---

**Article:**
Dongol, B. and Derrick, J. (2015) Interval-based data refinement: A uniform approach to true concurrency in discrete and real-time systems. Science of Computer Programming, 111. pp. 214-247. ISSN 0167-6423

https://doi.org/10.1016/j.scico.2015.05.005

---

# Interval-Based Data Refinement: A Uniform Approach to True Concurrency in Discrete and Real-Time Systems

Brijesh Dongol

*Department of Computer Science, Brunel University London, UK*

John Derrick

*Department of Computer Science, The University of Sheffield, UK*

**Abstract**

The majority of modern systems exhibit sophisticated concurrent behaviour, where several system components observe and modify the state with fine-grained atomicity. Many systems also exhibit truly concurrent behaviour, where multiple events may occur simultaneously. Data refinement, a correctness criterion to compare an abstract and a concrete implementation, normally admits interleaved models of execution only. In this paper, we present a method of data refinement using a framework that allows one to view a component's evolution over an interval of time, simplifying reasoning about true concurrency. By modifying the type of an interval, our theory may be specialised to cover data refinement of both discrete and real-time systems. We develop a sound interval-based forward simulation rule that enables decomposition of data refinement proofs, and apply this rule to verify data refinement for two examples: a simple concurrent program and a more in-depth real-time controller.

*Keywords:* Refinement, interval-based reasoning, true concurrency, discrete time systems, real-time systems

## 1. Introduction

Data refinement allows one to develop systems in a stepwise manner, enabling an abstract system to be incrementally replaced by a concrete implementation by guaranteeing that every observable behaviour of the concrete system is a possible observable behaviour of the abstract system. A benefit of such developments is the ability to reason at a level of abstraction suitable for the current stage of development, and the ability to introduce additional detail to a system via correctness-preserving transformations. A *representation relation* between concrete and abstract states is often used to link the internal states of the concrete and abstract systems. This enables the state representation at different levels of abstraction to differ. For example, a queue data type may be represented by a sequence in an abstract system and by a linked list in the corresponding concrete implementation, and hence, operations at the abstract level access and modify the sequence, whereas at the concrete level operations access and modify the linked list.

Over the years, numerous techniques for verifying data refinement have been developed for a number of application domains [12], including methods for refinement of concurrent [14] and real-time [25] systems. These methods use frameworks that formalise the behaviour of system components in the traditional manner, i.e., as relations between a pre and post state. Therefore, the refinement relations that are used to verify data refinement are also relations between an abstract and a concrete state.

In the context of true concurrency, pre/post-state relational models lack the expressive power to reason about simultaneous accesses and modifications to a system's state [48] as they inherently admit an interleaved execution semantics. Thus, one must perform an additional step of reasoning to prove that the true concurrency semantics is indeed captured by the interleaved semantics. In some instances, e.g., real-time systems, the pre/post-state relational model cannot be used to formalise *transient properties* [21, 19], which

*AInit*: $\neg grd$

| Process $ap$ | Process $aq$ |
|---|---|
| $ap_1$: **if** $grd$ **then** | $aq_1$: **if** $b$ **then** |
| $ap_2$: $m := 1$ | $aq_2$: $grd := true$ |
| $ap_3$: **else** $m := 2$ **fi** | $aq_3$: **else skip fi** |

Figure 1: Abstract program with guard $grd$

*CInit*: $v \leq u$

| Process $cp$ | Process $cq$ |
|---|---|
| $cp_1$: **if** $u < v$ **then** | $cq_1$: **if** $0 < w$ **then** |
| $cp_2$: $m := 1$ | $cq_2$: $v := u + 1$ |
| $cp_3$: **else** $m := 2$ **fi** | $cq_3$: **else** $v := u - 1$ **fi** |

Figure 2: Concrete program with guard $u < v$ in $cp_1$

are properties that only hold for a small instant of time, making them physically impossible to detect. Further difficulties arise for relational models when admitting real-world delays, where tolerances required of an implementation are difficult to record abstractly.

We aim to enable reasoning about the *evolution* of a system over its interval of execution [2, 44], which may comprise several system states. To this end, we use a framework of *interval predicates* [17, 21], which is inspired by both Interval Temporal Logic [39] and Duration Calculus [50]. Notable in our logic is that it incorporates reasoning about *apparent states evaluation* [30, 21], which allows one to take into account the low-level nondeterminism of expression evaluation at a higher level of abstraction. This makes it possible to model both fine-grained interleaving (in the case of concurrent programs) and sampling errors (in the case of real-time systems). Interval predicates have been used to reason about both discrete-time programs [23, 17] and real-time systems [21], however, there does not exist any native support for interval-based data refinement. The methods in [23, 17, 21] only cope with refinements where the concrete state space is a subset of the abstract.

The main contribution of this paper is an interval-based approach for verifying data refinement, which provides a logic for reasoning about refinement in the presence of true concurrency. Our framework is general in the sense that it presents uniform techniques to reason about both discrete-time and real-time systems — the type of reasoning to be performed can be specialised via different instantiations for the type of an interval. We develop a forward simulation rule for verifying data refinement, and present methods for decomposing proof obligations over common programming constructs. These are applied to verify data refinement of a simple concurrent program and a more complex real-time multi-pump system. For the real-time example, we incorporate the theory of time bands [9, 10], which simplifies reasoning about systems over multiple time granularities. Ours is the first method (to the best of our knowledge) to incorporate data refinement and time bands in system development.

This paper extends [15] by including additional explanations, and the real-time example is new to this paper. At a technical level, the definition of refinement has been improved from [15] to better integrate interval-based reasoning; the theory in [15] contained a mix of state and interval-based reasoning, which complicated parts of the logic. These issues have now been streamlined, allowing our theory and associated proofs to become more concise. The underlying notion of refinement is however unaltered from the notions in [15, 12]; namely, a concrete system refines an abstract system if, and only if, every observable behaviour of the concrete is a possible observable behaviour of the abstract.

Motivation and background material for the paper is presented in Section 2, clarifying our notions of state-based data refinement. Our interval-based refinement theory is presented in Section 3, and a methods for decomposing refinement proofs via simulation are presented in Section 4. Section 5 presents different methods for evaluation state predicates over intervals and provides background for our two examples. Methods for reasoning about fine-grained concurrency and a proof of our running example is presented in Section 6. A more complex refinement of a real-time multipump system is given in Section 7.

## 2. State-based data refinement

In this section, we present motivation for our interval-based model by reviewing data refinement for concurrent programs modelled in a framework of pre/post state relations [11, 12]. In particular, we describe

some of the commonly occurring difficulties when verifying refinement using forward simulation.

As a running example we consider the abstract program in Figure 1, written in the style of Feijen and van Gasteren [24], which consists of variables $grd, b \in \mathbb{B}$, $m \in \mathbb{N}$, initialisation *AInit* and processes *ap* and *aq*. Process *ap* is a sequential program with labels $ap_1$, $ap_2$, and $ap_3$ that tests whether *grd* holds (atomically), then executes $m := 1$ if *grd* evaluates to *true* and $m := 2$ otherwise. Process *aq* is similar. The program executes by initialising as specified by *AInit*, and then executing *ap* and *aq* concurrently by interleaving their atomic statements.

A *state* over $V \subseteq Var$ is of type $\Sigma_V \mathrel{\widehat{=}} V \to Val$, where *Var* is the type of a variable and *Val* is the generic type of a value, i.e., are mappings from variables to values. *Program counters* for each process are assumed to be implicitly included in each state to formalise the control flow of a program, e.g., the program in Figure 1 uses two program counters $pc_{ap}$ and $pc_{aq}$, where $pc_{ap} = ap_1$ is assumed to hold whenever control of process *ap* is at $ap_1$, i.e., if $pc_{ap} = ap_1$, then the next statement that *ap* will execute is the statement labelled $ap_1$. After execution of $ap_1$, the value of $pc_{ap}$ is updated so that either $pc_{ap} = ap_2$ or $pc_{ap} = ap_3$ holds, depending on the outcome of the evaluation of *grd*.

A program's initialisation is modelled by a relation, and each label corresponds to an atomic statement, whose behaviour is also modelled by a relation. Thus, a program generates a set of *traces*, each of which is a sequence of states. We assume sequences start with index 0 and are potentially infinite. One may characterise traces using an *execution*, which is a sequence of labels starting with initialisation. For example, a possible execution of the program in Figure 1 is

$$\langle AInit, ap_1, aq_1, aq_2, ap_3 \rangle \tag{1}$$

Using '.' for function application, we say an execution *ex corresponds* to a trace *tr* iff for each $i \in \mathrm{dom}.ex$, $(tr.i, tr.(i+1)) \in ex.i$ and either $\mathrm{dom}.tr = \mathrm{dom}.ex = \mathbb{N}$ or $size.(\mathrm{dom}.ex) = size.(\mathrm{dom}.tr) + 1$; we use labels and relations corresponding to the statement at the label interchangeably. An execution *ex* is *valid* iff $\mathrm{dom}.ex \neq \varnothing$, $ex.0$ is an initialisation, and *ex* corresponds to at least one trace, e.g., (1) above is valid. Not every execution is valid, e.g., $\langle AInit, ap_1, ap_2 \rangle$ is invalid because execution of $ap_1$ after *AInit* causes *grd* to evaluate to *false* and $pc_{ap}$ to be updated to $ap_3$, and hence, statement $ap_2$ cannot be executed.

Now consider the concrete program in Figure 2 that replaces *grd* by $u < v$ and *b* by $0 < w$, where *u*, *v* and *w* are fresh with respect to the program in Figure 1. Initially, $v \leq u$ holds. Furthermore, *cq* (modelling the concrete environment of *cp*) sets *v* to $u+1$ if *w* is positive and to $u-1$ otherwise. One may be interested in knowing whether the program in Figure 2 *data refines* the program in Figure 1, which defines conditions for the program in Figure 1 to be substituted by the program in Figure 2 [12]. Data refinement allows this replacement if every execution of the program in Figure 2 has a corresponding execution of the program in Figure 1, e.g., concrete execution $\langle CInit, cp_1, cq_1, cq_2, cp_3 \rangle$ has a corresponding abstract execution (1).

In general, representation of data within a concrete program differs from the representation in the abstract, and hence, one must distinguish between the disjoint sets of *observable* and *representation* variables, which respectively denote variables that can and cannot be observed. To verify data refinement, the abstract and concrete programs are associated with *finalisations*, which are relations between a representation and an observable state, defining the portion of the representation state that becomes observable. Note that the finalisations for our example programs in Figures 1 and 2 are not shown. This is because a verifier has freedom to choose a finalisation for the program at hand; different choices for the finalisation allow different parts of the program to become observable and affect the type of refinement that is captured by data refinement [14]. For the programs in Figures 1 and 2, we assume finalisations make the variable *m* observable. Hence, Figure 1 is data refined by Figure 2 if *ap* is able to execute $ap_2$ ($ap_3$) whenever *cp* is able to execute $cp_2$ ($cp_3$, respectively). We define a *finalised execution* of a program to be a valid execution concatenated with the finalisation of the program, e.g., $\langle AInit, ap_1, aq_1, aq_2, ap_3, AFin \rangle$ is a finalised execution of the program in Figure 1 generated from the valid execution (1). Valid executions are not necessarily complete, and hence, one may observe the state in the "middle" of a program's execution; an extreme example is $\langle AInit, AFin \rangle$, where the execution is finalised immediately after initialisation.

For the rest of this section, suppose we model initialisation as a relation from an observable state to a representation state, each label corresponds to a statement that is modelled by a relation between two

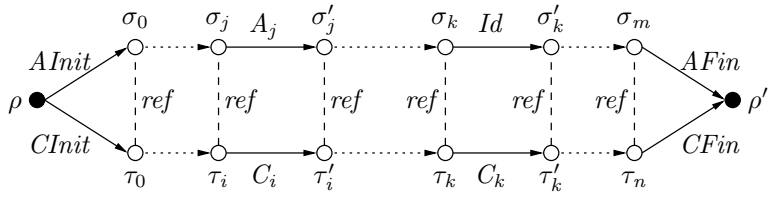



Figure 3: Data refinement via simulation

representation states, and a finalisation is a relation from a representation state to an observable state. Assuming '$\fatsemi$' denotes relational composition and $id$ is the identity relation, we define the *composition* of a sequence of relations $R$ as

$$comp.R \mathrel{\widehat{=}} \textbf{if } R = \langle\,\rangle \textbf{ then } id \textbf{ else } head.R \fatsemi comp.(tail.R)$$

which composes the relations of $R$ in order. We also define a function $rel$, which replaces each label in an execution by the relation corresponding to the statement of that label.

Some steps of the concrete program may correspond to *stuttering steps* of the abstract [4], and hence, there may not be a one-to-one correspondence between concrete and abstract executions. Stuttering is reflected in an abstract execution by allowing a finite number of labels '$Id$' to be interleaved with each finalised execution of the abstract program, where $Id$ is assumed to be different from all other labels, and the relation corresponding to label $Id$ is the identity relation $id$. Data refinement is defined with respect to a *correspondence function* that maps concrete labels to abstract labels.[1] A correspondence function is valid iff it maps concrete initialisations to abstract initialisations, concrete finalisations to abstract finalisations, each label of a non-stuttering concrete statement to a corresponding abstract statement, and each label of stuttering concrete statement to $Id$. A program $C$ is a *data refinement* of a program $A$ with respect to correspondence function $F$ iff $F$ is valid and for every finalised execution $exc$ of $C$,

- $exa \mathrel{\widehat{=}} \lambda\, i{:}\,\mathrm{dom}.exc \bullet F.(exc.i)$ is a finalised execution of $A$ (with possibly finite stuttering) and
- $comp.(rel.exc) \subseteq comp.(rel.exa)$ holds.

Proving data refinement directly from this formal definition is difficult as it requires induction over the traces. Instead, one often proves data refinement by verifying *simulation* between an abstract and concrete system, which requires the use of a *refinement relation* to link the internal representations of the abstract and concrete programs. We assume that a relation $r \in X \leftrightarrow Y$ is characterised by a function $fr \in X \to Y \to \mathbb{B}$ where $(x, y) \in r$ iff $fr.x.y.$ holds. Thus, a *state relation* over $Y, Z \subseteq Var$ is defined by its characteristic function $StateRel_{Y,Z} \mathrel{\widehat{=}} \Sigma_Y \to \Sigma_Z \to \mathbb{B}$. As depicted in Figure 3, a refinement relation $ref$ is a *forward simulation* between a concrete and abstract system with respect to correspondence function $F$ if:

1. whenever the concrete system can be initialised from an observable state $\rho$ to obtain a concrete representation state $\tau_0$, it must be possible to initialise the abstract system from $\rho$ to result in abstract representation state $\sigma_0$ such that $ref.\sigma_0.\tau_0$ holds,
2. for every concrete statement $C_i$, abstract state $\sigma_j$ and concrete state $\tau_i$, if $ref.\sigma_j.\tau_i$ holds and $C_i$ relates $\tau_i$ to $\tau'_i$, then there exists an abstract state $\sigma'_j$ and an $A_j$ such that $A_j = F.C_i$, $A_j$ relates $\sigma_j$ to $\sigma'_j$ and $ref.\sigma'_j.\tau'_i$ holds, and
3. finalising any abstract state $\sigma_m$ (using the abstract system's finalisation) and concrete state $\tau_m$ (using the concrete system's finalisation) results in the same observable state whenever $ref.\sigma_m.\tau_n$ holds.

[1] In general, correspondence functions may additionally refer to the concrete and abstract states [12].

$CInit$: $v \leq u$

| Process $cp$ | Process $cq$ |
|---|---|
| $cp_{1.1}$: $\left(\begin{array}{l} u_p := u \,; \\ v_p := v \end{array}\right)$ | $cq_1$: **if** $0 < w$ **then** |
| $cp_{1.2}$: $\sqcap$ | $cq_{2.1}$: $u_q := u$ ; |
| $cp_{1.3}$: $\left(\begin{array}{l} v_p := v \,; \\ u_p := u \end{array}\right)$; | $cq_{2.2}$: $v := u_q + 1$ |
| $cp_{1.4}$: | **else** |
| $cp_{1.5}$: **if** $u_p < v_p$ **then** | $cq_{3.1}$: $u_q := u$ ; |
| $cp_2$: $m := 1$ | $cq_{3.2}$: $v := u_q - 1$ |
| $cp_3$: **then** $m := 2$ **fi** | **fi** |

Figure 4: Making the atomicity of expression evaluation in Figure 2 explicit

| Concrete label | Abstract label |
|---|---|
| $CInit$ | $AInit$ |
| $cp_{1.1}$, $cp_{1.3}$, $cp_{1.5}$, $cq_{2.1}$, $cp_{2.2}$ | $Id$ |
| $cp_{1.2}$, $cp_{1.4}$ | $ap_1$ |
| $cp_i$ for $i \in \{2, 3\}$ | $ap_i$ |
| $cq_1$ | $aq_1$ |
| $cq_{2.2}$ | $aq_2$ |
| $cq_{3.2}$ | $aq_3$ |
| $CFin$ | $AFin$ |

Figure 5: Correspondence function for data refinement between Figure 4 and Figure 1

For models of computation that assume coarse-grained atomicity, expression evaluation is assumed to be instantaneous, and hence, establishing a data refinement is simpler. For example, under coarse-grained atomicity, one can establish data refinement between the programs in Figures 1 and 2 with respect to a correspondence function $F$, where $F.cp_i = ap_i$ and $F.cq_i = aq_i$ for $i \in \{1, 2, 3\}$. In particular, it is possible to establish a forward simulation using $cg\_ref$ below as the refinement relation, where $\sigma$ and $\tau$ are abstract and concrete states, respectively.

$$\begin{array}{rcl} var\_rel.\sigma.\tau & \widehat{=} & (\sigma.grd = (\tau.u < \tau.v)) \wedge (\sigma.b = (0 < \tau.w)) \wedge (\sigma.m = \tau.m) \\ cg\_ref.\sigma.\tau & \widehat{=} & var\_rel.\sigma.\tau \wedge \forall i{:}\{1,2,3\} \bullet (\sigma.pc_{ap} = ap_i \Rightarrow \tau.pc_{cp} = cp_i) \wedge \\ & & \qquad (\sigma.pc_{aq} = aq_i \Rightarrow \tau.pc_{cq} = cq_i) \end{array}$$

However, in a setting with fine-grained atomicity, verifying data refinement is more difficult because there may be interference from other processes while an expression is being evaluated [30]. Furthermore, the order in which variables are read within an expression is often not fixed, e.g., due to compiler optimisations. To take these assumptions into consideration in a framework of pre/post state relations, one must consider the lower-level program in Figure 4, where the guard evaluation at $cp_1$ in Figure 2 has been split into a number of simpler atomic statements using fresh variables $u_p$ and $v_p$ that are local to process $cp$. Via a nondeterministic choice '$\sqcap$', process $cp$ chooses between executions $cp_{1.1}$ ; $cp_{1.2}$ and $cp_{1.3}$ ; $cp_{1.4}$, which read the (global values) $u$ and $v$ into local variables $u_p$ and $v_p$, respectively, in two atomic steps. Evaluation of guard $u < v$ at $cp_1$ in Figure 2 is then replaced by evaluation of $u_p < v_p$. Similarly, in process $cq$, one must split both $cq_2$ and $cq_3$ because they read from and write to two different shared variables.

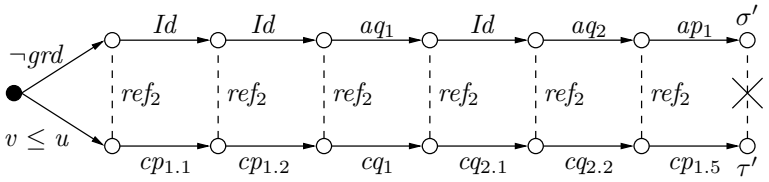
A proof of data refinement between the programs in Figures 1 and 2 using forward simulation with respect to $cg\_ref$ is now more difficult because an atomic evaluation of $grd$ has been split into several atomic statements. A data refinement with respect a naive correspondence function that maps:

- $cp_i$ to $Id$ for $i \in \{1.1, 1.2, 1.3, 1.4\}$, $cp_{1.5}$ to $ap_1$, $cp_2$ to $ap_2$, $cp_3$ to $ap_3$, and
- $cq_{2.1}$ and $cq_{3.1}$ to $Id$, $cq_1$ to $aq_1$, $cq_{2.2}$ to $aq_2$, and $cq_{3.1}$ to $aq_3$

cannot be verified using forward simulation. Here the correspondence function yields the following relation on program counters, where $CPS \mathrel{\widehat{=}} \{cp_{1.1}, cp_{1.2}, cp_{1.3}, cp_{1.4}, cp_{1.5}\}$.

$$cg\_rel_2.\sigma.\tau \;\widehat{=}\; \sigma.pc_{ap} = \begin{cases} ap_1 & \text{if } \tau.pc_{cp} \in CPS \\ ap_2 & \text{if } \tau.pc_{cp} = cp_2 \\ ap_3 & \text{if } \tau.pc_{cp} = cp_3 \end{cases} \quad \wedge \quad \sigma.pc_{aq} = \begin{cases} aq_1 & \text{if } \tau.pc_{cq} = cq_1 \\ aq_2 & \text{if } \tau.pc_{cq} \in \{cq_{2.1}, cq_{2.2}\} \\ aq_3 & \text{if } \tau.pc_{cq} \in \{cq_{3.1}, cq_{3.2}\} \end{cases}$$

The refinement relation then becomes $ref_2 \mathrel{\widehat{=}} var\_rel \wedge cg\_rel_2$. Now, consider the concrete execution below, which is matched to a corresponding abstract execution using $ref_2$.




This execution ultimately results in a step ($cp_{1.5}$) that cannot be matched abstractly in a manner that preserves the refinement relation. In particular, the concrete system transitions to a state $\tau'$ where $\tau'.pc_{cp} = cp_3$ and the abstract to a state $\sigma'$ where $\sigma'.pc_{ap} = ap_2$, and hence $cg\_rel_2.\sigma'.\tau'$ does not hold.

Such issues for forward simulation are well-known [11] — here the problem lies with the naively chosen correspondence function, which incorrectly maps $cp_{1.5}$ to $ap_1$ causing the non-determinism at the abstract level to be resolved later than it should be. The concrete step that fixes the outcome of the test at $cp_{1.5}$ actually occurs either at $cp_{1.2}$ or $cp_{1.4}$ — after executing these statements it will be possible to evaluate the guard $u_p < v_p$ (at $cp_{1.5}$). This outcome however is dependant on the value of the shared variable read, which can change due to interference.

Thus, it turns out that one must use the correspondence relation from Figure 5, which (somewhat unintuitively) matches execution of $cp_{1.4}$ (that assigns $u$ to $u_p$) with execution of $ap_1$ (that tests *grd*). The refinement relation used to prove forward simulation is more complicated than *cg_ref* and can be constructed using the correspondence function in Figure 5, but the details are elided.

Such difficulties in verifying relatively trivial modifications expose the complexities of concurrent program development via stepwise refinement. Further difficulties are encountered in the context of true concurrency as state-based models inherently capture concurrency as interleaving. Thus, one must not only show data refinement, but must perform an additional (non-trivial) reasoning step to show that the interleaved semantics does indeed accurately reflect the truly concurrent one. Real-time systems present yet another challenge; here, the environment continues to evolve as a program (e.g., controller) executes its statements. Further difficulties are encountered when using these models to reason about real-world delays and sampling issues, while transient properties may cause a system to become unimplementable [21]. To address such issues, this paper uses an interval-based logic to model system behaviour and we develop a method of data refinement for the logic. We prove refinement between the programs in Figures 1 and 2 using our interval-based setting in Section 6, and here, the abstract and concrete guard evaluation can be matched up as one would expect. We also present a more comprehensive development of a real-time pump controller in a step-wise manner (Section 7).

## 3. Data refinement over intervals

We aim to verify data refinement between systems whose behaviours are formalised by interval predicates. To this end, we present interval predicates in Section 3.1 and the novel concept of interval relations in Section 3.2. These are then used to formalise data refinement over intervals Section 3.3.

### *3.1. Interval predicates*

The *time domain* is an infinite set[2] $\mathcal{T} \subseteq \mathbb{R}$ such that $\forall t{:}\mathcal{T} \bullet \exists t', t''{:}\mathcal{T} \bullet t' < t < t''$, i.e., $\mathcal{T}$ is infinite in both the positive and negative directions. An *interval* of $\mathcal{T}$ is a contiguous subset of $\mathcal{T}$, and hence, the set of all intervals of $\mathcal{T}$ is given by:

$$Intv_{\mathcal{T}} \mathrel{\widehat{=}} \{\Delta \subseteq \mathcal{T} \mid \forall t, t'{:}\Delta \bullet \forall t''{:}\mathcal{T} \bullet t \leq t'' \leq t' \Rightarrow t'' \in \Delta\}$$

The type $Intv_{\mathcal{T}}$ may be used to model both discrete-time (e.g., by picking $\mathcal{T} = \mathbb{Z}$) and real-time (by picking $\mathcal{T} = \mathbb{R}$) systems. The theory we develop in the rest of this paper is kept general in the sense that it enables reasoning about both instantiations in a uniform manner.

[2] An even more abstract view is that $\mathcal{T}$ is a linearly ordered set, of which $\mathbb{R}$ is an instantiation.

The configuration of a system at each time is given by a *state*, which defines the values of the system's variables (see Section 2). A *state predicate* is of type $\mathit{StatePred}_V \mathrel{\widehat{=}} \Sigma_V \to \mathbb{B}$, where $V \subseteq \mathit{Var}$, is used to define properties of states. The behaviour of the system over all time is given by a *stream*, which is a function of type $\mathit{Stream}_{\mathcal{T},V} \mathrel{\widehat{=}} \mathcal{T} \to \Sigma_V$, mapping each element of $\mathcal{T}$ to a state over $V$. An *interval predicate* is used to define the behaviour of a stream over a particular interval, which has type $\mathit{IntvPred}_{\mathcal{T},V} \mathrel{\widehat{=}} \mathit{Intv}_{\mathcal{T}} \to \mathit{Stream}_{\mathcal{T},V} \to \mathbb{B}$. For the rest of the paper, we assume that the underlying type of the interval under consideration is fixed, and hence, to reduce notational complexity, we omit $\mathcal{T}$ whenever possible.

**Example 1.** Suppose we define

$$\sigma \mathrel{\widehat{=}} \{u \mapsto 0, v \mapsto 1, w \mapsto aaa\} \qquad s \mathrel{\widehat{=}} \lambda\, t \bullet \textbf{if } t \geq 10 \textbf{ then } \{u \mapsto 3t^2, v \mapsto 42, w \mapsto bbb\} \textbf{ else } \sigma$$
$$c_1 \mathrel{\widehat{=}} \lambda\, \sigma \bullet \sigma.u < \sigma.v \qquad c_2 \mathrel{\widehat{=}} \lambda\, \sigma \bullet \sigma.w = qqq$$
$$g \mathrel{\widehat{=}} \lambda\, \Delta, s \bullet \forall t{:}\, \Delta \bullet (s.t).u \geq 300$$

Then, $\sigma$ is a state; $c_1$ and $c_2$ are state predicates and $(c_1 \wedge \neg c_2).\sigma$ holds because both $c_1.\sigma$ and $\neg c_2.\sigma$ hold; $s$ is a stream; $g$ is an interval predicate and both $\neg g.[-3,3].s$ and $g.[10,100).s$ hold, where $[-3,3]$ is the closed interval from $-3$ to 3 and $[10,100)$ is the right-open interval from 10 to 100. □

There are several operators for evaluating interval and state predicates (see Sections 4.1 and 5.1). For data refinement, we use the following operators on a state predicate $c$, interval $\Delta$ and stream $s$.

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

Thus, $(\boxdot c).\Delta.s$ holds iff $c$ holds in all states of $s$ within $\Delta$, whereas $(\diamondsuit c).\Delta.s$ holds iff $c$ holds in some state of $s$ within $\Delta$.

**Example 2.** Interval predicate $g$ as defined in Example 1 may be written as $\lambda\, \Delta, s \bullet \boxdot(\lambda\, \sigma \bullet \sigma.u \geq 300).\Delta.s$. We assume pointwise lifting and omit the $\lambda$ terms in the expression, so the interval predicate above is simply written as $\boxdot(u \geq 300)$. Similarly, $g_0$ defined in Example 7 may be written more succinctly as $\diamondsuit c_1$. □

There are numerous interval predicate operators (e.g., [21, 39, 50]). For refinement, we use the following operator, where $g$ is an interval predicate, $\Delta$ is an interval, $s$ is a stream. We define $\mathsf{empty}.\Delta \mathrel{\widehat{=}} (\Delta = \varnothing)$, i.e., $\mathsf{empty}.\Delta$ holds iff $\Delta$ is the empty interval.

$$(\ominus\, g).\Delta.s \quad \mathrel{\widehat{=}} \quad \neg \mathsf{empty}.\Delta \wedge \exists \Delta_0{:}\, \mathit{Intv} \bullet \Delta_0 \propto \Delta \wedge \neg \mathsf{empty}.\Delta_0 \wedge g.\Delta_0.s$$

Thus, $(\ominus\, g).\Delta.s$ holds iff $\Delta$ is non-empty and there exists a non-empty interval $\Delta_0$ that immediately precedes $\Delta$ such that $g.\Delta_0.s$ holds.

### *3.2. Interval relations*

Interval predicates enable one to reason about properties that take time, however, only define properties over a single state space. Proving data refinement via simulation requires one to relate behaviours over a concrete state space to behaviours over an abstract space. Hence, we combine the ideas behind state relations and interval predicates and obtain *interval relations*, which were introduced in [15].

An *interval relation* over $Y$ and $Z$ relates streams of $Y$ and $Z$ over intervals and is a mapping of type $\mathit{IntvRel}_{Y,Z} \mathrel{\widehat{=}} \mathit{Intv} \to \mathit{Stream}_Y \to \mathit{Stream}_Z \to \mathbb{B}$. Operators $\boxdot$ and $\diamondsuit$ are extended to relations over an interval. In particular, for $r \in \mathit{StateRel}_{Y,Z}$, interval $\Delta$, and streams $y \in \mathit{Stream}_Y$, $z \in \mathit{Stream}_Z$, we define:

$$(\boxdot r).\Delta.y.z \mathrel{\widehat{=}} \forall t{:}\, \Delta \bullet r.(y.t).(z.t) \qquad (\diamondsuit r).\Delta.y.z \mathrel{\widehat{=}} \exists t{:}\, \Delta \bullet r.(y.t).(z.t)$$

Another example of an interval relation based on $r$ is

$$\lambda\, \Delta{:}\, \mathit{Intv}, y{:}\, \mathit{Stream}_Y, z{:}\, \mathit{Stream}_Z \bullet \forall t_1, t_2{:}\, \Delta \bullet t_1 < t_2 \Rightarrow r.(y.t_1).(z.t_2)$$

which states that $y.t_1$ is related to $z.t_2$ via $r$ if $t_1, t_2 \in \Delta$ and $t_1 < t_2$. Stating such a relationship would require introduction of auxiliary (history) variables in the case of state relations, but are straightforward when using interval relation.

Interval relations (like interval predicates) may refer to properties outside a given interval. When using them to reason about data refinement, one must use a restricted class of relations that only refer to properties within the interval under consideration. This is defined using the following function for streams $y$ and $z$ and interval $\Delta$, where '$\lhd$' denotes domain restriction.

$$y \overset{\Delta}{=\!=} z \quad \widehat{=} \quad (\Delta \lhd y = \Delta \lhd z)$$

Thus $y \overset{\Delta}{=\!=} z$ holds iff the states of $y$ and $z$ within $\Delta$ match, i.e., $\forall t{:}\,\Delta \bullet y.t = z.t$.

**Definition 3.** *An interval relation $R \in IntvRel_{Y,Z}$ is* externally independent *iff for any $\Delta \in Intv$, $y \in Stream_Y$, $z \in Stream_Z$, we have $R.\Delta.y.z \Rightarrow \forall y'{:}\,Stream_Y, z'{:}\,Stream_Z \bullet y \overset{\Delta}{=\!=} y' \wedge z \overset{\Delta}{=\!=} z' \wedge R.\Delta.y'.z'$.*

Thus, for example, $\ominus(\diamond r)$ is not externally independent, but $\boxdot r$ is.

*3.3. Interval-based refinement*

Interval predicates and relations provide the necessary theoretical background for our interval-based notion of refinement. First, we formalise the structure of a system in terms of an initialisation, main program and finalisation, which are defined over observable and representation variables.

Suppose $N, Z \subseteq Var$, respectively denote the sets of observable and representation variables. A *system* is defined by a tuple:

$$C \quad \widehat{=} \quad (CI, CP, CF)_{N,Z}$$

where $CI \in IntvRel_{N,Z}$ models the initialisation, $CP \in IntvPred_Z$ models the system processes, and $CF \in IntvRel_{Z,N}$ denotes system finalisation. We have two restrictions on *valid systems*, namely that:

1. $CI$ is externally independent, and
2. $CF$ is of the form $\boxdot r$ for some relation $r$.

For the rest of this paper, we assume the systems under consideration are valid. We say stream $s$ is an *observable behaviour* of a system $C$ over interval $\Delta$ in stream $z$ iff $obs.C.\Delta.z.s$ holds, where

$$obs.C.\Delta.z.s \quad \widehat{=} \quad (\ominus CI).\Delta.s.z \wedge CP.\Delta.z \wedge CF.\Delta.z.s$$

Thus, the system initialises as defined by $CI$ in some interval immediately preceding $\Delta$, executes as defined by $CP$ over $\Delta$, then finalises as defined by $CF$ in $\Delta$ to produce $s$. Using this, we define refinement as follows.

**Definition 4 (Refinement).** *An abstract system $A \mathrel{\widehat{=}} (AI, AP, AF)_{N,Y}$ is* refined *by a concrete system $C \mathrel{\widehat{=}} (CI, CP, CF)_{N,Z}$, denoted $A \sqsubseteq C$ iff*

$$\forall z{:}\,Stream_Z, \Delta{:}\,Intv, s{:}\,Stream_N \bullet obs.C.\Delta.z.s \Rightarrow \exists y{:}\,Stream_Y \bullet obs.A.\Delta.y.s \tag{2}$$

Thus, whenever the concrete system produces an observable behaviour $s$, there must exist an abstract stream $y$ such that the abstract system is able to produce $s$ as an observable behaviour.

By viewing both initialisation and finalisation as interval relations, Definition 4 is now much simpler than the definition of refinement in [15].

**Example 5.** Consider the abstract and concrete single process programs below.

| $AInit{:}\ a = 1$<br>$a{:=}\ a + a$ |
|---|

| $CInit{:}\ b = 2$<br>$a{:=}\ b$ |
|---|

Suppose that $a$ is observable for both abstract and concrete programs, i.e., $N = \{a\}$. The internal variables are $Y = \{a\}$ and $Z = \{a, b\}$, and the abstract and concrete programs are modelled by interval relations and predicates below, where $s$, $y$ and $z$ are observable, abstract and concrete streams, respectively. Operator ';' allows one to 'chop' a given interval, so that $g_1$ ; $g_2$ holds in a finite interval $\Delta$ if the interval can be partitioned into two parts so that $g_1$ holds for the first part and $g_2$ holds for the second (see Table 2 for a formal definition). Formalisation of the behaviours of the two programs above are based on the state relation $FR \mathrel{\widehat{=}} \lambda\, \sigma, \rho \bullet \sigma.a = \rho.a$, which states that the values of $a$ in the two given states are equal. Thus, we obtain:

$$\begin{array}{lcl@{\qquad}lcl} AI & \mathrel{\widehat{=}} & \lambda\, \Delta, s, y \bullet \neg \mathsf{empty}.\Delta \wedge \boxdot(a = 1).\Delta.y & CI & \mathrel{\widehat{=}} & \lambda\, \Delta, s, y \bullet \neg \mathsf{empty}.\Delta \wedge \boxdot(b = 2).\Delta.y \\ AP & \mathrel{\widehat{=}} & \exists k \bullet \boxdot(a + a = k);\ \boxdot(a = k) & CP & \mathrel{\widehat{=}} & \exists k \bullet \boxdot(b = k);\ \boxdot(a = k) \\ AF & \mathrel{\widehat{=}} & \boxdot FR & CF & \mathrel{\widehat{=}} & \boxdot FR \end{array}$$

Using Definition 4, it is possible to prove that both refinements $(AI, AP, AF)_{N,Y} \sqsubseteq (CI, CP, CF)_{N,Z}$ and $(CI, CP, CF)_{N,Z} \sqsubseteq (AI, AP, AF)_{N,Y}$ hold, i.e., the two systems are equivalent. □

The systems we aim to develop are however, much more complex than those in Example 5 and often use different internal representations of the observable behaviour. Furthermore, the number of steps the abstract and concrete systems take to produce an output may differ. Here, refinements must appeal to a *refinement relation*, which relates these internal representations and steps of the abstract and concrete systems, resulting in a *simulation-based* approach. The next section explains this approach in detail and describes how simulation-based proofs may be decomposed.

## 4. Simulation for interval-based refinement

As in the state-based setting, verifying data refinement directly is difficult. We develop a proof method akin to forward simulation, then develop numerous decomposition rules for commonly occurring operators. In Section 4.1, we present numerous operators for intervals, interval predicates and interval relations, which serves as background for the remainder of the section. Section 4.2 presents our notion of simulation and Section 4.3 presents decomposition rules.

### *4.1. Operators over intervals, interval predicates and relations*

This section reviews some operators and notations for intervals, interval predicates and interval relations that are used for the rest of the paper.

| Notation | Definition | Informal meaning |
|---|---|---|
| $\ell.\Delta$ | **if** $\mathsf{empty}.\Delta$ **then** $0$ **else** $\mathsf{lub}.\Delta - \mathsf{glb}.\Delta$ | length of $\Delta$ |
| $\mathsf{finite}.\Delta$ | $\mathsf{empty}.\Delta \vee \mathsf{lub}.\Delta \in \mathcal{T}$ | $\Delta$ has a finite upper bound |
| $\mathsf{infinite}.\Delta$ | $\neg \mathsf{finite}.\Delta$ | $\Delta$ has an infinite upper bound |
| $\Delta_1 < \Delta_2$ | $\neg \mathsf{empty}.\Delta_1 \wedge \neg \mathsf{empty}.\Delta_2 \wedge \forall t_1 \colon \Delta_1, t_2 \colon \Delta_2 \bullet t_1 < t_1$ | $\Delta_1$ occurs before $\Delta_2$ |
| $\Delta_1 \propto \Delta_2$ | $\mathsf{empty}.\Delta_1 \vee \mathsf{empty}.\Delta_2 \vee (\Delta_1 < \Delta_2 \wedge (\Delta_1 \cup \Delta_2 \in \mathit{Intv}_{\mathcal{T}}))$ | $\Delta_1$ immediately precedes $\Delta_2$ |
| $\frac{\Delta_1 \Delta_2}{\Delta}$ | $(\Delta_1 \propto \Delta_2) \wedge (\Delta = \Delta_1 \cup \Delta_2)$ | $\Delta_1$ and $\Delta_2$ partition $\Delta$ |

Table 1: Operators on intervals $\Delta$, $\Delta_1$ and $\Delta_2$

*Interval operators.* Table 1 lists some operators on intervals as well as their formal and informal meanings. We use $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ denote the *least upper bound* and *greatest lower bound* of $\Delta$, respectively. Note that $\mathsf{lub}.\Delta$ and $\mathsf{glb}.\Delta$ may not be elements of $\Delta$, e.g., if $\Delta \in \mathit{Intv}_{\mathbb{R}}$ is left open, then $\mathsf{glb}.\Delta \notin \Delta$. We define $\mathsf{glb}.\mathcal{T} \mathrel{\widehat{=}} -\infty$ and $\mathsf{lub}.\mathcal{T} \mathrel{\widehat{=}} \infty$, where $-\infty, \infty \notin \mathcal{T}$. If $\Delta_1 \propto \Delta_2$, we say $\Delta_1$ *adjoins* $\Delta_2$. Adjoining intervals $\Delta_1$

and $\Delta_2$ are disjoint (because $\Delta_1 < \Delta_2$), but contiguous across their boundary (because $\Delta_1 \cup \Delta_2 \in \mathit{Intv}_{\mathcal{T}}$). Furthermore, both $\Delta \propto \varnothing$ and $\varnothing \propto \Delta$ hold trivially for any interval $\Delta$. We say $\frac{\Delta_1 \Delta_2}{\Delta}$ holds iff $\Delta$ can be split into an initial portion $\Delta_1$ followed by $\Delta_2$.

*Interval predicates operators.* It is possible to define several interval predicate operators [39, 21, 50]; a selection of these are used in this paper. These are listed with their formal and informal meanings in Table 2. We assume *pointwise lifting* of boolean operators on interval predicates in the normal manner, e.g., if $g_1$ and $g_2$ are interval predicates, $\Delta$ is an interval and $s$ is a stream, we have $(g_1 \wedge g_2).\Delta.s = (g_1.\Delta.s \wedge g_2.\Delta.s)$.

| Notation | Definition | Informal meaning |
|---|---|---|
| $\underline{g}.\Delta.s$ | $g.\Delta.s \wedge \neg\mathsf{empty}.\Delta$ | $g$ holds and $\Delta$ is non-empty |
| $(\oplus\, g).\Delta.s$ | $\neg\mathsf{empty}.\Delta \wedge \exists \Delta_0 \bullet \Delta \propto \Delta_0 \wedge \underline{g}.\Delta_0.s$ | $g$ holds in some next interval |
| $(\Box g).\Delta.s$ | $\forall \Delta_0 \colon \mathit{Intv} \bullet \Delta_0 \subseteq \Delta \Rightarrow g.\Delta_0.s$ | $g$ holds in all subintervals |
| $(g_1\ ;\ g_2).\Delta.s$ | $\left(\exists \Delta_1, \Delta_2 \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge g_1.\Delta_1.s \wedge g_2.\Delta_2.s\right) \vee$ $(\mathsf{infinite}.\Delta \wedge g_1.\Delta.s)$ | $g_1$ holds then $g_2$ holds in $\Delta$, or $g_1$ holds in $\Delta$ and $\Delta$ is infinite |

Table 2: Interval predicate operators for an interval $\Delta$ and stream $s$

**Example 6.** Suppose $v$ is a variable such that in a stream $s$, both $\boxdot(v = 10).[0, 1).s$ and $\boxdot(v = 20).[1, 2].s$ hold. For example, $v$ may represent the voltage that instantaneously jumps from 10 to 20. Then we have that $(\boxdot(v = 10);\ \boxdot(v = 20)).[1, 2].s$ holds. However, $\Box(v = 20).[1, 2].s$ does not hold, but $\Box(v \leq 20).[1, 2].s$ does hold. □

When reasoning about programs and their properties, one must often reason about *universal implication*, i.e., if an interval predicate $g_1$ holds over an arbitrarily chosen interval $\Delta$ and stream $s$, then an interval predicate $g_2$ also holds over $\Delta$ and $s$. We define '$\Rrightarrow$' as follows; operators '$\equiv$' and '$\Lleftarrow$' are similarly defined.

$$g_1.\Delta \Rrightarrow g_2.\Delta \mathrel{\widehat{=}} \forall s \colon \mathit{Stream} \bullet g_1.\Delta.s \Rightarrow g_2.\Delta.s \qquad\qquad g_1 \Rrightarrow g_2 \mathrel{\widehat{=}} \forall \Delta \colon \mathit{Intv} \bullet g_1.\Delta \Rrightarrow g_2.\Delta$$

Interval predicates are assumed to be ordered using implication '$\Rrightarrow$' and the greatest fixed point allows $g^\omega$ to model both finite (including 0) and infinite iteration. This fixed point is guaranteed to exist by Knaster-Tarski because interval predicates form a complete lattice [22]. The iteration operators we use are given in Section 3. Note that both $g^{\omega+}$ and $g^\infty$ are special cases of $g^\omega$.

| Notation | Definition | Informal meaning |
|---|---|---|
| $g^\omega$ | $\nu z \bullet (g\ ;\ z) \vee \mathsf{empty}$ | potentially infinite iteration of $g$ |
| $g^{\omega+}$ | $g\ ;\ g^\omega$ | positive iteration of $g$ |
| $g^\infty$ | $\mu\, z \bullet g\ ;\ z$ | purely infinite iteration of $g$ |

Table 3: Iteration operators for an interval predicate $g$

**Example 7.** For $c_1$, $g$ and $s$ as defined in Example 1, if $g_0 \mathrel{\widehat{=}} \lambda\, \Delta, s \bullet \exists t \colon \Delta \bullet c_1.(s.t)$, then $(g_0\ ;\ g).[0, 100).s$ holds because both $g_0.[0, 10).s$ and $g.[10, 100).s$ hold. Note that there may be more than one possible way to split an interval when applying the definition of ';', e.g., both $g_0.[0, 20].s$ and $g.(20, 100).s$ also hold. □

*Interval relation operators.* Like interval predicates, we assume pointwise lifting of operators over state and interval relations in the normal manner and extend interval predicate operators from Table 2 to interval relations, e.g.,

$$(R_1\ ;\ R_2).\Delta.y.z \quad \widehat{=} \quad \left(\exists \Delta_1, \Delta_2 \colon \mathit{Intv} \bullet \tfrac{\Delta_1 \Delta_2}{\Delta} \wedge R_1.\Delta_1.y.z \wedge R_2.\Delta_2.y.z\right) \vee (\mathsf{infinite}.\Delta \wedge R_1.\Delta.y.z)$$

Thus, $R_1\ ;\ R_2$ holds in $\Delta$ for $y$ and $z$ iff either (a) $\Delta$ can be partitioned into adjoining intervals $\Delta_1$ and $\Delta_2$ such that $R_i$ holds in $\Delta_i$ for $y$ and $z$ or (b) $\mathsf{infinite}.\Delta$ holds and $R_1$ holds in $\Delta$ for $y$ and $z$.

If $R_1 \in \mathit{IntvRel}_{X,Y}$ and $R_2 \in \mathit{IntvRel}_{Y,Z}$ then for $\Delta \in \mathit{Intv}$, $x \in \mathit{Stream}_X$, $z \in \mathit{Stream}_Z$, we define the composition of $R_1$ and $R_2$ as

$$(R_1 \circ R_2).\Delta.x.z \quad \widehat{=} \quad \exists y \colon \mathit{Stream}_Y \bullet R_1.\Delta.x.y \wedge R_2.\Delta.y.z$$

Thus, $R_1 \circ R_2$ relates $x$ to $z$ in $\Delta$ iff there is an interval $y$ such that $R_1$ relates $x$ to $y$ in $\Delta$ and $R_2$ relates $y$ to $z$ in $\Delta$.

*4.2. Forward simulation*

In this section, we develop an interval-based notion of forward simulation. First, we review the state-based forward simulation rule in Figure 6, where $\sigma_0$, $\sigma$ are abstract states, $\tau_0$, $\tau$ are concrete states, *sref* is a state relation between abstract and concrete states, and $cp_i$ is a concrete step with corresponding abstract step $ap_i$. Here, if $\mathit{sref}(\sigma_0, \tau_0) \wedge cp_i(\tau_0, \tau)$ holds, then one must show that there exists a $\sigma$ such that $ap_i(\sigma_0, \sigma) \wedge \mathit{sref}(\sigma, \tau)$ holds.

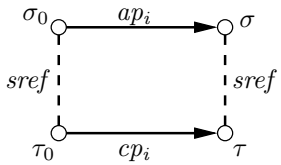



Figure 6: State-based forward simulation

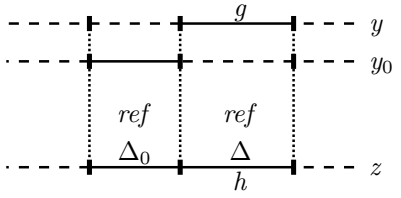



Figure 7: Visualisation of $\mathit{ref} \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right|$

For $Y, Z \subseteq \mathit{Var}$, assuming that $g \in \mathit{IntvPred}_Y$ and $h \in \mathit{IntvPred}_Z$ model the abstract and concrete systems, respectively, and that $\mathit{ref} \in \mathit{IntvRel}_{Y,Z}$ denotes the refinement relation, we define a notation $\mathit{ref} \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right|$, which denotes that $h$ *simulates* $g$ with respect to *ref*.

$$\begin{aligned} \mathit{ref} \bullet \left|\tfrac{Y\ :\ g}{Z\ :\ h}\right| \quad \widehat{=} \quad & \forall z \colon \mathit{Stream}_Z, \Delta, \Delta_0 \colon \mathit{Intv}, y_0 \colon \mathit{Stream}_Y \bullet \\ & (\Delta_0 \propto \Delta) \wedge \mathit{ref}.\Delta_0.y_0.z \wedge h.\Delta.z \Rightarrow \exists y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}.\Delta.y.z \wedge g.\Delta.y \end{aligned}$$

Thus, assuming the abstract and concrete streams are related by *ref* in the initial interval $\Delta_0$ and the concrete system (modelled by $h$) executes in an interval $\Delta$, it must be possible to execute the abstract system (modelled by $g$) over $\Delta$ such that *ref* holds between the abstract and concrete streams in $\Delta$. A visualisation of $\mathit{ref} \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right|$ is given in Figure 7. This is analogous to the state-based refinement diagram (Figure 6); *ref* holding between $y_0$ and $z$ in $\Delta_0$ corresponds to *sref* holding between $\sigma_0$ and $\tau_0$, execution of $h$ (and $g$) over $\Delta$ corresponds to the transitions from $\tau_0$ to $\tau$ (and $\sigma_0$ to $\sigma$), and *sref* holding between $\sigma$ and $\tau$ corresponds to *ref* holding between $y$ and $z$ in $\Delta$.

The following lemma establishes reflexivity and transitivity properties for $\mathit{ref} \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right|$.

**Lemma 8.** *Provided that* $\mathit{id}.\sigma.\tau \mathrel{\widehat{=}} (\sigma = \tau)$ *and both* $\mathit{ref}_1$ *and* $\mathit{ref}_2$ *are externally independent:*

$$\boxdot \mathit{id} \bullet \left|\tfrac{X\ :\ g}{X\ :\ g}\right| \qquad \text{(Reflexivity)}$$

$$\mathit{ref}_1 \bullet \left|\tfrac{X\ :\ f}{Y\ :\ g}\right| \wedge \mathit{ref}_2 \bullet \left|\tfrac{Y\ :\ g}{Z\ :\ h}\right| \ \Rightarrow\ (\mathit{ref}_1 \circ \mathit{ref}_2) \bullet \left|\tfrac{X\ :\ f}{Z\ :\ h}\right| \qquad \text{(Transitivity)}$$

Simulation is used to define an interval-based notion of *forward simulation* as follows.

**Definition 9 (Forward simulation).** *If* $A \mathrel{\widehat{=}} (AI, AP, AF)_{N,Y}$ *and* $C \mathrel{\widehat{=}} (CI, CP, CF)_{N,Z}$ *are abstract and concrete systems, respectively, and* $ref \in IntvRel_{Y,Z}$*, then we say* $ref$ *is a* forward simulation *from* $A$ *to* $C$ *iff* $ref$ *is externally independent,* $ref \bullet \left|\frac{Y\ :\ AP}{Z\ :\ CP}\right|$ *holds, and both of the following hold:*

$$\forall s{:}\, Stream_N, \Delta{:}\, Intv, z{:}\, Stream_Z \bullet CI.\Delta.s.z \;\Rightarrow\; \exists y{:}\, Stream_Y \bullet AI.\Delta.s.y \wedge ref.\Delta.y.z \tag{3}$$

$$\forall \Delta{:}\, Intv, z{:}\, Stream_Z, y{:}\, Stream_Y \bullet ref.\Delta.y.z \;\Rightarrow\; \forall s{:}\, Stream_N \bullet CF.\Delta.z.s \Rightarrow AF.\Delta.y.s \tag{4}$$

By (3) whenever the concrete initialisation holds for and observable stream $s$, interval $\Delta$ and concrete stream $z$, there must exist an abstract stream $y$ such that the abstract initialisation holds for $\Delta$, $s$ and $y$, and furthermore the refinement relation must hold in $\Delta$ between $y$ and $z$. By (4), assuming the refinement relation holds in $\Delta$ between $y$ and $z$, then for any observable stream $s$, the abstract finalisation holds in $\Delta$ whenever the concrete finalisation holds.

**Example 10.** Consider the abstract and concrete single process programs below.

| $AInit{:}\, true$<br>$a := a + 1$ | | $CInit{:}\, true$<br>$b := b + 1$ |
|---|---|---|

It is straightforward to see that the two programs have essentially the same functionality in that they both increment their internal variables $a$ and $b$. How does this reflect on the observed behaviour? We prove functional equivalence by making $a$ and $b$ visible as an observable variable $ab$ so that $N = \{ab\}$. We also have $Y = \{a\}$ and $Z = \{b\}$. In other words, $a$ and $b$ are internal representations of the observable variable $ab$ within the abstract and concrete systems, respectively. We define state relations $AFR \mathrel{\widehat{=}} \lambda\, \sigma, \rho \bullet \sigma.a = \rho.ab$ and $CFR \mathrel{\widehat{=}} \lambda\, \tau, \rho \bullet \sigma.b = \rho.ab$ as well as the following interval predicates and relations:

$$AP \;\mathrel{\widehat{=}}\; \exists k \bullet \boxdot(a + 1 = k);\ \boxdot(a = k) \qquad CP \;\mathrel{\widehat{=}}\; \exists k \bullet \boxdot(b + 1 = k);\ \boxdot(b = k)$$

$$AF \;\mathrel{\widehat{=}}\; \boxdot AFR \qquad CF \;\mathrel{\widehat{=}}\; \boxdot CFR$$

Then, defining

$$refR \;\mathrel{\widehat{=}}\; \lambda\, \sigma, \tau \bullet \sigma.a = \sigma.b \qquad ref \;\mathrel{\widehat{=}}\; \boxdot refR$$

it is possible to prove that $ref$ is a forward simulation from $(true, AP, AF)_{N,Y}$ to $(true, CP, CF)_{N,Z}$. □

The following theorem establishes soundness of our forward simulation rule with respect to interval-based data refinement.

**Theorem 11 (Soundness of forward simulation).** *Suppose the abstract and concrete systems are given by* $A \mathrel{\widehat{=}} (AI, AP, AF)_{N,Y}$ *and* $C \mathrel{\widehat{=}} (CI, CP, CF)_{N,Z}$*, respectively. Then* $A \sqsubseteq C$ *holds provided that there exists a* $ref \in IntvRel_{Y,Z}$ *such that* $ref$ *is a forward simulation between* $A$ *and* $C$*.*

Proof. We are required to prove (2). To this end, we fix $\Delta$, $z$ and $s$ and assume $obs.C.\Delta.z.s$ to obtain the following calculation.

$$\begin{array}{ll}
 & obs.C.\Delta.z.s \\
= & \quad \text{definition of } obs \text{ and } \ominus \\
 & \exists \Delta_0{:}\, Intv \bullet \Delta_0 \propto \Delta \wedge CI.\Delta_0.s.z \wedge CP.\Delta.z \wedge CF.\Delta.z.s \\
\Rightarrow & \quad \text{use } CI.\Delta.s.z, \text{ (3), and predicate logic} \\
 & \exists \Delta_0{:}\, Intv, y{:}\, Stream_Y \bullet \Delta_0 \propto \Delta \wedge AI.\Delta_0.s.y \wedge ref.\Delta_0.y.z \wedge CP.\Delta.z \wedge CF.\Delta.z.s \\
\Rightarrow & \quad \text{use } ref \bullet \left|\frac{Y\ :\ AP}{Z\ :\ CP}\right| \text{ and predicate logic} \\
 & \exists \Delta_0{:}\, Intv, y, y'{:}\, Stream_Y \bullet \Delta_0 \propto \Delta \wedge AI.\Delta_0.s.y \wedge (y \overset{\Delta_0}{=\!=} y') \wedge ref.\Delta.y'.z \wedge AP.\Delta.y' \wedge CF.\Delta.z.s \\
\Rightarrow & \quad AI \text{ is implicitly externally independent} \\
 & \exists \Delta_0{:}\, Intv, y'{:}\, Stream_Y \bullet \Delta_0 \propto \Delta \wedge AI.\Delta_0.s.y' \wedge ref.\Delta.y'.z \wedge AP.\Delta.y' \wedge CF.\Delta.z.s
\end{array}$$

$\Rightarrow$ using (4)
$\exists \Delta_0 : Intv, y' : Stream_Y \bullet \Delta_0 \propto \Delta \wedge AI.\Delta_0.s.y' \wedge ref.\Delta.y'.z \wedge AP.\Delta.y' \wedge AF.\Delta.y'.s$
$\Rightarrow$ definition of $\ominus$, and weakening
$\exists y' : Stream_Y \bullet (\ominus AI).\Delta.s.y' \wedge AP.\Delta.y' \wedge AF.\Delta.y'.s$
$=$ definition of *obs*
$\exists y' : Stream_Y \bullet obs.A.\Delta.y'.s$ □

**Example 12.** Using Theorem 11 and Example 10, we have that $(true, AP, AF)_{N,Y} \sqsubseteq (true, CP, CF)_{N,Z}$ holds. It is also possible to use forward simulation to prove that the refinement holds in the other direction, but to do this, one must use the interval relation $\boxdot(refR^{-1})$ as the forward simulation. □

*4.3. Decomposing simulations*

A benefit of state-based forward simulation [12] is the ability to decompose proofs and focus on individual steps of the concrete system. The proof obligation $ref \bullet \left|\frac{Y : g}{Z : h}\right|$ in the interval-based forward simulation definition (Definition 9) takes the entire interval of execution of the concrete and abstract systems into account. Hence, we develop a number of methods for simplifying proofs of $ref \bullet \left|\frac{Y : g}{Z : h}\right|$ by decomposing it over common programming constructs. Further simplifications are achieved by introducing notational conventions for commonly occurring forms of interval predicates.

As *ref* relates concrete and abstract streams, whereas the concrete system only refers to the concrete stream (similarly abstract system), we define $(g \uparrow).\Delta.y.z \mathrel{\widehat{=}} g.\Delta.y$ and $(g \downarrow).\Delta.y.z \mathrel{\widehat{=}} g.\Delta.z$, which allows one to lift interval predicates to the level of relations via a projection. It is straightforward to verify distribution of projection through interval predicate operators.

**Lemma 13 (Distribute projection).** *Suppose $g, g_1, g_2$ are interval predicates, $\odot \in \{;\ , \vee, \wedge, \Rightarrow, \Leftrightarrow\}$ and $\updownarrow \in \{\uparrow, \downarrow\}$. Then each of the following holds:*

$$(\neg g)\updownarrow \equiv \neg(g\updownarrow) \tag{5}$$
$$(g_1 \odot g_2)\updownarrow \equiv (g_1\updownarrow) \odot (g_2\updownarrow) \tag{6}$$
$$(g\updownarrow)^\omega \equiv g^\omega\updownarrow \tag{7}$$

Proving $ref \bullet \left|\frac{Y : g}{Z : h}\right|$ requires one to show that there exists an abstract stream $y$ such that $g$ holds for $y$ and *ref* holds between $y$ and a concrete stream $z$. Lemma 14 below enables decomposition of $ref \bullet \left|\frac{Y : g}{Z : h}\right|$. To this end, we introduce the following notation:

$$h \Vdash_{Y,Z} ref \mathrel{\widehat{=}} \forall z : Stream_Z, \Delta, \Delta_0 : Intv, y_0 : Stream_Y \bullet \Delta_0 \propto \Delta \wedge ref.\Delta_0.y_0.z \wedge h.\Delta.z \Rightarrow \exists y : Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref.\Delta.y.z$$

Notation $h \Vdash_{Y,Z} ref$ captures the notion that, assuming *ref* holds in some previous interval, $h$ executes in a manner consistent with respect to *ref* and some abstract stream. Within $h \Vdash_{Y,Z} ref$, interval relation *ref* can neither be weakened nor strengthened in the trivial manner because it appears in both the antecendent and consequent of the implication.

These notational conventions can be used to split the proof of $ref \in IntvRel_{Y,Z}$ as described by the following lemma.

**Lemma 14.** *For any $Y, Z \subseteq Var$ and $ref \in IntvRel_{Y,Z}$ that is externally independent, $ref \bullet \left|\frac{Y : g}{Z : h}\right|$ holds if $h \Vdash_{Y,Z} ref$ and $ref \wedge (h \downarrow) \Rrightarrow (g \uparrow)$ hold.*

Proof. This holds by expanding the definitions and applying the property that $p \Rightarrow (\exists x \bullet q \wedge r)$ holds if both $p \Rightarrow \exists x \bullet q$ and $\forall x \bullet p \wedge q \Rightarrow r$ hold. □

Condition $ref \wedge (h \downarrow) \Rrightarrow (g \uparrow)$ may be decomposed using standard rules of logic. On the other hand, decomposition of $h \Vdash_{Y,Z} ref$ depends on the structure of $h$, and the lemma below enables simplification for common programming constructs modelled by $h$. As the names imply, (Sequential) is used for sequential composition, (Iteration) is used for looping constructs (e.g., while loops) and (Non-deterministic choice) is used for non-deterministic choice (e.g., if-then-else statements). The fourth rule (Weaken) allows simplification of the interval predicate $h_1$ that models the program behaviour.

**Lemma 15.** *If $Y, Z \subseteq Var$, $h, h_1, h_2 \in IntvPred_Z$, $\epsilon \in \mathcal{T}$ is a constant, and $ref \in IntvRel_{Y,Z}$, then each of the following holds, provided $ref$ is externally independent.*

$$h_1 \Vdash_{Y,Z} ref \wedge h_2 \Vdash_{Y,Z} ref \Rightarrow (h_1 \,;\, h_2) \Vdash_{Y,Z} ref \qquad \text{(Sequential)}$$

$$h \Vdash_{Y,Z} ref \Rightarrow (h \wedge \epsilon \leq \ell)^{\omega+} \Vdash_{Y,Z} ref \qquad \text{(Iteration)}$$

$$(h \Vdash_{Y,Z} ref_1) \vee (h \Vdash_{Y,Z} ref_2) \Rightarrow h \Vdash_{Y,Z} (ref_1 \vee ref_2) \qquad \text{(Non-deterministic choice)}$$

$$(h_2 \Vdash_{Y,Z} ref) \wedge (h_1 \Rrightarrow h_2) \Rightarrow h_1 \Vdash_{Y,Z} ref \qquad \text{(Weaken)}$$

If a refinement relation operates on two disjoint portions of the stream, it is possible to split the refinement as follows. Disjointness allows one to prove mixed refinement, where the system states are split into disjoint subsets and different refinement relations are used to verify refinement between these subsets.

**Lemma 16 (Disjointness).** *Let $W, X, Y, Z \subseteq Var$ such that $Y \cap Z = \varnothing$, $W \cup X = Y$ and $W \cap X = \varnothing$. If $h_1, h_2 \in IntvPred_Z$, $ref_W \in IntvRel_{W,Z}$, $ref_X \in IntvRel_{X,Z}$, and $\star \in \{\wedge, \vee\}$, then the following holds provided both $ref_W$ and $ref_X$ are externally independent.*

$$(h_1 \Vdash_{W,Z} ref_W) \wedge (h_2 \Vdash_{X,Z} ref_X) \quad \Rightarrow \quad (h_1 \wedge h_2) \Vdash_{Y,Z} (ref_W \star ref_X)$$

With methods for proving refinement via forward simulation in place, we work towards our two examples showing how discrete and real-time refinement may be performed. To demonstrate the advantages of our interval-based approach, we show how different expression evaluation operators from [30] may be encoded in our logic, which is used to simplify reasoning about fine-grained concurrency and sampling. Our real-time example additionally includes some aspects of the time bands theory to enable reasoning about system components over multiple time granularities.

## 5. Evaluating state assertions over intervals

We aim to use our interval-based logic to simplify verification and construction of fine-grained concurrent programs and real-time systems. To this end, we aim to enable one to write complex expressions involving potentially many shared variables whose stability may not be guaranteed due to interference from the environment. Here, the order in which the variables in such expressions are read may not be fixed. One solution is to expand such expressions so that the assignments are made explicit (as done in Figure 4). However, as the number of possible permutations is factorial to the number of variables in each expression, such solutions are not ideal.

Interval-based reasoning provides the opportunity to use methods for nondeterministic expression evaluation [30], which captures the possible low-level interleavings (e.g., Figure 4) at a higher-level of abstraction. These have been imported into our interval-based setting in [21, 20]. We present two types of expression evaluation: *actual states evaluation*, where the given expression is evaluated instantaneously, and *apparent states evaluation*, where expressions are evaluated by reading the values of its variables at potentially different times. Actual states evaluation is captured by operators '$\boxdot$' and '$\diamondsuit$' (Section 3.1), which we use in Section 5.1 to formalise stability over an interval. Apparent states evaluations are defined in Section 5.2.

| Notation | Definition | Informal meaning |
|---|---|---|
| $\overleftarrow{c}$ | $\underline{\boxdot c}\,;\ true$ | $c$ holds at the end of the given interval |
| $\overrightarrow{c}$ | $\mathsf{finite}\,;\ \underline{\boxdot c}$ | $c$ holds at the start of the given interval |
| $\mathsf{stable}.c$ | $\ominus\overrightarrow{c} \wedge \boxdot c$ | $c$ is stable in the given interval |
| $\mathsf{stable}.v$ | $\exists k\colon Val \bullet \mathsf{stable}.(v = k)$ | $v$'s value is stable with respect to its previous value |
| $\mathsf{stable}.V$ | $\forall v\colon V \bullet \mathsf{stable}.v$ | all variables in $V$ are stable |
| $\mathsf{r\_stable}.v$ | $\exists k\colon Val \bullet \overrightarrow{v = k} \wedge \oplus \overleftarrow{v = k}$ | $v$'s value is right stable |
| $\mathsf{r\_stable}.V$ | $\forall v\colon V \bullet \mathsf{r\_stable}.v$ | all variables in $V$ right stable |

Table 4: State predicate operators and variable stability — $c$ is a state predicate, $\Delta$ is an interval, $s$ is a stream, $v$ a variable and $V$ a set of variables

*5.1. Stability*

Intervals may be open or closed at either end and our framework aims to provide a uniform treatment of both discrete and real-time systems. Table 4 provides operators for evaluating state predicates at the endpoints of an interval, which are in turn used to define predicate and variable stability.

The definitions of $\overleftarrow{c}$ and $\overrightarrow{c}$ are straightforward; recall that the underline operator $\underline{g}$ states that $g$ holds in the given interval and that the interval is non-empty (Table 2). For variable $v$, $\mathsf{stable}.v$ says that the value of $v$ throughout the current interval is unchanged from its value at the end of some previous interval. Such definitions are necessary because the ';' operator (which models sequential composition and forms the basis for iteration) partitions a given interval into disjoint subintervals. Thus, $v = k$ holding at the beginning of the current interval does not guarantee $v$'s value was $k$ at the end of the previous interval. Similarly, right stability of $v$ (i.e., $\mathsf{r\_stable}.v$) ensures that the value of $v$ is the same at the end of the current interval and the beginning of the next interval, which is sometimes necessary to link the behaviour of the current interval to an interval that follows immediately after.

Note that other, more general forms of evaluation are possible [30]; see also Section 5.2. Moreover, the definitions in Table 4 are independent of the underlying instantiations of $\mathcal{T}$.

**Example 17.** Suppose $v$ and $s$ are as defined in Example 6. Both $\overrightarrow{v = 10}.[0, 1).s$ and $\overleftarrow{v = 20}.[1, 2].s$ hold, i.e., the limit of the value of $v$ approaching time 1 from the left differs from the value at time 1. Therefore $(\neg\,\mathsf{stable}.v).[1, 2].s$ holds. Additionally, if $\boxdot(v = 20).(2, 3).s$ holds, then both $\mathsf{r\_stable}.v.[1, 2].s$ and $\mathsf{stable}.v.(2, 3).s$ hold. □

*5.2. Apparent states evaluation*

In the presence of possibly interfering processes e.g., due to concurrency or a real-time environment, the values of variables are unstable in the interval of evaluation. Therefore a model that assumes expressions containing multiple shared variables can be evaluated instantaneously may not be implementable without the introduction of contention inducing locks [1, 32]. As we have done in Figure 4, one may split expression evaluation into a number of atomic steps to make the underlying atomicity explicit. However, as already discussed, this approach is undesirable as it causes the complexity of expression evaluation to increase factorially with the number of variables in an expression — evaluation of an expression with $n$ (global) variables would require one to check $n!$ permutations of the read order.

Interval-based reasoning enables one to incorporate methods for nondeterministically evaluating state predicates over an evaluation interval [30], which allow the possible permutations in the read order of variables to be considered at a high level of abstraction. For this paper, we use *apparent states evaluators*, which allow one to evaluate an expression $e$ with respect to the set of states that occur within an interval. Each variable of $e$ is assumed to be read at most once in the interval of evaluation, but at potentially

different instances. An apparent state is generated by picking a value for each variable from the set of actual values of the variable over the interval of evaluation [30]. For $\Delta \in \mathit{Intv}$ and $s \in \mathit{Stream}_V$, we define:

$$\mathit{apparent}.\Delta.s \quad \widehat{=} \quad \{\sigma{:}\,\Sigma_V \mid \forall\, v{:}\, V \bullet \exists t{:}\, \Delta \bullet \sigma.v = s.t.v\}$$

Two useful operators for sets of apparent states evaluation allow one to formalise that $c$ *definitely* hold regardless of when the variables of $c$ are sampled (denoted $\circledast c$) and $c$ *possibly* hold for some possible sampling of its variables (denoted $\circledcirc c$), which are defined as follows.

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

Hence, $(\circledast c).\Delta.s$ holds iff any possible sampling of the variables of $c$ in $s$ within $\Delta$ results in values such that $c$ evaluates to *true*. Similarly, $(\circledcirc c).\Delta.s$ holds iff there is some possible way of sampling the variables of $c$ in $s$ within $\Delta$ such that $c$ evaluates to *true*. Note that for any $t \in \Delta$, $s.t \in \mathit{apparent}.\Delta.s$, i.e., every actual state is a possible apparent state. However, there may be a state $\sigma \in \mathit{apparent}.\Delta.s$ such that $\sigma \neq s.t$ for all $t \in \Delta$. Thus, $\circledast c \Rrightarrow \boxdot c$ and $\diamondsuit c \Rrightarrow \circledcirc c$, but the converse does not necessarily hold.

**Example 18.** Consider an evaluation of $u < v$ with coarse-grained atomicity within $\Delta$ in stream $z$ as depicted by Figure 8. Suppose $u := 1\,;\ v := 1$ is executed by another parallel process from an initial state that satisfies $u, v = 0, 0$, where the writes to $u$ and $v$ occur at times $t_1 \in \Delta$ and $t_2 \in \Delta$, respectively. Assuming no other (parallel) modifications to $u$ and $v$, the set of actual states of $z$ within $\Delta$ are:

$$SS \quad = \quad \{\{u \mapsto 0, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 0\}, \{u \mapsto 1, v \mapsto 1\}\}$$

Evaluating $u < v$ in the set of actual states above always results in *false*. This is depicted in Figure 8, where $\neg\diamondsuit(u < v)$ (i.e., $\boxdot(u \geq v)$) holds within $\Delta$ in $z$ because $u \geq v$ holds in $z$ regardless of where $u$ and $v$ are read within $\Delta$ because both $u$ and $v$ are read at the same time.

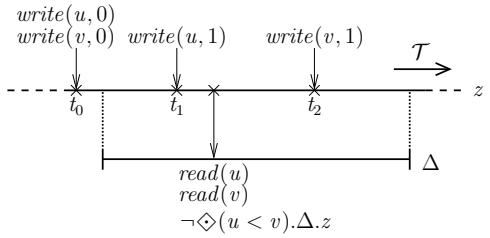



Figure 8: Actual states evaluation

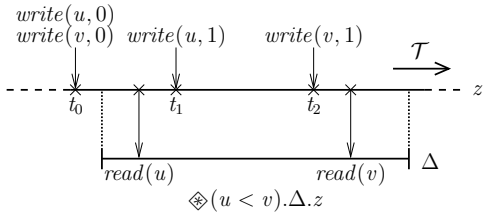



Figure 9: Apparent states evaluation

Now assume that process $p$ evaluates $u < v$ with fine-grained atomicity within $\Delta$. The set of apparent states corresponding to $\Delta$ is:

$$\mathit{apparent}.\Delta.s \quad = \quad SS \cup \{\{u \mapsto 0, v \mapsto 1\}\}$$

where the additional apparent state $\{u \mapsto 0, v \mapsto 1\}$ may be obtained by reading $u$ with value 0 (before $t_1$) and $v$ with value 1 (after $t_2$). Unlike the actual states evaluation, $u < v$ may result in *true* when evaluating in the apparent states as depicted in Figure 9. In particular, this occurs if $u$ is read before its value is updated to 1 and $v$ is read after its value is updated to 1.

Note that $v = v$ still only has one possible value, *true*, i.e., apparent states evaluation assumes that the same value of $v$ is used for both occurrences of $v$. □

Under certain stability assumptions on the variables of $c$, it is possible to strengthen both $\circledast c \Rrightarrow \boxdot c$ and $\diamondsuit c \Rrightarrow \circledcirc c$ into an equivalence. In particular, assuming $\mathit{vars}.c$ denotes the free variables of state predicate $c$, we obtain the following lemma from [30]. The theory in [30] is with respect to sets of states, which may be imported to our context by taking the states of the given stream within the interval under consideration as the set of states in [30].

**Lemma 19.** *If $v$ is a variable and $c$ a state predicate, then* $\mathsf{stable}.(vars.c \backslash \{v\}) \Rrightarrow (\circledast c = \boxdot c) \wedge (\circledcirc c = \diamondsuit c)$.

Thus, by Lemma 19, instability of at most one variable is enough to guarantee equivalence of actual state and apparent state evaluators.

## 6. Fine-grained concurrent programs

We now use our logic to model the behaviour of the programs in Figures 1 and 2, then prove data refinement between them using the framework that we have developed. Section 6.1 provides an interval-based semantics for both examples and Section 6.2 proves refinement between them for the semantics in Section 6.1.

### *6.1. An interval-based semantics*

We refrain from introducing a formal programming syntax with an interval-based semantics as this also introduces an additional layer of complexity. Instead, we present a possible interval-based semantics for our running example (Figures 1 and 2) directly by modelling the behaviours of both program using interval predicates. We interpret concurrency using a true concurrency semantics. Note that this interpretation below is not the only possibility — one could also, for example, give an interval-based interleaved semantics (e.g., [46, 7]). However, such a treatment would defeat the purpose of this example, which is to show that interval predicates form a natural basis for reasoning about true concurrency.

We formalise the behaviours of *AInit* and *CInit* as follows, where $\Delta \in \mathit{Interval}$, $s \in \mathit{Stream}_N$, $y \in \mathit{Stream}_Y$ and $z \in \mathit{Stream}_Z$.

$$\mathit{AInit}.\Delta.s.y \;\widehat{=}\; \underline{\boxdot \neg \mathit{grd}}.\Delta.y \qquad\qquad \mathit{CInit}.\Delta.s.z \;\widehat{=}\; \underline{\boxdot (v \leq u)}.\Delta.z$$

Hence, initialisation of the abstract systems ensures that $\neg \mathit{grd}$ holds throughout the given interval, and that the interval is non-empty. The concrete initialisation is similar.

As with state-based data refinement [13], we have freedom to instantiate the finalisations depending on which aspects of the systems we would like to make observable. For this example, we assume the internal representations of $m$ at both the abstract and concrete levels are equal to an observable variable $M$. Recall that for our system to be valid, finalisation must be an interval predicate of the form $\boxdot r$, where $r$ is a state relation. Therefore, for $\sigma \in \Sigma_Y$, $\rho \in \Sigma_N$ and $\tau \in \Sigma_Z$, we define

$$\mathit{fa}.\sigma.\rho \;\widehat{=}\; (\sigma.m = \rho.M) \qquad\qquad \mathit{fc}.\tau.\rho \;\widehat{=}\; (\sigma.m = \rho.M)$$

then obtain

$$\mathit{AFin} \;\widehat{=}\; \boxdot \mathit{fa} \qquad\qquad \mathit{CFin} \;\widehat{=}\; \boxdot \mathit{fc}$$

Next, we formalise the behaviours of the abstract and concrete processes. The *parallel composition* of processes $p$ and $q$ over an interval $\Delta$ is defined as the conjunction of the behaviours of both $p$ and $q$ over $\Delta$ (also see [16, 17]).[3] The behaviour of sequential composition is modelled using ';', while non-deterministic choice is modelled by disjunction. Note that we assume ';' binds more tightly than binary boolean operators. To model fine-grained concurrency, evaluation of a guard $b$ is interpreted as $\circledcirc b$ (see Section 5.2). Below, we assume the existence of a state predicate $\mathit{Term}.p$ that holds[4] iff process $p$ has terminated and we define

$$\mathsf{Term}_p \mathrel{\widehat{=}} \boxdot \mathit{Term}.p$$

[3] Note that others have also treated parallel composition as conjunction [1, 31]. However, Abadi and Lamport [1] admit interleaving within their conjunction operator while Hehner only admits disjoint concurrency [31].

[4] Formally, $\mathit{Term}.p$ can be encoded using program counters, but we leave this semi-formal here for simplicity. In particular, one could introduce a label *term* for termination, and define $\mathit{Term}.p \mathrel{\widehat{=}} (pc_p = \mathit{term})$.

Thus, we obtain:

$$\overbrace{\begin{pmatrix} \circledast grd\,;\ \underline{\boxdot(m=1)} \vee \\ \circledast \neg grd\,;\ \underline{\boxdot(m=2)} \end{pmatrix}\,;\ \mathsf{Term}_{ap}}^{\text{Process } ap} \quad \wedge \quad \overbrace{\begin{pmatrix} \circledast b\,;\ \underline{\boxdot(grd)} \vee \\ \circledast \neg b \end{pmatrix}\,;\ \mathsf{Term}_{aq}}^{\text{Process } aq} \tag{8}$$

$$\overbrace{\begin{pmatrix} \circledast(u<v)\,;\ \underline{\boxdot(m=1)} \vee \\ \circledast(u\geq v)\,;\ \underline{\boxdot(m=2)} \end{pmatrix}\,;\ \mathsf{Term}_{cp}}^{\text{Process } cp} \quad \wedge \quad \overbrace{\begin{pmatrix} \circledast(0<w)\,;\ \exists k \bullet \circledast(k=(u+1))\,;\ \underline{\boxdot(v=k)} \vee \\ \circledast(0\geq w)\,;\ \exists k \bullet \circledast(k=(u-1))\,;\ \underline{\boxdot(v=k)} \end{pmatrix}\,;\ \mathsf{Term}_{cq}}^{\text{Process } cq} \tag{9}$$

Condition (8) models the concurrent behaviour of processes $ap$ and $aq$. Process $ap$ either behaves as $\circledast grd\,;\ \underline{\boxdot(m=1)}$ ($grd$ evaluates to true, then the behaviour of $m := 1$ holds, i.e., the interval under consideration is non-empty and $m = 1$ holds throughout the interval) or $\circledast \neg grd\,;\ \underline{\boxdot(m=2)}$ ($\neg grd$ evaluates to true, then the behaviour of $m := 2$ holds). Process $aq$ is similar, but also models the assignment to $grd$. We now explain execution of the abstract system as formalised by (8). Note that ';' distributes over '$\vee$', but does not necessarily distribute over '$\wedge$' [22]. Furthermore, the points at which the intervals are partitioned using ';' within (8) and (9) are unsynchronised. For example, suppose process $ap$ behaves as $\circledast grd\,;\ \underline{\boxdot(m=1)}\,;\ \mathsf{Term}_{ap}$ and $aq$ behaves as $\circledast b\,;\ \underline{\boxdot grd}\,;\ \mathsf{Term}_{aq}$ within interval $\Delta$ of stream $y$, i.e.,

$$((\circledast grd\,;\ \underline{\boxdot(m=1)}\,;\ \mathsf{Term}_{ap}) \wedge (\circledast b\,;\ \underline{\boxdot grd}\,;\ \mathsf{Term}_{aq})).\Delta.y$$

holds for some interval $\Delta$ and abstract stream $y$. By pointwise lifting, this holds iff both of the following hold:

$$(\circledast grd\,;\ \underline{\boxdot(m=1)}\,;\ \mathsf{Term}_{ap}).\Delta.y \qquad\qquad (\circledast b\,;\ \underline{\boxdot grd}\,;\ \mathsf{Term}_{aq}).\Delta.y$$

The ';' operators within the two formulae above may split $\Delta$ independently. A visualisation of a possible splitting is given in Figure 10.

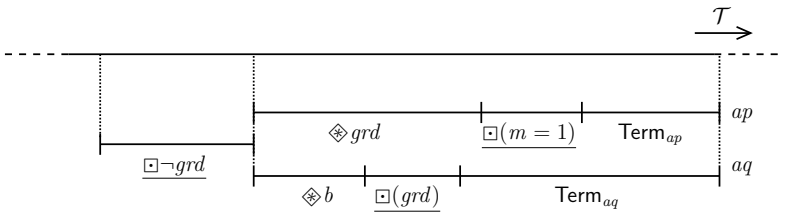



Figure 10: Possible execution of the abstract program in Figure 1

Condition (9), which models the concurrent execution of the concrete program has the same overall structure of (8). One point of difference is the semantics of assignments of the form $w := e$, where $e$ is an non-constant expression. Namely like [21, 17], we split an assignment into two parts; first $e$ is evaluated, which returns a constant value $k$, then the value of $w$ is updated to $k$. The evaluation of $e$ to $k$ may be modelled by $\circledast(e = k)$ to capture the fine-grained atomicity. Furthermore, to see that the semantics is accurate, we note that interval predicate $\circledast(u < v)$ is equivalent to

$$\exists u_p, v_p \bullet ((\diamond(u_p = u)\,;\ \diamond(v_p = v)) \vee (\diamond(v_p = v)\,;\ \diamond(u_p = u)))\,;\ (u_p < v_p)$$

Hence, the fine-grained behaviour of Figure 2 is captured without having to explicitly transform the guard evaluation at $cp_1$ into individual reads as done in Figure 4.

*6.2. Proving refinement*

With the necessary theoretical background and interval-based semantics of our example programs in place, we now verify data refinement between them. First, using Lemma 19, we simplify (8) and (9) and replace each occurrence of '$\diamondsuit$' by '$\fatsemi$' as follows:

$$\big(\fatsemi grd\,;\ \underline{\boxdot(m=1)} \vee \fatsemi\neg grd\,;\ \underline{\boxdot(m=2)}\big)\,;\ \mathsf{Term}_{ap} \wedge \big(\fatsemi b\,;\ \underline{\boxdot grd} \vee \fatsemi\neg b\big)\,;\ \mathsf{Term}_{aq} \qquad \text{(Abs-IP)}$$

$$\big(\fatsemi(u<v)\,;\ \underline{\boxdot(m=1)} \vee \fatsemi(u\geq v)\,;\ \underline{\boxdot(m=2)}\big)\,;\ \mathsf{Term}_{cp} \wedge \big(CQ1 \vee CQ2\big)\,;\ \mathsf{Term}_{cq} \qquad \text{(Conc-IP)}$$

where

$$CQ1 \mathrel{\widehat{=}} \fatsemi(0<w)\,;\ \exists k \bullet \fatsemi(k=(u+1))\,;\ \underline{\boxdot(v=k)}$$
$$CQ2 \mathrel{\widehat{=}} \fatsemi(0\geq w)\,;\ \exists k \bullet \fatsemi(k=(u-1))\,;\ \underline{\boxdot(v=k)}$$

The representation variables of the abstract and concrete programs are given by $Y = \{grd, m\}$ and $Z = \{u, v, m\}$, respectively. We prove forward simulation using *ref* defined below (recalling that relation *var_rel* is defined in Section 2):

$$ref \quad \mathrel{\widehat{=}} \quad \boxdot var\_rel \wedge \boxdot(\mathsf{Term}_{cp}{\downarrow} \Rightarrow \mathsf{Term}_{ap}{\uparrow}) \wedge \boxdot(\mathsf{Term}_{cq}{\downarrow} \Rightarrow \mathsf{Term}_{aq}{\uparrow})$$

By Theorem 11, we must prove each of the following:

$$\forall \Delta\colon Intv, z\colon Stream_Z, s\colon Stream_N \bullet CInit.\Delta.s.z \quad \Rightarrow \quad \exists y\colon Stream_Y \bullet AInit.\Delta.s.y \wedge ref.\Delta.y.z \qquad (10)$$

$$ref \bullet \left|\frac{Y\ :\ \text{(Abs-IP)}}{Z\ :\ \text{(Conc-IP)}}\right| \qquad (11)$$

$$\forall z\colon Stream_Z, y\colon Stream_Y, \Delta\colon Intv, s\colon Stream_N \bullet$$
$$ref.\Delta.y.z \wedge CFin.\Delta.z.s \quad \Rightarrow \quad AFin.\Delta.y.s \qquad (12)$$

These proofs proceed as follows.

*Proof of (10).* Fixing $\Delta$, $s$ and $z$, then instantiating *CInit*, *AInit* and *rep*, we have

$$\begin{array}{ll} & \underline{\boxdot(v\leq u)}.\Delta.z \Rightarrow \exists y\colon Stream_Y \bullet \underline{\boxdot\neg grd}.\Delta.y \wedge \\ & \qquad (\boxdot var\_rel \wedge \boxdot(\mathsf{Term}_{cp}{\downarrow} \Rightarrow \mathsf{Term}_{ap}{\uparrow}) \wedge \boxdot(\mathsf{Term}_{cq}{\downarrow} \Rightarrow \mathsf{Term}_{aq}{\uparrow})).\Delta.y.z \\ \Leftarrow & \quad \text{definition of } \mathsf{Term}_{cp} \text{ and } \mathsf{Term}_{ap} \\ & \underline{\boxdot(v\leq u)}.\Delta.z \Rightarrow \exists y\colon Stream_Y \bullet \underline{\boxdot\neg grd}.\Delta.y \wedge \boxdot var\_rel.\Delta.y.z \\ \Leftarrow & \quad \text{pick a } y \text{ such that for all } t \in \Delta,\ (\neg grd).(y.t) \\ & true \end{array}$$

□

*Proof of (11).* We use Lemma 14, which requires that we show that both of the following hold.

$$\text{(Conc-IP)} \quad \Vdash_{Y,Z} \quad ref \qquad (13)$$

$$ref \wedge \text{(Conc-IP)}{\downarrow} \quad \Rrightarrow \quad \text{(Abs-IP)}{\uparrow} \qquad (14)$$

The proof of (13) is trivial by construction. Expanding the definitions of (Abs-IP) and (Conc-IP), then applying some straightforward propositional logic, (14) holds if both of the following hold.

$$ref \wedge \left(\begin{pmatrix}\fatsemi(u<v)\,;\ \underline{\boxdot(m=1)} \vee \\ \fatsemi(u\geq v)\,;\ \underline{\boxdot(m=2)}\end{pmatrix}\,;\ \mathsf{Term}_{cp}\right)\Big\downarrow \Rrightarrow \left(\begin{pmatrix}\fatsemi grd\,;\ \underline{\boxdot(m=1)} \vee \\ \fatsemi\neg grd\,;\ \underline{\boxdot(m=2)}\end{pmatrix}\,;\ \mathsf{Term}_{ap}\right)\Big\uparrow \qquad (15)$$

$$ref \wedge \big(\big(CQ1 \vee CQ2\big)\,;\ \mathsf{Term}_{cq}\big)\downarrow \Rrightarrow \big(\big(\fatsemi b\,;\ \underline{\boxdot grd} \vee \fatsemi\neg b\big)\,;\ \mathsf{Term}_{aq}\big)\uparrow \qquad (16)$$

We present details of the more complex condition (16). This proof uses the fact that $\mathsf{stable}.u$ holds in interval in which *cq* executes.

$$
\begin{array}{ll}
 & \mathit{ref} \wedge \big(\big(CQ1 \vee CQ2\big)\,;\ \mathsf{Term}_{cq}\big)\downarrow \\
\Rrightarrow & \quad \text{Lemma 13 (distribute projection),} \\
 & \quad \boxdot(u < \infty) \text{ and } \mathit{ref} \text{ splits, `;' is monotonic} \\
 & \left(\mathit{ref} \wedge \begin{pmatrix} \diamondsuit(0 < w)\,;\ \exists k \bullet \diamondsuit(k-1 = u)\,;\ \underline{\boxdot(v = k)} \vee \\ \diamondsuit(0 \geq w)\,;\ \mathit{true} \end{pmatrix}\Big\downarrow\right)\,;\ (\mathit{ref} \wedge \mathsf{Term}_{cq})\downarrow \\
\Rrightarrow & \quad \mathsf{stable}.u,\ (\diamondsuit c\,;\ \mathit{true}) \Rrightarrow \diamondsuit c, \text{ definition of } \mathit{ref} \\
 & \left(\mathit{ref} \wedge \begin{pmatrix} \diamondsuit(0 < w)\,;\ \exists k \bullet \mathit{true}\,;\ \underline{\boxdot((v = k) \wedge (u = k-1))} \vee \\ \diamondsuit(0 \geq w) \end{pmatrix}\Big\downarrow\right)\,;\ (\mathsf{Term}_{aq}\uparrow) \\
\Rrightarrow & \quad \text{logic} \\
 & \left(\mathit{ref} \wedge \begin{pmatrix} \diamondsuit(0 < w)\,;\ \mathit{true}\,;\ \underline{\boxdot(u < v)} \vee \\ \diamondsuit(0 \geq w) \end{pmatrix}\Big\downarrow\right)\,;\ (\mathsf{Term}_{aq}\uparrow) \\
\Rrightarrow & \quad \diamondsuit c\,;\ \mathit{true} \Rrightarrow \diamondsuit c \\
 & \left(\mathit{ref} \wedge \begin{pmatrix} \diamondsuit(0 < w)\,;\ \underline{\boxdot(u < v)} \vee \\ \diamondsuit(0 \geq w) \end{pmatrix}\Big\downarrow\right)\,;\ (\mathsf{Term}_{aq}\uparrow) \\
\Rrightarrow & \quad \text{use } \mathit{ref}\text{, Lemma 13 (distribute projection)} \\
 & \big(\big(\diamondsuit b\,;\ \underline{\boxdot \mathit{grd}} \vee \diamondsuit \neg b\big)\,;\ \mathsf{Term}_{aq}\big)\uparrow
\end{array}
$$

The proof of (15) has a similar structure: first '$\downarrow$' then *ref* are distributed through ';', and then $\boxdot \mathit{var\_rel}$ from *ref* is used to transform predicate on the left of ';' to its abstract counterpart, while $\boxdot(\mathsf{Term}_{cp}\downarrow \Rightarrow \mathsf{Term}_{ap}\uparrow)$ is used to transform $\mathsf{Term}_{cp}$ to $\mathsf{Term}_{ap}$. □

*Proof of (12).* We fix $z$, $y$, $\Delta$ and $s$, then expand the definitions of *ref*, *CFin* and *AFin* to obtain the following calculation after weakening the antecedent:

$$
\begin{array}{ll}
 & \boxdot \mathit{var\_rel}.\Delta.y.z \wedge \boxdot \mathit{fc}.\Delta.z.s \Rightarrow \boxdot \mathit{fa}.\Delta.y.s \\
= & \quad \text{expanding definition of } \boxdot \text{ and logic} \\
 & \forall t\colon \Delta \bullet \mathit{var\_rel}.(y.t).(z.t) \wedge \mathit{fc}.(z.t).(s.t) \Rightarrow \mathit{fa}.(y.t).(s.t) \\
= & \quad \text{expanding definitions of } \mathit{var\_rel}, \mathit{fc} \text{ and } \mathit{fa} \\
 & \forall t\colon \Delta \bullet ((y.t).m = (z.t).m) \wedge ((z.t).m = (s.t).M) \Rightarrow ((y.t).m = (s.t).M) \\
= & \quad \text{logic} \\
 & \mathit{true}
\end{array}
$$

□

*Discussion.* The example demonstrates data refinement for concurrency and the benefits this provides. The proofs themselves are succinct (and consequently easier to understand) because the reasoning is performed at a high level of abstraction. Expression evaluation is assumed to take time and evaluation operators such as '$\diamondsuit$' and '$\circledcirc$' are used to capture the inherent nondeterminism that results from the fine-grained concurrent executions during the interval of evaluation. Thus, the translation of the program in Figure 2 to the lower-level program Figure 4 that makes the nondeterminism in expression evaluation explicit is unnecessary. Instead, one is able to provide a semantics for the program in Figure 2 directly. Moreover, the concrete and abstract executes are matched in a more intuitive manner: the concrete guard evaluations are matched to the abstract guard evaluations, and the concrete assignments are matched to the abstract assignments. Finally, unlike a state-based forward simulation proof, which requires that a verifier explicitly decides which of the concrete steps are non-stuttering, then find a corresponding abstract step for each non-stuttering step, interval-based reasoning allows one to remove this analysis step altogether because the stuttering steps may be combined to form a single interval predicate.

## 7. Real-time systems

In this section, we present a systematic refinement-based development of a real-time controller. Our theory incorporates parts of the time-bands methodology [10, 9] to allow reasoning about multiple time granularities, e.g., delays in sampling and turning physical components on/off. We review accuracy and precision as used in the time-bands theory in Section 7.1. We present our example in Section 7.2 and our

abstract system in Section 7.3, where we establish safety of the system. In Section 7.4, we describe numerous issues with the abstract system, which are used to motivate a more complex implementation. In Section 7.5, we prove data refinement between the abstract and concrete systems, which in turn implies that the concrete system is correct.

*7.1. A time-bands approach to accuracy and precision*

In addition to demonstrating that our method can cope with data refinement for real-time systems, we also show how the framework can be used to reason about robust specifications with real-world delays. In a truly concurrent system, a sampled value of any variable represents an approximation of the true value of the variable in the environment, and reasoning about these is, in general, difficult. For real-time systems, our approach incorporates the time-bands methodology [9, 10, 21] to simplify reasoning. We refrain from presenting all the details of the time bands framework [10] to enable our presentation to better focus on the data refinement proof at hand.

The time-bands framework enables reasoning over multiple time granularities. Each time band represents a different time granularity (e.g., minutes, seconds, milliseconds). Formally, we assume *TimeBand* denotes the type of a time band, and we define the *precision* of $B \in \mathit{TimeBand}$, denoted $\mathsf{pn}.B \in \mathbb{R}_{>0}$, to be the unit of time for $B$. For each time band $B$, we distinguish its *events*, which are assumed to take at most $\mathsf{pn}.B$ units of time to complete. The full time-bands theory allows many types of relationships between different time bands to be stated [10]. For our purposes, it is enough to allow different system components to be specified at different time bands and to map their behaviours to a *base time band* where time is a modelled using $\mathbb{R}$.

A variable may be specified at multiple time bands; the *accuracy* of the variable in a time band refers to the maximum change in the value of the variable within any event of the time band. In general, accuracy may refer to a function on one or more variables thus, we formalise the concept as follows. For a time band $B$ and function $f$, we let $\mathsf{acc}.f.B \in \mathbb{R}_{\geq 0}$ denote the accuracy of $f$ in $B$. To relate precision and accuracy to intervals and streams we use the following function, which for a real-valued function $f$ returns the *maximum change* between the values of $f$ within $\Delta$ in a stream $s$.

$$(\mathsf{mc}.f).\Delta.s \quad \widehat{=} \quad \textbf{if } \mathsf{empty}.\Delta \textbf{ then } 0 \textbf{ else } (\textbf{let } vs = \{val \mid \exists t{:}\, \Delta \bullet val = (s.t).f\} \textbf{ in } \mathsf{lub}.vs - \mathsf{glb}.vs)$$

**Example 20.** Consider a variable *pressure* in the interval $\Delta$ depicted below. Then $\mathsf{mc}.\mathit{pressure}.\Delta < k_1 - k_2$. Note that this does not place any constraints on the maximum instantaneous rate of change of *pressure*. For example, at time $t$, the rate of change of *pressure* exceeds $k_1 - k_2$. If *pressure* is an output of a component such as a pump, using $\mathsf{mc}$ to specify pump behaviour provides one with the freedom to use different pump implementations (e.g., centrifugal vs. rotary) [27], whose instantaneous rates of change differ, but have the same accuracy. □

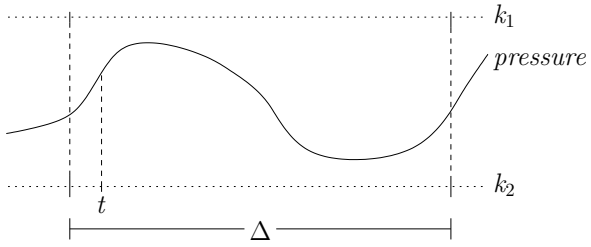



Figure 11: Maximum change in *pressure* over $\Delta$

We say $f$ is *consistent* with time band $B$ in interval $\Delta$ and stream $s$ iff $\mathit{consistent}.f.B.\Delta.s$ holds, where

$$\mathsf{con}.f.B.\Delta.s \quad \widehat{=} \quad \ell.\Delta \leq \mathsf{pn}.B \Rightarrow (\mathsf{mc}.f).\Delta.s \leq \mathsf{acc}.f.B$$

Thus, $\mathsf{con}.f.B.\Delta.s$ holds iff provided the length of $\Delta$ is below the precision of $B$, the maximum change to $f$ in $\Delta$ is below the accuracy of $f$ in $B$. Consistency is used to abstractly specify behaviours of events with respect to a time band. We extend this to sets of functions $F$ and sets time bands $BS$ as follows:

$$\mathsf{con}.F.B \mathrel{\widehat{=}} \forall f : F \bullet \mathsf{con}.f.B \qquad\qquad \mathsf{con}.f.BS \mathrel{\widehat{=}} \forall B : BS \bullet \mathsf{con}.f.B$$

The lemma below is from [21] and it allows one to relate sampled values to the actual values in the environment based on the accuracy of the variables being sampled.

**Lemma 21.** *If $x$ and $y$ are real-valued variables in time bands $B_x$ and $B_y$, respectively, and $\gg \in \{\geq, >\}$, then both of the following hold:*

$$\ell \leq min(\mathsf{pn}.B_x, \mathsf{pn}.B_y) \wedge \mathsf{con}.x.B_x \wedge \mathsf{con}.y.B_y \wedge \diamondsuit(x - \mathsf{acc}.x.B_x \gg y + \mathsf{acc}.y.B_y) \Rrightarrow \boxplus(x \gg y) \tag{17}$$

$$\ell \leq \mathsf{pn}.B_x \wedge \mathsf{con}.x.B_x \wedge \mathsf{stable}.y \wedge \diamondsuit(x - \mathsf{acc}.x.B \gg y) \Rrightarrow \boxplus(x \gg y) \tag{18}$$

Thus, (17) states that if the length of a sampling interval $\Delta$ is less than or equal to both $\mathsf{pn}.B_x$ and $\mathsf{pn}.B_y$, the given stream $s$ in consistent with respect to $x$ (in $B_x$) and $y$ (in $B_y$) and it is possible to sample $x$ and $y$ such that $x - \mathsf{acc}.x.B_x \gg y + \mathsf{acc}.y.B_y$ holds, then it must definitely be true that $x \gg y$ holds within $\Delta$ in $s$ (regardless of how $x$ and $y$ are sampled in $\Delta$). The second condition (18) follows from (17), where stability of $y$ is used to derive a simpler condition.

*7.2. A multi-pump system*

We develop a controller for the multipump system depicted Figure 12. Each pump removes water from a common reservoir into a common pipe. Such systems are often used in water distribution systems where the output of a single pump cannot guarantee adequate water pressure, or where levels of redundancy are required. We focus on the controller for a single pump (Figure 13), which must maintain the water pressure in the pipe between *High* and *Low*. The requirements for the pump are simple: the pump must be stopped (physically) when the water pressure exceeds *High*, and must be running (physically) when the water pressure is below *Low*.

We formalise these requirements as follows. The observable status of the pump is given by a variable *Pump*, whose value is of type $Status \mathrel{\widehat{=}} \{stopped, running, starting, stopping\}$. The observable water pressure at the *pressure* sensor in Figure 13 is given by a variable *Pressure*, which is a positive real-valued number. The pump aims to regulate *Pressure* between two critical values *Low* and *High* where $Low < High$. Thus, the safety requirements of the system are formalised by the interval predicates:

$$\boxdot(Pressure \geq High \quad \Rightarrow \quad Pump = stopped) \tag{19}$$

$$\boxdot(Pressure \leq Low \quad \Rightarrow \quad Pump = running) \tag{20}$$

Next, we develop an implementation system, which includes a controller and numerous environment assumptions, that ensures the safety requirements above are satisfied. The development proceeds in a step-wise manner as follows.

1. An abstract system that satisfies both safety requirements is developed. Parts of the abstract controller includes specification statements that must be further refined into executable code. The proof of safety is much simpler at the abstract level as the details of the system components are not fully designed.
2. Several issues with the abstract system are identified (see Section 7.4), which motivates development of a more complex concrete system. Here, further refinements including component decoupling is performed (see Figure 15) and the controller is specified using executable code.
3. Data refinement between the concrete and abstract systems is proved. Because the observable behaviours of the concrete are a subset of those of the abstract, this implies safety for the concrete system.

The time-bands theory is integrated into both the abstract and concrete levels to reason about delays in the controller and environment. This enables the formal specifications and reasoning to be more concise, which in turn improves readability.

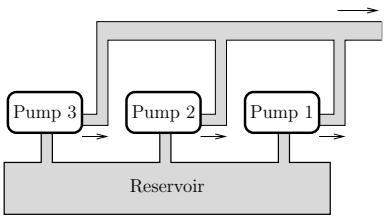



Figure 12: Multipump water distribution system

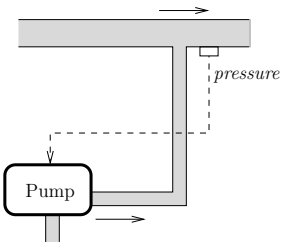



Figure 13: Single pump: dashed lines indicate sampling

*AInit*: $Stopped \wedge pressure \geq High$

**do** *true* **then**
  **if** $\neg Stopped \wedge pressure \geq High_T$ **then**
    $pump := stopped \dagger B_A$
  $\sqcap$ $\neg Running \wedge pressure \leq Low_T$ **then**
    $pump := running \dagger B_A$
  $\sqcap$ $[\![(21) \wedge (22) \wedge (23)]\!]$
  **endif**
**enddo**

Figure 14: Abstract controller pseudocode

*7.3. An abstract system*

In this section we develop an abstract system, which includes a (partially developed) pump controller and environment specification that satisfies the safety properties. At this level, we combine the delays of the controller and pump into a single time band $B_A$. This time band is decomposed into separate controller and pump time bands in the subsequent refinement (Section 7.4).

We first define the observable state of the system, and hence, start by defining the finalisation. To this end, we let $Y \mathrel{\widehat{=}} \{pressure, pump\}$ and assume that $\sigma \in \Sigma_Y$. The internal representation variables *pressure* and *pump* are mapped to observable variables $Pressure \in \mathbb{R}$ and $Pump \in Status$, respectively. We let $W \mathrel{\widehat{=}} \{Pressure, Pump\}$ and assume that $\rho \in \Sigma_W$ below to obtain the finalisation *AFin* below [5].

$$\begin{aligned} AFin\_Rel.\sigma.\rho &\mathrel{\widehat{=}} \sigma.pressure = \rho.Pressure \wedge \\ &\quad ((\sigma.pressure \geq High) \vee (\sigma.pressure \leq Low) \Rightarrow \sigma.pump = \rho.Pump) \\ AFin &\mathrel{\widehat{=}} \boxdot AFin\_Rel \end{aligned}$$

Thus, proof obligations (19) and (20) may be replaced by the following obligations on internal variables because $AFin \wedge ((21) \wedge (22))\uparrow \Rrightarrow ((19) \wedge (20))\downarrow$ holds.

$$\boxdot(pressure \geq High \quad \Rightarrow \quad pump = stopped) \tag{21}$$

$$\boxdot(pressure \leq Low \quad \Rightarrow \quad pump = running) \tag{22}$$

Defining $Stopped \mathrel{\widehat{=}} (pump = stopped)$ and $Running \mathrel{\widehat{=}} (pump = running)$, we obtain the controller in Figure 14 expressed using pseudocode. Note that the controller (Figure 14) uses *threshold* values $High_T$ and $Low_T$ (as opposed to critical values *High* and *Low*) to allow for the fact that *pressure* may change over the interval in which the *pump* is switched on/off.

Each iteration of the loop executes the main **if** statement, which nondeterministically chooses between one of the three guarded statements. At each iteration, input *pressure* is sampled and compared with high/low thresholds (given by $High_T$ and $Low_T$). The pump is switched off (on) whenever $pressure \geq High_T$ ($pressure \leq Low_T$) is detected and the pump is not already off (on). The third statement of the controller is an abstract specification [38, 21] that guarantees both (21) and (22) as well as (23) below [6]:

$$\overrightarrow{\neg Stopped \Rightarrow (pressure < High - Acc)} \wedge \overrightarrow{\neg Running \Rightarrow (pressure > Low + Acc)} \tag{23}$$

By (23), if *pump* is not *stopped* (*running*) at the end of the interval, then *pressure* is below $High - Acc$ (above $Low + Acc$). The specification statement is currently not executable by any machine, but using a refinement step (Section 7.4), we develop it into executable code.

[5] Note that it is possible to use a stronger finalisation so that the second conjunct in *AFin_Rel* is replaced by $\sigma.pump = \rho.Pump$, however, this potentially disallows future refinements of the pump behaviour when $Low < pressure < High$.

[6] Note that this condition is discovered as part of our proof using the methods in [21].

The first and second branches of the **if** statement of the abstract controller include $\dagger B_A$, which indicate that they are events of time band $B_A$. This means that for each iteration of the loop execution of the first two branches take $\mathsf{pn}.B_A$ time to complete.

Thus, the behaviour of each iteration of the **do** loop of the abstract controller is defined by interval predicate *PumpCont* below. Given that *outVars* is the set of output variables of the controller, each guard $c$ of the **if** statement (where $c$ is a state predicate) is modelled by interval predicate $\circledast c \wedge \mathsf{stable}.outVars$ [21], which holds for an interval $\Delta$ and stream $s$ if it is possible to sample the variables of $c$ in $s$ within $\Delta$ such that $c$ holds in the apparent state generated by sampling the variables of $c$ at potentially different times in $\Delta$. Additionally the output variables are stable during the interval of evaluation. The guard of the first branch of the **if** statement of the controller is therefore modelled by $\circledast(\neg Stopped \wedge (pressure \geq High_T)) \wedge \mathsf{stable}.pump$, which may be simplified to $\boxdot\neg Stopped \wedge \diamondsuit(pressure \geq High_T)$ (see Calculation 1 in Appendix B).

Therefore, we obtain the following interval predicates that model the guarded actions of the abstract controller. Recall that $\ell$ denotes the length of the interval.

$$
\begin{array}{rcl}
PumpCont_1 & \widehat{=} & ((\boxdot\neg Stopped \wedge \diamondsuit(pressure \geq High_T))\ ;\ \underline{\boxdot Stopped}) \wedge \mathsf{r\_stable}.pump \wedge \ell \leq \mathsf{pn}.B_A \\
PumpCont_2 & \widehat{=} & ((\boxdot\neg Running \wedge \diamondsuit(pressure \leq Low_T))\ ;\ \underline{\boxdot Running}) \wedge \mathsf{r\_stable}.pump \wedge \ell \leq \mathsf{pn}.B_A \\
PumpCont_3 & \widehat{=} & (21) \wedge (22) \wedge (23) \\
PumpCont & \widehat{=} & PumpCont_1 \vee PumpCont_2 \vee PumpCont_3
\end{array}
$$

To define the behaviour of the complete system one must also define the behaviour of the environment of the controller, which in this example controls the values of variable *pressure*. We obtain interval predicate *AEnv* below, which describes how *pressure* changes over an interval of length at most $\mathsf{pn}.B_A$.

$$AEnv \quad \widehat{=} \quad \Box(\mathsf{con}.pressure.B_A)$$

Thus, for any interval whose length is at most $\mathsf{pn}.B_A$, the maximum difference between two values of *pressure* is at most $\mathsf{acc}.pressure.B_A$.

We assume that the system is initially safe by assuming *pump* is initially *stopped* and that the *pressure* is sufficient (above *High*). The controller *PumpCont* executes with its environment *AEnv* in a truly concurrent manner and is defined by *AProg* below. Furthermore, because the controllers we consider are for reactive systems, we assume that they execute forever and hence use $^\infty$ iteration [7]. Thus we obtain the following, recalling that $W$ and $Y$ are the sets of observable and abstract internal variables, respectively.

$$
\begin{array}{rcl}
AInit\_Rel & \widehat{=} & \lambda\,\rho, \sigma \bullet \sigma.Stopped \wedge \sigma.pressure \geq High \\
AInit\_IR & \widehat{=} & \lambda\,\Delta, s, y \bullet \underline{\boxdot AInit\_Rel}.\Delta.s.y \wedge (\mathsf{r\_stable}.pump).\Delta.y \\
AProg & \widehat{=} & \ominus\, AInit \wedge PumpCont^\infty \wedge AEnv \\
ASys & \widehat{=} & (AInit\_IR, AProg, AFin)_{W,Y}
\end{array}
$$

Showing that a program satisfies a property (expressed as an interval predicate) is reduced to a proof that the behaviour of the program (expressed as an interval predicate) implies the property. The property under consideration may express safety, progress and timing requirements [21]. In particular, we show

$$AProg \quad \Rrightarrow \quad (21) \wedge (22) \tag{24}$$

This proof uses the techniques described in [21], and hence, its details are deferred to Appendix B.

*7.4. A concrete pump system*

The abstract system we have developed is safe with respect to the system requirements (formalised by (21) and (22)). However, there are issues in the abstract system that must be refined to obtain an implementation with an executable controller.

[7] Note that this is not a restriction of our theory, i.e., one could also just as easily model terminating or potentially terminating systems.

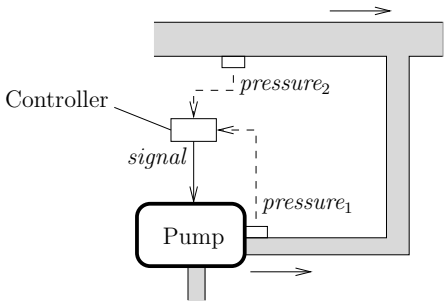



Figure 15: Single concrete pump

*CInit*: $Stopped \wedge pressure_1 + pressure_2 \geq High \wedge \neg signal$

**do** *true* **then**
  **if** $signal \wedge (pressure_1 + pressure_2 \geq CHigh_T)$ **then**
    $signal := false$
  $\sqcap$ $\neg signal \wedge (pressure_1 + pressure_2 \leq CLow_T)$ **then**
    $signal := true$
  **else idle**
  **endif** $\dagger B_C$
**enddo**

Figure 16: Concrete controller pseudocode

1. The pump controller contains a specification statement, which is not executable.
2. The controller directly modifies the various pump modes; in reality controllers set a signal, which in turn influences the pump mode.
3. There are only two pump modes: *running* and *stopped*, and the pump instantaneously switches between the two. In reality, physical constraints dictate that there are delays when switching between these modes, i.e., if a pump is running, there is a short duration of time before it can be stopped, and similarly if it is stopped, then there is a duration of time before the pump begins to run.
4. The abstract system is a simplification of the implementation because it only uses a single *pressure* reading. For the intended multipump implementation (see Figure 12), where each pump is individually controlled, one must typically make two pressure measurements $pressure_1$ and $pressure_2$ (see Figure 15) and determine the value of *pressure* from these [43].

Therefore, in the concrete system (see Figure 15), we perform three refinements.

1. The (software) pump controller, which reads pressure values and sends on/off signals, is decoupled from the (hardware) pump, which reads on/off signals and changes the status of the pump accordingly.
2. The value of the abstract reading *pressure* is inferred as the sum of two different pressure readings $pressure_1$ and $pressure_2$ in the concrete system as depicted in Figure 15.[8] For simplicity, we assume that the readings themselves are accurate.
3. The pump controller itself is refined so that the changes above are taken into account and that the controller does not contain any specification statements.

*Modelling the controller.* The controller has inputs $pressure_1$ and $pressure_2$, which are positive and real-valued, and an output *signal*, which is a boolean. Pseudocode for the controller is given in Figure 16. We use **else** to denote a branch whose guard holds iff all preceding guards evaluate to false. We will decompose the abstract controller into a concrete controller and a pump, each of which operate at different time granularities. Thus, we use $B_C$ and $B_P$ denote the time bands of the controller and pump, respectively. For our proof we assume:

$$\mathsf{pn}.B_C \leq \mathsf{pn}.B_A$$

i.e., that the sampling rate of the concrete implementation is potentially faster than the abstract. Other restrictions on the time bands (e.g., (36)) are introduced as part of the development.

The behaviour of the concrete controller is formalised by the interval predicates below. Like the abstract program using Lemma 19 and the fact that $\mathsf{stable}.signal$ signal holds, the guard of the first branch simplifies

[8] Here, for simplicity, we are assuming no other impeding factors such as friction loss at the pipes. One can account for additional factors affecting the total pressure in future refinements of the system.

to $\circledast(pressure_1 + pressure_2 \geq CHigh_T) \wedge \boxdot signal$. However, unlike the guard of $PumpCont_1$, further simplification of conjunct $\circledast(pressure_1 + pressure_2 \geq CHigh_T)$ is not possible because the expression itself refers to more than one variable whose value is not guaranteed to be stable [30]. Similarly, one must use the apparent states operator '$\circledast$' in both $Cont_2$ and $Cont_3$, as opposed to the simpler actual states evaluator '$\diamondsuit$'. We assume the existence of a constant lower limit $\epsilon$ where $0 < \epsilon$ on the time taken to execute each iteration of the controller, which ensures absence of Zeno-like behaviour.

$$Cont_1 \;\widehat{=}\; \left(\circledast(pressure_1 + pressure_2 \geq CHigh_T) \wedge \boxdot signal\right) ;\; \underline{\boxdot \neg signal} \wedge \mathsf{r\_stable}.signal \tag{25}$$

$$Cont_2 \;\widehat{=}\; \left(\circledast(pressure_1 + pressure_2 \leq CLow_T) \wedge \boxdot \neg signal\right) ;\; \underline{\boxdot signal} \wedge \mathsf{r\_stable}.signal \tag{26}$$

$$Cont_3 \;\widehat{=}\; \left(\begin{pmatrix}(\boxdot signal \wedge \circledast(pressure_1 + pressure_2 < CHigh_T)) \vee \\ (\boxdot \neg signal \wedge \circledast(pressure_1 + pressure_2 > CLow_T))\end{pmatrix} ;\; true\right) \wedge \mathsf{stable}.signal \wedge \mathsf{r\_stable}.signal \tag{27}$$

$$Cont \;\widehat{=}\; \left(\bigvee_{i:\{1,2,3\}} Cont_i\right) \wedge \epsilon \leq \ell \leq \mathsf{pn}.B_C \tag{28}$$

*Modelling the environment.* Due to the decoupling of the controller from the pump, the value of *pump* is no longer set by the controller directly; instead *pump* is an output of the Pump component, which forms part of the controller's environment (see Figure 16). We assume that events of the Pump component take at most $\mathsf{pn}.B_P$ time, and hence, the value of *pump* is formalised by the following interval predicates:

$$CStop \;\widehat{=}\; \Box\left(\boxdot\neg signal \Rightarrow \ominus\overrightarrow{\neg Stopped} \wedge ((\ell < \mathsf{pn}.B_P \wedge \boxdot Stopping) ;\; \boxdot Stopped) \vee \mathsf{stable}.Stopped\right)$$

$$CRun \;\widehat{=}\; \Box\left(\boxdot signal \Rightarrow \ominus\overrightarrow{\neg Running} \wedge ((\ell < \mathsf{pn}.B_P \wedge \boxdot Starting) ;\; \boxdot Running) \vee \mathsf{stable}.Running\right)$$

$$CPump \;\widehat{=}\; CStop \wedge CRun$$

By *CStop*, for any subinterval in which the *signal* stays *false*:

- if the pump was previously not stopped, then there is an interval of length less than $\mathsf{pn}.B_P$ over which the pump is stopping, followed by an interval in which the pump is stopped, and
- if the pump was previously stopped, then it remains stopped throughout the interval.

The behaviour of *CRun* is similar. Note that interval predicate *CPump* takes into account the time taken to physically start/stop pumps, i.e., a pump is not assumed to start/stop instantaneously.

The values $pressure_1$ and $pressure_2$ are under the control of the environment (although $pressure_1$ is also indirectly influenced by the controller, which controls the pump). Therefore the behaviour of the water pressure, which changes $pressure_1$ and $pressure_2$, is given by the following interval predicate:

$$CWater \;\widehat{=}\; \Box(\mathsf{con}.(pressure_1 + pressure_2).B_A \wedge \mathsf{con}.\{pressure_1, pressure_2\}.\{B_P, B_C\})$$

One must also define the accuracy of $pressure_1$ and $pressure_2$ with respect to the intervals of length $\mathsf{pn}.B_A$. The output *pressure* is approximated by the sum $pressure_1 + pressure_2$, therefore we require that $pressure_1 + pressure_2$ is at most $\mathsf{acc}.(pressure_1 + pressure_2).B_A$ for any interval of length at most $\mathsf{pn}.B_A$. Similarly, because $\mathsf{pn}.B_C \leq \mathsf{pn}.B_P$ (sampling takes place at a faster rate than the time taken to execute a pump event), we introduce the third conjunct to restrict the accuracy of $pressure_1$ and $pressure_2$ over an interval in which a pump event takes place.

*Modelling the system.* The concrete system is now straightforward to specify. Changes to values of *signal*, $pressure_1$, $pressure_2$ and *pump* occur in a truly concurrent manner. Therefore, the behaviour of the environment is given by the conjunction of the behaviours of each component and we obtain the following formalisation of the system.

$$CInit\_Rel \;\widehat{=}\; \lambda\,\rho, \tau \bullet (\tau.Stopped \wedge (\tau.pressure_1 + \tau.pressure_2 > High) \wedge \neg\tau.signal)$$

$$CInit\_IR \;\widehat{=}\; \lambda\,\Delta, s, z \bullet \underline{\boxdot CInit\_Rel}.\Delta.s.z \wedge (\mathsf{r\_stable}.signal).\Delta.z$$

$$
\begin{array}{rcl}
CInitSys & \widehat{=} & \lambda\,\rho \bullet CInit \\
CFin\_Rel & \widehat{=} & \lambda\,\tau, \rho \bullet \rho.Pressure = \tau.(pressure_1 + pressure_2) \wedge \rho.Pump = \tau.pump \\
CFin & \widehat{=} & \boxdot CFin\_Rel \\
Z & \widehat{=} & \{pressure_1, pressure_2, signal, pump\} \\
CProg & \widehat{=} & \ominus CInit \wedge Cont^{\infty} \wedge CPump \wedge CWater \\
CSys & \widehat{=} & (CInit\_IR, CProg, CFin)_{W,Z}
\end{array}
$$

*7.5. Verifying data refinement*

For *CSys* to be a refinement of *ASys*, the abstract pump must be stopped (running) whenever the concrete pump is stopped (running). Unlike *ASys* in which the controller directly modifies the *pump* variable, the concrete system has been decoupled to distinguish the (digital) controller from the (physical) pump, and hence, must deal with delays in sending the signal and in turning the pump on/off in response to the signal.

To show $ASys \sqsubseteq CSys$, we apply Theorem 11 using the interval relation *ref* below, where $\sigma \in \Sigma_Y$ and $\tau \in \Sigma_Z$.

$$
\begin{array}{rcl}
pressure\_rel & \widehat{=} & \lambda\,\sigma, \tau \bullet \tau.(pressure_1 + pressure_2) = \sigma.pressure \\
pump\_rel & \widehat{=} & \lambda\,\sigma, \tau \bullet (\tau.Running \Rightarrow \sigma.Running) \wedge (\tau.Stopped \Rightarrow \sigma.Stopped) \wedge \\
 & & \quad \tau.(Stopping \vee Starting) \Rightarrow \sigma.(\neg Stopped \wedge \neg Running) \\
ref & \widehat{=} & \boxdot(pressure\_rel \wedge pump\_rel)
\end{array}
$$

Thus, $pressure\_rel.\sigma.\tau$ holds iff value of *pressure* in $\sigma$ is the sum of the values of $pressure_1$ and $pressure_2$ in $\tau$. State relation $pump\_rel.\sigma.\tau$ holds iff *pump* is *running* (*stopped*) in $\sigma$ whenever *pump* is *running* (*stopped*) in $\tau$ and if the *pump* is *starting* or *stopping* in $\tau$, then *pump* is neither *stopped* nor *running* in $\sigma$. These are used by interval relation *ref*, where $ref.\Delta.y.z$ holds iff both *pressure_rel* and *pump_rel* hold between $y.t$ and $z.t$ for each $t \in \Delta$.

Using Theorem 11, we use forward simulation to prove refinement. The proofs of the initialisation and finalisation conditions are trivial, which leaves us with the following proof obligation.

$$
ref \bullet \left| \begin{smallmatrix} Y \,:\, PumpCont^{\infty} \wedge AEnv \\ \hline Z \,:\, Cont^{\infty} \wedge CWater \wedge CPump \end{smallmatrix} \right| \tag{29}
$$

Using Lemma 14 to decompose (29), then Lemma 13 to distribute projections, we obtain the following proof obligations.

$$
\begin{array}{rcll}
Cont^{\infty} \wedge CWater \wedge CPump & \Vdash_{Y,Z} & ref & (30) \\
ref \wedge (Cont\downarrow)^{\infty} \wedge (CWater\downarrow) \wedge (CPump\downarrow) & \Rrightarrow & AEnv\uparrow & (31) \\
ref \wedge (Cont\downarrow)^{\infty} \wedge (CWater\downarrow) \wedge (CPump\downarrow) & \Rrightarrow & (PumpCont\uparrow)^{\infty} & (32)
\end{array}
$$

*Proof of (30).* This obligation requires that for any interval and stream in which the concrete program executes, there exists an abstract stream related by *ref* over the same interval. We have the following calculation to further decompose (30).

$Cont^{\infty} \wedge CWater \wedge CPump \Vdash_{Y,Z} ref$
$\Leftarrow$ Theorem 15 (Weaken), $\Box c$ splits therefore
$Cont^{\infty} \wedge CWater \wedge CPump \Rightarrow (Cont \wedge CWater \wedge CPump)^{\infty}$
$(Cont \wedge CWater \wedge CPump)^{\infty} \Vdash_{Y,Z} ref$
$\Leftarrow$ Theorem 15 (Iteration)
$Cont \wedge CWater \wedge CPump \Vdash_{Y,Z} ref$
$=$ definition
$\forall \Delta_0, \Delta\colon Intv, y_0\colon Stream_Y, z\colon Stream_Z \bullet$
$\Delta_0 \propto \Delta \wedge ref.\Delta_0.y_0.z \wedge (Cont \wedge CWater \wedge CPump).\Delta.z \Rightarrow \exists y\colon Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref.\Delta.y.z$

This proof obligation is trivial by construction, i.e., one can construct a $y$ such that $y_0.t = y.t$ for each $t \in \Delta_0$ and $(y.t).pressure = (z.t).(pressure_1 + pressure_2)$ for each $t \in \Delta$.

The proofs of (31) and (32) use existing methods described in [21]; and hence their details are deferred to Appendix C. Using these proofs, we obtain the following restrictions on the values of the concrete and abstract threshold values. We define $Acc_i = \mathsf{acc}.pressure_i.B_C$ and $PAcc_i \mathrel{\widehat{=}} \mathsf{acc}.pressure_i.B_P$.

$$High_T \leq CHigh_T + Acc_1 + Acc_2 \tag{33}$$

$$Low_T \geq CLow_T - Acc_1 - Acc_2 \tag{34}$$

$$CLow_T \leq CHigh_T - PAcc_1 - PAcc_2 - 2(Acc_1 + Acc_2) \tag{35}$$

$$\mathsf{pn}.B_P + \mathsf{pn}.B_C \leq \mathsf{pn}.B_A \tag{36}$$

Informally speaking, condition (33) ensures that when $pressure_1 + pressure_2 \geq CHigh_T$ is detected by the first branch of the concrete controller, using the accuracy assumptions defined by $CWater$, one can ensure that $pressure_1 + pressure_2 \geq High_T$ holds in the abstract system. Condition (34) is similar. Condition (35) places a restriction on the levels $CHigh_T$ and $CLow_T$, and (36) defines a relationship between the precision of the concrete and abstract controllers and the pump.

*Discussion.* This example continues our research into the development of a concrete controllers from their abstract specification [21, 21]. Unlike our earlier research, this paper has allowed the state spaces of the abstract and concrete systems to change using data refinement to justify the replacement. This has has been beneficial for the pump example above, because the abstract controller is developed with respect to a system with a *pressure* sensor, which is later refined to a system with two sensors $pressure_1$ and $pressure_2$. Additional restrictions are introduced to ensure the *sampled* values of $pressure_1$ and $pressure_2$ (where the samples take place at different times in the sampling interval) imply the *actual* value of *pressure* at the abstract level. Furthermore, additional pump modes have been introduced later in the development.

As desirable in a refinement-based development, the proof that the system satisfies safety is performed at a higher-level of abstraction and a separate refinement step is performed, which ensures that the properties of the abstract level are preserved. That is, the proof of safety need not be redone at the concrete level.

## 8. Conclusions

Existing frameworks for data refinement model concurrency as an interleaving of the atomic system operations [4, 12]. This allows one to define a system's execution using its set of operations. The traces of a system after initialisation are generated by repeatedly picking an enabled operation from the set non-deterministically then executing the operation. However, such execution models turn out to be inadequate for reasoning about truly concurrent behaviour, e.g., about *transient* properties in the context of real-time systems [21], and can cause difficulties in verifying refinement.

The main contribution of this paper has been the development of an interval-based method for proving data refinement, building on our previous results on our interval-based methodology for reasoning about concurrent [17, 16] and real-time programs [22, 20]. In [17, 16, 22, 20], operation refinement over a single state space is used for step-wise derivation of code from an informal specification. However, these existing techniques cannot be used, for example, to prove refinement between the programs in Figures 1 and 2 or between the abstract and concrete systems in Section 7 because the state spaces at the abstract and concrete levels differ.

To enable decomposition, a simulation rule for proving data refinement has been developed and soundness of the rule with respect to the data refinement definition has been proved. At a technical level, our simulation rule uses refinement relations between *streams over two state spaces* within an interval, generalising traditional refinement relations over two states. Thus, the refinement relation may relate multiple concrete states to multiple abstract states.

Interval-based frameworks are effective for reasoning about true concurrency, and we have demonstrated our approach on a discrete-time concurrent program and a real-time pump controller. Notable in the logic is

that reasoning is performed in a uniform manner across both time domains. Furthermore, we have integrated non-deterministic evaluation operators [30] (to reason about fine-grained atomicity and sampling), and time bands [9, 10] (to reason about behaviours at multiple time granularities). This has in turn enabled many low-level details to be dealt with at a high-level of abstraction.

*Related and future work.* Interval-based methods for reasoning were developed by Moszkowski [39], which inspired the work on Duration Calculus for real time systems [50]. Both logics, however, only support verification of systems as opposed to their refinement-based development.

Over the years, numerous theories for data refinement have been developed. As far as we are aware, two of these are based on interval-based principles similar to ours. A framework that combines interval temporal logic and refinement has been defined by Bäumler et al [7], but their execution model strictly interleaves a component and its environment. As a result, our non-deterministic expression evaluation operators and true concurrency semantics cannot be easily incorporated into their framework. Furthermore, refinement is defined in terms of relations between the abstract and concrete states. Broy presents refinement between streams of different types of timed systems (e.g., discrete vs. real-time systems) [8]. However, these methods do not consider interval-based reasoning; instead the only methods available are those for reasoning at the level of streams.

From a true concurrency perspective, action refinement in a causal process-algebraic setting is studied in [36, 37] and a modal logic for reasoning about true concurrency is given in [5]. Formal approaches to real-time systems refinement has a long history (e.g., [26, 47, 28, 35, 41, 40]), but these frameworks must often use additional constraints to cope with timing-related delays. It is also possible to develop abstract specifications that overly constrain timing properties leading to implementation issues, e.g., timed automata may require properties to be reproved when an implementation suffers from clock perturbations. Such issues have given rise to, so called, robust and perturbed timed automata [28, 3], however, even simple safety properties for such automata are difficult to preserve under refinement [49]. Methods that extend relational semantics with an explicit time variable [35, 34] must introduce additional constraints such as volatile upper/lower bounds, which define the amount of time that an action must be enabled. Clock perturbations and volatility of actions are dealt with naturally by using an interval-based framework [29].

Mechanisation of our interval-based framework remains future work. There are model checking approaches (e.g., PRISM [33], UPPAAL [33]) based on transition systems, as well as as real-time theorem provers (e.g., KeYmaera [45]), based on a specialised hybrid-systems language. However, none of these support data refinement, non-deterministic expression evaluators or time bands. Our logic can be encoded into any existing higher-order theorem prover (e.g., Isabelle [42], KIV [6]), however, we seek to develop further algebraic abstractions of the interval predicate theory and develop an embedding at this algebraic level [18]. Such techniques would enable our proofs to be performed at an even higher level of abstraction.

**Acknowledgements** This work is sponsored by EPSRC Grant EP/J003727/1. We are indebted to our anonymous reviewers from this journal as well as those from REFINE 13 for their comments on the workshop version of this paper.

[1] Abadi, M., Lamport, L., 1995. Conjoining specifications. ACM Trans. Program. Lang. Syst. 17 (3), 507–534.
[2] Allen, J. F., 1983. Maintaining knowledge about temporal intervals. Commun. ACM 26 (11), 832–843.
[3] Alur, R., La Torre, S., Madhusudan, P., 2005. Perturbed timed automata. In: Morari, M., Thiele, L. (Eds.), HSCC. Vol. 3414 of LNCS. Springer, pp. 70–85.
[4] Back, R.-J., von Wright, J., 1994. Trace refinement of action systems. In: Jonsson, B., Parrow, J. (Eds.), CONCUR. Vol. 836 of LNCS. Springer, pp. 367–384.
[5] Baldan, P., Crafa, S., 2010. A logic for true concurrency. In: Gastin, P., Laroussinie, F. (Eds.), CONCUR. Vol. 6269 of LNCS. Springer, pp. 147–161.
[6] Balser, M., Reif, W., Schellhorn, G., Stenzel, K., 1998. KIV 3.0 for provably correct systems. In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (Eds.), Applied Formal Methods - FM-Trends 98. Vol. 1641 of LNCS. Springer, pp. 330–337.
[7] Bäumler, S., Schellhorn, G., Tofan, B., Reif, W., 2011. Proving linearizability with temporal logic. Formal Asp. Comput. 23 (1), 91–112.
[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. Springer-Verlag, pp. 74–88.
[10] Burns, A., Hayes, I. J., 2010. A timeband framework for modelling real-time systems. Real-Time Systems 45 (1-2), 106–142.
[11] de Roever, W. P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J., 2001. Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press.

[12] de Roever, W. P., Engelhardt, K., 1998. Data Refinement: Model-oriented Proof Theories and their Comparison. Vol. 46 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
[13] Derrick, J., Boiten, E. A., 2003. Relational concurrent refinement. Formal Asp. Comput. 15 (2-3), 182–214.
[14] Derrick, J., Boiten, E. A., 2009. Relational concurrent refinement: Automata. Electr. Notes Theor. Comput. Sci. 259, 21–34.
[15] Dongol, B., Derrick, J., 2013. Data refinement for true concurrency. In: Derrick, J., Boiten, E. A., Reeves, S. (Eds.), Refine. Vol. 115 of EPTCS. pp. 15–35.
[16] Dongol, B., Derrick, J., 2013. Simplifying proofs of linearisability using layers of abstraction. ECEASST 66.
[17] Dongol, B., Derrick, J., Hayes, I. J., 2012. Fractional permissions and non-deterministic evaluators in interval temporal logic. ECEASST 53.
[18] Dongol, B., Gomes, V. B. F., Struth, G., 2015. A program construction and verification tool for separation logic. In: MPC. LNCS. Springer, to appear. Tech report version (`http://arxiv.org/abs/1410.4439`).
[19] Dongol, B., Hayes, I. J., 2010. Compositional action system derivation using enforced properties. In: Bolduc, C., Desharnais, J., Ktari, B. (Eds.), MPC. Vol. 6120 of LNCS. Springer, pp. 119–139.
[20] Dongol, B., Hayes, I. J., 2013. Deriving real-time action systems in a sampling logic. Sci. Comput. Program. 78 (11), 2047–2063.
[21] Dongol, B., Hayes, I. J., Derrick, J., 2014. Deriving real-time action systems with multiple time bands using algebraic reasoning. Sci. Comput. Program. 85, 137–165.
[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 LNCS. Springer, pp. 50–65.
[23] Dongol, B., Travkin, O., Derrick, J., Wehrheim, H., 2013. A high-level semantics for program execution under total store order memory. In: Liu, Z., Woodcock, J., Zhu, H. (Eds.), ICTAC. Vol. 8049 of LNCS. Springer, pp. 177–194.
[24] Feijen, W. H. J., van Gasteren, A. J. M., 1999. On a Method of Multiprogramming. Springer Verlag.
[25] Fidge, C. J., 1993. Real-time refinement. In: Woodcock, J., Larsen, P. G. (Eds.), FME. Vol. 670 of Lecture Notes in Computer Science. Springer, pp. 314–331.
[26] Fidge, C. J., Utting, M., Kearney, P., Hayes, I. J., 1996. Integrating real-time scheduling theory and program refinement. In: Gaudel, M.-C., Woodcock, J. (Eds.), FME. Vol. 1051 of LNCS. Springer, pp. 327–346.
[27] Garay, P., 1996. Pump Application Desk Book, 3ed. Fairmont Press.
[28] 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.
[29] Gurovic, D., Fengler, W., Nutzel, J., 2000. Development of real-time system specifications through the refinement of duration interval petri nets. In: Systems, Man, and Cybernetics, 2000 IEEE International Conference on. Vol. 4. pp. 3098–3103.
[30] Hayes, I. J., Burns, A., Dongol, B., Jones, C. B., 2013. Comparing degrees of non-determinism in expression evaluation. The Computer Journal.
[31] Hehner, E. C. R., 1990. A practical theory of programming. Sci. Comput. Program. 14 (2-3), 133–158.
[32] Jones, C. B., Pierce, K. G., 2008. Splitting atoms with rely/guarantee conditions coupled with data reification. In: E., Butler, M. J., Bowen, J. P., Boca, P. (Eds.), ABZ. Vol. 5238 of Lecture Notes in Computer Science. Springer, pp. 360–377.
[33] Kwiatkowska, M. Z., Norman, G., Parker, D., 2011. PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (Eds.), CAV. Vol. 6806 of LNCS. Springer, pp. 585–591.
[34] Lamport, L., 2002. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[35] Liu, Z., Joseph, M., 2001. Verification, refinement and scheduling of real-time programs. Theoretical Computer Science 253 (1), 119 – 152.
[36] Majster-Cederbaum, M. E., Wu, J., 2001. Action refinement for true concurrent real time. In: ICECCS. IEEE Computer Society, pp. 58–68.
[37] Majster-Cederbaum, M. E., Wu, J., 2003. Towards action refinement for true concurrent real time. Acta Inf. 39 (8), 531–577.
[38] Morgan, C., 1994. Programming from specifications (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.
[39] Moszkowski, B. C., 2000. A complete axiomatization of interval temporal logic with infinite time. In: LICS. IEEE Computer Society, pp. 241–252.
[40] Murphy, D., Pitt, D. H., 1992. Real-timed concurrent refineable behaviours. In: Vytopil, J. (Ed.), FTRTFTS. Vol. 571 of LNCS. Springer, pp. 529–545.
[41] Olderog, E.-R., Dierks, H., 2008. Real-time systems - formal specification and automatic verification. Cambridge University Press.
[42] Paulson, L. C., 1994. Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). Vol. 828 of LNCS. Springer.
[43] Pedersen, G. K. M., Yang, Z., 2008. Efficiency optimization of a multi-pump booster system. In: Ryan, C., Keijzer, M. (Eds.), GECCO. ACM, pp. 1611–1618.
[44] Pelavin, R. N., Allen, J. F., 1987. A model for concurrent actions having temporal extent. In: Forbus, K. D., Shrobe, H. E. (Eds.), AAAI. Morgan Kaufmann, pp. 246–250.
[45] Platzer, A., Quesel, J.-D., 2008. Keymaera: A hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (Eds.), IJCAR. Vol. 5195 of LNCS. Springer, pp. 171–178.
[46] Schellhorn, G., Tofan, B., Ernst, G., Reif, W., 2011. Interleaved programs and rely-guarantee reasoning with ITL. TIME 0, 99–106.

[47] Scholefield, D., Zedan, H. S. M., He, J., 1993. Real-time refinement: Semantics and application. In: Borzyszkowski, A. M., Sokolowski, S. (Eds.), MFCS. Vol. 711 of LNCS. Springer, pp. 693–702.
[48] Tosic, P. T., Agha, G., 2004. Concurrency vs. sequential interleavings in 1-d threshold cellular automata. In: IPDPS. IEEE Computer Society, pp. 179–186.
[49] Wulf, M., Doyen, L., Markey, N., Raskin, J.-F., December 2008. Robust safety of timed automata. Form. Methods Syst. Des. 33, 45–84.
[50] Zhou, C., Hansen, M. R., 2004. Duration Calculus: A Formal Approach to Real-Time Systems. EATCS: Monographs in Theoretical Computer Science. Springer.

## Appendix A. Proofs of lemmas

**Lemma 8.** Provided that $id.\sigma.\tau \mathrel{\widehat{=}} \sigma = \tau$ and both $ref_1$ and $ref_2$ are externally independent:

$$\boxdot id \bullet \left|\frac{X\ :\ g}{X\ :\ g}\right| \qquad \text{(Reflexivity)}$$

$$ref_1 \bullet \left|\frac{X\ :\ f}{Y\ :\ g}\right| \wedge ref_2 \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right| \ \Rightarrow\ (ref_1 \circ ref_2) \bullet \left|\frac{X\ :\ f}{Z\ :\ h}\right| \qquad \text{(Transitivity)}$$

Proof. The proof of (Reflexivity) is trivial. We prove (Transitivity) as follows, where we assume that $\Delta_0, \Delta \in Intv$ such that $\Delta_0 \propto \Delta$, $x_0 \in Stream_X$ and $z \in Stream_Z$ are arbitrarily chosen. We have:

$$\begin{array}{ll} & (ref_1 \circ ref_2).\Delta_0.x_0.z \wedge h.\Delta.z \\ = & \quad \text{definition of } \circ \text{ and logic} \\ & \exists y_0 \colon Stream_Y \bullet ref_1.\Delta_0.x_0.y_0 \wedge ref_2.\Delta_0.y_0.z \wedge h.\Delta.z \end{array}$$

For an arbitrarily chosen $y_0 \in Stream_Y$, we prove the following.

$$\begin{array}{ll} & ref_1.\Delta_0.x_0.y_0 \wedge ref_2.\Delta_0.y_0.z \wedge h.\Delta.z \\ \Rightarrow & \quad \text{assumption } ref_2 \bullet \left|\frac{Y\ :\ g}{Z\ :\ h}\right| \\ & ref_1.\Delta_0.x_0.y_0 \wedge \exists y \colon Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref_2.\Delta.y.z \wedge g.\Delta.y \\ = & \quad \text{assume } y \text{ free in } ref_1.\Delta_0.x_0.y_0 \\ & \exists y \colon Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge ref_2.\Delta.y.z \wedge ref_1.\Delta_0.x_0.y_0 \wedge g.\Delta.y \\ = & \quad \text{construct a new stream } y_1 \in Stream_Y \text{ that matches } y_0 \text{ over } \Delta_0 \text{ and } y \text{ over } \Delta \\ & \exists y, y_1 \colon Stream_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge (y \overset{\Delta}{=\!=} y_1) \wedge ref_2.\Delta.y.z \wedge ref_1.\Delta_0.x_0.y_0 \wedge g.\Delta.y \\ \Rightarrow & \quad \text{use } y_0 \overset{\Delta_0}{=\!=} y_1 \text{ and } y \overset{\Delta}{=\!=} y_1 \\ & \exists y_1 \colon Stream_Y \bullet ref_2.\Delta.y_1.z \wedge ref_1.\Delta_0.x_0.y_1 \wedge g.\Delta.y_1 \\ \Rightarrow & \quad \text{logic, assumption } ref_1 \bullet \left|\frac{X\ :\ f}{Y\ :\ g}\right| \\ & \exists y_1 \colon Stream_Y, x \colon Stream_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge ref_2.\Delta.y_1.z \wedge ref_1.\Delta.x.y_1 \wedge f.\Delta.x \\ \Rightarrow & \quad \text{definition of } \circ \\ & \exists x \colon Stream_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge (ref_1 \circ ref_2).\Delta.x.z \wedge f.\Delta.x \end{array}$$

Before proving the next lemma, we introduce a *partition* of an interval, which is a possibly infinite sequence of adjoining intervals whose union equals $\Delta$. Given that $\mathrm{seq}.T$ denotes sequences of type $T$, the set of all partitions of an interval $\Delta$ is given by

$$partition.\Delta \ \mathrel{\widehat{=}}\ \left\{ \delta \in \mathrm{seq}.Intv \ \middle|\ (\Delta = \bigcup \mathrm{ran}.\delta) \wedge \forall i, j \colon \mathrm{dom}.\delta \bullet i < j \Rightarrow \delta.i < \delta.j \ \right\}$$

Note that if $\delta \in partition.\Delta$, then by definition $\delta$ is countable, and hence it disallows Zeno-like behaviour.

Two useful properties of intervals for modularity are splits and joins, which enable decomposing proof obligations over sequential composition and iteration constructs. We say that $g$ *splits* iff $g \Rrightarrow (g\ ;\ g)$ and

*g joins* iff $g^{\omega+} \Rrightarrow g$. Note that if $g$ splits, then $g \Rrightarrow g^{\omega}$ [22]. For example, $g$ in Example 1 both splits and joins, and $g_0$ in Example 7 joins but does not split. Interval predicate $\ell < 10$ splits but does not join.

**Lemma 15.** If $Y, Z \subseteq \mathit{Var}$, $h, h_1, h_2 \in \mathit{IntvPred}_Z$, $\epsilon \in \mathcal{T}$ is a constant, and $\mathit{ref} \in \mathit{IntvRel}_{Y,Z}$, then each of the following holds, provided *ref* is externally independent.

$$
\begin{array}{rcll}
h_1 \Vdash_{Y,Z} \mathit{ref} \wedge h_2 \Vdash_{Y,Z} \mathit{ref} & \Rightarrow & (h_1 \,;\, h_2) \Vdash_{Y,Z} \mathit{ref} & \text{(Sequential)} \\
h \Vdash_{Y,Z} \mathit{ref} & \Rightarrow & (h \wedge \epsilon \leq \ell)^{\omega+} \Vdash_{Y,Z} \mathit{ref} & \text{(Iteration)} \\
(h \Vdash_{Y,Z} \mathit{ref}_1) \vee (h \Vdash_{Y,Z} \mathit{ref}_2) & \Rightarrow & h \Vdash_{Y,Z} (\mathit{ref}_1 \vee \mathit{ref}_2) & \text{(Non-deterministic choice)} \\
(h_2 \Vdash_{Y,Z} \mathit{ref}) \wedge (h_1 \Rrightarrow h_2) & \Rightarrow & h_1 \Vdash_{Y,Z} \mathit{ref} & \text{(Weaken)}
\end{array}
$$

Proof (Sequential). For an arbitrarily chosen $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $y_0 \in \mathit{Stream}_Y$ and $z \in \mathit{Stream}_Z$, we have the following calculation.

$$
\begin{array}{ll}
 & \mathit{ref}.\Delta_0.y_0.z \wedge (h_1 \,;\, h_2).\Delta.z \\
= & \quad \text{definition of ';', logic} \\
 & \exists \Delta_1, \Delta_2 \colon \mathit{Intv} \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge \mathit{ref}.\Delta_0.y_0.z \wedge h_1.\Delta_1.z \wedge h_2.\Delta_2.z \\
\Rightarrow & \quad \Delta_0 \propto \Delta \text{ and } \Delta_1 \text{ a prefix of } \Delta, \text{ therefore } \Delta_0 \propto \Delta_1 \\
 & \quad \text{assumption } h_1 \Vdash_{Y,Z} \mathit{ref} \\
 & \exists \Delta_1, \Delta_2 \colon \mathit{Intv} \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge (\exists y_1 \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z) \wedge h_2.\Delta_2.z \\
= & \quad \text{logic} \\
 & \exists \Delta_1, \Delta_2 \colon \mathit{Intv}, y_1 \colon \mathit{Stream}_Y \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z \wedge h_2.\Delta_2.z \\
= & \quad \Delta_1 \propto \Delta_2 \text{ and assumption } h_2 \Vdash_{Y,Z} \mathit{ref} \\
 & \exists \Delta_1, \Delta_2 \colon \mathit{Intv}, y_1, y_2 \colon \mathit{Stream}_Y \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge (y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}.\Delta_1.y_1.z \wedge (y_1 \overset{\Delta_1}{=\!=} y_2) \wedge \mathit{ref}.\Delta_2.y_2.z \\
\Rightarrow & \quad \text{pick } y_3 \text{ such that } y_1 \overset{\Delta_0 \cup \Delta_1}{=\!=} y_3 \text{ and } y_2 \overset{\Delta_2}{=\!=} y_3 \\
 & \exists \Delta_1, \Delta_2 \colon \mathit{Intv}, y_3 \colon \mathit{Stream}_Y \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge \mathit{ref}.\Delta_1.y_3.z \wedge \mathit{ref}.\Delta_2.y_3.z \\
= & \quad \text{definition} \\
 & \exists y_3 \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge (\mathit{ref} \,;\, \mathit{ref}).\Delta.y_3.z \\
\Rightarrow & \quad \mathit{ref} \text{ joins} \\
 & \exists y_3 \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y_3) \wedge \mathit{ref}.\Delta.y_3.z
\end{array}
$$

Proof (Iteration). For an arbitrarily chosen $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $y_0 \in \Sigma_Y$ and $z \in \mathit{Stream}_Z$, we have the following calculation.

$$
\begin{array}{ll}
 & \mathit{ref}.\Delta_0.y_0.z \wedge (h \wedge \epsilon \leq \ell)^{\omega+}.\Delta.z \\
= & \quad \text{definition of '}\omega+\text{', logic} \\
 & \exists \delta \colon \mathit{partition}.\Delta \bullet \mathrm{dom}.\delta \neq \varnothing \wedge \mathit{ref}.\Delta_0.y_0.z \wedge \forall i \colon \mathrm{dom}.\delta \bullet h.(\delta.i).z \\
= & \quad \text{logic} \\
 & \exists \delta \colon \mathit{partition}.\Delta \bullet \mathit{ref}.\Delta_0.y_0.z \wedge h.(\delta.0).z \wedge \forall i \colon \mathrm{dom}.\delta \backslash \{0\} \bullet h.(\delta.i).z \\
\Rightarrow & \quad \text{assumption } h \Vdash_{Y,Z} \mathit{ref} \text{ and } \Delta_0 \propto \Delta, \text{ therefore } \Delta_0 \propto \delta.0 \\
 & \exists \delta \colon \mathit{partition}.\Delta, y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}.(\delta.0).y.z \wedge \forall i \colon \mathrm{dom}.\delta \backslash \{0\} \bullet h.(\delta.i).z \\
\Rightarrow & \quad \text{logic, assuming } h \Vdash_{Y,Z} \mathit{ref} \\
 & \exists \delta \colon \mathit{partition}.\Delta, y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}.(\delta.0).y.z \wedge \\
 & \qquad\qquad \forall i \colon \mathrm{dom}.\delta \backslash \{0\}, \exists y_i \colon \mathit{Stream}_Y \bullet (y_{i-1} \overset{\delta.i}{=\!=} y_i) \wedge \mathit{ref}.(\delta.i).y_i.z \\
\Rightarrow & \quad \text{pick } y' \text{ such that } y_0 \overset{\Delta_0}{=\!=} y' \text{ and for all } i \in \mathrm{dom}.\delta,\ y_i \overset{\delta.i}{=\!=} y' \\
 & \exists \delta \colon \mathit{partition}.\Delta, y' \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y') \wedge \mathit{ref}.(\delta.0).y'.z \wedge \forall i \colon \mathrm{dom}.\delta \backslash \{0\} \bullet \mathit{ref}.(\delta.i).y'.z \\
= & \quad \text{logic}
\end{array}
$$

$$
\begin{array}{ll}
& \exists y' \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y') \wedge \exists \delta \colon \mathit{partition}.\Delta \bullet \forall i \colon \mathrm{dom}.\delta \bullet \mathit{ref}.(\delta.i).y'.z \\
= & \quad \text{logic} \\
& \exists y' \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y') \wedge (\mathit{ref}^+).\Delta.y'.z \\
\Rightarrow & \quad \mathit{ref} \text{ joins} \\
& \exists y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y') \wedge \mathit{ref}.\Delta.y'.z
\end{array}
$$

Proof (Non-deterministic choice).

$$
\begin{array}{ll}
& (\mathit{ref}_1 \vee \mathit{ref}_2).\Delta_0.y_0.z \wedge h.\Delta.z \\
= & \quad \text{logic} \\
& (\mathit{ref}_1.\Delta_0.y_0.z \wedge h.\Delta.z) \vee (\mathit{ref}_2.\Delta_0.y_0.z \wedge h.\Delta.z) \\
\Rightarrow & \quad \text{assumption } (h \Vdash_{Y,Z} \mathit{ref}_1) \vee (h \Vdash_{Y,Z} \mathit{ref}_2), \text{ logic} \\
& \exists y_1, y_2 \colon \mathit{Stream}_Y \bullet ((y_0 \overset{\Delta_0}{=\!=} y_1) \wedge \mathit{ref}_1.\Delta.y_1.z) \vee ((y_0 \overset{\Delta_0}{=\!=} y_2) \wedge \mathit{ref}_2.\Delta.y_2.z) \\
\Rightarrow & \quad \text{logic} \\
& \exists y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge (\mathit{ref}_1 \vee \mathit{ref}_2).\Delta.y.z
\end{array}
$$

Proof (Weaken). For an arbitrarily chosen $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, $y_0 \in \Sigma_Y$ and $z \in \mathit{Stream}_Z$, we have the following calculation.

$$
\begin{array}{ll}
& \mathit{ref}.\Delta_0.y_0.z \wedge h_1.\Delta.z \\
\Rightarrow & \quad \text{assumption } h_1 \Rrightarrow h_2 \\
& \mathit{ref}.\Delta_0.y_0.z \wedge h_2.\Delta.z \\
\Rightarrow & \quad \text{assumption } h_2 \Vdash_{Y,Z} \mathit{ref} \\
& \exists y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge \mathit{ref}.\Delta.y.z
\end{array}
$$

□

**Lemma 16.** Suppose $W, X, Y, Z \subseteq \mathit{Var}$ such that $Y \cap Z = \varnothing$, $W \cup X = Y$ and $W \cap X = \varnothing$. If $h_1, h_2 \in \mathit{IntvPred}_Z$, $\mathit{ref}_W \in \mathit{IntvRel}_{W,Z}$, $\mathit{ref}_X \in \mathit{IntvRel}_{X,Z}$, and $\star \in \{\wedge, \vee\}$, then

$$
(h_1 \Vdash_{W,Z} \mathit{ref}_W) \wedge (h_2 \Vdash_{X,Z} \mathit{ref}_X) \quad \Rightarrow \quad (h_1 \wedge h_2) \Vdash_{Y,Z} (\mathit{ref}_W \star \mathit{ref}_X)
$$

Proof. Because $W \cup X = Y$ and $W \cap X = Y$, for any $y_0 \in \mathit{Stream}_Y$, we have that $y_0 = w_0 \Cup x_0$ for some $w_0 \in \mathit{Stream}_W$, $x_0 \in \mathit{Stream}_X$. Then for any $z \in \mathit{Stream}_Z$, $\Delta_0, \Delta \in \mathit{Intv}$ such that $\Delta_0 \propto \Delta$, we have the following calculation:

$$
\begin{array}{ll}
& (\mathit{ref}_W \star \mathit{ref}_X).\Delta_0.y_0.z \wedge (h_1 \wedge h_2).\Delta.z \\
\Rightarrow & \quad \text{assumption } y_0 = w_0 \Cup x_0 \\
& (\mathit{ref}_W.\Delta_0.w_0.z \star \mathit{ref}_X.\Delta_0.x_0.z) \wedge (h_1 \wedge h_2).\Delta.z \\
\Rightarrow & \quad \wedge \text{ distributes over } \star, \text{ logic} \\
& (\mathit{ref}_W.\Delta_0.w_0.z \wedge h_1.\Delta.z) \star (\mathit{ref}_X.\Delta_0.x_0.z \wedge h_2.\Delta.z) \\
\Rightarrow & \quad \text{assumption } (h_1 \Vdash_{W,Z} \mathit{ref}_W) \wedge (h_2 \Vdash_{X,Z} \mathit{ref}_X) \\
& (\exists w \colon \mathit{Stream}_W \bullet (w_0 \overset{\Delta_0}{=\!=} w) \wedge \mathit{ref}_W.\Delta.w.z) \star (\exists x \colon \mathit{Stream}_X \bullet (x_0 \overset{\Delta_0}{=\!=} x) \wedge \mathit{ref}_X.\Delta.x.z) \\
= & \quad \text{logic, assumption } W \cap X = \varnothing \\
& \exists w \colon \mathit{Stream}_W, x \colon \mathit{Stream}_X \bullet (w_0 \Cup x_0 \overset{\Delta_0}{=\!=} w \Cup x) \wedge (\mathit{ref}_W.\Delta.w.z \star \mathit{ref}_X.\Delta.x.z) \\
= & \quad \text{logic, assumption } y_0 = w_0 \Cup x_0 \\
& \exists w \colon \mathit{Stream}_W, x \colon \mathit{Stream}_X \bullet (y_0 \overset{\Delta_0}{=\!=} w \Cup x) \wedge (\mathit{ref}_W \star \mathit{ref}_X).\Delta.(w \Cup x).z \\
= & \quad W \cup X = Y \text{ and } W \cap X = \varnothing \\
& \exists y \colon \mathit{Stream}_Y \bullet (y_0 \overset{\Delta_0}{=\!=} y) \wedge (\mathit{ref}_W \star \mathit{ref}_X).\Delta.y.z
\end{array}
$$

**Lemma 13** Suppose $g, g_1, g_2$ are interval predicates, $\epsilon \in \mathcal{T}$ is a constant, $\odot \in \{;\ , \vee, \wedge, \Rightarrow, =\}$ and $\updownarrow \in \{\uparrow, \downarrow\}$. Then each of the following holds:

$$(\neg g)\updownarrow \;=\; \neg(g\updownarrow) \tag{A.1}$$
$$(g_1 \odot g_2)\updownarrow \;=\; (g_1\updownarrow) \odot (g_2\updownarrow) \tag{A.2}$$
$$((g \wedge \epsilon \leq \ell)\updownarrow)^\omega \;=\; (g \wedge \epsilon \leq \ell)^\omega \updownarrow \tag{A.3}$$

Proof (A.1). Suppose $\Delta$ is an interval and $y$, $z$ are streams. The proof for $\uparrow$ is as follows. The proof for $\downarrow$ is similar.

$$\begin{array}{ll} & ((\neg g)\uparrow).\Delta.y.z \\ = & \quad \text{definition of } \uparrow \\ & (\neg g).\Delta.y \\ = & \quad \text{lifting over interval predicates} \\ & \neg(g.\Delta.y) \\ = & \quad \text{definition of } \uparrow \\ & \neg((g\uparrow).\Delta.y.z) \end{array}$$

Proof (A.2). We present the proof for $;$ and $\uparrow$. The proofs of the other cases are similar. Suppose $\Delta$ is an interval and $y$, $z$ are streams.

$$\begin{array}{ll} & ((g_1\ ;\ g_2)\uparrow).\Delta.y.z \\ = & \quad \text{definition of } \uparrow \\ & (g_1\ ;\ g_2).\Delta.y \\ = & \quad \text{definition of } ; \\ & (\exists \Delta_1, \Delta_2 \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge g_1.\Delta_1.y \wedge g_2.\Delta_2.y) \vee (\mathsf{infinite} \wedge g_1).\Delta.y \\ = & \quad \text{definition of } \uparrow \\ & (\exists \Delta_1, \Delta_2 \bullet \frac{\Delta_1 \Delta_2}{\Delta} \wedge (g_1\uparrow).\Delta_1.y.z \wedge (g_2\uparrow).\Delta_2.y.z) \vee (\mathsf{infinite} \wedge (g_1\uparrow)).\Delta.y.z \\ = & \quad \text{definition of } ; \\ & ((g_1\uparrow)\ ;\ (g_2\uparrow)).\Delta.y.z \end{array}$$

Proof (A.3). Suppose $\Delta$ is an interval and $y$, $z$ are streams. As above, we verify the property for $\uparrow$; the proof for $\downarrow$ is similar.

$$\begin{array}{l} ((g \wedge \epsilon \leq \ell)\uparrow)^\omega.\Delta.y.z \\ = \exists \delta{:}\, \mathit{partition}.\Delta \bullet \forall i{:}\, \mathrm{dom}.\delta \bullet ((g \wedge \epsilon \leq \ell)\uparrow).(\delta.i).y.z \\ = \exists \delta{:}\, \mathit{partition}.\Delta \bullet \forall i{:}\, \mathrm{dom}.\delta \bullet (g \wedge \epsilon \leq \ell).(\delta.i).y \\ = ((g \wedge \epsilon \leq \ell)^\omega).\Delta.y \\ = ((g \wedge \epsilon \leq \ell)^\omega\uparrow).\Delta.y.z \end{array}$$

## Appendix B. Proofs for the abstract pump

This appendix provides proofs and calculation to prove properties of the abstract pump. The proof uses the following property. The operator '$\circledast$' cannot normally be distributed within logical conjunction, i.e., for state predicates $c_1$ and $c_2$, although $\circledast(c_1 \wedge c_2) \Rrightarrow \circledast c_1 \wedge \circledast c_2$ holds, the implication does not necessarily hold in the other direction. A useful exception is when the variables of $c_1$ and $c_2$ are disjoint.

**Lemma 22.** *If $v$ is a variable and $c_1$ and $c_2$ are state predicates such that $vars.c_1 \cap vars.c_2 = \varnothing$, then $\circledast(c_1 \wedge c_2) \equiv \circledast c_1 \wedge \circledast c_2$.*

*Calculation 1.*

$$
\begin{array}{ll}
 & \circledast(\neg \mathit{Stopped} \wedge (\mathit{pressure} \geq \mathit{High}_T)) \wedge \mathsf{stable}.\mathit{pump} \\
\equiv & \quad \text{definition of } \mathit{Stopped}\text{, Lemma 22} \\
 & \circledast(\mathit{pump} \neq \mathit{stopped}) \wedge \circledast(\mathit{pressure} \geq \mathit{High}_T) \wedge \mathsf{stable}.\mathit{pump} \\
\equiv & \quad \text{Lemma 19} \\
 & \diamondsuit(\mathit{pump} \neq \mathit{stopped}) \wedge \diamondsuit(\mathit{pressure} \geq \mathit{High}_T) \wedge \mathsf{stable}.\mathit{pump} \\
\equiv & \quad \text{using } \mathsf{stable}.\mathit{pump}\text{, definition of } \mathit{Stopped} \\
 & \boxdot\neg \mathit{Stopped} \wedge \diamondsuit(\mathit{pressure} \geq \mathit{High}_T)
\end{array}
$$

□

*Proof of (24).* These proofs require the following constraint on the values of $\mathit{Low}$, $\mathit{High}$, $\mathit{Low}_T$ and $\mathit{High}_T$.

$$\mathit{Low} + \mathsf{acc}.\mathit{pressure}.B_A \quad \leq \quad \mathit{Low}_T \quad < \quad \mathit{High}_T \quad \leq \quad \mathit{High} - \mathsf{acc}.\mathit{pressure}.B_A \tag{B.1}$$

Thus, the low threshold value of $\mathit{Low}_T$ must be at least $\mathit{Low}+\mathsf{acc}.\mathit{pressure}.B_A$, the high threshold value $\mathit{High}_T$ at most $\mathit{High} - \mathsf{acc}.\mathit{pressure}.B_A$, and the high threshold value must be above $\mathit{Low}_T$. These restrictions allow the controller enough time to react to a high/low values of *pressure* and turn the pump on/off accordingly, i.e., before the *pressure* goes below *Low* and above *High*.

Expressed as an interval predicate, the behaviour of a real-time controller is often of the form $\ominus \mathit{init} \wedge (g_1 \vee g_2 \vee \cdots \vee g_n)^\infty$, where *init* formalises the initialisation and each $g_i$ models the behaviour of a single branch of the main **if** statement [21, 19]. We obtain the following lemma that allows one to consider maximal iterations of each branch $g_j$.

**Lemma 23 ([21]).** *Suppose $g = \bigvee_{i:0..n} g_i$, where each $g_i$ is an interval predicate, init an interval predicate and*

$$
\begin{array}{rcl}
\mathit{not\_jth.g.j} & \widehat{=} & (\bigvee_{i:0..j-1} g_i) \vee (\bigvee_{i:j+1..n} g_i) \\
\mathit{iter.g.j} & \widehat{=} & \ominus(\mathit{init} \vee \mathit{not\_jth.g.j}) \wedge g_j^{\omega+}
\end{array}
$$

*where not_jth.g.j consists of all disjuncts of g except $g_j$ and iter.g.j denotes an iteration of $g_j$ where a previous interval either satisfies the initialisation init or a disjunct from not_jth.g.j. Then,*

$$\ominus \mathit{init} \wedge g^\infty \quad \Rrightarrow \quad (\bigvee_{j:0..n} \mathit{iter.g.j})^\infty$$

Thus, by Lemma 23, if a controller behaves as $\ominus \mathit{init} \wedge g^\infty$ over an interval $\Delta$, then

$$(\bigvee_{j:0..n} \mathit{iter.g.j})^\infty$$

holds over $\Delta$. To enable simplification of the behaviour of the abstract controller using Lemma 23, we define the following interval predicates.

$$
\begin{array}{rcl}
\mathit{PC}_i & \widehat{=} & \mathit{AEnv} \wedge \mathit{PumpCont}_i \\
\mathit{PC} & \widehat{=} & \bigvee_{i:\{1,2,3\}} \mathit{PC}_i
\end{array}
$$

Then, we obtain the following calculation, which proves that the behaviour of the environment *AEnv* can be moved within each iteration.

$$
\begin{array}{ll}
 & \mathit{AProg} \\
\equiv & \quad \text{definition} \\
 & \mathit{AEnv} \wedge \ominus \mathit{AInit} \wedge \mathit{PumpCont}^\infty \\
\Rrightarrow & \quad \text{logic, } \mathit{AEnv} \text{ splits} \\
 & \ominus \mathit{AInit} \wedge (\mathit{AEnv} \wedge \mathit{PumpCont})^\infty \\
\equiv & \quad \mathit{AEnv} \wedge \mathit{PumpCont} \equiv \mathit{PC} \text{ and } {}^\infty \text{ is monotonic} \\
 & \ominus \mathit{AInit} \wedge \mathit{PC}^\infty \\
\Rrightarrow & \quad \text{Lemma 23} \\
 & \ominus \mathit{AInit} \wedge (\bigvee_{i:\{1,2,3\}} \mathit{iter.PC.i})^\infty
\end{array}
$$

The proof then proceeds as follows:

$$
\begin{array}{ll}
 & (24) \\
\Leftarrow & \quad \text{calculation above, logic} \\
 & (\bigvee_{i:\{1,2,3\}} iter.PC.i)^{\infty} \Rrightarrow (21) \wedge (22) \\
\Leftarrow & \quad \boxdot c \wedge \boxdot d \equiv \boxdot(c \wedge d) \text{ and } \boxdot c \text{ joins} \\
 & (\bigvee_{i:\{1,2,3\}} iter.PC.i)^{\infty} \Rrightarrow \\
 & (\boxdot((pressure \geq High \Rightarrow Stopped) \wedge (pressure \leq Low \Rightarrow Running)))^{\infty} \\
\Leftarrow & \quad {}^{\infty} \text{ is monotonic} \\
 & (\bigvee_{i:\{1,2,3\}} iter.PC.i) \Rrightarrow \\
 & \boxdot((pressure \geq High \Rightarrow Stopped) \wedge (pressure \leq Low \Rightarrow Running)) \\
= & \quad \text{logic, } \boxdot c \wedge \boxdot d \equiv \boxdot(c \wedge d) \\
 & \bigwedge_{i:\{1,2,3\}} \big(iter.PC.i \Rrightarrow \boxdot(pressure \geq High \Rightarrow Stopped) \wedge \boxdot(pressure \leq Low \Rightarrow Running)\big) \quad (Req_1)
\end{array}
$$

We now perform case analysis on $i$. We present the proof of $i = 1$ below. We have the following calculation for conjunct $\boxdot(pressure \leq Low \Rightarrow Running)$ in proof obligation $(Req_1)$ above.

$$
\begin{array}{ll}
 & iter.PC.1 \Rrightarrow \boxdot(pressure \leq Low \Rightarrow Running) \\
\Leftarrow & \quad \text{definition of } iter.PC.1, \diamondsuit c \text{ joins} \\
 & (AEnv \wedge \diamondsuit(pressure \geq High_T) \wedge \ell \leq \mathsf{pn}.B_A)^{\omega+} \Rrightarrow \boxdot(pressure \leq Low \Rightarrow Running) \\
\Leftarrow & \quad \text{Lemma 21, } AEnv \text{ and } \ell \leq \mathsf{pn}.B_A \\
 & (\boxdot(pressure \geq High))^{\omega+} \Rrightarrow \boxdot(pressure \leq Low \Rightarrow Running) \\
\Leftarrow & \quad \boxdot c \text{ joins} \\
 & \boxdot(pressure \geq High) \Rrightarrow \boxdot(pressure \leq Low \Rightarrow Running) \\
= & \quad \text{by (B.1) } Low < High, \text{ logic} \\
 & true
\end{array}
$$

To prove conjunct $\boxdot(pressure \geq High \Rightarrow Stopped)$ in $(Req_1)$, we first perform the following calculation, which allows one to perform case analysis on the interval predicate that held prior to execution of $PC_1$. In particular, we have:

$$
\begin{array}{ll}
 & iter.PC.1 \\
\equiv & \quad \text{expand } iter.PC.1 \\
 & (\ominus AInit \wedge PC_1^{\omega+}) \vee \bigvee_{j\in\{1,2,3\}} (\ominus PC_j \wedge PC_1^{\omega+}) \\
\equiv & \quad PC_1 \text{ falsifies its own guard} \\
 & (\ominus AInit \wedge PC_1) \vee \bigvee_{j\in\{1,2,3\}} (\ominus PC_j \wedge PC_1)
\end{array}
$$

Case $AInit$:

$$
\begin{array}{ll}
 & \ominus AInit \wedge PC_1 \\
\Rrightarrow & \quad AInit \Rrightarrow \underline{\boxdot Stopped} \wedge \mathsf{r\_stable}.pump \\
 & \overleftarrow{Stopped} \wedge PC_1 \\
\Rrightarrow & \quad PC_1 \Rrightarrow \overleftarrow{\neg Stopped} \\
 & false
\end{array}
$$

Case $j = 1$:
This case reduces to *false* because $PC_1$ falsifies its own guard, therefore $\ominus PC_1 \wedge PC_1$ is impossible.

Case $j = 2$:

$$
\begin{array}{ll}
 & \ominus PC_2 \wedge PC_1 \\
\Rrightarrow & \quad \text{definition of } PC_2 \\
 & \ominus \left( \begin{array}{l} \diamondsuit(\mathit{pressure} \leq \mathit{Low}_T) \wedge \mathit{AEnv} \wedge \\ \ell \leq \mathsf{pn}.B_A \end{array} \right) \wedge PC_1 \\
\Rrightarrow & \quad \text{Lemma 21, } \mathit{AEnv} \text{ and } \ell \leq \mathsf{pn}.B_A \\
 & \ominus \boxdot(\mathit{pressure} \leq \mathit{Low} + \mathsf{acc}.\mathit{pressure}.B_A) \wedge PC_1 \\
\Lleftarrow & \quad \mathit{pressure} \text{ is continuous} \\
 & \overleftarrow{\mathit{pressure} \leq \mathit{Low} + \mathsf{acc}.\mathit{pressure}.B_A} \wedge PC_1 \\
\Rrightarrow & \quad PC_1 \Rrightarrow \mathit{AEnv} \wedge \ell \leq \mathsf{pn}.B_A \\
 & \boxdot(\mathit{pressure} \leq \mathit{Low} + 2 \times \mathsf{acc}.\mathit{pressure}.B_A) \\
\Rrightarrow & \quad \mathit{Low} + \mathsf{acc}.\mathit{pressure}.B_A < \mathit{High} - \mathsf{acc}.\mathit{pressure}.B_A \\
 & \boxdot(\mathit{pressure} < \mathit{High})
\end{array}
$$

Case $j = 3$:

$$
\begin{array}{ll}
 & \ominus PC_3 \wedge PC_1 \\
\Rrightarrow & \quad \text{definition of } PC_3 \\
 & \{(23)\}\ PC_1 \\
\Rrightarrow & \quad \text{case } \overrightarrow{\mathit{Stopped}} \text{ reduces to } \mathit{false} \\
 & \quad \text{by guard of } PC_1 \\
 & \ominus \overrightarrow{\mathit{pressure} < \mathit{High} - \mathsf{acc}.\mathit{pressure}.B_A} \wedge PC_1 \\
\equiv & \quad \mathit{pressure} \text{ is continuous} \\
 & (\overleftarrow{\mathit{pressure} < \mathit{High} - \mathsf{acc}.\mathit{pressure}.B_A}) \wedge PC_1 \\
\Rrightarrow & \quad \text{Lemma 21 use } \mathit{AEnv} \wedge \ell \leq \mathsf{pn}.B_A \\
 & \boxdot(\mathit{pressure} < \mathit{High})
\end{array}
$$

Finally, returning to the proof of $(\mathit{Req}_1)$, we have

$$
\begin{array}{ll}
 & \mathit{iter}.PC.1 \Rrightarrow \boxdot(\mathit{pressure} \geq \mathit{High} \Rightarrow \mathit{Stopped}) \\
\Llefttarrow & \quad \text{calculations above} \\
 & \boxdot(\mathit{pressure} < \mathit{High}) \Rrightarrow \boxdot(\mathit{pressure} \geq \mathit{High} \Rightarrow \mathit{Stopped}) \\
\Llefttarrow & \quad \text{logic} \\
 & \mathit{true}
\end{array}
$$

This completes the case analysis, and hence, the proof for $PC_1$. The proof for $PC_2$ is symmetric and $PC_3$ satisfies (21) $\wedge$ (22) by definition, which allows us to conclude that the abstract system is safe with respect to requirement (21) $\wedge$ (22).

## Appendix C. Proofs for the concrete pump

This section contains some proofs for the concrete pump.

*Proof of (31).* This proof obligation requires that we show the behaviours of the concrete and abstract systems match. We define the following shorthand:

$$
\begin{array}{rcl}
\mathit{AAcc} & \widehat{=} & \mathsf{mc}.\mathit{pressure} \leq \mathsf{acc}.\mathit{pressure}.B_A \\
\mathit{CAcc} & \widehat{=} & \mathsf{mc}.\mathit{pressure}_1 + \mathsf{mc}.\mathit{pressure}_2 \leq \mathsf{acc}.\mathit{pressure}.B_A
\end{array}
$$

then obtain the following calculation.

$$
\begin{array}{ll}
 & \mathit{ref} \wedge (\mathit{Cont}\downarrow)^\infty \wedge (\mathit{CWater}\downarrow) \wedge (\mathit{CPump}\downarrow) \Rrightarrow \mathit{AEnv}\uparrow \\
\Llefttarrow & \quad \text{logic} \\
 & \mathit{ref} \wedge (\mathit{CWater}\downarrow) \Rrightarrow (\mathit{AEnv}\uparrow) \\
\Llefttarrow & \quad \text{logic, distribute projection} \\
 & \mathit{ref} \Rrightarrow \Box(\ell \leq \mathsf{pn}.B_A \Rightarrow \mathit{CAcc}\downarrow) \Rightarrow \Box(\ell \leq \mathsf{pn}.B_A \Rightarrow \mathit{AAcc}\uparrow) \\
\Llefttarrow & \quad \text{logic: } \Box(g \Rightarrow h) \Rrightarrow (\Box g \Rightarrow \Box h) \text{ and } \mathit{ref} \text{ splits} \\
 & \mathit{ref} \Rrightarrow \Box((\mathit{CAcc}\downarrow) \wedge \ell \leq \mathsf{pn}.B_A \Rightarrow (\mathit{AAcc}\uparrow)) \\
\Llefttarrow & \quad \mathit{ref} \text{ splits, logic} \\
 & \mathit{ref} \wedge (\mathit{CAcc}\downarrow) \wedge \ell \leq \mathsf{pn}.B_A \Rrightarrow (\mathit{AAcc}\uparrow) \\
\Llefttarrow & \quad \text{definitions} \\
 & \mathit{true}
\end{array}
$$

*Proof of (32)..* This proof pertains to showing that the concurrent behaviour of the concrete controller and environment together implements the behaviour of the abstract controller using refinement relation *ref*. For this proof, we require use conditions (33), (34), and (35).

We aim to examine several consecutive iterations of the **do** loop of the concrete controller. Here, it is possible to make several simplifications.

- By Lemma 21 and (35), $Cont_1\ ;\ Cont_2\ ;\ Cont^\omega$ and $Cont_2\ ;\ Cont_1\ ;\ Cont^\omega$ both reduce to *false*.
- Because $Cont_1$ and $Cont_2$ falsify their own guards, both $Cont_1;\ Cont_1;\ Cont^\omega$ and $Cont_2;\ Cont_2;\ Cont^\omega$ reduce to *false*.
- Because $CInit \Rrightarrow \boxdot\neg signal \wedge \mathsf{r\_stable}.signal$, interval predicate $\ominus\, CInit \wedge Cont_1$ reduces to *false*.
- Finally, because $CInit \Rrightarrow \boxdot(pressure_1 + pressure_2 \geq CHigh_T)$ using (35) and Lemma 21, case $\ominus\, CInit \wedge Cont_2$ reduces to *false*.

Therefore, the behaviour of the concrete controller may be rewritten using:

$$Cont^\infty \quad \equiv \quad (CC_1 \vee CC_2 \vee CC_3)^\infty \tag{C.1}$$

where

$$\begin{array}{lcl} CC_1 & \widehat{=} & \ominus\, Cont_3 \wedge ((Cont_1 \wedge \ell \leq \mathsf{pn}.B_C)\ ;\ (Cont_3 \wedge \ell \leq \mathsf{pn}.B_C)^{\omega+}) \wedge (\mathsf{finite} \Rightarrow \oplus\, Cont_2) \\ CC_2 & \widehat{=} & \ominus\, Cont_3 \wedge ((Cont_2 \wedge \ell \leq \mathsf{pn}.B_C)\ ;\ (Cont_3 \wedge \ell \leq \mathsf{pn}.B_C)^{\omega+}) \wedge (\mathsf{finite} \Rightarrow \oplus\, Cont_1) \\ CC_3 & \widehat{=} & \ominus\, CInit \wedge (Cont_3 \wedge \ell \leq \mathsf{pn}.B_C)^{\omega+} \end{array}$$

By $CC_1$, $Cont_3$ was previously executing and the current interval can be split so that $Cont_1$ executes in the first part (which switches the pump on) then $Cont_3$ executes iteratively in the second (causing the controller to be idle so that all of the outputs are stable). Furthermore, the current interval is either infinite ($Cont_3$ executes forever), or there is some next interval in which $Cont_2$ executes. Interval predicate $CC_2$ is similar. By $CC_3$, the program initialised in some immediately preceding interval $Cont_3$ executes iteratively in the current interval. Using this, we perform the following simplification of the antecedent of (32).

$$\begin{array}{ll} & ref \wedge (Cont\downarrow)^\infty \wedge (CWater\downarrow) \wedge (CPump\downarrow) \\ \equiv & \quad \text{(C.1)} \\ & ref \wedge ((CC_1 \vee CC_2 \vee CC_3)^\infty\downarrow) \wedge (CWater\downarrow) \wedge (CPump\downarrow) \\ \Rrightarrow & \quad ref \wedge (CWater\downarrow) \wedge (CPump\downarrow) \text{ splits} \\ & (ref \wedge ((CC_1 \vee CC_2 \vee CC_3)\downarrow) \wedge (CWater\downarrow) \wedge (CPump\downarrow))^\infty \end{array}$$

To prove that the predicate above implies the behaviour of the abstract system, we perform case analysis on the behaviour of each iteration and each disjunct separately.

For case $CC_1$, we use yet another property on sampled values and lengths of intervals. Namely, if the sum of $pressure_1$ and $pressure_2$ in the current interval is possibly above $CHigh_T$, and in some next interval of length $\mathsf{pn}.B_C$ the sum is below $CLow_T$, then the length of the current interval must exceed $\mathsf{pn}.B_P + \mathsf{pn}.B_C$, i.e.,

$$\begin{array}{l} \circledast(pressure_1 + pressure_2 \geq CHigh_T) \\ \wedge\ \oplus(\circledast(pressure_1 + pressure_2 \leq CLow_T) \wedge \ell \leq \mathsf{pn}.B_C) \end{array} \quad \Rrightarrow \quad \ell > \mathsf{pn}.B_P + \mathsf{pn}.B_C \tag{C.2}$$

We then obtain the following calculation for the execution of a single iteration of $CC_1$. Assuming that *CWater* and *CPump* hold, we have the following calculation, where we define $pressure_{12} \mathrel{\widehat{=}} pressure_1 + pressure_2$. The calculation shows that

$$\begin{array}{lll} CC_1 & \Rrightarrow & \big(\diamondsuit(pressure_{12} \geq High_T) \wedge \boxdot\neg Stopped \wedge \ell \leq \mathsf{pn}.B_C\big)\ ;\ (\boxdot Stopping \wedge \ell < \mathsf{pn}.B_P)\ ; \\ & & \underline{\boxdot(Stopped \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A))} \end{array}$$

i.e., that whenever $CC_1$ holds,

- there is an initial interval of length at most $\mathsf{pn}.B_C$ in which $pressure_{12}$ is actually above $High_T$, and the pump is actually stopped throughout the interval, followed by
- an interval of length at most $\mathsf{pn}.B_P$ the pump is stopping throughout this interval, followed by
- an interval in which the pump is stopped and the actual value of $pressure_{12}$ is higher than $Low + \mathsf{acc}.pressure.B_A$.

Note the differences from the sampled predicates (that use '$\circledast$') in $CC_1$ to predicates on the actual states (that use '$\diamond$' and '$\boxdot$') in the consequent.

$$
\begin{array}{ll}
 & CC_1 \\
\Rrightarrow & \quad \text{definition of } CC_1\text{, logic} \\
 & ((Cont_1 \wedge \ell \leq \mathsf{pn}.B_C)\,;\ (Cont_3 \wedge \ell \leq \mathsf{pn}.B_C)^{\omega+}) \wedge (\mathsf{infinite} \vee \oplus Cont_2) \\
\Rrightarrow & \quad \text{definition of } Cont_1 \\
 & \left(\begin{array}{l}((\circledast(pressure_{12} \geq CHigh_T) \wedge \boxdot signal)\,;\ \underline{\boxdot\neg signal}\ \wedge\ \ell \leq \mathsf{pn}.B_C)\,; \\ \underbrace{(\neg signal}\ \wedge\ Cont_3 \wedge \ell \leq \mathsf{pn}.B_C)^{\omega+}\end{array}\right) \wedge \\
 & (\mathsf{infinite} \vee \oplus Cont_2) \\
\Rrightarrow & \quad \text{condition (C.2) and } Cont_2 \Rightarrow \circledast(pressure_{12} \leq CLow_T) \wedge \ell \leq \mathsf{pn}.B_C \\
 & \quad \text{Lemma 21 assuming } CHigh_T \geq High_T + \mathsf{acc}.pressure_1.B_C + \mathsf{acc}.pressure_2.B_C \\
 & \left(\begin{array}{l}((\circledast(pressure_{12} \geq CHigh_T) \wedge \boxdot signal)\,;\ \underline{\boxdot\neg signal} \wedge \ell \leq \mathsf{pn}.B_C)\,; \\ \underline{(\boxdot(\neg signal \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A)))^{\omega+}}\end{array}\right) \wedge \\
 & (\ell > \mathsf{pn}.B_P + \mathsf{pn}.B_C) \\
\Rrightarrow & \quad \boxdot c \text{ joins and Lemma 21 using (33)} \\
 & \left(\begin{array}{l}((\diamond(pressure_{12} \geq High_T) \wedge \boxdot signal)\,;\ \underline{\boxdot\neg signal} \wedge \ell \leq \mathsf{pn}.B_C)\,; \\ \boxdot(\neg signal \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A))\end{array}\right) \wedge \\
 & (\ell > \mathsf{pn}.B_P + \mathsf{pn}.B_C) \\
\Rrightarrow & \quad \text{logic and '; ' is associative} \\
 & \left(\begin{array}{l}(\diamond(pressure_{12} \geq High_T) \wedge \boxdot signal \wedge \ell \leq \mathsf{pn}.B_C)\,;\ \underline{\boxdot\neg signal}\,; \\ \underline{\boxdot(\neg signal \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A))}\end{array}\right) \wedge \\
 & (\ell > \mathsf{pn}.B_P + \mathsf{pn}.B_C) \\
\Rrightarrow & \quad \text{by } CPump\text{, } \boxdot(signal \Rightarrow \neg Stopped) \\
 & \quad \text{use } \ell > \mathsf{pn}.B_P + \mathsf{pn}.B_C \text{ and assumption } CPump \\
 & (\diamond(pressure_{12} \geq High_T) \wedge \boxdot\neg Stopped \wedge \ell \leq \mathsf{pn}.B_C)\,; \\
 & (\boxdot Stopping \wedge \ell < \mathsf{pn}.B_P)\,;\ \underline{\boxdot(Stopped \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A))}
\end{array}
$$

Therefore we have the following calculation, where *ref* is used to transform a property over a concrete state space to an abstract state space. In particular, we show that execution of $CC_1$

$$
\begin{array}{ll}
 & ref \wedge (CC_1\downarrow) \wedge (CWater\downarrow) \wedge (CPump\downarrow) \\
\Rrightarrow & \quad \text{using } ref \text{ and calculation above} \\
 & \big(\diamond(pressure \geq High_T) \wedge \boxdot\neg Stopped \wedge \ell \leq \mathsf{pn}.B_C\big)\uparrow\,;\ (\boxdot\neg Stopped \wedge \ell < \mathsf{pn}.B_P)\uparrow\,; \\
 & \qquad\qquad \underline{\boxdot(Stopped \wedge (pressure > Low + \mathsf{acc}.pressure.B_A))}\uparrow \\
\Rrightarrow & \quad \boxdot c \text{ joins, condition (36)} \\
 & (\diamond(pressure \geq High_T) \wedge \boxdot\neg Stopped \wedge \ell < \mathsf{pn}.A)\uparrow\,; \\
 & \underline{\boxdot(Stopped \wedge (pressure > Low + \mathsf{acc}.pressure.B_A))}\uparrow \\
\Rrightarrow & \quad \text{logic, } \boxdot c \text{ splits} \\
 & \left(\begin{array}{l}(\diamond(pressure \geq High_T) \wedge \boxdot\neg Stopped)\,;\ \boxdot Stopped\ \wedge \\ \mathsf{r\_stable}.pump \wedge \ell \leq \mathsf{pn}.A\end{array}\right)\Big\uparrow\,; \\
 & \underline{\boxdot(Stopped \wedge (pressure > Low + \mathsf{acc}.pressure.B_A))}\uparrow \\
\Rrightarrow & \quad \text{definitions} \\
 & PumpCont_1\uparrow\,;\ PumpCont_3\uparrow
\end{array}
$$

For case, $CC_2$, we apply a similar calculation to show $CC_2$ implies $PumpCont_2 \uparrow ; PumpCont_3 \uparrow$.

Finally, for case $CC_3$, we use *CInit* to conclude that $\boxdot(Stopped \wedge (pressure_{12} > Low + \mathsf{acc}.pressure.B_A))$ holds, which trivially implies (21) $\wedge$ (22) $\wedge$ (23).

Thus, we obtain our desired result (32), which concludes our proof.