms of equivalent specification commands. The environment used for the equivalent command is usually the same as that for the command being defined. Hence, we introduce the abbreviation

$$C \mathrel{\widehat{=}}_\rho C'$$

to stand for

$$\mathcal{M}_\rho\,(C) \mathrel{\widehat{=}} \mathcal{M}_\rho\,(C')$$

We use the abbreviation that the default assumption in a specification command is *true*.

Definition 5 (default assumption) *Given an environment ρ, provided $\vec{x}$ is contained in the program variables, $\hat{\rho}$, and R is a predicate as defined in Def. 4 (specification), then*

$$\star\vec{x}\colon [R] \mathrel{\widehat{=}}_\rho \star\vec{x}\colon [\mathit{true},\ R]$$

For the remainder of the paper, we assume that the frame and the predicates in the assumptions and effects of specification commands satisfy the well-formedness constraints outlined in Def. 4 (specification). In addition, we

make use of Def. 5 (default-assumption) without explicit reference to its definition.

The refinement rules for weakening an assumption and strengthening an effect carry over to the real-time refinement calculus.

Law 6 (weaken assumption) *Given an environment of variables, ρ, provided* $P \mathbin{@} \tau \Rrightarrow P' \mathbin{@} \tau$,

$$\star\vec{x}\colon [P,\ \ R] \sqsubseteq_\rho \star\vec{x}\colon [P',\ \ R]\ .$$

When applying this law we can use the fact that from $P \Rrightarrow P'$ one can deduce that $P \mathbin{@} \tau \Rrightarrow P' \mathbin{@} \tau$. That gives a special case of the law for dealing with properties that are not time dependent. This special case is identical to the weaken-precondition law of Morgan's calculus [18].

Proof. The law follows from the equivalent standard refinement calculus law.

$$\begin{array}{ll}
& \mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ \ R]) \\
\sqsupseteq & \text{Def. 4 (specification)} \\
& \tau\colon [P \mathbin{@} \tau,\ \ R \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket)] \\
\sqsubseteq & \text{Standard weaken precondition} \\
& \tau\colon [P' \mathbin{@} \tau,\ \ R \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket)] \\
\sqsupseteq & \text{Def. 4 (specification)} \\
& \mathcal{M}_\rho\,(\star\vec{x}\colon [P',\ \ R])\quad \square
\end{array}$$

Law 7 (strengthen effect) *Given an environment, ρ, provided*

$$\begin{array}{l}
\quad (P \mathbin{@} \tau_0) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \wedge (R' \mathbin{@} (\tau_0, \tau)) \\
\Rrightarrow R \mathbin{@} (\tau_0, \tau)
\end{array}$$

then

$$\star\vec{x}\colon [P,\ \ R] \sqsubseteq_\rho \star\vec{x}\colon [P,\ \ R']\ .$$

This law is often used to replace an effect R by another R' that is equivalent under the assumptions P. In that case the refinement is an equivalence. In the case where the properties are not time dependent, a special case of the proviso is the following

$$P_0 \wedge R' \Rrightarrow R$$

where P_0 stands for the predicate P with all occurrences of τ replaced by τ_0, and all unindexed occurrences of every variable, v, that is in the frame or is an external input, replaced by v_0. This corresponds to the equivalent law in Morgan's calculus [18].

Proof. The law follows from the equivalent standard refinement calculus law.

$$
\begin{array}{ll}
 & \mathcal{M}_\rho\,(\star\vec{x}{:}\,[P,\ \ R]) \\
\sqsupseteq & \text{Def. 4 (specification)} \\
 & \tau{:}\,[P\ @\ \tau,\ \ R\ @\ (\tau_0,\tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])] \\
\sqsubseteq & \text{From the proviso using standard strengthen postcondition} \\
 & \tau{:}\,[P\ @\ \tau,\ \ R'\ @\ (\tau_0,\tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])] \\
\sqsupseteq & \text{Def. 4 (specification)} \\
 & \mathcal{M}_\rho\,(\star\vec{x}{:}\,[P,\ \ R'])\ \ \Box
\end{array}
$$

A requirement that a variable in the frame remains stable for the duration of a command can be achieved by removing the variable from the frame.

Law 8 (contract frame) *Given an environment, ρ, and disjoint vectors of program variables, $\vec{x}$ and $\vec{v}$,*

$$\star\vec{x},\vec{v}{:}\,[P,\ \ R \wedge \mathit{stable}(\vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])] \sqsupseteq_\rho \star\vec{v}{:}\,[P,\ \ R]$$

Proof.

$$
\begin{array}{ll}
 & \mathcal{M}_\rho\,(\star\vec{x},\vec{v}{:}\,[P,\ \ R \wedge \mathit{stable}(\vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])]) \\
\sqsupseteq & \text{Def. 4 (specification)} \\
 & \tau{:}\left[P\ @\ \tau,\ \begin{array}{l} R\ @\ (\tau_0,\tau) \wedge \mathit{stable}(\vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \wedge \\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus (\vec{x},\vec{v}), [\![\tau_0 \mathinner{..} \tau]\!]) \end{array}\right] \\
\sqsupseteq & \text{As } \vec{x} \text{ is stable and } \vec{x} \text{ and } \vec{v} \text{ are disjoint} \\
 & \tau{:}\,[P\ @\ \tau,\ \ R\ @\ (\tau_0,\tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{v}, [\![\tau_0 \mathinner{..} \tau]\!])] \\
\sqsupseteq & \text{Def. 4 (specification)} \\
 & \mathcal{M}_\rho\,(\star\vec{v}{:}\,[P,\ \ R])\ \ \Box
\end{array}
$$

A variable can always be removed from the frame.

Law 9 (remove from frame) *Given an environment, ρ, and disjoint vectors of program variables, $\vec{x}$ and $\vec{v}$,*

$$\star\vec{x},\vec{v}{:}\,[P,\ \ R] \sqsubseteq_\rho \star\vec{v}{:}\,[P,\ \ R]$$

Proof.

$$
\begin{array}{ll}
 & \star\vec{x},\vec{v}{:}\,[P,\ \ R] \\
\sqsubseteq_\rho & \text{Law 7 (strengthen-effect)} \\
 & \star\vec{x},\vec{v}{:}\,[P,\ \ R \wedge \mathit{stable}(\vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])] \\
\sqsupseteq_\rho & \text{Law 8 (contract-frame)} \\
 & \star\vec{v}{:}\,[P,\ \ R]\ \ \Box
\end{array}
$$

2.5 Assumptions

Assumptions may state properties of the variables at the point at which they occur. Hence in an assumption, an unindexed reference to a variable, v, is interpreted as $v(\tau)$. Assumptions can also state properties of the value of variables at other times by using explicit indices. Assumptions take no time, and hence there is no need for τ_0 or zero-subscripted variables, within assumptions.

Definition 10 (assumption) *An assumption, $\star\{P\}$, is well-formed provided P does not refer to τ_0 or zero-subscripted variables. Given an environment, ρ, an assumption is equivalent to a specification command that takes no time and has the assumption as its precondition.*

$$\star\{P\} \mathrel{\widehat{=}}_\rho \star[P,\ \tau_0 = \tau]$$

From this definition we can deduce

$\mathcal{M}_\rho(\star\{P\})$
$\sqsubseteq$ Def. 10 (assumption)
$\mathcal{M}_\rho(\star[P,\ \tau_0 = \tau])$
$\sqsubseteq$ Def. 4 (specification)
τ: $[P\ @\ \tau,\ \tau_0 = \tau]$
$\sqsubseteq$ Standard contract frame
$[P\ @\ \tau,\ \mathit{true}]$
$\sqsubseteq$ Standard refinement calculus
$\{P\ @\ \tau\}$

The last line above represents a standard refinement calculus assumption, while the first line is an assumption in the real-time refinement calculus.

For the remainder of this paper we assume that all assumptions are well-formed, i.e., they do not refer to τ_0 or to zero-subscripted variables.

Law 11 (weaken assumption command) *Given an environment, ρ, provided $P\ @\ \tau \Rrightarrow P'\ @\ \tau$,*

$$\star\{P\} \sqsubseteq_\rho \star\{P'\}$$

Proof. The law follows directly from Def. 10 (assumption) and Law 6 (weaken-assumption). □

2.6 Sequential composition

The definition of sequential composition carries over from the standard refinement calculus, provided we assume that the start time of the second component is identical to the finish time of the first component.

Definition 12 (sequential composition) *Given an environment, ρ,*

$$\mathcal{M}_\rho\,(C; D) \mathrel{\widehat{=}} (\mathcal{M}_\rho\,(C)) \circ (\mathcal{M}_\rho\,(D))$$

where '$\circ$' is used here to represent standard sequential composition, i.e., functional composition of the predicate transformers.

Law 13 (separate assumption) *Given an environment, ρ,*

$$\star\vec{x}\colon [U \wedge P,\ R] \sqsupseteq_\rho \star\{U\}\,;\ \star\vec{x}\colon [P,\ R]$$

Proof.

$\mathcal{M}_\rho\,(\star\vec{x}\colon [U \wedge P,\ R])$
$\sqsupseteq$ Def. 4 (specification)
$\tau\colon [U @ \tau \wedge P @ \tau,\ R @ (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])]$
$\sqsupseteq$ Standard refinement calculus
$\{U @ \tau\} \circ \tau\colon \left[P @ \tau,\ \begin{array}{l} R @ (\tau_0, \tau) \\ \wedge\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \end{array}\right]$
$\sqsupseteq$ Def. 10 (assumption); Def. 4 (specification)
$\mathcal{M}_\rho\,(\star\{U\}) \circ \mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R])$
$\sqsupseteq$ Def. 12 (sequential-composition)
$\mathcal{M}_\rho\,(\star\{U\}\,;\ \star\vec{x}\colon [P,\ R])$ $\square$

Law 14 (establish assumption) *Given an environment, ρ, and a predicate V, such that neither τ_0 nor zero-subscripted variables occur in V, provided $R @ (\tau_0, \tau) \Rrightarrow V @ \tau$ then*

$$\star\vec{x}\colon [P,\ R] \sqsupseteq_\rho \star\vec{x}\colon [P,\ R]\,;\ \star\{V\}$$

For time independent properties, we can use the proviso $R \Rrightarrow V$ as a special case of this law, because from that we can deduce that $R @ (\tau_0, \tau) \Rrightarrow V @ \tau$.

Proof.

$\mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R])$
$\sqsupseteq$ Def. 4 (specification)
$\tau\colon [P @ \tau,\ R @ (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!])]$
$\sqsupseteq$ Standard refinement calculus
$\tau\colon \left[P @ \tau,\ \begin{array}{l} R @ (\tau_0, \tau) \\ \wedge\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \end{array}\right] \circ \{V @ \tau\}$
$\sqsupseteq$ Def. 4 (specification); Def. 10 (assumption)
$\mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R]) \circ \mathcal{M}_\rho\,(\star\{V\})$
$\sqsupseteq$ Def. 12 (sequential-composition)
$\mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R]\,;\ \star\{V\})$ $\square$

2.7 Doing nothing

The identity of sequential composition is the command that changes no variables and takes no time.

Definition 15 (skip) *For any environment, ρ,*

$$\star\mathbf{skip} \mathrel{\widehat{=}}_\rho \star\left[\tau_0 = \tau\right]$$

In terms of a standard specification command, $\star\mathbf{skip}$ has an empty frame, and hence is equivalent to $\mathbf{skip}$ in the standard refinement calculus.

$$\mathcal{M}_\rho\,(\star\mathbf{skip}) \sqsupseteq \tau\colon \left[\tau_0 = \tau\right] \sqsupseteq \left[\mathit{true}\right] \sqsupseteq \mathbf{skip}$$

Law 16 (skip identity) *For any environment, ρ,*

$$\star\mathbf{skip};\, C \sqsupseteq_\rho C \sqsupseteq_\rho C;\ \star\,\mathbf{skip}$$

Proof.

$\mathcal{M}_\rho\,(\star\mathbf{skip};\, C)$
$\sqsupseteq$ Def. 12 (sequential-composition)
$\mathcal{M}_\rho\,(\star\mathbf{skip}) \circ \mathcal{M}_\rho\,(C)$
$\sqsupseteq$ Def. 15 (skip)
$\mathbf{skip} \circ \mathcal{M}_\rho\,(C)$
$\sqsupseteq$ Standard refinement calculus
$\mathcal{M}_\rho\,(C)$
$\sqsupseteq$ Standard refinement calculus
$\mathcal{M}_\rho\,(C) \circ \mathbf{skip}$
$\sqsupseteq$ Def. 15 (skip)
$\mathcal{M}_\rho\,(C) \circ \mathcal{M}_\rho\,(\star\mathbf{skip})$
$\sqsupseteq$ Def. 12 (sequential-composition)
$\mathcal{M}_\rho\,(C;\ \star\,\mathbf{skip})$ $\Box$

An assumption may be removed by refining it to $\star\mathbf{skip}$ and then using Law 16 (skip-identity).

Law 17 (remove assumption) *For any environment, ρ,*

$$\star\left\{P\right\} \sqsubseteq_\rho \star\mathbf{skip}$$

Proof.

$\star\left\{P\right\}$
$\sqsupseteq_\rho$ Def. 10 (assumption)
$\star\left[P,\ \ \tau_0 = \tau\right]$
$\sqsubseteq_\rho$ Law 6 (weaken-assumption)
$\star\left[\tau_0 = \tau\right]$
$\sqsupseteq_\rho$ Def. 15 (skip)
$\star\mathbf{skip}$ $\Box$

A command that does nothing, but may take time, is **idle**.

Definition 18 (idle) *For any environment, ρ,*

$$\mathbf{idle} \mathrel{\widehat{=}}_\rho \star [\mathit{true}]$$

In terms of a standard specification command we have

$$\mathcal{M}_\rho(\mathbf{idle}) \sqsubseteq \tau\colon [\tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho}, \lbrack\!\lbrack \tau_0 \ldots \tau \rbrack\!\rbrack)]$$

Hence $\mathbf{idle} \sqsubseteq_\rho \star [\tau_0 \leq \tau]$.

Law 19 (skip idle) *For any environment, ρ,*

$$\mathbf{idle} \sqsubseteq_\rho \star\mathbf{skip}$$

Proof.

$$\begin{array}{ll} & \mathbf{idle} \\ \sqsubseteq_\rho & \text{Def. 18 (idle)} \\ & \star [\mathit{true}] \\ \sqsubseteq_\rho & \text{Law 7 (strengthen-effect)} \\ & \star [\tau_0 = \tau] \\ \sqsubseteq_\rho & \text{Def. 15 (skip)} \\ & \star\mathbf{skip} \quad \Box \end{array}$$

2.8 Invariant properties

Many properties are invariant over the execution of particular commands. Two interesting cases are predicates that are invariant over the execution of an **idle** command, and predicates that are invariant over the execution of a specification command with frame $\vec{x}$. We refer to these as *idle-invariant* and *frame-invariant* predicates, respectively. In fact, idle-invariant predicates are the special case of frame-invariant predicates when the frame is empty.

Definition 20 (frame invariant) *Given an environment, ρ, a predicate, P, that contains no references to τ_0 or zero-subscripted variables, is* frame-invariant *with respect to frame $\vec{x}$, if and only if*

$$(P \mathbin{@} \tau_0) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \lbrack\!\lbrack \tau_0 \ldots \tau \rbrack\!\rbrack) \Rrightarrow P \mathbin{@} \tau$$

Definition 21 (idle invariant) *Given an environment, ρ, a predicate, P, is* idle-invariant *if and only if P is frame-invariant with respect to the empty frame.*

For both types of properties there are syntactic checks that are sufficient (but not necessary) to ensure that the properties are invariant. A property, P, is idle-invariant if it is invariant over the execution of an **idle** command. During the execution of an **idle** command, the program variables (outputs and local variables) are stable. Hence, if the property only refers to program variables, it is invariant. If the property refers to τ, the property may be invalidated because the **idle** command may take time. For example, the property $\tau \leq D$ is not invariant if the **idle** command executes until after time D. For this reason our syntactic check disallows references to τ. Similarly, references to external inputs are not guaranteed stable over time. Hence we exclude unindexed references to external inputs as well. For frame-invariant properties we add the restriction that there are no unindexed references to variables in the frame.

Law 22 (frame-invariant property) *Given an environment, ρ, a frame, $\vec{x}$, such that $\vec{x} \subseteq \hat{\rho}$, and a predicate, P, then P is frame-invariant with respect to the frame $\vec{x}$, provided P does not involve τ_0, τ or zero-subscripted variables, and has no unindexed references to program variables that are in the frame, $\vec{x}$, or to external inputs.*

Proof. According to Def. 20 (frame-invariant), because P does not contain τ, $P \mathbin{@} \tau_0$ is P with every occurrence of an unindexed variable, v, replaced by $v(\tau_0)$. However, every such v is a program variable that is not in the frame, and hence is stable, so $v(\tau_0) = v(\tau)$. Therefore $P \mathbin{@} \tau_0 \Rrightarrow P \mathbin{@} \tau$ as required. □

Law 23 (idle-invariant property) *Given an environment, ρ, and a predicate, P, then P is idle-invariant, provided P does not involve τ_0, τ or zero-subscripted variables, and has no unindexed references to external inputs.*

Proof. The proof follows directly from Def. 21 (idle-invariant), and Law 22 (frame-invariant-property) with an empty frame. □

Law 24 (frame invariant) *Given an environment, ρ, a frame, $\vec{x}$, and a predicate, P, that is frame invariant with respect to $\vec{x}$, then*

$$\star\vec{x}{:}\,[P \wedge Q,\ \ P \wedge R] \sqsubseteq_\rho \star\vec{x}{:}\,[P \wedge Q,\ \ R]$$

Proof. The refinement from right to left is a trivial application of strengthening the effect. The refinement from left to right is also an application of strengthening the effect. The proviso for the latter is

$$(P \wedge Q \mathbin{@} \tau_0) \wedge (R \mathbin{@} (\tau_0, \tau)) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \Rrightarrow (P \wedge R \mathbin{@} (\tau_0, \tau))$$

but as P is frame invariant with respect to $\vec{x}$,

$$(P \mathbin{@} \tau_0) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\!| \tau_0 \mathinner{..} \tau |\!]) \Rrightarrow P \mathbin{@} \tau$$

and hence the proviso holds. □

Idling only changes time so any property invariant over time is maintained by **idle**.

Law 25 (idle invariant) *Given an environment, ρ, if P is an idle-invariant predicate, then*

$$\star\,[P,\ \ P] \sqsubseteq_\rho \mathbf{idle}$$

Proof. If P is idle-invariant, then by definition it is frame-invariant for the empty frame. Hence, we use Law 24 (frame-invariant) with an empty frame and Q and R both *true*.

$\star\,[P,\ \ P]$
$\sqsupseteq_\rho$ Law 24 (frame-invariant)
$\star\,[P,\ \ \mathit{true}]$
$\sqsubseteq_\rho$ Law 6 (weaken-assumption)
$\star\,[\mathit{true}]$
$\sqsupseteq_\rho$ Def. 18 (idle)
idle □

Assumptions that are invariant over time and do not depend on variables in the frame are invariant over the execution of a specification command.

Law 26 (frame-invariant assumption) *Given an environment, ρ, a frame, $\vec{x}$, and a predicate, P, that is frame-invariant with respect to $\vec{x}$, then*

$$\star\,\{P\}\ ;\ \star\vec{x}{:}\,[Q,\ \ R] \sqsupseteq_\rho \star\,\{P\}\ ;\ \star\vec{x}{:}\,[Q,\ \ R]\ ;\ \star\,\{P\}$$

Proof.

$\star\,\{P\}\ ;\ \star\vec{x}{:}\,[Q,\ \ R]$
$\sqsupseteq_\rho$ Law 13 (separate-assumption)
$\star\vec{x}{:}\,[P \wedge Q,\ \ R]$
$\sqsupseteq_\rho$ Law 24 (frame-invariant)
$\star\vec{x}{:}\,[P \wedge Q,\ \ P \wedge R]$
$\sqsupseteq_\rho$ Law 14 (establish-assumption)
$\star\vec{x}{:}\,[P \wedge Q,\ \ P \wedge R]\ ;\ \star\,\{P\}$
$\sqsupseteq_\rho$ Law 24 (frame-invariant)
$\star\vec{x}{:}\,[P \wedge Q,\ \ R]\ ;\ \star\,\{P\}$
$\sqsupseteq_\rho$ Law 13 (separate-assumption)
$\star\,\{P\}\ ;\ \star\vec{x}{:}\,[Q,\ \ R]\ ;\ \star\,\{P\}$ □

Law 27 (idle-invariant assumption) *Given an environment, ρ, and an idle-invariant predicate, P, then*

$$\star\{P\}\,;\,\mathbf{idle} \sqsupseteq_\rho \star\{P\}\,;\,\mathbf{idle};\ \star\{P\}$$

Proof. The law follows from Law 26 (frame-invariant-assumption) with an empty frame and Def. 18 (idle). □

Definition 28 (frame-stable expression) *Given an environment, ρ, and a frame, $\vec{x}$, then an expression, D, that contains no references to τ_0 or zero-subscripted variables, is* frame stable *with respect to $\vec{x}$, if and only if*

$$\tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \Rrightarrow (D \mathbin{@} \tau_0) = (D \mathbin{@} \tau)$$

Law 29 (frame-stable expression) *Given an environment, ρ, and a frame, $\vec{x}$, an expression, D, is frame stable with respect to $\vec{x}$, provided D contains no occurrence of τ_0, τ, or zero-subscripted variables, and has no unindexed references to variables in the frame, $\vec{x}$, or unindexed references to external inputs.*

Proof. Because D does not contain τ, $D \mathbin{@} \tau_0$ is D with every unindexed variable, v, replaced by $v(\tau_0)$, but because D does not contain unindexed references to variables in the frame or external inputs, each such v is stable, so $v(\tau_0) = v(\tau)$. Hence, $D \mathbin{@} \tau_0 = D \mathbin{@} \tau$. □

Definition 30 (idle-stable expression) *Given an environment, ρ, an expression, D, is* idle stable, *if and only if D is a frame-stable expression with respect to the empty frame.*

Law 31 (idle-stable expression) *Given an environment, ρ, an expression, D, is idle stable, provided D contains no occurrence of τ_0, τ, or zero-subscripted variables, and has no unindexed references to external inputs.*

Proof. The proof follows from Def. 30 (idle-stable-expression) and Law 29 (frame-stable-expression) with an empty frame. □

There is a difference between a predicate being idle-invariant and a (Boolean) expression being idle-stable. For an idle-invariant predicate, if it is true before an idle, then it must be true after the idle, but it may be false before the idle and become true on executing the idle. However, for an idle-stable (Boolean) expression, it must have the same value before and after the idle. For example, if D is an idle-stable expression, then $D \leq \tau$ is idle-invariant, but not idle-stable because $D \leq \tau$ being false at τ does not preclude it being true at some later time.

2.9 Assignment

Because the evaluation of the expressions in an assignment takes time, we require that the expressions in assignments are idle-stable.

Definition 32 (assignment) *Given an environment, ρ, a frame, $\vec{x}$, such that $\vec{x} \subseteq \hat{\rho}$, and a vector, $\vec{D}$, of idle-stable expressions, where the lengths of $\vec{x}$ and $\vec{D}$ are the same and the types of the corresponding elements in $\vec{x}$ and $\vec{D}$ are assignment compatible*

$$\vec{x} := \vec{D} \mathrel{\widehat{=}}_\rho \star\vec{x}: \left[(\vec{x} \mathbin{@} \tau) = (\vec{D} \mathbin{@} \tau_0) \right]$$

For all assignment commands in the remainder of this paper we assume that the lengths of the vectors $\vec{x}$ and $\vec{D}$ are the same, and that the corresponding elements are assignment compatible.

Law 33 (assignment) *Given an environment, ρ, a frame, $\vec{x}$, and a vector, $\vec{D}$, of idle-stable expressions, provided*

$$(P \mathbin{@} \tau_0) \wedge (\vec{x} \mathbin{@} \tau) = (\vec{D} \mathbin{@} \tau_0) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\!\!\lfloor \tau_0 \ldots \tau \rceil\!\!]) \\ \Rrightarrow R \mathbin{@} (\tau_0, \tau)$$

then

$$\star\vec{x}: [P,\ \ R] \sqsubseteq_\rho \vec{x} := \vec{D}.$$

If the properties do not involve time then the following special case of the proviso can be used

$$P_0 \wedge \vec{x} = \vec{D}_0 \Rrightarrow R$$

where P_0 and D_0 are the same as P and D, respectively, but with every unindexed occurrence of a variable, v, that is in the frame or is an external input, replaced by v_0.

Proof.

$\star\vec{x}: [P,\ \ R]$
$\sqsubseteq_\rho$ Law 7 (strengthen-effect) using proviso; Law 6 (weaken-assumption)
$\star\vec{x}: \left[(\vec{x} \mathbin{@} \tau) = (\vec{D} \mathbin{@} \tau_0) \right]$
$\sqsupseteq_\rho$ Def. 32 (assignment)
$\vec{x} := \vec{D}$ □

2.10 Reading an external input

An external input may change without the action of the program directly causing the change. For example, an input buffer register may change when an input is received. The command $x : \mathbf{read}(v)$ reads the external variable v and places its value in the program variable x. The value selected is any of the values that v takes during the execution of the command. In practice, one often requires that v is stable for the duration of the command, so that the value read is uniquely defined.

Definition 34 (read) *Given an environment, ρ, provided x is contained in the program variables, $\hat{\rho}$, and v is an external input ($v \in \rho.in$),*

$$x : \mathbf{read}(v) \mathrel{\widehat{=}}_\rho \star x{:}\ [x \in v(\!|\llbracket \tau_0 \mathinner{..} \tau \rrbracket|\!)]$$

where $v(\!|\llbracket \tau_0 \mathinner{..} \tau \rrbracket|\!)$ is the image of the set $\llbracket \tau_0 \mathinner{..} \tau \rrbracket$ through v, i.e., all the values of v during the closed interval from τ_0 to τ.

2.11 Reading the clock

The $\mathbf{gettime}$ command returns the current time. The value returned is between the start and finish times of the command. Because we use real numbers to model time, we cannot have a type in the programming language corresponding exactly to *Time*. To overcome this we introduce the programming language type `time`, where `time` $\subset$ *Time* (although `time` may be represented within the implementation as an integer in some appropriate units, e.g., nanoseconds).

Definition 35 (gettime) *Given an environment, ρ, provided x is a program variable ($x \in \hat{\rho}$) of type* `time`,

$$x : \mathbf{gettime} \mathrel{\widehat{=}}_\rho \star x{:}\ [x \in \llbracket \tau_0 \mathinner{..} \tau \rrbracket]$$

Note that we assume a perfect clock with no drift, or perhaps we should say that the time base of our system is that provided by the implementation of $\mathbf{gettime}$, rather than some more global time, such as Universal Time Coordinated (UTC).

Also note that we assume that a clock tick will occur during the execution of the $\mathbf{gettime}$ command, so that there exists an element of `time` in the range $\llbracket \tau_0 \mathinner{..} \tau \rrbracket$. This is a reasonable assumption provided the clock resolution is fine enough — close to the machine instruction time — but for a larger clock resolution the definition may have to be modified to take the clock resolution into account. In that case we could define $\mathbf{gettime}$ by

$$x : \mathbf{gettime} \mathrel{\widehat{=}}_\rho \star x{:}\ [x \in \llbracket \tau_0 - \mathit{resolution} \mathinner{..} \tau \rrbracket]$$

where *resolution* is the time between clock ticks. For the remainder of this paper we assume the simpler definition.

There is also the problem of the finite bound on the implementation of type `time`. We assume that the implementation of type `time` is sufficiently large (at least well beyond the year 10,000) that this will not be a problem in practice. In practice, an implementation only guarantees to meet the specification until the end of `time`, not until the end of *Time*.

2.12 Delay until

A delay command guarantees that its completion is after the specified time.

Definition 36 (delay) *Given an environment, ρ, provided D is an idle-stable time-valued expression,*

$$\mathbf{delay\ until}\, D \mathrel{\widehat{=}}_\rho \star \left[D \leq \tau\right].$$

2.13 Deadline

The deadline directive allows a time deadline to be specified. It is the compiler's responsibility to ensure the deadline is met by the generated code. If the compiler cannot, it must report a compile-time error.

Definition 37 (deadline) *Given an environment, ρ, provided D is a time-valued expression, which may include references to logical constants but no references to τ_0 or zero-subscripted variables,*

$$\mathbf{deadline}\, D \mathrel{\widehat{=}}_\rho \star \left[\tau_0 = \tau \wedge \tau \leq D\right].$$

Because the **deadline** command takes no time, there is no need to require D to be idle-stable. From the definition of the specification command we have

$$\begin{array}{ll} & \mathcal{M}_\rho\,(\mathbf{deadline}\, D) \\ \sqsupseteq & \tau\colon \left[\tau_0 = \tau \wedge \tau \leq D \mathbin{@} \tau \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho}, \left[\!\left[\tau_0 \mathinner{..} \tau\right]\!\right])\right] \\ \sqsupseteq & \text{As } \left[\!\left[\tau_0 \mathinner{..} \tau\right]\!\right] = \{\tau\} \\ & \tau\colon \left[\tau_0 = \tau \wedge \tau \leq D \mathbin{@} \tau\right] \\ \sqsupseteq & \left[\tau \leq D \mathbin{@} \tau\right] \end{array}$$

In refinement calculus terms, a deadline directive is a coercion. A timing path analysis is required to show that the deadline directive is always reached before its deadline. In that context the directive can be eliminated. Otherwise the program cannot be successfully compiled. For more discussion on the deadline directive see Sections 1.2, 3 and 5.2.

2.14 Logical constants

Logical constants carry over from the standard refinement calculus. Note that, unlike variables, logical constants are not implicit functions of time. The definition of a logical constant block is given in terms of a predicate transformer, whose goal may not contain free occurrences of the introduced name. To account for this, the definition also includes the ability to systematically change the name of a logical constant to some fresh name.

Definition 38 (logical constant) *Given an environment, ρ, provided u does not occur free in G or var(ρ),*

$$\mathcal{M}_\rho\,(|[\,\mathbf{con}\,u : T \bullet C\,]|)\,(G) \mathrel{\widehat{=}} (\exists\, u : T \bullet \mathcal{M}_\rho\,(C)\,(G))$$

and provided y does not occur free in C,

$$|[\,\mathbf{con}\,u : T \bullet C\,]| \sqsubseteq_\rho |[\,\mathbf{con}\,y : T \bullet C\,[\tfrac{y}{u}]\,]|$$

Law 39 (logical constant) *Given an environment, ρ, provided u does not occur in C or var(ρ), and $(\exists\, u : T \bullet P \mathbin{@} \tau)$,*

$$C \sqsubseteq_\rho |[\,\mathbf{con}\,u : T \bullet \star\{P\}\,;\,C\,]|$$

Proof. Assume *u* does not occur free in *G* (otherwise change *u* to a fresh name).

$\mathcal{M}_\rho\,(|[\,\mathbf{con}\,u : T \bullet \star\{P\}\,;\,C\,]|)\,(G)$
$\equiv$ Def. 38 (logical-constant)
$(\exists\, u : T \bullet \mathcal{M}_\rho\,(\star\{P\}\,;\,C)\,(G))$
$\equiv$ Def. 12 (sequential-composition); Def. 10 (assumption)
$(\exists\, u : T \bullet P \mathbin{@} \tau \wedge \mathcal{M}_\rho\,(C)\,(G))$
$\equiv$ *u* does not occur free in *C* or *G* or *var*(ρ), and hence $\mathcal{M}_\rho\,(C)\,(G)$
$(\exists\, u : T \bullet P \mathbin{@} \tau) \wedge \mathcal{M}_\rho\,(C)\,(G)$
$\equiv$ from assumption
$\mathcal{M}_\rho\,(C)\,(G)$ $\square$

Law 40 (remove logical constant) *Given an environment, ρ, provided u does not occur in C or var(ρ), and T is non-empty,*

$$|[\,\mathbf{con}\,u : T \bullet C\,]| \sqsubseteq_\rho C$$

Proof. This is a special case of Law 39 (logical-constant) with *P* the predicate *true*. $\square$

2.15 Refinement to a sequential composition

The refinement of a specification command to a sequential composition of specification commands follows the same approach as in the standard refinement calculus. One must devise an intermediate predicate Q that holds on termination of the first component, and hence also for the assumption of the second component. Because we have assumed that there is no time delay between the execution of the two commands, τ in the effect of the first component refers to the same time as τ in the assumption of the second component. We would also like to allow references to τ_0 within the effect of the first component. However, references to τ_0 are not allowed within the assumptions of the second component. To cope with this we use the approach used by Morgan [18] to handle zero-subscripted variables within a sequential composition and introduce a logical constant u to stand for the start time of the first component, and replace all occurrences of τ_0 by u within the assumptions of the second component. In the desired overall effect, R, τ_0 refers to the commencement of the whole sequential composition (not the commencement of the second component). Hence the references to τ_0 in the effect of the second component are also replaced by u.

Law 41 (sequential composition) *Given an environment, ρ, provided u is a fresh name,*

$$\begin{array}{l} \quad \star\vec{x}\colon [P,\ \ R] \\ \sqsubseteq_\rho \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [u = \tau \wedge P,\ \ Q]\ ;\ \star\vec{x}\colon [Q \mathbin{@} (u,\tau),\ \ R \mathbin{@} (u,\tau)]\ ]| \end{array}$$

Q may contain unindexed references to variables. For example, a reference to v is treated as $v(\tau)$. Because τ is the finish time within Q in the first command as well as being the start time within Q in the second command, references to v within the two occurrences of Q refer to the value of v at the same time.

Proof.

$$\begin{array}{l} \quad \star\vec{x}\colon [P,\ \ R] \\ \sqsubseteq_\rho \text{ Law 39 (logical-constant); } u \text{ fresh} \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star \{u = \tau\}\ ;\ \star\vec{x}\colon [P,\ \ R]\ ]| \end{array}$$

We proceed by refining the body of the block.

$$\begin{array}{l} \quad \mathcal{M}_\rho\,(\star \{u = \tau\}\ ;\ \star\vec{x}\colon [P,\ \ R]) \\ \sqsubseteq \text{ Law 13 (separate-assumption); Def. 4 (specification)} \\ \quad \tau\colon [u = \tau \wedge P \mathbin{@} \tau,\ \ R \mathbin{@} (\tau_0,\tau) \wedge \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \lceil \tau_0 \mathinner{..} \tau \rceil)] \\ \sqsubseteq \text{ Standard strengthen postcondition} \end{array}$$

τ: $\left[u = \tau \wedge P \mathbin{@} \tau,\ \ R \mathbin{@} (u,\tau) \wedge u \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket u \mathinner{..} \tau \rrbracket)\right]$

$\sqsubseteq$ Standard sequential composition

τ: $\left[u = \tau \wedge P \mathbin{@} \tau,\ \begin{array}{l} Q \mathbin{@} (u,\tau) \wedge u \leq \tau \wedge \\ \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket u \mathinner{..} \tau \rrbracket) \end{array}\right] \circ$

τ: $\left[\begin{array}{ll} Q \mathbin{@} (u,\tau) \wedge u \leq \tau \wedge & R \mathbin{@} (u,\tau) \wedge u \leq \tau \wedge \\ \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket u \mathinner{..} \tau \rrbracket)\text{'} & \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket u \mathinner{..} \tau \rrbracket) \end{array}\right]$

$\sqsubseteq$ Standard strengthen postcondition (twice)

τ: $\left[u = \tau \wedge P \mathbin{@} \tau,\ \begin{array}{l} Q \mathbin{@} (\tau_0,\tau) \wedge \tau_0 \leq \tau \wedge \\ \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right] \circ$

τ: $\left[\begin{array}{ll} Q \mathbin{@} (u,\tau) \wedge u \leq \tau \wedge & R \mathbin{@} (u,\tau) \wedge \tau_0 \leq \tau \wedge \\ \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket u \mathinner{..} \tau \rrbracket)\text{'} & \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right]$

$\sqsubseteq$ Def. 4 (specification); standard weaken precondition

$\mathcal{M}_\rho\,(\star\vec{x}\colon [u = \tau \wedge P,\ \ Q]) \circ$

τ: $\left[Q \mathbin{@} (u,\tau) \mathbin{@} \tau,\ \begin{array}{l} R \mathbin{@} (u,\tau) \mathbin{@} (\tau_0,\tau) \wedge \tau_0 \leq \tau \wedge \\ \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right]$

$\sqsubseteq$ Def. 4 (specification)

$\mathcal{M}_\rho\,(\star\vec{x}\colon [u = \tau \wedge P,\ \ Q]) \circ \mathcal{M}_\rho\,(\star\vec{x}\colon [Q \mathbin{@} (u,\tau),\ \ R \mathbin{@} (u,\tau)])$

$\sqsupseteq$ Def. 12 (sequential-composition)

$\mathcal{M}_\rho\,(\star\vec{x}\colon [u = \tau \wedge P,\ \ Q]\ ;\ \star\vec{x}\colon [Q \mathbin{@} (u,\tau),\ \ R \mathbin{@} (u,\tau)])$ $\Box$

If Q and R do not involve τ_0 or zero-subscripted variables, the above rule can be simplified.

Law 42 (simple sequential composition) *Given an environment, ρ, provided Q and R do not involve τ_0 or zero-subscripted variables,*

$$\star\vec{x}\colon [P,\ \ R] \sqsubseteq_\rho \star\vec{x}\colon [P,\ \ Q]\ ;\ \star\vec{x}\colon [Q,\ \ R]$$

Proof.

$\star\vec{x}\colon [P,\ \ R]$

$\sqsubseteq_\rho$ Law 41 (sequential-composition); Law 6 (weaken-assumption)

$[\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ \ Q]\ ;\ \star\vec{x}\colon [Q \mathbin{@} (u,\tau),\ \ R \mathbin{@} (u,\tau)]\ ]\!]$

$\sqsupseteq_\rho$ from proviso $Q \mathbin{@} (u,\tau) = Q \mathbin{@} (\tau_0,\tau)$ and similarly for R

$[\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ \ Q]\ ;\ \star\vec{x}\colon [Q,\ \ R]\ ]\!]$

$\sqsubseteq_\rho$ Law 40 (remove-logical-constant)

$\star\vec{x}\colon [P,\ \ Q]\ ;\ \star\vec{x}\colon [Q,\ \ R]$ $\Box$

A timing deadline in the effect of a specification command may be separated out into a deadline command.

Law 43 (separate deadline) *Given an environment, ρ, provided D is a time-valued expression, which may include references to logical constants but no references to τ_0 or zero-subscripted variables,*

$$\star\vec{x}\colon [P,\ \ R \wedge \tau \leq D] \sqsubseteq_\rho \star\vec{x}\colon [P,\ \ R]\ ;\ \mathbf{deadline}\, D$$

Proof.

$\star\vec{x}\colon [P,\ R \wedge \tau \leq D]$
$\sqsubseteq_\rho$ Law 41 (sequential-composition); Law 6 (weaken-assumption)
$\quad [\![\mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \star\vec{x}\colon [P,\ R]\ ;\ \star\vec{x}\colon [R \mathbin{@} (u,\tau),\ (R \wedge \tau \leq D) \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ as τ_0 does not occur in D; Law 9 (remove-from-frame)
$\quad [\![\mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ R]\ ;\ \star\ [R \mathbin{@} (u,\tau),\ R \mathbin{@} (u,\tau) \wedge \tau \leq D]\ ]\!]$
$\sqsubseteq_\rho$ Law 7 (strengthen-effect)
$\quad [\![\mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \star\vec{x}\colon [P,\ R]\ ;\ \star\ [R \mathbin{@} (u,\tau),\ R \mathbin{@} (u,\tau) \wedge \tau_0 = \tau \wedge \tau \leq D]\ ]\!]$
$\sqsubseteq_\rho$ Law 7 (strengthen-effect); Law 6 (weaken-assumption)
$\quad [\![\mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ R]\ ;\ \star\ [\tau_0 = \tau \wedge \tau \leq D]\ ]\!]$
$\sqsubseteq_\rho$ Law 40 (remove-logical-constant); Def. 37 (deadline)
$\quad \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{deadline}\, D\ \Box$

2.16 Prefix and postfix idle commands

In this section we examine the properties required by specification commands in order to be able to prefix (postfix) them with an **idle** command. To prefix a specification by an **idle** command, the assumption of the specification must be idle-invariant, and additionally, the effect of the specification must be able to tolerate the prefix of the **idle** command. We refer to an effect with this property as being *pre-idle-invariant*.

Definition 44 (pre-idle-invariant) *Given an environment, ρ, a predicate R (which may involve τ_0 and zero-subscripted variables) is* pre-idle-invariant, *if and only if for any u not occurring in R or var(ρ),*

$$R \mathbin{@} (\tau_0, \tau) \wedge u \leq \tau_0 \wedge \mathit{stable}(\hat{\rho}, \lceil u \mathinner{..} \tau_0 \rceil) \Rrightarrow R \mathbin{@} (u, \tau)$$

Law 45 (pre-idle-invariant property) *Given an environment, ρ, if a predicate R has no references to τ_0 and all references to zero-subscripted variables in R are to program variables, then R is pre-idle-invariant.*

Proof. As R does not contain τ_0, $R \mathbin{@} (u,\tau)$ is R with every occurrence of a zero-subscripted variable, v_0, replaced by $v(u)$, but all such variables are stable over the interval $\lceil u \mathinner{..} \tau_0 \rceil$, and hence $v(u) = v(\tau_0)$. Therefore, $R \mathbin{@} (\tau_0, \tau) \Rrightarrow R \mathbin{@} (u, \tau)$. $\Box$

Law 46 (idle before) *Given an environment, ρ, provided P is idle-invariant, and R is pre-idle-invariant, then*

$$\star\vec{x}\colon [P,\ R]\ \sqsubseteq_\rho\ \mathbf{idle};\ \star\vec{x}\colon [P,\ R]$$

Proof. The refinement from right to left is a straightforward application of Law 19 (skip-idle) followed by Law 16 (skip-identity). The refinement from left to right follows.

$\star\vec{x}\colon [P,\ R]$
$\sqsubseteq_\rho$ Law 41 (sequential-composition); Law 6 (weaken-assumption)
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ P \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil \tau_0 \mathinner{..} \tau \rceil)]\ ;$
$\qquad \star\vec{x}\colon [P \mathbin{@} (u,\tau) \wedge u \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil u \mathinner{..} \tau \rceil),\ R \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ Law 9 (remove-from-frame); Law 7 (strengthen-effect); no τ_0 in P
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \star\,[P,\ P]\ ;\ \star\vec{x}\colon [P \wedge u \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil u \mathinner{..} \tau \rceil),\ R \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ Law 25 (idle-invariant)
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \mathbf{idle};\ \star\vec{x}\colon [P \wedge u \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil u \mathinner{..} \tau \rceil),\ R \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ Law 7 (strengthen-effect); R pre-idle-invariant
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet \mathbf{idle};\ \star\vec{x}\colon [P \wedge u \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil u \mathinner{..} \tau \rceil),\ R]\ ]\!]$
$\sqsubseteq_\rho$ Law 6 (weaken-assumption); Law 40 (remove-logical-constant)
$\quad \mathbf{idle};\ \star\vec{x}\colon [P,\ R]$ $\Box$

An **idle** command may be postfixed to a specification command provided the effect of the specification tolerates the **idle** command. We refer to effects with this property as being *post-idle-invariant*.

Definition 47 (post-idle-invariant) *Given an environment, ρ, a predicate R is* post-idle-invariant *if and only if for any u not occurring in R or $\mathit{var}(\rho)$, $R \mathbin{@} (u,\tau)$ is idle-invariant. That is,*

$$R \mathbin{@} (u,\tau_0) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho}, \lceil \tau_0 \mathinner{..} \tau \rceil) \Rrightarrow R \mathbin{@} (u,\tau)$$

Law 48 (idle after) *Given an environment of variables, ρ, provided R is post-idle-invariant,*

$$\star\vec{x}\colon [P,\ R] \sqsupseteq_\rho \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{idle}$$

Proof. The refinement from right to left follows from Law 19 (skip-idle) followed by Law 16 (skip-identity). The refinement from left to right follows.

$\star\vec{x}\colon [P,\ R]$
$\sqsubseteq_\rho$ Law 41 (sequential-composition); Law 6 (weaken-assumption)
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ R]\ ;\ \star\vec{x}\colon [R \mathbin{@} (u,\tau),\ R \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ Law 9 (remove-from-frame)
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ R]\ ;\ \star\,[R \mathbin{@} (u,\tau),\ R \mathbin{@} (u,\tau)]\ ]\!]$
$\sqsubseteq_\rho$ Law 25 (idle-invariant); R post-idle-invariant
$\quad [\![\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{idle}\ ]\!]$
$\sqsubseteq_\rho$ Law 40 (remove-logical-constant)
$\quad \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{idle}$ $\Box$

2.17 Local variables

The definition of a local variable block in the real-time language involves expanding the set of program variables for the command within the block. Hence we need to give the definition in terms of the meaning function $\mathcal{M}$ introduced in Sect. 2.2, instead of using the abbreviated form of definition. Because allocating and deallocating a local variable may take time, **idle** commands are used to represent possible time delays. In the definition of *AddVar* below, a fresh variable name, v', is used in place of the name v in order to allow the local variable to have the same name as an existing variable. The renaming is necessary because the local variable block must ensure the stability of the existing variable. Of course, if v itself is a fresh name, the renaming is unnecessary.

Definition 49 (local block) *Given an environment, ρ, provided that the type T is nonempty,*

$$|[\ \mathbf{var}\ v : T \bullet C\ ]| \sqsupseteq_\rho \mathbf{idle}; \mathit{AddVar}_{v:T}(C); \mathbf{idle}$$

where $\mathit{AddVar}_{v:T}$ is defined as follows: provided v' is a fresh name ($v' \notin \mathit{var}(\rho)$), and v' does not occur free in the goal predicate, G,

$$\mathcal{M}_\rho\,(\mathit{AddVar}_{v:T}(C))\,(G) \mathrel{\widehat{=}} (\forall v' : \mathit{Time} \to T \bullet \mathcal{M}_{\rho'}\left(C\left[\tfrac{v'}{v}\right]\right)(G))$$

where ρ' is the environment ρ with its local variables extended by v', that is, $\rho' = \rho +_l v'$ where

$$\begin{array}{|l} \_ +_l \_ : \mathit{Env} \times \mathit{Ident} \to \mathit{Env} \\ \hline (\rho +_l v' = \rho') \Leftrightarrow \\ (\rho'.\mathit{in} = \rho.\mathit{in} \wedge \rho'.\mathit{out} = \rho.\mathit{out} \wedge \rho'.\mathit{local} = \rho.\mathit{local} \cup \{v'\}) \end{array}$$

An alternative approach is not to treat local variables as timed traces, but as standard refinement calculus local variables. With the alternative approach one can only use v and v_0 to reference the value of local variables, but it does allow an expand frame law [18].

Law 50 (introduce variable) *Given an environment, ρ, and a variable name, v, provided v does not occur free within P or R or $\vec{x}$, T is nonempty, P is idle-invariant and R is both pre- and post-idle-invariant, then*

$$\star\vec{x}\colon [P,\ R] \sqsupseteq_\rho |[\ \mathbf{var}\ v : T \bullet \star v, \vec{x}\colon [P,\ R]\ ]|$$

Proof. First we introduce the **idle** commands in Def. 49 (local-block).

$\star\vec{x}\colon [P,\ R]$
$\sqsupseteq_\rho$ Law 46 (idle-before); Law 48 (idle-after)
$\mathbf{idle};\ \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{idle}$

Next we show that $\mathit{AddVar}_{v:T}(\star v, \vec{x}\colon [P,\ R])$ is equivalent to $\star\vec{x}\colon [P,\ R]$. Let $\rho' = \rho +_l v'$ and assume v' does not occur free in G.

$\mathcal{M}_\rho\,(\mathit{AddVar}_{v:T}(\star v, \vec{x}\colon [P,\ R])\,(G)$
$\equiv$ Definition of *AddVar*; v does not occur in P, R and $\vec{x}$
$(\forall v' : \mathit{Time} \to T \bullet \mathcal{M}_{\rho'}\,(\star v', \vec{x}\colon [P,\ R])\,(G))$
$\equiv$ Def. 4 (specification)
$(\forall v' : \mathit{Time} \to T \bullet P \mathbin{@} \tau \wedge$
$\quad (\forall \tau \bullet R \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}((\hat{\rho}, v') \setminus (v', \vec{x}), [\![\tau_0 \mathinner{..} \tau]\!])$
$\qquad \Rightarrow G))$
$\equiv$ as v' does not occur in P, R, G or $\vec{x}$; v' is disjoint from $\hat{\rho}$; T nonempty
$P \mathbin{@} \tau \wedge (\forall \tau \bullet R \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \Rightarrow G)$
$\equiv$ Def. 4 (specification)
$\mathcal{M}_\rho\,(\star\vec{x}\colon [P,\ R])\,(G)$ $\Box$

Example: Relative delay. A relative delay of length D, where D is an idle-stable time-valued expression, can be specified by the following command.

$$\star\,[\tau_0 + D \le \tau]$$

It can be implemented by getting the current time and then using an (absolute) delay until command to delay until the current time plus D.

$\star\,[\tau_0 + D \le \tau]$
$\sqsubseteq_\rho$ Law 50 (introduce-variable); $\tau_0 + D \le \tau$ pre- and post-idle-invariant
$\quad [\![\mathbf{var}\, c : \mathtt{time} \bullet \star c\colon [\tau_0 + D \le \tau]\ ]\!]$
$\sqsubseteq_\rho$ Law 41 (sequential-composition)
$\quad [\![\mathbf{var}\, c : \mathtt{time} \bullet [\![\mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \star c\colon [u = \tau,\ \tau_0 \le c]\ ;\ \star c\colon [u \le c,\ u + D \le \tau]\ ]\!]\ ]\!]$
$\sqsubseteq_\rho$ Law 7 (strengthen-effect); Law 9 (remove-from-frame)
$\quad [\![\mathbf{var}\, c : \mathtt{time} \bullet [\![\mathbf{con}\, u : \mathit{Time} \bullet$
$\qquad \star c\colon [u = \tau,\ c \in [\![\tau_0 \mathinner{..} \tau]\!]]\ ;\ \star\,[u \le c,\ c + D \le \tau]\ ]\!]\ ]\!]$
$\sqsubseteq_\rho$ Law 6 (weaken-assumption); Law 40 (remove-logical-constant)
$\quad [\![\mathbf{var}\, c : \mathtt{time} \bullet \star c\colon [c \in [\![\tau_0 \mathinner{..} \tau]\!]]\ ;\ \star\,[c + D \le \tau]\ ]\!]$
$\sqsubseteq_\rho$ Def. 35 (gettime); Def. 36 (delay)
$\quad [\![\mathbf{var}\, c : \mathtt{time} \bullet c : \mathbf{gettime};\ \mathbf{delay\ until}\, c + D\ ]\!]$

2.18 Outputs

Like local variables, outputs are program variables, but they differ from local variables in that they are externally visible.

Definition 51 (output) *Given an environment, ρ, provided $v \notin var(\rho)$, then for all goal predicates, G, which* may *include references to v,*

$$\mathcal{M}_\rho\,(\mathbf{output}\, v : T \bullet C)\,(G) \mathrel{\widehat=} (\forall\, v : Time \to T \bullet \mathcal{M}_{\rho +_o v}\,(C)\,(G))$$

where

$$\begin{array}{|l} \_ +_o \_ : Env \times Ident \to Env \\ \hline (\rho +_o v = \rho') \Leftrightarrow \\ (\rho'.in = \rho.in \wedge \rho'.out = \rho.out \cup \{v\} \wedge \rho'.local = \rho.local) \end{array}$$

Note that, unlike local variables, it is not possible to change the name of an output, and that the goal, G, may refer to the output, v. The only other difference between local variables and output variables is that output variables are global (not stack-allocated), so we assume that no time is needed to allocate or deallocate them.

2.19 Inputs

External inputs are not under the control of the program. Hence they are not considered to be program variables. In addition, external inputs are externally visible. The only way a program can affect the value of an external input is by modifying an output, that indirectly controls the value of the input, via the environment.

Definition 52 (input) *Given an environment, ρ, provided $v \notin var(\rho)$, then for all goal predicates, G,*

$$\mathcal{M}_\rho\,(\mathbf{input}\, v : T \bullet C)\,(G) \mathrel{\widehat=} (v \in Time \to T) \wedge \mathcal{M}_{\rho +_i v}\,(C)\,(G)$$

where

$$\begin{array}{|l} \_ +_i \_ : Env \times Ident \to Env \\ \hline (\rho +_i v = \rho') \Leftrightarrow \\ (\rho'.in = \rho.in \cup \{v\} \wedge \rho'.out = \rho.out \wedge \rho'.local = \rho.local) \end{array}$$

The purpose of declaring inputs is to introduce them as part of the name space, as well as declare their type. Note that, unlike local variables, it is not possible to change the name of an input, and that the goal, G, may refer to the input, v.

2.20 Alternation

An alternation is defined in terms of a standard alternation. The guards are required to be idle-stable expressions so that they are stable during their evaluation. For the definition we use their value at the start time of the whole alternation. The time taken to evaluate the guards and to exit a branch of the alternation is accounted for by including **idle** commands at the beginning and end of each branch of the alternation. Although the **idle** commands are defined to allow an arbitrary delay, **deadline** commands placed elsewhere in the program ensure that they cannot take too long.

Definition 53 (alternation) *Given an environment, ρ, provided $B_0, \ldots, B_n$ are idle-stable Boolean-valued expressions, then*

$$\mathcal{M}_\rho \begin{pmatrix} \star \textbf{ if } B_0 \to C_0 \\ \vdots \\ [\!] \; B_n \to C_n \\ \textbf{fi} \end{pmatrix} \mathrel{\widehat{=}} \begin{array}{l} \textbf{if } B_0 \mathbin{@} \tau \to \mathcal{M}_\rho\,(\textbf{idle}; C_0; \textbf{idle}) \\ \vdots \\ [\!] \; B_n \mathbin{@} \tau \to \mathcal{M}_\rho\,(\textbf{idle}; C_n; \textbf{idle}) \\ \textbf{fi} \end{array}$$

Introducing an alternation is similar to the standard refinement calculus rule.

Law 54 (alternation) *Given an environment, ρ, provided P is an idle-invariant predicate, $B_0, \ldots, B_n$ are Boolean-valued idle-stable expressions, and R is a predicate that is both pre- and post-idle-invariant, then*

$$\begin{array}{l} \star\vec{x}\colon \left[P \wedge (B_0 \vee \ldots \vee B_n),\ \ R\right] \\ \sqsubseteq_\rho \\ \star \textbf{ if } B_0 \to \star\vec{x}\colon \left[P \wedge B_0,\ \ R\right] [\!] \ldots [\!] \; B_n \to \star\vec{x}\colon \left[P \wedge B_n,\ \ R\right] \textbf{ fi} \end{array}$$

Proof. The introduction of the standard alternation relies on the equivalence of $P \wedge (B_0 \vee \ldots \vee B_n) \mathbin{@} \tau$ and $(P \mathbin{@} \tau) \wedge ((B_0 \mathbin{@} \tau) \vee \ldots \vee (B_n \mathbin{@} \tau))$.

$$\begin{array}{l} \mathcal{M}_\rho\,(\star\vec{x}\colon \left[P \wedge (B_0 \vee \ldots \vee B_n),\ \ R\right]) \\ \sqsubseteq \text{ Def. 4 (specification)} \\ \quad \tau\colon \left[P \wedge (B_0 \vee \ldots \vee B_n) \mathbin{@} \tau,\ \begin{array}{l} R \mathbin{@} (\tau_0, \tau) \\ \wedge\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right] \\ \sqsubseteq \text{ Standard alternation introduction} \\ \quad \textbf{if } B_0 \mathbin{@} \tau \to \tau\colon \left[P \wedge B_0 \mathbin{@} \tau,\ \begin{array}{l} R \mathbin{@} (\tau_0, \tau) \\ \wedge\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right] \\ \quad \vdots \\ \quad [\!] \; B_n \mathbin{@} \tau \to \tau\colon \left[P \wedge B_n \mathbin{@} \tau,\ \begin{array}{l} R \mathbin{@} (\tau_0, \tau) \\ \wedge\ \tau_0 \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right] \\ \quad \textbf{fi} \\ \sqsubseteq \text{ Def. 4 (specification)} \end{array}$$

$$
\begin{array}{l}
\quad \textbf{if } B_0 \mathbin{@} \tau \to \mathcal{M}_\rho \left(\star \vec{x}\colon \left[P \wedge B_0,\ \ R\right]\right) \\
\quad \vdots \\
\quad [\!] \ B_n \mathbin{@} \tau \to \mathcal{M}_\rho \left(\star \vec{x}\colon \left[P \wedge B_n,\ \ R\right]\right) \\
\quad \textbf{fi} \\
\sqsubseteq \text{ Law 46 (idle-before); Law 48 (idle-after)} \\
\quad \textbf{if } B_0 \mathbin{@} \tau \to \mathcal{M}_\rho \left(\textbf{idle};\ \star \vec{x}\colon \left[P \wedge B_0,\ \ R\right] ;\ \textbf{idle}\right) \\
\quad \vdots \\
\quad [\!] \ B_n \mathbin{@} \tau \to \mathcal{M}_\rho \left(\textbf{idle};\ \star \vec{x}\colon \left[P \wedge B_n,\ \ R\right] ;\ \textbf{idle}\right) \\
\quad \textbf{fi} \\
\sqsubseteq \text{ Def. 53 (alternation)} \\
\quad \mathcal{M}_\rho \left(\star \textbf{ if } B_0 \to \star \vec{x}\colon \left[P \wedge B_0,\ \ R\right] [\!] \ \ldots [\!] \ B_n \to \star \vec{x}\colon \left[P \wedge B_n,\ \ R\right] \textbf{ fi}\right) \ \ \square
\end{array}
$$

2.21 Iteration

An iteration is defined in terms of a standard iteration. The guard is restricted so that it is stable during its evaluation. To account for the delay to evaluate the guard and iterate or exit the loop, **idle** commands are introduced. To allow for loop exit overheads, including the case when the guard is initially false, an **idle** is added after the loop.

Definition 55 (iteration) *Given an environment, ρ, and a Boolean-valued idle-stable expression, B, which does not contain references to τ_0 or zero-subscripted variables,*

$$
\begin{array}{l}
\mathcal{M}_\rho \left(\star \textbf{ do } B \to C \textbf{ od}\right) \\
\widehat{=} \\
\textbf{do } B \mathbin{@} \tau \to \mathcal{M}_\rho \left(\textbf{idle};\ C;\ \textbf{idle}\right) \textbf{ od} \circ \mathcal{M}_\rho \left(\textbf{idle}\right)
\end{array}
$$

A multi-branch iteration may be defined in a similar manner. Introducing an iteration is similar to the standard refinement calculus rule.

Law 56 (iteration timing) *For the introduction of a loop one needs to supply a Boolean-valued guard expression, B, an invariant, INV, and a variant expression, V, which evaluates to an element of a well-founded set with ordering $\prec$, provided INV and B hold. Given an environment, ρ, provided B and V are idle-stable expressions, and INV is idle-invariant, and none of B, V and INV contain references to τ_0 or zero-subscripted variables, and u is a logical constant,*

$$
\begin{array}{l}
\quad \star \vec{x}\colon \left[INV \wedge u = \tau,\ \ \neg\, B \wedge INV'\right] \\
\sqsubseteq_\rho \\
\quad \star \textbf{ do } B \to \star \vec{x}\colon \left[B \wedge INV',\ \ INV' \wedge V \prec V_0\right] \textbf{ od}
\end{array}
$$

where $INV' \mathrel{\widehat{=}} INV \wedge u \leq \tau \wedge stable(\hat{\rho}\backslash\vec{x}, [\![u \mathinner{..} \tau]\!])$ *and* V_0 *is* V *with every unindexed reference to a variable,* v*, replaced by* v_0*, and every occurrence of* τ *replaced by* τ_0*.*

Proof. As the first step in the proof, we introduce the **idle** command at the end of Def. 55 (iteration). The introduction relies on $\neg B \wedge INV'$ being post-idle-invariant.

$$
\begin{array}{l}
\quad \star\vec{x}\colon [INV \wedge u = \tau,\ \neg B \wedge INV'] \\
\sqsupseteq_\rho \text{ Law 48 (idle-after)} \\
\quad \star\vec{x}\colon [INV \wedge u = \tau,\ \neg B \wedge INV'] \,;\, \mathbf{idle}
\end{array}
$$

We now proceed to refine the first component.

$$
\begin{array}{l}
\quad \mathcal{M}_\rho\,(\star\vec{x}\colon [INV \wedge u = \tau,\ \neg B \wedge INV']) \\
\sqsupseteq \text{ Def. 4 (specification)} \\
\quad \tau\colon \left[INV \wedge u = \tau \mathbin{@} \tau,\ \begin{array}{l} \neg B \wedge INV' \mathbin{@} (\tau_0, \tau) \\ \wedge\ \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \end{array}\right] \\
\sqsubseteq \text{ Definition of } INV' \\
\quad \tau\colon [INV' \mathbin{@} \tau,\ \neg B \wedge INV' \mathbin{@} \tau] \\
\sqsubseteq \text{ Standard iteration with variant } V \mathbin{@} \tau \\
\quad \mathbf{do}\, B \mathbin{@} \tau \to \tau\colon [B \wedge INV' \mathbin{@} \tau,\ (INV' \mathbin{@} \tau) \wedge (V \mathbin{@} \tau \prec V \mathbin{@} \tau_0)]\ \mathbf{od} \\
\sqsubseteq \text{ Definition of } INV'\text{; strengthen with } \tau_0 \leq \tau \\
\quad \mathbf{do}\, B \mathbin{@} \tau \to \\
\qquad \tau\colon \left[B \wedge INV' \mathbin{@} \tau,\ \begin{array}{l} INV' \wedge V \prec V_0 \mathbin{@} (\tau_0, \tau) \\ \wedge\ \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\![\tau_0 \mathinner{..} \tau]\!]) \end{array}\right] \\
\quad \mathbf{od} \\
\sqsupseteq \text{ Def. 4 (specification)} \\
\quad \mathbf{do}\, B \mathbin{@} \tau \to \mathcal{M}_\rho\,(\star\vec{x}\colon [B \wedge INV',\ INV' \wedge V \prec V_0])\ \mathbf{od}
\end{array}
$$

We now concentrate on the body of the loop. The following step relies on the fact that $B \wedge INV'$ is idle-invariant, and that $INV' \wedge V \prec V_0$ is both pre- and post-idle-invariant.

$$
\begin{array}{l}
\quad \star\vec{x}\colon [B \wedge INV',\ INV' \wedge V \prec V_0] \\
\sqsupseteq_\rho \text{ Law 46 (idle-before); Law 48 (idle-after)} \\
\quad \mathbf{idle};\ \star\vec{x}\colon [B \wedge INV',\ INV' \wedge V \prec V_0] \,;\, \mathbf{idle}
\end{array}
$$

Combining the above together we get the following.

$$
\begin{array}{l}
\quad \mathbf{do}\, B \mathbin{@} \tau \to \\
\qquad \mathcal{M}_\rho\,(\mathbf{idle};\ \star\vec{x}\colon [B \wedge INV',\ INV' \wedge V \prec V_0] \,;\, \mathbf{idle}) \\
\quad \mathbf{od} \circ \mathcal{M}_\rho\,(\mathbf{idle}) \\
\sqsupseteq \text{ Def. 55 (iteration)} \\
\quad \mathcal{M}_\rho\,(\star\,\mathbf{do}\, B \to \star\vec{x}\colon [B \wedge INV',\ INV' \wedge V \prec V_0]\ \mathbf{od})\ \ \square
\end{array}
$$

A simpler rule for iteration does not involve timing aspects.

Law 57 (iteration) *For the introduction of a loop one needs to supply a Boolean-valued guard expression, B, an invariant, INV, and a variant expression V, which evaluates to an element of a well-founded set with ordering $\prec$, provided INV and B hold. Given an environment, ρ, provided B and V are idle-stable expressions, and INV is idle-invariant, and none of B, V and INV contain references to τ_0 or zero-subscripted variables,*

$$\begin{array}{l} \quad \star\vec{x}\colon [\mathit{INV},\ \neg B \wedge \mathit{INV}] \\ \sqsubseteq_\rho \\ \quad \star\,\mathbf{do}\, B \to \star\vec{x}\colon [B \wedge \mathit{INV},\ \mathit{INV} \wedge V \prec V_0]\ \mathbf{od} \end{array}$$

where V_0 is V with every unindexed reference to a variable, v, replaced by v_0, and every occurrence of τ replaced by τ_0.

Proof. We let $\mathit{INV}' \mathrel{\widehat{=}} \mathit{INV} \wedge u \le \tau \wedge \mathit{stable}(\hat{\rho} \setminus \vec{x}, [\![u \ldots \tau]\!])$, where u is a fresh logical constant, and make use of Law 56 (iteration-timing).

$$\begin{array}{l} \quad \star\vec{x}\colon [\mathit{INV},\ \neg B \wedge \mathit{INV}] \\ \sqsubseteq_\rho \text{ Law 39 (logical-constant); Law 13 (separate-assumption)} \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [u = \tau \wedge \mathit{INV},\ \neg B \wedge \mathit{INV}]\ ]| \\ \sqsubseteq_\rho \text{ Law 7 (strengthen-effect)} \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star\vec{x}\colon [u = \tau \wedge \mathit{INV},\ \neg B \wedge \mathit{INV}']\ ]| \\ \sqsubseteq_\rho \text{ Law 56 (iteration-timing)} \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star\,\mathbf{do}\, B \to \star\vec{x}\colon [B \wedge \mathit{INV}',\ \mathit{INV}' \wedge V \prec V_0]\ \mathbf{od}\ ]| \\ \sqsubseteq_\rho \text{ Law 7 (strengthen-effect); Law 6 (weaken-assumption)} \\ \quad |[\ \mathbf{con}\, u : \mathit{Time} \bullet \star\,\mathbf{do}\, B \to \star\vec{x}\colon [B \wedge \mathit{INV},\ \mathit{INV} \wedge V \prec V_0]\ \mathbf{od}\ ]| \\ \sqsubseteq_\rho \text{ Law 40 (remove-logical-constant)} \\ \quad \star\,\mathbf{do}\, B \to \star\vec{x}\colon [B \wedge \mathit{INV},\ \mathit{INV} \wedge V \prec V_0]\ \mathbf{od}\ \ \square \end{array}$$

Example: Absolute delay. As an example of a real-time refinement using a loop we give an implementation of a delay until command in terms of an iteration involving the **gettime** command and a primitive delay command, **tick**, that delays for at least some fixed time $\epsilon > 0$.

$$\mathbf{tick} \mathrel{\widehat{=}}_\rho \star\,[\tau_0 + \epsilon \le \tau]$$

We begin from the specification of an absolute delay.

$$\begin{array}{l} \quad \star\,[D \le \tau] \\ \sqsubseteq_\rho \text{ Law 50 (introduce-variable); Law 7 (strengthen-effect)} \\ \quad |[\ \mathbf{var}\, c : \mathtt{time} \bullet \star c\colon [D \le c \le \tau]\ ]| \\ \sqsubseteq_\rho \text{ Law 42 (simple-sequential-composition)} \\ \quad |[\ \mathbf{var}\, c : \mathtt{time} \bullet \star c\colon [c \le \tau]\ ;\ \star\, c\colon [c \le \tau,\ D \le c \le \tau]\ ]| \\ \sqsubseteq_\rho \text{ Law 7 (strengthen-effect); Def. 35 (gettime)} \\ \quad |[\ \mathbf{var}\, c : \mathtt{time} \bullet c : \mathbf{gettime};\ \star\, c\colon [c \le \tau,\ D \le c \le \tau]\ ]| \end{array}$$

The remaining specification command refines to an iteration, with invariant, $c \leq \tau$, and variant, $\left\lfloor \frac{D-c}{\epsilon} \right\rfloor$. For this refinement we make use of the fact that the loop introduction rule only requires the variant to be a member of the well-founded set (in this case the natural numbers) provided both the invariant and the loop guard hold.

$$
\begin{array}{l}
\quad \star c\colon \left[c \leq \tau,\ \ D \leq c \leq \tau\right] \\
\sqsubseteq_\rho \text{ Law 57 (iteration)} \\
\quad \star\, \mathbf{do}\, c < D \rightarrow \star c\colon \left[c < D \wedge c \leq \tau,\ \ c \leq \tau \wedge \left\lfloor \frac{D-c}{\epsilon} \right\rfloor < \left\lfloor \frac{D-c_0}{\epsilon} \right\rfloor\right] \,\mathbf{od}
\end{array}
$$

The body of the loop can be refined as follows.

$$
\begin{array}{l}
\quad \star c\colon \left[c < D \wedge c \leq \tau,\ \ c \leq \tau \wedge \left\lfloor \frac{D-c}{\epsilon} \right\rfloor < \left\lfloor \frac{D-c_0}{\epsilon} \right\rfloor\right] \\
\sqsubseteq_\rho \text{ Law 41 (sequential-composition); Law 6 (weaken-assumption)} \\
\quad [\!| \,\mathbf{con}\, u : \mathit{Time} \bullet \star c\colon \left[c < D \wedge c \leq \tau,\ \ c_0 \leq \tau_0 \wedge \tau_0 + \epsilon \leq \tau\right]; \\
\qquad\quad \star c\colon \left[c(u) \leq u \wedge u + \epsilon \leq \tau,\ \ c \leq \tau \wedge \left\lfloor \frac{D-c}{\epsilon} \right\rfloor < \left\lfloor \frac{D-c(u)}{\epsilon} \right\rfloor\right] \,|\!] \\
\sqsubseteq_\rho \text{ Law 9 (remove-from-frame); Law 7 (strengthen-effect)} \\
\quad [\!| \,\mathbf{con}\, u : \mathit{Time} \bullet \star \left[c < D \wedge c \leq \tau,\ \ \tau_0 + \epsilon \leq \tau\right]; \\
\qquad\quad \star c\colon \left[c(u) \leq u \wedge u + \epsilon \leq \tau,\ \ c \in \left[\!\left[\tau_0 \mathinner{..} \tau\right]\!\right]\right] \,|\!] \\
\sqsubseteq_\rho \text{ Law 6 (weaken-assumption); Law 40 (remove-logical-constant)} \\
\quad \star \left[\tau_0 + \epsilon \leq \tau\right];\ \star c\colon \left[c \in \left[\!\left[\tau_0 \mathinner{..} \tau\right]\!\right]\right] \\
\sqsubseteq_\rho \text{ Definition of } \mathbf{tick}\text{; Def. 35 (gettime)} \\
\quad \mathbf{tick};\, c : \mathbf{gettime}
\end{array}
$$

The collected code for the absolute delay follows.

$$
\begin{array}{l}
[\!| \,\mathbf{var}\, c : \mathtt{time} \bullet c : \mathbf{gettime}; \\
\qquad \star\, \mathbf{do}\, c < D \rightarrow \mathbf{tick};\, c : \mathbf{gettime}\ \mathbf{od} \,|\!]
\end{array}
$$

The refinement of an absolute delay has been given here as a simple, but interesting, example of a real-time refinement involving a loop. In practice, the explicit use of the command **tick** is unnecessary because the time taken by the loop overhead is bounded below by some constant $\epsilon > 0$. One could allow for such overheads in Def. 55 (iteration) by replacing the first **idle** command by a **tick**, and developing a more complex refinement rule. This has been done for the development of a loop introduction rule that uses a fixed deadline to guarantee termination [8].

2.22 Procedures

A procedure call may involve delays due to the overheads of entering and exiting the procedure.

Definition 58 (procedure call) *Consider a procedure C defined via*

$$C \mathrel{\widehat=} \mathbf{procedure}\, S$$

where S is the specification of C. A call on the procedure, $\star\mathbf{call}\, C$*, is defined to be equivalent to executing some implementation of the procedure, but with delays before and after to allow for calling overheads.*

$$\mathbf{idle};\ S;\ \mathbf{idle} \sqsubseteq_\rho \star\mathbf{call}\, C$$

Law 59 (procedure call) *Given an environment,* ρ*, provided P is idle-invariant, and R is both pre- and post-idle-invariant, then, if a procedure C is defined via*

$$C \mathrel{\widehat=} \mathbf{procedure} \star \vec{x}\colon [P,\ R]$$

then

$$\star\vec{x}\colon [P,\ R] \sqsubseteq_\rho \star\mathbf{call}\, C$$

Proof.

$\star\vec{x}\colon [P,\ R]$
$\sqsupseteq_\rho$ Law 46 (idle-before); Law 48 (idle-after)
$\mathbf{idle};\ \star\vec{x}\colon [P,\ R]\ ;\ \mathbf{idle}$
$\sqsubseteq_\rho$ Def. 58 (procedure-call)
$\star\mathbf{call}\, C$ □

Parameters to procedures can be handled in the same way as in the standard calculus [17] (the timing analysis must account for any additional overhead).

3 The extended programming language

The extended programming language contains all the commands in the wide-spectrum language, except for the specification command. In addition, the extended language has the following restrictions:

- the variables in the frame of an assignment, a **read**, or a **gettime** command should be program variables, not external inputs;
- the variable being read by a **read** command should be an external input;
- expressions appearing in assignments, guards and delays should be programming language expressions, that is, they may only reference program variables (outputs and local variables) but may not reference external inputs or τ_0 or τ or logical constants or zero-subscripted variables;

– the expression in a deadline command should be a programming language expression, except that it may also reference logical constants (but recall that deadline directives must be removed by a timing analysis process before the program is finally code);
– the program may contain logical constants, provided the only references to logical constants are in assumptions and deadline commands.

The deadline command is quite different from the other primitive commands because, if the deadline has already passed when the command is reached, it is infeasible. In general, such a command cannot be considered part of the executable subset of the language. However, if one can show that the deadline command is always guaranteed to be reached before its deadline passes, then it can be eliminated. In standard refinement calculus terms the deadline command, $\mathbf{deadline}\, D$, is a coercion of the form $\lceil \tau \leq D \rceil$. The standard refinement calculus rule for removing an assumption-coercion pair can be applied to a deadline command.

$$\{\tau \leq D\}\,;\ \lceil \tau \leq D \rceil \sqsubseteq \mathbf{skip}$$

That allows a deadline command to be eliminated, provided the time at which it is reached is before its deadline, D.

For the purposes of real-time refinement we retain the deadline command as a primitive command in our extended programming language. We rely on a timing analysis being performed on the machine code generated for the program. If the analysis guarantees that all deadlines will be reached in time, the deadlines may be elided and the resulting program is executable, but if the timing analysis fails, the program is rejected. The failure of the analysis could either be because the program contains unmet deadlines, or because the analysis is not sophisticated enough to determine that all deadlines are met. In the latter case the program, although correct, is still rejected because it has not been shown to met all deadlines. Timing analysis is discussed in more detail in the example in Sect. 5.2.

In order to specify some deadlines it is advantageous to allow the deadline command to refer to logical constants. For example, the following specifies that the assignment, $x := 3$, takes at most 10 attoseconds to execute.

$$|[\, \mathbf{con}\, s : \mathit{Time} \bullet \star \{\tau = s\}\,;\, x := 3;\ \mathbf{deadline}\, s + 10\mathit{attoseconds}\,]| \quad (11)$$

Logical constants are not code, but they may be removed provided there are no references to them within the program. In the above example the timing analysis, if successful, will remove the deadline command and hence one reference to the logical constant. The other reference to the logical constant is in the assumption, but assumptions can always be removed. Hence a successful timing analysis of the above fragment allows all references to s to be removed, and hence s can be removed. In order to accommodate

fragments such as that above, we allow the extended language to include logical constants, provided the only references to logical constants are in assumptions and deadline commands.

In some cases the timing analysis fails because the code generated for the target machine takes too long and fails to meet the deadline. In such cases a faster machine or a better compiler or a combination of both may be able to meet the deadline. In other cases it may be impossible to generate code to meet the deadline for any machine. For example, the sequence

$$\textbf{delay until}\, d;\\ \textbf{deadline}\, d - 1\,\mathsf{s}$$

has a timing constraint of minus one second. It is impossible to meet this deadline on any machine. For our refinement theory we do not need to distinguish whether deadlines are impossible or not, but we hope that the vigilant programmer will not generate code with impossible deadlines. Of course, the analysis process can be applied to detect intrinsically impossible deadlines before the program is even compiled. That form of analysis can be applied to partially refined programs at any stage to check for impossible deadlines. Indeed, analysis of partially refined programs may be used at any stage to assist the engineer to determine the timing constraints on the partially refined code, and hence determine whether an implementation is likely to meet its deadlines. It may highlight the paths with the tightest constraints, where the most care needs to be taken.

4 Feasibility

The reader may be familiar with the notion of feasibility and Dijkstra's *law of the excluded miracle* from the standard refinement calculus [3]. The definition of feasibility for the standard refinement calculus is not appropriate for the real-time calculus. In this section we develop a definition of *real-time feasibility* and show its relationship to standard feasibility. This section may be skipped by those readers more interested in the application of the refinement calculus; the following sections do not rely on it.

An unusual feature of our semantics (Def. 4 (specification)) is that each command operates by updating *only* the special time variable τ. All other program variables are timed traces and are simply *constrained* by the command. This means that most of our commands are *infeasible* in the standard refinement calculus. For example,

$$\star v\colon \left[\tau \leq 9,\ \ \tau \leq 10 \wedge v(\tau) = 3\right]$$

is equivalent to the standard specification command

$$\tau\colon \left[\tau \leq 9,\ \begin{array}{l} \tau \leq 10 \wedge v(\tau) = 3 \\ \wedge\ \tau_0 \leq \tau \wedge \mathit{stable}(\hat{\rho} \setminus v, \llbracket \tau_0 \mathinner{..} \tau \rrbracket) \end{array}\right] \tag{12}$$

which is only feasible if the trace v already has the constant value 3 (at least up to time 10) in the initial state, because the trace v does not appear in the frame of the equivalent standard specification command, so the specification cannot update it. Yet according to our refinement laws, this specification can be implemented by the assignment $v := 3$, plus a deadline of one second, so intuitively the specification should be feasible.

It is quite common in the standard refinement calculus for valid developments to include some infeasible program fragments [16]. They typically appear in some context that avoids the infeasible part of their behaviour, so that the whole surrounding program remains feasible. However, according to the standard test (*Dijkstra's Law of the Excluded Miracle:* $S(\mathit{false}) = \mathit{false}$), virtually all commands are infeasible with our semantics. The standard test is inappropriate for our calculus for the following two reasons:

1. We designed the semantics to exclude program output and local traces from the frame. This means that most of our refinements are done in the 'magical' portion of the predicate transformer lattice, but it allows us to use total traces (avoiding undefinedness issues), and gives a more elegant semantics than alternative approaches, where each command *extends* or *overwrites* a segment of the traces [23].
2. Our real-time variables are modeled as traces over time, and most program fragments constrain only a small segment of the traces, typically in the range $\{t : \mathit{Time} \mid \tau_0 < t \leq \tau\}$, which we shall write as $(\!\!-\tau_0 \ldots \tau -\!\!]$. We want programs that constrain traces outside this range of times to be treated as infeasible. We view output traces as being observable (up to the current time, τ), from outside the system, so do not want programs to be able to change history! That is, any program that constrains an output trace v at time $t \leq \tau_0$ should be infeasible, because it is attempting to further constrain an output that may have already been observed. Similarly, programs that attempt to make time go backwards should be infeasible.

The standard feasibility test is too coarse to make these distinctions that we want for our real-time calculus. So we now propose a more suitable feasibility test.

For an environment, ρ, recall that $\hat{\rho}$ stands for the program variables (i.e., the outputs and local variables) of ρ. We would like to quantify over (partial) traces over the variables in ρ and $\hat{\rho}$. We use '$\forall \rho : \mathit{Time} \rightarrow T_\rho$' to stand for quantification over traces of all variables in ρ; T_ρ stands for the type of the variables in ρ. Similarly, '$\forall \hat{\rho} : (\!\!-\tau_0 \ldots \tau -\!\!] \rightarrow T_{\hat{\rho}}$' stands for the quantification over partial traces from τ_0 to τ of all program variables.

We define a test for identifying predicates that depend upon traces of program variables only up to time τ, not on the future portion of those traces [23].

Definition 60 (Future-Independent) *Given an environment, ρ, a predicate Q on the variables of ρ is* future independent *according to the following definition.*

$$\begin{array}{l} \mathit{FutureIndependent}_\rho(Q) \mathrel{\widehat=} \\ \quad (\forall\, \rho : \mathit{Time} \to T_\rho;\ \tau_0, \tau, \tau' : \mathit{Time} \bullet \tau_0 \le \tau \le \tau' \Rightarrow \\ \quad\quad (\forall\, \hat\rho' : (\!|\, \tau \mathinner{..} \tau' \,|\!) \to T_{\hat\rho} \bullet Q \Leftrightarrow Q \left[\frac{\hat\rho \oplus \hat\rho'}{\hat\rho}\right]))) \end{array}$$

where $\hat\rho \oplus \hat\rho'$ stands for the trace $\hat\rho$ overridden by the partial trace $\hat\rho'$ in the interval for which $\hat\rho'$ is defined, i.e., $(\!|\, \tau \mathinner{..} \tau' \,|\!)$.

All of our real-time commands are defined in terms of conjunctive predicate transformers with frame τ. From relational-decomposition results in the standard refinement calculus [13] we know that every conjunctive command C (in an environment ρ) is equivalent to a specification command

$$\tau\colon [\mathit{assump}_\rho(C),\ \ \mathit{effect}_\rho(C)]$$

where

$$\begin{array}{l} \mathit{assump}_\rho(C) \mathrel{\widehat=} \mathcal{M}_\rho\,(C)\,(\mathit{true}) \\ \mathit{effect}_\rho(C) \mathrel{\widehat=} (\neg\, \mathcal{M}_\rho\,(C)\,(\tau \neq \tau'))\left[\frac{\tau, \tau_0}{\tau', \tau}\right] \end{array}$$

The assumption of the specification command is just the condition under which it is guaranteed to terminate. In general, for a predicate transformer, PT, $\neg\, PT(\neg\, R)$, characterises those states from which PT is not guaranteed to achieve $\neg\, R$, that is, it characterises those states from which it is possible for PT to achieve R. The definition of $\mathit{effect}_\rho(C)$, characterises those states from which there exists a time τ' such that it is possible for C to terminate with τ equal to τ'. The renaming is used to convert the predicate in terms of τ for the start time and τ' for the finish time, into a predicate with τ_0 for the start time and τ for the finish time. See [13] for more details.

Hence for a conjunctive command, C,

$$\mathcal{M}_\rho\,(C)\,(R) \equiv \mathit{assump}_\rho(C) \wedge (\forall\, \tau : \mathit{Time} \bullet \mathit{effect}_\rho(C) \Rightarrow R)\left[\frac{\tau}{\tau_0}\right]$$

For example, in the case of a real-time specification command, $\star\vec{v}\colon [A,\ \ E]$, the $\mathit{assump}_\rho(\ldots)$ and $\mathit{effect}_\rho(\ldots)$ terms can be derived from A and E.

Theorem 1 *For any specification command, $\star\vec{v}\colon [A,\ \ E]$, in an environment ρ, we have:*

$$\begin{array}{l} \mathit{assump}_\rho(\star\vec{v}\colon [A,\ \ E]) \equiv A \mathbin{@} \tau \\ \mathit{effect}_\rho(\star\vec{v}\colon [A,\ \ E]) \equiv \\ \quad A \mathbin{@} \tau_0 \Rightarrow E \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge \mathit{stable}(\hat\rho \setminus \vec{v}, (\!|\, \tau_0 \mathinner{..} \tau \,|\!)) \end{array}$$

Proof. The theorem follows from Def. 4 (specification) plus the definitions of $assump_\rho(C)$ and $effect_\rho(C)$ above. □

The assumption of a real-time feasible command should depend on the value of the program variables only up to the start time of the command, and the effect of a command should depend on the value of the program variables only up to the completion time of the command. In addition, for a command to be *real-time feasible*, for any trace, $\hat{\rho}$, of program variables that satisfies the assumption of the command, there must exist a partial trace, $\hat{\rho}'$ of the program variables over the execution interval of the command, such that the effect of the command holds for the trace $\hat{\rho}$ updated over the interval $\left(\!\!\left[\tau_0 \mathinner{..} \tau \right]\!\!\right]$ with the partial trace $\hat{\rho}'$, i.e., the effect should hold for $\hat{\rho} \oplus \hat{\rho}'$.

Definition 61 (RTFeasible) *Given an environment ρ, and a conjunctive command, C,*

$$
\begin{array}{l}
RTFeasible_\rho(C) \mathrel{\widehat{=}} \\
\quad FutureIndependent_\rho(assump_\rho(C)) \wedge \\
\quad FutureIndependent_\rho(effect_\rho(C)) \wedge \\
\quad (\forall \tau_0 : Time;\ \rho : Time \to T_\rho \bullet \\
\qquad assump_\rho(C)\left[\frac{\tau_0}{\tau}\right] \Rightarrow \\
\qquad\quad (\exists \tau : Time;\ \hat{\rho}' : \left(\!\!\left[\tau_0 \mathinner{..} \tau \right]\!\!\right) \to T_{\hat{\rho}} \bullet effect_\rho(C)\left[\frac{\hat{\rho} \oplus \hat{\rho}'}{\hat{\rho}}\right]))
\end{array}
$$

Applying *RTFeasible* to example (12) above (with $\rho.in = \{\}$, $\hat{\rho} = \{v\}$), we see that the assumption and effect are both future-independent, so we get:

$$
\begin{array}{l}
RTFeasible_\rho(\star v{:}\ [\tau \le 9,\ \ \tau \le 10 \wedge v(\tau) = 3]) \\
\equiv (\forall \tau_0 : Time;\ v : Time \to \mathbb{N} \bullet \tau_0 \le 9 \Rightarrow \\
\qquad (\exists \tau : Time;\ v' : \left(\!\!\left[\tau_0 \mathinner{..} \tau \right]\!\!\right) \to \mathbb{N} \bullet \\
\qquad\quad (\tau \le 10 \wedge v(\tau) = 3 \wedge \tau_0 \le \tau)\left[\frac{v \oplus v'}{v}\right])) \\
\equiv true
\end{array}
$$

It is interesting to look at how *RTFeasible* judges other simple specifications. Assume that the environment ρ has a single input variable *e* and two output variables, *x* and *y*, and that the predicate *E* is future independent.

1. $RTFeasible_\rho(\star x{:}\ [false,\ E]) = true$
2. $RTFeasible_\rho(\star x{:}\ [true,\ true]) = true$
3. $RTFeasible_\rho(\star x{:}\ [true,\ false]) = false$
4. $RTFeasible_\rho(\star x{:}\ [true,\ \tau = \tau_0 - 1]) = false$
5. $RTFeasible_\rho(\star x{:}\ [true,\ \tau < 9]) = false$
6. $RTFeasible_\rho(\star x{:}\ [\tau < 8,\ \tau < 9]) = true$
7. $RTFeasible_\rho(\star x{:}\ [true,\ y(\tau) = 2]) = false$
8. $RTFeasible_\rho(\star x{:}\ [true,\ x(\tau) = 2]) = true\ (*)$
9. $RTFeasible_\rho(\star x{:}\ [true,\ x(\tau) = y(\tau)]) = true\ (*)$
10. $RTFeasible_\rho(\star x{:}\ [\tau < 8,\ \tau < 9 \wedge x(\tau) = e(10)]) = true\ (*)$
11. $RTFeasible_\rho(\star x{:}\ [\tau < 8,\ \tau \leq 9 \wedge x(\tau) = e(9)]) = true\ (*)$
12. $RTFeasible_\rho(\star\ [true,\ \tau_0 = \tau \leq 9] = false$

The starred lines show where *RTFeasible* differs from standard feasibility. Note that in all the examples, y is not in the frame, so is required to be stable during $[\tau_0 \ldots \tau]$. Specifications 8, 9 and 10 are feasible because x is in the frame, so the $\exists$ in the definition of *RTFeasible* allows the x trace to be updated during times $(\tau_0 \ldots \tau]$. Specification 8 could be refined to $x := 2$ and 9 could be refined to $x := y$ (no deadlines are required, since the specifications do not give any upper bound for the finish time, τ).

It is interesting to note that specification 10 passes the feasibility test even though it refers to the input e at time 10, which is in the future, since it terminates with $\tau < 9$. This illustrates that our feasibility test allows some non-causal specifications (where information may need to flow backwards in time) to be treated as feasible, even though it may not be possible to generate code for such specifications. In our experience, specifications often refer to future input values to express assumptions about how those inputs change over time, and there is no simple test that distinguishes such innocuous future references from those that imply non-causal behaviour.

Although example 11 passes the feasibility test, in practice it cannot be refined to our target language. One reason for this is that our target language does not provide any command for reading an input value at an exact time (only sometime between τ_0 and τ) – such a command would be impossible to implement without knowing more about how the value of e varies over time. Even if there were such a command, specification 11 requires the read of e to be done at time τ, which would leave no time to store the value into x (this problem would be detected when the code is submitted to the timing analysis phase of our methodology).

This again illustrates that our feasibility test provides only an approximation to practical feasibility. Any program that is rejected by our feasibility test is definitely not implementable. However, some programs that pass the feasibility test may still not be implementable, because the specification language can express relationships and properties (like references to future

input values, or other non-causal effects) that are not expressible in the target programming language or because timing requirements are too tight. This is as expected, we want specification languages to be more expressive than implementation languages and the analysis of code timings is deferred to a later stage. The ultimate assurance of feasibility comes only when a specification has been refined to code and all the timing requirements of that code have been checked.

In the final example, 12, the command is equivalent to $\mathbf{deadline}\,9$. As expected, this fails the real-time feasibility test, but that does not mean that programs that contain deadline commands are infeasible. The extended programming language allows deadline commands, but in order to get to code in the target programming language, all deadlines must be removed by a timing analysis phase that shows that the generated code will reach each deadline command before its stated deadline.

The next theorem relates $\mathit{RTFeasible}_\rho(C)$ to the standard feasibility of an entire process. Any top-level process can be written as a conjunctive command, C, within a block that declares the inputs and outputs for the process: $|[\ \mathbf{input}\, x : S;\ \mathbf{output}\, v : T \bullet C\ ]|$. If the body of a process satisfies *RTFeasible*, then the whole process is feasible in the traditional sense. Intuitively, this result validates our definition of *RTFeasible*.

Theorem 2 (Process-Feasibility) *Given a conjunctive command, C, if C is real-time feasible then the top-level process containing just C, with all the inputs and outputs in C's environment explicitly declared within the process, is (standard) feasible.*

$$\mathit{RTFeasible}_\rho(C) \Rrightarrow \mathcal{M}_\phi\,(|[\ \mathbf{input}\, x : S;\ \mathbf{output}\, v : T \bullet C\ ]|)\,(\mathit{false}) = \mathit{false}$$

where ϕ is the empty environment, and $\rho = \phi +_i x +_o v$.

Proof. Assume $\mathit{RTFeasible}_\rho(C)$.

$\mathcal{M}_\phi\,(|[\ \mathbf{input}\, x : S;\ \mathbf{output}\, v : T \bullet C\ ]|)\,(\mathit{false})$

$\equiv$ Def. 52 (input); Def. 51 (output)

$x \in \mathit{Time} \to S \land (\forall v : \mathit{Time} \to T \bullet \mathcal{M}_\rho\,(C)\,(\mathit{false}))$

$\Rrightarrow$ drop first conjunct

$(\forall v : \mathit{Time} \to T \bullet \mathcal{M}_\rho\,(C)\,(\mathit{false}))$

$\equiv$ Since C is conjunctive

$(\forall v : \mathit{Time} \to T \bullet \mathit{assump}_\rho(C) \land (\forall \tau : \mathit{Time} \bullet \neg\, \mathit{effect}_\rho(C))\left[\frac{\tau}{\tau_0}\right])$

$\equiv$ distributing $\forall$ over $\land$

$(\forall v : \mathit{Time} \to T \bullet \mathit{assump}_\rho(C)) \land$

$$(\forall v : \mathit{Time} \rightarrow T \bullet (\forall \tau : \mathit{Time} \bullet \neg\, \mathit{effect}_\rho(C)) \left[\frac{\tau}{\tau_0}\right]$$

$\Rrightarrow$ Rewriting $\mathit{assump}_\rho(C)$ using definition of $\mathit{RTFeasible}_\rho(C)$

$$(\forall v : \mathit{Time} \rightarrow T \bullet$$
$$(\exists \tau : \mathit{Time}; v' : (\!|\, \tau_0 \mathinner{..} \tau \,|\!] \rightarrow T \bullet \mathit{effect}_\rho(C) \left[\frac{v \oplus v'}{v}\right]) \left[\frac{\tau}{\tau_0}\right]) \wedge$$
$$(\forall v : \mathit{Time} \rightarrow T \bullet (\forall \tau : \mathit{Time} \bullet \neg\, \mathit{effect}_\rho(C)) \left[\frac{\tau}{\tau_0}\right])$$

$\Rrightarrow$ Using last conjunct (with v being $v \oplus v'$) to rewrite $\mathit{effect}_\rho(C) \left[\frac{v \oplus v'}{v}\right]$.

$$(\forall v : \mathit{Time} \rightarrow T \bullet (\exists \tau : \mathit{Time}; v' : (\!|\, \tau_0 \mathinner{..} \tau \,|\!] \rightarrow T \bullet \mathit{false}))$$

$\Rrightarrow \mathit{false}$ □

In conclusion, the definition of *RTFeasible* provides a useful check that can be applied to detect most infeasible specifications, even though it is more complex than the standard feasibility test. Fortunately, it is not essential to check real-time feasibility when developing a program by refinement. If the refinement process ends with code, then because code, with the notable exception of the deadline command, is real-time feasible, there is no need to check feasibility except for the deadline commands. Feasibility of deadline commands comes down to ensuring that the deadline command is reached at a time before its deadline. To check this we need to check the actual execution time of the code leading up to the deadline via a timing analysis process. The role of the deadline command is discussed in more detail in Sections 3 and 5.2.

5 An example specification and refinement

In this section we illustrate the use of sequential real-time refinement on the simple example of a message receiver.

5.1 Specification of a message receiver

The environment. Before giving the specification of the operation to be performed, we need to set up the environment of the receiver. The receiver takes input from an input buffer register, *in*, that is assumed to hold the characters of the message over time. The message is to be assembled and placed in the local variable *msg*, which is a sequence of characters with indices starting from one.

$$\mathbf{input}\ \mathit{in} : \mathtt{char};\ \mathbf{var}\ \mathit{msg} : \mathrm{seq}\,\mathtt{char}$$

Each character of the message is available in the input buffer for only 80 microseconds and the characters appear in the input buffer at intervals of 100 microseconds.

$$\mathbf{const}\ chsep = 100\,\mu s;\ chdef = 80\,\mu s$$

The receiver begins execution at time *start*, which corresponds to the time at which a start-of-text character was identified in the input stream. The characters of the message follow the start-of-text character at regular intervals of duration *chsep*.

$$\mathbf{con}\ start : Time$$

The *i*th character of the message is therefore available in the input buffer during the interval from $start + chsep * i$ until $start + chsep * i + chdef$.

$$\mathbf{let}\ interval \mathrel{\widehat=} (\lambda\, i : \mathtt{nat} \bullet [\![start + chsep * i \mathinner{..} start + chsep * i + chdef]\!]) \tag{13}$$

The **let** construct introduces a syntactic abbreviation.

We would like to state that the input buffer is stable over every interval up to the first interval that contains an end-of-text (*ETX*) character. However, in general there may be more than one character in the input buffer during an interval. Hence we first introduce the relation *chin*, that relates each index, *i*, to every character in the input buffer during the *i*th interval. (Later we will see that *chin* is a function up to the first end-of-text character.)

$$\mathbf{let}\ chin \mathrel{\widehat=} \{i : \mathtt{nat}; ch : \mathtt{char} \mid ch \in in(\!|interval(i)|\!) \bullet i \mapsto ch\} \tag{14}$$

The receiver assumes that an end-of-text (*ETX*) character appears in the input stream.

$$\star\{\exists\, i : \mathtt{nat} \bullet i \neq 0 \wedge ETX \in in(\!|interval(i)|\!)\} \tag{15}$$

This assumption does not depend on τ. We state it once as an assumption, rather than including it explicitly in the assumption of the specification (i) below and numerous specification commands within the refinement. Because the assumption is frame-invariant, it can be distributed through the refinement using Law 26 (frame-invariant-assumption), but we do not bother to show these steps explicitly.

The message to be received consists of all the characters in the input stream up until the first end-of-text character. We introduce *mx* to stand for that position.

$$\mathbf{let}\ mx \mathrel{\widehat=} min\{i : \mathtt{nat} \mid i \neq 0 \wedge ETX \in in(\!|interval(i)|\!)\} \tag{16}$$

The receiver also assumes that the input buffer is stable (with the appropriate character of the message) for each character time up until the first end-of-text character.

$$\star \{\forall i : 1 \mathinner{..} mx \bullet stable(in, interval(i))\} \tag{17}$$

From the above properties we can deduce that the stream of input characters up to the first end-of-text character is uniquely defined, that is, the relation *chin*, with its domain restricted to indices in the range 1 through to *mx*, inclusive, relates each such index to a single character, and hence it is a sequence of characters.

$$((1 \mathinner{..} mx) \lhd chin) \in \mathrm{seq}\, \mathtt{char} \tag{18}$$

The binary operator '$\lhd$' takes a set and a relation and returns the relation with its domain restricted to the elements in the set.

The specification. Given the above environment, the receiver should extract the characters from the input stream up to, but not including, the first end-of-text character, and place them in the variable *msg*. The receiver process is assumed to start at time *start*, and is required to complete within one character time of the first end-of-text character.

$$\star msg\colon \left[start = \tau, \quad \begin{array}{l} msg = (1 \mathinner{..} mx - 1) \lhd chin \wedge \\ \tau \leq start + chsep * (mx + 1) \end{array} \right] \tag{i}$$

5.2 Final program and timing analysis

In order to give the reader an idea of the way in which the delay and deadline commands are used in practice, we present the final code for the receiver (Fig. 1) before detailing the refinement of the receiver specification. The receiver program determines an (upper bound) approximation to the start time in the variable *st*, and then proceeds to read all the characters of the message until an end-of-text character is found. The program in Fig. 1 is not the most efficient implementation of the specification. Its form was chosen to make the relationship between the original specification and the refined program as simple as possible.

The final program includes deadline commands. These cannot be compiled to machine code, and an analysis of the program is required to determine whether the deadlines are met. The analysis is performed in two phases. Firstly, paths of the program that end at a deadline command are analysed to determine the timing constraint on each path. In the second phase the machine code generated for each program path is analysed to determine whether or not it meets the constraint on the path. In this section we concentrate on the first phase of determining the paths and their associated timing constraints.

$$
\begin{array}{l}
A :: \star\{start = \tau\}; \\
\quad |[\ \mathbf{var}\ st : \mathtt{time} \bullet \\
\qquad B :: st : \mathbf{gettime}; \\
\qquad C :: \star\{start \le st\}; \\
\qquad\quad |[\ \mathbf{var}\ n : \mathtt{nat}; ch : \mathtt{char} \bullet \\
\qquad\qquad \mathbf{let}\ INV \mathrel{\widehat{=}} \left(\begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \le n \le mx \wedge start \le st \wedge ch = chin(n) \end{array}\right) \bullet \\
\qquad\qquad n, msg := 1, \langle\rangle; \\
\qquad\quad D :: \mathbf{delay\ until}\ st + chsep * n; \\
\qquad\qquad ch : \mathbf{read}(in); \\
\qquad\quad E :: \mathbf{deadline}\ start + chsep * n + chdef; \\
\qquad\qquad \star\,\mathbf{do}\ ch \neq ETX \rightarrow \\
\qquad\qquad\qquad n, msg := n+1, msg \frown \langle ch \rangle; \\
\qquad\qquad\quad F :: \mathbf{delay\ until}\ st + chsep * n; \\
\qquad\qquad\qquad ch : \mathbf{read}(in); \\
\qquad\qquad\quad G :: \mathbf{deadline}\ start + chsep * n + chdef \\
\qquad\qquad \mathbf{od}; \\
\qquad\qquad \star\{n = mx\} \\
\qquad\quad]| \\
\quad]|; \\
H :: \mathbf{deadline}\ start + chsep * (mx + 1)
\end{array}
$$

Fig. 1 The receiver program (without procedure *readchar*)

Determining timing path constraints. To make life simpler in the analysis, we require that delay commands are reached before their specified delay times. That allows delays to be used as the starting point of paths. This is a stricter requirement than necessary and, although it simplifies the analysis, it may lead to a valid program failing the analysis.

The first path we consider is the path from A to D in Fig. 1. It consists of the allocation of the local variable *st*, reading the current time into *st*, allocating *n* and *ch*, and assigning initial values to *n* and *msg*. The starting time of the path is *start*, and the path should finish before $st + chsep * n$, if the delay is to be reached before its specified delay time. In the path *n* is assigned the value 1. Hence the deadline is actually $st + chsep$. The time available to execute the path is the difference between the starting and finishing times: $st + chsep - start$. This expression contains the variable *st*, which contains the retrieved clock time. The minimum, and hence most pessimistic, value for the path time is when *st* is minimal. The smallest feasible value of *st*

is *start* plus the minimum execution time for the path *A*–*B*. Hence we can express the time constraint for the path *A*–*D* as

$$start + min(A\text{–}B) + chsep - start = min(A\text{–}B) + chsep$$

Aside: A better bound on the minimum value of *st* can be determined at the level of machine code, where the internals of the implementation of **gettime** are available. We do not consider such refinements of the path analysis techniques in this paper.

Having determined the constraint on path *A*–*D* in order to ensure that *D* is reached before its delay time, we may assume *D* is reached before its delay time for the analysis of the remainder of the program. The next path we consider is from *D* to *E*. It includes the lateness of the delay command (*D*), and the read command. Execution must reach *E* before $start + chsep * n + chdef$. Hence the time constraint for this path is

$$start + chsep * n + chdef - (st + chsep * n) = start + chdef - st$$

Again this contains the variable *st*, but this time the constraint is minimal when *st* is maximal. An upper bound on the value of *st* is *start* plus the maximum execution time of the path *A*–*C*. Hence the time constraint on the path *D*–*E* is

$$start + chdef - (start + max(A\text{–}C)) = chdef - max(A\text{–}C)$$

For our analysis we require the delay command at *F* to be reached before its delay time. Hence we consider the path from *D* to *F*, which consists of the lateness of the delay at *D*, the read command, evaluating the loop guard (to true), and updating the values of *n* and *msg*. The starting time for the path is $st + chsep * n$. The finishing time is also $st + chsep * n$, but along the path *n* has been incremented. To allow for the increment we express the finishing time in terms of the values of the variables at the start of the path; that gives the finishing time as $st + chsep * (n + 1)$. Hence the time constraint on the path is

$$st + chsep * (n + 1) - (st + chsep * n) = chsep$$

The path from *F* to *G* has the same time constraint as the path *D*–*E*.

The next path we consider is from the delay at *F*, through the remainder of the body of the loop, looping back to the start of the loop, and then back down to *F*. Its starting and finishing times are both $st + chsep * n$, but along the path *n* has been incremented. Hence the timing constraint for this path is the same as for path *D*–*F*.

The next path we consider begins at *F*, executes the remainder of the body of the loop, loops back to the start of the loop and evaluates the guard (to false)

and exits the loop; it then exits the two local variable blocks, deallocating the associated variables, before reaching *H*. The path has a starting time of $st + chsep * n$ and a finishing time of $start + chsep * (mx + 1)$. That gives a constraint of $start + chsep * (mx + 1) - (st + chsep * n)$. On exit from the loop we know that $n = mx$ from the assumption just after the loop. That assumption was generated as part of the refinement (Sect. 5.3) in order to allow the timing analysis access to that information. The assumption, $n = mx$, allows the constraint to be simplified to

$$start + chsep * (n + 1) - (st + chsep * n) = start + chsep - st$$

As with the path *D*–*E*, this is minimal when *st* is maximal. This gives the time constraint

$$start + chsep - (start + max(A\text{–}C)) = chsep - max(A\text{–}C)$$

The final path we need to consider is from *D* to *H*. It corresponds to the case when the loop guard is false on its first evaluation, and the body of the loop is never entered. The constraint on the path is

$$start + chsep * (mx + 1) - (st + chsep * n)$$

This is the same constraint as for the path *F*–*H*.

Timing analysis of machine code paths. Having determined constraints on every timing path in the program, the final phase is to analyse the machine code generated by the compiler for each path to determine whether it meets its time constraint. In order to perform the analysis, the timing points in the high-level language program need to be mapped by the compiler to points in the generated machine code. In addition, any optimisations performed by the compiler need to take into account the position of timing points; code involving accesses or updates to external inputs or outputs may not be moved across a timing point. These issues are discussed in more detail in [10], and timing analysis of machine code sequences is discussed in [11].

Timing path constraints and refinement. Note that determining the timing constraints on paths is, in general, a non-trivial process. Finding all the deadline directives and extracting their deadline times is straightforward, but determining a suitable corresponding starting point for the path requires some intelligence. In addition, simplifying the timing constraint on a path may require the use of non-trivial properties of the program. These properties need to be passed from the refinement process to the timing analysis process via assumptions in the program code. For example, in the receiver program the property that $n = mx$ on termination of the loop is required to simplify the timing constraints on paths exiting the loop. The assumptions on a path give

a collection of constraints that can be used to simplify its timing constraint. Timing constraint determination can be automated provided the programs being analysed are of a sufficiently simple form. See [5] for a more detailed discussion of timing constraint analysis and [4] for preliminary work on automation of constraint determination.

The integration of refinement and timing analysis into the one development process makes it possible to use higher-level program properties to assist in the timing analysis, and hence simplify constraints that could not be resolved using just the program code. That may require modification of the refinement of a program in order to include assumptions that enable timing path constraints to be simplified.

In the receiver example all the timing constraints can be simplified to constant values. This is because we have taken care to generate a program with sensible timing constraints. The refinement process does not preclude the generation of non-constant timing constraints, such as data dependent time constraints. However, it is possible for a refinement to generate time constraints that are so complex that an automated timing analysis process cannot resolve them. In such cases, the timing analysis process should fail, and identify the reason for failure.

In the receiver program, the analysis of timing constraints has been simplified by the fact that there is a timing point inside the loop, which effectively cuts the cycle in the loop. Without a cut point in a loop, the whole of the loop becomes part of a path. One would not expect the timing constraint on a path containing a loop to be constant. Instead it would be an expression involving a variable that corresponds to the number of times the loop is executed. In such a case, determining the timing constraint expression is no different to a path that does not involve a loop, but the analysis of the time taken by the machine code for the loop also needs to take into account the number of times the loop is executed.

5.3 Refinement

In the refinement we do not generate exactly the code given in Fig. 1, instead we introduce a procedure, *readchar*, which corresponds to the task of reading a single character. The code including the procedure is given in Fig. 2. We return to timing path analysis with procedures in Sect. 5.5.

For the refinement we assume the environment described in Sect. 5.1. The refinement begins from the specification (i) from Sect. 5.1, which we repeat here.

$$\star msg\colon \left[start = \tau,\ \begin{array}{l} msg = (1\,..\,mx-1) \lhd chin \wedge \\ \tau \leq start + chsep * (mx+1) \end{array} \right] \qquad \text{(i)}$$

$$
\begin{array}{l}
\mathit{readchar} \mathrel{\widehat{=}} \mathbf{procedure} \star \mathit{ch}\colon \left[\mathit{start} \le \mathit{st},\ \ \mathit{ch} = \mathit{chin}(n)\right] \\
\qquad \sqsubseteq_\rho \ [\!| \quad I :: \mathbf{delay\ until}\, \mathit{st} + \mathit{chsep} * n; \\
\qquad\qquad\qquad\quad \mathit{ch} : \mathbf{read}(\mathit{in}); \\
\qquad\qquad\qquad\quad J :: \mathbf{deadline}\, \mathit{start} + \mathit{chsep} * n + \mathit{chdef} \\
\qquad\qquad\quad |\!] \\
K :: \star\left\{\mathit{start} = \tau\right\}; \\
\quad [\!| \, \mathbf{var}\, \mathit{st} : \mathtt{time} \bullet \\
\qquad L :: \mathit{st} : \mathbf{gettime}; \\
\qquad M :: \star\left\{\mathit{start} \le \mathit{st}\right\}; \\
\qquad\quad [\!| \, \mathbf{var}\, n : \mathtt{nat}; \mathit{ch} : \mathtt{char} \bullet \\
\qquad\qquad \mathbf{let}\ \mathit{INV} \mathrel{\widehat{=}} \left(\begin{array}{l} \mathit{msg} = (1\,..\,n-1) \lhd \mathit{chin} \wedge \\ 1 \le n \le \mathit{mx} \wedge \mathit{start} \le \mathit{st} \wedge \mathit{ch} = \mathit{chin}(n) \end{array}\right) \bullet \\
\qquad\qquad n, \mathit{msg} := 1, \langle\rangle; \\
\qquad\quad N :: \star\mathbf{call}\, \mathit{readchar}; \\
\qquad\qquad \star\, \mathbf{do}\, \mathit{ch} \ne \mathit{ETX} \to \\
\qquad\qquad\qquad n, \mathit{msg} := n + 1, \mathit{msg} \frown \langle \mathit{ch} \rangle; \\
\qquad\qquad\quad O :: \star\mathbf{call}\, \mathit{readchar} \\
\qquad\qquad \mathbf{od}\ ; \\
\qquad\qquad \star\left\{n = \mathit{mx}\right\} \\
\qquad\quad |\!] \\
\quad |\!]; \\
P :: \mathbf{deadline}\, \mathit{start} + \mathit{chsep} * (\mathit{mx} + 1)
\end{array}
$$

Fig. 2 Message receiver program with procedure

Separate out the finishing deadline. The second conjunct of the effect of the specification is clearly a time deadline. This can be separated out.

$$
\begin{array}{ll}
& \text{(i)} \\
\sqsubseteq_\rho & \text{Law 43 (separate-deadline)} \\
& \star\mathit{msg}\colon \left[\mathit{start} = \tau,\ \ \mathit{msg} = (1\,..\,\mathit{mx}-1) \lhd \mathit{chin}\right]; \qquad \text{(ii)} \\
& \mathbf{deadline}\, \mathit{start} + \mathit{chsep} * (\mathit{mx} + 1)
\end{array}
$$

At this stage of the refinement, one can perform a trivial timing analysis of the partially refined program. It gives a timing constraint of $\mathit{chsep} * (\mathit{mx}+1)$ for the whole receiver process. As this is linear in the number of characters to be read, it does not raise any alarms.

Capture the starting time. The starting time of the program is given by the logical constant *start*. Although deadline directives may make use of logical

constants, other commands in the final program code, such as delays, may not reference logical constants. The next few refinement steps are done with the foresight that the variable *st* is required as an (upper bound) approximation to the value of *start*. (The obvious initial refinement is to use *start* instead of *st*, but then to get code one needs to then eliminate all references to *start*.) The application of Law 50 (introduce-variable) relies on the fact that the predicate, $start \leq \tau$, is idle-invariant, and the predicate, $msg = (1 \,..\, mx - 1) \lhd chin$, is both pre- and post-idle-invariant.

(ii)

$\sqsubseteq_\rho$ Law 13 (separate-assumption); Law 6 (weaken-assumption)

$\star \{start = \tau\}$; $\star\, msg{:}\, [start \leq \tau,\ \ msg = (1 \,..\, mx - 1) \lhd chin]$

$\sqsubseteq_\rho$ Law 50 (introduce-variable) for *st*

$\star \{start = \tau\}$;

$[\![\ \mathbf{var}\, st : \mathtt{time} \bullet$

$\quad \star st, msg{:}\, [start \leq \tau,\ \ msg = (1 \,..\, mx - 1) \lhd chin]$ (iii)

$]\!]$

We separate out the capture of the start time via a sequential composition.

(iii)

$\sqsubseteq_\rho$ Law 42 (simple-sequential-composition)

$\star st, msg{:}\, [start \leq \tau,\ \ start \leq st]$; (iv)

$\star st, msg{:}\, [start \leq st,\ \ msg = (1 \,..\, mx - 1) \lhd chin]$ (v)

Reading the current time gives an upper bound on the start time.

(iv)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on *msg*; Law 7 (strengthen-effect)

$\star st{:}\, [start \leq \tau,\ \ st \in [\!-\tau_0 \,...\, \tau-\!]]$

$\sqsubseteq_\rho$ Law 6 (weaken-assumption); Def. 35 (gettime)

$st : \mathbf{gettime}$

Set up for loop. To read in the characters of the message, a loop that sequences through the characters is required. Local variables *ch* and *n* are introduced to keep track of the current character and index in the message. The following makes use of the fact that the predicate, $start \leq \tau$, is idle-invariant, and the predicate, $msg = (1 \,..\, mx - 1) \lhd chin$, is both pre- and post-idle-invariant, for the application of Law 50 (introduce-variable).

(v)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on *st*; Law 50 (introduce-variable)

$$|[\ \mathbf{var}\ ch : \texttt{char};\ n : \texttt{nat} \bullet$$
$$\star ch, n, msg: [start \leq st,\ \ msg = (1\,..\,mx-1) \lhd chin] \qquad \text{(vi)}$$
$$]|$$

The loop invariant indicates that the characters up to position $n-1$ have been placed in the message and that *ch* is the next (*n*th) character in the input stream. In addition, *n* remains within the range 1 to *mx*, and *st* is an (upper bound) approximation to *start*. A sequential composition corresponding to the loop initialisation and the loop itself is introduced.

(vi)

$\sqsubseteq_\rho$ Law 7 (strengthen-effect)

$$\mathbf{let}\ INV \mathrel{\widehat=} \begin{pmatrix} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \leq n \leq mx \wedge start \leq st \wedge ch = chin(n) \end{pmatrix} \bullet$$
$$\star ch, n, msg: [start \leq st,\ \ INV \wedge n = mx]$$

$\sqsubseteq_\rho$ Law 42 (simple-sequential-composition)

$$\star ch, n, msg: [start \leq st,\ \ INV]\ ; \qquad \text{(vii)}$$
$$\star ch, n, msg: [INV,\ \ INV \wedge n = mx] \qquad \text{(viii)}$$

Initialisation. The initialisation for the loop is performed via a sequential composition, which first establishes the first three conjuncts of *INV*, and then establishes the last conjunct.

(vii)

$\sqsubseteq_\rho$ Law 42 (simple-sequential-composition)

$$\star ch, n, msg: \left[start \leq st,\ \begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \leq n \leq mx \wedge start \leq st \end{array}\right]; \qquad \text{(ix)}$$
$$\star ch, n, msg: \left[\begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \leq n \leq mx \wedge start \leq st \end{array},\ \ INV\right] \qquad \text{(x)}$$

The initialisation of *n* and *msg* establishes all but one requirement of the loop invariant.

(ix)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on *ch*; Law 33 (assignment)

$$n, msg := 1, \langle\rangle$$

Read one character. The final requirement of the loop invariant is to read the first character into *ch*.

(x)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on *n* and *msg*

$$\star ch\colon \left[\begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \le n \le mx \wedge start \le st \end{array},\ \begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \le n \le mx \wedge start \le st \wedge \\ ch = chin(n) \end{array}\right]$$

$\sqsubseteq_\rho$ Law 24 (frame-invariant); Law 6 (weaken-assumption)

$$\star ch\colon \left[start \le st,\ \ ch = chin(n)\right] \qquad \text{(xi)}$$

At this point rather than perform the refinement of the above specification command, we introduce a procedure corresponding to the command. The introduction of this procedure is done with the foresight that we need it again later in the refinement.

$$readchar \mathrel{\widehat{=}} \mathbf{procedure} \star ch\colon \left[start \le st,\ \ ch = chin(n)\right] \qquad \text{(xii)}$$

Now the specification command can be implemented by a call on the procedure. The introduction of the call makes use of the fact that the predicate, $start \le st$, is idle-invariant, and the predicate, $ch = chin(n)$, is both pre- and post-idle-invariant.

(xi)

$\sqsubseteq_\rho$ Law 59 (procedure-call)

$\star$**call** *readchar*

Loop guard. The obvious termination condition for a loop refining (viii) is $n = mx$. However, *mx* is an abbreviation which, if expanded, contains a reference to the external input *in*. That is not permitted in a guard. However, *mx* corresponds to the position of the first end-of-text character in the input stream. From the loop invariant *n* is less than or equal to *mx*. Hence, from (16), $n = mx$ if and only if $chin(n) = ETX$. However, from the invariant $ch = chin(n)$. Hence $n = mx$ if and only if $ch = ETX$. Because the assumption $\star\{n = mx\}$ is required for the timing analysis performed in Sect. 5.2, it is separated out before being replaced by $ch = ETX$ in the effect.

(viii)

$\sqsubseteq_\rho$ Law 14 (establish-assumption); Law 7 (strengthen-effect)

$$\star ch, n, msg\colon \left[INV,\ \ INV \wedge ch = ETX\right]; \qquad \text{(xiii)}$$
$$\star\{n = mx\}$$

Introduce loop. The variant expression for the loop is $mx - n$. Because the loop invariant bounds n by mx, this expression is always a natural number (a well-founded set). The introduction of the loop makes use of the fact that the predicate *INV* is idle-invariant, and the expressions $ch \neq ETX$ and $mx - n$ are idle-stable.

(xiii)

$\sqsubseteq_\rho$ Law 57 (iteration)

$\star\,\mathbf{do}\; ch \neq ETX \rightarrow$

$\quad \star ch, n, msg\colon \left[ch \neq ETX \wedge INV,\ \ INV \wedge mx - n < mx - n_0\right]$ (xiv)

$\mathbf{od}$

Loop body. The body of the loop must make progress by increasing n. It must also maintain the loop invariant. We introduce a sequential composition, in which the first component establishes all the required conditions except $ch = chin(n)$. The latter is re-established by the second component.

(xiv)

$\sqsubseteq_\rho$ Law 41 (sequential-composition)

$|[\ \mathbf{con}\, u : Time \bullet$

$$\star^{ch, n,}_{msg:} \begin{bmatrix} u = \tau \wedge & msg = (1\,..\,n-1) \lhd chin \wedge \\ ch \neq ETX \wedge, & 1 \leq n \leq mx \wedge start \leq st \wedge \\ INV & mx - n < mx - n_0 \end{bmatrix}; \qquad \text{(xv)}$$

$$\star^{ch, n,}_{msg:} \begin{bmatrix} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \leq n \leq mx \wedge start \leq st \wedge \\ mx - n < mx - n(u) \end{bmatrix}, \begin{bmatrix} INV \wedge \\ mx - n < mx - n(u) \end{bmatrix} \qquad \text{(xvi)}$$

$]|$

In the application of Law 41 (sequential-composition), the logical constant u is introduced to represent the start time of the sequential composition. Hence in the effect of the first component $n_0 = n(\tau_0) = n(u)$, and in the second component $n(u)$ is used throughout to refer to the value of n at the commencement of the whole composition.

The next character of the message is already in *ch*; this can be appended to the message. When that action is combined with incrementing n, the first two conjuncts of the effect, as well as the decrease of the variant are established. Because *st* is not modified, the conjunct $start \leq st$ is invariant.

(xv)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on *ch*; Law 33 (assignment)

$n, msg := n + 1, msg \frown \langle ch \rangle$

The remaining conjunct of the invariant that needs to be re-established is $ch = chin(n)$.

(xvi)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame) on n and msg

$$\star ch\colon \left[\begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \le n \le mx \wedge start \le st \wedge \\ mx - n < mx - n(u) \end{array}\right. , \left.\begin{array}{l} msg = (1\,..\,n-1) \lhd chin \wedge \\ 1 \le n \le mx \wedge start \le st \wedge \\ mx - n < mx - n(u) \wedge \\ ch = chin(n) \end{array}\right]$$

$\sqsubseteq_\rho$ Law 24 (frame-invariant); Law 6 (weaken-assumption)

$\star ch\colon [start \le st,\ \ ch = chin(n)]$

$\sqsubseteq_\rho$ Law 59 (procedure-call)

$\star$**call** *readchar*

That completes the refinement of the receiver, except for the refinement of the procedure *readchar*.

5.4 The procedure to read a single character

We start from the specification of the procedure *readchar*, which we repeat here.

$$readchar \mathrel{\widehat{=}} \mathbf{procedure} \star ch\colon [start \le st,\ \ ch = chin(n)] \qquad \text{(xii)}$$

A delay is required until the start of the time interval corresponding to the nth character. From definitions (13) and (14) in Sect. 5.1, the start of the nth character is $start + chsep * n$.

(xii)

$\sqsubseteq_\rho$ Law 42 (simple-sequential-composition)

$\star ch\colon [start \le st,\ \ start + chsep * n \le \tau]$; (xvii)

$\star ch\colon [start + chsep * n \le \tau,\ \ ch = chin(n)]$ (xviii)

In order that the delay is code, it must not reference logical constants. Hence the (upper bound) approximation, *st*, to *start* is used.

(xvii)

$\sqsubseteq_\rho$ Law 9 (remove-from-frame); Law 7 (strengthen-effect)

$\star\, [start \le st,\ \ st + chsep * n \le \tau]$

$\sqsubseteq_\rho$ Law 6 (weaken-assumption); Def. 36 (delay)

delay until $st + chsep * n$

The character must be read before the end of the interval corresponding to the nth character, which from (13) is $start + chsep * n + chdef$.

(xviii)

$\sqsubseteq_\rho$ Law 7 (strengthen-effect)

$$\star ch{:}\left[start + chsep * n \le \tau,\ \begin{array}{l} ch \in in(\!|\tau_0 \ldots \tau|\!) \wedge \\ \tau \le start + chsep * n + chdef \end{array}\right]$$

$\sqsubseteq_\rho$ Law 6 (weaken-assumption); Law 43 (separate-deadline)

$$\star ch{:}\left[ch \in in(\!|\tau_0 \ldots \tau|\!)\right];$$
$$\mathbf{deadline}\ start + chsep * n + chdef \qquad \text{(xix)}$$

The external input *in* can only be accessed via the **read** command.

(xix)

$\sqsubseteq_\rho$ Def. 34 (read)

$$ch : \mathbf{read}(in)$$

The final program is collected in Fig. 2. We note that the program still includes references to the logical constant *start*, but because *start* is only referenced in assumptions and deadline directives, we have a valid program in our extended programming language.

5.5 Timing analysis with procedure calls

The timing analysis of the program containing the procedure *readchar* is similar to that for the program without the procedure. We illustrate the difference by examining a few corresponding paths. We use the notation *N*:*I* to refer to position *I* in the procedure called from position *N* in the program. The path *A*–*D* in Fig. 1 corresponds to the path *K*–*N*:*I* in Fig. 2. The latter path consists of the allocation of the variable *st*, the command to get the current time, allocation of *n* and *ch*, initialisation of *n* and *msg*, and the call entry overhead of procedure *readchar* at *N*. Both paths have the same time constraint.

The path *I*–*J* in the version with procedures corresponds to both the paths *D*–*E* and *F*–*G*. These paths all contain the same code and have the same time constraint. The path *D*–*F* in the version without procedures corresponds to the path *N*:*I*–*O*:*I* in the version with procedures. That is the path starting from *I* that reads a character, exits the call on procedure *readchar* at *N*, evaluates the loop guard (to *true*), updates *n* and *msg*, and enters the procedure *readchar* for a second time at *O*. The timing constraint on this path is the same as for *D*–*F*. The remaining paths in the version with procedures are analysed in a similar fashion. The only differences are the overheads for entering and exiting the procedure.

6 Conclusions

The main advantage of the sequential real-time refinement calculus presented here is that, to developers, it appears to be a straightforward extension of the standard refinement calculus. Although it has a different underlying semantics, most of the standard refinement laws carry over, and the real-time extended programming language is a superset of the standard target language.

Our conventions of using x and x_0 as abbreviations for $x(\tau)$ and $x(\tau_0)$ are intended to make refinements of functional components (i.e., not involving time) as close to Morgan's calculus [18] as possible. In practice, a development in the real-time calculus is similar to standard refinement calculus development, but with the addition of steps to separate out timing constraints and refine them into real-time language constructs. It is the ability to partition refinements into components dealing with time and those dealing with functional requirements that allows conventional refinement of the functional requirements [6].

The specifications provided by our approach are quite general. Not only do they allow time limits to be specified, but they also allow the detailed specification of the behaviour of outputs over time, as well as assumptions about the behaviour of inputs over time.

Real-time specifications may contain timing deadlines, both explicitly and implicitly. To be able to refine such specifications to code in a real-time programming language, it too must include mechanisms for specifying deadlines in the code. The approach we have taken is to extend a real-time programming language with the deadline directive. This extension provides a *machine-independent* sequential real-time programming language. An advantage of our approach is that deadlines are associated with paths through the code. This is more flexible and realistic than over specifying deadlines by requiring each language construct to have a maximum execution time.

Of course to compile a program in our extended programming language, we must ensure that the compiled code meets all timing deadlines. This requires a sophisticated timing analysis procedure. Hence our seemingly simple extension of adding a deadline directive leads to a more complex 'compilation' process. However, such analysis is essential to any approach to ensuring deadlines are met by the machine code of a hard real-time system.

In general, the analysis may need to make use of quite general properties of a program. For example, in the analysis of the exit path from the loop in the receiver example, the assumption, $n = mx$, generated during the refinement, was essential to be able to simplify the timing constraint on the exit path. The combination of refinement and timing analysis within one framework, allows such properties to be passed from the refinement to the timing analysis as assumptions in the code. Even so, resolving timing constraints is non-trivial.

Details of techniques for extracting timing path constraints are given in [5]. Analysing machine code to determine worst-case execution time bounds is dealt with by [11].

In this paper we have developed a refinement calculus for a real-time extension of Dijkstra's language. Interestingly, the techniques are able to cope with the nondeterministic constructs in the language. We believe that the techniques may be applied to develop a real-time refinement calculus for other sequential, real-time programming languages, such as Ada. Of course, we only consider the sequential subset of Ada, and additional research is required to cope with Ada exceptions, etc. Another construct that is common in real-time programming languages is the time-out; that is an area for future work.

The motivation for the definition of the specification command comes from Mahony's real-time refinement calculus [12]. That calculus allows specification of system behaviour over all time, and allows refinement of a specification to a set of truly parallel processes. The sequential real-time refinement calculus could be used to refine each such process to sequential code, but more work is required to properly integrate concurrency into our calculus.

Acknowledgements. We would like to thank Colin Fidge, Stephen Grundon, Jim Grundy, Brendan Mahony, Raymond Nickson, Trevor Vickers and Luke Wildman for feedback on earlier drafts of this paper; the members of IFIP Working Group 2.3 on Programming Methodology for feedback on this topic; and the anonymous referees whose feedback led to a number of improvements in the paper and the inclusion of Sect. 4. Ian Hayes would like to acknowledge the support of the the Australian Research Council (ARC) Large Grant A49801500, *A Unified Formalism for Concurrent Real-time Software Development*, The University of Queensland Special Studies Programme for the second half of 1996, and the hospitality of both the Oxford University Computing Laboratory and the Department of Computer Science at the Australian National University. Part of Mark Utting's contribution was funded by the Information Technology Division of DSTO.

References

1. R.-J. Back: Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980
2. R.-J. Back, J. von Wright: Refinement Calculus: A Systematic Introduction. Berlin Heidelberg New York: Springer 1998
3. E.W. Dijkstra. A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall 1976
4. S. Grundon: Timing constraint analysis for real-time programming. Honours thesis, Department of Computer Science, The University of Queensland, November 1996
5. S. Grundon, I. J. Hayes, C. J. Fidge: Timing constraint analysis. In: C. McDonald (ed.) Computer Science '98: Proc. 21st Australasian Computer Science Conf. (ACSC'98), Perth, 4–6 Feb., pp. 575–586. Berlin Heidelberg New York: Springer 1998

6. I. J. Hayes: Separating timing and calculation in real-time refinement. In: J. Grundy, M. Schwenke, T. Vickers (eds) International Refinement Workshop and Formal Methods Pacific 1998, pp. 1–16. Berlin Heidelberg New York: Springer 1998
7. I. J. Hayes, B. P. Mahony: Using units of measurement in formal specifications. Formal Asp. Comput. 7(3): 329–347 (1995)
8. I. J. Hayes, M. Utting: Deadlines are termination. In: D. Gries, W.-P. de Roever (eds) IFIP TC2/WG2.2, 2.3 International Conference on Programming Concepts and Methods (PROCOMET'98), pp. 186–204. London: Chapman and Hall 1998
9. J. Hooman: Assertional specification and verification. In: M. Joseph (ed.) Real-time Systems: Specification, Verification and Analysis, Chap. 5, pp. 97–146. Englewood Cliffs, N.J.: Prentice Hall 1996
10. 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, volume 1300 of Lecture Notes in Computer Science, pp. 1274–81. Berlin Heidelberg New York: Springer 1997
11. Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, Chong Sang Kim: An accurate worst case timing analysis for RISC processors. IEEE Trans. Software Eng. 21(7): 593–604 (1995)
12. B. P. Mahony: The Specification and Refinement of Timed Processes. PhD thesis, Department of Computer Science, The University of Queensland, 1992
13. B. P. Mahony: Calculating the least conjunctive refinement and promotion in the refinement calculus. Formal Asp. Comput. 11: 75–105 (1999)
14. B. P. Mahony, I. J. Hayes: A case study in timed refinement: A central heater. In: Proc. BCS/FACS Fourth Refinement Workshop, Workshops in Computing, pp. 138–149. Berlin Heidelberg New York: Springer 1991
15. B. P. Mahony, I. J. Hayes: Using continuous real functions to model timed histories. In: P. A. Bailes (ed.) Proc. of the 6th Australian Software Engineering Conference (ASWEC91), pp. 257–270. Australian Computer Society, July 1991
16. C. C. Morgan: Data refinement using miracles. Inform. Process. Lett. 26(5): 243–246 (1988)
17. C. C. Morgan: Procedures, parameters, and abstraction: Separate concerns. Sci. Comput. Program. 11(1): 17–28 (1988)
18. C. C. Morgan: Programming from Specifications, 2nd edn. Englewood Cliffs, N.J.: Prentice Hall 1994
19. J.M. Morris: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Program. 9(3): 287–306 (1987)
20. D. J. Scholefield: A Refinement Calculus for Real-Time Systems. PhD thesis, Department of Computer Science, University of York, U.K., 1992
21. D. J. Scholefield, H. Zedan, He Jifeng: A specification-oriented semantics for the refinement of real-time systems. Theor. Comput. Sci. 131: 219–241 (1994)
22. M. Utting, C. J. Fidge: A real-time refinement calculus that changes only time. In: He Jifeng (ed.) Proc. 7th BCS/FACS Refinement Workshop, Electronic Workshops in Computing. Springer, July 1996. URL http://www.springer.co.uk/eWiC/Workshops/7RW.html
23. M. Utting, C. J. Fidge: Refinement of infeasible real-time programs. In: Proc. Formal Methods Pacific '97, Discrete Mathematics and Theoretical Computer Science, pp. 243–262, Wellington, New Zealand, July 1997. Springer