Nordic Journal of Computing

Continuous Action Systems as a Model for Hybrid Systems[*]

Ralph-Johan Back, Luigia Petre and Ivan Porres
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
{Ralph.Back,Luigia.Petre,Ivan.Porres}@abo.fi

Abstract. Action systems have been used successfully to describe discrete systems, i.e., systems with discrete control acting upon a discrete state space. In this paper we define *continuous action systems* that extend the action system approach to also model hybrid systems, i.e., systems with discrete control over a continuously evolving state. The semantics of continuous action systems is defined in terms of traditional action systems and their properties are proved using standard action systems proof techniques. We describe the essential features of continuous action systems, show that they can be used to describe a diverse range of hybrid systems and illustrate the framework by a collection of examples.

CR Classification: F.3.1 F.4.1 D.2.4

Key words: Refinement Calculus, Action Systems, Hybrid Systems.

1. Introduction

A system using discrete control over its continuously evolving processes is referred to as a *hybrid system*. The use of formal methods and models to describe hybrid systems has attracted quite a lot of attention in the last years, with a number of different models and formalisms being proposed in the literature (see e.g., [2, 13, 9]). In this paper we continue this line of research, essentially proposing what we believe is a new and very general model for hybrid systems, based on the *action system* paradigm.

Action systems [4, 6] have been used successfully to model discrete systems, i.e., systems with discrete control upon a discrete state space. Their original purpose was to model concurrent and distributed systems. In this paper we show that the action system framework can be adapted to model hybrid systems. An important advantage of this adaption is that standard modeling and proof techniques, developed for ordinary action systems, can be reused to model and reason about hybrid systems.

Our extension of action systems to hybrid systems is based on a new approach to describing the state of a system. Essentially, our state attributes will range over functions of time, rather than just over values. This allows

[*]This article is an extended version of the conference paper published by Springer-Verlag in [5].

an attribute to capture not only its present value, but also the whole history of values that the attribute has had, as well as the *default* future values that the attribute will receive. Updating a state attribute is restricted so that only its future behavior can be changed, not its past behavior. We will refer to action systems with this model of state as *continuous action systems*. Continuous action systems are inspired by, but differ from, the extension of action systems to hybrid systems described in [15].

Proofs about action system properties are based on the refinement calculus [8]. This extends the programming logic based on weakest precondition predicate transformers proposed in [10]. Action systems are intended to be stepwise developed and the correctness of these steps to be verified within the refinement calculus. As continuous action systems are a consistent extension of action systems, we get an implicit notion of refinement also for them. Even though the refinement of hybrid systems is not the purpose of this paper, the approach we adopt for hybrid systems fits well into the refinement calculus framework and it can be used for systems where correct construction is a central concern.

The refinement calculus is based on higher-order logic, which in turn is an extension of simply typed lambda calculus. Functions are defined by λ-abstraction and can be used without explicit definition and naming. As an example, the function that calculates the successor of a natural number is defined as $(\lambda n \cdot n + 1)$. We denote by $f.x$ the application of the function f to the argument x, so that, e.g., $(\lambda n \cdot n + 1).1 = 2$. A binary relation $R \subseteq A \times B$ is here considered as a function $R : A \to \mathcal{P}B$, i.e., mapping elements in A to sets of elements in B.

We proceed as follows. The action system model is briefly reviewed in Section 2. In Section 3 we define the continuous action systems and explain their semantics; we also point out their generality. Section 4 contains examples of hybrid systems, modeled using our framework. In Section 5 we show how to prove safety properties for continuous action systems and in section 6 we show how to construct more complex continuous action systems. Conclusions and comparisons to related work are presented in Section 7.

2. Action Systems

We start by giving a brief overview of the action systems formalism. An action system consists essentially of a *discrete state space* updated by a *discrete control* mechanism [1]. The state of the system is described using *attributes*. For specifying them, we define a finite set *Attr* of *attribute names* and assume that each attribute name in *Attr* is associated with a non-empty set of *values*. This set of values is the *type* of the attribute. If the attribute x takes values from *Val*, we say that x has the type *Val* and we write it as $x : Val$. The name and the type completely specify the attribute. We

[1] We use a simple model of state here, see [8] for a more advanced model of state, which better supports proofs of program properties in higher order logic.

consider several predefined types, like Real for the set of real numbers, Real_+ for the set of non-negative real numbers, and Bool for the boolean values $\{\mathsf{F}, \mathsf{T}\}$. The value of an attribute can be read in an expression, and its value can be changed by an assignment, in the usual way.

An *action system* consists of a finite set of attributes used to observe and manipulate the *state* of the system and a finite set of *actions* that act upon the attributes. The set of actions models the *control mechanism* over the state of the system. An action system $\mathcal{A}$ has the following form:

$$\mathcal{A} \quad \triangleq \quad |[\mathsf{var}\ x \bullet S_0\ ;\mathsf{do}\ A_1 \Box\ \ldots \Box\ A_m\ \mathsf{od}\ ]| : y \tag{1}$$

Here $x = x_1, \ldots, x_n$ are the *local attributes* of the system, S_0 is a statement initializing them, while $A_i = g_i \to S_i$, $i = 1, \ldots, m$, are the *actions* of the system. The boolean expression g_i is the *guard* of the action A_i and S_i is the *body* of the action. The attributes $y = y_1, \ldots, y_k$ are defined in the environment of $\mathcal{A}$, being called *imported attributes.* Attributes in x may be *exported*, in the sense that they can be read, or written, or both read and written by environment actions. In this case, we decorate these attributes with $-$, $+$ or $*$, respectively. An action '$g \to S$' is an atomic guarded statement that is executed in the loop '$\mathsf{do}\ \ldots\ \mathsf{od}$ ' only when g is *enabled*, i.e., when g evaluates to T. The actions are executed in the loop as long as there are enabled actions. Hence, the loop is just Dijkstra's guarded iteration statement. The body S of an action is defined as follows:

$$S \quad ::= \quad \mathsf{abort} \mid \mathsf{skip} \mid x :\!= e \mid \{x :\!= x' | R\} \mid \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid S_1\ ; S_2$$

Here x is a list of attributes, e is a corresponding list of expressions, x' is a list of variables standing for unknown values, and R is a relation specified in terms of x and x'. Intuitively 'skip' is the stuttering action, '$x :\!= e$' is a multiple assignment, '$\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$' is the conditional composition of two statements, and '$S_1 ; S_2$' is the sequential composition of two statements. The action 'abort' always fails and is used to model disallowed behaviors. Given a relation $R(x, x')$ and a list of attributes x, we denote by $\{x :\!= x'|R\}$ the *non-deterministic assignment* of some value $x' \in R.x$ to x (the effect is the same as abort, if $R.x = \emptyset$). The semantics of the actions language has been defined in terms of weakest preconditions in a standard way [10]. Thus, for any predicate q, we define

$$\begin{array}{lll}
wp(\mathsf{abort}, q) & = & \mathsf{F} \\
wp(\mathsf{skip}, q) & = & q \\
wp(x :\!= e, q) & = & q[x := e] \\
wp(\{x :\!= x'|R\}, q) & = & (\forall x' \in R.x \cdot q[x := x']) \\
wp(S_1\ ;\ S_2, q) & = & wp(S_1, wp(S_2, q)) \\
\multicolumn{3}{l}{wp(\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, q) = \mathsf{if}\ g\ \mathsf{then}\ wp(S_1, q)\ \mathsf{else}\ wp(S_2, q)\ \mathsf{fi}}
\end{array}$$

where $q[x := e]$ stands for the result of substituting all the free occurrences of the attributes x in the predicate q.

The execution of an action system is as follows. The initialization S_0 will set the attributes to some specific values, using a sequence of possibly non-deterministic assignments. Then, enabled actions are repeatedly chosen and executed. The chosen actions will change the values of the attributes in a way that is determined by the action body. Two or more actions can be enabled at the same time, in which case one of them is non-deterministically chosen for execution. By interleaving atomic actions in a non-deterministic fashion, action systems can model parallel execution. The computation terminates when no action is enabled. Termination of an action system means the termination of the control over the system. This means that the state will evolve no more, fixing the final values of the attributes forever.

An action system is not usually regarded in isolation, but as a part of a more complex system. The rest of the system (the *environment*) communicates with the action system using shared (imported and exported) attributes, referred to as the *global* attributes. We can model other means of communication as well, using the action systems framework, but this is out of the scope of this paper [7].

A large action system is constructed from smaller ones using parallel composition. We define the parallel composition of two action systems

$$\begin{aligned}\mathcal{A} &= |[\mathsf{var}\ x : T \bullet S_0\ ; \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m\ \mathsf{od}\ ]| : y\\ \mathcal{A}' &= |[\mathsf{var}\ x' : T' \bullet S_0'\ ; \mathsf{do}\ g_1' \to S_1' \Box\ \ldots \Box\ g_n' \to S_n'\ \mathsf{od}\ ]| : y'\end{aligned}$$

as the action system $\mathcal{A} \parallel \mathcal{A}'$, defined by

$$\begin{aligned}\mathcal{A} \parallel \mathcal{A}' \ \triangleq\ & |[\mathsf{var}\ x : T, x' : T' \bullet S_0\ ; S_0'\ ;\\ & \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m \Box\ g_1' \to S_1' \Box\ \ldots \Box\ g_n' \to S_n'\\ & \mathsf{od}]| : (y \cup y') - (z \cup z)'\end{aligned}$$

We assume here that the attributes x and x' are disjoint. The initializations S_0 and S_0' may only refer to the respective attributes x and x', so the order in which the initializations are executed does not matter. The imported attributes in $\mathcal{A} \parallel \mathcal{A}'$ are the union of the attributes imported by $\mathcal{A}$ and $\mathcal{A}'$, but without the exported attributes z and z' of the two systems. The actions consist of the union of the actions of $\mathcal{A}$ and $\mathcal{A}'$.

In the next section we specify a notion of *time* and show how to model attributes that are functions of time. These extensions to the action systems formalism define a new model for hybrid systems.

3. Continuous Action Systems

A system using a discrete control over its continuously evolving state is referred to as a hybrid system. In this section we introduce continuous action systems, an extension of the action system formalism to model hybrid systems.

A *continuous action system* consists of a finite set of time-dependent attributes (the continuously evolving state) together with a finite set of actions (the discrete control) that act upon the attributes. It is of the form in (2):

$$\mathcal{C} \triangleq |[\mathsf{var}\ x : \mathsf{Real}_+ \to T \bullet S_0 \,;\, \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m\ \mathsf{od}\ ]| : y \qquad (2)$$

The time domain is modeled with the predefined type Real_+. The attributes x are functions of time and form the state of the system; they can range over discrete or continuous value domains T. Intuitively, the execution of $\mathcal{C}$ proceeds as follows. There is an implicit attribute *now*, that shows the present time. Initially $now = 0$. The initialization S_0 assigns initial time functions to the attributes $x_1, \ldots, x_n$. These time functions describe the default future behavior of the attributes, whose values may, thereby, change with the progress of time. The system will then start evolving according to these functions, with time (as measured by *now*) moving forward continuously. The guards of the actions may refer the value of *now*, as may expressions in the action bodies and the initialization statements.

Actions are urgent. As soon as one of the conditions $g_1, \ldots, g_m$ becomes true, the system chooses one of the *enabled* actions, say $g_i \to S_i$, for execution. The choice is non-deterministic if there is more than one such action. The body S_i of the action is then executed. Execution is atomic and instantaneous. It will usually change some attributes by changing their present and future behavior. Attributes that are not changed will behave as before. After the changes stipulated by S_i have been done, the system will evolve to the next time instance when one of the actions is enabled, and the process is repeated. The next time instance when an action is enabled may well be the same as the previous, i.e., time needs not to progress between the execution of two enabled actions. This is usually the case when the system is doing some (discrete, logical) computation to determine how to proceed next. Such a computation does not take any time. It is possible that after a certain time instance, none of the actions will be enabled anymore. This just means that, after this time instance, the system will continue to evolve forever according to the functions last assigned to the attributes. Hence, termination of the discrete control over the system only fixes the time functions corresponding to the attributes, and not their values.

As an example of a continuous action system consider the system shown in Fig. 1. We write $x :- e$ for an assignment rather than $x := e$, to emphasize that only the future behavior of the attribute x is changed to the function e and the past behavior remains unchanged.

The attributes x and *clock* are first initialized to the constant function $(\lambda t \cdot 0)$ and the switching function *up* is set to the constant function $(\lambda t \cdot \mathsf{F})$. The guard of the first action is immediately enabled at time 0, so the first action's body is executed immediately. The future behaviors of *clock* and x are changed to start increasing linearly from 0, and the future behavior of *up* is changed to the constant function $(\lambda t \cdot \mathsf{T})$, i.e., *up* is set to be true in all future time instances. After this, the system starts to evolve by advancing

```
Saw ≜ |(  var $x, clock : \mathsf{Real}_+ \to \mathsf{Real}, up : \mathsf{Real}_+ \to \mathsf{Bool}$
            • $x :- (\lambda t \cdot 0)\ ;clock :- (\lambda t \cdot 0)\ ;up :- (\lambda t \cdot \mathsf{F})$;
          do $x.now = 0 \wedge \neg up.now \to$
                $clock :- (\lambda t \cdot t - now)$;
                $x :- clock\ ; up :- (\lambda t \cdot \mathsf{T})$
           □ $x.now = 1 \wedge up.now \to$
                $clock :- (\lambda t \cdot t - now)$;
                $x :- (\lambda t \cdot 1 - clock.t)$;
                $up :- (\lambda t \cdot \mathsf{F})$
          od
      )|
```

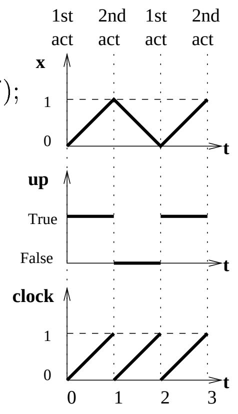


Fig. 1: Continuous action system *Saw* (left) and its behavior (right).

time continuously. In particular, the value of x increases linearly, depending on time. When x gets value 1 (expressed by $x.now = 1$), the second action is enabled. The clock is then first reset, the future behavior of x is changed to decrease linearly with the clock value, and the future behavior of up is set to the constant F. This continues until x reaches 0 ($x.now = 0$), when the first action is again enabled, changing x to increase again, and so on. The effect of these two actions is a saw-tooth-like behavior, where the value of x alternatively increases and decreases forever. The evolution of the system is also described in Fig. 1, showing each attribute on the same time domain together with the points in time where a discrete action is performed.

An even simpler system is as follows:

$$Sine \quad \triangleq \quad |(\ \mathsf{var}\ x : \mathsf{Real}_+ \to \mathsf{Real} \bullet\ x :- (\lambda t \cdot \sin t)\ )|$$

Here we have omitted the '`do ... od` ' construct because there are no actions at all that can change the system behavior. Thus, the attribute x of this system will behave as a continuous *sine* wave, forever.

One of the main advantages of the continuous action systems model for hybrid computation is that both discrete and continuous behavior can be described in the same way. In particular, if the attributes are only assigned constant functions, then we obtain a discrete computation.

3.1 Semantics of continuous action systems

Let $\mathcal{C}$ be the continuous action system in (2). We explain the meaning of $\mathcal{C}$ by translating it into an ordinary action system. Its semantic interpretation is defined by the following (discrete) action system $\bar{\mathcal{C}}$:

$$\begin{array}{lll} \bar{\mathcal{C}} \triangleq & |[& \mathsf{var} & now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to T \\ & & \bullet & now := 0 \,;\, S_0 \,;\, N; \\ & & \mathsf{do} & g_1 \to S_1 \,;\, N \square \;\ldots \square\; g_m \to S_m \,;\, N \;\mathsf{od} \\ &]| : & y & \end{array} \tag{3}$$

Here the attribute *now* is declared, initialized, and updated explicitly. It models the time moments that are of interest for the system, i.e., the starting time and the succeeding moments when some action is enabled. The value of *now* is updated by the statement $N \triangleq now := \mathsf{next}.gg.now$. Here $gg = g_1 \vee ... \vee g_m$ is the disjunction of all guards of the actions and next is defined by

$$\mathsf{next}.gg.t \triangleq \begin{cases} min\{t' \geq t \mid gg.t'\}, & \text{if } \exists\, t' \geq t \text{ such that } gg.t' \\ t, & \text{otherwise.} \end{cases} \tag{4}$$

The function next models the moments of time when at least one action is enabled. Only at these moments can the future behavior of attributes be modified. The presence of the function *min* means that all actions are urgent. If no action will ever be enabled, then the second branch of the definition will be followed, and the attribute *now* will denote the moment of time when the last discrete action was executed. In this case the discrete control terminates, i.e. the attributes will evolve forever according to the functions last assigned. One can observe from the definition of next that it is possible for an infinite number of actions to be enabled at the same moment. This implies that an infinite number of actions can be executed in no time, a feature generally known as a Zeno behavior. The treatment of Zeno behavior in the context of continuous action systems seems rather involved and it is planned to be thoroughly addressed in the future.

In this paper we assume that the minimum in the definition of next always exists, when at least one guard is enabled in the present or future. Continuous action systems not satisfying this requirement are considered ill-defined. E.g., the minimum in the definition of next cannot be calculated when there are strict inequalities in the time-dependent guards, see (5) below.

$$\begin{array}{ll} Bad \triangleq & |(\;\mathsf{var}\; x : \mathsf{Real}_+ \to \mathsf{Real} \\ & \quad\bullet\; x :- (\lambda t \cdot t); \\ & \quad \mathsf{do}\; now > 1 \to x :- (\lambda t \cdot 0)\; \mathsf{od} \\ &)| \end{array} \tag{5}$$

The *future update* $x :- e$ is defined by

$$x :- e \triangleq x := x/now/e$$

$$x/t_0/e \triangleq (\lambda t \cdot \mathsf{if}\; t < t_0 \;\mathsf{then}\; x.t \;\mathsf{else}\; e.t \;\;\mathsf{fi})$$

For instance, consider the future update $clock :- (\lambda t \cdot t - now)$ in the example in Fig. (1). We have that

$$\begin{array}{rl} & clock :- (\lambda t \cdot t - now) \\ = & clock := clock/now/(\lambda t \cdot t - now) \\ = & clock := (\lambda t \cdot \textsf{if } t < now \textsf{ then } clock.t \textsf{ else } (\lambda t \cdot t - now).t \textsf{ fi}) \\ = & clock := (\lambda t \cdot \textsf{if } t < now \textsf{ then } clock.t \textsf{ else } t - now \textsf{ fi}) \end{array}$$

Thus, only the future behavior of *clock* is changed in this assignment. It is important to note that all the attributes of a continuous action system are functions of time, except for *now*. For instance, the statement $x :- (\lambda t \cdot t)$ updates the default future of x with an increasing function, while $x :- (\lambda t \cdot now)$ updates it with a constant function.

This explication of a continuous action system shows it essentially as a collection of time functions $x_1, \ldots, x_n$ over the non-negative reals, defined in a stepwise manner. The steps form a sequence of intervals $I_0, I_1, I_2, \ldots$, where each interval I_k is either a left closed interval of the form $[t_i, t_{i+1})$ or a closed interval of the form $[t_i, t_i]$, i.e., a point. The action system determines a family of functions $x_1, \ldots, x_n$ which are stepwise defined over this sequence of intervals and points. The extremes of these intervals correspond to the control points of the system where a discrete action is performed. In the example in Fig. (1), the sequence of intervals is $[0], [0,1), [1,2), [2,3), \ldots$

As such, the continuous action system is best understood as the limit of a sequence of approximations of the time functions $x_1, \ldots, x_n$, defined over successively longer and longer intervals $[0, t_i)$, where $i = 0, 1, 2, \ldots$. Looking at the example in Fig. (1) in this way, its sequence of initial segments is $[0], [0,1), [0,2), [0,3), \ldots$ and its defined approximations are successively

$$\begin{array}{rcl} x.t & = & 0, \quad 0 \le t \\ x.t & = & t, \quad 0 \le t \\ x.t & = & \begin{cases} t, & 0 \le t < 1, \\ 1 - t, & 1 \le t \end{cases} \\ x.t & = & \begin{cases} t, & 0 \le t < 1, \\ 1 - t, & 1 \le t < 2, \\ t - 2, & 2 \le t \end{cases} \\ & \vdots & \end{array}$$

Each attribute has a defined history of its past, i.e., on the interval $[0, now)$, its present value in the point $[now]$, and a default future. The execution of an action can modify the present value of a variable and its default future, but not its past. It is important to note that such a definition does not necessarily determine a single function for x_i. Because of the non-deterministic choices involved, there might be a collection of such function tuples that are allowed by the continuous action system, and we do not know which one of these will actually be the one the system follows. Thus, the system behavior may only be determined up to a certain tolerance, and any system behavior that is within these limits is possible.

3.2 Systems using differential equations

The behavior of a dynamic system is often described using a system of differential equations. We can allow this kind of definitions by introducing the shorthand

$$\dot{x} :- f \;\triangleq\; \{x :- y \mid y.now = x.now \wedge \dot{y} = f.y, y \geq now\}$$

This will assign to x a time function that satisfies the given differential equation and which is such that the function x is continuous at *now*. As an example, if $f = (\lambda t \cdot c)$, where c is a constant value, then we have that

$$\dot{x} :- (\lambda t \cdot c) \quad \triangleq \quad x :- (\lambda t \cdot x.now + c * (t - now))$$

Hence, we can use continuous action systems to express hybrid systems using either explicit functional expressions or implicit differential equations.

3.3 Real-time systems

We can use clock variables or timers to measure the passage of time and to correlate the execution of an action with the time. A *clock variable* is an attribute that measures the time elapsed since it was set to zero. Assume that c is an attribute of type Real. We then use the following definition for resetting the clock c:

$$reset(c) \;\triangleq\; c :- (\lambda t \cdot t - now)$$

This definition is just a convenience for correlating the behavior of our system with the passage of the time. Since a clock variable is just a regular attribute, we can define as many clocks as needed and reset them independently. It is also possible to do arithmetic operations with clock variables, to use time constrains as guards, or to refer to past values of an attribute. For instance, $x.(now - 1) = 1$ will hold one time unit after $x.now = 1$ holds. Hence, continuous action systems can be used to model real-time systems.

Summarizing, continuous action systems represent a new model for hybrid computation that is general enough to be used for several different purposes. This approach can model both continuous and discrete state-based systems, systems whose behavior can be determined up to a certain tolerance, dynamic systems, as well as real-time systems. The following section presents two examples in more detail.

```
$\mathcal{T}rain \triangleq$ |(  var $d, v : \mathsf{Real}_+ \rightarrow \mathsf{Real}, v', d' : \mathsf{Real}_+ \rightarrow \mathsf{Real}$
          $state : \mathsf{Real}_+ \rightarrow \{accelerating, cruising, decelerating, stopped\}$
          • $d :- (\lambda t \cdot \frac{a * t^2}{2})\,;\, v :- (\lambda t \cdot a * t);$
          $\{v' :- v'' | v' - \varepsilon \leq v'' \leq v' + \varepsilon\};$
          $state :- accelerating;$
        do $state.now = accelerating \wedge v.now = v' \rightarrow$
          $v :- (\lambda t \cdot v');$
          $d :- (\lambda t \cdot d.now + v * (t - now));$
          $\{d' :- d'' | d' - \varepsilon' \leq d'' \leq d' + \varepsilon'\};$
          $state :- cruising$
        $\Box$ $state.now = cruising \wedge d.now = d' \rightarrow$
          $v :- (\lambda t \cdot v' - a * (t - now));$
          $d :- (\lambda t \cdot d.now + v' * (t - now) - \frac{a}{2} * (t - now)^2);$
          $state :- decelerating$
        $\Box$ $state.now = decelerating \wedge v.now = 0 \rightarrow$
          $d :- (\lambda t \cdot d.now);$
          $v :- (\lambda t \cdot 0);$
          $state :- stopped$
        od
      )|
```

Fig. 2: Train movement as a continuous action system.

4. Example Models

In this section we show how simple hybrid systems can be described as continuous action systems. We model the movement of a train as well as a press that reacts to external signals from the environment. A more complex case study using continuous action systems can be found in [3].

4.1 Train

The train example has been presented before in [15], using differential actions. The approach taken here, based on continuous action systems, has an explicit expression of the functions describing the continuous behavior.

The train is initially in position 0, having the velocity 0 and it starts moving, by speeding up with acceleration a until it reaches the velocity v'. From this point on, the train keeps going with this constant speed v'. When it reaches the position d', it starts decelerating with acceleration $-a$, until it reaches velocity 0 and stops. The actual velocity and acceleration have the tolerances ε and ε', respectively. In Fig. 2 we specify the movement of the train as a continuous action system. We use the convention that an assignment $x :- c$, where x is a function $Real_+ \rightarrow T$ and c if of type T, stands for $x :- (\lambda t \cdot c)$ (i.e., the pointwise extended constant function, rather than the constant itself, is assigned to x).

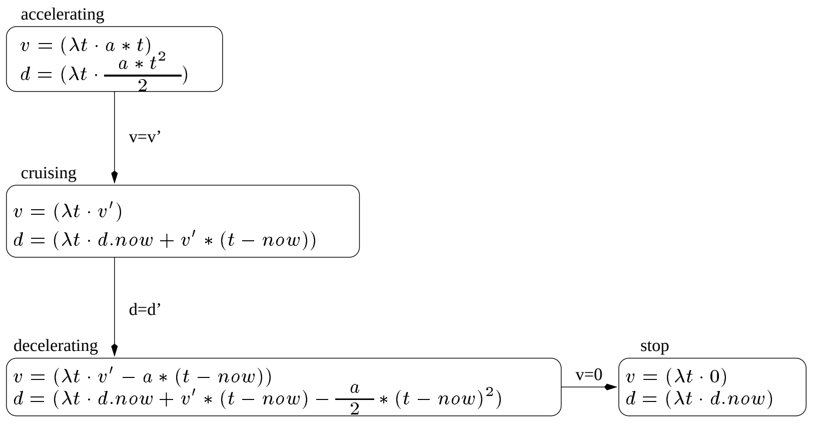

accelerating
$v = (\lambda t \cdot a * t)$
$d = (\lambda t \cdot \frac{a * t^2}{2})$
v=v'
cruising
$v = (\lambda t \cdot v')$
$d = (\lambda t \cdot d.now + v' * (t - now))$
d=d'
decelerating
$v = (\lambda t \cdot v' - a * (t - now))$
$d = (\lambda t \cdot d.now + v' * (t - now) - \frac{a}{2} * (t - now)^2)$
v=0
stop
$v = (\lambda t \cdot 0)$
$d = (\lambda t \cdot d.now)$

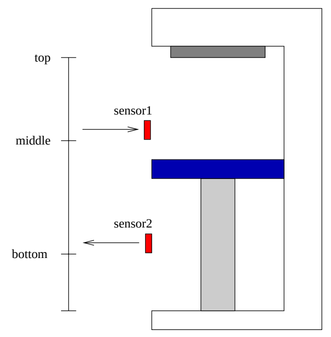
Fig. 3: State diagram for the $\mathcal{T}rain$ system.

Fig. 3 shows another description of the train system, in terms of a state diagram. This kind of description is also known as a *hybrid automata* [2]. Here the states are associated with the continuously evolving behavior, while the transitions are labeled with the guards.

The train model is sequential, as shown by Fig. 3, but non-deterministic. A change of behavior is performed when a certain position or velocity is reached, i.e., when a guard depending on that position or velocity holds. However, that position or velocity is non-deterministically computed in the previous state. The attributes v' and d' are local. They are modified when the train is in the previous state to an approximation of their initial value. Thus, the system changes its time behavior based on deterministic guards, but non-determinism is achieved through non-deterministic assignments to attributes before they are tested in guards.

4.2 Press

Our second example concerns a system that reacts to external signals produced by the environment. Fig. 4 shows a model of a press from a metal processing factory [12]. The press works as follows. First, its lower part is raised to the middle position. Then an upper conveyor belt feeds a metal blank into the press. When the press is loaded (signaled by $sensor_1$ being true), the lower part of the press is raised to the top position and the blank is forged. The press will then move down until the bottom position and the forged blank is placed into a lower conveyor belt. When the press is unloaded (signaled by $sensor_2$ being true), the lower part is raised to the middle position, ready for being loaded again. The press works cyclically and keeps evolving from one phase to another. We model these phases with a *task* attribute in the continuous action system *Press* shown in Fig. 4. This attribute has the discrete values *loading*, *pressing*, *moving2unload*, *unloading*, *moving2load*. The continuous attribute p shows the position of the press plate and is, at different moments in time, a linearly increasing, a linearly decreasing or a constant function of time. The positions of reference for the press, i.e. *bottom*, *middle*, and *top*, are given as parameters.



```
$\mathcal{P}ress \triangleq$ |(  var $p, c : \mathsf{Real}_+ \to \mathsf{Real}$,
              $task : \mathsf{Real}_+ \to \{loading, pressing, moving2unload,$
                                   $unloading, moving2load\}$
            • $reset(c)\,; p :- middle\,; task :- loading;$
          do $task.now = loading \wedge sensor_1.now \to$
              $reset(c);$
              $p :- (\lambda t \cdot middle + v * c.t);$
              $task :- pressing$
           □ $task.now = pressing \wedge p.now = top \to$
              $reset(c);$
              $p :- (\lambda t \cdot top - v * c.t);$
              $task :- moving2unload$
           □ $task.now = moving2unload \wedge p.now = bottom \to$
              $p :- (\lambda t \cdot bottom);$
              $task :- unloading$
           □ $task.now = unloading \wedge \neg sensor_2.now \to$
              $reset(c);$
              $p :- (\lambda t \cdot bottom + v * c.t);$
              $task :- moving2load$
           □ $task.now = moving2load \wedge p.now = middle \to$
              $p :- middle;$
              $task :- loading$
          od
        )|    $: sensor_1, sensor_2$
```

Fig. 4: Press functioning as a continuous action system.

The press is a typical part of a control system. Control systems are essentially composed from several components that work together in order to meet the requirements of the overall system. An important feature of a component consists in its interaction with the environment. For the press, this interaction with the environment (the conveyor belts) is modeled with several sensors, modeled by imported boolean attributes that can be changed by the environment at any time. It can be seen here how an imported attribute reflects the evolution of another component of a complex system.

5. Safety Properties

Properties of continuous action systems can be established by proving that these properties hold for the corresponding discrete action systems. Hence, there is no special proof theory for continuous action systems, but the standard proof theory for action systems suffices (with the exception that we exclude Zeno behavior and ill-defined systems). In this paper, we concentrate on safety properties since in many cases they are essential properties that we want to establish initially for hybrid systems.

A common characterization for a safety property is that nothing 'bad' happens during the life time of the system. Put in another way, a safety property S is a 'good' property G that always holds, i.e., $S = (\forall t \geq 0 \cdot G.t)$. We can establish S for the action system $\mathcal{C}$ in (2), by proving that a property

$$I \quad \stackrel{\wedge}{=} \quad (\forall t \mid 0 \leq t < now \cdot G'.t)$$

is an invariant of the corresponding discrete action system $\bar{\mathcal{C}}$, where $(\forall t \geq 0 \cdot G'.t \Rightarrow G.t)$. This implies the safety property, provided that the system does not have a Zeno behavior (i.e., *now* will go to infinity in the system). The safety property S holds when the system is started in an initial state satisfying P, if and only if the following conditions are satisfied for $\bar{\mathcal{C}}$:

$$\begin{aligned} (\forall t \geq 0 \cdot G'.t \Rightarrow G.t) \\ P \Rightarrow wp(now := 0 \,;\, S_0 \,;\, N, I) \\ I \wedge g_i \Rightarrow wp(S_i \,;\, N, I), \quad i = 1, \ldots, m \end{aligned}$$

Consider the press example of section 4.2 and the following two safety properties. First, we want to prove that the movable plate of the press does not pass the limits of the machine. Formally this is expressed by $(\forall t \geq 0 \cdot bottom \leq p.t \leq top)$, where p is the vertical position of the plate. In the proofs, we assume that $bottom \leq middle \leq top$. Second, we also want to prove that p is a continuous function on Real_+. We need to choose an invariant I that allows us to establish the safety property $(\forall t \geq 0 \cdot bottom \leq p.t \leq top) \wedge (p\ continuous\ on\ \mathsf{Real}_+)$ using the proof rule above.

For the first conjunct of the safety property, an invariant of the form $(\forall t \mid 0 \leq t \leq now \cdot bottom \leq p.t \leq top)$ would be sufficient. However, to prove the global continuity property, we need a stronger invariant, which also ensures that the press remains in the correct position during the loading and unloading operations. The following invariant I is sufficient for establishing the required safety property:

$$\begin{aligned} I \;\; \overset{\triangle}{=} \;\; & (p \; continuous \; on \; [0, now] \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot bottom \leq p.t \leq top) \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\ & (\forall t \mid 0 \leq t \leq now \cdot task.t = unloading \Rightarrow p.t = bottom)) \end{aligned}$$

The first of the three conditions above is obviously discharged. Hence, the proof must establish that the invariant is satisfied by the initialization from the moment 0 until the first moment an action is enabled, and during the time elapsed between the execution of two actions. Discharging the second of the three conditions above, as well as the third condition for $i = 1$ is demonstrated in Appendix A. Proofs of a more complex model, using differential equations, are shown in [3].

6. Composing continuous action systems

The previous examples only describe isolated systems. For modeling complex systems, where several different subsystems or components evolve concurrently, we need to formally define the composition of continuous action systems. We define the parallel composition of two continuous action systems similarly with the parallel composition of discrete action systems in Section 2. For parallel composition, we may also need to hide certain attributes of the system when describing more complex systems, but we ignore this aspect here for brevity.

Thus, if we have the continuous action systems $\mathcal{C}$ and $\mathcal{C}'$ below:

$$\begin{aligned} \mathcal{C} &= |(\mathsf{var}\ x : \mathsf{Real}_+ \to T \bullet S_0 \,;\, \mathsf{do}\ g_1 \to S_1 \Box \ \ldots \Box \ g_m \to S_m \ \mathsf{od}\ )| : y \\ \mathcal{C}' &= |(\mathsf{var}\ x' : \mathsf{Real}_+ \to T' \bullet S_0' \,;\, \mathsf{do}\ g_1' \to S_1' \Box \ \ldots \Box \ g_n' \to S_n' \ \mathsf{od}\ )| : y' \end{aligned}$$

then their parallel composition is the continuous action system $\mathcal{C} \parallel \mathcal{C}'$, defined by

$$\begin{aligned} \mathcal{C} \parallel \mathcal{C}' \overset{\triangle}{=} |(\;\; & \mathsf{var}\ x : \mathsf{Real}_+ \to T, x' : \mathsf{Real}_+ \to T \bullet S_0 \,;\, S_0'; \\ & \mathsf{do}\ g_1 \to S_1 \Box \ \ldots \Box \ g_m \to S_m \Box \ g_1' \to S_1' \Box \ \ldots \Box \ g_n' \to S_n' \ \mathsf{od} \\ )| &: (y \cup y') - (z \cup z') \end{aligned}$$

We assume here that x and x' are disjoint. We need to combine the continuous action systems before we translate them into discrete action systems, because the local attribute *now* would appear in both $\bar{\mathcal{C}}$ and $\bar{\mathcal{C}}'$. By combining the continuous action systems first, we ensure that the combined system uses a single *now* attribute, which is checked by actions from both components.

Parallel composition essentially combines the attributes of the two component systems and, therefore, their continuous evolution. As the actions in the parallel composition are the combined actions of the two systems, discrete changes will usually occur more frequently. An action in one system may depend on an attribute in the other system, which may be again modified by actions of the former system. This means that the behavior of a system in a parallel composition is usually different from the behavior of the system standing alone.

We exemplify parallel composition by extending the press model from Section 4.2 with two conveyor belts for feeding and removing the blanks into and from the press. These conveyor belts are operated by corresponding engines. If there are no blanks over a belt, its engine should stop. The movements of the two belts are independent of each other and take place at constant speeds Θ_1 and Θ_2.

The press and the belts share four sensors, which become active when a blank crosses them. We assume that $sensor_3$ and $sensor_2$ hold when a blank enters them while $sensor_1$ and $sensor_4$ hold when a blank leaves them. We assume that the sensors are always deactivated between two consecutive pieces. This can be ensured by restricting the minimum distance between blanks. The controller of a conveyor belt keeps track of how many pieces are on the belt, so it can disconnect the engine when it is empty. The formal specification of the top conveyor belt is as follows:

```
TopBelt  ≜  |(  var $items : \mathsf{Real}_+ \to \mathsf{Int}$,
                    $speed : \mathsf{Real}_+ \to \mathsf{Real}$
                  • $items, speed :- 0, 0$;
                do $sensor_3.now \to items :- items + 1$
                □ $sensor_1.now \to items :- items - 1$
                □ $speed.now \neq 0 \wedge items.now = 0 \to speed :- 0$
                □ $speed.now = 0 \wedge items.now \neq 0 \to speed :- \Theta_1$
                od
            )| : $sensor_1, sensor_3$
```

The specification of the bottom conveyor belt is similar, using $sensor_2$ and $sensor_4$ instead of $sensor_1$ and $sensor_3$. The complete model of the system consists of the parallel composition of the press with the top and the bottom conveyor belts: $\mathcal{F}actory = \mathcal{P}ress \parallel \mathcal{T}op\mathcal{B}elt \parallel \mathcal{B}ottom\mathcal{B}elt$.

7. Conclusions and Related Work

In this paper we have shown how to extend the action systems framework for modeling hybrid systems, by introducing the notion of a continuous action system. The attributes in continuous action systems are functions over time and they are updated in a way that only changes their present and future behavior. Essentially, this amounts to extending the notion of state with both a history and a default future, thus generalizing the classical action systems approach that only handles the present state.

This extension allows us to model systems that combine discrete control with continuous behavior, the latter either defined by explicit functions of time or by differential equations. We have shown how to prove safety properties of continuous action systems using the classical invariant method. We have also shown that the continuous action system model provides a simple way of defining the parallel composition of hybrid systems, using communication by means of shared variables. Continuous action systems are general and can be used to model dynamic, real-time and tolerant systems, as well as complex control systems, see for example [3].

The idea of extending an existing formalism to model real-time systems by introducing a variable representing the time was already presented by Abadi and Lamport in [1]. We follow the same approach here, extending an existing formalism to handle hybrid systems, instead of creating a new formalism specific for such systems. This provides a clear advantage since now we can apply all the previous results on action systems to study real-time and hybrid models.

Rönkkö and Ravn [14] have already proposed a model for combining action systems and continuous behavior, called *hybrid action system*. This approach is not intended for modeling real-time systems and has no implicit notion of time. The continuous evolution of a variable is modeled as a special kind of an atomic *differential* action. A differential action cannot be interrupted and its bounds are specified in advance. This affects the parallel composition of systems, since different simultaneous actions must be combined into a sequence of atomic actions. In the worst case, the parallel composition of two systems with n and m actions leads to a system with $n*m$ actions. In our model, parallel composition of two such systems gives a continuous action system with $n+m$ actions. This is a major simplification for handling large systems.

These advantages still exist when comparing our formalism with the *hybrid automata* [2]. The number of states in the parallel composition of two hybrid automata is also the product of the number of states of the original automata. Moreover, the continuous action system formalism is more expressive than hybrid automata: it allows explicit failures of the system (modeled with the abort statement), as well as references to historical values of the attributes in guards and expressions. Compared to hybrid automata, our model also allows the attributes to be selectively updated: only those attributes that are changed need to be mentioned in an action. There are still some other important differences between our formalism and the hybrid automata. In the hybrid automata formalism, transitions are fired synchronously, while in the action system formalism actions are selected and executed asynchronously. Continuous action systems actions are urgent, i.e., they are executed as soon as they are enabled. There is no notion of location and location invariant related to continuous action systems, i.e., there is no relation between the guard of an action and the attributes that it updates.

Another interesting model for hybrid systems is provided by the *phase transition systems* [11]. In this model, the continuous behavior of the system is modeled using a finite set of activities, from which only one can be enabled at a certain time. Thus, a single activity completely defines the continuous behavior of a system at a given time. Compared to this model, our approach allows attributes to be selectively updated.

Continuous action systems model of specification is currently used for synthesizing complex control systems. The next step in the development of this formalism is to illustrate the stepwise refinement concept. This would provide for the derivation of executable control programs that are correct with respect to their specification, given in terms of continuous actions systems.

Acknowledgment We would like to thank Cristina Cerschi, Mauno Rönkkö and Hannu Toivonen for our inspiring discussions as well as the anonymous referees for their useful comments on the topics covered in this paper.

References

[1] Abadi, M. and Lamport, L. 1994. An Old-Fashioned Receipe for Real Time. *ACM Transactions on Programming Languages and Systems 16*, 5 (September), 1543–1571.

[2] Alur, R., Courcoubetis, C., Henzinger, T.A., and Ho, P.H. 1993. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I*, Number 736 in LNCS. Springer-Verlag, 209–229.

[3] Back, R. J. and Cerschi, C. 2000. Modeling and Verifying a Temperature Control System Using Hybrid Action Systems. In *Proc. of the 5th Int. Workshop in Formal Methods for Industrial Critical Systems.*

[4] Back, R. J. and Kurki-Suonio, R. 1983. Decentralization of Process Nets with Centralized Control. In *2nd Symp. on Principles of Distributed Computing*, Number 873 in LNCS. ACM SIGACT-SIGOPS, 131–142.

[5] Back, R. J., Petre, L., and Porres, I. 2000. Generalizing Action Systems to Hybrid Systems. In *6th International Symposium in Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2000)*, Number 1926 in LNCS. Springer-Verlag, 202–213.

[6] Back, R. J. and Sere, K. 1991. Stepwise Refinement of Parallel Algorithms. In *Science of Computer Programming 13*, 133–180.

[7] Back, R. J. and Sere, K. 1994. From Action Systems to Modular Systems. In *Formal Methods Europe (FME '94)*, Number 873 in LNCS. Springer-Verlag, 1–25.

[8] Back, R. J. and von Wright, J. 1998. *Refinement Calculus - A Systematic Introduction*. Springer-Verlag.

[9] Branicky, M.S. 1996. General hybrid dynamical systems: modeling, analysis and control. In *Hybrid Systems III*, Number 1066 in LNCS. Springer-Verlag, 186–200.

[10] Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice-Hall International.

[11] Kesten, Y., Manna, Z., and Pnueli, A. 1998. Verification of Clocked and Hybrid Systems. In *Lectures on Embedded Systems*. Number 1494 in LNCS. Springer Verlag, 4–73.

[12] Lewerentz, C. and Lindner, T. 1995. *Formal Development of Reactive Systems: Case Study Production Cell.*. Number 891 in LNCS. Springer-Verlag.

[13] Nerode, A. and Kohn, W. 1993. Models for Hybrid Systems: automata, topologies, controllability, observability. In *Hybrid Systems I*, Number 736 in LNCS. Springer-Verlag, 317–356.

[14] Rönkkö, M. and Ravn, A.P. 1999. Action Systems with Continuous Behaviour. In *Proceedings of the Hybrid Systems V*, Number 1567 in LNCS. Springer-Verlag, 304–323.

[15] Rönkkö, M. and Sere, K. 1999. Refinement and Continuous Behaviour. In *Hybrid Systems: Computation and Control*, Number 1569 in LNCS. Springer-Verlag, 223–237.

Appendix A. Proofs

We want to ensure that the model of the press always preserves the invariant described in Section 5. We first prove below that the initialization statement establishes this invariant.

$wp(now := 0\,;reset(c)\,;p :- (\lambda t \cdot middle)\,;task :- loading\,;now := next.gg.now, I)$

$\equiv$ { wp rule for seq. comp. and assig., see (4) for the formula of $next.gg.now$}

$$\begin{array}{l}
wp(now := 0\,;reset(c)\,;p :- (\lambda t \cdot middle)\,;task :- loading,\\
\quad (\forall now' \mid now' = min\{t' \geq now| \; (task.t' = loading \wedge sensor_1.t') \vee\\
\qquad\qquad (task.t' = pressing \wedge p.t' = top) \vee\\
\qquad\qquad (task.t' = moving2unload \wedge p.t' = bottom) \vee\\
\qquad\qquad (task.t' = unloading \wedge sensor_2.t') \vee\\
\qquad\qquad (task.t' = moving2load \wedge p.t' = middle)\\
\qquad \} \cdot\\
\quad p\; continuous\; on\; [0, now'] \wedge\\
\quad (\forall t|0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge\\
\quad (\forall t|0 \leq t \leq now' \cdot task.t = loading \Rightarrow p.t = middle) \wedge\\
\quad (\forall t|0 \leq t \leq now' \cdot task.t = unloading \Rightarrow p.t = bottom))
\end{array}$$

$\equiv$ {wp rule for seq. comp. and assig., future update, splitting ranges, predicate calc.}

$$\begin{array}{l}
wp(now := 0\,;reset(c)\,;p :- (\lambda t \cdot middle),\\
\quad (\forall now' \mid now' = min\{t' \geq now| \; (loading = loading \wedge sensor_1.t') \vee\\
\qquad\qquad (loading = pressing \wedge p.t' = top) \vee\\
\qquad\qquad (loading = moving2unload \wedge p.t' = bottom) \vee\\
\qquad\qquad (loading = unloading \wedge sensor_2.t') \vee\\
\qquad\qquad (loading = moving2load \wedge p.t' = middle)\\
\qquad \} \cdot\\
\quad p\; continuous\; on\; [0, now'] \wedge\\
\quad (\forall t|0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge\\
\quad (\forall t|0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge\\
\quad (\forall t|now \leq t \leq now' \cdot loading = loading \Rightarrow p.t = middle) \wedge\\
\quad (\forall t|0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom) \wedge\\
\quad (\forall t|now \leq t \leq now' \cdot loading = unloading \Rightarrow p.t = bottom)))
\end{array}$$

$\equiv$ {predicate calc.}

$$\begin{array}{l}
wp(now := 0\,;reset(c)\,;p :- (\lambda t \cdot middle),\\
\quad (\forall now' \mid now' = min\{t' \geq now|sensor_1.t'\} \cdot\\
\quad p\; continuous\; on\; [0, now'] \wedge\\
\quad (\forall t|0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge\\
\quad (\forall t|0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge\\
\quad (\forall t|now \leq t \leq now' \cdot p.t = middle) \wedge\\
\quad (\forall t|0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom)))
\end{array}$$

$\equiv$ {wp rule for seq. comp. and assig., future update}

$$\begin{array}{l}
wp(now := 0\,;reset(c),\\
\quad (\forall now' \mid now' = min\{t' \geq now|sensor_1.t'\} \cdot\\
\quad p/now/middle\; continuous\; on\; [0, now'] \wedge\\
\quad (\forall t|0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge\\
\quad (\forall t|now \leq t \leq now' \cdot bottom \leq middle \leq top) \wedge\\
\quad (\forall t|0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge\\
\quad (\forall t|now \leq t \leq now' \cdot middle = middle) \wedge\\
\quad (0 \leq t < now \wedge task.t = unloading \Rightarrow p.t = bottom)))
\end{array}$$

$\equiv$ {$bottom \leq middle \leq top$ is an assumpt. of the model, predicate calc. }

$$
\begin{array}{ll}
& wp(now := 0\,;\,reset(c), \\
& \quad (\forall now' \mid now' = min\{t' \geq now | sensor_1.t'\} \cdot \\
& \quad p/now/middle\ continuous\ on\ [0, now'] \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom))) \\
\equiv & \{wp \text{ rule for seq. comp., future update}\} \\
& wp(now := 0, wp(reset(c), \\
& \quad (\forall now' \mid now' = min\{t' \geq now | sensor_1.t'\} \cdot \\
& \quad (\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \mathsf{else}\ middle\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\
& \quad (\forall t | 0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom)))) \\
\equiv & \{c \text{ does not appear in the postcond., } wp \text{ rule for assig. applied twice}\} \\
& (now' = min\{t' \geq 0 | sensor_1.t'\} \cdot \\
& \quad (\lambda t \cdot \mathsf{if}\ t < 0\ \mathsf{then}\ p.t\ \mathsf{else}\ middle\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge \\
& \quad (\forall t | 0 \leq t < 0 \cdot bottom \leq p.t \leq top) \wedge \\
& \quad (\forall t | 0 \leq t < 0 \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\
& \quad (\forall t | 0 \leq t < 0 \cdot task.t = unloading \Rightarrow p.t = bottom)) \\
\equiv & \{\text{assume } t < 0 \equiv \mathsf{F}, \text{ predicate calc.}\} \\
& (\forall now' \mid now' = min\{t' \geq 0 | sensor_1.t'\} \cdot \\
& \quad (\lambda t \cdot middle)\ continuous\ on\ [0, now']) \\
\equiv & \{(\lambda t \cdot middle) \text{ is a ct. funct., so it is continuous}\} \\
& \mathsf{T}
\end{array}
$$

The last step in the proof above assumes that the minimum exists ($sensor_1$ will eventually be on).Below we consider whether the system behaves correctly after the execution of the *first* action.

$$
\begin{array}{ll}
& p\ continuous\ on\ [0, now] \wedge \\
& (\forall t | 0 \leq t \leq now \cdot bottom \leq p.t \leq top) \wedge \\
& (\forall t | 0 \leq t \leq now \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\
& (\forall t | 0 \leq t \leq now \cdot task.t = unloading \Rightarrow p.t = bottom) \wedge \\
& task.now = loading \wedge sensor_1.now \\
\vdash & wp(reset(c)\,;\,p :- (\lambda t \cdot middle + v * c.t)\,;\,task :- pressing\,;\,now := next.gg.now, I) \\
\equiv & \{wp \text{ rule for seq. comp.(twice), } wp \text{ rule for assig., see (4) for } next.gg.now\} \\
& wp(reset(c)\,;\,p :- (\lambda t \cdot middle + v * c.t), wp(task :- pressing, \\
& \quad (\forall now' | now' = min\{t' \geq now | \ (task.t' = loading \wedge sensor_1.t') \vee \\
& \qquad\qquad (task.t' = pressing \wedge p.t' = top) \vee \\
& \qquad\qquad (task.t' = moving2unload \wedge p.t' = bottom) \vee \\
& \qquad\qquad (task.t' = unloading \wedge sensor_2.t') \vee \\
& \qquad\qquad (task.t' = moving2load \wedge p.t' = middle)) \\
& \qquad \} \cdot \\
& \quad p\ continuous\ on\ [0, now'] \wedge \\
& \quad (\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge \\
& \quad (\forall t | 0 \leq t \leq now' \cdot task.t = loading \Rightarrow p.t = middle) \wedge \\
& \quad (\forall t | 0 \leq t \leq now' \cdot task.t = unloading \Rightarrow p.t = bottom))) \\
\equiv & \{wp \text{ rule for seq. comp. and assig., future update, splitting ranges, predicate calc.}\} \\
& wp(reset(c)\,;\,p :- (\lambda t \cdot middle + v * c.t), \\
& \quad (\forall now' | now' = min\{t' \geq now | \ (pressing = loading \wedge sensor_1.t') \vee \\
& \qquad\qquad (pressing = pressing \wedge p.t' = top) \vee \\
& \qquad\qquad (pressing = moving2unload \wedge p.t' = bottom) \vee \\
& \qquad\qquad (pressing = unloading \wedge sensor_2.t') \vee \\
& \qquad\qquad (pressing = moving2load \wedge p.t' = middle)) \\
& \qquad \} \cdot
\end{array}
$$

$p\ continuous\ on\ [0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | now \leq t \leq now' \cdot pressing = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom) \wedge$
$(\forall t | now \leq t \leq now' \cdot pressing = unloading \Rightarrow p.t = bottom))$

$\equiv$ {predicate calc., wp rule for seq. comp.}
$wp(reset(c), wp(p :- (\lambda t \cdot middle + v * c.t),$
$(\forall now' | now' = min\{t' \geq now | p.t' = top\}) \cdot$
$p\ continuous\ on\ [0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | 0 \leq t < now \cdot task.t = loading \Rightarrow p.t = middle) \wedge$
$(\forall t | 0 \leq t < now \cdot task.t = unloading \Rightarrow p.t = bottom))$

$\equiv$ {invariant assumpt.}
$wp(reset(c), wp(p :- (\lambda t \cdot middle + v * c.t),$
$(\forall now' | now' = min\{t' \geq now | p.t' = top\}) \cdot$
$p\ continuous\ on\ [0, now'] \wedge$
$(\forall t | 0 \leq t \leq now' \cdot bottom \leq p.t \leq top)))$

$\equiv$ {future update}
$wp(reset(c), (\forall now' | now' = min\{t' \geq now | middle + v * c.t' = top\} \cdot$
$p/now/(\lambda t \cdot middle + v * c.t)\ continuous\ on\ [0, now'] \wedge$
$(\forall t | 0 \leq t < now \cdot bottom \leq p.t \leq top) \wedge$
$(\forall t | now \leq t < now' \cdot bottom \leq p.t \leq top)))$

$\equiv$ {invariant assumpt.}
$wp(reset(c), (\forall now' | now' = min\{t' \geq now | middle + v * c.t' = top\} \cdot$
$p/now/(\lambda t \cdot middle + v * c.t)\ continuous\ on\ [0, now'] \wedge$
$(\forall t | now \leq t < now' \cdot bottom \leq p.t \leq top)))$

$\equiv$ {$reset(c) = (c :- (\lambda t \cdot t - now))$, wp rule for assig., future update}
$(\forall now' | now' = min\{t' \geq now | middle + v * (t' - now) = top\} \cdot$
$(\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \ \mathsf{else}\ middle + v * (t - now)\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge$
$(\forall t | now \leq t \leq now' \cdot bottom \leq middle + v * (t - now) \leq top))$

$\equiv$ {computing min function}

- $min\{t' \geq now | middle + v * (t' - now) = top\}$
  $=$ {assumption $top > middle > 0$, $v > 0$}
  $$now + \frac{top - middle}{v}$$

$\cdot\ (\forall now' | now' = now + \dfrac{top - middle}{v}\ \cdot$
$(\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \ \mathsf{else}\ middle + v * (t - now)\ \mathsf{fi})\ continuous\ on\ [0, now'] \wedge$
$(\forall t | now \leq t \leq now' \cdot bottom \leq middle + v * (t - now) \leq top))$

$\equiv$ {1-point rule}
$(\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \ \mathsf{else}\ middle + v * (t - now)\ \mathsf{fi})$
$$continuous\ on\ [0, now + \frac{top - middle}{v}] \wedge$$
$$(\forall t | now \leq t \leq now + \frac{top - middle}{v} \cdot bottom \leq middle + v * (t - now) \leq top)$$

$\equiv$ {assumpt. $v > 0, bottom < middle < top$, real analysis, properties of T}
$(\lambda t \cdot \mathsf{if}\ t < now\ \mathsf{then}\ p.t\ \ \mathsf{else}\ middle + v * (t - now)\ \mathsf{fi})$
$$continuous\ on\ [0, now + \frac{top - middle}{v}]$$

$\equiv$ {p cont. on $[0, now], p.now = middle,\ lim_{t \rightarrow now^-} p.t = middle = p.now$}
T

We have proved that the first action preserves the invariant of the system. The proofs for the other actions follow a similar pattern as the proofs presented so far.