ELSEVIER

Science of Computer Programming 43 (2002) 161–192

Science of Computer Programming

www.elsevier.com/locate/scico

Reasoning about real-time repetitions: terminating and nonterminating

Ian Hayes

School of Computer Science and Electrical Engineering, The University of Queensland, Brisbane 4072, Australia

Abstract

It is common for a real-time system to contain a nonterminating process monitoring an input and controlling an output. Hence, a real-time program development method needs to support nonterminating repetitions. In this paper we develop a general proof rule for reasoning about possibly nonterminating repetitions. The rule makes use of a Floyd–Hoare-style loop invariant that is maintained by each iteration of the repetition, a Jones-style relation between the pre- and post-states on each iteration, and a deadline specifying an upper bound on the starting time of each iteration. The general rule is proved correct with respect to a predicative semantics.

In the case of a terminating repetition the rule reduces to the standard rule extended to handle real time. Other special cases include repetitions whose bodies are guaranteed to terminate, nonterminating repetitions with the constant true as a guard, and repetitions whose termination is guaranteed by the inclusion of a fixed deadline. © 2002 Elsevier Science B.V. All rights reserved.

1. Introduction

Our overall goal is to provide a method for the formal development of real-time programs. One problem with real-time programs is that the timing characteristics of a program are not known until it is compiled for a particular machine, whereas we would prefer a machine-independent program development method. The approach we have taken is to extend a real-time programming language with a *deadline* command [1] that allows timing constraints to be incorporated into a real-time program. The deadline command has a simple semantics: it takes no time to execute and guarantees to complete by a given time. For example, the following code reads the value of the

E-mail address: ianh@csee.uq.edu.au (I. Hayes).

0167-6423/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-6423(02)00024-2

input d_1 into the local variable x, calculates $f(x)$, assigns it to y, and writes y to the output d_2. The special variable τ stands for the current time. The starting time of the commands is captured in the auxiliary variable m, and the final command is a deadline of $m + U$; this ensures that the commands complete within U seconds of them beginning.

$$\begin{array}{l} m := \tau; \quad \text{-- } \tau \text{ is the current time variable} \\ x : \mathbf{read}(d_1); \\ y := f(x); \\ d_2 := y; \\ \mathbf{deadline}\ \ m + U \end{array} \tag{1}$$

In isolation a deadline command cannot be implemented, but if it can be shown that all execution paths leading to a deadline command reach it before its deadline, then it can be removed. We consider such checking to be part of an extended compilation phase for the program, rather than part of the program development phase. Unfortunately, there is the possibility that the compiled code may not meet all the deadlines. In this case the program is not suitable and either we need to redevelop (parts of) the program, or alternatively find a faster machine or a compiler that generates better code.

The deadline command allows *machine-independent* real-time programs to be expressed. It also allows one to separate out timing constraints to leave components that are purely calculations [4]; these components can then be developed as in the non-real-time calculus. To date we have developed a sequential real-time refinement calculus [9,11] that can be viewed as an extension [4] of the standard sequential refinement calculus [19]. In this paper we generalise our earlier work on rules for nonterminating repetitions [6] so that it makes use of a relational approach similar to that of Jones [18]. We provide a single general rule for introducing possibly nonterminating repetitions. The rule subsumes all our earlier rules, and the relational approach also considerably simplifies the use of the rule in practice. We give a predicative semantics [7] for the real-time language in the style of Hehner [13,12] and Hoare and He [15]. Within this framework we give a simpler relational-style semantics for repetitions, and prove the general rule correct with respect to this semantics.

1.1. Related work

Hooman and Van Roosmalen [17] have developed a platform-independent approach to real-time software development similar to ours. Their approach makes use of timing annotations that are associated with commands. The annotations allow the capture in auxiliary timing variables of the time of occurrence of significant events that occur with the associated command, and the expression of timing deadlines on the command relative to such timing variables. They give an example similar to (1) using their notation:

$$\begin{array}{l} in(d_1, x)[m?]; \\ y := f(x); \\ out(d_2, y)[< m + U] \end{array}$$

The constructs in square brackets are timing annotations [17, Section 2]. On the input the annotation 'm?' indicates that the time at which the input occurs should be assigned to timing variable m, and on the output the annotation '$<m+U$' requires the output to take effect before $m+U$, i.e., within U time units of the input time. Hooman and Van Roosmalen keep timing annotations separate from the rest of the program. They give Hoare-like rules for reasoning about programs in their notation, but there is no semantics against which to justify the rules. The rules given in this paper are more general than those given by Hooman and Van Roosmalen. In addition the use of idle-invariant properties simplifies the application of the rules in practice, and the semantics given in this paper could be used to justify their Hoare axioms.

Section 2 outlines the real-time refinement calculus. Section 3 defines a possibly nonterminating repetition construct. Section 4 develops refinement laws for introducing possibly nonterminating repetitions. Section 5 gives examples of the application of the laws, and Section 6 discusses timing constraint analysis for this example. Section 7 gives a proof of the general law.

2. Real-time refinement calculus

We model time by nonnegative real numbers:

$$Time \mathrel{\widehat{=}} \{r : \mathbf{real}_\infty \mid 0 \leqslant r < \infty\},$$

where $\mathbf{real}_\infty$ is the set of real numbers including infinities, and operators on the reals are extended to allow infinite arguments. The real-time refinement calculus makes use of a special real-valued variable, τ, for the current time. To allow for nonterminating programs, we allow τ to take on the value infinity.

$$Time_\infty \mathrel{\widehat{=}} Time \cup \{\infty\}.$$

In real-time programs we distinguish four kinds of variables:

- inputs, which are under the control of the environment of the program;
- outputs, which are under the control of the program;
- local variables, which are under the control of the program, but unlike outputs are not externally visible; and
- auxiliary variables, which are similar to local variables, but are only used for reasoning about the program and do not appear in the machine code generated by a compiler; assignments to auxiliary variables take no time.

Inputs and outputs are modelled as total functions from *Time* to the declared type of the variable. Note that it is not meaningful to talk about the value of a variable at time infinity. Only the (special) current time variable, τ, may take on the value infinity. Within the semantics of a command, local and auxiliary variables are modelled by their before and after values. We sometimes need to refer to the set of all variables in scope. We call this ρ. It is partitioned into $\rho.in$, $\rho.out$, $\rho.local$ and $\rho.aux$. We use the term *state* to refer to the combination of the local and auxiliary variables, the abbreviation $\rho.v$ to stand for the state variables, and decorations of $\rho.v$, such as $\rho.v_0$, to stand for decorated state variables.

The semantics of the real-time language follows an approach similar to that of Utting and Fidge [20]. In this paper we represent the semantics of a command by a predicate in a form similar to that of Hehner [13,12] and Hoare and He [15]. The predicate relates the start time of a command, τ_0, and the initial values of the local and auxiliary variables to its finish time, τ (which may be infinity) and the final values of the local and auxiliary variables, as well as constraining the traces of the outputs over time. All our commands insist that time does not go backwards: $\tau_0 \leqslant \tau$.

The meaning function, $\mathscr{M}$, takes the variables in scope ρ, and a command C, and returns the corresponding predicate $\mathscr{M}_\rho(C)$. As for Hehner, refinement of commands is defined as reverse entailment:

$$C \sqsubseteq_\rho D \mathrel{\widehat{=}} \mathscr{M}_\rho(C) \Leftarrow \mathscr{M}_\rho(D),$$

where the reverse entailment holds for all possible values of the variables, including τ_0 and τ. When the environment ρ is clear from the context, it is omitted.

2.1. Real-time specification command

We define a possibly nonterminating real-time *specification command* with syntax, $\infty x : [P, Q]$, where x is a vector of variables called the *frame*, P is the *assumption* made by the specification, and Q is its *effect*. The syntax is similar to that of Morgan [19] except that there is an '∞' symbol at the beginning to indicate that the command might not terminate.

P is assumed to hold at the start time of the command. P is a *single-state predicate*, that is, it contains no references to τ_0 or zero-subscripted state (i.e., local and auxiliary) variables. P may contain references to the input and output variable traces. The effect Q is a *relation* that constrains the output traces and relates the start time τ_0 as well as the initial (zero-subscripted) state variables, and the finish time τ as well as the final state variables.

The frame, x, of a specification command lists those outputs and state variables that may be modified by the command. All other outputs in scope, i.e., in $\rho.out$ but not x, are defined to be stable for the duration of the command. The predicate $stable(z, S)$ states that the variable z has the same value over all the times in the set S:

$$stable(z, S) \mathrel{\widehat{=}} S \neq \{\} \Rightarrow (\exists y \bullet z(\!|S|\!) = \{y\}),$$

where $z(\!|S|\!)$ is the image of the set S through the function z. We allow the first argument of stable to be a vector of variables, in which case all variables in the vector are stable. The notation $[\!s \ldots t\!]$ stands for the closed interval of times from s to t, and $(\!s \ldots t\!)$ stands for the open interval. We also allow half-open, half-closed intervals. The notation $\rho.out \setminus x$ stands for the set of output variables $(\rho.out)$ minus the set of variables in x.

Any state variables which are not in the frame of a specification command are unchanged. We introduce the predicate $eq(out, t_0, t, z_0, z)$ to capture the fact that the outputs *out* are stable from t_0 until t and that the pre-state, z_0, equals the post-state, z. In the case of the states, if the final time t is infinity, then the state variables do not have any counterpart in reality. Hence, the equality between z_0 and z is not required if t is infinity.

$$eq(out, t_0, t, z_0, z) \mathrel{\widehat{=}} stable(out, [t_0 \dots t]) \wedge (t \neq \infty \Rightarrow z_0 = z).$$

Definition 1 (*Real-time specification*). Given variables, ρ, a frame, x, contained in $\rho.out \cup \rho.local \cup \rho.aux$, a single-state predicate, P, and a relation, Q, the meaning of a possibly nonterminating *real-time specification* command is defined by the following. (Recall that $\rho.v$ stands for the state variables.)

$$\mathcal{M}_\rho(\infty\ x\colon\ [P,\ Q]) \mathrel{\widehat{=}} \tau_0 \leqslant \tau \wedge$$
$$(\tau_0 < \infty \wedge P[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow Q \wedge eq(\rho.out \backslash x, \tau_0, \tau, \rho.v_0 \backslash x_0, \rho.v \backslash x)).$$

As abbreviations, if the assumption, P, is omitted, then it is taken to be *true*, and if the frame is empty the ':' is omitted. Note that if assumption P does not hold initially the command still guarantees that time does not go backwards.

Because the time variable may take on the value infinity, the above specification command allows nontermination. If the command does not terminate then the final value of the state has no counterpart in reality. Hence it does not make sense to write specifications that require for example the final value of a local variable z to be zero and the command to not terminate: $z = 0 \wedge \tau = \infty$. There is no program code that can implement such a specification, so it is of little use. The following property states the condition under which the meaning of a command is independent of the final values of the state variables if the command does not terminate.

Definition 2 (*Nontermination state independent*). For a command, C, that is well-formed in an environment, ρ, C is *nontermination state independent* provided, $\tau = \infty \Rrightarrow (\mathcal{M}_\rho(C) \Leftrightarrow (\exists \rho.v \bullet \mathcal{M}_\rho(C)))$.

All the primitive real-time commands defined in Section 2.2 satisfy this property, and compound commands preserve it. Hence, the only commands that may not satisfy it are specification commands, because the effect relation Q may constrain the final state at time infinity. We require all specifications to satisfy this healthiness property as well.

All of the executable commands (Section 2.2) in our language only constrain the values of the outputs over the time interval over which they execute. Typically, the effect of a specification command only constrains the values of outputs over the execution interval of the command: $(\tau_0 \dots \tau]$. However, we do not put any such restriction in the definition of a specification command because, although the effect may constrain the value of outputs before τ_0 or after τ, the assumption of the specification may be strong enough to allow the effect to be replaced by one that only constrains the outputs over the execution interval of the command. Such 'replacement' steps are part of the refinement process. For example, if the effect constrains the value of the outputs before τ_0, then in order for the specification to be implementable, the assumption should have at least as strong a constraint on the outputs before τ_0, in which case the effect can be replaced by one that does not constrain the outputs before τ_0. It is also possible for the effect to constrain the value of the outputs after τ. For example, for a central heater controller, the effect of a specification may require the temperature to be above some lower limit *mintemp* for some time interval after τ. This is implementable provided the assumption of the specification implies that the rate of change of the temperature over time is limited. The specification can be implemented by ensuring the temperature is above *mintemp* by a large enough margin to ensure the temperature remains over *mintemp* over the required interval assuming the maximum rate of fall of the temperature.

2.2. Real-time commands

Other real-time commands can be defined in terms of equivalent specification commands. We define: a terminating (no '∞' prefix) specification command, $x : [P, Q]$; the null command, **skip**, that does nothing and takes no time; a command, **idle**, that does nothing but may take time; a multiple assignment; an assignment for auxiliary variables that takes no time; a command, **read**, to sample a value from an external input; a command, **gettime**, to determine the current time; and the deadline command. External outputs may be modified using assignments.

Definition 3 (*Real-time commands*). Given a vector of noninput variables, x; a single-state predicate P; a relation Q; a vector of idle-stable expressions, E, of the same length as x and assignment compatible with x; a vector of auxiliary variables, y; a vector of expressions, F, of the same length as y and assignment compatible with y; a noninput variable, z; an input i that is assignment compatible with z; a noninput variable t of type time; and a time-valued expression D; the real-time commands are defined as follows.

$$
\begin{array}{l}
x\colon [P,\ Q] \mathrel{\widehat{=}} \infty\ x\colon\ \left[P,\ Q \wedge \tau < \infty\right] \\
\mathbf{skip} \mathrel{\widehat{=}} [\tau_0 = \tau] \\
\mathbf{idle} \mathrel{\widehat{=}} [\tau_0 \leqslant \tau] \\
x := E \mathrel{\widehat{=}} x\colon\ [x @ \tau = E[\frac{\rho.v_0}{\rho.v}] @ \tau_0] \\
y := F \mathrel{\widehat{=}} y\colon\ [y = F[\frac{\rho.v_0}{\rho.v}] @ \tau_0 \wedge \tau = \tau_0] \\
z : \mathbf{read}\,(i) \mathrel{\widehat{=}} z\colon\ [z @ \tau \in i(\!|\, [\tau_0 \ldots \tau]\, |\!)] \\
t : \mathbf{gettime} \mathrel{\widehat{=}} t\colon\ [\tau_0 \leqslant t @ \tau \leqslant \tau] \\
\mathbf{deadline}\, D \mathrel{\widehat{=}} [\tau_0 = \tau \leqslant D @ \tau]
\end{array}
$$

In the definition of **skip** and **idle** we make use of a terminating specification (no '∞') with an empty frame and a default assumption of *true*. Note that τ is implicitly in the frame of such a specification, and hence in the case of **idle** it may take time. Below we use the same mechanism to represent guards, which may take time to evaluate.

We allow expressions used in assignments and guards to refer to the value of an output. Such references are to the current value of the output. Hence for an expression, E, we use the notation $E @ s$ to stand for E with all free occurrences of τ replaced by s, and all occurrences of any input or output, y, replaced by $y(s)$. The expressions used in assignments and guards are assumed to be *idle-stable*, that is, their value does not change over time provided all outputs are stable and the state does not change. In practice, this usually means that an idle-stable expression cannot refer to the special time variable, τ, or to the value of external inputs. In the definition, the '@' operator does not affect state variables, and hence the same state variables appear in $E @ \tau_0$ and $E @ \tau$.

Definition 4 (*Idle-stable*). Given variables, ρ, an expression E is *idle-stable* provided, $\tau_0 \leqslant \tau < \infty \wedge \text{stable}(\rho.out, [\tau_0 \ldots \tau]) \Rightarrow E @ \tau_0 = E @ \tau$.

The **deadline** command is novel. It takes no time and guarantees to complete by the given deadline. It is not possible to implement a deadline command by generating code. Instead, we need to check that the code generated for a program that contains a deadline command will always reach the deadline command by its deadline [3].

The (demonic) nondeterministic choice between two commands may behave as either of the two commands.

Definition 5 (*Choice*). Given commands C_1 and C_2, the *nondeterministic choice* between C_1 and C_2, written $C_1 \mathbin{[\!]} C_2$, is defined by

$$\mathcal{M}_\rho(C_1 \mathbin{[\!]} C_2) \mathrel{\widehat{=}} \mathcal{M}_\rho(C_1) \vee \mathcal{M}_\rho(C_2).$$

A command is refined by a choice between two commands if and only if it is refined by each of the commands, and a choice is refined by either of its alternatives.

Law 6 (Choice). *For any commands C, C_1 and C_2,*

$$(C \sqsubseteq C_1 \mathbin{[\!]} C_2) \equiv (C \sqsubseteq C_1) \wedge (C \sqsubseteq C_2),$$
$$C_1 \mathbin{[\!]} C_2 \sqsubseteq C_1 \quad \textit{and} \quad C_1 \mathbin{[\!]} C_2 \sqsubseteq C_2.$$

Nondeterministic choice may be generalised to a choice over a set of commands.

Definition 7 (*General choice*). Given a nonempty set of commands SC, the *generalised nondeterministic choice* over the set of commands, written $[\!]\, SC$, is defined by

$$\mathcal{M}_\rho([\!]\, SC) \mathrel{\widehat{=}} (\exists C : SC \bullet \mathcal{M}_\rho(C)).$$

A command is refined by a generalised choice over a set of commands SC if and only if it is refined by every command in SC, and a general choice is refined by each of its alternatives.

Law 8 (General choice). *Given a command C, and a nonempty set of commands SC,*

$$(C \sqsubseteq [\!]\, SC) \equiv (\forall C' : SC \bullet C \sqsubseteq C')$$
$$(\forall C : SC \bullet [\!]\, SC \sqsubseteq C).$$

Because we allow nonterminating commands, we need to be careful with our definition of sequential composition. If the first command of the sequential composition does not terminate, then we want the effect of the sequential composition on the values of the outputs over time to be the same as the effect of the first command. This is achieved by ensuring that for any command in our language, if it is ‘executed’ at $\tau_0 = \infty$, it has no effect. For the specification command this is achieved by the assumption $\tau_0 < \infty$ in Definition 1 (real-time specification).

Law 9 (Nontermination preserved). *Given an environment ρ, and a well-formed command C, the following holds*: $\tau_0 = \infty \Rightarrow (\mathcal{M}_\rho(C) \Leftrightarrow \tau = \infty)$.

The definition of sequential composition combines the effects of the two commands via a hidden intermediate state. First, we introduce a forward relational composition operator, ‘$\fatsemi$’.

Definition 10 (*Relational composition*). Given variables ρ and two relations R_1 and R_2 the (forward) *relational composition* of R_1 and R_2 is defined as follows

$$R_1 \fatsemi R_2 \mathrel{\widehat{=}} \exists \tau' : Time_\infty;\ \rho.v' : T_v \bullet R_1[\tfrac{\tau',\rho.v'}{\tau,\rho.v}] \wedge R_2[\tfrac{\tau',\rho.v'}{\tau_0,\rho,v_0}],$$

where T_v is the type of the state variables $\rho.v$.

Definition 11 (*Sequential composition*). Given variables ρ, and real-time commands C_1 and C_2, their *sequential composition* is defined as the relational composition of their meaning predicates.

$$\mathcal{M}_\rho(C_1; C_2) \mathrel{\widehat{=}} \mathcal{M}_\rho(C_1) \fatsemi \mathcal{M}_\rho(C_2).$$

The following law is a generalisation of the standard law for refining a specification to a sequential composition of specifications. For the termination case both commands must terminate. The first establishes the intermediate single-state predicate I as well as the relation R_1 between the start and finish states of the first command. The second command assumes I initially and establishes the single-state predicate S as well as the relation R_2 between its initial and final states. Hence, the sequential composition establishes S as well as the relational composition of R_1 and R_2 between its initial and final states.

For the nontermination case either the first command does not terminate and establishes Q_1, or the first command terminates establishing I and R_1 and the second command does not terminate and establishes Q_2. The overall effect is thus either Q_1 or the composition of R_1 and Q_2.

Law 12 (Sequential composition with relation). *Given single-state predicates P, I and S, and relations R_1, R_2, Q_1 and Q_2,*

$$\begin{array}{l} \quad \infty\ x\colon [P,\ (\tau < \infty \wedge S \wedge (R_1 \fatsemi R_2)) \vee (\tau = \infty \wedge (Q_1 \vee (R_1 \fatsemi Q_2)))] \\ \sqsubseteq \infty\ x\colon [P,\ (\tau < \infty \wedge I \wedge R_1) \vee (\tau = \infty \wedge Q_1)]; \\ \quad \infty\ x\colon [I,\ (\tau < \infty \wedge S \wedge R_2) \vee (\tau = \infty \wedge Q_2)]. \end{array}$$

Taking Q_1 and Q_2 as *false* reduces the rule back to the standard law of Jones [18] for terminating commands.

3. Definition of a real-time repetition command

A real-time repetition is similar to a conventional repetition, except that we take into account timing properties. To give the reader an idea of the differences between a real-time repetition and a standard repetition, we give the characteristic recurrences of both. A standard repetition,

$$SDO \mathrel{\widehat{=}} \mathbf{do}\, B \rightarrow C \ \mathbf{od},$$

satisfies the recurrence

$$SDO = \mathbf{if}\, B \rightarrow C;\ SDO \mathbin{[\!]} \neg\ B \rightarrow \mathbf{skip}\ \ \mathbf{fi}\,.$$

In the standard calculus, this can be rewritten in the following form:

$$SDO = ([B];\ C;\ SDO \mathbin{[\!]} [\neg\ B]),$$

where ';' has higher priority than '$[\!]$'; guarded commands of the form '$B \rightarrow S$' are rewritten in the equivalent form of a guard followed by the command ($[B];\ S$); the **if**-**fi** is replaced by a demonic choice ($[\!]$) because the guards are complementary; and ($[\neg\ B]; \mathbf{skip}$) is replaced by its equivalent, $[\neg\ B]$.

For the real-time repetition,

$$DO \mathrel{\widehat{=}} \mathbf{do}\, B \rightarrow C \ \mathbf{od},$$

there must exist a strictly positive time d, such that the following recurrence holds:

$$DO = |[\ \mathbf{aux}\, u : Time \bullet u := \tau;\ [B @ \tau];\ C;\ [u + d \leqslant \tau]\ ]|;\ DO\ \ [\!]\, [\neg\ B @ \tau].$$

The auxiliary variable u captures the start time of a single iteration. The guard $[B @ \tau]$ allows the first alternative to be executed if the guard evaluates to true. Note that in the real-time case the guard evaluation may take time but must terminate. The delay until (absolute) time $u + d$ at the end of an iteration ensures that each iteration takes a minimum time, d. This rules out Zeno-like behaviour in which, for example, each iteration takes half the time of the previous iteration. The value of d can be arbitrarily small (e.g., 1 attosecond), but it must be greater than zero. A repetition of the form **do** *true* $\rightarrow$ … **od** typically has the minimum overhead; its implementation may take no time to evaluate the guard, but there will be a minimum time overhead for the branch back to the start of the repetition.

The boolean expression B is assumed to be idle-stable. That is, its value does not change with just the passage of time if the variables under the control of the program are stable. In practice, this means B cannot refer to the current time variable, τ, or to external inputs (which may change over time). We assume the guard evaluation terminates, but we place no explicit upper bound on the time taken for guard evaluation, because guard expressions may be arbitrarily complex. For a particular application there may be a time bound on guard evaluation, but this is catered for by using explicit deadline commands within the body of the repetition. There is no need for a separate upper bound on the guard evaluation time in the definition of the repetition.

After completing the command, C, in the body of the repetition, it repeats the guard evaluation. The delay until $u + d$ at the end of the iteration ensures the minimum execution time for each iteration; if the rest of the iteration has already taken at least d time units then the delay need take no time. Because there is no explicit upper limit on the termination time of the delay, it also allows for the time taken for the repetition to branch back to the guard evaluation.

The exit alternative of the repetition, $[\neg B @ \tau]$, allows for the time taken to evaluate the guard (to false) and exit the repetition, including the case if the guard of the repetition is false initially. We place no explicit time bounds on this command in the definition, but for a particular application the code following the repetition may include deadline commands, which explicitly introduce a time constraint. There is no lower time bound on the exit alternative because the repetition $\mathbf{do}\, \mathit{false} \rightarrow \ldots\ \mathbf{od}$ can be implemented by **skip**, which takes no time.

In order to define the behaviour of a repetition, we introduce an abbreviation to stand for the effect of one iteration of the repetition.

$$\mathit{ITER} \mathrel{\widehat=} |[\ \mathbf{aux}\, u : \mathit{Time} \bullet u := \tau;\ [B @ \tau];\ C;\ [u + d \leqslant \tau]\ ]|.$$

The repetition may either complete a finite number of iterations or iterate forever. In the finite case the last iteration either terminates and establishes $\neg B @ \tau$ or it does not terminate because C does not terminate. We introduce the notation C^* to stand for any finite number of repetitions of a command C. It is defined as the nondeterministic choice over the natural numbers of each finite number of iterations (including zero) of C.

Definition 13 (*Finite iterations*). For a command C,

$$C^* \mathrel{\widehat=} [] \{ i : \mathbb{N} \bullet C^i \},$$

where $C^0 \mathrel{\widehat=} \mathbf{skip}$, and for $n : \mathbb{N}$, $C^{n+1} \mathrel{\widehat=} C^n; C$.

The finite number of iterations case for a repetition can then be defined as

$$\mathit{ITER}^*; [\neg B @ \tau].$$

This takes care of both the case in which the last iteration terminates, in which case it establishes $\neg B @ \tau$, and the case in which the last iteration does not terminate, in which case by Law 9 (nontermination preserved) $\neg B @ \tau$ has no effect. If the repetition guard is initially false, then the only possibility is zero executions of *ITER*, which corresponds to **skip**.

The other possibility is that the repetition executes an infinite number of iterations. We introduce the notation C^∞ to stand for this. To define C^∞ we introduce an infinite sequence of times, $\vec{t}$, and an infinite sequence of states (local and auxiliary variables), $\vec{v}$. Each adjacent pair of time and state in the sequences are related by C. The first time ($\vec{t}_0$) and state ($\vec{v}_0$) correspond to the overall initial time (τ_0) and state ($\rho.v_0$). Because an infinite number of iterations of a command that takes no time has no counterpart in reality, we require that each execution of the iterated command takes some minimum time d. This also avoids unrealistic Zeno-like behaviour.

Definition 14 (*Infinite iterations*). For a command C, such that for some strictly positive time d any execution of C guarantees to take at least d time units, i.e., $\tau_0 < \infty \wedge \mathscr{M}_\rho(C) \Rrightarrow \tau_0 + d \leqslant \tau$,

$$\begin{aligned}\mathscr{M}_\rho(C^\infty) \mathrel{\widehat{=}} \tau = \infty \wedge (\tau_0 < \infty \Rightarrow \\ (\exists \vec{t} : \mathbb{N} \rightarrow \mathit{Time}; \vec{v} : \mathbb{N} \rightarrow T_v \bullet \vec{t}_0 = \tau_0 \wedge \vec{v}_0 = \rho.v_0 \wedge (\forall i : \mathbb{N} \bullet \mathscr{M}_\rho(C_{i+1})))),\end{aligned}$$

where $\rho.v$ is the vector of state variables in the environment and T_v is the corresponding type of $\rho.v$, and

$$C_{i+1} \mathrel{\widehat{=}} C\left[\frac{\vec{t}_i, \vec{t}_{i+1}, \vec{v}_i, \vec{v}_{i+1}}{\tau_0, \tau, \rho.v_0, \rho.v}\right].$$

Note that none of the times in the sequence $\vec{t}$ may be infinity (as constrained by the type of the sequence) because in order to have an infinite number of iterations each iteration must terminate. Also note that C^∞ does not define any final value of the state $\rho.v$, because there is no such final state. We are now in a position to define a real-time repetition.

Definition 15 (*Repetition*). Given variables ρ, a boolean-valued, idle-stable expression B, a command C, and fresh names d and u,

$$\begin{aligned}&\mathscr{M}_\rho(\mathbf{do}\, B \rightarrow C\ \mathbf{od}) \mathrel{\widehat{=}} \\ &\qquad [\!]\,\{d : \mathit{Time} \mid 0 < d \bullet (\mathit{ITER}^*;\ [\neg\ B @ \tau]\,[\!]\,\mathit{ITER}^\infty)\},\end{aligned}$$

where $\mathit{ITER} \mathrel{\widehat{=}} |[\ \mathbf{aux}\, u : \mathit{Time};\ u := \tau;\ [B @ \tau];\ C;\ [u + d \leqslant \tau]\ ]|$.

Note that there is just one choice made for d, and that value is used for all iterations of the repetition. That rules out, for example, successive iterations of a repetition choosing progressively smaller values of d, and hence it rules out Zeno-like behaviour. A particular implementation of a repetition will determine a suitable value of d. Our implementation-independent approach allows any value.

4. General repetition introduction law

In this section we develop laws that make use of loop invariant approach of Floyd [2] and Hoare [14] and the relational approach of Jones [18] for real-time, possibly nonterminating repetitions. A loop invariant is assumed to hold initially and must be maintained by every iteration of a repetition. If the repetition terminates, the invariant holds in the final state and in addition the guard is false.

If we assume that an invariant, I, holds immediately before a repetition starts, we would like to assume that both its guard B and I hold at the start of the execution of the command, C, within the body of the repetition. However, there is a period of time corresponding to the guard evaluation between the two points in the program. Because B is assumed to be idle-stable, it will still hold at the start of the execution of C. For the invariant, I, we need the condition that, if I holds before evaluation of the guard, it will still hold after the evaluation. This is equivalent to I being invariant over the execution of an **idle** command, and we refer to this property as I being *idle-invariant*. All outputs are stable for the duration of an **idle** command, and the state variables do not change.

Definition 16 (*Idle-invariant*). Given variables ρ, a single-state predicate P is *idle-invariant* provided

$$\tau_0 \leqslant \tau < \infty \wedge stable(\rho.out, [\tau_0 \ldots \tau]) \wedge P[\tfrac{\tau_0}{\tau}] \Rrightarrow P.$$

The conditions idle-stable and idle-invariant differ in that for the former the value does not change over the execution of an idle command, whereas for the latter, if the value holds before, then it holds after. The latter differs from the former in that for idle-invariance, if the predicate is false beforehand, then it may become true during the execution of the idle.

If the command, C, in the body maintains the invariant, then on termination of C, I holds, and because I is idle-invariant, it will still hold after the delay at the end of the repetition body, and hence at the start of the next iteration, as required. The assumption that I is idle-invariant places restrictions on how I can refer to the current time variable, τ, because τ increases on execution of an idle command, and on how I refers to external inputs, because these may change over the execution of an idle command. For example, predicates of the form $D \leqslant \tau$, where D is an idle-stable expression, are idle-invariant, but predicates of the form $\tau \leqslant D$ are not because the passage of time may cause τ to exceed D. If I can be expressed in a form that does not refer to the current time, τ, and all references to external inputs are explicitly indexed with expressions that are idle-stable, then I is idle-invariant. In practice, the link between the current time, τ, and the invariant, I, is made through a time-valued program variable that approximates τ.

The standard laws on which ours are based are those of Jones [18] which, in addition to an invariant, make use of a relation between initial and final states. If a relation R holds between the initial and final states of the body of the repetition on each iteration, then if the repetition terminates the transitive closure of the relation, R^*, holds between the initial and final states of the whole repetition. We define transitive closure and iteration of relations. A relation iterated zero times corresponds to the identity relation on the current time and state.

Definition 17 (*Transitive closure*). Given a relation R, its *transitive closure*, R^*, is defined as follows.

$$R^* \mathrel{\widehat{=}} (\exists i : \mathbb{N} \bullet R^i),$$

where $R^0 \mathrel{\widehat{=}} \tau_0 = \tau \wedge \rho.v_0 = \rho.v$, and for a natural number i, $R^{i+1} \mathrel{\widehat{=}} R^i \mathbin{⨟} R$.

An alternative way to view the transitive closure of a relation R is that there is a sequence of intermediate times and states with adjacent pairs of times and states related by R.

$$R^* \equiv (\exists i : \mathbb{N};\ \vec{t} : \mathbb{N} \to Time;\ \vec{v} : \mathbb{N} \to T_v \bullet \vec{t}_0 = \tau_0 \wedge \vec{v}_0 = \rho.v_0 \wedge \\ \vec{t}_i = \tau \wedge \vec{v}_i = \rho.v \wedge (\forall j : \mathbb{N} \bullet j < i \Rightarrow R_{j+1})),$$

where $R_{j+1} \mathrel{\widehat{=}} R[\vec{t}_j, \vec{t}_{j+1}, \vec{v}_j, \vec{v}_{j+1} / \tau_0, \tau, \rho.v_0, \rho.v]$.

As with the invariant we need to be careful about the time intervals corresponding to the guard evaluation and the minimum delay (branch back). Hence, we require that the relations used are impervious to these idle periods. We use the terms *pre-idle-invariant* and *post-idle-invariant* to refer to relations that are impervious to pre and post, respectively, idle periods. If the only references to τ_0 and τ are as indices of outputs, the relation is both pre- and post-idle-invariant. We introduce the relation *IDLE*, which corresponds to the meaning of the **idle** command.

$$IDLE \mathrel{\widehat{=}} \tau_0 \leqslant \tau \wedge (\tau_0 < \infty \Rightarrow \tau < \infty \wedge eq(\rho.out, \tau_0, \tau, \rho.v_0, \rho.v)).$$

Definition 18 (*Pre-idle-invariant*). A relation R is *pre-idle-invariant* provided, $\tau_0 < \infty \Rrightarrow (IDLE \mathbin{⨟} R \Rightarrow R)$.

Definition 19 (*Post-idle-invariant*). A predicate R is *post-idle-invariant* provided, $\tau < \infty \Rrightarrow (R \mathbin{⨟} IDLE \Rightarrow R)$.

For a nonterminating repetition, there is no final state, and hence no concept of the loop invariant holding in the final state. Instead, we use a strategy similar to that of Hooman [16, p. 129]. There is a sequence of times corresponding to the starting times of executions of the repetition body at which the invariant is true. If the body always terminates, then the sequence is infinite. In that case, for any time, τ', after the start time of the repetition, there is always some later time, τ, at which both the invariant, I, and the guard, B, hold. In addition, the state at that time is related to the initial state of the whole repetition by R^*. Because the loop invariant is idle-invariant, it cannot express upper bounds on the current time, τ. In order to express such bounds we introduce a deadline command, **deadline** D, at the beginning of the body of the repetition. Hence, we can also deduce that $\tau \leqslant D$ holds at the beginning of every iteration. Assuming that the frame of the body of the repetition is x, the overall effect that holds for a nonterminating repetition with a terminating body is

$$I_\infty \mathrel{\widehat{=}} (\forall \tau' : Time \bullet (\exists \tau : Time; \rho.v : T_v \bullet \tau' \leqslant \tau \wedge B @ \tau \wedge \\ \tau \leqslant D @ \tau \wedge I \wedge R^* \wedge eq(\rho.out \backslash x, \tau_0, \tau, \rho.v_0 \backslash x, \rho.v \backslash x))).$$

In the case in which there are an infinite number of iterations of the repetition, there are infinite sequences of times and states such that R holds between adjacent pairs of times and states. In addition, there is some minimum separation between adjacent pairs of times. We introduce the notation R^∞ to capture this relationship.

Definition 20 (*Infinite iteration*). Given a relation R, its *infinite iteration*, R^∞, is defined as follows.

$$R^\infty \mathrel{\widehat{=}} (\exists\, d : Time;\ \vec{t} : \mathbb{N} \to Time;\ \vec{v} : \mathbb{N} \to T_v \bullet 0 < d \,\wedge \\ \vec{t}_0 = \tau_0 \wedge \vec{v}_0 = \rho.v_0 \wedge (\forall i : \mathbb{N} \bullet \vec{t}_i + d \leqslant \vec{t}_{i+1} \wedge R_{i+1})),$$

where $R_{i+1} \mathrel{\widehat{=}} R[\vec{t}_i, \vec{t}_{i+1}, \vec{v}_i, \vec{v}_{i+1} / \tau_0, \tau, \rho.v_0, \rho.v]$.

If R is a well-founded relation then R^∞ is equivalent to false because, by their very definition, well-founded relations rule out the possibility of an infinite sequence of successively related states. Note that R^∞ does not refer to either the final time τ or the final state $\rho.v$. The relation R may constrain the value of outputs, typically over the interval from τ_0 to τ. For example, R may state that an output o is stable from τ_0 through until τ, in which case R^∞ guarantees that o is stable from τ_0 forever (until ∞).

Finally, we combine the above discussion into a single law. The body of the repetition consists of a deadline command followed by a specification. The specification can assume that the initial time is before the deadline and that both the guard and the invariant hold initially. The body of the repetition either terminates, reestablishing I and establishing R between its initial and final states, or it fails to terminate but establishes Q. This repetition refines a specification that assumes that the invariant holds initially and either,

- terminates in a state in which the guard is false, the invariant holds, and the relation R^* is established between the initial and final states;
- fails to terminate because the body failed to terminate, and overall establishes $(R^* \mathbin{⨾} Q)$; or
- fails to terminate because the body always terminates but the guard always remains true, in which case the predicate I_∞ and infinite iteration of the relation, R^∞, are established.

Law 21 (Repetition). *Given an idle-stable, boolean-valued expression, B; a single-state, idle-invariant predicate, I; an idle-stable, time-valued expression, D; a pre-idle-invariant relation Q; and a pre- and post-idle invariant relation R; then*

$$\infty\ x\colon \left[I,\ \begin{matrix}(\tau < \infty \wedge \neg\, B@\tau \wedge I \wedge R^*) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* \mathbin{⨾} Q)))\end{matrix} \right]$$

$$\sqsubseteq \mathbf{do}\, B \to \mathbf{deadline}\, D; \\ \qquad \infty\ x\colon \left[B@\tau \wedge \tau \leqslant D@\tau \wedge I,\ \begin{matrix}(\tau < \infty \wedge I \wedge R) \vee \\ (\tau = \infty \wedge Q)\end{matrix} \right] \\ \mathbf{od}.$$

Note that I may not refer to τ_0 or initial variables because I is used both in the specification, in which τ_0 is the start time of the whole repetition, and in the body of the repetition, in which τ_0 is the start time of an iteration. In order to refer to the start time of the whole repetition within I it is necessary to introduce a fresh auxiliary variable to stand for the start time.

Taking Q as the predicate *false* gives the following special case.

Law 22 (Repetition–terminating body). *Given an idle-stable boolean-valued expression, B; a single-state idle-invariant predicate, I; a pre- and post-idle-invariant relation R; and an idle-stable, time-valued expression, D; then*

$$\begin{array}{l} \infty\, x\colon [I,\ (\tau < \infty \wedge \neg\, B@\tau \wedge I \wedge R^*) \vee (\tau = \infty \wedge I_\infty \wedge R^\infty)] \\ \sqsubseteq \mathbf{do}\, B \rightarrow \mathbf{deadline}\, D; \\ \qquad\qquad x\colon [B@\tau \wedge \tau \leqslant D@\tau \wedge I,\ I \wedge R] \\ \quad \mathbf{od}. \end{array}$$

If R is a well-founded relation then $R^\infty \equiv false$, and hence the infinite iteration alternative is ruled out. In addition, if D is infinity, the deadline introduces no constraint whatsoever (**deadline** $\infty \sqsupseteq$ **skip**) and the law reduces to the following.

Law 23 (Terminating repetition). *Given an idle-stable, boolean-valued expression, B; a single-state, idle-invariant predicate, I; and a pre- and post-idle-invariant, well-founded relation R; then*

$$x\colon [I,\ \neg\, B@\tau \wedge I \wedge R^*] \sqsubseteq \mathbf{do}\, B \rightarrow x\colon [B@\tau \wedge I,\ I \wedge R]\ \mathbf{od}.$$

This is the standard law for refinement to a terminating repetition given in the relational form [18].

A repetition with a constant true guard never terminates.

Law 24 (Repetition-true guard). *Given a single-state, idle-invariant predicate, I; an idle-stable, time-valued expression, D; a pre-idle-invariant relation Q; and a pre- and post-idle invariant relation R; then*

$$\begin{array}{l} \infty\, x\colon \left[I,\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* \mathbin{⨾} Q)))\right] \\ \sqsubseteq \mathbf{do}\, true \rightarrow \mathbf{deadline}\, D; \\ \qquad\qquad \infty\, x\colon \left[\tau \leqslant D@\tau \wedge I,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)\right] \\ \quad \mathbf{od}. \end{array}$$

A special case of this law is for an always terminating body. As with Law 22 this can be handled by choosing Q to be *false*.

For a repetition with a terminating body ($Q \equiv false$), if the deadline D is constant for the duration of the entire repetition, then the repetition is guaranteed to terminate. This follows because I_∞ is false, due to the fact that τ' cannot be less than the fixed value D for all times τ'.

Law 25 (Deadline as termination). *Given an idle-stable boolean-valued expression*, B; *a single-state*, *idle-invariant predicate*, I; *a pre- and post-idle-invariant relation* R; *and an idle-stable*, *time-valued expression*, D, *which does not include any references to variables in the frame*; *then*

$$\begin{array}{l} x\colon [I \wedge D@\tau < \infty,\ \neg\, B@\tau \wedge I \wedge R^*] \\ \sqsubseteq \mathbf{do}\, B \rightarrow \mathbf{deadline}\, D;\ x\colon [B@\tau \wedge \tau \leqslant D@\tau \wedge I,\ I \wedge R]\ \mathbf{od}. \end{array}$$

This is a variant of a law presented elsewhere [10], but here it is just a special case of the general law.

5. Example

Before giving a proof of the general law in Section 7, to illustrate our approach we use the example of a conveyor belt that transports objects which are measured for their size and then sorted into a corresponding bin. This example was used in an earlier paper illustrating the use of auxiliary variables [5]. Here the relational law for the repetition makes the proof of correctness considerably simpler. A light beam is used to detect objects, and measure their size. The boolean input *beam* represents the detection of the light beam: its value is false (no light) at time t if and only if there is an object on the conveyor blocking the beam at time t. (We ignore failures of the light beam, etc.) The boolean output *lbin* selects between a bin for large objects (if it is true) and a bin for small objects (if it is false).

The objects on the conveyor belt have a minimum length and separation. This translates to there being a minimum time, *MinW*, for which *beam* is false while an object passes the beam, and a minimum time, *MinS*, for which *beam* is true between objects. Hence, *beam* can only change a finite number of times within any finite time interval. For such finitely variable inputs we introduce some notational conventions. For i a natural number, the notation $beam \downarrow i$ stands for the time at which *beam* makes transition number i from *true* to *false*, and $beam \uparrow i$ stands for the time at which *beam* makes its transition number i from *false* to *true*. The assumption of finite variability means that these times are well defined for every natural number provided *beam* makes an infinite number of transitions over all time. If there are only a finite number of transitions over all time, then if i is greater than the total number of up transitions, we define $beam \uparrow i$ to be infinity, and similarly for down transitions. If we assume that $beam \downarrow i < beam \uparrow i$ (the other case is symmetric) then

$$\begin{array}{l} beam(\!|\lbrace\!0 \ldots beam \downarrow 0\rbrace\!|\!) = \{true\} \\ \forall i : \mathbb{N} \bullet (beam \downarrow i < \infty \Rightarrow beam(\!|\lbrace\! beam \downarrow i \ldots beam \uparrow i \rbrace\!|\!) = \{false\}) \wedge \\ \qquad (beam \uparrow i < \infty \Rightarrow beam(\!|\lbrace\! beam \uparrow i \ldots beam \downarrow i+1 \rbrace\!|\!) = \{true\}) \end{array}$$

We assume that there is no object on the conveyor for an initial period of at least *MinS*. As well as giving the types of the variables and constants, we also give their units of measurement [8].

$$\textbf{input}\ beam : boolean; \textbf{output}\ lbin : boolean;$$
$$\textbf{const}\ MinS = 40\,\mathsf{ms}; MinW = 20\,\mathsf{ms}; MaxW = 40\,\mathsf{ms};$$

$$\begin{aligned} A \mathrel{\widehat{=}} & MinS \leqslant beam \downarrow 0 \land \forall i : \mathbb{N} \bullet \\ & \quad (beam \downarrow i < \infty \Rightarrow MinW \leqslant beam \uparrow i - beam \downarrow i \leqslant MaxW) \land \\ & \quad (beam \uparrow i < \infty \Rightarrow MinS \leqslant beam \downarrow (i+1) - beam \uparrow i) \end{aligned}$$

The task of the program is to measure the size of the passing objects, and select the bin into which they are to be placed. We assume that the conveyor moves with a constant velocity of *vel* metres per second. (The assumption of constant velocity is unrealistic, but simplifies our example; a nonconstant velocity can be accommodated within the current specification by decreasing the value of *mrgn* below.) The size of an object can only be measured approximately. Hence, the specification allows a margin of error, *mrgn*, in determining whether an object is large or small. If an object is of size greater than or equal to $limit + mrgn$ then it must go in the large bin. If its size is less than or equal to $limit - mrgn$ it must go in the small bin. Objects with sizes between $limit - mrgn$ and $limit + mrgn$ can go in either bin. The predicate *ObjSize* relates object j to the bins it is allowed to be placed in.

$$\begin{aligned} & \textbf{const}\ vel = 1\,\mathsf{m/s}; limit = 30\,\mathsf{mm}; mrgn = 1\,\mathsf{mm}; \\ & ObjSize(j,b) \mathrel{\widehat{=}} \textbf{let}\ sz = vel * (beam \uparrow j - beam \downarrow j) \bullet \\ & \quad (sz > limit + mrgn \Rightarrow b) \land (sz < limit - mrgn \Rightarrow \neg\, b) \end{aligned}$$

The output *lbin* controls the bin selector. In order for the object to be placed in the correct bin, *lbin* should have the correct value from time *bin_select* after the end of object j, i.e., time $beam \uparrow j$, through until *bin_stable* after the end of the object is detected. We introduce the predicate *ObjBin* to abbreviate this condition.

$$\begin{aligned} & \textbf{const}\ bin\_select = 10\,\mathsf{ms}; bin\_stable = 30\,\mathsf{ms}; \\ & ObjBin(j) \mathrel{\widehat{=}} (\exists\, b : boolean \bullet ObjSize(j,b) \land \\ & \qquad lbin(\!|\, (\!\!(beam \uparrow j + bin\_select \ldots beam \uparrow j + bin\_stable)\!\!) \,|\!) = \{b\}) \end{aligned}$$

The program is specified using a nonterminating specification command with a termination time, τ, of infinity.

$$\infty\ lbin\colon \left[A,\ \tau = \infty \land (\forall i : \mathbb{N} \bullet beam \downarrow i < \infty \Rightarrow ObjBin(i))\right] \tag{2}$$

Before going through the details of the refinement of the above specification, we give the final machine-independent program in Fig. 1. It makes use of a procedure *Await* that waits for the beam to attain the value of its first parameter and returns an approximation to the time at which this occurs. The program makes use of the auxiliary variable j which counts the objects as they pass. The local variables *st* and *et* capture the start and finish times of object j (approximately), and the variable *size* is used to calculate the (approximate) size of the object from the time it took to pass and its velocity. If the calculated size is greater than or equal to *limit* then *lbin* is set to *true*, otherwise it is set to *false*. It is assumed that the program starts when the current time, τ, is at least *MinS* seconds before the first object passes through the beam.

```
|[ $\boxed{\textbf{aux}\, j : natural}$;
A :: $\{\tau \leqslant beam \downarrow 0 - MinS\}$;
    $\boxed{j := 0}$;
    do true →
        |[ var st, et : natural ms; size : natural mm;
        B :: st ← Await(false, $\boxed{beam \downarrow j}$); -- start at $beam \downarrow j$
        C :: et ← Await(true, $\boxed{beam \uparrow j}$); -- end at $beam \uparrow j$
             size := (et − st) ∗ vel;
             lbin := (limit ⩽ size);
        D :: $\boxed{\textbf{deadline}\, beam \uparrow j + bin\_select}$;
             delay until et + bin_stable;
             $\boxed{j := j + 1}$
        ]|
    od
]|
```

Fig. 1. Main program.

In addition to the expected standard code there are deadline commands, a number of uses of the auxiliary variable, j, and auxiliary parameters; these are highlighted within boxes. These are used to ensure that the operation of the program takes place in a timely fashion. No code needs to be generated for any of the highlighted constructs. Their purpose is to facilitate reasoning and to allow the specification of timing constraints via deadline commands.

The task of procedure *Await* is to wait until *beam* takes on the value of its first argument, *val*, and return in result *pt* (an approximation to) the time at which the value of *beam* changes to *val*. To allow simpler specification of the procedure, an auxiliary parameter is used: *event* gives the (future) time of the awaited change. The value of *beam* from the time of the call up until *event* (an interval that may be empty) is the complement of *val*, and once it changes to *val* it remains equal to *val* for a time of at least *err*. If the value of *beam* never changes, then *Await* never returns. Otherwise it returns the result, *pt*, which is an approximation to *event*.

$$\textbf{const}\, err = mrgn/vel; \ \{err \leqslant MinS \wedge err \leqslant MinW\};$$

$$\textbf{procedure}\ pt : time \leftarrow Await(val : boolean;\ \textbf{aux}\, event : Time_\infty) =$$

$$\infty\ pt: \left[\begin{array}{ll} & event_0 = \tau = \infty \vee \\ beam(\!|\{\tau \ldots event\}|\!) \subseteq \{\neg\, val\} \wedge & (event < \infty \wedge \tau < \infty \wedge \\ beam(\!|\{event \ldots event + err\}|\!) = \{val\}' & event \leqslant pt \wedge \\ & pt \leqslant event + err) \end{array}\right]$$

The implementation of *Await* in Fig. 2 repeatedly tests the value of *beam* until it changes to equal *val*. Hence, when a value equal to *val* is read from *beam*, the time must be after *event*. The read must be completed before *event* + *err* in order to ensure that the procedure is not detecting some later change of *beam* to *val*. Hence, the deadline after the read. If the value read is equal to *val* the repetition terminates and one can deduce that *event* is less than or equal to the current time, τ. The deadline

```
|[var $p : boolean$;
    $\boxed{\textbf{aux}\ before : Time_\infty}$;
    $\boxed{before := event}$;
    $p := \neg val$;
    do $p \neq val \rightarrow$
        $\{(p = val \Rightarrow event \leqslant \tau) \wedge (p \neq val \Rightarrow before \leqslant event)\}$;
        $\boxed{\textbf{deadline}\ event + err}$;
    $E ::$ $\boxed{before := \tau}$;
        $p$ : read ($beam$);
    $F ::$ $\boxed{\textbf{deadline}\ event + err}$;
    od;
    $\{event \leqslant \tau\}$;
    $pt$ : gettime;
$G ::$ $\boxed{\textbf{deadline}\ event + err}$
]|
```

Fig. 2. Body of procedure Await.

after the **gettime** ensures that the value of *pt* is a close enough approximation to *event*. If the repetition never terminates then for any time, τ', there is a later time, τ, at which $\tau \leqslant event + err$. As there is no upper bound on τ' and $\tau' \leqslant \tau \leqslant event + err$, this implies *event* must be infinity. The deadline at the start of the body of the repetition is introduced as a consequence of the form of the law for introducing a repetition, although in this case it is subsumed by the tighter deadline labelled F. Note that the latter deadline is not subsumed by the deadline at G. If both deadlines within the body of the repetition were removed then it is possible for the read to always occur while *beam* is not equal to *val*, and hence for the repetition to never terminate. However, if the repetition never terminates, the deadline at G is never reached and does not have to be considered.

5.1. Refinement of the main program

To refine the specification (2) of the main program to the code in Fig. 1 we make use of Law 24 (repetition-true guard). The desired effect of (2) is

$$\tau = \infty \wedge (\forall i : \mathbb{N} \bullet beam \downarrow i < \infty \Rightarrow ObjBin(i)). \tag{3}$$

If $beam \downarrow j = \infty$ for some j, then $beam \downarrow k = \infty$ for all values of k greater than or equal to j. Hence we can split effect (3) into the case in which there is a never ending stream of objects on the conveyor, and the case in which there is only a finite number of objects.

$$\begin{aligned}&\tau = \infty \wedge \\ &((\forall i : \mathbb{N} \bullet beam \downarrow i < \infty \wedge ObjBin(i)) \vee \\ &\ (\exists m : \mathbb{N} \bullet (\forall i : \mathbb{N} \bullet i < m \Rightarrow beam \downarrow i < \infty \wedge ObjBin(i)) \ \wedge \ beam \downarrow m = \infty)).\end{aligned} \tag{4}$$

We need to devise relations R and Q such that the first alternative corresponds to R^{∞} and the second to $R^* \mathbin{⨾} Q$. The obvious choice is to have iteration i of the repetition establish $ObjBin(i)$ provided $beam \downarrow i < \infty$. To accomplish this we introduce an auxiliary variable j, that counts iterations. A suitable pre- and post-idle-invariant relation R is defined as follows:

$$R \mathrel{\widehat{=}} beam \downarrow j_0 < \infty \wedge ObjBin(j_0) \wedge j = j_0 + 1.$$

If there is only a finite number of objects in total, $beam \downarrow j = \infty$ for some j, and no further processing takes place once that is reached. Hence, we take Q to be the following pre-idle-invariant relation. (Q is written in terms of j_0 rather than j because in this case τ is infinity and there is no final state.)

$$Q \mathrel{\widehat{=}} beam \downarrow j_0 = \infty.$$

Using Definition 20 (infinite iteration) we instantiate R^{∞} with the sequence of values of the auxiliary variable j represented by the vector $\vec{j}$, i.e., $\vec{j}_0$ is the initial value of j and $\vec{j}_i$ is the value of j after iteration i.

$$\begin{array}{l}(\exists d : Time;\ \vec{t} : \mathbb{N} \to Time;\ \vec{j} : \mathbb{N} \to \mathbb{N} \bullet 0 < d \wedge \vec{t}_0 = \tau_0 \wedge \vec{j}_0 = j_0 \wedge \\ \quad (\forall i : \mathbb{N} \bullet \vec{t}_i + d \leqslant \vec{t}_{i+1} \wedge beam \downarrow \vec{j}_i < \infty \wedge ObjBin(\vec{j}_i) \wedge \vec{j}_{i+1} = \vec{j}_i + 1)).\end{array}$$

Assuming j has been initialised to zero, i.e., $j_0 = \vec{j}_0 = 0$, then because $\vec{j}_{i+1} = \vec{j}_i + 1$ for all i, we can deduce $\vec{j}_i = i$ for all i. Hence, ignoring the timing information, the above implies the following, which is the first alternative of our expanded requirement (4).

$$(\forall i : \mathbb{N} \bullet beam \downarrow i < \infty \wedge ObjBin(i))).$$

For the finite iterations case $R^* \mathbin{⨾} Q$ is the following, if we assume j is initially zero. Here we make use of the sequence version of transitive closure.

$$\begin{array}{l}\quad \exists m : \mathbb{N};\ \vec{t} : \mathbb{N} \to Time;\ \vec{j} : \mathbb{N} \to \mathbb{N} \bullet \vec{t}_0 = \tau_0 \wedge \vec{j}_0 = 0 \wedge \\ \qquad (\forall i : \mathbb{N} \bullet i < m \Rightarrow beam \downarrow \vec{j}_i < \infty \wedge ObjBin(\vec{j}_i) \wedge \vec{j}_{i+1} = \vec{j}_i + 1) \wedge \\ \qquad beam \downarrow m = \infty \\ \Rrightarrow \text{ignoring timing information; } \vec{j}_i = i \\ \quad \exists m : \mathbb{N} \bullet (\forall i : \mathbb{N} \bullet i < m \Rightarrow beam \downarrow i < \infty \wedge ObjBin(i)) \wedge \\ \qquad beam \downarrow m = \infty\end{array}$$

This is the second alternative of the expanded requirement (4). Applying Law 24 (repetition-true guard) with the deadline D infinity (i.e., the deadline can be ignored), I the predicate *true*, and R and Q as defined above gives the following repetition:

$$\begin{array}{l}\quad (2) \\ \sqsubseteq \mathbf{do}\ true \to \\ \qquad \infty\ j, lbin\colon \begin{bmatrix}(\tau < \infty \wedge beam \downarrow j_0 < \infty \wedge ObjBin(j_0) \wedge j = j_0 + 1) \vee \\ (\tau = \infty \wedge beam \downarrow j_0 = \infty)\end{bmatrix} \\ \quad \mathbf{od}\end{array} \tag{5}$$

To refine the body of the repetition we introduce three local variables, *st*, *et* and *size* to store the start and finish times of the next object on the conveyor and the size of the object, respectively, and refine the body to the code given in Fig. 1. The details of these steps are similar to standard refinement steps and do not make use of repetitions; hence they are omitted here.

5.2. Refinement of procedure Await

The refinement of procedure *Await* introduces a local variable p and auxiliary variable *before* as shown in Fig. 2. For the repetition we make use of an invariant that relates the most recent sample of the value of *beam* to the time of occurrence of *event*:

$$\begin{aligned} I \;\widehat{=}\; & (p = val \Rightarrow event \leqslant \tau) \wedge (p \neq val \Rightarrow before \leqslant event) \wedge \\ & beam(\!| \{\!\!\{ \tau \ldots event \}\!\!\} |\!) \subseteq \{\neg\, val\} \wedge \\ & beam(\!| \{\!\!\{ event \ldots event + err \}\!\!\} |\!) = \{val\} \end{aligned}$$

Note that I is idle-invariant. If we factor out the initialisation of p and *before* to establish the invariant, and the setting of *pt* after the repetition (as shown in Fig. 2), the specification of the repetition is as follows.

$$\infty\ p, before\colon \left[I,\ (\tau < \infty \wedge event \leqslant \tau) \vee (\tau = \infty \wedge event_0 = \infty)\right]. \qquad (6)$$

We use Law 22 (repetition–terminating body) with a guard of $p \neq val$, invariant I above, relation R just *true*, and a deadline of $event + err$.

```
    (6)
 ⊑ do $p \neq val \rightarrow$
        deadline $event + err$;
        $p, before\colon [p \neq val \wedge \tau \leqslant event + err \wedge I,\ I]$
    od
```

This is a valid refinement provided

$$\begin{aligned} & (\tau < \infty \wedge p = val \wedge I) \vee (\tau = \infty \wedge I_\infty) \\ \Rrightarrow\, & (\tau < \infty \wedge event \leqslant \tau) \vee (\tau = \infty \wedge event_0 = \infty) \end{aligned}$$

The terminating case follows from the definition of I. For the nonterminating case I_∞ implies the following predicate.

$$\begin{aligned} & (\forall \tau' : Time \bullet (\exists \tau : Time;\ p : boolean;\ before : Time_\infty;\ event : Time_\infty \bullet \\ & \qquad \tau' \leqslant \tau \wedge p \neq val \wedge \tau \leqslant event + err \wedge I \wedge event = event_0)) \\ \Rrightarrow\, & (\forall \tau' : Time \bullet \tau' \leqslant event_0 + err) \\ \Rrightarrow\, & event_0 = \infty \end{aligned}$$

The refinement of the body of this repetition is standard and does not make use of any further repetitions so we omit the details here.

6. Timing constraint analysis

In order for compiled machine code to implement the machine-independent program it must guarantee to meet all the deadlines. The auxiliary variables and parameters

```
E :: $before := \tau$;
     $p$ : **read** ($beam$);
F :: **deadline** $event + err$;
     $[p \neq val]$;
     $\{(p = val \Rightarrow event \leqslant \tau) \wedge (p \neq val \Rightarrow before \leqslant event)\}$;
     **deadline** $event + err$;
E :: $before := \tau$;
     $p$ : **read** ($beam$);
F :: **deadline** $event + err$
```

Fig. 3. Repetition path in Await.

introduced above aid this analysis. There are three deadlines within the procedure *Await* (Fig. 2). The deadline at the start of the repetition is subsumed by the more stringent requirement of the deadline at (F). The deadline (F) within the repetition is reached initially from the entry to the procedure, and subsequently on each iteration. We defer analysis of the entry path to the analysis of the main program, because the context of the main program is necessary for the analysis. For an iteration we consider the path (shown in Fig. 3) that starts at the assignment to *before* (E), reads the value of *beam* into p, passes through the deadline (F), restarts the body of the repetition because p is not equal to val, performs the assignment to *before* (E), reads the value of *beam*, and reaches the deadline (F). The guard evaluation is represented by $[p \neq val]$, which indicates that in order for the path to be followed, p must not be equal to val at that point in the path. The initial time assigned to *before*, i.e., the time at which the path begins execution, must be before time *event* because the value of p was not equal to val, and the final deadline on the path is $event + err$. Hence, if the path is guaranteed to execute in less than time *err*, it will always meet its deadline. If this path is guaranteed to reach its deadline then any path with this as a suffix is also guaranteed to meet the final deadline. A similar analysis can be performed for the same path as in Fig. 3 but extended to exit the repetition because $p = val$, and read the current time into *pt*, before reaching the final deadline (G). The constraint on this path is also *err*.

The analysis of the main program has to take into account deadlines within the procedure calls. There is a path (shown in Fig. 4) that starts at (A) in Fig. 1. The path initialises j to 0, enters the repetition, allocates the local variables *st*, *et* and *size*, makes the first call to *Await* (B), and within *Await* allocates and assigns the local and auxiliary variables corresponding to the formal value parameters, allocates the local variable p, extends the auxiliary variables with *before*, initialises these, and follows the path into the repetition, ending at the deadline (F) of $event + err$. The last deadline is labelled $B.F$ to indicate that it is the deadline labelled F within the call to *Await* labelled B. The initial assertion guarantees the start time of the path is less than or equal to $beam \downarrow 0 - MinS$. For this call to *Await*, *event* is $beam \downarrow 0$ and hence the final deadline is $beam \downarrow 0 + err$. Therefore, a suitable constraint on the path is

$$beam \downarrow 0 + err - (beam \downarrow 0 - MinS) = MinS + err = 41\ \mathrm{ms}.$$

```
    A :: {$\tau \leqslant beam \downarrow 0 - MinS$};
         $j := 0$;
         [$true$];
         alloc var $st, et : natural$ ms; $size : natural$ mm;
    B :: $st \leftarrow Await(false, beam \downarrow j)$;
         alloc var $val : boolean$;
         alloc aux $event : Time_\infty$;
         $val, event := false, beam \downarrow j$;
         alloc var $p : boolean$;
         alloc aux $before : Time_\infty$;
         $before := event$;
         $p := \neg\, val$;
         [$p \neq val$];
         {$(p = val \Rightarrow event \leqslant \tau) \wedge (p \neq val \Rightarrow before \leqslant event)$};
         deadline $event + err$;
  B.E :: $before := \tau$;
         $p$ : read ($beam$);
  B.F :: deadline $event + err$;
```

Fig. 4. Initial path in main program.

If this path is guaranteed to execute in a time of less than 41 ms then the deadline is guaranteed to be reached.

The remaining timing paths are analysed in a similar manner. We briefly summarise them. Another path starts as for this one, continues on to exit the repetition on the first evaluation of its guard, and ends at deadline $B.G$. The timing constraint for this path is the same as the one above, for the same reasons. The path starting with the final deadline in the first call to *Await* ($B.G$) and ending at the deadline in the second call to *Await* ($C.F$) has a start time before $beam \downarrow j + err$ and a deadline of $beam \uparrow j + err$. Therefore a constraint that guarantees that the final deadline will be met is

$$(beam \uparrow j + err) - (beam \downarrow j + err) = beam \uparrow j - beam \downarrow j,$$

which from our assumptions is greater than $MinW$. Hence if the code on the path executes in less than time 20 ms, the deadline will be met.

The next deadline we consider is the one occurring in the repetition within the main program. It has a deadline of $beam \uparrow j + bin\_select$. The path begins at the final deadline ($C.G$). within the second call to *Await*, which has a deadline of $event + err$, where *event* for the second call is $beam \uparrow j$. The path exits the call, calculates *size*, and sets *lbin*, before reaching the deadline (D). A suitable constraint on this path that guarantees the final deadline will be met is

$$(beam \uparrow j + bin\_select) - (beam \uparrow j + err) = bin\_select - err = 9\,\text{ms}.$$

The final path we consider begins at the deadline (D) within the main loop, delays until $et + bin\_stable$, increments j, deallocates the local variables *st*, *et*, and *size*, iterates back to the start of the repetition, allocates the local variables *st*, *et*, and *size*, enters the first call to *Await* (B), and progresses down to the deadline ($B.F$) within the repetition. The final deadline is $event + err$ and for this case *event* is $beam \downarrow j$. However, along the path j has been incremented, and hence in terms of the initial value of j for the path the deadline is $beam \downarrow (j+1) + err$. Therefore, a suitable constraint is

$$\begin{aligned}&(beam \downarrow (j+1) + err) - (beam \uparrow j + bin\_select)\\ &\quad = (beam \downarrow (j+1) - beam \uparrow j) + err - bin\_select.\end{aligned}$$

From our initial assumptions $beam \downarrow (j+1) - beam \uparrow j$ is greater than or equal to $MinS$, and hence we can use the constant constraint

$$MinS + err - bin\_select = 31\,\mathsf{ms}.$$

In the above analyses we have assumed that the constants $MinS$, vel, etc., have been supplied with the specification. However, an alternative approach allows them to be treated symbolically and some of them determined by a combination of the other constants and the worst-case execution times for the paths in the program. For example, the velocity of the conveyor belt, vel, determines $MinS$, $MinW$ and err in terms of the minimum separation distance between objects, minimum length of objects, and the error margin, $mrgn$, respectively. Hence, an alternative approach is to vary the velocity of the conveyor belt to ensure that all deadlines within the program are met.

7. Proof of general repetition law

In this section we give a proof of Law 21 (repetition). The other repetition laws given earlier are simple corollaries of this law. As abbreviations we introduce S to stand for the left side of the refinement in Law 21 (repetition), i.e.,

$$S \mathrel{\widehat{=}} \infty\ x\colon \left[I,\ \begin{array}{l}(\tau < \infty \wedge \neg\, B @ \tau \wedge I \wedge R^*) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* \mathbin{⨟} Q)))\end{array} \right]$$

and $ITER_L$ to stand for $ITER$ with C instantiated to the body of the repetition in the law:

$$\begin{array}{l} ITER_L \mathrel{\widehat{=}} |[\mathbf{aux}\, u : Time \bullet u := \tau;\ [B @ \tau];\ \mathbf{deadline}\, D; \\ \qquad\qquad \infty\ x\colon \left[B @ \tau \wedge \tau \leqslant D @ \tau \wedge I,\quad \begin{array}{l}(\tau < \infty \wedge I \wedge R) \vee \\ (\tau = \infty \wedge Q)\end{array} \right]; \\ \qquad\qquad [u + d \leqslant \tau] \\ \qquad]| \end{array}$$

From Definition 15 (repetition) we need to show

$$\begin{aligned}& S \sqsubseteq [\!]\, \{d : Time \mid 0 < d \bullet (ITER_L^*; [\neg\, B @ \tau] \,[\!]\, ITER_L^\infty)\}\\ \equiv\ & \text{Law 8 (general choice)}\\ & (\forall d : Time \mid 0 < d \bullet (S \sqsubseteq (ITER_L^*; [\neg\, B @ \tau] \,[\!]\, ITER_L^\infty)))\\ \equiv\ & \text{Law 6 (choice)}\\ & (\forall d : Time \mid 0 < d \bullet (S \sqsubseteq ITER_L^*; [\neg\, B @ \tau]) \wedge (S \sqsubseteq ITER_L^\infty))\end{aligned}$$

We divide our proof into two major components showing, respectively, the two refinements of S for all strictly positive times, d.

7.1. A finite number of iterations

We would like to show

$$S \sqsubseteq ITER_L^*; [\neg\, B @ \tau]. \tag{7}$$

Our first step is to factor out the negation of the guard from the effect of S so that both sides of refinement (7) end with the negation of the guard.

$$\begin{array}{l} \quad S \\ \sqsubseteq \text{Law 28 (separate post guard) in Appendix A} \\ \quad T; [\neg\, B @ \tau], \end{array}$$

where $T \mathrel{\widehat=} \infty\, x\colon \left[I, \begin{array}{l} (\tau < \infty \wedge I \wedge R^*) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* \fatsemi Q))) \end{array} \right].$

By monotonicity of refinement, to show (7) it is sufficient to show the following.

$$\begin{array}{l} \quad T \sqsubseteq ITER_L^* \\ \equiv \text{Definition 13 (finite iterations)} \\ \quad T \sqsubseteq \, [] \{ i : \mathbb{N} \bullet ITER_L^i \} \\ \equiv \text{Law 8 (general choice)} \\ \quad (\forall i : \mathbb{N} \bullet T \sqsubseteq ITER_L^i) \end{array}$$

At this stage in order to simplify the proof we introduce an abstraction, IT, of $ITER_L$ that is sufficient to show the above refinement, where

$$IT \mathrel{\widehat=} \infty\, x\colon \left[\begin{array}{l} \tau_0 + d \leqslant \tau \;\wedge \\ \left(I[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l} (B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\tfrac{\rho.v_0}{\rho.v}] \;\wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)) \end{array} \right) \right) \end{array} \right].$$

Lemma 26 (below) shows that $IT \sqsubseteq ITER_L$. Hence by monotonicity of refinement, all we need to prove is

$$(\forall i : \mathbb{N} \bullet T \sqsubseteq IT^i),$$

which we show by induction. Case $i = 0$ reduces to showing $T \sqsubseteq$ **skip**. The equivalent specification command to **skip** has the effect $\tau_0 = \tau$. By Law 27 (strengthen effect) given in Appendix A, it is sufficient to show

$$\begin{array}{l} \tau_0 < \infty \wedge I[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \wedge \tau_0 = \tau \wedge eq(\rho.out, \tau_0, \tau, \rho.v_0, \rho.v) \Rightarrow \\ \tau < \infty \wedge I \wedge R^* \end{array}$$

which follows because both the following hold.

$$\begin{array}{l} \tau_0 < \infty \wedge I[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \wedge \tau_0 = \tau \wedge eq(\rho.out, \tau_0, \tau, \rho.v_0, \rho.v) \Rightarrow I, \\ \tau_0 < \infty \wedge \tau_0 = \tau \wedge eq(\rho.out, \tau_0, \tau, \rho.v_0, \rho.v) \Rightarrow R^0 \Rightarrow R^*. \end{array}$$

For the inductive step we assume that $T \sqsubseteq IT^n$ holds for n a natural number, and are required to show $T \sqsubseteq IT^{n+1}$. We start from the right side.

$$\begin{array}{l}
\quad IT^{n+1} \\
\sqsubseteq \text{ Definition 13 (finite iterations)} \\
\quad IT^n; IT \\
\sqsupseteq \text{ inductive assumption} \\
\quad T; IT \\
\sqsubseteq \ T; \infty\ x\colon \left[\begin{array}{l} \tau_0 + d \leqslant \tau\ \wedge \\ \left(I[\frac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l}(B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\frac{\rho.v_0}{\rho.v}]\ \wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q))\end{array}\right)\right)\end{array}\right] \\
\sqsupseteq \text{ definition of } T\text{; Law 27 (strengthen effect)} \\
\quad \infty\ x\colon \left[I,\ \begin{array}{l}(\tau < \infty \wedge I \wedge R^*) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* ⨾ Q)))\end{array}\right]; \\
\quad \infty\ x\colon [I,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)] \\
\sqsupseteq \text{ Law 12 (sequential composition with relation)} \\
\quad \infty\ x\colon \left[I,\ \begin{array}{l}(\tau < \infty \wedge I \wedge (R^* ⨾ R)) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* ⨾ Q) \vee (R^* ⨾ Q)))\end{array}\right] \\
\sqsupseteq \text{ as } (R^* ⨾ R) \Rrightarrow R^* \\
\quad T
\end{array}$$

Before proceeding to the infinite iterations case we validate that IT is a valid abstraction of $ITER_L$.

Lemma 26. *Given an idle-stable, boolean-valued expression, B; an idle-stable, time-valued expression, D; a time-valued constant, d; an idle-invariant, single-state predicate, I; a pre-idle-invariant relation, Q; and a pre- and post-idle-invariant relation, R,*

$$IT \sqsubseteq ITER_L$$

Proof. We begin from the definition of IT.

$$\infty\ x\colon \left[\begin{array}{l} \tau_0 + d \leqslant \tau\ \wedge \\ \left(I[\frac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l}(B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\frac{\rho.v_0}{\rho.v}] \wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q))\end{array}\right)\right)\end{array}\right]$$

$\sqsubseteq$ introduce auxiliary u to capture start time

$$\begin{array}{l}
|[\ \mathbf{aux}\, u : \mathit{Time} \bullet u := \tau; \\
\quad \infty\ x\colon \left[\begin{array}{l} u + d \leqslant \tau\ \wedge \\ \left(I[\frac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l}(B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\frac{\rho.v_0}{\rho.v}] \wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q))\end{array}\right)\right)\end{array}\right] \\
]|.
\end{array} \tag{8}$$

We separate the final delay.

$$\begin{array}{ll}
& (8) \\
\sqsubseteq & \text{Law 28 (separate post-guard)} \\
& \infty\ x\colon \left[I[\frac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l} (B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\frac{\rho.v_0}{\rho.v}] \wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)) \end{array} \right) \right]; \qquad (9) \\
& [u + d \leqslant \tau]
\end{array}$$

Next we factor out the guard and the deadline.

$$\begin{array}{ll}
& (9) \\
\sqsubseteq & \text{equivalent specification} \\
& \infty\ x\colon \left[I, \begin{array}{l} (B @ \tau_0 \wedge \tau_0 \leqslant D @ \tau_0)[\frac{\rho.v_0}{\rho.v}] \wedge \\ ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)) \end{array} \right] \\
\sqsubseteq & \text{Law 29 (separate pre guard) in Appendix A} \\
& [B @ \tau \wedge \tau \leqslant D @ \tau]; \qquad (10) \\
& \infty\ x\colon [B @ \tau \wedge \tau \leqslant D @ \tau \wedge I,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)] \qquad (11)
\end{array}$$

We separate out the deadline as a post-guard. Because the deadline appears after the guard evaluation, it also implies that start time of the guard evaluation must be before the deadline as required in the original specification.

$$\begin{array}{ll}
& (10) \\
\sqsubseteq & \text{Law 28 (separate post guard)} \qquad (12) \\
& [B @ \tau]; [\tau \leqslant D @ \tau].
\end{array}$$

The upper bound on the start time of the repetition body (12) is implemented by **deadline** D. Combining the above refinement steps proves the lemma. □

7.2. An infinite number of iterations

To complete the proof of Law 21 (repetition) we need to show $S \sqsubseteq ITER_L^{\infty}$. Again we can make use of Lemma 26 and monotonicity to reduce the task to showing $S \sqsubseteq IT^{\infty}$. Because we are dealing with the infinite iteration case, we start by refining S as follows.

$$\begin{array}{ll}
& \infty\ x\colon \left[I, \begin{array}{l} (\tau < \infty \wedge \neg\, B @ \tau \wedge I \wedge R^*) \vee \\ (\tau = \infty \wedge ((I_\infty \wedge R^\infty) \vee (R^* \fatsemi Q))) \end{array} \right] \\
\sqsubseteq & \text{Law 27 (strengthen effect)} \\
& \infty\ x\colon [I,\ \tau = \infty \wedge I_\infty \wedge R^\infty]
\end{array}$$

Now we show that this refines to IT^{∞} using Definition 14 (infinite iterations). First, we note that $\tau_0 < \infty \wedge \mathscr{M}_\rho(IT) \Rightarrow \tau_0 + d \leqslant \tau$, where $0 < d$, and hence the definition is applicable.

$$
\begin{array}{l}
\quad \mathscr{M}_\rho\,(IT^\infty) \\
\equiv \text{Definition 14 (infinite iterations)} \\
\quad \tau_0 \leqslant \tau \wedge (\tau_0 < \infty \Rightarrow \tau = \infty \ \wedge \\
\qquad (\exists \vec{t} : \mathbb{N} \to \mathit{Time};\ \vec{v} : \mathbb{N} \to T_v \bullet \vec{t}_0 = \tau_0 \wedge \vec{v}_0 = \rho.v_0 \ \wedge \\
\qquad\quad (\forall i : \mathbb{N} \bullet \mathscr{M}_\rho\,(IT_{i+1}))))
\end{array}
$$

Let us concentrate on the universally quantified term. We extend the abbreviation IT_{i+1}, given with Definition 14 (infinite iterations), to apply to predicates as well as commands. For example, $I_i = I[\vec{t}_{i-1}, \vec{t}_i, \vec{v}_{i-1}, \vec{v}_i / \tau_0, \tau, \rho.v_0, \rho.v]$, which because τ_0 and $\rho.v_0$ do not appear in I reduces to $I[\vec{t}_i, \vec{v}_i / \tau, \rho.v]$.

$$
\begin{array}{l}
\quad \forall i : \mathbb{N} \bullet \mathscr{M}_\rho\,(IT_{i+1}) \\
\equiv \forall i : \mathbb{N} \bullet \vec{t}_i \leqslant \vec{t}_{i+1} \wedge (\vec{t}_i < \infty \Rightarrow \vec{t}_i + d \leqslant \vec{t}_{i+1} \ \wedge \\
\qquad \left(I_i \Rightarrow \left(\begin{array}{l} B_i @ \vec{t}_i \wedge \vec{t}_i \leqslant D_i @ \vec{t}_i \ \wedge \\ ((\vec{t}_{i+1} < \infty \wedge I_{i+1} \wedge R_{i+1}) \vee (\vec{t}_{i+1} = \infty \wedge Q_{i+1})) \end{array} \right) \right) \wedge \\
\qquad eq(\rho.out \setminus x, \vec{t}_i, \vec{t}_{i+1}, \vec{v}_i \setminus x_i, \vec{v}_{i+1} \setminus x_{i+1}))
\end{array}
$$

All the $\vec{t}_i$ terms are finite (from their type), and hence their comparisons with infinity can be simplified.

$$
\begin{array}{l}
\forall i : \mathbb{N} \bullet \vec{t}_i + d \leqslant \vec{t}_{i+1} \ \wedge \\
\qquad (I_i \Rightarrow B_i @ \vec{t}_i \wedge \vec{t}_i \leqslant D_i @ \vec{t}_i \wedge I_{i+1} \wedge R_{i+1}) \ \wedge \\
\qquad eq(\rho.out \setminus x, \vec{t}_i, \vec{t}_{i+1}, \vec{v}_i \setminus x_i, \vec{v}_{i+1} \setminus x_{i+1})
\end{array}
$$

For all i, $I_i \Rightarrow I_{i+1}$. Hence provided I_0 holds, I_i will hold for all i. However, from the context $I_0 = I[\vec{t}_0, \vec{v}_0 / \tau, \rho.v] = I[\tau_0, \rho.v_0 / \tau, \rho.v]$. Therefore, the above implies the following.

$$
I[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\forall i : \mathbb{N} \bullet \begin{array}{l} \vec{t}_i + d \leqslant \vec{t}_{i+1} \wedge B_i @ \vec{t}_i \wedge \vec{t}_i \leqslant D_i @ \vec{t}_i \wedge I_i \ \wedge \\ R_{i+1} \wedge eq(\rho.out \setminus x, \vec{t}_i, \vec{t}_{i+1}, \vec{v}_i \setminus x_i, \vec{v}_{i+1} \setminus x_{i+1}) \end{array} \right)
$$

Because R_{i+1} holds for all i, we can deduce that R^∞ holds overall, and that the transitive closure of R holds between the initial state and all the intermediate states, i.e., $R^*[\vec{t}_i, \vec{v}_i / \tau, \rho.v]$ holds for all i. Similarly, the stability of the outputs over all the adjacent intervals implies that the outputs are stable from the initial time up to every time $\vec{t}_i$, and the local and auxiliary variables that are not in the frame are at each step equal to their initial values.

$$
I[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \Rightarrow \left(\begin{array}{l} \forall i : \mathbb{N} \bullet \left(\begin{array}{l} \vec{t}_i + d \leqslant \vec{t}_{i+1} \wedge B_i @ \vec{t}_i \wedge \vec{t}_i \leqslant D_i @ \vec{t}_i \ \wedge \\ I_i \wedge R^*[\tfrac{\vec{t}_i, \vec{v}_i}{\tau, \rho.v}] \wedge \\ \qquad eq(\rho.out \setminus x, \tau_0, \vec{t}_i, \rho.v_0 \setminus x_0, \vec{v}_i \setminus x_i) \end{array} \right) \wedge \\ R^\infty \wedge eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \end{array} \right)
$$

The above implies for each i that $\tau_0 + i * d \leqslant \vec{t_i}$, and hence because d is strictly positive, for any time τ' there exists an index i such that $\tau' \leqslant \vec{t_i}$.

$$I[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \Rightarrow \left(\begin{array}{l} \forall \tau' : \mathit{Time} \bullet \exists i : \mathbb{N} \bullet \\ \qquad \left(\begin{array}{l} \tau' \leqslant \vec{t_i} \wedge B_i @ \vec{t_i} \wedge \vec{t_i} \leqslant D_i @ \vec{t_i} \wedge I_i \wedge R^*[\tfrac{\vec{t_i},\vec{v_i}}{\tau,\rho.v}] \\ \wedge\ eq(\rho.out \setminus x, \tau_0, \vec{t_i}, \rho.v_0 \setminus x_0, \vec{v_i} \setminus x_i) \end{array} \right) \wedge \\ R^\infty \wedge eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \end{array} \right)$$

The elements $\vec{t_i}$ and $\vec{v_i}$ provide witnesses for the existentially quantified τ and $\rho.v$ in the following.

$$I[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \Rightarrow \left(\begin{array}{l} \forall \tau' : \mathit{Time} \bullet \exists \tau : \mathit{Time}; \rho.v : T_v \bullet \\ \qquad \left(\begin{array}{l} \tau' \leqslant \tau \wedge B @ \tau \wedge \tau \leqslant D @ \tau \wedge I \wedge R^* \wedge \\ eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \end{array} \right) \wedge \\ R^\infty \wedge eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \end{array} \right)$$

Placing this back in the original context and noting the definition of I_∞, we can complete the proof of the infinite number of iterations case.

$$\begin{array}{l} \mathscr{M}_\rho(IT^\infty) \\ \Rrightarrow \tau_0 \leqslant \tau \wedge \left(\left(\begin{array}{l} \tau_0 < \infty \wedge \\ I[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \end{array} \right) \Rightarrow \left(\begin{array}{l} \tau = \infty \wedge I_\infty \wedge R^\infty \wedge \\ eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \end{array} \right) \right) \\ \equiv \mathscr{M}_\rho(\infty\ x\colon\ [I,\ \tau = \infty \wedge I_\infty \wedge R^\infty]). \end{array}$$

8. Conclusions

Nonterminating repetitions are commonly required in real-time control applications. Hence, a real-time program development method needs to support their use. The primary advantage of the approach taken in this paper is that we develop code for a *machine-independent* real-time programming language, and hence do not need to consider the detailed execution times of language constructs as part of the development process. This is achieved through the simple mechanism of adding a deadline command to our programming language. The approach allows the real-time calculus to appear to be a straightforward extension of the standard refinement calculus [4]. Of course, the compilation process now has the added burden of checking that the deadlines are met [3].

As with the standard refinement calculus, it is advantageous to devise refinement laws that make use of loop invariants and relations [18]. We have developed a general refinement law for introducing repetitions that encompasses all of our previous laws as special cases. In addition, the use of the relations within the law allows the application of the law to be considerably simpler, as illustrated in the development of the main program repetition in the example in Section 5.

In order to reason about repetitions in a machine-independent manner, we require that the loop invariant be idle-invariant, so that it holds over the executions of the guard evaluation and branch back phases of repetition execution. This restricts the form of the invariant and blurs the link between the invariant and the current time variable, τ. To reestablish the link between the invariant and the time at which the invariant is true, a deadline command can be added to the start of the repetition body. In a similar manner the relations used in the rule need to be pre- and post-idle-invariant in order to cope with the time periods corresponding to guard evaluation and branch back.

The infinite number of iterations case is the most interesting to deal with. If the repetition body always terminates and establishes a relation R, then there is an infinite sequence of states with each adjacent pair in the sequence related by R. Hence for the whole repetition one can deduce the overall effect R^{∞}. In addition, the loop invariant, the guard and the deadline condition are true at an infinite number of progressively increasing times. That leads to our final refinement law for nonterminating repetitions.

We have presented a predicative semantics for our real-time language, and within that framework given a simpler relational-style semantics for possibly nonterminating repetitions than the semantics given in our earlier paper [6]. In the semantics of the repetition we considered two cases: a finite number of iterations of the repetition (including the case where the last iteration fails to terminate), and an infinite number of iterations. The relational-style semantics leads to a considerably simpler proof of the refinement law.

Acknowledgements

This research was funded by Australian Research Council (ARC) Large Grant A49801500, *A Unified Formalism for Concurrent Real-time Software Development*. I would like to thank Colin Fidge, Karl Lermer, Mark Utting and the excellent (but anonymous) referees for feedback on earlier drafts of this paper, and the members of IFIP Working Group 2.3 on Programming Methodology for feedback on this topic, especially Rick Hehner for his advice on how to simplify our approach, and Cliff Jones on his rules that make use of relations.

Appendix A. Additional laws

Law 27 (Strengthen effect). *Provided*

$$\tau_0 < \infty \wedge P[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}] \wedge \tau_0 \leqslant \tau \wedge eq(\rho.out \setminus x, \tau_0, \tau, \rho.v_0 \setminus x_0, \rho.v \setminus x) \wedge Q' \Rrightarrow Q$$

then $\infty\ x\colon [P,\ Q] \sqsubseteq \infty\ x\colon [P,\ Q']$.

The proof of this law follows directly from Definition 1 (real-time specification).

Law 28 (Separate post guard). *Provided P and X are single-state predicates, I is a single-state, idle-invariant predicate, R is a post-idle-invariant relation, and Q is a* *relation*,

$$\begin{array}{l} \quad \infty\ x\colon [P,\ (\tau < \infty \wedge X \wedge I \wedge R) \vee (\tau = \infty \wedge Q)] \\ \sqsubseteq \infty\ x\colon [P,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)];\ [X]. \end{array}$$

Proof. Because R is post-idle-invariant composing it with the *IDLE* relation has no effect: $\tau < \infty \Rrightarrow (R \fatsemi \mathit{IDLE}) \Rightarrow R$. Therefore, the left side of the law is refined by the following:

$$\begin{array}{l} \quad \infty\ x\colon \left[P,\ \begin{array}{l}(\tau < \infty \wedge X \wedge I \wedge (R \fatsemi \mathit{IDLE})) \vee \\ (\tau = \infty \wedge (Q \vee (R \fatsemi \mathit{false})))\end{array}\right] \\ \sqsubseteq \text{Law 12 (sequential composition with relation)} \\ \quad \infty\ x\colon [P,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)]; \\ \quad \infty\ x\colon [I,\ (\tau < \infty \wedge X \wedge I \wedge \mathit{IDLE}) \vee (\tau = \infty \wedge \mathit{false})]. \end{array} \tag{A.1}$$

The second component can be refined to a guard.

$$\begin{array}{l} \quad \text{(A.1)} \\ \sqsubseteq \text{contract frame; } \tau < \infty \text{ implies it terminates} \\ \quad [I, X \wedge I \wedge \mathit{IDLE}] \\ \sqsubseteq \text{as } I \text{ is idle-invariant; empty frame implies } \mathit{IDLE}\text{; weaken assumption} \\ \quad [X]. \end{array}$$

Law 29 (Separate pre-guard). *Provided P is a single-state, idle-invariant predicate, I is a single-state predicate, Q and R are pre-idle-invariant relations, and X is a single-state predicate such that*

$$\tau_0 < \infty \wedge \mathit{IDLE} \wedge X \Rrightarrow X[\tfrac{\tau_0, \rho.v_0}{\tau, \rho.v}]$$

then

$$\begin{array}{l} \quad \infty\ x\colon \left[P,\ X[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \wedge ((\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q))\right] \\ \sqsubseteq [X]; \infty\ x\colon [P \wedge X,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)]. \end{array}$$

Proof. The left side of the above is refined by the following because R and Q are pre-idle-invariant.

$$\begin{array}{l} \quad \infty\ x\colon \left[P,\ \begin{array}{l}\left(\tau < \infty \wedge I \wedge \left(\left(X[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \wedge \mathit{IDLE}\right) \fatsemi R\right)\right) \vee \\ \left(\tau = \infty \wedge \left(\mathit{false} \vee \left(\left(X[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \wedge \mathit{IDLE}\right) \fatsemi Q\right)\right)\right)\end{array}\right] \\ \sqsubseteq \text{Law 12 (sequential composition with relation)} \\ \quad \infty\ x\colon \left[P,\ \begin{array}{l}\left(\tau < \infty \wedge P \wedge X \wedge X[\tfrac{\tau_0,\rho.v_0}{\tau,\rho.v}] \wedge \mathit{IDLE}\right) \vee \\ (\tau = \infty \wedge \mathit{false})\end{array}\right]; \\ \quad \infty\ x\colon \left[P \wedge X,\ (\tau < \infty \wedge I \wedge R) \vee (\tau = \infty \wedge Q)\right]. \end{array} \tag{A.2}$$

The first component can be refined to a guard.

$$
\begin{array}{l}
\quad \text{(A.2)} \\
\sqsubseteq \text{ contract frame; } \tau < \infty \text{ implies it terminates; assumption about } X \\
\quad [P, P \wedge X \wedge \mathit{IDLE}] \\
\sqsubseteq \; P \text{ idle-invariant; weaken assumption; empty frame implies } \mathit{IDLE} \\
\quad [X].
\end{array}
$$

References

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

[2] R.W. Floyd, Assigning meaning to programs, Math. Aspects of Comput. Sci. 19 (1967) 19–32.

[3] S. Grundon, I.J. Hayes, C.J. Fidge, Timing constraint analysis, in: C. McDonald (Ed.), Computer Science '98: Proc. 21st Australasian Computer Sci. Conf. (ACSC'98), Perth, 4–6 Feb., Springer, 1998, pp. 575–586.

[4] I.J. Hayes, Separating timing and calculation in real-time refinement, in: J. Grundy, M. Schwenke, T. Vickers (Eds.), Int. Refinement Workshop and Formal Methods Pacific 1998, Springer, 1998, pp. 1–16.

[5] I.J. Hayes, Real-time program refinement using auxiliary variables, in: M. Joseph (Ed.), Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, Vol. 1926, Springer, 2000, pp. 170–184.

[6] I.J. Hayes, Reasoning about non-terminating loops using deadline commands, in: R. Backhouse, J.N. Oliveira (Eds.), Proc. Mathematics of Program Construction, Lecture Notes in Computer Science, Vol. 1837, Springer, 2000, pp. 60–79.

[7] I.J. Hayes, A predicative semantics for real-time refinement, Technical Report 01-15, Software Verification Research Centre, The University of Queensland, Brisbane 4072, Australia, May 2001.

[8] I.J. Hayes, B.P. Mahony, Using units of measurement in formal specifications, Formal Aspects of Computing 7 (3) (1995) 329–347.

[9] I.J. Hayes, M. Utting, Coercing real-time refinement: a transmitter, in: D.J. Duke, A.S. Evans (Eds.), BCS-FACS Northern Formal Methods Workshop (NFMW'96), Springer, 1997.

[10] 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), Chapman and Hall, 1998, pp. 186–204.

[11] I.J. Hayes, M. Utting, A sequential real-time refinement calculus, Acta Informatica 37 (6) (2001) 385–448.

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

[13] E.C.R. Hehner, A Practical Theory of Programming, Springer, 1993.

[14] C.A.R. Hoare, An axiomatic approach to computer programming, Comm. ACM 12 (1969) 576–580, 583.

[15] C.A.R. Hoare, He Jifeng, Unifying Theories of Programming, Prentice-Hall, 1998.

[16] J. Hooman, Assertional specification and verification, in: M. Joseph (Ed.), Real-time Systems: Specification, Verification and Analysis, Prentice-Hall, 1996, pp. 97–146 (chapter 5).

[17] J. Hooman, O. van Roosmalen, Formal design of real-time systems in a platform-independent way, Parallel and Distributed Computing Practices 1 (2) (1998) 15–30.

[18] C.B. Jones, Program specification and verification in VDM, Technical Report UMCS-86-10-5, Department of Computer Science, University of Manchester, 1986.

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

[20] 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.