

Deriving real-time action systems in a sampling logic

Brijesh Dongol and Ian J. Hayes

The University of Queensland, Australia

Abstract

Action systems have been shown to be applicable for modelling and constructing systems in both discrete and hybrid domains. We present a novel sampling logic semantics for action systems that facilitates reasoning about the truly concurrent behaviour between an action system and its environment. By reasoning over the apparent states, the sampling logic allows us to determine whether a state predicate is definitely or possibly true over an interval. We present a semantics for action systems that takes the time taken to sample inputs and evaluate expressions (and hence guards) into account. We develop a temporal logic based on the sampling logic that facilitates formalisation of safety, progress, real-time and transient properties. Then, we incorporate this logic to the method of enforced properties, which facilitates stepwise refinement of action systems.

Keywords: Sampling logic, action systems, real-time properties, true concurrency, safety-critical systems

1. Introduction

The theory of action systems is well developed and has been applied to a number of different problem domains. Due to the simplicity of the model, action systems have been used as a basis for several theories of program development such as refinement [1] and the verify-while-develop paradigm using enforced properties [2]. Part of the simplicity of action systems is due to the inherent interleaving semantics assumption, which ensures that statements are executed one after another (even after parallel composition), and hence, a concurrent system is viewed as a sequential program with a non-deterministic choice over all parallel actions. However, an interleaving semantics is problematic for truly concurrent systems, where execution of parallel statements may overlap, e.g., real-time systems where the environment of a program executes in parallel with the program in a truly concurrent manner.

In this paper we incorporate the sampling logic of Burns and Hayes [3] into the action systems framework, which allows the state of the environment to be determined using “sampling activities”. This theory allows us to model true concurrency between the parallel processes of a system. In particular, we describe how the real-time behaviour of an action system may be formalised in a manner that takes into account sampling of its concurrently evolving environment.

When using a state-based approach, it is difficult to determine the length of time for which a state has been enabled, which is problematic for the specification of progress properties. For example, consider a requirement $X \rightsquigarrow Y$ where X and Y are state predicates, i.e., whenever X holds then eventually must Y become true. During the execution of a system, X may only be true for a brief amount of time, say an attosecond, which means it is impossible for a real system to reliably determine that X held. Thus, although $X \rightsquigarrow Y$ may be a desired property of the system, it may be unimplementable. Using our sampling logic, property $X \rightsquigarrow Y$ may be specified as $(\otimes X) \rightsquigarrow (\square Y)$, i.e., whenever X is definitely true over a sampling interval, then there must eventually be a sampling interval within which Y holds. Here, we have stated that X must hold for long enough for it to be detected by a sampling event. If X only holds for an attosecond, then our system need not guarantee that Y eventually holds (unless the sampling intervals are of attosecond length). Furthermore, X may refer to input variables that are evolving during the sampling event and hence, X must hold over all states that may be apparent to the system, i.e., if it is possible to detect $\neg X$, then the system need not guarantee that Y eventually holds.

Our focus for this paper is the formal development of actions systems from its specification, as opposed to verification of pre-existing systems. The method of derivation is inspired by Dijkstra’s verify-while-develop paradigm

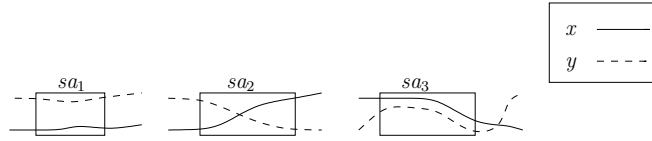


Figure 1: Sampling activities

[4], which has been related to program refinement [5, 2, 6] using “enforced properties”. An enforced property is a temporal logic formula [7] that restricts the traces of a system to those that satisfy the property being enforced. By using a temporal logic on relations, the verify-while-develop method has been shown to be useful for developing programs in a compositional manner [2]. However, the example development in [2] also elucidated some deficiencies in the framework — in particular, time-dependent properties could not be properly expressed.

Thus, we present a novel linear temporal logic [7] on sampling predicates, i.e., our temporal logic uses predicates on sampling activities as the base element, as opposed to state predicates. Then, we incorporate our logic into a framework of enforced properties, which allows us to systematically refine the code from the specification. As an example, we present the specification of an industrial press using our logic, and describe how we may derive its action system controller.

1.1. Related work

Several approaches to extending action systems with real-time behaviour have been proposed [8, 9, 10, 11] and there are several existing frameworks for modelling real-time system (e.g., Hybrid automata [12], TLA+ [13]). However, none of these frameworks take sampling into account. Furthermore, these methods are used to show that a concrete system is an implementation (or refinement) of its abstract counterpart, as opposed to this paper where we derive the action system code from its abstract specification.

Fidge and Wellings describe a method of extending action systems with actions that consume time [8]. Unlike our model where time is implicitly advanced, Fidge and Wellings use a ‘tick’ action to model the passage of time and accessibility restrictions are used to allow multiple processes to modify the same variable. Rönkkö et al extend action systems with actions that describe continuous changes to the state using a differential relation [9, 14]. A weakest precondition for differential actions is provided, however, due to the generality of the definition, the first derivative of the *evolution function* is required to be continuous. Back et al present refinement relations for continuous action systems [11], which have been extended by Meinicke and Hayes [10]. We present a method for refining action systems using enforced properties, which is an inherently different method than downward/upward (or forward/backward) simulation [6].

This paper is structured as follows. In Section 2 we present our sampling logic and in Section 3, we present the syntax and semantics of action systems. In Section 4, we describe enforced properties and a temporal logic on sampling predicates. We present an example derivation in Section 5.

2. Sampling logic

A reactive controller uses *sampling activities* to determine the state of its continuously evolving environment. Because sampling activities take time and because the environment operates in parallel with the controller in a truly concurrent manner, the controller can be prone to *sensor errors* (where the sensors have inaccuracies in measuring the environment), *timing precision errors* (where there is a range of possible sampled values due to imprecise timing of when the sample is taken), and *sampling anomalies* (where a sampling activity that samples more than one environment variable returns a non-existent state). In this paper, we focus on timing precision errors and sampling anomalies — reasoning about the sensors errors is straightforward.

To motivate sampling anomalies, we consider the three sampling activities sa_1 , sa_2 and sa_3 in Fig. 1, where environment variables x and y are sampled. Sampling activity sa_1 will return $x < y$ regardless of when the values of x and y are read within the sampling interval, because it is *definitely* true that $x < y$. Sampling activity sa_2 may return either $x > y$, $x = y$ or $x < y$ because it is *possibly* true that $x > y$, $x = y$, and $x < y$ hold. Activity sa_3 may

have a sampling anomaly. Although $x < y$ holds throughout sa_3 , because x and y are sampled at different times, it is possible for sa_3 to return $x > y$, $x = y$, and $x < y$.

We present a framework for real-time reasoning in Section 2.1 and formalise the concepts of definitely and possibly in Section 2.2, which facilitates reasoning about sampling activities. We present the concepts of stability and invariance in Section 2.3.

2.1. Interval predicates

We base our logic on traces over *contiguous intervals*. We consider an interval to be a closed contiguous subset of *Time* (represented by real numbers \mathbb{R}). An interval has type

$$Interval \hat{=} \{ \Omega \subseteq \mathbb{R} \mid \Omega \neq \{ \} \wedge \forall t, t' \in \Omega \bullet t < t' \Rightarrow \forall t'' : \mathbb{R} \bullet t < t'' < t' \Rightarrow t'' \in \Omega \}$$

Thus, if t and t' are in the interval Ω , then all real numbers between t and t' are also in Ω . For an interval $\Omega \in Interval$, we let $lub.\Omega$ and $glb.\Omega$ denote the least upper and greatest lower bounds of Ω , respectively where ‘.’ denotes function application. We need to refer to the intervals that directly follow a given interval. Thus, for an interval $\Omega \in Interval$, we define:

$$post.\Omega \hat{=} \{ \Omega' : Interval \mid (\Omega \cup \Omega' \in Interval) \wedge lub.\Omega = glb.\Omega' \}$$

We use $\ell.\Omega$ (equal to $lub.\Omega - glb.\Omega$) denote the length of Ω .

We define a *state space* as $\Sigma_{Var} \hat{=} Var \rightarrow Val$ where Var is a set of variables and Val a set of values. We leave out the subscript if Var is clear from the context. A *predicate* over a type X is given by $\mathcal{P}X \hat{=} X \rightarrow \mathbb{B}$, a *state* is a member of Σ , and a *state predicate* is a member of $\mathcal{P}\Sigma$. The (real-time) trace of system behaviours is given by $Stream_{Var} \hat{=} Time \rightarrow \Sigma_{Var}$ which is a total function from times to states with variables Var . We assume that each state is associated with a variable τ of type $Time$ whose value is determined for a stream $s \in Stream$ and time $t \in Time$ as $s_t.\tau \hat{=} t$.

A *stream predicate* is a member of $\mathcal{P}Stream$, and an *interval predicate* has type $IntvPred \hat{=} Interval \rightarrow \mathcal{P}Stream$. Interval predicates allow us to reason about the behaviour of a stream over a given contiguous interval. For a state predicate c , interval $\Omega \in Interval$, we define the *always* and *sometime* operators as follows:

$$\begin{aligned} (\boxtimes c).\Omega.s &\hat{=} \forall t : \Omega \bullet c.s_t \\ (\boxdot c).\Omega.s &\hat{=} \exists t : \Omega \bullet c.s_t \end{aligned}$$

For an expression $e \in \Sigma \rightarrow Val$, interval $\Omega \in Interval$ and stream $s \in Stream$, we define:

$$\begin{aligned} \overrightarrow{e}.\Omega.s &\hat{=} \lim_{t \rightarrow lub.\Omega} c.s_t \\ \overleftarrow{e}.\Omega.s &\hat{=} \lim_{t \rightarrow glb.\Omega} c.s_t \end{aligned}$$

Thus, $(\overrightarrow{e}.\Omega.s)$ and $(\overleftarrow{e}.\Omega.s)$ hold iff e holds in the right and left limits of Ω , respectively. We require the limits to be well defined. For our purposes it is sufficient to assume that e is piecewise continuous.

Given interval predicates $p, q \in IntvPred$ and intervals $\Omega \in Interval$, we define:

$$\begin{aligned} (p; q).\Omega &\hat{=} \exists \Omega_1, \Omega_2 : Interval \bullet (\Omega_1 \cup \Omega_2 = \Omega) \wedge (\Omega_2 \in post.\Omega_1) \wedge p.\Omega_1 \wedge q.\Omega_2 \\ [p].\Omega &\hat{=} (p; true).\Omega \\ (\boxplus p).\Omega &\hat{=} \forall \Omega' : Interval \bullet \Omega' \subseteq \Omega \Rightarrow p.\Omega' \end{aligned}$$

The *chop* operator ‘;’ allows the given interval to be split into two so that p holds for the first part and q holds for the second [15]. An interval predicate p holds over initial portion of an interval iff $[p]$ holds. The *everywhere* operator, \boxplus , requires the given interval predicate to hold over all subintervals of the given interval.

We find the *weak unless* operator useful for specifying properties of real-time systems [16]. For state predicates c and d , we define:

$$c \mathbf{V} d \hat{=} \overleftarrow{c} \Rightarrow \boxtimes c \vee (\boxtimes c; \overleftarrow{d})$$

Thus, if c holds at the start of the given interval, then either c holds throughout the interval, or the interval can be chopped so that c holds throughout the first portion and d holds at the start of the second.

We assume point-wise lifting of the boolean operators on stream and interval predicates in the normal manner, e.g., if p and q are interval predicates, $\Omega \in Interval$ is an interval and $s \in Stream$ is a stream, we have $(p \wedge q).\Omega.s = (p.\Omega.s \wedge q.\Omega.s)$. We define

$$\begin{aligned} p \Rightarrow q &\hat{=} \forall \Omega: Interval \bullet \forall s: Stream \bullet p.\Omega.s \Rightarrow q.\Omega.s \\ p \equiv q &\hat{=} \forall \Omega: Interval \bullet \forall s: Stream \bullet p.\Omega.s = q.\Omega.s \end{aligned}$$

In this paper, as much as possible, we reason at the level of intervals and implicitly assume lifting to the underlying streams.

2.2. Definitely and possibly

Given a set of states $SS \in \mathbb{P}\Sigma$, we define:

$$values.SS \hat{=} \lambda v: Var \bullet \{x: Val \mid \exists \sigma: SS \bullet x = \sigma.v\}$$

That is, $values.SS$ returns a state that maps each variable $v \in Var$ to the set of values that v may have in SS . Then, given that $sv: Var \rightarrow \mathbb{P} Val$ is a function that maps variables to sets of values, we define

$$apparent.sv \hat{=} \{\sigma: \Sigma \mid \forall v: Var \bullet \sigma.v \in sv.v\}$$

which generates the set of all states from the values mapping of a variable. These represent all possible values that a sampling activity may return. For example, suppose x and y are sensors and

$$SS = \{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$$

and SS contains all real states that occur during a sampling interval. Because x and y are sampled at different instants, it is possible to read x in the state $\{x \mapsto 0, y \mapsto 0\}$ and read y in the state $\{x \mapsto 1, y \mapsto 1\}$, which causes a sampling anomaly because the sampling activity returns $\sigma = \{x \mapsto 0, y \mapsto 1\}$ which is not an element of SS . To accommodate the sampling anomaly, we may construct

$$\begin{aligned} values.SS &= \{x \mapsto \{0, 1\}, y \mapsto \{0, 1\}\} \\ apparent.(values.SS) &= \{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 1\}\} \end{aligned}$$

where the latter corresponds to the set of all apparent states (including σ) that are possible when sampling over the actual states SS .

To reason about sampling anomalies, we define a function $states$, that returns the set of all states that occur within a real-time interval of a given stream, and a function av that returns the set of apparent values of the variables. In particular, if $\Omega \in Interval$ is a contiguous interval and $s \in Stream$ is a stream, we define:

$$\begin{aligned} states.\Omega.s &\hat{=} \{\sigma: \Sigma \mid \exists t: \Omega \bullet \sigma = s.t\} \\ av.\Omega.s &\hat{=} apparent.(values.(states.\Omega.s)) \end{aligned}$$

Using functions $states$ and av , we formalise state predicates that are *definitely* true (denoted \otimes) and *possibly* true (denoted \odot) over a given contiguous interval $\Omega \in Interval$ and stream $s \in PStream$ as follows:

$$\begin{aligned} (\otimes c).\Omega.s &\hat{=} \forall \sigma: av.\Omega.s \bullet c.\sigma \\ (\odot c).\Omega.s &\hat{=} \exists \sigma: av.\Omega.s \bullet c.\sigma \end{aligned}$$

If $(\otimes c).\Omega.s$ holds, then c holds for every apparent state in the interval Ω . Sampling predicate $\odot c$ is similar. Note that because $ss \subseteq apparent.(values.ss)$ for any set of states $ss \subseteq \Sigma$, both of the following hold for any interval $\Omega \in Interval$ and stream $s \in Stream$:

$$\begin{aligned} \otimes c &\Rightarrow \boxtimes c \\ \boxtimes c &\Rightarrow \odot c \end{aligned}$$

That is, if c holds in all apparent states within Ω of s , then c holds in all real states within Ω of s . Similarly, if c holds in a real state within Ω of s , then c holds in some apparent state within Ω of s . We have several useful properties of \boxtimes and \odot . Given a state predicate c , we have:

$$\neg \boxtimes c \equiv \odot \neg c \quad (1)$$

$$\boxtimes c \Rightarrow \odot c \quad (2)$$

$$\boxtimes c \wedge \boxtimes d \equiv \boxtimes (c \wedge d) \quad (3)$$

$$\odot c \vee \odot d \equiv \odot (c \vee d) \quad (4)$$

$$\boxtimes c \vee \boxtimes d \Rightarrow \boxtimes (c \vee d) \quad (5)$$

$$\odot (c \wedge d) \Rightarrow \odot c \wedge \odot d \quad (6)$$

The following lemma states that if c is possibly true at the start or end of an interval, then p is possibly true for the whole interval.

Lemma 1. $(\odot c; true) \vee (true; \odot c) \Rightarrow \odot c$

Lemma 2. *If the free variables of state predicates c and d are disjoint, then*

$$\boxtimes (c \Rightarrow d) \equiv \odot c \Rightarrow \boxtimes d$$

The proof of this lemma below may be found in [3].

2.3. Stability and invariance

We are often required to state that the values of some variables do not change, i.e., they are *stable* over an interval. For a variable x of type X and a set of variables $V \subseteq Var$, we define:

$$st.x \hat{=} \forall k: X \bullet \boxtimes (x = k)$$

$$st.V \hat{=} \forall x: V \bullet st.x$$

Thus, $st.x$ holds iff the value of x remains unchanged from its initial value in the given interval. If $st.V$ holds, then each variable in V is stable. The following lemmas describe some relationships between st , \boxtimes and \square .

Lemma 3. *Given that x is a boolean-valued variable, each of the following hold:*

$$st.x \Rightarrow \boxtimes x \vee \boxtimes \neg x$$

$$\square x \wedge st.x \equiv \boxtimes x$$

Sampling anomalies are only present if multiple (evolving) inputs are sampled within an interval. Thus, for a predicate, say c , that only refer to a single input variable, it is equivalent to state that c definitely holds and that c holds everywhere within an interval. This is formalised by the following lemma.

Lemma 4. *If state predicate c has at most one free variable that is not stable, then*

$$\boxtimes c \equiv \boxtimes \square c$$

$$\odot c \equiv \square \odot c$$

Invariance of a state predicate over an interval may also be defined. In particular, we define:

$$inv.c \hat{=} \overleftarrow{c} \Rightarrow \boxtimes c \quad (7)$$

Note that c may be invariant in an interval even if the variables in c are not stable in the interval.

3. Action systems

We present the syntax of action systems in Section 3.1 and their semantics in Section 3.2. We present trace refinement of real-time action systems in Section 3.3 and formalise action systems with frames in Section 3.4.

3.1. Syntax

Because we assume true concurrency between an action system \mathcal{A} and its environment, we use $\mathcal{A}.I \subseteq \text{Var}$, $\mathcal{A}.O \subseteq \text{Var}$, and $\mathcal{A}.L \subseteq \text{Var}$ to distinguish input, output, and local variables of \mathcal{A} . We use $\mathcal{A}.Var \hat{=} \mathcal{A}.I \cup \mathcal{A}.O \cup \mathcal{A}.L$ for the variables of \mathcal{A} and assume that $\mathcal{A}.I$, $\mathcal{A}.O$ and $\mathcal{A}.L$ are pairwise disjoint.

Definition 1. Suppose b is a predicate, \mathbf{y} is a vector of output variables and \mathbf{S} is a vector of set-valued expressions that has the same length as \mathbf{y} . Then the syntax of an action A has type:

$$\begin{aligned} S &::= \text{idle} \mid \mathbf{y} : \in \mathbf{S} \\ A &::= b \rightarrow S \mid A_1 \sqcap A_2 \end{aligned}$$

An assignment statement is a special case of non-deterministic assignment. In particular, for a variable x and an expression E :

$$x := E \hat{=} x : \in \{E\}$$

Definition 2. Given that A_0 and A are actions, an *action system* $\mathcal{A} \hat{=} A_0$; **do** A **od** consists of an *initialisation action* A_0 followed by a (possibly infinite) loop that executes action A .

We define a function grd , which returns the guard of the given action. We define:

$$\begin{aligned} grd.(b \rightarrow S) &\hat{=} b \\ grd.(A_1 \sqcap A_2) &\hat{=} grd.A_1 \vee grd.A_2 \end{aligned}$$

Using grd , we define a shorthand **else** as follows:

$$A \text{ else } S \hat{=} A \sqcap (\neg grd.A \rightarrow S)$$

3.2. Semantics

A software controller interacts with its environment in a truly concurrent manner, and hence the inputs to the controller may change during the evaluation of an expression (which may be a guard). We assume that for each iteration of the main loop of the action system, the expressions (including guards) are evaluated so that each input variable is sampled at most once, then the output variables are updated. Sampling inputs once within a sampling interval avoids anomalies when evaluating two guards that refer the the same input variable. For example, guards $x < 0$ and $x \geq 0$ could both evaluate to false (or both to true) within a single sampling interval if x increases past 0 during the interval and the value of x is sampled twice (once for each guard evaluation). Our model of sampling inputs once per sampling interval also avoids anomalies in guarded assignments. For example, action $x < 0 \rightarrow y := x$ should not assign a positive value to y , however, if different samples are used for the two occurrences of x , it is possible for y to obtain a positive value after execution of the action. Because both sampling and expression evaluation take time and the environment is executing in parallel with the action system, each input variable may have set of possible values. Hence, we use a sampling logic to distinguish whether a predicate is definitely or possibly true over an interval.

The behaviour of a statement over the interval in which it executes is given by function $beh: A \rightarrow \text{IntvPred}$, which returns an interval predicate for the given action. Within an action system with outputs O :

$$beh.(b \rightarrow \text{idle}) \hat{=} \odot b \wedge st.O \tag{8}$$

$$beh.(b \rightarrow \mathbf{y} : \in \mathbf{S}) \hat{=} (\exists \mathbf{k} \bullet (\odot(b \wedge (\mathbf{k} \in \mathbf{S})) \wedge st.\mathbf{y}); \otimes(\mathbf{y} = \mathbf{k})) \wedge st.(O \setminus \{\mathbf{y}\}) \tag{9}$$

$$beh.(A_1 \sqcap A_2) \hat{=} beh.A_1 \vee beh.A_2 \tag{10}$$

Thus, $b \rightarrow \text{idle}$ executes if b is possibly true and its execution leaves each output variable stable. The real-time behaviour of a guarded non-deterministic assignment, (9), chops the given interval into two, where guards and expressions are evaluated in the first portion, and the assignment is performed in the second. During the guard evaluation, there must be some apparent state in which b holds and each S_i evaluates to some value containing k_i . Furthermore, the value of \mathbf{y} is stable during the guard and expression evaluation. Following the expression evaluation, the value of \mathbf{y} is set to \mathbf{k} . During execution of the guarded assignment, the value of each output variable different from \mathbf{y} is stable. To avoid infeasible assignment statements [17], we require the following healthiness condition:

$$\odot b \Rightarrow \forall i \bullet \textcircled{\ast}(\mathbf{S}_i \neq \{\}) \quad (11)$$

that is, none of the sets \mathbf{S}_i is empty provided b holds. The behaviour of a non-deterministic choice between two actions A_1 and A_2 , (10), allows the behaviour of A_1 or A_2 to be chosen.

Lemma 5. *If $x \in O$ is a boolean-valued output variable, both of the following hold:*

$$\text{beh.}(x \rightarrow x := \text{false}) \equiv (\boxtimes x; \boxtimes \neg x) \wedge \text{st.}(O \setminus \{x\}) \quad (12)$$

$$\text{beh.}(\neg x \rightarrow x := \text{true}) \equiv (\boxtimes \neg x; \boxtimes x) \wedge \text{st.}(O \setminus \{x\}) \quad (13)$$

The closed intervals have type:

$$CI \hat{=} \{\Delta: \text{Interval} \mid \exists t: \mathbb{R} \bullet \Delta = [t..\infty) \vee \exists t': \mathbb{R} \bullet t \leq t' \wedge \Delta = [t..t']\}$$

where $[t..\infty)$ represents the infinite interval starting with t and $[t..t']$ represents the closed interval from t to t' . Note that an interval $\Delta \in CI$ may be a point interval, i.e., $\Delta = \{t\}$ for some $t \in \mathbb{R}$. Furthermore, because adjacent closed intervals overlap at a single point, for any interval $\Delta \in CI$, stream s and expression e :

$$\forall \Delta': \text{post.}\Delta \bullet \vec{e}.\Delta.s = \overleftarrow{e}.\Delta'.s \quad (14)$$

We define a set

$$CIPart \hat{=} \{z: \text{seq.}CI \mid (\bigcup_{i: \text{dom.}z} z_i) \in CI \wedge \forall i: \text{dom.}z \setminus \{0\} \bullet \text{lub.}z_{i-1} = \text{glb.}z_i\}$$

which contains all possible partitions of every interval.

A *trace* of an action system \mathcal{A} is a sequence of closed intervals, where the first interval is defined by the behaviour of the initialisation of \mathcal{A} and each successive interval is defined by the main action of \mathcal{A} . To avoid Zeno-like behaviour, we assume that each action takes at least $\epsilon \in \mathbb{R}$ units of time to execute, where $0 < \epsilon$. We also assume that the time taken for an action to be performed is bounded by a *sampling period*, $\rho \in \mathbb{R}$, where $\epsilon < \rho$ [3]. That is, the action within the **do** loop of an action system is guaranteed to be executed within ρ units of time. Thus, ϵ and ρ provide lower and upper bounds on the time taken to execute each action, respectively.

Definition 3. The set of all *traces* of action system \mathcal{A} is given by $TT.\mathcal{A}$, which is defined as:

$$TT.\mathcal{A} \hat{=} \left\{ z: CIPart, s: \text{Stream}_{\mathcal{A}.Var} \mid \begin{array}{l} (\text{beh.}A_0).z_0.s \wedge (\forall i: \text{dom.}z \setminus \{0\} \bullet (\text{beh.}A).z_i.s) \wedge \\ (\forall i: \text{dom.}z \bullet \epsilon < \ell.z_i \leq \rho) \end{array} \right\}$$

The set of all *complete traces* is given by $Tr.\mathcal{A}$ which is defined as follows:

$$Tr.\mathcal{A} \hat{=} \{(z, s): TT.\mathcal{A} \mid \text{dom.}z = \mathbb{N} \vee \exists \Delta: \text{post.}(\text{last.}z) \bullet (\odot \neg \text{grad.}A).\Delta.s\}$$

Thus, each complete trace of \mathcal{A} represents an infinite or terminating execution of \mathcal{A} . Execution of \mathcal{A} may terminate if it is possible for the guard of A to be false within a sampling interval. If $\textcircled{\ast} \neg \text{grad.}A$ holds within a sampling interval, then \mathcal{A} is guaranteed to terminate, however, if $\odot \neg \text{grad.}A$ holds, then it may be possible for $\odot \text{grad.}A$ to hold as well, in which case, depending on when the inputs are sampled, \mathcal{A} may either continue executing or terminate. Also note that $\text{grad.}(A \text{ else } S) = \text{true}$ for any action A , and hence **do**(A **else** S) **od** is non-terminating.

3.3. Trace refinement

A concrete action system \mathcal{C} trace refines an abstract action system \mathcal{A} iff every observable behaviour of \mathcal{C} is a possible observable behaviour of \mathcal{A} . Given that $\mathcal{A}.IO = \mathcal{A}.I \cup \mathcal{A}.O$ is the set of *observable variables*, we let $rL: Stream_{\mathcal{A}.Var} \rightarrow Stream_{\mathcal{A}.IO}$ be the function that removes local (unobservable) variables from the given stream by restricting the states in s to the observable variables $\mathcal{A}.IO$. For a partition $z \in CIPart$, we define $\bigcup z \triangleq \bigcup_{i:\text{dom}.z} z_i$ and define stream refinement between action systems as follows.

Definition 4. Action system \mathcal{C} trace refines \mathcal{A} (denoted $\mathcal{A} \sqsubseteq_{Tr} \mathcal{C}$) iff

$$\forall(z, s): Tr.\mathcal{C} \bullet \exists(z', s'): Tr.\mathcal{A} \bullet ((\bigcup z) \triangleleft (rL.s)) = ((\bigcup z') \triangleleft (rL.s'))$$

Lemma 6. If $Tr.\mathcal{C} \subseteq Tr.\mathcal{A}$ then $\mathcal{A} \sqsubseteq_{Tr} \mathcal{C}$.

An action A is refined by action C iff any behaviour of C is a possible behaviour of A . A refinement may, for example, reduce the non-determinism or strengthen the guard of an action. In the context of action system refinement, we may only refine an action in the **do** loop if it the new action does not introduce new states in which termination of the action system is possible [18]. We say an action A is *refined* by an action C , denoted $A \sqsubseteq C$, iff $beh.C \Rightarrow beh.A$ holds. If $A \sqsubseteq C$ and $C \sqsubseteq A$, we write $A \sqsubseteq C$. Refinement of actions may be related to trace refinement of action systems using the following lemma.

Lemma 7. Suppose $\mathcal{A} \hat{=} A_0$; **do** A **od** and $\mathcal{C} \hat{=} C_0$; **do** C **od**. Then $\mathcal{A} \sqsubseteq_{Tr} \mathcal{C}$ holds provided: $(A_0 \sqsubseteq C_0) \wedge (A \sqsubseteq C) \wedge (\otimes\text{grad}.A \Rightarrow \otimes\text{grad}.C)$.

In Lemma 7, condition $\otimes\text{grad}.A \Rightarrow \otimes\text{grad}.C$ follows because of the requirement that there must be a terminating trace of \mathcal{A} for every terminating trace of \mathcal{C} . Because \mathcal{A} can terminate if $\odot\neg\text{grad}.A$ holds, we have:

$$\begin{aligned} \odot\neg\text{grad}.C &\Rightarrow \odot\neg\text{grad}.A \\ &= \{(1)\} \\ \neg\otimes\text{grad}.C &\Rightarrow \neg\otimes\text{grad}.A \\ &= \{\text{contrapositive}\} \\ \otimes\text{grad}.A &\Rightarrow \otimes\text{grad}.C \end{aligned}$$

Lemma 8. If A, A_1 and A_2 are actions such that $A_1 \sqsubseteq A_2$, then each of the following holds:

$$A_1 \sqsubseteq A_1 \sqcap A_2 \tag{15}$$

$$\text{do } A \sqcap A_1 \text{ od} \sqsubseteq \text{do } A \sqcap A_1 \sqcap A_2 \text{ od} \tag{16}$$

3.4. Frames

When developing an implementation, it is often necessary to allow additional internal behaviour without introducing new observable traces. A formal and elegant way of achieving this within the refinement calculus is by using *frames* [19, 20]. We use \parallel for multiple assignment and it is defined for an action inductively as follows:

$$\begin{aligned} \text{idle} \parallel \mathbf{y} : \in \mathbf{S} &\hat{=} \mathbf{y} : \in \mathbf{S} \\ (\mathbf{y}_1 : \in \mathbf{S}_1) \parallel (\mathbf{y}_2 : \in \mathbf{S}_2) &\hat{=} \mathbf{y}_1, \mathbf{y}_2 : \in \mathbf{S}_1, \mathbf{S}_2 \\ (b \rightarrow S) \parallel (\mathbf{y} : \in \mathbf{S}) &\hat{=} b \rightarrow (S \parallel (\mathbf{y}_2 : \in \mathbf{S}_2)) \\ (A_1 \sqcap A_2) \parallel (\mathbf{y} : \in \mathbf{S}) &\hat{=} (A_1 \parallel (\mathbf{y} : \in \mathbf{S})) \sqcap (A_2 \parallel (\mathbf{y} : \in \mathbf{S})) \end{aligned}$$

Definition 5. If $\mathcal{A} \hat{=} A_0$; **do** A **od** is an action system, and $x \notin \mathcal{A}.Var$ is a new local variable of type X , then

$$\llbracket \text{var } x \bullet \mathcal{A} \rrbracket \hat{=} (A_0 \parallel x : \in X); \text{do } (A \parallel x : \in X) \text{ od}$$

Note that $\llbracket \text{var } x \bullet \mathcal{A} \rrbracket.L = \mathcal{A}.L \cup \{x\}$. The following lemma facilitates introduction of a new variable to an action system in a manner that preserves trace refinement.

Lemma 9. If \mathcal{A} is an action system, $x \notin \mathcal{A}.Var$ is a variable then $\mathcal{A} \sqsubseteq_{Tr} \llbracket \text{var } x \bullet \mathcal{A} \rrbracket$.

Application of Lemma 9 to an action system constitutes a single refinement step. In our approach, further refinements may be performed by restricting the possible values of the variables in the frame by introducing new enforced properties to the action system, which effectively reduces non-determinism.

4. Enforcing behaviour

We present a temporal logic on interval predicates in Section 4.1 and present our logic of enforced properties in Section 4.2.

4.1. A temporal logic

We specify properties of a program and its environment by specifying properties over the complete traces of the action system. In particular, we define a linear temporal logic on interval predicates (as opposed to state predicates [7]).

Definition 6. Given that $p \in \text{IntvPred}$ is an interval predicate, F is an ILTL formula, $z \in \text{seq}.CI$ is a sequence of intervals, $s: \text{Stream}$ is a stream and $i \in \text{dom}.z$, we define:

$$\begin{aligned}
((z, s), i) \vdash p &\hat{=} p.z_i.s \\
((z, s), i) \vdash \Box F &\hat{=} \forall j: \text{dom}.z \bullet j \geq i \Rightarrow (((z, s), j) \vdash F) \\
((z, s), i) \vdash \Diamond F &\hat{=} \exists j: \text{dom}.z \bullet j \geq i \wedge (((z, s), j) \vdash F) \\
((z, s), i) \vdash F_1 \mathcal{U} F_2 &\hat{=} \exists j: \text{dom}.z \bullet j \geq i \wedge (((z, s), j) \vdash F_2) \wedge (\forall k: i..j-1 \bullet (((z, s), k) \vdash F_1)) \\
((z, s), i) \vdash \Pi p &\hat{=} p.(\bigcup_{j \geq i} z_j).s
\end{aligned}$$

The notation $((z, s), i) \vdash F$ states that ILTL formula F holds for sequence of intervals z starting from index $i \in \text{dom}.z$. By definition, $((z, s), i) \vdash p$ holds iff p holds in the interval z_i . Operators \Box , \Diamond and \mathcal{U} express *always*, *eventually* and *until*, respectively. Formula $((z, s), i) \vdash \Pi p$ states that interval predicate p must hold for the interval consisting of the union of all intervals from z_i onwards. Note that

$$\Pi \boxtimes c = \Box \boxtimes c \quad (17)$$

We define some shorthand operators:

$$\begin{aligned}
F_1 \mathcal{W} F_2 &\hat{=} \Box F_1 \vee (F_1 \mathcal{U} F_2) \\
F_1 \rightsquigarrow F_2 &\hat{=} \Box(F_1 \Rightarrow \Diamond F_2)
\end{aligned}$$

Thus, F_1 *unless* F_2 holds (denoted $F_1 \mathcal{W} F_2$) iff either F_1 always holds or F_1 continues to hold until F_2 holds. F_1 *leads-to* F_2 (denoted $F_1 \rightsquigarrow F_2$) iff F_2 eventually holds whenever F_1 holds. For state predicates c and d , we define an operator \mathcal{V} as follows:

$$c \mathcal{V} d \hat{=} \overline{c} \Rightarrow (\boxtimes c \mathcal{W} [\boxtimes c \wedge \odot d]) \quad (18)$$

Hence, $c \mathcal{V} d$ holds iff given that c holds initially, either c definitely holds in each interval for the rest of the trace or c definitely holds in each interval until c definitely holds and d possibly holds in the initial portion an interval. Note that \mathbf{V} and \mathcal{V} are both weak unless properties, however, \mathbf{V} applies to the real states of the system whereas \mathcal{V} applies to the apparent states.

We distinguish between properties that the initialisation of an action system must satisfy against those that the system must maintain after initialisation. In this paper, we focus on the derivations of action systems after initialisation. Derivation of initialisation statements can be addressed in a similar manner to the methods in this paper, although, the properties themselves may be non-trivial [21].

Definition 7. An action system \mathcal{A} *maintains* a temporal formula F , denoted $\mathcal{A} \models F$, iff $\forall tr: \text{Tr}.\mathcal{A} \bullet (tr, 1) \vdash F$ holds.

Note that the first interval of each trace is ignored because it describes the initialisation of the action system. That is, maintenance of a property only considers the behaviour of the action system after it has been initialised.

ILTL may be used to formalise a number of different program properties. However using ILTL directly is not straightforward because we must reason at the level of traces. Instead, we present some calculational rules for proving that an action system satisfies particular forms of ILTL formulae. These rules are inspired by their discrete ILTL equivalents [6, 22].

Lemma 10. *If c and d are state predicates, then:*

$$\neg c \wedge \Box \text{inv}.c \Rightarrow \Box \boxtimes c \quad (19)$$

The following lemmas relate temporal properties on the stream to the the behaviour of an action system.

Lemma 11. *If c and d are state predicates such that each of c and d have at most one free variable that is not stable, then*

$$\Box((\boxtimes \neg c; \boxtimes c) \vee \text{inv}.c \vee [\text{inv}.c \wedge \Box d]) \Rightarrow \Box(c \vee d)$$

Lemma 12. *If c and d are state predicates, then*

$$\Box(\neg c \Rightarrow \boxtimes c \vee [\boxtimes c \wedge \odot d]) \Rightarrow \Box(c \vee d)$$

4.2. Enforced properties

An enforced property is an ILTL formula that restricts the traces of an action system to those that satisfy the enforced property [2, 5].

Definition 8. Action system \mathcal{A} with *enforced maintenance property* F , denoted $\mathcal{A} ? F$, is an action system such that $\text{Tr}(\mathcal{A} ? F) \hat{=} \{tr: \text{Tr}.\mathcal{A} \mid (tr, 1) \vdash F\}$.

Thus, although $\text{Tr}.\mathcal{A}$ may contain traces that do not satisfy F , by definition, F is guaranteed to hold for any trace of $\mathcal{A} ? F$. The goal then is to obtain an action system \mathcal{B} with no enforced properties such that $\mathcal{A} ? F \sqsubseteq \mathcal{B}$, i.e., \mathcal{B} is a refinement of \mathcal{A} whose traces satisfy F .

The following lemma describes trace refinement of action systems with enforced properties.

Lemma 13. *For action systems \mathcal{A} and \mathcal{C} , and ILTL formulae F and F' each of the following holds:*

$$\mathcal{A} \sqsubseteq_{\text{Tr}} \mathcal{A} ? F \quad (20)$$

$$\mathcal{A} ? F \sqsubseteq_{\text{Tr}} \mathcal{A} ? F' \quad \text{provided } F' \Rightarrow F \quad (21)$$

$$\mathcal{A} ? F \sqsubseteq_{\text{Tr}} \mathcal{C} ? F \quad \text{provided } \mathcal{A} \sqsubseteq_{\text{Tr}} \mathcal{C} \quad (22)$$

$$\mathcal{A} ? F \sqsubseteq_{\text{Tr}} \mathcal{A} \quad \text{provided } \mathcal{A} \models F \quad (23)$$

$$\mathcal{A} ? (F \wedge F') \sqsubseteq_{\text{Tr}} (\mathcal{A} ? F) ? F' \quad (24)$$

By Lemma 13, introducing a new enforced property or strengthening existing enforced properties results in a refinement. If an action system without an enforced property refines another, then the refinement holds with the enforced property included. An enforced property, say F , may be removed from an action system if the action system without enforced property F satisfies F . Furthermore, enforcing a conjunction of enforced properties is equivalent to enforcing the properties one at a time.

To allow finer-grained control over the enforced properties, we specify enforced interval predicates over actions as follows:

$$\text{beh.}(A ! p) \hat{=} \text{beh.}A \wedge p \quad (25)$$

Lemma 14. *For actions A , A_1 and A_2 , and interval predicates p and q , each of the following holds:*

$$A ! (p \wedge q) \sqsubseteq (A ! p) ! q \quad (26)$$

$$A ! (p \vee q) \sqsubseteq (A ! p) \sqcap (A ! q) \quad (27)$$

$$A ! p \sqsubseteq A \quad \text{provided } \text{beh.}A \Rightarrow p \quad (28)$$

$$(A_1 \sqcap A_2) ! p \sqsubseteq (A_1 ! p) \sqcap (A_2 ! p) \quad (29)$$

The theorem below states that an enforcing temporal formula $\Box p$ on an action system (where p is an interval predicate) is equivalent to enforcing p on the main action of the action system.

Theorem 15. *Given that $\mathcal{A} \hat{=} A_0$; **do** A **od** is an action system and p is an interval predicate,*

$$\mathcal{A} ? \Box p \sqsubseteq_{\text{Tr}} A_0; \text{do}(A ! p) \text{od}$$

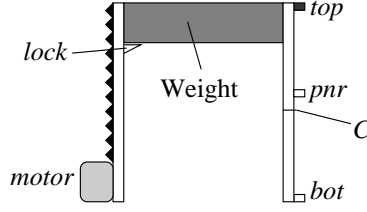


Figure 2: Industrial press

5. Industrial press example

We present a derivation of a software controller for the industrial press in Fig. 2. A weight is used to compress objects placed at the bottom, the *motor* used to lift the weight, the *lock* is used to keep the weight locked at the top, and sensors *top*, *pnr* and *bot* are used to determine the position of the weight within the press (at the top, below the point of no return and at the bottom). The user may press or release a button (not shown) to turn the motor on/off. We develop a controller for an industrial press [23, 21].

5.1. Specification and initial action system

The controller has inputs $top, pnr, bot, pressed \in I$ which represent the values of the different sensors and outputs $locked, on \in O$ that are used to control the lock and the motor. As described in [21], we require the motor to be off when the weight is locked at the top. If the user presses the button the lock must be released, which causes the weight to fall. The motor must remain off while the weight is falling unless the user releases the button and either the weight has reached the bottom (*bot* sensor is on) or is above the point of no return (*pnr* sensor is on). In both of these cases, the motor should be turned on to lift the weight. To prevent damage to the motor, a falling weight that is below the critical level *C* must not be lifted unless the weight has reached *bot*. Note that *C* is placed below the *pnr* sensor to allow the controller some time to detect that the weight has dropped below the *pnr* sensor. Once the weight is lifting, the motor must remain on until the weight is locked at *top*, regardless of the state of the button.

5.1.1. Environment

The relationship between the sensors and the state of the motor and lock is formalised as follows.

$$\text{II} \boxtimes ((top \Rightarrow \neg pnr \wedge \neg bot) \wedge (bot \Rightarrow pnr)) \quad (30)$$

$$\text{II} \boxplus (\boxtimes (on \vee locked) \Rightarrow inv.top) \quad (31)$$

By (30), if the *top* sensor is on, then the *pnr* and *bot* sensors must be off, and if the *bot* sensor is on, then the *pnr* sensor also on. By (31), for any interval, say Δ , if $on \vee locked$ holds throughout Δ , then *top* is invariant in Δ . Note that we cannot be guaranteed properties such as $\boxtimes on \Rightarrow inv.pnr$ over every interval, i.e., that if the motor is on throughout the interval then *pnr* is invariant. This is because the weight may be just above *pnr* when the motor is switched on and momentum of the weight takes the weight below *pnr* although the motor is on for the whole interval. This however is allowable as long as the weight does not drop below *C*.

The formula *Env* below describes our assumptions of the environment.

$$Env \hat{=} (30) \wedge (31)$$

Formula *Env* is essentially a rely condition [24, 25], however, unlike Jones, where execution of an action system is interleaved with the execution of the environment, we assume that the environment executes with the action system in a truly concurrent manner.

Other environment properties that describe the time taken to turn sensors on and off may also be specified. However, to simplify the presentation, we elide the details of properties that are not relevant to the derivation in this paper.

5.1.2. Timing

We assume that the weight can be safely lifted provided that the motor is turned on within M time units of a falling weight passing the pnr sensor. This guarantees that the weight will not be travelling too fast for the system to safely decelerate the weight and bring it to the top. In particular, we require

$$\Box(\ell \leq M) \quad (32)$$

which states that the length of each sampling period is at most M units. We introduce (32) to the program as an enforced property. This implicitly places a restriction on the sampling period ρ , i.e., (32) is satisfied by any implementation that ensures every iteration of the final action system can be completed within ρ units. The value of ρ depends on the complexity of the code of the final action system. By treating (32) as an enforced property, we may defer deciding on the value of ρ until the final action system code has been developed.

Note that (32) does not guarantee that the motor is turned on within M units; (32) only guarantees that an iteration of the `do` loop completes within M units. Thus, we require that the system satisfies:

$$\boxtimes(\neg top \wedge \neg pnr \wedge \neg pressed) \Rightarrow \Box on \quad (33)$$

That is, for any sampling interval, if it is definitely true that the weight is between below top and above pnr and the button is not pressed, then then motor must possibly be on. This could mean that the weight is being lifted, in which case $\odot on$ will trivially be true, or the weight is falling, in which case the weight must be aborted. Note that the aborting case only needs to be guaranteed if $\boxtimes(\neg top \wedge \neg pnr \wedge \neg pressed)$ holds. So for instance, (supposing that the weight is falling and is far above pnr) if the button is released then pressed within a single sampling interval, the controller may sample the pressed state and not turn the motor on. Consequently, if a button is released halfway through a sampling interval, say Δ , and remains released in the next sampling interval, say Δ' , the controller only needs to guarantee that the weight is aborted in Δ' . We require that (33) holds in every sampling interval, and hence our timing requirement is formalised by:

$$Timing \hat{=} (32) \wedge \Box(33)$$

We note that a condition equivalent to (33) is unimplementable in an state-based model without a sampling logic because a state-based model would only consider the value of a guard at the start of the interval. That is, property $c \rightsquigarrow d$ where c and d are state predicates may not be implementable in a real-time environment because c may only hold for a very brief amount of time (e.g., an attosecond), and it may not be possible to detect that c was true. For example, (33) translates to $\neg top \wedge \neg pnr \wedge \neg pressed \rightsquigarrow on$ where predicates on both sides of \rightsquigarrow are on states. In a real-time system, the state in which $\neg top \wedge \neg pnr \wedge \neg pressed$ holds may only hold for a brief amount of time, e.g., if the button is released when the falling weight is just above the pnr sensor. In contrast, (33) requires $\neg top \wedge \neg pnr \wedge \neg pressed$ to be definitely true over an entire sampling interval in order to guarantee that the motor is switched on. This discovery of this unimplementable property in a state-based system is described in [2, pg 136].

5.1.3. Safety and progress

The *safety* properties of the system are obtained from the informal description and are formalised by the formulae below. These properties consist of variables of the action system and the environment.

$$\boxtimes(locked \Rightarrow top) \quad (34)$$

$$locked \quad \mathbf{V} \quad pressed \quad (35)$$

$$on \quad \mathbf{V} \quad locked \quad (36)$$

Formula (34) states that the top sensor is on provided that the weight is locked. By (35), if the weight is locked at the start of the interval then either it remains locked for the rest of the interval or it remains on until the button is pressed. Condition (36) has a similar structure to (35).

It would seem that a requirement that states that the weight continues to fall until $(\neg pnr \vee bot) \wedge \neg pressed$ holds is necessary. However, because this requires more than one input to be sampled, it is possible for a falling weight to

be aborted without $(\neg pnr \vee bot) \wedge \neg pressed$ holding for any state in the stream. We discuss this property with more detail in Section 5.1.4.

The controller must satisfy the following *progress* properties:

$$\textcircled{\otimes}(pressed \wedge \neg on) \rightsquigarrow \square \neg locked \quad (37)$$

$$\textcircled{\otimes}(bot \wedge \neg pressed) \rightsquigarrow \square on \quad (38)$$

$$\textcircled{\otimes}locked \rightsquigarrow \square \neg on \quad (39)$$

$$\textcircled{\otimes}(on \wedge top) \rightsquigarrow \square locked \quad (40)$$

By (37), if the button is definitely pressed and the motor is definitely off, then the controller eventually unlocks the weight. Conditions (38), (39) and (40) have a similar structure. Note that predicates the right hand side of the \rightsquigarrow in (37)-(40) use \square as opposed to \odot , which requires that the state predicate hold over the real states of the system, as opposed to the apparent states. Although makes no difference to (37)-(40), for a state predicate, say c , that refers to more than on input variable, showing that $\square c$ holds is in general more difficult than showing $\odot c$ holds.

We define the following ILTL formulae which categorise the different requirements of the system:

$$Safe \hat{=} \square((34) \wedge (35) \wedge (36))$$

$$Prog \hat{=} (37) \wedge (38) \wedge (39)$$

Note that both *Safe* and *Prog* are linked to the intervals generated by execution of the action system. Furthermore, *Safe* and *Prog* are properties that must be maintained by execution of the action system after initialisation.

5.1.4. Transient properties

Our sampling logic is particularly useful for reasoning about *transient properties*, where an input changes from true to false while a different input changes from false to true within a single sampling interval. Thus, it may be possible to sample that both inputs are true (or both are false) even if only one of the inputs is true in every state of the stream.

For our industrial press example, consider the case for a falling weight that is above pnr at time t and below pnr at $t + \rho$, i.e., the weight drops below pnr within the sampling interval $[t, t + \rho]$. Suppose t' where $t < t' < t + \rho$ is the least time for which pnr holds in $[t, t + \rho]$, i.e., the weight reaches pnr at time t' . Now for some small k , suppose $(\boxtimes pressed).[t, t' + k]$ and $(\boxtimes \neg pressed).[t' + k, t + \rho]$ hold, i.e., the button is released at $t' + k$. Thus, $\boxtimes \neg(\neg pnr \wedge \neg pressed).[t, t + \rho]$ holds, i.e., there is no state in the stream for which $\neg pnr \wedge \neg pressed$ holds. However, the sampling event may sample $\neg pnr$ within $[t, t')$ and $\neg pressed$ within $[t' + k, t + \rho]$, i.e., $\neg pnr \wedge \neg pressed$ is possibly true within $[t, t + \rho]$. Hence, specifying a the safety condition for leaving the motor turned off as $\square(\neg locked \wedge \neg on) \vee ((bot \vee \neg pnr) \wedge \neg pressed)$ is impossible to implement. Instead, we require that

$$\neg locked \wedge \neg on \quad \mathcal{V} \quad (bot \vee \neg pnr) \wedge \neg pressed \quad (41)$$

By (41), if the weight is not locked and the motor is off, then they remain unlocked and off, respectively for the rest of the trace, or we are able to sample $(bot \vee \neg pnr) \wedge \neg pressed$ within a sampling interval. We would like the transient property to hold for all sampling intervals, and hence define

$$Trans \hat{=} \square(41)$$

Note that both (33) and (41) refer to actions that turn the motor on and are crucial to ensuring safe operation of the industrial press although they are different types of properties. Condition (33) requires that the controller must react by turning the motor *on* if $\neg top \wedge \neg pnr \wedge \neg pressed$ definitely holds within a sampling interval, while (41) states that the motor may be turned on even if the aborting condition, $\neg pnr \wedge \neg pressed$, is only apparently true.

5.2. Derivation

We now present the derivation of the main part of the controller. In Section 5.2.1 we derive the initial action system, in Section 5.2.2 we replace the ILTL formulae in *Safe* to simplify the calculation, in Section 5.2.3, we calculate the actions (together with the required guards) from the requirements and in Section 5.2.4 we consider progress properties.

$$\begin{array}{l} \mathbf{do} \quad true \rightarrow on, locked : \in \mathbb{B}, \mathbb{B} \\ \mathbf{od}?(Env \wedge Timing \wedge Safe \wedge Prog \wedge Trans) \end{array}$$

Figure 3: Initial action system

$$\begin{array}{l} \mathbf{do} \quad true \rightarrow on, locked : \in \mathbb{B}, \mathbb{B}!((33) \wedge (43) \wedge (44) \wedge (45) \wedge (46)) \\ \mathbf{od}?(Env \wedge (32) \wedge (42) \wedge Prog) \end{array}$$

Figure 4: Distribute actions

5.2.1. Initial program

Starting with enforced the ILTL formulae we arrive at action system (Fig. 3). The action system contains a single action that may modify the values of on and $locked$ (of type \mathbb{B}) to any value within their type. However, due to the enforced properties, the action system is guaranteed to be correct, i.e., the action system in Fig. 3 meets our requirements. We derive an action system with actions that maintain the enforced properties so that they need not be enforced.

5.2.2. Replace ILTL formulae

We aim to respectively replace the formulae with \mathcal{V} in $Safe$ with interval predicates that allow use of Theorem 15, which in turn allows the general enforced properties to be embedded within the actions. By Lemma 10 condition $\Box(34)$ holds if $(42) \wedge \Box(43)$ holds:

$$\overleftarrow{\neg locked} \vee \overleftarrow{top} \quad (42)$$

$$inv.(\neg locked \vee top) \quad (43)$$

Note that condition (42) is a requirement of the initialisation, while (43) should be satisfied by each action of the action system. By Lemma 11, conditions (35) and (36) respectively hold due to each of the following:

$$(\boxtimes \neg locked; \boxtimes locked) \vee inv.locked \vee [inv.locked \wedge \Box pressed] \quad (44)$$

$$(\boxtimes \neg on; \boxtimes on) \vee inv.on \vee [inv.on \wedge \Box locked] \quad (45)$$

Similarly, using Lemma 12, the transient property (41) holds if the following holds:

$$\overleftarrow{\neg locked \wedge \neg on} \Rightarrow \boxtimes(\neg locked \wedge \neg on) \vee [\boxtimes(\neg locked \wedge \neg on) \wedge \odot((bot \vee \neg pnr) \wedge \neg pressed)] \quad (46)$$

We define

$$Safe' \triangleq (42) \wedge \Box((43) \wedge (44) \wedge (45) \wedge (46))$$

and use (21) to introduce replace $Safe$ in the action system by $Safe'$. Then, using Theorem 15, we distribute the interval predicates $(43) \wedge (44) \wedge (45) \wedge (46)$ within the main action. Using Theorem 15 to distribute (33) within the main action, we obtain the program in Fig. 4.

5.2.3. Calculate actions

We must derive an action system that turns the motor and lock on/off, however, in order to satisfy the requirements, we must calculate the guards required of each action.

Turning the motor off. We consider the implications of action $true \rightarrow on := false$ against the safety condition $Safe'$. By definition of $beh.(true \rightarrow on := false)$, because $st.locked$ holds, (44) is trivial. Conditions (33) and (46) can be discharged using Lemma 5 and strengthening the guard to on . Condition (43) may be discharged by adding $locked$ as a conjunct to the guard which by Lemma 3 and $st.locked$ ensures $\boxtimes locked$. The guard $on \wedge locked$ also ensures $[on \wedge locked]$ holds, which implies (45). Hence, we obtain action:

$$on \wedge locked \rightarrow on := false$$

Turning the motor on. We consider action $true \rightarrow on := true$, which trivially satisfies (33). Because $st.locked$ holds, by strengthening the guard to $\neg locked$, by Lemma 3, we obtain $\boxtimes \neg locked$, and hence conditions (43) and (44) may be discharged. Then by adding conjunct $\neg on$ to the guard, by Lemma 5, we obtain $\overleftarrow{\neg on}$, which allows us to discharge (45). Now, because $\overleftarrow{\neg locked} \wedge \neg on$ holds, to satisfy (45), we strengthen the guard with conjunct $(bot \vee \neg pnr) \wedge \neg pressed$, which guarantees the second disjunct of the consequent of (46). Thus, we obtain action:

$$\neg locked \wedge \neg on \wedge (bot \vee \neg pnr) \wedge \neg pressed \rightarrow on := true$$

Unlocking the weight. We check safety against action $true \rightarrow locked := false$. Conditions (33), (44), (45) and (46) are satisfied by strengthening the guard to $locked \wedge \neg on \wedge pressed$. Thus, we obtain action:

$$locked \wedge \neg on \wedge pressed \rightarrow locked := false$$

We then have the following calculation:

$$\begin{aligned} & beh.(locked \wedge \neg on \wedge pressed \rightarrow locked := false) \\ \Rightarrow & \{ \text{definition of } beh, \text{ Lemma 5 and (6)} \} \\ & (\boxtimes locked \wedge \odot \neg on \wedge \odot pressed); \otimes \neg locked \\ \Rightarrow & \{(31)\} \\ & (\boxtimes locked \wedge \odot pressed \wedge inv.top); \boxtimes \neg locked \\ \Rightarrow & \{ \text{logic} \} \\ & (43) \end{aligned}$$

Locking the weight. We consider the behaviour of $true \rightarrow locked := true$. To satisfy (43), we strengthen the guard with top , which by the definition of beh gives us $\odot top$; $\otimes locked$. However, this does not ensure that $\otimes top$ holds in the second portion of the chop. Thus, we strengthen the guard again to ensure that the motor is on, which gives us the following calculation:

$$\begin{aligned} & beh.(top \wedge on \rightarrow locked := true) \\ \Rightarrow & \{ \text{definition of } beh \text{ and (1)} \} \\ & (\odot top \wedge \odot on; \otimes locked) \wedge st.on \\ \Rightarrow & \{ st.on, \text{ Lemma 1} \} \\ & (\odot top; \otimes locked) \wedge \boxtimes on \\ \Rightarrow & \{ \boxtimes on \text{ and (31)} \} \\ & (\odot top; \otimes top \wedge \otimes locked) \wedge \boxtimes on \end{aligned}$$

Because the left hand side of the chop above is not guaranteed to satisfy (43), we introduce $\neg locked$ as a conjunct to the guard. Hence, by Lemma 5 and Lemma 4, we obtain:

$$\begin{aligned} & (\odot top \wedge \boxtimes \neg locked; \boxtimes top \wedge \boxtimes locked) \wedge \boxtimes on \\ \Rightarrow & \{ \text{logic} \} \\ & (\odot top \wedge \boxtimes \neg locked; \boxtimes top \wedge \boxtimes locked) \wedge \boxtimes (\neg locked \vee top) \wedge \boxtimes on \\ \Rightarrow & \{ \text{logic} \} \\ & (33) \wedge (43) \wedge (44) \wedge (45) \wedge (46) \end{aligned}$$

Thus, we obtain action:

$$\neg locked \wedge on \wedge top \rightarrow locked := true$$

We introduce the actions to the action system using Lemma 8 and obtain the action system in Fig. 5, where the actions have been labelled a_1, \dots, a_6 for reference. Note we have not yet removed a_6 from the action system because the actions may need to be modified, or new actions may need to be introduced to satisfy progress.

do $locked \wedge on \rightarrow on := false$	(a ₁)
$\sqcap \neg locked \wedge \neg on \wedge bot \wedge \neg pressed \rightarrow on := true$	(a ₂)
$\sqcap \neg locked \wedge \neg on \wedge \neg pnr \wedge \neg pressed \rightarrow on := true$	(a ₃)
$\sqcap locked \wedge pressed \rightarrow locked := false$	(a ₄)
$\sqcap \neg locked \wedge on \wedge top \rightarrow locked := true$	(a ₅)
$\sqcap true \rightarrow on, locked : \in \mathbb{B}, \mathbb{B}! ((33) \wedge (43) \wedge (44) \wedge (45) \wedge (46))$	(a ₆)
od ?($Env \wedge (32) \wedge (42) \wedge Prog$)	

Figure 5: Introduce actions

do $locked \wedge on \rightarrow on := false$	(a ₁)
$\sqcap \neg locked \wedge \neg on \wedge bot \wedge \neg pressed \rightarrow on := true$	(a ₂)
$\sqcap \neg locked \wedge \neg on \wedge \neg pnr \wedge \neg pressed \rightarrow on := true$	(a ₃)
$\sqcap locked \wedge pressed \rightarrow locked := false$	(a ₄)
$\sqcap \neg locked \wedge on \wedge top \rightarrow locked := true$	(a ₅)
else idle	(a ₆)
od ?($Env \wedge (32) \wedge (42)$)	

Figure 6: Final action system

5.2.4. Progress properties

The concurrent behaviour of the environment is modelled using enforced ILTL formulae, which allows an action system to execute with its environment in a truly concurrent manner. Thus, the controller is essentially a sequential program, which means the proof of progress is straightforward. In particular, given the following formulae

$$\otimes(pressed \wedge \neg on) \Rightarrow \square \neg locked \quad (47)$$

$$\otimes(bot \wedge \neg pressed) \Rightarrow \square on \quad (48)$$

$$\otimes locked \Rightarrow \square \neg on \quad (49)$$

$$\otimes(on \wedge top) \Rightarrow \square locked \quad (50)$$

we define

$$Prog' \triangleq (47) \wedge (48) \wedge (49) \wedge (50)$$

and use Lemma 13 to replace $Prog$ by $\square Prog'$. Then using Theorem 15 and Lemma 14, we distribute $Prog'$ within the actions of the action system. Because $\otimes(pressed \wedge \neg on)$ causes all actions except a_4 to become blocked, and execution of a_4 establishes $\neg locked$ condition (47) is trivially satisfied. Similarly, due to enforced property Env , $\otimes(bot \wedge \neg pressed)$ disables all actions except a_2 and execution of a_2 establishes on , i.e., (48) is trivial. The proofs of (49) and (50) are similar.

To complete the proof of safety, we refine the non-deterministic assignment in a_6 to **idle**, and to complete the proof of progress, we must strengthen the guard of a_6 to **else**, which prevents a_6 from executing when the antecedents of (47)-(50) hold.

Thus, we obtain the action system in Fig. 6. For the purposes of this paper, we leave (42) enforced because it is not a maintenance property, i.e., it may be discharged by deriving the initialisation of the system. On the other hand, Env must remain enforced to achieve correctness of the system because it describes the behaviour of the environment, i.e., the assumptions under which the controller operates. The timing property (32) may be discharged during implementation of the code, i.e., by choosing hardware that ensures each iteration of the **do** loop can be completed within ρ units of time.

6. Conclusion

We have presented a logic of sampling, which we have used to present a novel semantics for action systems that allows us to assume true concurrency between an action system and its environment. This logic takes into account the

fact that expression evaluation takes time and that the values of the inputs may change during expression evaluation. The logic also allows us to detect and reason about sampling anomalies and timing precision errors that may occur during sampling. We have presented a temporal logic based on on sampling predicates to facilitate reasoning about the safety, progress and real-time properties of action systems with real-time environments.

We have formalised trace refinement between real-time action systems, then presented lemmas that allow introduction of fresh unobservable variables and reduction of non-determinism in a manner that preserves trace refinement. Further lemmas for stepwise derivation are provided by incorporating the theory of enforced properties into the framework. We present the derivation of a controller for an industrial press as an example.

Acknowledgements. This research was supported by Australian Research Council Discovery Grant (DP0987452) and The University of Queensland New Staff Research Fund.

References

- [1] R.-J. R. Back, K. Sere, Stepwise refinement of action systems, in: Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University, Springer-Verlag, 1989, pp. 115–138.
- [2] B. Dongol, I. J. Hayes, Compositional action system derivation using enforced properties, in: C. Bolduc, J. Desharnais, B. Ktari (Eds.), MPC, Vol. 6120 of LNCS, Springer, 2010, pp. 119–139.
- [3] A. Burns, I. J. Hayes, A timeband framework for modelling real-time systems, *Real-Time Systems* 45 (1) (2010) 106–142.
- [4] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
- [5] B. Dongol, I. J. Hayes, Enforcing safety and progress properties: An approach to concurrent program derivation, in: 20th Australian Software Engineering Conference, IEEE Computer Society, 2009, pp. 3–12.
- [6] B. Dongol, Progress-based verification and derivation of concurrent programs, Ph.D. thesis, The University of Queensland (2009).
- [7] Z. Manna, A. Pnueli, *Temporal Verification of Reactive and Concurrent Systems: Specification*, Springer-Verlag New York, Inc., 1992.
- [8] C. J. Fidge, A. J. Wellings, An action-based formal model for concurrent real-time systems, *Formal Aspects of Computing* 9 (1997) 175–207.
- [9] M. Rönkkö, A. P. Ravn, K. Sere, Hybrid action systems, *Theoretical Computer Science* 290 (1) (2003) 937 – 973.
- [10] L. Meinicke, I. J. Hayes, Continuous action system refinement, in: T. Uustalu (Ed.), *Mathematics of Program Construction*, Vol. 4014 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 316–337.
- [11] R.-J. R. Back, L. Petre, I. Porres, Generalizing action systems to hybrid systems, in: M. Joseph (Ed.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Vol. 1926 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2000, pp. 73–91.
- [12] R. S. N. Lynch, F. Vaandraager, Hybrid I/O automata, *Information and Computation* 185 (1) (2003) 105–157.
- [13] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2002.
- [14] M. Rönkkö, A. P. Ravn, Action systems with continuous behaviour, in: *Hybrid Systems V*, Springer-Verlag, London, UK, 1999, pp. 304–323.
- [15] C. Zhou, M. R. Hansen, *Duration Calculus: A Formal Approach to Real-Time Systems*, EATCS: Monographs in Theoretical Computer Science, Springer, 2004.
- [16] B. Dongol, I. J. Hayes, P. J. Robinson, Reasoning about real-time teleo-reactive programs, Tech. Rep. SSE-2010-01, Division of Systems and Software Engineering Research, School of Information Technology and Electrical Engineering, The University of Queensland (2010).
- [17] R.-J. R. Back, J. von Wright, *Refinement Calculus: A Systematic Introduction*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
- [18] R.-J. R. Back, J. von Wright, Trace refinement of action systems, in: *CONCUR '94: Proceedings of the Concurrency Theory*, Springer-Verlag, 1994, pp. 367–384.
- [19] C. Morgan, T. Vickers, *On the Refinement Calculus*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
- [20] B. Mahony, The least conjunctive refinement and promotion in the refinement calculus, *Formal Aspects of Computing* 11 (1999) 75–105.
- [21] I. J. Hayes, Dynamically detecting faults via integrity constraints, in: M. Butler, C. B. Jones, A. Romanovsky, E. Troubitsyna (Eds.), *Methods, Models and Tools for Fault Tolerance*, Vol. 5454 of LNCS, Springer, 2009, pp. 85–103.
- [22] K. M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley Longman Publishing Co., Inc., 1988.
- [23] J. McDermid, T. Kelly, Industrial press: Safety case, Tech. rep., High Integrity Systems Engineering Group, University of York (1996).
- [24] J. W. Coleman, C. B. Jones, A structural proof of the soundness of rely/guarantee rules, *J. Log. Comput.* 17 (4) (2007) 807–841.
- [25] C. B. Jones, Tentative steps toward a development method for interfering programs, *ACM Transactions on Programming Languages and Systems* 5 (4) (1983) 596–619.