Deriving real-time action systems with multiple time bands using algebraic reasoning

Brijesh Dongol[1]

Department of Computer Science,
The University of Sheffield S1 4DP, UK

Ian J. Hayes[2]

School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

John Derrick[1]

Department of Computer Science,
The University of Sheffield S1 4DP, UK

Abstract

The verify-while-develop paradigm allows one to incrementally develop programs from their specifications using a series of calculations against the remaining proof obligations. This paper presents a derivation method for real-time systems with realistic constraints on their behaviour. We develop a high-level interval-based logic that provides flexibility in an implementation, yet allows algebraic reasoning over multiple granularities and sampling multiple sensors with delay. The semantics of an action system is given in terms of interval predicates and algebraic operators to unify the logics for an action system and its properties, which in turn simplifies the calculations and derivations.

1. Introduction

Modern cyber-physical systems are implemented using a digital controller that executes by sampling the various system sensors, performing some computation, then signalling the components being controlled to change their behaviour in accordance with the system requirements. This paper presents methods to formally derive controllers from the system specifications, where the controllers periodically sample the environment and signal various components when necessary. We use a logic for reasoning about complex systems with events in different time granularities, e.g., sampling events for different components may occur at different rates, and these rates can also depend on the properties of the system being measured. The components being controlled can also operate at different time granularities, e.g., the effect of a motor reaching operating speed may occur in a different time band than the effect of a switch that powers on a motor.

The derivation method builds on our method of enforced properties [15, 16, 17, 19], which uses the verify-while-develop paradigm to incrementally obtain program code from the underlying specifications. Our framework incorporates a logic of time bands [9, 10, 49], which allows one to formalise properties at different time granularities and define relationships between these properties. Behaviours of components at fine levels of granularity often involve interactions that may not necessarily be observed when assuming a

Email addresses: B.Dongol@sheffield.ac.uk (Brijesh Dongol), Ian.Hayes@itee.uq.edu.au (Ian J. Hayes), J.Derrick@dcs.shef.ac.uk (John Derrick)

[1]Brijesh Dongol and John Derrick are supported by EPSRC Grant EP/J003727/1.

[2]Ian Hayes is supported by ARC Discovery Grant DP130102901.

coarse level of atomicity. Development of a system assuming coarse-grained atomicity can be problematic if the atomicity assumptions cannot be realised by the system under development, causing the developed system to become invalid. On the other hand, consideration of fine-grained interactions results in an increase in the complexity of the reasoning. In this paper, we use a high-level logic that allows one to describe the observable states that may occur when sampling variables at finer time-bands [10, 19, 21, 30].

We present our methods using the action systems framework, which has been used as a basis for several theories of program refinement [3, 7, 4, 5]. In its simplest form, an action system consists of a set of actions (i.e., guarded statements) and a loop that at each iteration non-deterministically chooses then executes an enabled action from the set of actions. Thus, periodic sampling is naturally supported by the framework.

To model the behaviour of a program in an environment one may include actions corresponding to the program and its environment within a single action system [7, 17] so that the actions corresponding to the controller and its environment are interleaved with each other. However, in the context of real-time reactive systems, this model turns out to be problematic because for example it is unable to properly address *transient properties* [19, 20]. Such properties only hold for a brief amount of time, say an attosecond, and hence, a real-world implementation would never be able to reliably detect the property. A theoretical model that considers interleaving of controller and environment actions would require that the implementation does detect a transient property, which is unrealistic. Instead, an implementation should be allowed to ignore transient properties because they cannot be reliably detected. In this paper, like [19, 20], we modify the semantics so that an action system executes with its environment in a truly concurrent manner. This allows one to develop a theoretical model that properly addresses transient properties — an implementation is only required to handle non-transient properties.

This paper adds to our series of papers on program derivation [16, 17, 19, 20, 24, 25]. Of these, [16, 24, 25] consider concurrent program derivation and [17, 19, 20] consider real-time programs. Our papers [17, 19, 20] increasingly consider more realistic assumptions in concurrent real-time systems and the most advanced of these [19] allows one to consider sampling issues and components that operate over multiple time granularities. However, the framework itself has become increasingly complex and becomes a bottleneck to achieving scalable derivations because program properties are expressed in an interval-based LTL-style [40] logic, whereas the requirements are expressed using interval predicates. As a result, a derivation step is required to transform the interval predicate requirements to the level of the program.

In this paper, we remove this bottleneck by defining a semantics for action systems using algebraic operators in the style of Back and von Wright [6]. However, unlike Back and von Wright, we address real-time issues by basing our semantics on an algebra for interval predicates [22, 35, 36]. This allows one to improve uniformity across the model by enabling one to use interval predicates to express system requirements as well as program behaviour. Thus, the additional interval-based LTL logic from [19, 20] is completely avoided, and all proofs are carried out at the level of interval predicates.

To enable compositionality, we use rely/guarantee-style reasoning [12, 31, 38, 39], where the *rely* condition is an interval predicate that specifies the behaviour of the environment. Unlike Jones, [12, 38] who defines rely-guarantee reasoning in a relational setting, we asume that rely conditions are interval predicates that may specify real-time behaviour [21]. The underlying theory also includes methods for reasoning about delays and feedback.

This paper is structured as follows. In Section 2, we present a motivating example consisting of two pumps and two water tanks. In Section 3, we present our background theory and in Section 4, we present methods for reasoning about multiple time bands and sampling of multiple sensors. In Section 5 we present a novel algebraic semantics for action systems, which includes constructs for reasoning about enforced properties and rely conditions. We also discuss action system refinement. We use the theory from the earlier sections to derive a controller for our motivating example in Section 6. We consider some related work in Section 7 and present some concluding thoughts in Section 8.

2. Example: A two-pump system

Throughout this paper, we consider a system consisting of two water tanks $Tank_1$, $Tank_2$ and two pumps $Pump_1$, $Pump_2$ depicted in Fig. 1 (also see [1]). The environment (of the system) adds water to $Tank_1$ and

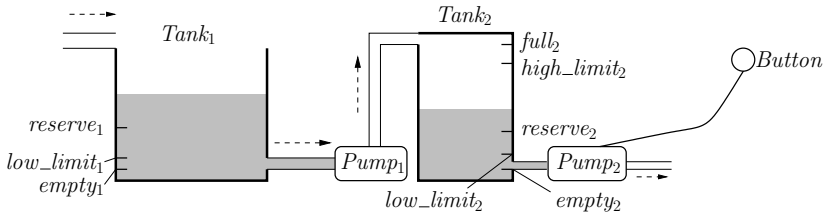

Figure 1: Two-pump system

does not affect $Tank_2$. We assume that $Tank_1$ is allowed to overflow, but $Tank_2$ is not. $Pump_1$ removes water from $Tank_1$ and fills $Tank_2$. $Pump_2$ only operates if *Button* is pressed and removes water from $Tank_2$. Aichernig *et al.* [1] describe the following requirements. We have adapted their informal specification to clarify the input/output behaviours of the pumps and to better distinguish safety (**S1**, **S2** and **S3**) and progress (**P1**, **P2** and **P3**) properties. Note that a progress property to turn $Pump_1$ off is not needed because it is implied by safety properties **S1** and **S2**.

S1. Whenever $water_1$ (the water level in $Tank_1$) is $empty_1$ or below, $Pump_1$ must be stopped.

S2. Whenever $water_2$ (the water level in $Tank_2$) is $full_2$ or above, $Pump_1$ must be stopped.

S3. Whenever $water_2$ is $empty_2$ or below, $Pump_2$ must be stopped.

P1. If $water_1$ is above $reserve_1$ ($Tank_1$ has enough water) and $water_2$ is below $reserve_2$ ($Tank_2$ is about to run dry), then $Pump_1$ must *eventually* be turned on.

P2. If *Button* is pressed and $water_2$ is above $reserve_2$ ($Tank_2$ has enough water), then $Pump_2$ must *eventually* be turned on.

P3. If *Button* is released, then $Pump_2$ must *eventually* be turned off.

Thus, we must keep track of water levels $reserve_1$ and $empty_1$ in $Tank_1$ and $full_2$, $reserve_2$, $empty_2$ in $Tank_2$. For $i \in \{1, 2\}$, we distinguish between signal on_i that starts/stops $Pump_i$, and $mode_i$ that determines the state of the $Pump_i$, e.g., $mode_i = running$ and $mode_i = stopped$ hold iff $Pump_i$ is physically running and stopped, respectively. To simplify the presentation we define

$$\begin{aligned} Running_i &\;\widehat{=}\; mode_i = running \\ Stopped_i &\;\widehat{=}\; mode_i = stopped \end{aligned}$$

Note that $Pump_i$ may also be associated with other modes such as *starting*, *stopping*, *offline*, etc.

A (digital) controller for $Pump_2$ must sample both the water level in $Tank_2$ and the state of *Button*, perform some processing, then send on/off signals to $Pump_2$ if necessary. Each of these phases takes time. Furthermore, the components operate at different time granularities and hence have different notions of precision (the amount of time that may be regarded as instantaneous [9, 10]). For example, $water_1$ may have a precision of 30 seconds (i.e., there is no significant change in the water level in $Tank_1$ within 30 seconds) and $Pump_2$ turns on/off with precision 1 second (i.e., it takes $Pump_2$ at most 1 second to reach its operating speed or to come to a stop). Formally reasoning about the system in a manner that properly addresses each of these timing aspects is complicated [18, 20, 29]. To reduce the complexity of the reasoning, formal frameworks often simplify specifications by assuming that certain aspects of the system (e.g., sampling) are instantaneous or take a negligible amount of time. However, it is well-known that such simplifications can cause complications during implementation. In particular, the developed specifications become unimplementable because their timing requirements cannot be satisfied by any real system [29, 51].

3. A logic for multiscale real-time specifications

To enable reasoning about truly concurrent behaviour, our framework uses an interval-based logic, which we present in Section 3.1. Chop and iteration, which are used to model sequential composition and loops, respectively, are presented in Section 3.2, and operators on interval predicates are presented in Section 3.3. In Section 3.4 we present methods for reasoning about properties over the actual states of a system. Variations of this theory appear in several of our papers [18, 19, 20, 21]. We note that there are some changes to the notation from these earlier papers, which will be explained where necessary. In Section 3.5 we present methods for identifying Zeno-like behaviours.

3.1. Intervals and interval predicates

Interval-based reasoning was first proposed by Moszkowski for discrete systems [43] as the Interval Temporal Logic, then extended by Zhou and Hansen [52] for continuous systems as the Duration Calculus. We have developed a variation of this theory that addresses the difficulties of reasoning at the boundary between two intervals by requiring that two adjoining intervals are disjoint. This allows one to use the same framework to reason about both discrete and hybrid properties [18, 19, 21, 22].

We model time using the real numbers and let *Intv* denote the set of all intervals, which are contiguous subsets of time, i.e.,

$$Intv \mathrel{\widehat=} \{\Delta \subseteq \mathbb{R} \mid \forall t, t' : \Delta, t'' : \mathbb{R} \bullet t < t'' < t' \Rightarrow t'' \in \Delta\}$$

Note that both $\mathbb{R} \in Intv$ and $\varnothing \in Intv$ hold. We let $\mathsf{glb}.\Delta$ and $\mathsf{lub}.\Delta$ denote the *greatest lower* and *least upper bounds* of interval Δ, respectively, where '.' denotes function application. We use $\mathsf{fin}.\Delta$ and $\mathsf{inf}.\Delta$ to denote that the least upper bound is not ∞ and is ∞, respectively. We define $\mathsf{lub}.\varnothing = -\infty$ and $\mathsf{glb}.\varnothing = \infty$. We let $\mathsf{empty}.\Delta$ denote that the interval Δ is empty. Thus, we define:

$$\begin{aligned} \mathsf{fin}.\Delta &\mathrel{\widehat=} \mathsf{lub}.\Delta \neq \infty \\ \mathsf{inf}.\Delta &\mathrel{\widehat=} \mathsf{lub}.\Delta = \infty \\ \mathsf{empty}.\Delta &\mathrel{\widehat=} \Delta = \varnothing \end{aligned}$$

For intervals Δ, Δ_1 and Δ_2, we define the *length* of Δ and *adjoins* relation between Δ_1 and Δ_2 as follows.

$$\begin{aligned} \ell.\Delta &\mathrel{\widehat=} \textbf{if } \mathsf{empty}.\Delta \textbf{ then } 0 \textbf{ else } (\mathsf{lub}.\Delta - \mathsf{glb}.\Delta) \\ \Delta_1 \propto \Delta_2 &\mathrel{\widehat=} (\forall t_1 : \Delta_1, t_2 : \Delta_2 \bullet t_1 < t_2) \wedge (\Delta_1 \cup \Delta_2 \in Intv) \end{aligned}$$

Hence, $\Delta_1 \propto \Delta_2$ holds iff either Δ_1 or Δ_2 is empty or Δ_1 immediately precedes Δ_2. Note that if $\Delta_1 \propto \Delta_2$ they must be contiguous across their boundary and must also be disjoint.

The sets of all subintervals, prefixes, suffixes of an interval Δ are given by $sub.\Delta$, $prefix.\Delta$ and $suffix.\Delta$, respectively, where:

$$\begin{aligned} sub.\Delta &\mathrel{\widehat=} \{\Delta' : Intv \mid \Delta' \subseteq \Delta\} \\ prefix.\Delta &\mathrel{\widehat=} \{\Delta' : sub.\Delta \mid \exists \Delta'' \bullet \Delta' \propto \Delta'' \wedge \Delta = \Delta' \cup \Delta''\} \\ suffix.\Delta &\mathrel{\widehat=} \{\Delta' : sub.\Delta \mid \exists \Delta'' \bullet \Delta'' \propto \Delta' \wedge \Delta = \Delta'' \cup \Delta'\} \end{aligned}$$

Note that the set containment ensures the only prefix of the empty interval is the empty interval.

Given that variable names are taken from the set *Var*, a *state space* over a set of variables $V \subseteq Var$ is given by $State_V \mathrel{\widehat=} V \to Val$, which is a total function from variables in V to values in *Val*. A *state* is a member of $State_V$. The (dense) stream of states over V is given by $Stream_V \mathrel{\widehat=} \mathbb{R} \to State_V$, which is a total function from real numbers (representing time) to states. A *predicate* over a type T is given by $\mathcal{P}T \mathrel{\widehat=} T \to \mathbb{B}$, where $\mathbb{B}$ is the type of a boolean (e.g., a *stream predicate* is a member of $\mathcal{P}Stream_V$). An interval stream predicate, which we shorten to *interval predicate*, has type $IntvPred_V \mathrel{\widehat=} Intv \to \mathcal{P}Stream_V$. We write *State*, *Stream* and *IntvPred* for $State_V$, $Stream_V$ and $IntvPred_V$, respectively, when the set V is clear from the context.

Our intention is to use interval predicates to specify the behaviour of a system over an interval. For example, if c is a state predicate one may define an interval predicate $\diamond c$, which holds for an interval Δ and stream s if c holds in s sometime in Δ, i.e., $(\diamond c).\Delta.s$ holds if there is a time $t \in \Delta$ such that $c.(s.t)$.

We assume pointwise lifting of the boolean operators on stream and interval predicates in the normal manner, e.g., if p_1 and p_2 are interval predicates, Δ is an interval and s is a stream, we have $(p_1 \wedge p_2).\Delta.s = (p_1.\Delta.s \wedge p_2.\Delta.s)$, $\mathsf{true}.\Delta.s = \mathit{true}$ and $\mathsf{false}.\Delta.s = \mathit{false}$. Furthermore, properties of intervals such as fin, inf and empty are assumed to be defined for interval predicates via lifting. When reasoning about programs and their properties, we must often state that if an interval predicate p_1 holds over an arbitrarily chosen interval Δ and stream s, then an interval predicate p_2 also holds over Δ and s. Hence, we define universal implication over intervals and streams as follows. Operators '$\equiv$' and '$\Lleftarrow$' are similarly defined.

$$\begin{aligned} p_1.\Delta \Rrightarrow p_2.\Delta &\;\widehat{=}\; \forall s\colon \mathit{Stream} \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s \\ p_1 \Rrightarrow p_2 &\;\widehat{=}\; \forall \Delta\colon \mathit{Intv} \bullet p_1.\Delta \Rrightarrow p_2.\Delta \end{aligned}$$

A stream describes the behaviour of a system over all time, and an interval predicate describes the behaviour over a given interval. However, because the stream already encodes the behaviour over all time, it is possible to reason about properties outside a given interval in a straightforward manner. For an interval predicate p, interval Δ and stream s, we define the following.

$$\begin{aligned} (\ominus p).\Delta.s &\;\widehat{=}\; \Delta \neq \varnothing \wedge \exists \Delta'\colon \mathit{Intv} \bullet \Delta' \neq \varnothing \wedge (\Delta' \propto \Delta) \wedge p.\Delta'.s \\ (\oplus p).\Delta.s &\;\widehat{=}\; \Delta \neq \varnothing \wedge \exists \Delta'\colon \mathit{Intv} \bullet \Delta' \neq \varnothing \wedge (\Delta \propto \Delta') \wedge p.\Delta'.s \end{aligned}$$

Thus $(\ominus p).\Delta$ and $(\oplus p).\Delta$ hold iff p holds in some interval that immediately precedes and follows Δ, respectively.

3.2. Chop and iteration

Much of the logic is built on the *chop* operator ';' [44, 52]. However, unlike Moszkowski [44], we have a dense notion of time and unlike the duration calculus [52] (in which intervals are always closed), the intervals we use may be open/closed at either end. For interval predicates p_1 and p_2, interval Δ and stream s, we define:

$$(p_1 \;;\; p_2).\Delta.s \;\widehat{=}\; (\exists \Delta_1, \Delta_2\colon \mathit{Intv} \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge p_1.\Delta_1.s \wedge p_2.\Delta_2.s) \vee (\mathsf{inf}.\Delta \wedge p_1.\Delta.s)$$

Thus $(p_1 \;;\; p_2).\Delta$ holds iff either Δ can be split into two adjoining intervals so that p_1 holds for the first interval and p_2 holds for the second, or the given interval Δ has a least upper bound of ∞ and $p_1.\Delta$ holds. Note that empty is the unit of ';', and hence $(\mathsf{empty} \;;\; p) \equiv p \equiv (p \;;\; \mathsf{empty})$ [22].

Using chop, we define the following iteration operators. We assume that interval predicates are ordered using universal entailment ($\Rrightarrow$) with false the least element and true the greatest [22].

$$\begin{aligned} p^{\omega} &\;\widehat{=}\; \nu q \bullet (p \;;\; q) \vee \mathsf{empty} \\ p^{*} &\;\widehat{=}\; \mu q \bullet (p \;;\; q) \vee \mathsf{empty} \\ p^{\infty} &\;\widehat{=}\; \nu q \bullet (p \;;\; q) \end{aligned}$$

Iterations p^{ω} and p^{*} are greatest and least fixed point of $\lambda q \bullet (p \;;\; q) \vee \mathsf{empty}$, respectively, where p^{ω} allows both finite and infinite iterations of p, and p^{*} only models finite iteration of p. Iteration p^{∞} is the greatest fixed point of $\lambda q \bullet (p \;;\; q)$ and only allows infinite iterations of p, unless p holds for some infinite length suffix of the given interval.

Because we have a dense notion of time, there is a possibility for an iteration p^{ω} to behave in a Zeno-like manner, where p iterates an infinite number of times within a finite interval. We can rule out Zeno-like behaviour in our implementations because there is a physical lower limit on the time taken to perform each iteration and hence a specification that *allows* Zeno-like behaviour can be safely ignored. However, we must be careful not to *require* Zeno-like behaviour, which would cause our specifications to become unimplementable.

The following lemmas allow one to fold/unfold infinite and finite iterations and to prove iterative definitions via induction [6, 22, 36]. For an interval predicate p, we let $p^+ \mathrel{\widehat{=}} p\ ;\ p^*$ and $p^{\omega+} \mathrel{\widehat{=}} p\ ;\ p^\omega$ denote the finite and possibly infinite positive iterations of p, respectively.

Lemma 1 (folding/unfolding)**.** *For any interval predicate p, each of the following holds:*

$$\begin{array}{rcl} p^\omega & \equiv & \mathsf{empty} \vee p^{\omega+} \\ p^* & \equiv & \mathsf{empty} \vee p^+ \\ p^\infty & \equiv & p\ ;\ p^\infty \end{array}$$

Lemma 2 (induction)**.** *For any interval predicates p, q and r, each of the following holds:*

$$\begin{array}{rcl} q \Rrightarrow (p\ ;\ q) \vee r & \Rightarrow & q \Rrightarrow (p^\omega\ ;\ r) \\ (p\ ;\ q) \vee r \Rrightarrow q & \Rightarrow & (p^*\ ;\ r) \Rrightarrow q \\ q \Rrightarrow (p\ ;\ q) & \Rightarrow & q \Rrightarrow p^\infty \end{array}$$

3.3. Interval predicate operators

We define a number of operators on interval predicates that enable reasoning about both safety and progress properties, which are defined in terms of chop (;) and iteration ($^\omega$). We may also define operators $\Box$ and $\Diamond$ on interval predicates, where $\Box p.\Delta$ and $\Diamond p.\Delta$ hold iff p holds in *all subintervals* and *some subinterval* of Δ, respectively. Like [36, 44, 52], we define these operators in terms of chop as follows.

$$\begin{array}{rcl} \overleftarrow{\Diamond} p & \mathrel{\widehat{=}} & p\ ;\ \mathit{true} \\ \overrightarrow{\Diamond} p & \mathrel{\widehat{=}} & \mathsf{fin}\ ;\ p \\ \Diamond p & \mathrel{\widehat{=}} & \mathsf{fin}\ ;\ p\ ;\ \mathit{true} \end{array} \qquad \begin{array}{rcl} \overleftarrow{\Box} p & \mathrel{\widehat{=}} & \neg\overleftarrow{\Diamond}\neg p \\ \overrightarrow{\Box} p & \mathrel{\widehat{=}} & \neg\overrightarrow{\Diamond}\neg p \\ \Box p & \mathrel{\widehat{=}} & \neg\Diamond\neg p \end{array}$$

Hence, $\overleftarrow{\Diamond} p.\Delta$ ($\overrightarrow{\Diamond} p.\Delta$) holds iff p holds for some prefix (suffix) of Δ (which may include all of Δ) and $\Diamond p$ holds iff p holds for some subinterval of Δ. Note that because true holds for all intervals, including the empty interval, $\overrightarrow{\Diamond} p \Rrightarrow \Diamond p$ and similarly, $\mathsf{empty} \Rrightarrow \mathsf{fin}$, hence $\overleftarrow{\Diamond} p \Rrightarrow \Diamond p$. Interval predicates $\overleftarrow{\Box} p.\Delta$, $\overrightarrow{\Box} p.\Delta$ and $\Box p.\Delta$ hold iff p holds in all prefixes, all suffixes and all subintervals of Δ, respectively. We have the following calculations, which can be used to convince oneself that this formulation corresponds to its informal meaning.

$$\begin{array}{ll} & (\neg\overleftarrow{\Diamond}\neg p).\Delta \\ = & \quad \text{definitions} \\ & \neg(\neg p\ ;\ \mathit{true}).\Delta \\ = & \quad \text{logic} \\ & \neg(\exists \Delta' \colon \mathit{prefix}.\Delta \bullet (\neg p).\Delta') \\ = & \quad \text{logic} \\ & (\forall \Delta' \colon \mathit{prefix}.\Delta \bullet p.\Delta') \end{array}$$

A similar calculation holds for $\overrightarrow{\Box} p$ and $\Box p$ which allows one to determine that $\overrightarrow{\Box} p.\Delta$ holds iff p holds for all suffixes of Δ and $\Box p.\Delta$ holds iff p holds for all subintervals of Δ.

The operators defined above allow p to hold on the empty interval, which simplifies the algebraic definitions and relationships between properties. Thus, for example, if $p.\varnothing$ holds, $\overleftarrow{\Diamond} p.\Delta$, $\overrightarrow{\Diamond} p.\Delta$ and $\Diamond p.\Delta$ are trivially true for any interval Δ. However, one would often like to specify properties that are strictly non-empty. To this end, we define notation

$$\underline{p} \mathrel{\widehat{=}} p \wedge \neg\mathsf{empty}$$

In the context of state-based traces, where b_1 and b_2 are state predicates $b_1 \rightsquigarrow b_2$ holds for a trace tr if whenever $b_1.(tr.i)$ holds for an index $i \in \mathrm{dom}.tr$, then there is an index j such that $j \geq i$ and $b_2.(tr.j)$

[11, 40]. An LTL-like leads-to operator may be defined for interval predicates in terms of the above operators as follows.

$$p_1 \rightsquigarrow p_2 \quad \widehat{=} \quad \overrightarrow{\Box}(\overleftarrow{\Diamond} p_1 \Rightarrow \Diamond p_2)$$

Thus, if $\overrightarrow{\Box}(\overleftarrow{\Diamond} p_1 \Rightarrow \Diamond p_2).\Omega.s$ holds, then for any suffix Δ of Ω, if $p_1.\Delta_1.s$ holds for some prefix of Δ, then $p_2.\Delta_2.s$ must hold for some subinterval Δ_2 of Δ. As one might expect [11], $\rightsquigarrow$ is both reflexive and transitive.

Certain types of interval predicate are useful for compositional proofs, where the predicates may be moved in and out of the chop and iteration operators. Similar properties for compositionality have been observed by Höfner and Möller [36] and also by von Karger [47, 48].

Definition 1 (splits/joins). For an interval predicate p, we say

- p *splits* iff $p \Rrightarrow \Box p$ holds
- p *joins* iff $p^{\omega+} \Rrightarrow p$ holds.

If interval predicate p splits, then p holds for any subinterval of Δ provided $p.\Delta$ holds. If p joins then $p.\Delta$ holds provided p holds iteratively in Δ with at least one iteration. For example, $\Box p$ splits because $\Box\Box p \equiv \Box p$ and $\Diamond p$ joins because $(\Diamond p \,;\, (\Diamond p)^{\omega}) \Rrightarrow \Diamond p$ holds. However, $\Box p$ may not join and $\Diamond p$ may not split. We have the following lemma relating interval predicates that split and join [22].

Lemma 3. *For interval predicates p, q, q_1 and q_2, each of the following holds:*

(a)	$p \wedge (q_1 \,;\, q_2) \Rrightarrow (p \wedge q_1) \,;\, (p \wedge q_2)$	*provided p splits*
(b)	$p \wedge q^{\omega} \Rrightarrow (p \wedge q)^{\omega}$	*provided p splits*
(c)	$(p \wedge q_1) \,;\, (p \wedge q_2) \Rrightarrow p \wedge (q_1 \,;\, q_2)$	*provided p joins*
(d)	$(p \wedge q)^{\omega+} \Rrightarrow p \wedge q^{\omega+}$	*provided p joins*

Proof. The proofs of (a) and (c) are straightforward. By Lemma 2, (b) holds if

$$p \wedge q^{\omega} \quad \Rrightarrow \quad ((p \wedge q) \,;\, (p \wedge q^{\omega})) \vee \mathsf{empty}$$

which is proved as follows:

$$\begin{array}{ll} & p \wedge q^{\omega} \\ \equiv & \quad \text{Lemma 1} \\ & p \wedge (q \,;\, q^{\omega} \vee \mathsf{empty}) \\ \equiv & \quad \text{distributivity} \\ & (p \wedge (q \,;\, q^{\omega})) \vee (p \wedge \mathsf{empty}) \\ \Rrightarrow & \quad \text{assumption } p \text{ splits and part } (a), \text{ logic} \\ & ((p \wedge q) \,;\, (p \wedge q^{\omega})) \vee \mathsf{empty} \end{array}$$

For (d), we have that $(p \wedge q)^{\omega+} \Rrightarrow p^{\omega+}$ by monotonicity and therefore because p joins, $(p \wedge q)^{\omega+} \Rrightarrow p$. Again, by monotonicity, $(p \wedge q)^{\omega+} \Rrightarrow q^{\omega+}$ and the result follows. □

3.4. Evaluating state predicates over actual states

A state predicate over a set of variables V is a member of $\mathcal{P}State_V$, which may be used to denote whether a property does or does not hold on a state. Because there are multiple states of a stream within a non-point interval, there are several possible ways of evaluating a state predicate with respect to a given interval and stream [30]. In particular, we distinguish between evaluations over actual states (which are the states that actually occur within an interval) and apparent states (which are the states that a controller determines by sampling the various inputs). Operators for evaluation in the actual states are given below and for apparent states in Section 4.3.

We must often specify properties on the actual states of a stream within an interval. Thus, we define the *always* '$\boxdot$' and *sometime* '$\diamondsuit$' operators as follows.[3]

$$\begin{aligned}(\boxdot c).\Delta.s &\mathrel{\widehat{=}} \forall t{:}\,\Delta \bullet c.(s.t)\\ (\diamondsuit c).\Delta.s &\mathrel{\widehat{=}} \exists t{:}\,\Delta \bullet c.(s.t)\end{aligned}$$

Example 1. For variable x and stream s such that $s.0.x = 41$ and $(\boxdot \mathring{x}).[0,2].s = 1$ hold, where $\mathring{x}$ denotes the rate of change of variable x, i.e., derivative with respect to time (c.f. [33]), we have:

- $(\boxdot(x < 42)\,;\ \boxdot(x \geq 42)).[0,2].s$ holds because there exist adjoining intervals $[0,1)$ and $[1,2]$, such that both $(\boxdot(x < 42)).[0,1).s$ and $(\boxdot(x \geq 42)).[1,2].s$ hold;
- $(\boxdot(x \leq 42)\,;\ \boxdot(x > 42)).[0,2].s$ holds because there exist adjoining intervals $[0,1]$ and $(1,2]$, such that both $(\boxdot(x \leq 42)).[0,1].s$ and $(\boxdot(x > 42)).(1,2].s$ hold;
- however $(\boxdot(x < 42)\,;\ \boxdot(x > 42)).[0,2].s$ does not hold.

Intervals may be open or closed at either end, and hence, defining the value of a variable at the ends of an interval is non-trivial. For instance, suppose we are interested in defining the value of variable v at the right end of an interval Δ. If Δ is right closed, the value of v at the right end of Δ is simply its value at the least upper bound of Δ. However, if Δ is right open, because $\mathsf{lub}.\Delta \notin \Delta$, one must take the limit of v approaching $\mathsf{lub}.\Delta$. Because we only assume piecewise continuity, it is possible for the values of v approaching $\mathsf{lub}.\Delta$ from the left and right to differ (e.g., if there is a point of discontinuity at $\mathsf{lub}.\Delta$). To ensure that the value of v for the right end of an open interval Δ is sensible, we take the limit of v approaching $\mathsf{lub}.\Delta$ from below. A similar argument applies to the value of v at the left end of Δ. We use $\lim\limits_{x \to a^+} f.x$ and $\lim\limits_{x \to a^-} f.x$ to denote the limit of $f.x$ as x tends to a from above and below, respectively. To ensure that the limits are well defined, for each $s \in \mathit{Stream}_V$ and $v \in V$, we assume that $(\lambda\, t \bullet s.t.v)$ is piecewise continuous.

For a variable v, interval Δ, and stream s, we define the following, where we assume $\perp$ denotes an undefined value.

$$\overrightarrow{v}.\Delta.s \mathrel{\widehat{=}} \begin{cases} s.(\mathsf{lub}.\Delta).v & \text{if } \mathsf{lub}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{lub}.\Delta^-} s.t.v & \text{if } \mathsf{lub}.\Delta \in \mathbb{R} \setminus \Delta \\ \perp & \text{otherwise} \end{cases}$$

$$\overleftarrow{v}.\Delta.s \mathrel{\widehat{=}} \begin{cases} s.(\mathsf{glb}.\Delta).v & \text{if } \mathsf{glb}.\Delta \in \Delta \\ \lim\limits_{t \to \mathsf{glb}.\Delta^+} s.t.v & \text{if } \mathsf{glb}.\Delta \in \mathbb{R} \setminus \Delta \\ \perp & \text{otherwise} \end{cases}$$

Thus, if Δ is right closed, then the value of $\overrightarrow{v}$ in Δ is the value of v at the least upper bound of Δ, if Δ is right open and $\mathsf{lub}.\Delta \in \mathbb{R}$, the value of $\overrightarrow{v}$ is the value of v as it approaches $\mathsf{lub}.\Delta$ from the left, otherwise the value is undefined. The interpretation of $\overleftarrow{v}.\Delta$ is similar.

We interpret boolean expressions that use $\overleftarrow{v}$ and $\overrightarrow{v}$ using pointwise lifting. For example, if k is a constant, we have $(\overleftarrow{v} = k).\Delta.s = (\overleftarrow{v}.\Delta.s = k.\Delta.s) = (\overleftarrow{v}.\Delta.s = k)$.

Example 2. For x and s as defined in Example 1, both $(\overrightarrow{x} = 42).[0,1).s$ and $(\overleftarrow{x} = 42).[1,2].s$ hold.

Example 3. Suppose y is a piecewise continuous discrete variable, such that in a stream s, both $\boxdot(y = 10).[0,1).s$ and $\boxdot(y = 20).[1,2].s$ hold. For example, y may represent the voltage that instantaneously jumps from 10 to 20. Then both $(\overrightarrow{y} = 10).[0,1).s$ and $(\overleftarrow{y} = 20).[1,2].s$ hold, i.e., the limit of the value of y approaching time 1 from the left differs from the value at time 1.

[3]The notation for always and sometime has changed from that in [18, 19, 20, 21].

For a variable v and set of variables V we define the following:

$$\begin{aligned}\mathsf{stable}.v &\mathrel{\widehat{=}} \exists k\colon Val \bullet \ominus(\overrightarrow{v} = k) \wedge \boxdot(v = k)\\ \mathsf{stable}.V &\mathrel{\widehat{=}} \forall v\colon V \bullet \mathsf{stable}.v\end{aligned}$$

Hence, a variable v is stable, denoted $\mathsf{stable}.v$, iff its value does not change from its value at the right end of some immediately preceding interval, and $\mathsf{stable}.V$ holds iff each variable in V is stable. Such definitions of stability are necessary because adjoining intervals are disjoint, and hence $\ominus(\overrightarrow{v} = k).\Delta$ does not necessarily imply $(\overleftarrow{v} = k).\Delta$ and vice versa. However, for a continuous variable v, we have

$$\mathsf{stable}.v \equiv \exists k\colon Val \bullet (\overleftarrow{v} = k) \wedge \boxdot(v = k)\ .$$

Lemma 4. *If x is a boolean-valued variable, both of the following hold:*

$$\mathsf{stable}.x \quad \Rrightarrow \quad \boxdot x \vee \boxdot\neg x \tag{1}$$

$$\diamondsuit x \wedge \mathsf{stable}.x \quad \Rrightarrow \quad \boxdot x \tag{2}$$

Proof. Condition (1) holds by case analysis on the value of x at the end of the immediately preceding interval. For condition (2), we have the following calculation:

$$\begin{array}{ll} & \diamondsuit x \wedge \mathsf{stable}.x\\ \Rrightarrow & \quad (1)\\ & \diamondsuit x \wedge (\boxdot x \vee \boxdot\neg x)\\ \equiv & \quad \diamondsuit c \Rrightarrow \neg\mathsf{empty}\\ & \diamondsuit x \wedge \neg\mathsf{empty} \wedge (\boxdot x \vee \boxdot\neg x)\\ \equiv & \quad \text{logic and } \boxdot x \wedge \neg\mathsf{empty} \Rrightarrow \diamondsuit x\\ & \boxdot x \end{array}$$

□

Notation $\overleftarrow{c}$ and $\overrightarrow{c}$ are used to denote the values of state predicates at the left and right ends of an interval, respectively, where

$$\begin{aligned}\overleftarrow{c} &\mathrel{\widehat{=}} \overleftarrow{\diamondsuit\underline{\boxdot c}}\\ \overrightarrow{c} &\mathrel{\widehat{=}} \overrightarrow{\diamondsuit\underline{\boxdot c}}\end{aligned}$$

For any state predicates c, c_1 and c_2, each of the following hold:

$$\begin{aligned}\overleftarrow{c_1 \wedge c_2} &\equiv \overleftarrow{c_1} \wedge \overleftarrow{c_2}\\ \overleftarrow{c_1 \vee c_2} &\equiv \overleftarrow{c_1} \vee \overleftarrow{c_2}\\ \overleftarrow{\neg c} &\Rrightarrow \neg\overleftarrow{c}\end{aligned}$$

Note that for a variable v and constant k, $(\overleftarrow{v} = k)$ may not imply $\overleftarrow{v = k}$, and vice versa. To see this, consider the following example:

Example 4. For x and s as defined in Example 1, $(\overleftarrow{x \neq 42}).(1,2].s$ holds even though $(\overleftarrow{x} = 42).(1,2].s$ holds. Clearly, the value of x at time 1 is 42. Hence, the left limit of the variable x within $(1,2]$ is 42 because the value of x will be arbitrarily close to 42 as we approach the greatest lower bound 1 from the right. However, the left limit of the predicate $x = 42$ is *false* because $x = 42$ is evaluated in states within the interval $(1,2]$, and $x \neq 42$ for each of these states. For the closed interval $[1,2]$, both $(\overleftarrow{x} = 42).[1,2].s$ and $(\overleftarrow{x = 42}).[1,2].s$ hold because the value of x in state $s.1$ is 42.

Using the values at the beginning and ends of an interval, we define the following property of state predicates. We say that a state predicate c is invariant over an interval Δ iff $\mathsf{inv}.c.\Delta$ holds where:

$$\mathsf{inv}.c \quad \mathrel{\widehat{=}} \quad \ominus\overrightarrow{c} \Rightarrow \boxdot c$$

Thus, if $\mathsf{inv}.c.\Delta$ holds, then c holds in each of the actual states within Δ provided that c holds over some interval that immediately precedes Δ. Note that c may be invariant in an interval even if the variables in c change within the interval.

A variable is right stable iff its value does not change at the end of the current interval and a set of variables is right stable iff each variable in the set is right stable. Thus, we define

$$\begin{array}{rcl} \mathsf{right\_stable}.v & \widehat{=} & \exists k\colon Val \bullet \overrightarrow{v = k} \wedge \oplus \overleftarrow{(v = k)} \\ \mathsf{right\_stable}.V & \widehat{=} & \forall v\colon V \bullet \mathsf{right\_stable}.v \end{array}$$

Note that there is a fundamental difference between the definitions of stable and $\mathsf{right\_stable}$. If v is a continuous variable, $\mathsf{stable}.v \equiv \exists k\colon Val \bullet (\overleftarrow{v} = k) \wedge \boxdot(v = k)$, whereas $\mathsf{right\_stable}.v$ cannot be simplified so that the immediately following interval need not be considered because the predicate $v = k$ must hold at the end of the current interval, and at the start of the interval that immediately follows the given interval.

3.5. Zeno-like behaviour

Because we have a dense notion of time, it is possible for interval predicates to specify Zeno-like behaviour, e.g., a state predicate may switch between true and false an infinite number of times within a finite interval. As mentioned earlier, a specification that allows Zeno-like behaviour is not problematic because a real system will not behave in a Zeno-like manner. However, one must take care not to require Zeno-like behaviour, which would mean the system specification is unimplementable.

We assume sequences have natural number indices (starting from 0) and may be infinite.

Definition 2 (partition). The set of all *partitions* of an interval Δ is given by

$$\mathit{partition}.\Delta \quad \widehat{=} \quad \{z \in \mathrm{seq}.\mathit{Intv} \mid (\Delta = \bigcup \mathrm{ran}.z) \wedge (\forall i \in \mathrm{dom}.z \bullet z.i \neq \varnothing \wedge (i \neq 0 \Rightarrow z.(i-1) \propto z.i))\}$$

The set of *non-Zeno partitions* of Δ is given by

$$\mathit{nz\_partition}.\Delta \quad \widehat{=} \quad \{z \in \mathit{partition}.\Delta \mid \ell.\Delta \neq \infty \Rightarrow \mathrm{dom}.z \neq \mathbb{N}\}$$

Thus, z is a non-Zeno partition of Δ iff z has a finite number of elements whenever Δ is finite.

Definition 3 (alternates). For a state predicate c, interval Δ, partition $z \in \mathit{partition}.\Delta$ and stream s, we define

$$\mathit{alt}.c.z.s \quad \widehat{=} \quad \forall i \in \mathrm{dom}.z \bullet (\boxdot c \vee \boxdot \neg c).(z.i).s \wedge (i > 0 \Rightarrow ((\boxdot c).(z.(i-1)).s = (\boxdot \neg c).(z.i).s))$$

Thus, $\mathit{alt}.c.z.s$ holds iff z contains a single interval Δ and either $(\boxdot c).\Delta.s$ or $(\boxdot \neg c).\Delta.s$ holds, or c alternates between $\boxdot c$ and $\boxdot \neg c$ holding within the partition z.

Definition 4. A state predicate c is *non-Zeno* in stream s within interval Δ, denoted $(NZ.c).\Delta.s$, iff there exists a $z \in \mathit{nz\_partition}.\Delta$ such that $\mathit{alt}.c.z.s$ holds.

Note that, if such a z exists, then it is unique. Further note that $NZ.c$ splits for any state predicate c.

The lemma below allows one to decompose a proof of an interval predicate that joins by considering subintervals in which some state predicate c is true and false separately. The lemma requires that the chosen c is non-Zeno — if c is true on the irrationals and false on the rationals then the iteration operator (used in joins) will be invalid [23].

Lemma 5. *If p is an interval predicate that joins and c is a state predicate, then the following hold:*

$$\Box(\boxdot c \Rightarrow p) \wedge \Box(\boxdot \neg c \Rightarrow p) \wedge NZ.c \quad \Rrightarrow \quad p \tag{3}$$

$$\Box(\ominus \overrightarrow{\neg c} \wedge \boxdot c \Rightarrow p) \wedge \Box(\ominus \overrightarrow{c} \wedge \boxdot \neg c \Rightarrow p) \wedge NZ.c \quad \Rrightarrow \quad (\ominus \overrightarrow{c} \wedge \overleftarrow{\neg c}) \vee (\ominus \overrightarrow{\neg c} \wedge \overleftarrow{c}) \Rightarrow p \tag{4}$$

$$\Box(\ominus \overrightarrow{\neg c} \wedge \boxdot c \Rightarrow p) \wedge \Box(\ominus \overrightarrow{c} \wedge \boxdot \neg c \Rightarrow p) \wedge NZ.c \quad \Rrightarrow \quad \overrightarrow{\Box}((\ominus \overrightarrow{c} \wedge \overleftarrow{\neg c}) \vee (\ominus \overrightarrow{\neg c} \wedge \overleftarrow{c}) \Rightarrow p) \tag{5}$$

For each of the proofs below, we assume $V \subseteq \mathit{Var}$, $\Delta \in \mathit{Intv}$ and $s \in \mathit{Stream}_V$ are arbitrarily chosen, and that $(\mathit{NZ}.c).\Delta.s$ holds. The proofs are trivial if Δ is empty. The proofs below cover the cases when Δ is non-empty.

Proof of (3).

$(\mathit{NZ}.c).\Delta.s$
$\Rightarrow$ definition
$\exists z{:}\, \mathit{nz\_partition}.\Delta \bullet \mathit{alt}.c.z.s$
$\Rightarrow$ assumption $(\Box(\boxdot c \Rightarrow p) \wedge \Box(\boxdot \neg c \Rightarrow p)).\Delta.s$
$\exists z{:}\, \mathit{nz\_partition}.\Delta \bullet \forall i{:}\, \mathrm{dom}.z \bullet p.(z.i).s$
$\Rightarrow$ p joins
$p.\Delta.s$

Proof of (4).

$(\mathit{NZ}.c).\Delta.s$
$\Rightarrow$ definition
$\exists z{:}\, \mathit{nz\_partition}.\Delta \bullet \mathit{alt}.c.z.s$
$\Rightarrow$ $\mathit{alt}.c.z.s$ and assumption $(\ominus \overrightarrow{c} \wedge \overleftarrow{\neg c}) \vee (\ominus \overrightarrow{\neg c} \wedge \overleftarrow{c}).\Delta.s$
$\exists z{:}\, \mathit{nz\_partition}.\Delta \bullet \forall i{:}\, \mathrm{dom}.z \bullet ((\ominus \overrightarrow{c} \wedge \boxdot \neg c) \vee (\ominus \overrightarrow{\neg c} \wedge \boxdot c)).(z.i).s$
$\Rightarrow$ assumption $\Box(\ominus \overrightarrow{\neg c} \wedge \boxdot c \Rightarrow p) \wedge \Box(\ominus \overrightarrow{c} \wedge \boxdot \neg c \Rightarrow p).\Delta.s$
$\exists z{:}\, \mathit{nz\_partition}.\Delta \bullet \forall i{:}\, \mathrm{dom}.z \bullet p.(z.i).s$
$\Rightarrow$ p joins
$p.\Delta.s$

Proof of (5). For an arbitrarily chosen $\Delta' \in \mathit{suffix}.\Delta$, because $\mathit{NZ}.c$ splits, we have $\mathit{NZ}.c.\Delta'.s$. Furthermore, $\Box p$ splits for any interval predicate p, so we also have $(\Box(\ominus \overrightarrow{\neg c} \wedge \boxdot c \Rightarrow p) \wedge \Box(\ominus \overrightarrow{c} \wedge \boxdot \neg c \Rightarrow p)).\Delta'.s$, i.e., the conditions of (4) are satisfied in Δ', and hence, the result follows. □

4. Time bands and sampling

In Section 4.1 we present the notion of time bands—these allow one to formalise properties at multiple time scales. Section 4.2 discusses sampling activities and demonstrates the typical structure of real-time controllers and a range of issues that these controllers must handle. Apparent states and related operators are introduced in Section 4.3—these allow one to reason about sampling activities. Finally, in Section 4.4 we present methods for deducing the ranges of possible actual values of sampled variables.

4.1. Time bands

Several frameworks for reasoning about properties over multiple time granularities have been proposed. Moszkowski presents a method of abstracting between different time granularities for interval temporal logic using a projection operator for a discrete interval temporal logic [42]. Guelev and Hung present a projection operator for the duration calculus. Although computation is assumed to take time, the time taken is assumed to be negligible [28]. Henzinger presents a theory of timed refinement where sampling events are executed by a separate process [34]. Broy [8] presents a timed refinement framework that formalises the relationships between dense and discrete time where sampling is considered be a discretisation of dense streams. We use the time bands framework, which generalises reasoning over multiple time granularities and encapsulates the methods in [8, 28, 34, 42]. Furthermore, it includes methods for reasoning about sampling. Note that by mapping other frameworks such as Z, or automata to an appropriate interval-based semantics, it is possible to adapt these frameworks to cover sampling.

The central idea behind the time bands framework is that it associates each system component with a time band, which formalises the time granularity and precision of the component. It is possible to define relationships between different time bands, which in turn establishes timing relationships between the components at these time bands. Each time band is associated with a precision, which defines the maximum

```
SamplingBlock(i) ≙
  if t_i ≤ τ_loc →
       v_i := read(input_i) ;
       if c_i1 → output(val_i1)
       else if c_i2 → output(val_i2)
       ...
       else if c_im → output(val_im)
       fi ;
  fi ;
  t_i := t_i + Period_i
```

```
t_1, ..., t_n := τ, ..., τ ;
do true →
     τ_loc := τ ;
     // Sampling period 1
     SamplingBlock(1) ;
     // Sampling period 2
     SamplingBlock(2) ;
     ...
     // Sampling period n
     SamplingBlock(n)
od
```

Figure 2: Typical structure of a real-time controller

amount of time taken to execute any event of that time band. Events are considered to be instantaneous in their time band, but may be mapped to an activity in a finer-grained time band, within which the event may be observed to be an activity that takes time.

We assume that the set of all time bands is given by the primitive type *TimeBand* [9, 10]. Each time band may be associated with *events* that execute within the *precision* of the time band. We use ρ: *TimeBand* $\to \mathbb{R}_{>0}$ to denote the precision of the given time band. An event in a time band β is guaranteed to be completed within $\rho.\beta$.

4.2. Sampling activities

A reactive controller uses *sampling activities* to determine the state of its continuously evolving environment. Because sampling activities take time and because the environment operates in parallel with the controller in a truly concurrent manner, sampling activities can be prone to *sensor errors* (where the sensors have inaccuracies in measuring the environment), *timing precision errors* (where there is a range of possible sampled values due to imprecise timing of when the sample is taken), and *sampling anomalies* (where a sampling activity that samples more than one environment variable at slightly different times returns an apparent state that does not exist at any single instant of time). In this paper, we focus on timing precision errors and sampling anomalies — reasoning about the sensor errors is a straightforward extension.

Example 5. As an example consider the program in Fig. 2 which represents a typical implementation. The program initially sets the values of $t_1, t_2, \ldots, t_n$ to the current time τ then executes a non-terminating loop with n sampling blocks. Each block represents a different set of inputs sampled with different time periods. Within sampling block i, a vector of inputs $\mathbf{input_i}$ is read and stored in a vector of variables $\mathbf{v_i}$. The program outputs vector of outputs $\mathbf{val_{ik}}$ if guard c_{ik} holds. The time t_i is then updated so that the timer expires and $\mathbf{input_i}$ is resampled after $Period_i$ units of time have elapsed. Typically, each of these sampling blocks takes place over differing time periods because different components are controlled at different rates, and hence the values of $Period_i$ are different.

Real-time controllers often evaluate an expression over an interval by sampling the variables of the expression at different instants within the sampling interval. For the example in Fig. 2, although the **read** command samples the vector of inputs $\mathbf{input_i}$, each input is read at a slightly different time. Hence, reasoning about an expression evaluation that samples two or more variables can be problematic. To minimise such sampling anomalies similar samples are taken within the same sampling block.

Example 6. The control flow diagram for the two-pump system is given in Fig. 3, where controller C must sample variables $water_1$, $water_2$, and *pressed* (as indicated by the dotted lines) and outputs on_1 and on_2 to signal $Pump_1$ and $Pump_2$, respectively. $Pump_i$ takes on_i as input and outputs $mode_i$. Note that $mode_1$ affects both $Tank_1$ and $Tank_2$, whereas $mode_2$ only affects $Tank_2$.

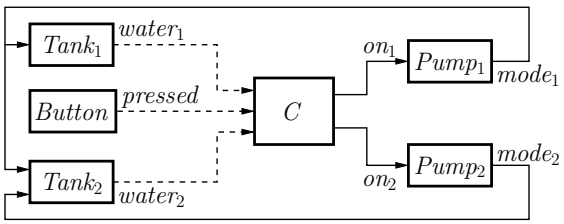

Figure 3: Control flow diagram for two-pump example

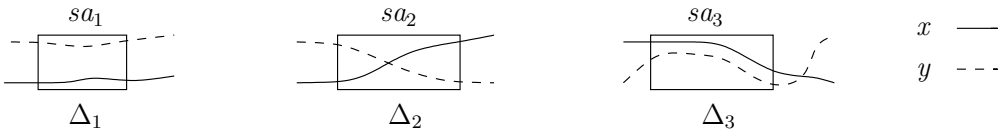

Figure 4: Sampling activities

Example 7. Consider the three sampling activities sa_1, sa_2 and sa_3 in Fig. 4, where environment variables x and y are sampled at different times within the interval. Event sa_1 will return $x < y$ regardless of when x and y are read within the sampling interval because $x < y$ *definitely* holds for all sampled values of x and y. Event sa_2 may return either $x > y$, $x = y$ or $x < y$ because it is *possibly* true that $x > y$, $x = y$ and $x < y$ hold. Event sa_3 may have a sampling anomaly. Although $x > y$ holds throughout the interval in which sa_3 occurs, because x and y are sampled at different times, it is *possible* for sa_3 to return either $x > y$, $x = y$ or $x < y$.

Note that with the controller scheme presented in Fig. 2, variables are sampled at most once, hence, if a variable occurs multiple times within an expression, the same sampled value is used for each occurrence of the variable. Hence, expression $x = x$ is guaranteed to evaluate to true regardless of how x changes within the evaluation interval, however, $x > y$ may evaluate to false even if $\boxdot(x > y)$ holds [10, 18, 30] as in sa_3 in Figure 4.

Using a sampling logic over intervals allows one to resolve transient behaviour and hence avoid formalisation of unimplementable specifications. We say a state predicate is *transient* in a stream if the predicate only holds for a brief (e.g., an attosecond) amount of time, whereby it is not possible to reliably detect that the predicate held [17, 20].

4.3. Apparent states

We use the set of apparent states of $s \in \mathit{Stream}_V$ within interval Δ (denoted $\mathit{apparent}.\Delta.s$) to reason about sampling-based expression evaluation. An apparent state over an interval Δ gives a value for each variable x that is a value of x in Δ but the values of different variables x and y within an apparent state may be (sampled) at different times within Δ. We define the set of all possible apparent states over an interval Δ for a stream s as follows.

$$\mathit{apparent}.\Delta.s \;\widehat{=}\; \{\sigma{:}\,\mathit{State}_V \mid \forall\, v{:}\, V \bullet \sigma.v \in \{t{:}\,\Delta \bullet (s.t).v\}\}$$

where $\{t{:}\,\Delta \bullet (s.t).v\}$ is equal to $\{x \in \mathit{Val} \mid \exists t{:}\,\Delta \bullet x = (s.t).v\}$. To obtain the apparent states, for each variable v we first generate $\{t{:}\,\Delta \bullet (s.t).v\}$, the set of possible values of v within the interval, then generate the set of all possible states using these values. We formalise state predicates that are *definitely* true (denoted $\circledast$) and *possibly* true (denoted $\circledcirc$) over a given interval Δ and stream s as follows:

$$(\circledast c).\Delta.s \;\widehat{=}\; \forall \sigma{:}\, \mathit{apparent}.\Delta.s \bullet c.\sigma$$

$$(\circledast c).\Delta.s \;\widehat{=}\; \exists \sigma\colon apparent.\Delta.s \bullet c.\sigma$$

Hence, $(\boxtimes c).\Delta.s$ and $(\circledast c).\Delta.s$ hold iff c holds in every and in some apparent state of s within the interval Δ, respectively.

Example 8. For intervals Δ_1, Δ_2 and Δ_3 as given in Fig. 4, one can deduce both $(\boxtimes(x < y)).\Delta_1$ and $(\circledast(x < y) \wedge \circledast(x \geq y)).\Delta_2$. For se_3 (the event with a sampling anomaly), $(\circledast(x \leq y)).\Delta_3$ holds, despite the fact that $\boxdot(x > y).\Delta_3$ holds.

One can derive a number of relationships between operators $\boxtimes$, $\circledast$, $\boxdot$ and $\diamondsuit$. Because the set of actual states over an interval Δ is a subset of the apparent states over Δ, one can derive the following [30].

$$\boxtimes c \;\Rrightarrow\; \boxdot c \tag{6}$$
$$\diamondsuit c \;\Rrightarrow\; \circledast c \tag{7}$$

However, the converse of each implication is not necessarily true.

Sampling anomalies can only be present if multiple (evolving) variables are sampled within an interval. Thus, for a predicate, say c, that only refers to a single non-stable variable, it is equivalent that c definitely holds and that c holds everywhere within an interval [30]. We let $vars.c$ denote the set of free variables in state predicate c. The following lemma states that if all but one variable of c is stable over an interval Δ, then c definitely holds in Δ iff c always holds in Δ and c possibly holds in Δ iff c holds at some time in Δ [30].

Lemma 6. *For any state predicate c and variable v,* $\mathsf{stable}.(vars.c \setminus \{v\}) \Rrightarrow (\boxtimes c = \boxdot c) \wedge (\circledast c = \diamondsuit c)$.

Example 9. As an example, we develop a formalisation of the informal requirements of the two-pump system in Section 1.

Safety. The safety properties **S1**, **S2** and **S3** are properties that are required to hold when the controller is operating. The relationship between the water level and the pump modes must hold in the actual states of the system, as opposed to the apparent states observed by the controller. Hence, each of **S1**, **S2**, and **S3** are formalised using '$\boxdot$'. However, the requirements as stated in **S1**, **S2** and **S3** are too strong, and hence, unimplementable. For example, **S1** may be violated if $water_1$ is just above $empty_1$ and the $Pump_1$ is running at initialisation, whereby $water_1$ drops below $empty_1$ as soon as the action system is started. Such circumstances are beyond the control of the action system controller, therefore, we weaken requirements **S1**, **S2**, and **S3** to assume some properties about the initial state of the system. In particular, we rephrase **S1**, **S2** and **S3** as **T1**, **T2** and **T3**, respectively below. The modified portions of the requirements are highlighted in boxes.

T1. Provided that $Pump_1$'s mode is initially *stopped* and $Pump_1$ is not signalled on, whenever $water_1$ (the water level in $Tank_1$) is $empty_1$ or below, $Pump_1$ must be stopped.

T2. Provided that $Pump_1$'s mode is initially *stopped* and $Pump_1$ is not signalled on, whenever $water_2$ (the water level in $Tank_2$) is $full_2$ or above, $Pump_1$ must be stopped.

T3. Provided that $Pump_2$'s mode is initially *stopped* and $Pump_2$ is not signalled on, whenever $water_2$ is $empty_2$ or below, $Pump_2$ must be stopped.

We combine **T1** and **T2** and formalise them as (8) in Fig. 5, and formalise **T3** as (9). This is possible because $\boxdot(c_1 \wedge c_2) \equiv \boxdot c_1 \wedge \boxdot c_2$. By (8), provided that the $Pump_1$ is initially stopped and the on_1 signal is off, over the interval in which the program is executing, in all actual (as opposed to apparent) states of the stream, if $water_1$ is below $empty_1$ or $water_2$ is above $full_2$, then $Pump_1$ must be stopped. Note that $Stopped_1$ within (8) states that $Pump_1$ has physically come to a stop, which we distinguish from the signal $\neg on_1$ that causes $Pump_1$ to stop. Condition (9) is similar. The (implicit) interval over which these properties must hold is the interval over which the controller is operating.

$$\ominus(\overrightarrow{Stopped_1 \wedge \neg on_1}) \quad \Rightarrow \quad \boxdot((water_1 \leq empty_1) \vee (water_2 \geq full_2) \Rightarrow Stopped_1) \tag{8}$$

$$\ominus(\overrightarrow{Stopped_2 \wedge \neg on_2}) \quad \Rightarrow \quad \boxdot(water_2 \leq empty_2 \Rightarrow Stopped_2) \tag{9}$$

$$\circledast\big((water_1 > reserve_1) \wedge (water_2 < reserve_2)\big) \wedge \ell \geq \rho.TB_{water} \quad \rightsquigarrow \quad \underline{\boxdot on_1} \tag{10}$$

$$\circledast\big((water_2 > reserve_2) \wedge pressed\big) \wedge \ell \geq \rho.TB_{water} \quad \rightsquigarrow \quad \underline{\boxdot on_2} \tag{11}$$

$$\circledast\neg pressed \wedge \ell \geq \rho.TB_{water} \quad \rightsquigarrow \quad \underline{\boxdot \neg on_2} \tag{12}$$

Figure 5: Formalisation of the two-pump system requirements

Progress. The progress properties **P1**, **P2** and **P3** must also be further refined because they describe properties on the controller's sampled view of the environment. For example, **P1** states "If $water_2$ is below $reserve_2$ and $water_1$ is above $reserve_1$, then $Pump_1$ must eventually be turned on." The state predicate corresponding to the 'if' part of this requirement is $(water_2 < reserve_2) \wedge (water_1 > reserve_1)$, which would be difficult for an implementation to satisfy if stated in terms of actual states because it would require the controller to turn the pump on as soon as $(water_2 < reserve_2) \wedge (water_1 > reserve_1)$ holds. In reality, it may be possible for $(water_2 < reserve_2) \wedge (water_1 > reserve_1)$ to go undetected by the controller because it is transient. Furthermore, because $water_1$ and $water_2$ are sampled at different instants within the same sampling period, the controller may determine that $(water_2 < reserve_2) \wedge (water_1 > reserve_1)$ holds even if it does not hold in any actual state. Hence, we reword progress properties **P1**, **P2** and **P3** to clarify the length of time for which the properties must hold, and the fact that we are intending to using a sampling logic. For simplicity, we assume that the time bands of the water in both tanks is the same, i.e., $TB_{water} \in TimeBand$.

Q1. If it is definitely the case that $water_1$ is above $reserve_1$ and $water_2$ is below $reserve_2$ for at least the precision of TB_{water}, then $Pump_1$ must eventually be turned on.

Q2. If it is definitely the case that the *Button* is pressed and $water_2$ is above $reserve_2$ for at least the precision of TB_{water}, then $Pump_2$ must eventually be turned on.

Q3. If *Button* is definitely not pressed for at least the precision of TB_{water}, $Pump_2$ must eventually be turned off.

By using the keyword *definitely*, it is clear that one must use sampling operators to formalise the progress properties **Q1**, **Q2** and **Q3** to take into account the fact that sampling may take place at different times within a sampling period. Furthermore, the precision of this sampling period has been clarified — the 'if' part of the predicates must hold for at least the precision of TB_{water} to guarantee that the controller is able to detects the property and react accordingly.

Progress properties **Q1**, **Q2** and **Q3** are formalised as (10), (11) and (12), respectively. Condition (10) states that if there exists a prefix of length greater than or equal to $\rho.TB_{water}$ (the precision of TB_{water}) in which it is definitely the case that $water_2$ is below $reserve_2$ and $water_1$ is above $reserve_1$, then $pump_1$ must be switched on. Conditions (11) and (12) are similar.

4.4. Deducing actual values from sampled values

Due to the various delays involved in a real system, a sampled value of any variable represents an approximation of the true value of the variable. In this section, we present some techniques for relating the sampled values of a variable to its true values in the environment.

We define the following interval predicates, which are useful for reasoning about sampling events, where c is a state predicate and d is a real-valued constant.

$$\begin{aligned} \circledast_d c &\;\widehat{=}\; (\ell \leq d) \Rightarrow \circledast c \\ \circledcirc_d c &\;\widehat{=}\; (\ell \leq d) \wedge \circledcirc c \end{aligned}$$

Hence, $(\circledast_d c).\Delta$ holds iff c definitely holds within Δ provided that the length of Δ is at most d. Similarly, $(\circledcirc_d c).\Delta$ holds iff c possibly holds within Δ and the length of Δ is at most d. Note that $\neg\circledast_d c \equiv \circledcirc_d \neg c$.

Because sampling approximates the true value of an environment variable, we must reason about how the value of a variable changes within an interval [18]. For a real-valued variable v, the maximum difference to v in stream s within Δ is given by $(\mathit{diff}.v).\Delta.s$, where:

$$(\mathit{diff}.v).\Delta.s \;\widehat{=}\; \textbf{if } \mathsf{empty}.\Delta \textbf{ then } 0 \textbf{ else } (\mathsf{let}\ vs = \{t{:}\,\Delta \bullet (s.t).v\}\ \mathsf{in}\ \mathsf{lub}.vs - \mathsf{glb}.vs)$$

Note that for any real-valued variable v, $\mathsf{stable}.v \Rrightarrow (\mathit{diff}.v = 0)$.

Sampled real-valued variables in a time band β are related to their true values within an event of β using the *accuracy* of the variable in β [18]. In particular, we let $\mathsf{acc}.v \in \mathit{TimeBand} \to \mathbb{R}^{\geq 0}$ denote the accuracy of variable v in a given time band. The maximum change to v within an event of time band β is an assumption on the environment. To enable this assumption to be stated more succinctly, we define a predicate:

$$\mathit{DIFF}.v \;\widehat{=}\; \forall \beta{:}\,\mathit{Timeband} \bullet \Box(\ell \leq \rho.\beta \Rightarrow \mathit{diff}.v \leq \mathsf{acc}.v.\beta)$$

The lemma below allows one to relate sampled values to the actual values in the environment based on the accuracy of the variables being sampled.

Lemma 7. *If x and y are real-valued variables, β is a time band and $\gg \in \{\geq, >\}$, then both of the following hold:*

$$\begin{aligned} \mathit{DIFF}.x \wedge \mathit{DIFF}.y \wedge \circledcirc_{\rho.\beta}(x - \mathsf{acc}.x.\beta \gg y + \mathsf{acc}.y.\beta) &\Rrightarrow \circledast(x \gg y) \\ \mathit{DIFF}.x \wedge \mathsf{stable}.y \wedge \circledcirc_{\rho.\beta}(x - \mathsf{acc}.x.\beta \gg y) &\Rrightarrow \circledast(x \gg y) \end{aligned}$$

Proof. For any Δ and s, if $(\mathit{DIFF}.x \wedge \ell \leq \rho.\beta).\Delta.s$, then $\boxdot(\overleftarrow{x} + \mathsf{acc}.x.\beta \geq x \geq \overleftarrow{x} - \mathsf{acc}.x.\beta).\Delta.s$ (and similarly for y). Therefore, if $(\mathit{DIFF}.x \wedge \mathit{DIFF}.y \wedge \circledcirc_{\rho.\beta}(x - \mathsf{acc}.x.\beta \gg y + \mathsf{acc}.y.\beta)).\Delta.s$, then $\circledast(x \gg y).\Delta.s$. The proof of the second property is similar. □

4.5. Delay

As shown in Fig. 3, a control system often consists of inputs that are fed back to the controller via the environment. That is, the inputs to a controller from its environment influence the controller's outputs, and these outputs in turn influence the environment. There are typically delays involved in feedback and hence we define the following interval predicate for variables v_i, v_o (respectively representing the input and output), and delay $D \in \mathbb{R}$ where $D \geq 0$.

$$\mathit{delay}(v_i, v_o, D).\Delta.s \;\widehat{=}\; \forall t{:}\,\Delta \bullet s.t.v_o = s.(t - D).v_i$$

Note that there is a separation of concerns between delays and variable accuracy, i.e., when reasoning about delay, one need not worry about the variable accuracy. Delays are strictly concerned with the lag between inputs and outputs.

5. Action systems with time bands

The action systems that we develop must enable reasoning about the assumed behaviour of the environment and the behaviour that the action system is yet to implement. Following Jones [12, 31, 38, 39],

to enable compositionality, the behaviour of the environment of an action system is formalised by its *rely* condition. However, unlike Jones, who assumes rely conditions are interleaved with those of the component under consideration, we assume rely conditions are interval predicates that are assumed to hold over the interval in which an action executes. Our derivation method uses enforced properties [15, 17], which are formulae that restrict the behaviour of the system under development to those that satisfy the formulae. We first present enforced properties on actions, which allow finer-grained control over the execution of an action system.

We present the syntax and semantics of actions and action systems in Section 5.1 and present methods for refining both actions and action systems in Section 5.2.

5.1. Action systems

The controllers we develop involve real-time properties, and hence, one must clearly identify the inputs and outputs of the system components to ensure implementability. The components of the action systems we develop are often part of a larger *context* of variables. The *frame* of a component defines the variables of the context that the component may modify — variables of the context that are outside the frame must be right stable. If a component modelled by interval predicate p has a frame F, we use syntax $F{:}\,[p]$ to denote a framed interval predicate. A framed interval predicate in a larger context $F \subseteq V \subseteq \mathit{Var}$ is denoted $[\![\, F{:}\,[p] \,]\!]_V$, where

$$[\![\, F{:}\,[p] \,]\!]_V \quad \widehat{=} \quad p \wedge \mathsf{stable}.(V \setminus F) \wedge \mathsf{right\_stable}.\, V$$

Here, p formalises the behaviour of the component over an interval. The context V and frame F ensure that variables in the context but outside of the frame are stable, and all variables of the context are right stable. The syntax of an action system may be considered to be shorthand for framed interval predicates. We let $[\![\, p \,]\!]_V$ denote the framed interval predicate $[\![\, \varnothing{:}\,[p] \,]\!]_V$, i.e., a framed interval predicate with an empty frame.

The abstract syntax of actions is given below, which we note differs from the standard syntax of Back *et al* [4, 6, 7].

Definition 5 (action). Suppose b is a state predicate, $\widehat{\mathbf{v}}$, $\mathbf{v}$ and $\mathbf{y}$ are vectors of variables, $\mathbf{e}$ is a vector of expressions, F is a set of variables, p and r are interval predicates, β is a time band, and $D \in \mathbb{R}_{\geq 0}$. The abstract syntax of an *action* A is given by:

$$\begin{array}{lcl} A & ::= & b \rightarrow S \mid A_1 \,[\!]\, A_2 \mid \textsc{Enf}\, p \bullet A \mid \textsc{Rely}\, r \bullet A \mid \mathrm{FB}_D(\widehat{\mathbf{v}} \setminus \mathbf{v}) \bullet A \mid A \dagger \beta \mid A^{\omega} \\ S & ::= & \mathbf{idle} \mid \mathbf{y} := \mathbf{e} \mid F{:}\,[p] \end{array}$$

The primitive **idle** is a *statement* that does nothing but may take time to execute and $\mathbf{y} := \mathbf{e}$ is the *assignment* statement. $F{:}\,[p]$ is a *specification*, which is not directly executable, and hence, needs to be refined to an implementation that can be executed. Action $b \rightarrow S$ is a *guarded action* consisting of statement S with guard b. Action $A_1 \,[\!]\, A_2$ is the non-deterministic choice between A_1 and A_2 and $\textsc{Enf}\, p \bullet A$ denotes an action A with enforced condition p, $\textsc{Rely}\, r \bullet A$ denotes an action A with rely condition r, $\mathrm{FB}_D(\widehat{\mathbf{v}} \setminus \mathbf{v}) \bullet A$ denotes an action A with delayed feedback, $A \dagger \beta$ denotes an action in a timeband β and A^{ω} denotes the possibly infinite iteration of A.

A software controller executes with its environment in a truly concurrent manner, and hence the inputs to the controller may change during the evaluation of the expressions of the action (which includes the guards of the action). We assume that for each iteration of the main loop of the action system, the expressions (including guards) are evaluated so that each input variable is sampled at most once, the expressions are evaluated then the output variables are updated as required. Sampling inputs at most once per sampling interval avoids anomalies when evaluating two guards that refer to the same input variable. For example, guards $x < 0$ and $x \geq 0$ could both evaluate to false (or both to true) within a single sampling interval if x increases past 0 during the interval and the value of x is sampled twice (once for each guard evaluation). Our model of sampling inputs once per sampling interval also avoids anomalies in guarded assignments. For

example, action $x < 0 \rightarrow y := x$ should not assign a positive value to y, however, if different samples are used for the two occurrences of x, it is possible for y to obtain a positive value after execution of the action. Because both sampling and expression evaluation take time and the environment executes in parallel with the action system in a truly concurrent manner, each input variable may have a set of possible values. Hence, we use a sampling logic to distinguish whether a predicate is definitely or possibly true over an interval. Note that the semantics of actions state that at most one sample of each variable is taken and that the same sample is used for each occurrence of the variable.

As we have already mentioned, the syntax in Definition 5 may be considered to be shorthand for framed interval predicates in an output context $V \subseteq \mathit{Var}$. We assume that $\mathbf{y} = \langle y_0, \ldots, y_n \rangle$ is a vector of variables, $\mathbf{e} = \langle e_0, \ldots, e_n \rangle$ is a vector of expressions, $\mathbf{k} = \langle k_0, \ldots, k_n \rangle$ is a vector of constants, and for a vector $\mathbf{v}$, that $\mathit{Set}.\mathbf{v} \mathrel{\widehat{=}} \{v_i \mid i \in \mathrm{dom}.\mathbf{v}\}$. We let $\mathit{vars}.p$ denote the set of free variables of interval predicate p.

$$[\![\, b \rightarrow \mathbf{idle} \,]\!]_V \mathrel{\widehat{=}} \mathsf{fin} \wedge [\![\, \circledast b \,]\!]_V \tag{13}$$

$$[\![\, b \rightarrow \mathbf{y} := \mathbf{e} \,]\!]_V \mathrel{\widehat{=}} \mathsf{fin} \wedge \exists\, \mathbf{k} \bullet [\![\, \circledast(b \wedge \mathbf{k} = \mathbf{e}) \,]\!]_V \,;\; [\![\, \mathit{Set}.\mathbf{y}\colon [\forall i\colon 0..n \bullet (\mathsf{stable}.y_i \,;\; \underline{\boxdot(y_i = k_i)})] \,]\!]_V \tag{14}$$

$$[\![\, b \rightarrow F\colon [p] \,]\!]_V \mathrel{\widehat{=}} (\mathsf{fin} \wedge [\![\, \circledast b \,]\!]_V) \,;\; [\![\, F\colon [p] \,]\!]_V \tag{15}$$

$$[\![\, A_1 \,[]\, A_2 \,]\!]_V \mathrel{\widehat{=}} [\![\, A_1 \,]\!]_V \vee [\![\, A_2 \,]\!]_V \tag{16}$$

$$[\![\, \textsc{Enf}\ p \bullet A \,]\!]_V \mathrel{\widehat{=}} p \wedge [\![\, A \,]\!]_V \tag{17}$$

$$[\![\, \textsc{Rely}\ r \bullet A \,]\!]_V \mathrel{\widehat{=}} r \Rightarrow [\![\, A \,]\!]_V \qquad \text{provided } \mathit{vars}.r \cap V = \varnothing \tag{18}$$

$$[\![\, \mathrm{FB}_D(\widehat{\mathbf{v}} \setminus \mathbf{v}) \bullet A \,]\!]_V \mathrel{\widehat{=}} \exists \widehat{\mathbf{v}} \bullet \mathit{delay}(\mathbf{v}, \widehat{\mathbf{v}}, D) \wedge [\![\, A \,]\!]_V \qquad \text{provided } \mathit{Set}.\widehat{\mathbf{v}} \cap V = \varnothing \wedge \mathit{Set}.\mathbf{v} \subseteq V \tag{19}$$

$$[\![\, A \dagger \beta \,]\!]_V \mathrel{\widehat{=}} [\![\, A \,]\!]_V \wedge \ell \leq \rho.\beta \tag{20}$$

$$[\![\, A^\omega \,]\!]_V \mathrel{\widehat{=}} ([\![\, A \,]\!]_V)^\omega \tag{21}$$

By (13), the guarded action $b \rightarrow \mathbf{idle}$ executes iff b is possibly true and its execution leaves each output variable both stable and right stable. The interval of execution of each $b \rightarrow \mathbf{idle}$ is of finite length, i.e., $b \rightarrow \mathbf{idle}$ must terminate. By (14), a guarded assignment to vector of variables $b \rightarrow \mathbf{y} := \mathbf{e}$ consists of two portions, where the guard and expressions $\mathbf{e}$ are evaluated in the first portion in the same apparent state, and $\mathbf{y}$ is updated to the new values in the second. Furthermore, execution of $b \rightarrow \mathbf{y} := \mathbf{e}$ ensures each output V not in $\mathbf{y}$ is stable and that each variable in V is right stable. By (15), the behaviour of $b \rightarrow F\colon [p]$ holds if it is possible for b to hold in some apparent state, followed by an interval in which the framed interval predicate $[\![\, F\colon [p] \,]\!]_V$ holds. We leave out the guard b if b is *true*, i.e., we write S for $\mathit{true} \rightarrow S$.

By (16), the behaviour of a non-deterministic choice between two actions A_1 and A_2 holds if the behaviour of either A_1 or A_2 holds. By (17), the behaviour of action $\textsc{Enf}\ p \bullet A$ guarantees that both the behaviour of A and the enforced condition p hold. Note that action $\textsc{Enf}\ \mathit{false} \bullet A$ has no behaviours, i.e., it is possible to enforce unimplementable behaviour. Hence, we typically introduce or strengthen an enforced property in the weakest possible manner to allow greater flexibility in an implementation. By (18), an action A with a rely condition behaves as the action under the assumption that the rely condition holds. Note that the behaviour of $\textsc{Rely}\ \mathit{false} \bullet A$ is chaotic, i.e., any behaviour is allowed, and that the behaviour of $\textsc{Rely}\ r \bullet A$ is undefined if $\mathit{vars}.r \cap V \neq \varnothing$. By (19), the behaviour of an action with feedback is the behaviour of the action with the guarantee that the fed back values are delayed by the given amount, by (20) the behaviour of an action in a time band is the behaviour of the action together with the guarantee that the action is completed within the precision of the time band and by (21), the behaviour of an iteration A^ω over an interval is the behaviour of A iterated over the interval. Note that action A^∞ can be treated as a special case of A^ω, namely $A^\omega \sqsupseteq_V A^* \,[]\, A^\infty$.

The syntax of an action system consists of an initialisation followed by a potentially infinite loop that executes a non-deterministic choice over a set of guarded statements [6].

Definition 6 (action system). For a state predicate I, state predicates b_i and statements S_i where $i \in 0..n$, the abstract syntax of an *action system* is given by $\textsc{Init}\ I \bullet \mathbf{do}\ [\!]_{i:0..n}\, b_i \rightarrow S_i\ \mathbf{od}$.

For an action system $\mathscr{A} = \textsc{Init}\ I \bullet \mathbf{do}\ [\!]_{i:0..n}\, b_i \rightarrow S_i\ \mathbf{od}$, we define $\mathit{init}_V.\mathscr{A} \mathrel{\widehat{=}} \overrightarrow{I} \wedge \mathsf{right\_stable}.V$, $\mathit{guard}.\mathscr{A} \mathrel{\widehat{=}} \bigvee_{i:0..n} b_i$, and $\mathit{action}.\mathscr{A} \mathrel{\widehat{=}} [\!]_{i:0..n}\, b_i \rightarrow S_i$.

This paper is concerned with the development of controllers for reactive systems, which are typically non-terminating. Hence, we assume that $\bigvee_{i:0..n} b_i$ holds. To simplify the notation, we define a non-deterministic choice with an **else** case as follows

$$(\mathop{[]}_{i:0..n} b_i \to S_i) \textbf{ else } S_{n+1} \quad \widehat{=} \quad (\mathop{[]}_{i:0..n} b_i \to S_i) \mathbin{[]} (\textstyle\bigwedge_{i:0..n} \neg b_i \to S_{n+1})$$

Prior to defining the semantics of an action system, we must determine its set of input and output variables. We first determine the sets of variables and output variables of statements. Hence, we define functions *AllVars* and *OutVars* as follows:

$$\begin{aligned} \mathit{AllVars}.\textbf{idle} &\mathrel{\widehat{=}} \varnothing & \qquad \mathit{OutVars}.\textbf{idle} &\mathrel{\widehat{=}} \varnothing \\ \mathit{AllVars}.(\mathbf{v} := \mathbf{e}) &\mathrel{\widehat{=}} \mathit{Set}.\mathbf{v} \cup \textstyle\bigcup_{e:\mathit{Set}.\mathbf{e}} \mathit{vars}.e & \qquad \mathit{OutVars}.(\mathbf{v} := \mathbf{e}) &\mathrel{\widehat{=}} \mathit{Set}.\mathbf{v} \\ \mathit{AllVars}.(F\colon [p]) &\mathrel{\widehat{=}} F \cup \mathit{vars}.p & \qquad \mathit{OutVars}.(F\colon [p]) &\mathrel{\widehat{=}} F \end{aligned}$$

The set of output variables and all variables of an action system given by the syntax in Definition 6 are then defined as follows.

$$\mathit{AllVars}.\mathscr{A} \mathrel{\widehat{=}} \mathit{vars}.(\mathit{guard}.\mathscr{A}) \cup \textstyle\bigcup_{i:0..n} \mathit{AllVars}.S_i \qquad \mathit{OutVars}.\mathscr{A} \mathrel{\widehat{=}} \textstyle\bigcup_{i:0..n} \mathit{OutVars}.S_i$$

The set of input variables of an action system $\mathscr{A}$ is given by $\mathit{InVars}.\mathscr{A} \mathrel{\widehat{=}} \mathit{AllVars}.\mathscr{A} \setminus \mathit{OutVars}.\mathscr{A}$.

Like actions, the syntax of an action system may be regarded as shorthand for a framed interval predicate, and may occur within a wider output context.[4] Hence, we have the following definition, where V is a set of variables such that $\mathit{InVars}.\mathscr{A} \cap V = \varnothing$ and $\mathit{OutVars}.\mathscr{A} \subseteq V$.

$$[\![\, \mathscr{A} \,]\!]_V \quad \widehat{=} \quad \ominus \mathit{init}_V.\mathscr{A} \Rightarrow [\![\, (\mathit{action}.\mathscr{A})^\infty \,]\!]_V \tag{22}$$

Note that an initialisation can often refer to output variables, and hence, defining an initialisation using a RELY construct is inadequate. Back and von Wright present an algebraic definition that allows termination of the action system [6]. The methods in this paper apply to reactive systems, which we assume are non-terminating, but can be extended to reason about potentially terminating behaviour in a straightforward manner. To avoid Zeno-like behaviour, we implicitly assume the existence of a constant lower bound $\epsilon \in \mathbb{R}_{>0}$ on the time taken execute each iteration of the action system, i.e., $\mathit{action}.\mathscr{A} \Rrightarrow \ell \geq \epsilon$ is implicitly assumed.

As with actions, we allow the syntax of action systems to be extended with rely and enforced conditions. In particular, we define

$$\begin{aligned} [\![\, \textsc{Enf}\ p \bullet \mathscr{A} \,]\!]_V &\quad \widehat{=} \quad p \wedge [\![\, \mathscr{A} \,]\!]_V \\ [\![\, \textsc{Rely}\ r \bullet \mathscr{A} \,]\!]_V &\quad \widehat{=} \quad r \Rightarrow [\![\, \mathscr{A} \,]\!]_V \qquad \text{provided } \mathit{vars}.r \cap V = \varnothing \end{aligned}$$

The following lemma allows one to make a stronger assumption about each iteration of the action system under consideration.

Lemma 8. *The following holds for any action system $\mathscr{A}$ and set of variables V:*

$$[\![\, \mathscr{A} \,]\!]_V \quad \equiv \quad \ominus \mathit{init}_V.\mathscr{A} \Rightarrow [\![\, \big(\textsc{Rely}\ \ominus(\mathit{init}_V.\mathscr{A} \vee [\![\, \mathit{action}.\mathscr{A} \,]\!]_V) \bullet \mathit{action}.\mathscr{A}\big)^\infty \,]\!]_V$$

Proof. The proof follows trivially by unfolding the definition of an action system. □

Hence, if $[\![\, \mathscr{A} \,]\!]_V$ holds in an interval, then for each iteration of the main action, it is possible to deduce that in some previous interval either the action system was initialised, or some main action was executed.

The lemma below allows one to consider an iterated execution of a single action with the assumption that either the action system has just initialised or a different action was executing in some previous interval. We let A^+ and $A^{\omega+}$ denote the finite and possibly infinite positive iterations of action A, respectively.

[4] In our other work, we defined the semantics of action systems using traces consisting of sequences of adjoining intervals [19, 20]. In this paper, we simplify the semantics by defining the behaviour using the iteration operator, which allows us to avoid a new semantic layer in the framework.

$CP_1 \mathrel{\widehat{=}}$ RELY $r_1 \wedge DIFF.water_1 \wedge DIFF.water_2 \bullet$
ENF (8) $\wedge$ (10) $\bullet$
INIT $I_1 \bullet$
do
$on_1 : [true]$
od

Figure 6: Initial action system for $Pump_1$

$CP_2 \mathrel{\widehat{=}}$ RELY $r_2 \wedge DIFF.water_2 \bullet$
ENF (9) $\wedge$ (11) $\wedge$ (12) $\bullet$
INIT $I_2 \bullet$
do
$on_2 : [true]$
od

Figure 7: Initial action system for $Pump_2$

Lemma 9. *Suppose $\mathscr{A}$ is an action system, V is set of variables, and $action.\mathscr{A} = [\!]_{i:0..n} A_i$ where each A_i is either a guarded statement or specification action. Let $not\_jth.\mathscr{A}.j \mathrel{\widehat{=}} ([\!]_{i:0..j-1} A_i) \,[\!]\, ([\!]_{i:j+1..n} A_i)$, i.e., $not\_jth.\mathscr{A}.j$ consists of all actions of $action.\mathscr{A}$ but not A_j, and suppose*

$$iter.\mathscr{A}.j \mathrel{\widehat{=}} \text{RELY} \ominus(init_V.\mathscr{A} \vee [\![not\_jth.\mathscr{A}.j]\!]_V) \bullet \text{ENF inf} \vee \oplus(\overleftarrow{\neg\, guard.A_j}) \bullet (A_j)^{\omega+}$$

denotes the positive iteration of A_j where either the action system has just been initialised, or some other action was previously executing. Then, we have:

$$[\![\mathscr{A}]\!]_V \quad \equiv \quad \ominus init_V.\mathscr{A} \Rightarrow [\![([\!]_{j:0..n} iter.\mathscr{A}.j)^{\infty}]\!]_V$$

Proof. By Lemma 8, $[\![\mathscr{A}]\!]_V \equiv \ominus init_V.\mathscr{A} \Rightarrow [\![(\text{RELY} \ominus(init_V.\mathscr{A} \vee [\![action.\mathscr{A}]\!]_V) \bullet action.\mathscr{A})^{\infty}]\!]_V$. The proof then follows by case analysis because $\ominus [\![action.\mathscr{A}]\!]_V \equiv \ominus [\![A_j]\!]_V \vee \ominus [\![not\_jth.\mathscr{A}.j]\!]_V$. □

Parallel composition of two action systems may also be defined. We use $\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}$ to denote the parallel composition of action systems $\mathscr{A}$ and $\mathscr{B}$. For the program $\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}$ to be well defined, we require that both $[\![\mathscr{A}]\!]_V$ and $[\![\mathscr{B}]\!]_W$ are well defined, and that the inputs and outputs of $\mathscr{A}$ are distinct from the outputs of $\mathscr{B}$, i.e.,

$$(InVars.\mathscr{A} \cup OutVars.\mathscr{A}) \cap OutVars.\mathscr{B} = \varnothing \tag{23}$$

That is, $\mathscr{B}$ cannot modify the inputs and outputs of $\mathscr{A}$ but the outputs of $\mathscr{A}$ may be used as inputs to $\mathscr{B}$ and furthermore $\mathscr{A}$ and $\mathscr{B}$ may share inputs but not outputs. Hence, $\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}$ is not necessarily equivalent to $\mathscr{B} \,{}_W\overrightarrow{\|}_V \mathscr{A}$. Within $\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}$, action systems $\mathscr{A}$ and $\mathscr{B}$ execute in a truly concurrent manner. If $\mathscr{A}$ and $\mathscr{B}$ are action systems such that (23) holds and V and W are sets of variables such that $V, W \subseteq Var$, then

$$[\![\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}]\!]_{V \cup W} \quad \mathrel{\widehat{=}} \quad [\![\mathscr{A}]\!]_V \wedge [\![\mathscr{B}]\!]_W$$

A special case of parallel composition is *simple parallelism*, denoted $\mathscr{A} \| \mathscr{B}$, where no output of $\mathscr{A}$ is an input to $\mathscr{B}$ and vice versa, i.e., $\mathscr{A} \| \mathscr{B}$ is defined iff (23) $\wedge$ $(OutVars.\mathscr{A} \cap InVars.\mathscr{B} = \varnothing)$ holds. Note that $InVars.\mathscr{A} \cap InVars.\mathscr{B}$ may be non-empty, i.e., $\mathscr{A}$ and $\mathscr{B}$ may share inputs. Unlike $\mathscr{A} \,{}_V\overrightarrow{\|}_W \mathscr{B}$, $\mathscr{A} \| \mathscr{B}$ is equivalent to $\mathscr{B} \| \mathscr{A}$.

Our derivation method starts with the interval predicates that represent the system requirements, which are used to motivate a simple action system. Correctness of this initial action system is guaranteed by restricting its behaviour with enforced properties. This program is then incrementally refined until executable code is obtained.

Example 10. As an example, we specify an initial action system controller for the two-pump system in Section 1 (see Fig. 6 and Fig. 7). We explain the action system CP_1 — a similar explanation applies to CP_2. Controller CP_1 consists of input variables $water_1$ and $water_2$, output variable on_1 and initial condition I_1 (which is yet to be developed). The controller is assumed to operate in an environment that satisfies $DIFF.water_1$ and $DIFF.water_2$, which constrain the maximum rate of change of $water_1$ and $water_2$ in every time band, and condition r_1, which is yet to be determined.

The main action for the initial version of the CP_1 (i.e., the controller of $Pump_1$) is

$$on_1 : [true]$$

which allows the output on_1 to be set to true or false non-deterministically. Although this initial action is liberal and allows arbitrary modification of on_1, execution of the action systems are constrained by the enforced property (8) $\wedge$ (10), which ensures that CP_1 is correct.

We develop the system as the simple parallel composition between CP_1 and CP_2, which allows the pumps to be controlled independently (see Fig. 6 and Fig. 7), i.e., the controller is given by

$$CP \quad \widehat{=} \quad CP_1 \parallel CP_2 \tag{24}$$

Example 11. At this stage of the derivation, it is possible to define some of the properties of the pumps and their effects on the water level of both tanks.

$$\boxdot(Stopped_1 \Rightarrow (\mathring{wa}ter_1 \geq 0) \wedge (\mathring{wa}ter_2 \leq 0)) \tag{Rely-1}$$

$$\boxdot(Stopped_2 \Rightarrow (\mathring{wa}ter_2 \geq 0)) \tag{Rely-2}$$

Hence, by (Rely-1), if $Pump_1$ is stopped, the rate of change of $water_1$ is non-negative and the rate of change of $water_2$ is non-positive. We obtain an inequality for $water_2$ because $Pump_2$ may be causing the water level in $Tank_2$ to drop. Condition (Rely-2) is similar. Note that a condition such as $\boxdot(Running_2 \Rightarrow (\mathring{wa}ter_2 \leq 0))$ would place an implicit restriction on the capacities of the pumps, namely that the $Pump_1$ has a lower capacity than $Pump_2$.

One can also describe the effect that the signal on_i has on $Pump_i$. In particular, it is possible to define the time taken to turn the pumps on and off using the time band of the pump. For simplicity, we assume that the time bands of $Pump_1$ and $Pump_2$ are both TB_{pump}.

Example 12. For $i \in \{1, 2\}$, we have

$$\Box\left(\boxdot on_i \Rightarrow \begin{pmatrix} \ominus \overrightarrow{\neg Running_i} \Rightarrow \\ ((\ell \leq \rho.TB_{pump} \wedge \boxdot Starting_i)\,;\ \boxdot Running_i) \end{pmatrix} \wedge \mathsf{inv}.Running_i\right) \tag{Rely-3}$$

$$\Box\left(\boxdot \neg on_i \Rightarrow \begin{pmatrix} \ominus \overrightarrow{\neg Stopped_i} \Rightarrow \\ ((\ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_i)\,;\ \boxdot Stopped_i) \end{pmatrix} \wedge \mathsf{inv}.Stopped_i\right) \tag{Rely-4}$$

By (Rely-3), for any subinterval Δ of the given interval, if on_i holds throughout Δ, then

1. if $Pump_i$ is not running at the end of some immediately preceding interval of Δ, then $Pump_i$ is in a starting mode for at most the first $\rho.TB_{pump}$ units of Δ before the $Pump_i$ becomes running, and
2. mode $Running_i$ is invariant in Δ, i.e., if $Running_i$ holds at the end of some previous interval of Δ, then $Running_i$ holds throughout Δ.

Condition (Rely-4) is similar. Note that (Rely-3) also allows for $Running_i$ to instantaneously switch from false to true because $\ell \leq \rho.TB_{pump} \wedge \boxdot Starting_i$ is trivially satisfied by an empty interval.

5.2. Action system refinement

Our method of derivation allows programs to be developed in an incremental manner. In particular, we calculate the effect of (partially) developed actions on the enforced properties, which generates new properties and actions. However, unlike [14, 25, 26], we disallow arbitrary modifications to the program. Instead, each change must be justified by a lemma/theorem that ensures each new version is a refinement of the current version [20, 17].

Definition 7 (action refinement). For a set of variables V, an action A is *refined* by an action C, denoted $A \sqsubseteq_V C$, iff $[\![C]\!]_V \Rrightarrow [\![A]\!]_V$ holds. We say $A \sqsupseteq\sqsubseteq_V C$ holds iff both $A \sqsubseteq_V C$ and $C \sqsubseteq_V A$ hold.

An action C refines an action A iff every behaviour of C is a potential behaviour of A. Relation $\sqsubseteq_V$ is clearly reflexive and transitive. Furthermore, because actions in a context are treated as shorthand for framed interval predicates and refinement is defined in terms of universal implication, we obtain a number of straightforward monotonicity rules for refining actions. Below, we assume that both $[\![A]\!]_V$ and $[\![C]\!]_V$ are well defined when we write $A \sqsubseteq_V C$.

$$b \rightarrow S \sqsubseteq_V c \rightarrow S \quad \text{provided } c \Rrightarrow b \tag{Mono-1}$$

$$b \rightarrow \mathbf{y} := \mathbf{e} \sqsubseteq_V b \rightarrow \mathbf{y} := \mathbf{f} \quad \text{provided } \forall \mathbf{k} \bullet \circledast(b \wedge \mathbf{f} = \mathbf{k}) \Rrightarrow \circledast(\mathbf{e} = \mathbf{k}) \tag{Mono-2}$$

$$b \rightarrow F{:}[p] \sqsubseteq_V b \rightarrow G{:}[q] \quad \text{provided } F \supseteq G \text{ and } q \wedge \mathsf{stable}.(V \setminus G) \Rrightarrow p \tag{Mono-3}$$

$$A_1 \,[\!]\, A_2 \sqsubseteq_V C_1 \,[\!]\, C_2 \quad \text{provided } A_1 \sqsubseteq_V C_1 \text{ and } A_2 \sqsubseteq_V C_2 \tag{Mono-4}$$

$$\textsc{Enf}\ p \bullet A \sqsubseteq_V \textsc{Enf}\ q \bullet C \quad \text{provided } q \Rrightarrow p \text{ and } A \sqsubseteq_V C \tag{Mono-5}$$

$$\textsc{Rely}\ r \bullet A \sqsubseteq_V \textsc{Rely}\ q \bullet C \quad \text{provided } r \Rrightarrow q \text{ and } A \sqsubseteq_V C \tag{Mono-6}$$

$$\mathrm{FB}_D(\widehat{\mathbf{v}} \setminus \mathbf{v}) \bullet A \sqsubseteq_V \mathrm{FB}_D(\widehat{\mathbf{v}} \setminus \mathbf{v}) \bullet C \quad \text{provided } A \sqsubseteq_V C \tag{Mono-7}$$

$$A \dagger \beta \sqsubseteq_V C \dagger \gamma \quad \text{provided } A \sqsubseteq_V C \text{ and } \rho.\gamma \leq \rho.\beta \tag{Mono-8}$$

$$A^\infty \sqsubseteq_V C^\infty \quad \text{provided } A \sqsubseteq_V C \tag{Mono-9}$$

By definition, action $\textsc{Enf}\ p \bullet A$ satisfies p, but the program $\textsc{Enf}\ p \bullet A$ may not be executable. Hence, we develop an action C via a series of refinements in the verify-while-develop paradigm such that the behaviour of C satisfies p, allowing one to remove the enforced property p.

The following refinement rules allow one to add/remove rely conditions, enforced conditions and time bands to/from an action.

$$C \sqsubseteq_V \textsc{Enf}\ p \bullet C \tag{AR-1}$$

$$C \sqsubseteq_V C \dagger \beta \tag{AR-2}$$

$$\textsc{Rely}\ r \bullet C \sqsubseteq_V C \tag{AR-3}$$

$$\textsc{Rely}\ r \bullet \textsc{Enf}\ p \bullet C \sqsubseteq_V \textsc{Rely}\ r \bullet C \quad \text{provided } r \wedge [\![C]\!]_V \Rrightarrow p \tag{AR-4}$$

Hence, one may always add an enforced condition, and time band, and remove a rely condition from an action. By (AR-4), one may remove an enforced property under a rely condition if the rely condition together with the action system under consideration imply the enforced property. Note that condition (AR-3) is analogous to weakening the rely condition to *true*, which is similar to weakening a precondition. Using (AR-3) to weaken the rely condition of an action allows one to make syntactic simplifications to an action system without affecting the guard of the action or overall the action system.

It is also straightforward to prove the following distributivity properties for non-deterministic choice and iteration. Note how distributivity within iteration is only possible for rely conditions and enforced properties that split or join.

$$F{:}[p \vee q] \sqsupseteq_V F{:}[p] \,[\!]\, F{:}[q] \tag{Dist-1}$$

$$(\textsc{Enf}\ p \bullet A_1 \,[\!]\, A_2) \sqsupseteq_V (\textsc{Enf}\ p \bullet A_1) \,[\!]\, (\textsc{Enf}\ p \bullet A_2) \tag{Dist-2}$$

$$(\textsc{Rely}\ r \bullet A_1 \,[\!]\, A_2) \sqsupseteq_V (\textsc{Rely}\ r \bullet A_1) \,[\!]\, (\textsc{Rely}\ r \bullet A_2) \tag{Dist-3}$$

$$(\textsc{Rely}\ r \bullet C^\omega) \sqsubseteq_V (\textsc{Rely}\ r \bullet C)^\omega \quad \text{provided } r \text{ splits} \tag{Dist-4}$$

$$(\textsc{Enf}\ p \bullet C^{\omega+}) \sqsubseteq_V (\textsc{Enf}\ p \bullet C)^{\omega+} \quad \text{provided } p \text{ joins} \tag{Dist-5}$$

$$(\textsc{Enf}\ p \bullet C)^\omega \sqsubseteq_V (\textsc{Enf}\ p \bullet C^\omega) \quad \text{provided } p \text{ splits} \tag{Dist-6}$$

$$(\textsc{Enf}\ p^\omega \bullet C^\omega) \sqsubseteq_V (\textsc{Enf}\ p \bullet C)^\omega \tag{Dist-7}$$

$$(A_1 \,[\!]\, A_2) \dagger \beta \sqsupseteq_V (A_1 \dagger \beta) \,[\!]\, (A_2 \dagger \beta) \tag{Dist-8}$$

The distributivity rules for $^\omega$ can be extended to rules about $^\infty$ in a straightforward manner. We present miscellaneous refinement rules below.

$$A \sqsupseteq_V A \,[\!]\, C \quad \text{provided } A \sqsubseteq_V C \tag{Act-Intro-Rem}$$

$$A_1 \,[\!]\, A_2 \sqsubseteq_V A_1 \qquad \text{(Reduce-ND)}$$

$$\textsc{Enf}\ p_1 \bullet (\textsc{Enf}\ p_2 \bullet A) \sqsupseteq_V \textsc{Enf}(p_1 \wedge p_2) \bullet A \qquad \text{(Enf-Assoc)}$$

$$\textsc{Rely}\ r_1 \bullet (\textsc{Rely}\ r_2 \bullet A) \sqsupseteq_V \textsc{Rely}(r_1 \wedge r_2) \bullet A \qquad \text{(Rely-Shunt)}$$

Rule (Act-Intro-Rem) allows one to introduce and remove actions during a derivation, (Reduce-ND) reduces the non-determinism, (Enf-Assoc) is akin to associativity for enforced properties and (Rely-Shunt) is akin to shunting for rely conditions.

The timeband of a (digital) controller is often of higher precision than the timeband of the (physical) components being controlled. For example, several sampling periods may take place before a stopped pump becomes running because the timeband of the controller is of higher precision than the timeband of the pump. To simplify reasoning about such systems, for a natural number i, we define a finite iteration of an action A to be an action A^i, where

$$[\![\, A^i \,]\!]_V \;\widehat{=}\; \begin{cases} \mathsf{empty} & \text{if } i = 0 \\ [\![\, A \,]\!]_V \,;\; [\![\, A^{i-1} \,]\!]_V & \text{otherwise} \end{cases}$$

For timebands β and γ, we have the following refinement property for a finitely iterated action A^i.

$$A^i \dagger \beta \sqsubseteq_V (A \dagger \gamma)^i \qquad \text{provided } i \times \rho.\gamma \leq \rho.\beta \qquad \text{(Fin-Iter)}$$

The lemma below states that a positive iteration of a guarded statement can be reduced to a single iteration if the action under consideration falsifies its own guard.

Lemma 10. *Suppose V is a set of variables, $x \in V$ is a boolean variable. Then both of the following hold:*

$$(x \to x := \mathit{false})^{\omega+} \;\sqsupseteq_V\; x \to x := \mathit{false}$$
$$(\neg x \to x := \mathit{true})^{\omega+} \;\sqsupseteq_V\; \neg x \to x := \mathit{true}$$

Proof. The first property is proved as follows. The proof of the second is analagous. We assume Empty is an action whose behaviour is empty and False an action whose behaviour is false.

$(x \to x := \mathit{false})^{\omega+}$
$\sqsupseteq_V$ definition
$(x \to x := \mathit{false})\,;\ (x \to x := \mathit{false})^{\omega}$
$\sqsupseteq_V$ unfolding ($^\omega$)
$(x \to x := \mathit{false})\,;\ (\mathsf{Empty} \,[\!]\, ((x \to x := \mathit{false})\,;\ (x \to x := \mathit{false})^{\omega}))$
$\sqsupseteq_V$ distributivity and logic
$(x \to x := \mathit{false}) \,[\!]\, ((x \to x := \mathit{false})\,;\ (x \to x := \mathit{false})\,;\ (x \to x := \mathit{false})^{\omega})$
$\sqsupseteq_V$ behaviour of assignment
$(x \to x := \mathit{false}) \,[\!]\, ((x \to x := \mathit{false})\,;\ \mathsf{False}\,;\ (x \to x := \mathit{false})^{\omega})$
$\sqsupseteq_V$ False is a left annihilator
$(x \to x := \mathit{false}) \,[\!]\, ((x \to x := \mathit{false})\,;\ \mathsf{False})$
$\sqsupseteq_V$ assignments are terminating, logic
$(x \to x := \mathit{false}) \,[\!]\, \mathsf{False}$
$\sqsupseteq_V$ logic
$x \to x := \mathit{false}$ □

Unlike actions, refinement of action systems is defined in terms of a potentially wider context. We consider notions such as data refinement [13] to be part of future work. However, our notion of refinement allows fresh variables to be introduced to an implementation.

Definition 8 (action system refinement). An action system $\mathscr{A}$ is *refined* by an action system $\mathscr{C}$, denoted $\mathscr{A} \sqsubseteq \mathscr{C}$, iff for any context $V \subseteq \mathit{Var}$ such that $\mathit{OutVars}.\mathscr{C} \subseteq \mathit{OutVars}.\mathscr{A} \subseteq V$, $[\![\, \mathscr{C} \,]\!]_V \Rrightarrow [\![\, \mathscr{A} \,]\!]_V$. We say $\mathscr{A} \sqsupseteq \mathscr{C}$ iff both $\mathscr{A} \sqsubseteq \mathscr{C}$ and $\mathscr{C} \sqsubseteq \mathscr{A}$ hold.

An action system is also defined to be shorthand for a framed interval predicate (see (22)), hence, the refinement rules for actions may be extended to action systems in a straightforward manner.

Lemma 11. *Refinement of the parallel composition of action systems is monotonic.*

$$\mathscr{A}\ {}_V\overrightarrow{\|}_W \mathscr{B} \sqsubseteq \mathscr{C}\ {}_V\overrightarrow{\|}_W \mathscr{D} \qquad \textit{provided } [\![\,\mathscr{C}\,]\!]_V \Rrightarrow [\![\,\mathscr{A}\,]\!]_V \textit{ and } [\![\,\mathscr{D}\,]\!]_W \Rrightarrow [\![\,\mathscr{B}\,]\!]_W \qquad \text{(Par-Comp)}$$

Note that if $\mathscr{A} \sqsubseteq \mathscr{C}$ and $\mathscr{B} \sqsubseteq \mathscr{D}$ hold, then both $[\![\,\mathscr{C}\,]\!]_V \Rrightarrow [\![\,\mathscr{A}\,]\!]_V$ and $[\![\,\mathscr{D}\,]\!]_W \Rrightarrow [\![\,\mathscr{B}\,]\!]_W$ hold, and hence, $\mathscr{A}\ {}_V\overrightarrow{\|}_W \mathscr{B} \sqsubseteq \mathscr{C}\ {}_V\overrightarrow{\|}_W \mathscr{D}$ holds.

Theorem 12. *An action system $\mathscr{A}$ is* refined *by an action system $\mathscr{C}$, denoted $\mathscr{A} \sqsubseteq \mathscr{C}$, iff for any $V \subseteq Var$ such that $OutVars.\mathscr{C} \subseteq OutVars.\mathscr{A} \subseteq V$, $init_V.\mathscr{A} \Rrightarrow init_V.\mathscr{C}$ and $action.\mathscr{A} \sqsubseteq_V action.\mathscr{C}$.*

Proof. We are required to prove that $[\![\,\mathscr{C}\,]\!]_V \Rrightarrow [\![\,\mathscr{A}\,]\!]_V$, which by definition and logic holds if

$$\big(\ominus\, init_V.\mathscr{C} \Rightarrow [\![\,(action.\mathscr{C})^\infty\,]\!]_V\big) \wedge \ominus\, init_V.\mathscr{A} \quad \Rrightarrow \quad [\![\,(action.\mathscr{A})^\infty\,]\!]_V$$

By monotonicity and because the guards of $\mathscr{A}$ and $\mathscr{C}$ are total, this holds if $init_V.\mathscr{A} \Rrightarrow init_V.\mathscr{C}$ and $[\![\,action.\mathscr{C}\,]\!]_V \Rrightarrow [\![\,action.\mathscr{A}\,]\!]_V$, which are discharged by our assumptions. □

As an example, we perform a simple refinement by introducing some additional enforced properties that express *maintenance conditions*. These strengthen the given progress properties so that the pump is not arbitrarily turned on and off. To this end, we define a maintenance operator [19], which states that a property c_1 is maintained unless the variables are sampled in a manner that c_2 holds. For state predicates c_1 and c_2, we define:

$$c_1\, \mathcal{M}\, c_2 \quad \widehat{=} \quad \ominus \overrightarrow{c_1} \Rightarrow \boxdot c_1 \vee \circledast c_2$$

Hence $(c_1\, \mathcal{M}\, c_2).\Delta$ holds if provided that c_1 holds at the end of some interval that immediately precedes Δ then either c_1 holds throughout Δ, or $\circledast c_2$ holds. As there is no upper limit on the length of the intervals, the variables in $\circledast c_2$ could be sampled at widely different times, which is potentially problematic. However, $c_1\, \mathcal{M}\, c_2$ are typically requirements on the execution of a single iteration of an action system, and hence the times at which the variables in c_2 are sampled are restricted by the time band of the action system.

A maintenance property on actions is related to properties of the program as follows. In particular, if a possibly infinite iteration of $c_1\, \mathcal{M}\, c_2$ holds in interval Δ and c_1 holds at the end of some immediately preceding interval of Δ, then either c_1 holds throughout Δ or there is an initial portion of Δ in which $\boxdot c_1$ holds and a suffix of Δ in which $\circledast c_2$ holds. Note that

$$((\ell \leq d) \wedge (c_1\, \mathcal{M}\, c_2))^\omega \wedge \ominus \overrightarrow{c_1} \quad \Rrightarrow \quad \boxdot c_1 \vee \overleftarrow{\Diamond}(\boxdot c_1\ ;\ \circledast_d c_2)$$

Furthermore, because $\boxdot c_1$ holds in an empty interval, the consequent above is equivalent to the interval predicate $\boxdot c_1 \vee (\boxdot c_1\ ;\ \overleftarrow{\Diamond}\circledast_d c_2) \vee \overleftarrow{\Diamond}\circledast_d c_2$ i.e., it is possible for c_2 to immediately evaluate to true.

Example 13. For controller CP_1, enforced property (8) expresses a requirement for the times at which $Pump_1$ *must* be stopped and (10) expresses the conditions under which $Pump_1$ *must* become running. However, condition (10) does not disallow a running pump to be turned off arbitrarily. There are also no restrictions on turning the pump on. Similar arguments also apply to CP_2, and hence, we introduce the maintenance requirements below. Note that the conditions for turning the pumps on and off are not symmetric.

M1. If on_1 holds then it continues to hold unless $water_1$ is at low_limit_1 or below, or $water_2$ is at $high\_limit_2$ or above.

M2. If $\neg on_1$ holds then it continues to hold unless $water_1$ is above $reserve_1$ and $water_2$ is below $reserve_2$.

M3. If on_2 holds then it continues to hold unless the button is released, or $water_2$ is at low_limit_2 or below.

$MP_1 \mathrel{\widehat{=}}$
RELY $r_1 \wedge DIFF.water_1 \wedge DIFF.water_2$ •
ENF (8) $\wedge$ (10) •
INIT I_1 •
do
  $\boxed{\text{ENF (M1)} \wedge \text{(M2)} \bullet on_1\colon [true]}$
od

Figure 8: Introduce maintenance condition to CP_1

$MP_2 \mathrel{\widehat{=}}$
RELY $r_2 \wedge DIFF.water_2$ •
ENF (9) $\wedge$ (11) $\wedge$ (12) •
INIT I_2 •
do
  $\boxed{\text{ENF (M3)} \wedge \text{(M4)} \bullet on_2\colon [true]}$
od

Figure 9: Introduce maintenance condition to CP_2

M4. If $\neg on_2$ holds then it continues to hold unless the button is pressed and $water_2$ is above $reserve_2$.

Note that condition **M1** must allow $Pump_1$ to be turned off before $water_1$ drops below $empty_1$ because by **T1**, the pump must (physically) be off if $water_1$ ever reaches $empty_1$. A similar argument applies to **M2**, **M3** and **M4**. These properties are formalised below using the maintains unless operator.

$$on_1 \mathcal{M} ((water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)) \tag{M1}$$
$$\neg on_1 \mathcal{M} ((water_1 > reserve_1) \wedge (water_2 < reserve_2)) \tag{M2}$$
$$on_2 \mathcal{M} (\neg pressed \vee (water_2 \leq low\_limit_2)) \tag{M3}$$
$$\neg on_2 \mathcal{M} (pressed \wedge (water_2 > reserve_2)) \tag{M4}$$

We introduce (M1) and (M2) to the main action of CP_1, and (M3) and (M4) to the main action of CP_2 to obtain the action systems MP_1 and MP_2 in Fig. 8 and Fig. 9, respectively. By (Par-Comp) and then (Mono-5) twice, we can deduce that $CP \sqsubseteq MP_1 \| MP_2$.

6. Derivation

In this section, we present a derivation of an action system controller for both pumps. We first refine the safety and progress conditions in Sections 6.1 and 6.2, respectively. We then derive the actions for both controllers in 6.3 and discharge all remaining proof obligations in 6.4.

6.1. Refine safety conditions

We first rework the safety requirement (8) so that it is stated in terms of inputs and outputs. The proof requires that we strengthen the initialisation I_1 of the action system controller so that the following holds:

$$I_1 \Rightarrow \neg on_1 \wedge Stopped_1 \tag{Init-1}$$

Supposing $X \mathrel{\widehat{=}} (water_1 \leq empty_1) \vee (water_2 \geq full_2)$, we obtain the calculation below.

$$\begin{array}{ll}
 & \ominus(\overrightarrow{I_1} \wedge \mathsf{right_stable}.on_1) \Rightarrow (8) \\
\Leftarrow & \quad \text{condition (Init-1)} \\
 & \ominus(\overrightarrow{\neg on_1 \wedge Stopped_1}) \wedge \overleftarrow{\Diamond}\boxdot\neg on_1 \Rightarrow \boxdot(X \Rightarrow Stopped_1) \\
\Leftarrow & \quad \text{by (Rely-4), } \ominus \overrightarrow{Stopped_1} \wedge \boxdot\neg on_1 \Rrightarrow \boxdot Stopped_1 \\
 & \ominus(\overrightarrow{\neg on_1 \wedge Stopped_1}) \wedge \overleftarrow{\Diamond}\boxdot(\neg on_1 \wedge Stopped_1) \Rightarrow \boxdot(X \Rightarrow Stopped_1) \\
\Leftarrow & \quad \text{logic } \overleftarrow{\Diamond}\boxdot c \equiv \boxdot c \vee (\boxdot c \,;\, \overleftarrow{\neg c}) \\
 & \ominus(\overrightarrow{\neg on_1 \wedge Stopped_1}) \wedge (\boxdot(\neg on_1 \wedge Stopped_1) \vee (\boxdot(\neg on_1 \wedge Stopped_1) \,;\, \overleftarrow{on_1})) \Rightarrow \boxdot(X \Rightarrow Stopped_1) \\
\Leftarrow & \quad \boxdot Stopped_1 \Rrightarrow \boxdot(X \Rightarrow Stopped_1), \text{ logic} \\
 & \Big(\big(\ominus(\overrightarrow{\neg on_1 \wedge Stopped_1}) \wedge \boxdot(\neg on_1 \wedge Stopped_1)\big) \,;\, (\ominus \overrightarrow{\neg on_1} \wedge \overleftarrow{on_1})\Big) \Rightarrow \boxdot(X \Rightarrow Stopped_1) \\
\Leftarrow & \quad \boxdot c \text{ joins}
\end{array}$$

$$\Big(\Big(\ominus(\overrightarrow{\neg on_1 \wedge Stopped_1}) \wedge \boxdot(\neg on_1 \wedge Stopped_1)\Big) \Rightarrow \boxdot(X \Rightarrow Stopped_1)\Big)\ ;\ ((\ominus \overrightarrow{\neg on_1} \wedge \overleftarrow{on_1}) \Rightarrow \boxdot(X \Rightarrow Stopped_1))$$

$\Leftarrow$ $\quad \boxdot Stopped_1 \Rrightarrow \boxdot(X \Rightarrow Stopped_1)$
$\quad true\ ;\ ((\ominus \overrightarrow{\neg on_1} \wedge \overleftarrow{on_1}) \Rightarrow \boxdot(X \Rightarrow Stopped_1))$

To satisfy the condition above, because $\boxdot(X \Rightarrow Stopped_1)$ joins, the property above holds by (5) of Lemma 5 provided each of the following holds.

$$NZ.on_1 \tag{on1-NZ}$$
$$\Box(\ominus \overrightarrow{\neg on_1} \wedge \boxdot on_1 \Rightarrow \boxdot(X \Rightarrow Stopped_1)) \tag{25}$$
$$\Box(\ominus \overrightarrow{on_1} \wedge \boxdot \neg on_1 \Rightarrow \boxdot(X \Rightarrow Stopped_1)) \tag{26}$$

Condition (25) may be satisfied by enforcing the stronger condition:

$$\boxdot(on_1 \Rightarrow (water_1 > empty_1) \wedge (water_2 < full_2)) \tag{S1}$$

That is, whenever signal on_1 holds, $\neg X$ must hold. For (26), we have

(26)
$\equiv$ $\quad$ case analysis
$\Box\Big(\ominus \overrightarrow{on_1} \wedge \boxdot \neg on_1 \wedge (\ominus \overrightarrow{\neg Stopped_1} \vee \ominus \overrightarrow{Stopped_1}) \Rightarrow \boxdot(X \Rightarrow Stopped_1)\Big)$
$\equiv$ $\quad$ (Rely-4)
$\Box\big(\ominus \overrightarrow{on_1} \wedge (((\ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_1)\ ;\ \boxdot Stopped_1) \vee \boxdot Stopped_1) \Rightarrow \boxdot(X \Rightarrow Stopped_1)\big)$
$\equiv$ $\quad$ logic, $\boxdot Stopped_1$ case is trivial
$\Box\big(((\ominus \overrightarrow{on_1} \wedge \ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_1)\ ;\ \boxdot Stopped_1) \Rightarrow \boxdot(X \Rightarrow Stopped_1)\big)$
$\Leftarrow$ $\quad \boxdot c$ joins, the suffix for which $Stopped_1$ holds is trivial
$\Box\big(\ominus \overrightarrow{on_1} \wedge \ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_1 \Rightarrow \boxdot(X \Rightarrow Stopped_1)\big)$

Because $Stopping_1 \Rightarrow \neg Stopped_1$, the final property above may be satisfied by enforcing

$$\Box(\ominus \overrightarrow{on_1} \wedge \ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_1 \Rightarrow \boxdot(water_1 > empty_1) \wedge \boxdot(water_2 < full_2)) \tag{S2}$$

We perform a similar derivation for (9). In particular, we strengthen the initialisation I_2 and require

$$I_2 \Rightarrow \overrightarrow{\neg on_2 \wedge Stopped_2} \tag{Init-2}$$

Condition (9) then holds if each of the following holds.

$$NZ.on_2 \tag{on2-NZ}$$
$$\Box(\ominus \overrightarrow{\neg on_2} \wedge \boxdot on_2 \Rightarrow \boxdot(\neg Stopped_2 \Rightarrow (water_2 > empty_2))) \tag{27}$$
$$\Box(\ominus \overrightarrow{on_2} \wedge \boxdot \neg on_2 \Rightarrow \boxdot(\neg Stopped_2 \Rightarrow (water_2 > empty_2))) \tag{28}$$

Condition (27) is implied by (S3) below and for (28), we use case analysis and (Rely-4), which allows us to reduce the conjunct to (S4) below.

$$\boxdot(on_2 \Rightarrow (water_2 > empty_2)) \tag{S3}$$
$$\Box(\ominus \overrightarrow{on_2} \wedge \ell \leq \rho.TB_{pump} \wedge \boxdot Stopping_2 \Rightarrow \boxdot(water_2 > empty_2)) \tag{S4}$$

Using (Par-Comp) then (Mono-5) twice, we replace (8) in Fig. 6 by the stronger property (S1) $\wedge$ (S2) and (9) in Fig. 7 by (S3) $\wedge$ (S4). Because $\boxdot c$ joins for any state predicate c, (S1) and (S3) hold if $(S1)^\infty$ and $(S3)^\infty$ hold, respectively. Hence (S1) and (S3) may be turned into enforced properties of the main action. We obtain the action systems in Fig. 10 and Fig. 11 where $MP_1 \| MP_2 \sqsubseteq SP_1 \| SP_2$.

$SP_1 \mathrel{\widehat{=}}$
RELY $r_1 \wedge DIFF.water_1 \wedge DIFF.water_2$ •
ENF $\boxed{\text{(S2)} \wedge \text{(on1-NZ)}} \wedge$ (10) •
INIT I_1 •
do
  ENF (M1) $\wedge$ (M2) $\wedge \boxed{\text{(S1)}}$ • on_1: $[true]$
od

Figure 10: Refining safety condition in MP_1

$SP_2 \mathrel{\widehat{=}}$
RELY $r_2 \wedge DIFF.water_2$ •
ENF $\boxed{\text{(S4)} \wedge \text{(on2-NZ)}} \wedge$ (11) $\wedge$ (12) •
INIT I_2 •
do
  ENF (M3) $\wedge$ (M4) $\wedge \boxed{\text{(S3)}}$ • on_2: $[true]$
od

Figure 11: Refining safety condition in MP_2

6.2. Refine progress conditions

We use the following lemma to refine the progress conditions of the program. A proof of a similar lemma is given in [19].

Lemma 13. *Suppose c_1 and c_2 are state predicates, $\epsilon, n \in \mathbb{R}_{>0}$. Then*

$$((\epsilon \le \ell \le n) \wedge (\circledast c_1 \Rightarrow \oplus \overleftarrow{c_2}))^{\infty} \quad \Rightarrow \quad \circledast c_1 \wedge (\ell \ge 2n) \rightsquigarrow \underline{\odot c_2}$$

Proof. Suppose interval Ω and stream s are arbitrarily chosen and that $((\epsilon \le \ell \le n) \wedge (\circledast c_1 \Rightarrow \oplus \overleftarrow{c_2}))^{\infty}.\Omega.s$ holds. Then there exists an infinite-size partition $z \in part.\Omega$ such that for each $i \in \mathrm{dom}.z$,

$$((\epsilon \le \ell \le n) \wedge (\circledast c_1 \Rightarrow \oplus \overleftarrow{c_2})).(z.i).s$$

holds, i.e.,

$$((\epsilon \le \ell \le n) \wedge \circledast c_1).(z.i).s \Rightarrow \exists \Delta\colon sub.(\textstyle\bigcup_{j:\mathbb{N} \wedge j > i} z.j) \bullet \underline{\odot c_2}.\Delta.s \tag{29}$$

For the consequent, we have the following calculation.

$$\begin{array}{ll} & (\circledast c_1 \wedge (\ell \ge 2n) \rightsquigarrow \underline{\odot c_2}).\Omega.s \\ = & \quad \text{definitions} \\ & \forall \Delta\colon suffix.\Omega \bullet (\exists \Delta_1\colon prefix.\Delta \bullet (\circledast c_1 \wedge (\ell \ge 2n)).\Delta_1.s) \Rightarrow \exists \Delta_2\colon sub.\Delta \bullet \underline{\odot c_2}.\Delta_2.s \\ = & \quad \text{logic} \\ & \forall \Delta\colon suffix.\Omega \bullet \forall \Delta_1\colon prefix.\Delta \bullet (\circledast c_1 \wedge (\ell \ge 2n)).\Delta_1.s \Rightarrow \exists \Delta_2\colon sub.\Delta \bullet \underline{\odot c_2}.\Delta_2.s \end{array}$$

Hence, by (29), the proof holds if for an arbitrarily chosen $\Delta \in suffix.\Omega$ and $\Delta_1 \in prefix.\Delta$, whenever

$$(\circledast c_1 \wedge (\ell \ge 2n)).\Delta_1.s$$

then for any partition $z \in part.\Omega$ such that $(\epsilon \le \ell \le n).(z.i)$ for each $i \in \mathrm{dom}.z$, there exists $j \in \mathrm{dom}.z$ such that $z.j \in sub.\Delta_1$. Such a j is guaranteed to exist due to the length of Δ_1. Because $\circledast c_1$ splits, we have $(\epsilon \le \ell \le n \wedge \circledast c_1).(z.j).s$, and the result follows by assumption (29). □

Hence, $\circledast c_1 \wedge (\ell \ge 2n) \rightsquigarrow \underline{\odot c_2}$ holds in an interval if there is an infinite iteration of the interval predicate $(\epsilon \le \ell \le n) \wedge (\circledast c_1 \Rightarrow \oplus \overleftarrow{c_2})$, which states that c_2 is established whenever $\circledast c_1$ holds and the length of each iteration is at most n.

Given the properties below

$$\circledast((water_1 > reserve_1) \wedge (water_2 < reserve_2)) \Rightarrow \oplus \overleftarrow{on_1} \tag{P1}$$

$$\circledast((water_2 > reserve_2) \wedge pressed) \Rightarrow \oplus \overleftarrow{on_2} \tag{P2}$$

$$\circledast \neg pressed \Rightarrow \oplus \overleftarrow{\neg on_2} \tag{P3}$$

by using Lemma 13, properties (10), (11) and (12) hold if $((\ell \leq \frac{\rho.TB_{water}}{2}) \wedge (\text{P1}))^{\infty}$, $((\ell \leq \frac{\rho.TB_{water}}{2}) \wedge (\text{P2}))^{\infty}$ and $((\ell \leq \frac{\rho.TB_{water}}{2}) \wedge (\text{P3}))^{\infty}$ hold, respectively. Condition $\ell \leq \frac{\rho.TB_{water}}{2}$ can be satisfied by using (AR-2) to introduce a time band TB_{cont} to the action system together with the requirement

$$\rho.TB_{cont} \leq \frac{\rho.TB_{water}}{2} \qquad \text{(Rely-5)}$$

which states that the precision of the time band of the controller is at most half the precision of the water time band. Conditions (P1), (P2) and (P3) may be satisfied by strengthening the enforced property of the main action of SP_1 and SP_2. Thus, we obtain the programs in Fig. 12 and Fig. 13. By (Par-Comp), then (Mono-5) and (Dist-7) twice, we have the refinement $SP_1 \| SP_2 \sqsubseteq PP_1 \| PP_2$.

$PP_1 \mathrel{\widehat{=}}$
RELY $r_1 \wedge DIFF.water_1 \wedge DIFF.water_2$ •
ENF (S2) $\wedge$ (on1-NZ) •
INIT I_1 •
do
  ENF (M1) $\wedge$ (M2) $\wedge$ (S1) $\wedge$ $\boxed{\text{(P1)}}$ •
  $on_1 : [true]$
  $\boxed{\dagger\, TB_{cont}}$
od

Figure 12: Refining progress condition in SP_1

$PP_2 \mathrel{\widehat{=}}$
RELY $r_2 \wedge DIFF.water_2$ •
ENF (S4) $\wedge$ (on2-NZ) •
INIT I_2 •
do
  ENF (M3) $\wedge$ (M4) $\wedge$ (S3) $\wedge$ $\boxed{\text{(P2)} \wedge \text{(P3)}}$ •
  $on_2 : [true]$
  $\boxed{\dagger\, TB_{cont}}$
od

Figure 13: Refining progress condition in SP_2

6.3. Refine actions

We are now in a position to introduce statements to the action system by refining the guarded specification statements. We focus on the derivation of the controller for $Pump_1$ and elide the full details of the $Pump_2$ controller. Like [17, 19, 20], our strategy is to leave the framed specification statement until the final derivation step to enable undoing of previous derivations if desired. We also defer calculations against the system-level safety condition (S2) to a later stage of the derivation.

Set on_1 to true. The progress property (P1) forces one to introduce a guarded statement that sets on_1 to true if $(water_1 > reserve_1) \wedge (water_2 < reserve_2)$ holds.[5] Such an action does not conflict with (M2) and to ensure that there is no conflict with (M1), we add a conjunct $\neg on_1$ to the guard. Hence, we obtain:

$$\neg on_1 \wedge (water_1 > reserve_1) \wedge (water_2 < reserve_2) \rightarrow on_1 := true \dagger TB_{cont} \qquad \text{(on1-true)}$$

The action (on1-true) above may conflict with (S1) because $(water_1 > empty_1) \wedge (water_2 < full_2)$ may be *false* (i.e., $(water_1 \leq empty_1) \vee (water_2 \geq full_2)$ may hold) when on_1 is set to *true*. To solve this, we use the guard $(water_1 > reserve_1) \wedge (water_2 < reserve_2)$ of (on1-true), rely condition $DIFF.water_1 \wedge DIFF.water_2$, time band TB_{cont} of PP_1 and introduce an assumption:

$$(reserve_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{cont}) \wedge (reserve_2 \leq full_2 - \mathsf{acc}.water_2.TB_{cont}) \qquad \text{(Rely-6)}$$

We have the following calculation:

$$\begin{array}{ll} & [\![\, (\neg\, on_1 \wedge (water_1 > reserve_1) \wedge (water_2 < reserve_2) \rightarrow on_1 := true) \dagger\, TB_{cont} \,]\!]_{on_1} \\ \Rrightarrow & \text{Lemma 7, (Rely-6) and observations above} \end{array}$$

[5] Because we require that each action system contains an **else** case, at least one of the guards of the action system would have covered the case $(water_1 > reserve_1) \wedge (water_2 < reserve_2)$ and by requirement $(P1)$, would have forced the introduction of $on_1 := true$.

$\circledast(water_1 \geq empty_1) \wedge \circledast(water_2 \leq full_2)$
$\Rrightarrow$ $\circledast c \Rrightarrow \boxdot c$, logic
$\boxdot((water_1 \geq empty_1) \wedge (water_2 \leq full_2))$
$\Rrightarrow$ logic
(S1)

Introduction of statement (on1-true) to the program is justified as follows:

ENF (M1) $\wedge$ (M2) $\wedge$ (S1) $\wedge$ (P1) $\bullet$ $on_1 : [true]$
$\sqsubseteq_V$ (Act-Intro-Rem)
ENF (M1) $\wedge$ (M2) $\wedge$ (S1) $\wedge$ (P1) $\bullet$ $on_1 : [true]$
$[]$ ENF (M1) $\wedge$ (M2) $\wedge$ (S1) $\wedge$ (P1) $\bullet$ $(water_1 > reserve_1) \wedge (water_2 < reserve_2) \rightarrow on_1 := true$
$\sqsubseteq_V$ discussion above, (AR-4)
ENF (M1) $\wedge$ (M2) $\wedge$ (S1) $\wedge$ (P1) $\bullet$ $on_1 : [true]$
$[]$ $(water_1 > reserve_1) \wedge (water_2 < reserve_2) \rightarrow on_1 := true$

Set on_1 to false. By contrapositivity, the safety property (S1) requires on_1 to be *false* if $(water_1 \leq empty_1) \vee (water_2 \geq full_2)$ holds. Hence, we derive a second statement that sets on_1 to false. As an initial attempt we have:

$$(water_1 \leq empty_1) \vee (water_2 \geq full_2) \rightarrow on_1 := false \tag{30}$$

The guard of this condition is however too weak because for example, $water_1$ may continue to drop after $water_1 \leq empty_1$ has been evaluated, but before on_1 is set to false. This is discovered formally because it is impossible to prove the required property:

$$[\![(water_1 \leq empty_1) \vee (water_2 \geq full_2) \rightarrow on_1 := false]\!]_{on_1} \quad \Rrightarrow \quad \text{(S1)}$$

Hence, we leave (S1) enforced for the time being. To ensure that (M1) and (M2) are satisfied, we modify action (30) as follows:

$$\text{ENF (S1)} \bullet on_1 \wedge ((water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)) \rightarrow on_1 := false \dagger TB_{cont} \qquad \text{(on1-false)}$$

Conjunct $(water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)$ of the guard ensures (M1) and conjunct on_1 ensures (M2). Finally, to satisfy progress requirement (P1), we require

$$(low\_limit_1 \leq reserve_1) \wedge (high\_limit_2 \geq reserve_2) \tag{Rely-7}$$

which together with conjunct $(water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)$ of the guard falsifies the antecedent of (P1). Introduction of (on1-false) to the program is justified using (Act-Intro-Rem) and the discussion above.

An **idle** *action.* The action system we aim to derive implements a reactive system, and hence, the disjunction of all guards of the action system must simplify to *true*. Hence, it is common to include an 'else' branch whose corresponding statement is **idle**, which leaves the state unmodified. However, because the environment of the action system is assumed to execute with the action system in a truly concurrent manner, there is no guarantee that execution of **idle** preserves the system requirements.

We aim to remove the original guarded specification action and the enforced property within (on1-false) at a later stage of the derivation and the **idle** action will correspond to an **else** case. Hence we consider the conjunction of the negation of the guards of (on1-true) and (on1-false) and the action below.

$$\begin{array}{l}(on_1 \Rightarrow (water_1 > low\_limit_1) \wedge (water_2 < high\_limit_2)) \wedge \\ (\neg on_1 \Rightarrow (water_1 \leq reserve_1) \vee (water_2 \geq reserve_2))\end{array} \rightarrow \textbf{idle} \dagger TB_{cont} \tag{idle}$$

Action (idle) trivially satisfies (M1) and (M2). For (P1), we perform case analysis on $\overleftarrow{on_1}$. For case $\overleftarrow{on_1}$, by the behaviour of a guarded **idle** action, we have $\oplus \overleftarrow{on_1}$, and hence, (P1) holds trivially. For case $\overleftarrow{\neg on_1}$, the

$AP_1 \mathrel{\widehat{=}}$
RELY $r_1 \wedge DIFF.water_1 \wedge DIFF.water_2$ •
ENF (S2) $\wedge$ (on1-NZ) •
INIT I_1 •
do

	$\neg on_1 \wedge (water_1 > reserve_1) \wedge (water_2 < reserve_2)$	$\rightarrow$	$on_1 := true$
$[]$	ENF (S1) • $on_1 \wedge ((water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2))$	$\rightarrow$	$on_1 := false$
else idle			
$\dagger\, TB_{cont}$			

od

Figure 14: Introduce actions to turn $Pump_1$ on and off

guard of (idle) ensures that the antecedent $(water_1 > reserve_1) \wedge (water_2 < reserve_2)$ of (P1) is falsified. To prove (S1), we again perform case analysis on $\overleftarrow{on_1}$. The $\overleftarrow{\neg on_1}$ case is trivial because the behaviour of (idle) ensures $\boxdot \neg on_1$. For case $\overleftarrow{on_1}$, we require $(water_1 > empty_1) \wedge (water_2 < full_2))$, which is guaranteed by the guard $(water_1 > low\_limit_1) \wedge (water_2 < high\_limit_2)$ using Lemma 7.

$$(low\_limit_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{cont}) \wedge (high\_limit_2 \leq full_2 - \mathsf{acc}.water_2.TB_{cont}) \qquad \text{(Rely-8)}$$

As before, introduction of (idle) to the program is justified using (Act-Intro-Rem) and the discussion above. At this stage, we may also remove the original specification statement $on_1 : [true]$, which is justified using (Reduce-ND). After removal of $on_1 : [true]$, we may simplify the guard of (idle) to **else**. Thus, we obtain action system in Fig. 14 where $PP_1 \sqsubseteq AP_1$.

6.4. Discharge remaining enforced conditions

Discharge (S1) in (on1-false). We now derive conditions that enable enforced property (S1) in (on1-false) to be discharged. We define

$$on_1 \wedge ((water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)) \rightarrow on_1 := false \dagger TB_{cont} \qquad \text{(on1-false-b)}$$

to be the action (on1-false) but without the enforced property (S1). Using Lemma 8, we show that the action below is refined by (on1-false-b), which allows us to remove the enforced property (S1).

$$\text{RELY} \ominus (\overrightarrow{I_1} \vee [\![\, action.AP_1 \,]\!]_{on_1}) \bullet \text{(on1-false)}$$

Case $\overrightarrow{I_1}$ reduces to *false* because by (Init-1), $I_1 \Rightarrow \neg on_1$. Hence, we consider $\ominus [\![\, action.AP_1 \,]\!]_{on_1}$. By the definition of non-deterministic choice (16), we may consider each of the actions individually. Case $\ominus [\![\, \text{(on1-false)} \,]\!]_{on_1}$ is trivial because $[\![\, \text{(on1-false)} \,]\!]_{on_1} \Rrightarrow \oplus \overleftarrow{\neg on_1}$, i.e., there can never be two consecutive iterations of $on_1 := false$. Case $\ominus [\![\, \text{(on1-true)} \,]\!]_{on_1}$ may be satisfied by assuming the rely condition below, which we note subsumes (Rely-6).

$$(reserve_1 \geq empty_1 + 2\mathsf{acc}.water_1.TB_{cont}) \wedge (reserve_2 \leq full_2 - 2\mathsf{acc}.water_2.TB_{cont}) \qquad \text{(Rely-6a)}$$

Hence we have the following calculation, representing the case where $Pump_1$ is switched on and then immediately switched off.

ENF $\ominus [\![\, \text{(on1-true)} \,]\!]_{on_1}$ • (on1-false-b)
$\sqsupseteq_{\{on_1\}}$ (Rely-6a), guard of (on1-true), $DIFF.water_1$ and $DIFF.water_2$, and Lemma 7 (Mono-5) to strengthen enforced property

$$\text{ENF} \ominus \left(\begin{array}{l} (\overrightarrow{water_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{cont}}) \wedge \\ (\overrightarrow{water_2 \geq full_2 - \mathsf{acc}.water_2.TB_{cont}}) \end{array} \right) \bullet \text{(on1-false-b)}$$

$\sqsupseteq_{\{on_1\}}$ $water_1$ and $water_2$ are continuous variables, logic (Mono-5)

$$\text{ENF} \left(\begin{array}{l} (\overleftarrow{water_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{cont}}) \wedge \\ (\overleftarrow{water_2 \geq full_2 - \mathsf{acc}.water_2.TB_{cont}}) \end{array} \right) \bullet \text{(on1-false-b)}$$

$\sqsupseteq_{\{on_1\}}$ (Rely-6a), guard of (on1-true), $DIFF.water_1$ and $DIFF.water_2$, and Lemma 7, (Mono-5)

$$\text{ENF} \boxdot ((water_1 \geq empty_1) \wedge (water_2 \geq full_2)) \bullet \text{(on1-false-b)}$$

$\sqsupseteq_{\{on_1\}}$ logic, (Mono-5)

(on1-false)

In the proof above, for each iteration of the action system one can use the fact that there is an immediately preceding interval that satisfies either the initial condition or the body of the action system.

Finally, we consider case $\ominus \llbracket$ (idle) $\rrbracket_{on_1}$, which represents the case where the program is executing **idle** immediately prior to setting on_1 to *false*. To prove this case assume the following property, which supercedes (Rely-8).

$$(low\_limit_1 \geq empty_1 + 2\mathsf{acc}.water_1.TB_{cont}) \wedge (high\_limit_2 \leq full_2 - 2\mathsf{acc}.water_2.TB_{cont}) \qquad \text{(Rely-8a)}$$

Then we have the following calculation.

$$\text{ENF} \ominus \llbracket \text{ (idle) } \rrbracket_{on_1} \bullet \text{(on1-false-b)}$$

$\sqsupseteq_{\{on_1\}}$ expand definition of (idle), $\neg on_1$ case is trivially true, (Mono-5)

$$\text{ENF} \ominus \llbracket (on_1 \Rightarrow (water_1 > low\_limit_1) \wedge (water_2 < high\_limit_2)) \rightarrow \textbf{idle} \rrbracket_{on_1} \bullet \text{(on1-false-b)}$$

$\sqsupseteq_{\{on_1\}}$ similar calculation to $\ominus \llbracket$ (on1-true) $\rrbracket_{on_1}$ case but using (Rely-8a)

(on1-false)

Discharge (S2). Safety condition (S2) is slightly different from the other properties we have considered because it involves multiple iterations of the action system. In particular, because the precision of the pump is lower than the precision of the time band of the controller, several iterations of the action system controller may be executed while the $Pump_1$ is in mode *stopping*. To satisfy (S2), while the pump is stopping, we must ensure that $water_1$ is above $empty_1$ and $water_2$ is below $full_2$. Because $\boxdot c$ joins, $(\text{S2})^{\infty} \Rrightarrow$ (S2), and hence, using Lemma 9, (Dist-7) and (Dist-8) we must prove each of the following. Below, we have applied Lemma 10 to reduce the positive iteration of (on1-true) and (on1-false-b) to a single iteration.

$$\left\llbracket \text{ENF} \ominus \overrightarrow{I_1} \vee \ominus \llbracket \text{ (on1-false-b) } [] \text{ (idle) } \rrbracket_{on_1} \bullet \text{(on1-true)} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (31)$$

$$\left\llbracket \text{ENF} \ominus \overrightarrow{I_1} \vee \ominus \llbracket \text{ (on1-true) } [] \text{ (idle) } \rrbracket_{on_1} \bullet \text{(on1-false-b)} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (32)$$

$$\left\llbracket \text{ENF} \ominus \overrightarrow{I_1} \vee \ominus \llbracket \text{ (on1-true) } [] \text{ (on1-false-b) } \rrbracket_{on_1} \bullet \text{(idle)}^{\omega+} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (33)$$

Cases (31) and (32) are trivial because their behaviour imply $\diamondsuit on_1$, which in turn implies $\diamondsuit \neg Stopping_1$, and hence the antecedent of (S2) is falsified. For case (33), we perform case analysis on $\overleftarrow{on_1}$. Case $\overleftarrow{on_1}$ is trivial. For case $\overleftarrow{\neg on_1}$, we define, which focusses on the $\neg on_1$ case,

$$(\neg on_1 \Rightarrow (water_1 \leq reserve_1) \vee (water_2 \geq reserve_2)) \rightarrow \textbf{idle} \dagger TB_{cont} \qquad \text{(idle-not-on1)}$$

and obtain the following proof obligations:

$$\left\llbracket \text{ENF} \ominus \overrightarrow{I_1} \wedge \overleftarrow{\neg on_1} \bullet \text{(idle-not-on1)}^{\omega+} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (34)$$

$$\left\llbracket \text{ENF} \ominus \llbracket \text{ (on1-true) } \rrbracket_{on_1} \wedge \overleftarrow{\neg on_1} \bullet \text{(idle-not-on1)}^{\omega+} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (35)$$

$$\left\llbracket \text{ENF} \ominus \llbracket \text{ (on1-false-b) } \rrbracket_{on_1} \wedge \overleftarrow{\neg on_1} \bullet \text{(idle-not-on1)}^{\omega+} \right\rrbracket_{on_1} \Rrightarrow \text{(S2)} \qquad (36)$$

$FP_1 \mathrel{\widehat{=}}$
RELY (Rely-1) $\wedge$ (Rely-2) $\wedge$ (Rely-3) $\wedge$ (Rely-4) $\wedge$ (Rely-5) $\wedge$ (Rely-7) $\wedge$
(Rely-8b) $\wedge$ (Rely-8c) $\wedge$ $DIFF.water_1 \wedge DIFF.water_2$ •
INIT $\neg on_1 \wedge Stopped_1$ •
do
$\neg on_1 \wedge (water_1 > reserve_1) \wedge (water_2 < reserve_2) \quad \rightarrow \quad on_1 := true$
$[]$ $on_1 \wedge ((water_1 \leq low\_limit_1) \vee (water_2 \geq high\_limit_2)) \quad \rightarrow \quad on_1 := false$
else idle
$\dagger\, TB_{cont}$
od

Figure 15: Final $Pump_1$ controller

Case (34) is trivial using (Rely-4) because $I_1 \Rightarrow Stopped_1$ and (36) is trivial because $\ominus [\![\text{(on1-true)}]\!]_{on_1} \wedge \overleftarrow{\neg on_1}$ reduces to *false*. To prove (36), when the controller switches from executing (on1-false-b) to (idle-not-on1) $water_1$ is at least $empty_1 + \mathsf{acc}.water_1.TB_{pump}$ and $water_2$ are at most $full_2 - \mathsf{acc}.water_2.TB_{pump}$. Note that the accuracy is with respect to the pump time band as opposed to the previous properties that were with respect to the time band of the controller.

$$[\![\text{(on1-false-b)}]\!]_{on_1} \Rrightarrow \overrightarrow{water_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{pump}} \wedge \overrightarrow{water_2 \geq full_2 - \mathsf{acc}.water_2.TB_{pump}} \quad (37)$$

Using (37), the proof of (36) is straightforward. Informally speaking, $water_1$ is guaranteed to be above $empty_1$ and $water_2$ is guaranteed to be below $full_1$ for $\ell \leq \rho.TB_{pump}$ while $\boxdot Stopping_1$ holds. By (Rely-4) $Stopped_1$ holds after $\rho.TB_{pump}$ units of time have elapsed. A formal proof of a similar property is given in [21, 19]. Because (on1-false-b) only tests to see if $water_1$ is below low_limit_1 and $water_2$ is above $high\_limit_1$, we use the behaviour of the controller before execution of (on1-false-b) to prove (37). In particular, we have

$$\text{ENF}(\ominus \overrightarrow{I_1} \vee \ominus [\![action.AP_1]\!]_{on_1}) \bullet \text{(on1-false-b)} \quad \Rrightarrow \quad \overrightarrow{water_1 \geq empty_1 + \mathsf{acc}.water_1.TB_{pump}} \wedge \overrightarrow{water_2 \geq full_2 - \mathsf{acc}.water_2.TB_{pump}} \quad (38)$$

This is similar to the proof that (S1) holds and may be discharged using the following assumption.

$$low\_limit_1 \geq empty_1 + 2\mathsf{acc}.water_1.TB_{cont} + \mathsf{acc}.water_1.TB_{pump} \quad \text{(Rely-8b)}$$
$$high\_limit_2 \leq full_2 - 2\mathsf{acc}.water_2.TB_{cont} - \mathsf{acc}.water_1.TB_{pump} \quad \text{(Rely-8c)}$$

Note that by using (Rely-7), both of the following hold:

$$reserve_1 \geq empty_1 + 2\mathsf{acc}.water_1.TB_{cont} + \mathsf{acc}.water_1.TB_{pump}$$
$$reserve_2 \leq full_2 - 2\mathsf{acc}.water_2.TB_{cont} - \mathsf{acc}.water_1.TB_{pump}$$

Non-Zeno condition (on1-NZ). In any real implementation, there will be a constant $\epsilon \in \mathbb{R}_{>0}$, that represents the physical lower limit on the time taken to perform each iteration of the action system. By assuming that on_1 is only modified by the action system in Fig. 15, the non-Zeno condition on on_1 (on1-NZ) may be trivially discharged.

The final controller is given in Fig. 15, where the rely condition and initialisation have been made explicit. It is straightforward to show that $AP_1 \sqsubseteq FP_1$, and hence because $\sqsubseteq$ is transitive, we have that $CP_1 \sqsubseteq FP_1$, i.e., FP_1 is an executable refinement of the original specification CP_1.

7. Related work

The theory of action systems is well developed and has been applied to a number of different problem domains. Due to the simplicity of the model, action systems have been used as a basis for several theories

of program refinement [3, 4, 7, 41, 46, 50]. Part of the simplicity of action systems is due to the inherent interleaving semantics assumption, which ensures that statements are executed one after another (even under parallel composition), and hence, a concurrent system is viewed as a sequential program with a non-deterministic choice over all parallel actions.

To enable reasoning about real-time properties, Fidge and Wellings describe a method of extending action systems with actions that consume time [27]. Unlike our model where time is implicitly advanced, Fidge and Wellings use a 'tick' action to model the passage of time, and accessibility restrictions are used to allow multiple processes to modify the same variable. Rönkkö *et al* extend action systems with actions that describe continuous changes to the state using differential equations [45, 46]. A prioritised alternating model of execution is used to ensure that the (discrete) controller actions are able to execute. Hybrid action systems have been extended to *qualitative action systems* [1], but that work is focused on methods for testing real-time systems as opposed to their formal verification/derivation. A weakest precondition for differential actions is provided, however, due to the generality of the definition, the first derivative of the *evolution function* is required to be continuous. Back *et al* present refinement relations for continuous action systems [2], which have been extended by Meinicke and Hayes [41]. Westerlund and Plosila present timed action systems and their refinement using weakest preconditions [50]. Their framework only allows before and after states to be considered and hence cannot be used to reason about durative behaviour.

Ultimately, the interleaving execution model is problematic in contexts such as real-time and multi-core systems where the environment evolves with the controller in a truly concurrent manner. In such contexts, one must address issues with sampling multiple variables over a time interval [10, 20, 30] and be able to reason about transient properties [17, 20]. Furthermore, as software controllers are increasingly used in complex cyber-physical systems, it becomes important to be able to reason over multiple time granularities [9, 10, 18, 30].

There are several frameworks for real-time refinement [32, 37], however, these frameworks do not include a sampling logic. Broy [8] presents interaction refinement in a real-time context and considers refinement between different abstractions of time. Sampling is considered to be an abstraction (via discretisation) of continuous behaviour, however, the framework does not include methods for reasoning about sampling.

An alternative algebraic formulation of LTL [40] and ITL [44] is given by von Karger [48]. However, unlike von Karger, who aims to understand the algebraic characteristics of temporal operators, we aim to present definitions that are suitable for program derivation. Other algebraic properties for interval-based reasoning have been considered in [22, 35, 36]. Methods for reasoning about looping constructs in a discrete setting are given by Back and von Wright [6]. Our aims in this paper have not been to develop an algebra for derivation, but to exploit algebraic reasoning during a derivation.

Our previous paper [20] presents a method for reasoning about hardware/software interaction, which captures properties of (software) signals and their corresponding (physical) effects with respect to the delays and accuracy of the interacting components. For this paper, the generality of our algebraic theory has meant that the interaction between hardware and software has not required any special treatment.

8. Conclusions and future work

This paper continues our research into methods for program derivation using the verify-while-develop paradigm. The method of enforced properties [15, 16] has been extended to enable development of action systems in a compositional manner [17]. The logic in [17] considers traces that consist of pre/post state relations, develops a temporal logic on relations and assumes that environment transitions are interleaved with those of an action system. Although the framework facilitates compositional derivation of action systems code, the underlying interleaving semantics assumption could not properly address sampling anomalies and transient properties. Hence, the framework was generalised so that traces consisted of adjoining intervals together with a sampling logic (Section 3.4), which allowed sampling-related issues to be properly addressed [20]. However, the logic in [20] does not adequately handle specifications over multiple time granularities. Instead, hardware is assumed to react and take effect instantaneously, which is unrealistic.

This paper incorporates a time bands theory [10] into action systems and we develop an interval-based semantics for reasoning about sampling over continuous environments. We develop an algebraic framework

that supercedes Interval Linear Temporal Logic [20]. We have developed high-level methods that use time bands to simplify reasoning about hardware/software interaction. As an example, we have derived an action system controller for a real-time pump. Notable in our derivation is the development of side conditions that formalise the assumptions on the environment and the derivation of relationships between threshold and critical levels based on the (different) time bands of the controller and pump.

As part of future work, we aim to further develop the theories for parallel composition of action systems by developing (compositional) rely/guarantee-style methods. We also aim to explore the links between action systems and teleo-reactive programs [18, 21]. In particular it will be interesting to consider a development method that starts with a teleo-reactive program (whose semantics are closer to abstract specifications) and refining the teleo-reactive program to an action system. We also consider development of mechanisation of this derivation method to be future work. With the simplifications that we have achieved in our framework, we are now confident that tools to support program derivation can be developed.

A possible generalisation of the two-pump example would be to consider different time bands for each pump. However, for the purposes of this paper, such a generalisation does not provide any new insights into the methodology because the pumps are not interacting directly with each other. That is, the methods we have developed for relating the time bands of the controller, water level and pump (which do interact) can be applied to more sophisticated systems. We aim to consider such examples after developing mechanisation.

References

[1] Aichernig, B. K., Brandl, H., Krenn, W., 2009. Qualitative action systems. In: Breitman, K., Cavalcanti, A. (Eds.), ICFEM. Vol. 5885 of LNCS. Springer, pp. 206–225.

[2] Back, R.-J., Petre, L., Porres, I., 2000. Generalizing action systems to hybrid systems. In: Joseph, M. (Ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems. Vol. 1926 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 73–91.

[3] Back, R.-J., Sere, K., 1991. Stepwise refinement of action systems. Structured Programming 12 (1), 17–30.

[4] Back, R.-J., von Wright, J., 1994. Trace refinement of action systems. In: CONCUR '94: Proceedings of the Concurrency Theory. Springer-Verlag, pp. 367–384.

[5] Back, R.-J., von Wright, J., 1998. Refinement Calculus: A Systematic Introduction. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

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

[7] Back, R.-J., von Wright, J., 2003. Compositional action system refinement. Formal Asp. Comput. 15 (2-3), 103–117.

[8] Broy, M., 2001. Refinement of time. Theor. Comput. Sci. 253 (1), 3–26.

[9] Burns, A., Baxter, G., 2006. Time bands in systems structure. In: Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective. Springer-Verlag, pp. 74–88.

[10] Burns, A., Hayes, I. J., 2010. A timeband framework for modelling real-time systems. Real-Time Systems 45 (1), 106–142.

[11] Chandy, K. M., Misra, J., 1988. Parallel Program Design: A Foundation. Addison-Wesley Longman Publishing Co., Inc.

[12] Coleman, J. W., Jones, C. B., 2007. A structural proof of the soundness of rely/guarantee rules. J. Log. Comput. 17 (4), 807–841.

[13] de Roever, W. P., Engelhardt, K., 1996. Data Refinement: Model-oriented proof methods and their comparison. No. 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[14] Dijkstra, E. W., 1975. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18 (8), 453–457.

[15] Dongol, B., 2009. Progress-based verification and derivation of concurrent programs. Ph.D. thesis, The University of Queensland.

[16] Dongol, B., Hayes, I. J., 2009. Enforcing safety and progress properties: An approach to concurrent program derivation. In: 20th Australian Software Engineering Conference. IEEE Computer Society, pp. 3–12.

[17] Dongol, B., Hayes, I. J., 2010. Compositional action system derivation using enforced properties. In: Bolduc, C., Desharnais, J., Ktari, B. (Eds.), Mathematics of Program Construction. Vol. 6120 of Lecture Notes in Computer Science. Springer, pp. 119–139.

[18] Dongol, B., Hayes, I. J., 2011. Approximating idealised real-time specifications using time bands. In: AVoCS 2011. Vol. 46 of ECEASST. EASST, pp. 1–16.

[19] Dongol, B., Hayes, I. J., 2012. Deriving real-time action systems controllers from multiscale system specifications. In: Gibbons, J., Nogueira, P. (Eds.), MPC. Vol. 7342 of Lecture Notes in Computer Science. Springer, pp. 102–131.

[20] Dongol, B., Hayes, I. J., 2012. Deriving real-time action systems in a sampling logic. Science of Computer Programming (To appear).
URL `http://www.sciencedirect.com/science/article/pii/S0167642312001360`

[21] Dongol, B., Hayes, I. J., 2012. Rely/guarantee reasoning for teleo-reactive programs over multiple time bands. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (Eds.), IFM. Vol. 7321 of Lecture Notes in Computer Science. Springer, pp. 39–53.

[22] Dongol, B., Hayes, I. J., Meinicke, L., Solin, K., 2012. Towards an algebra for real-time programs. In: Kahl, W., Griffin, T. G. (Eds.), RAMICS. Vol. 7560 of Lecture Notes in Computer Science. Springer, pp. 50–65.
[23] Dongol, B., Hayes, I. J., Robinson, P. J., 2013. Reasoning about goal-directed real-time teleo-reactive programs. Formal Aspects of Computing(Accepted 19-11-2012).
URL http://dx.doi.org/10.1007/s00165-012-0272-1
[24] Dongol, B., Mooij, A. J., 2006. Progress in deriving concurrent programs: Emphasizing the role of stable guards. In: Uustalu, T. (Ed.), 8th International Conference on Mathematics of Program Construction. Vol. 4014 of LNCS. Springer, pp. 140–161.
[25] Dongol, B., Mooij, A. J., March 2008. Streamlining progress-based derivations of concurrent programs. Formal Aspects of Computing 20 (2), 141–160.
[26] Feijen, W. H. J., van Gasteren, A. J. M., 1999. On a Method of Multiprogramming. Springer Verlag.
[27] Fidge, C. J., Wellings, A. J., 1997. An action-based formal model for concurrent real-time systems. Formal Aspects of Computing 9, 175–207.
[28] Guelev, D. P., Hung, D. V., 2002. Prefix and projection onto state in duration calculus. Electr. Notes Theor. Comput. Sci. 65 (6), 101–119.
[29] Gupta, V., Henzinger, T. A., Jagadeesan, R., 1997. Robust timed automata. In: Proceedings of the International Workshop on Hybrid and Real-Time Systems. Springer-Verlag, London, UK, pp. 331–345.
[30] Hayes, I. J., Burns, A., Dongol, B., Jones, C. B., 2013. Comparing degrees of non-determinism in expression evaluation. The Computer Journal.
URL http://comjnl.oxfordjournals.org/content/early/2013/02/05/comjnl.bxt005.abstract
[31] Hayes, I. J., Jackson, M. A., Jones, C. B., 2003. Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (Eds.), FME. Vol. 2805 of Lecture Notes in Computer Science. Springer, pp. 154–169.
[32] Hayes, I. J., Utting, M., 2001. A sequential real-time refinement calculus. Acta Inf. 37 (6), 385–448.
[33] Henzinger, T. A., 1996. The theory of hybrid automata. In: LICS '96. IEEE Computer Society, Washington, DC, USA, pp. 278–292.
[34] Henzinger, T. A., Qadeer, S., Rajamani, S. K., 1999. Assume-guarantee refinement between different time scales. In: CAV '99. Springer-Verlag, London, UK, pp. 208–221.
[35] Höfner, P., Möller, B., 2008. Algebraic neighbourhood logic. J. Log. Algebr. Program. 76 (1), 35–59.
[36] Höfner, P., Möller, B., 2009. An algebra of hybrid systems. J. Log. Algebr. Program. 78 (2), 74–97.
[37] Hooman, J., van Roosmalen, O. S., 2000. An approach to platform independent real-time programming: (1) formal description. Real-Time Systems 19 (1), 61–85.
[38] Jones, C. B., 1983. Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5 (4), 596–619.
[39] Jones, C. B., Hayes, I. J., Jackson, M. A., 2007. Deriving specifications for systems that are connected to the physical world. In: Jones, C. B., Liu, Z., Woodcock, J. (Eds.), Formal Methods and Hybrid Real-Time Systems. Vol. 4700 of Lecture Notes in Computer Science. Springer, pp. 364–390.
[40] Manna, Z., Pnueli, A., 1992. Temporal Verification of Reactive and Concurrent Systems: Specification. Springer-Verlag New York, Inc.
[41] Meinicke, L., Hayes, I. J., 2006. Continuous action system refinement. In: Uustalu, T. (Ed.), Mathematics of Program Construction. Vol. 4014 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 316–337.
[42] Moszkowski, B. C., 1995. Compositional reasoning about projected and infinite time. In: ICECCS. IEEE Computer Society, pp. 238–245.
[43] Moszkowski, B. C., 1997. Compositional reasoning using interval temporal logic and Tempura. In: de Roever, W. P., Langmaack, H., Pnueli, A. (Eds.), COMPOS. Vol. 1536 of LNCS. Springer, pp. 439–464.
[44] Moszkowski, B. C., 2000. A complete axiomatization of interval temporal logic with infinite time. In: LICS. pp. 241–252.
[45] Rönkkö, M., Ravn, A. P., 1999. Action systems with continuous behaviour. In: Hybrid Systems V. Springer-Verlag, London, UK, pp. 304–323.
[46] Rönkkö, M., Ravn, A. P., Sere, K., 2003. Hybrid action systems. Theoretical Computer Science 290 (1), 937 – 973.
[47] von Karger, B., 2000. A calculational approach to reactive systems. Sci. Comput. Program. 37 (1-3), 139–161.
[48] von Karger, B., 2000. Temporal algebra. In: Backhouse, R. C., Crole, R. L., Gibbons, J. (Eds.), Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. Vol. 2297 of Lecture Notes in Computer Science. Springer, pp. 309–385.
[49] Wei, K., Woodcock, J., Burns, A., June 2010. Formalising the timebands model in timed circus. Tech. rep., University of York.
[50] Westerlund, T., Plosila, J., 2007. Time aware system refinement. Electr. Notes Theor. Comput. Sci. 187, 91–106.
[51] Wulf, M., Doyen, L., Markey, N., Raskin, J.-F., December 2008. Robust safety of timed automata. Form. Methods Syst. Des. 33, 45–84.
[52] Zhou, C., Hansen, M. R., 2004. Duration Calculus: A Formal Approach to Real-Time Systems. EATCS: Monographs in Theoretical Computer Science. Springer.