Generalizing Action Systems to Hybrid Systems

R.-J. Back, L. Petre and I. Porres

Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
{Ralph-Johan.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 extend the action system approach to hybrid systems by defining *continuous action systems*. These are systems with discrete control over a continuously evolving state, whose semantics is defined in terms of traditional action systems. We show that continuous action systems are very general and can be used to describe a diverse range of hybrid systems. Moreover, the properties of continuous action systems are proved using standard action systems proof techniques.

1 Introduction

A system using discrete control over 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]). 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 systems* paradigm.

Action systems [4] have been used successfully to model discrete systems, i.e., systems that use a 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 model 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 variables will range over functions over time, rather than just over values. This allows a variable to capture not only its present value, but also the whole history of values that the variable has had, as well as the default future values that the variable will receive. Updating a state variable is restricted so that only the future behavior of the variable 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 [14].

Proofs about action system properties are based on the refinement calculus [7]. This extends the programming logic based on weakest precondition predicate transformers that was proposed in [10]. Action systems are intended to be stepwise developed, the correctness of these refinement steps being verified within the refinement calculus. Thereby, we get an implicit notion of refinement also for continuous action systems. 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 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 $R : A \rightarrow \mathcal{P}B$, 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. We define the continuous action systems in Section 3. Their semantics is specified by explaining how to translate them into ordinary action systems. 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. Conclusions and comparisons to related work are presented in Section 6.

2 Action Systems

We start by giving a brief overview of the action systems formalism. An action system is essentially a *discrete state space* updated by a *discrete control* mechanism. The state of the system is described using *attributes* or *program variables.* 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 $x : \mathit{Val}$. We 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 $\{\mathsf{F}, \mathsf{T}\}$.

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. This set of actions models the *control mechanism* over the state of the system. An action system $\mathcal{A}$ has the following form:

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

Here $x : \mathit{Val} = x_1 : \mathit{Val}_1, \ldots, x_n : \mathit{Val}_n$ are the *local attributes* of the system, S_0 is a statement that initializes the attributes, while '$A_i = g_i \rightarrow S_i$', $i = 1, \ldots, m$, 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 the action system and 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 A of the form '$g \rightarrow S$' is a guarded statement that can be executed only when g is enabled, i.e., when g evaluates to T. The body S of an action is defined by the following syntax:

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

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

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

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

The execution of an action system is as follows. The initialization S_0 will set the attributes x 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 chosen for execution, in a demonically non-deterministic way. The computation terminates when no action is enabled. Actions systems model parallel execution by interleaving atomic actions in a demonically non-deterministic fashion.

In the following, 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 mechanism over a 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 together with a finite set of actions that act upon them. The attributes can range over discrete or continuous domains and form the state of the system. A continuous action system is of the form:

$$\mathcal{C} \triangleq |(\mathsf{var}\ x : \mathsf{Real}_+ \to Val \bullet S_0\ ; \mathsf{do}\ g_1 \to S_1 \square\ \ldots \square\ g_m \to S_m \mathsf{od}\ )| : y \qquad (1)$$

Intuitively, executing a continuous action system proceeds as follows. There is an implicit variable *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.

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 nondeterministic 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 future behavior. 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. 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 does not need 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 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 the system will continue to evolve forever according to the functions last assigned to the attributes.

As an example of a continuous action system consider the system in Fig. 1. 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 increase 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 T in all the future time instances. After this, the system starts to evolve by advancing time continuously. In particular, the value of x increases linearly, depending on time. When x gets value 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 value of *up* is set to the constant F. This continues until x reaches 0, when the first action is again enabled, changing x to increase again, and so on. The effect of these two actions is a sawtooth-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

```
Saw ≜ |( var x, clock : Real₊ → Real ; up : Real₊ → Bool
       • x :− (λt · 0) ; clock :− (λt · 0) ; up :− (λt · F);
      do x.now = 0 ∧ ¬up.now →
            clock :− (λt · t − now);
            x :− clock ; up :− (λt · T)
       □ x.now = 1 ∧ up.now →
            clock :− (λt · t − now);
            x :− (λt · 1 − clock.t);
            up :− (λt · F)
      od
    )|
```

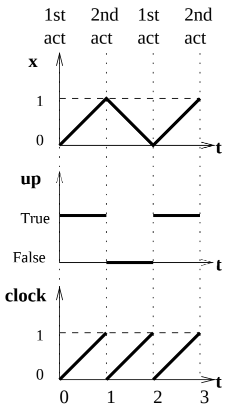


Fig. 1. Continuous action system *Saw* (left) and its behavior (right).

in time where a discrete action is performed. We see that a continuous action system is just a non-deterministic way of defining a collection of time dependent functions. One of the main advantages of this model for hybrid computation is that both discrete and continuous behavior can be described in the same way. In particular, if the attributes are assigned only constant functions, we obtain a discrete computation.

Semantics of continuous action systems. Let $\mathcal{C}$ be the continuous action system in (1). We explain the meaning of $\mathcal{C}$ by translating it into an ordinary action system. Its semantical interpretation is given by the following (discrete) action system $\bar{\mathcal{C}}$:

$$\begin{array}{rll} \bar{\mathcal{C}} \;\triangleq\; |[& \mathsf{var} & now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to Val \\ & \bullet & now := 0 \,;\, S_0 \,;\, N; \\ & \mathsf{do} & g_1 \to S_1 \,;\, N \Box \;\ldots \Box\; g_m \to S_m \,;\, N \;\mathsf{od} \\ ]| : y & & \end{array} \tag{2}$$

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

$$\mathsf{next}.gg.t \;\triangleq\; \begin{cases} min\{t' \geq t \;\mid\; gg.t'\}, & \text{if } \exists\; t' \geq t \text{ such that } gg.t' \\ t, & \text{otherwise.} \end{cases} \tag{3}$$

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. 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 and the attributes will evolve forever according to the functions last assigned. We assume in this paper 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 that do not satisfy this requirement are considered ill-defined.

The *future update* $x :- e$ is defined by $x :- e \;\hat{=}\; x := x/now/e$ where $x/t_0/e \;\hat{=}\; (\lambda t \cdot \text{if } t < t_0 \text{ then } x.t \text{ else } e.t \text{ fi})$. Thus, only the future behavior of x is changed by the future update. It is important to note that all the attributes of a continuous action system are functions of time, except for *now*. As an example, 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. We write $x :- c$ as a shorthand for $x :- (\lambda t \cdot c)$ when c is a constant function.

This explication of a continuous action system shows it essentially as a collection of time functions $x_0, \ldots, x_n$ 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 $[t_i \ldots t_{i+1})$ or a closed interval of the form $[t_i, t_i]$, i.e., a point. The action system determines a family of functions $x_0, \ldots, x_n$ 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 *Saw* example, the sequence of intervals is $[0], [0 \ldots 1), [1 \ldots 2), [2 \ldots 3), \ldots$ As such, the continuous action system can be best understood as the limit of a sequence of approximations of the time functions $x_0, \ldots, x_n$, defined over successively longer and longer intervals $[0 \ldots t_i)$, where $i = 0, 1, 2, \ldots$. Looking at the example in this way, its sequence of initial segments is $[0], [0 \ldots 1), [0 \ldots 2), [0 \ldots 3), \ldots$ and the defined approximations are successively:

$$x_0.t = 0, 0 \le t;\;\; x_1.t = t, 0 \le t;\;\; x_2.t = \begin{cases} t, & 0 \le t < 1 \\ 1 - t, & 1 \le t \end{cases};\;\; x_3.t = \begin{cases} t, & 0 \le t < 1 \\ 1 - t, & 1 \le t < 2 \\ t - 2, & 2 \le t \end{cases}$$

For each attribute x_i there is a defined history of its past, i.e. 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 an attribute 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 cannot 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.

Another important observation regards the possibility of *Zeno behavior*. That is, our definition does not guarantee that the sequence of generated intervals will cover all the non-negative reals. They might only cover an initial segment of these. In this case, there is a limit point of time that the action system reaches when the number of iterations reaches infinity. These systems are well-defined but the simple explication of the behavior of the hybrid system is then not sufficient. For this, we further assume that the system is restarted at the limit point, and repeat the process again. This is meaningful if all the attribute values converge to a well-defined value in the limit. This restart can be carried out as many times as needed. Thus, a continuous action systems may have multiple limit points in its execution. However, the standard action system semantics does not allow multiple limit points, so this is a point where the semantics has to be extended. For simplicity, we assume in the sequel that there is no Zeno-behavior and a single limit point is sufficient. The absence of Zeno behavior means that the action system will define the values of the attributes for the whole domain of Real_+.

A simple way of reaching a limit point is when a control computation (where the time does not advance) does not terminate. This means that the continuous behavior of the system is stuck at the last time instance reached. Non-termination of the control computation is most certainly undesired and unintended. This means that is desirable to prove that control computations where time does not advance always terminate.

Composing continuous action systems In order to model complex hybrid systems, where several different subsystems or components evolve concurrently, we need to formally define the composition of continuous action systems. Two actions systems communicate by means of imported and exported variables. We can also model other means of communication using the action systems framework [6], but this is out of the scope of this paper. For parallel composition, we may also need to rename certain attributes of the system when describing more complex systems, but we ignore this aspect here for brevity.

We define the parallel composition of two continuous systems by using essentially the parallel composition operator for ordinary action systems [5]. Thus, if we have two continuous action systems $\mathcal{C}$ and $\mathcal{C}'$ as in (1), then their parallel composition is the continuous action system $\mathcal{C} \parallel \mathcal{C}'$ defined as follows:

$$\begin{array}{rl} \mathcal{C} \parallel \mathcal{C}' \triangleq & |[\ \mathsf{var}\ x : \mathsf{Real}_+ \to \mathit{Val}, x' : \mathsf{Real}_+ \to \mathit{Val}'; \\ & \quad \bullet\ S_0\ ; S_0'; \\ & \quad \mathsf{do}\ g_1 \to S_1 \Box\ \ldots \Box\ g_m \to S_m \Box\ g_1' \to S_1' \Box\ \ldots \Box\ g_n' \to S_n'\ \mathsf{od} \\ &]| : (y \cup y') - (z \cup z') \end{array} \tag{4}$$

where the unprimed entities originally belonged to $\mathcal{C}$ and the primed entities to $\mathcal{C}'$. We assume here that the variables x and x' are disjoint. We need to combine the continuous action systems before we translate them into discrete action systems, because the local variable *now* appears in both $\bar{\mathcal{C}}$ and $\bar{\mathcal{C}}'$. By combining the continuous action systems first, we ensure that $\mathcal{C} \parallel \mathcal{C}'$ uses a single *now* variable, which is checked by actions from both components.

Thus, parallel composition essentially combines the attributes of the two component systems and, therefore, their continuous evolution. Because 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 component system may depend on an attribute in the other component 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 when it is alone.

4 Modeling Systems

In this section we illustrate how a hybrid system can be described as a continuous action system. We show how to model real-time systems, systems using differential equations, and also a press that reacts to external signals from the environment.

We can use clock variables 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 a system with the passage of the time. Since a clock variable is 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, e.g. $x.(now - 1)$. Hence, continuous action systems can be used to model real-time systems.

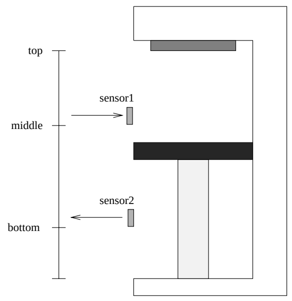
The behavior of a dynamic system is often described using a system of differential equations. We can allow this kind of definitions by introducing the shorthand

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

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 $\dot{x} :- \; (\lambda t \cdot c) \equiv x :- \; (\lambda t \cdot x.now + c * (t - now))$. Thus, we can use continuous action systems to express hybrid systems using either explicit functional expressions or implicit differential equations.

An example of a press from a metal processing factory [12] is shown in Fig. 2. The press works as follows. First, its lower part is raised until the middle position. Then an upper conveyor belt feeds a metal blank into the press. When the press is loaded (signalled by $sensor_1$ being T), the lower part of the press is raised until 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 (signalled by $sensor_2$ being T), its 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 $\mathcal{P}ress$



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

Fig. 2. Press functioning as a continuous action system.

shown in Fig. 2. This attribute can have 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.

The press example is a typical part of a control system. This kind of systems are essentially composed from several components that work together in order to meet the requirements of the overall system. Thus, an important feature of a component is its interaction with the environment. In the case of the press the interaction with the environment (two conveyor belts) is modeled with several sensors. The sensors are modeled as imported attributes that can be changed by the environment at any time. The press reads the values that $sensor_1$ and $sensor_2$ display, but these values are updated by the environment in a way we are not interested in here.

Other types of hybrid systems can be modelled as well using continuous action systems. Some more examples can be found in [8,14].

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 may need to consider multiple limit points, as was mentioned earlier). In this paper, we concentrate on safety properties, as in many cases they are the kind of properties that we want to initially establish for hybrid systems.

A common characterization for a safety property is that nothing 'bad' happens during the lifetime of the system. Put in another way, a safety property is a 'good' property G that always holds, i.e., $(\forall t \geq 0 \cdot G.t)$. We can establish this property for the action system $\mathcal{C}$ in (1) by proving that a property $I \triangleq (\forall t \mid 0 \leq t < \mathit{now} \cdot G'.t)$ 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 and does not terminate (i.e., *now* will go to infinity in the system). More precisely, the safety property G holds when the system is started in an initial state satisfying P, if and only if the following three conditions are satisfied for $\bar{\mathcal{C}}$:

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

Consider the press example in Fig. 2. We consider 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 $(\forall t \geq 0 \cdot \mathit{bottom} \leq p.t \leq \mathit{top})$, where p is the vertical position of the plate. Second, we 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 $(\forall t \geq 0 \cdot \mathit{bottom} \leq p.t \leq \mathit{top}) \wedge (p\ \mathit{continuous\ on}\ \mathsf{Real}_+)$ using the proof rule above.

For the first conjunct of the safety property, an invariant of the form $(\forall t \mid 0 \leq t \leq \mathit{now} \cdot \mathit{bottom} \leq p.t \leq \mathit{top})$ 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:

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

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. The discharging of the proof obligations can be found in [8].

6 Conclusions and Related Work

In this paper we have shown how to generalize the action systems framework for modeling hybrid systems, by introducing the notion of continuous action systems. We model attributes in continuous action systems as functions over time that are updated in a way that only changes their present and future behavior. Essentially, this amounts to extending the notion of state with both an 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 also shown that the continuous action systems model provides a simple way of defining the parallel composition of hybrid systems, using communication by means of imported and exported attributes. Finally, we explained how to prove safety properties of continuous action systems using the classical invariant method. We illustrated these concepts with a simple example, while a complete case study can be found in [3].

The idea of extending an existing formalism to model real-time systems by introducing a variable representing the time was 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, as we can reuse all the previous results on action systems to study real-time and hybrid systems models.

Rönkkö and Ravn [14] have already proposed a model for combining action systems and continuous behavior, called *hybrid action systems*. In their model, the continuous evolution of a variable is modeled as a special kind of atomic action. An atomic 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. Also, there is no implicit notion of time in their approach, which is not intended for modeling real-time systems. 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. Note that in the hybrid automata formalism, transitions are fired synchronously, while in the action system formalism actions are selected and executed asynchronously. The continuous action system formalism is more expressive than hybrid automata, as it allows 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.

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

The next step in the development of the continuous action systems formalism is to illustrate their stepwise refinement. This will provide for the derivation of executable control programs that are correct with respect to their specification, given as a continuous action system.

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

References

1. M. Abadi and L. Lamport. An old-fashioned receipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.
2. R. Alur, C. Courcoubetis, T.A. Henzinger, and P.H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Revn, and H. Rischel, editors, *Hybrid Systems I*, volume LNCS 736, pages 209–229. Springer-Verlag, 1993.
3. R. J. R. Back and C. Cerschi. 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*, 2000, to appear.
4. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *2nd Symp. on Principles of Distributed Computing*, volume LNCS 873, pages 131–142. ACM SIGACT-SIGOPS, 1983.
5. R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. In *Science of Computer Programming 13*, pages 133–180, 1991.
6. R. J. R. Back and K. Sere. From action systems to modular systems. In *Formal Methods Europe (FME '94)*, volume LNCS 873, pages 1–25. Springer-Verlag, 1994.
7. R. J. R. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction.* Springer-Verlag, 1998.
8. R.J. Back, L. Petre, and I. Porres. Generalizing action systems to hybrid systems. Technical Report 307, TUCS Turku Centre for Computer Science, 1999.
9. M.S. Branicky. General hybrid dynamical systems: modeling, analysis and control. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume LNCS 1066, pages 186–200. Springer-Verlag, 1996.
10. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.
11. Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. In *Lectures on Embedded Systems*, volume LNCS 1494, pages 4–73. Springer-Verlag, 1998.
12. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.*, volume LNCS 891. Springer-Verlag, 1995.
13. A. Nerode and W. Kohn. Models for hybrid systems: automata, topologies, controllability, observability. In R.L. Grossman, A. Nerode, A.P. Revn, and H. Rischel, editors, *Hybrid Systems I*, volume LNCS 736, pages 317–356. Springer-Verlag, 1993.
14. M. Rönkkö and A.P. Ravn. Action systems with continuous behaviour. In P. J. Antsaklis, W. Kohn, M. Lemmon, A. Nerode, and S. Sastry, editors, *Hybrid Systems V*, volume LNCS 1567, pages 304–323. Springer-Verlag, 1999.