Continuous Action System Refinement

Larissa Meinicke and Ian J. Hayes

School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Qld. 4072, Australia
{larissa, ianh}@itee.uq.edu.au

Abstract. Action systems are a framework for reasoning about discrete reactive systems. Back, Petre and Porres have extended these action systems to *continuous action systems*, which can be used to model hybrid systems. In this paper we define a refinement relation, and develop practical data refinement rules for continuous action systems.

The meaning of continuous action systems is expressed in terms of a mapping from continuous action systems to action systems. First, we present a new mapping from continuous action systems to action systems, such that Back's definition of trace refinement is correct with respect to it. Second, we present a stream semantics that is compatible with the trace semantics, but is preferable to it because it is more general. Although action system trace refinement rules are applicable to continuous action systems with a stream semantics, they are not complete. Finally, we introduce a new data refinement rule that is valid with respect to the stream semantics and can be used to prove refinements that are not possible in the trace semantics, and we analyse the completeness of our new rule in conjunction with the existing trace refinement rules.

1 Introduction

Action systems [4, 5] can be used to model discrete systems. Back, Petre and Porres extended action systems to *continuous action systems*, so that they could be used to model hybrid systems [1]. A hybrid system is one in which both continuous and discrete behaviour are modelled. In continuous action systems, variables are modelled as continuous timed streams and a special variable that represents the current time is introduced. Discrete (instantaneous) actions are used to update the continuous timed streams.

In the work of Back et al. [1], the behaviour of a continuous action system is defined in terms of an equivalent action system. This means that the definition of action system trace refinement [2] may be applied to continuous action systems. With respect to the definition of trace refinement, there are problems with the mapping from continuous action systems to action systems given by Back et al. [1]:

- It allows aborting action systems to be refined by ones that modify past behaviours.
- It requires the future values of output streams after every action to be preserved by refinement.

© Springer-Verlag Berlin Heidelberg 2006

The first problem allows a continuous action system to be refined by one that we consider to not faithfully preserve the behaviour of the continuous action system with respect to the intended interpretation. The second problem overly restricts allowable refinements. We provide a variation on their mapping from continuous action systems that addresses these problems.

Even though the new mapping avoids these problems the definition of action system trace refinement [2] is still overly restrictive because it requires the timing of actions to be preserved by refinement. To overcome this problem we introduce the notion of stream semantics. In our stream semantics, the behaviour of a continuous action system is expressed in terms of the set of continuous timed streams that it generates. We formally define stream semantics for continuous action systems in terms of their trace semantics, and compare trace and stream semantics. We find that trace refinement implies stream refinement, but that the converse does not hold. As a result, we argue that stream semantics should be used instead of the trace semantics because it is more general.

Practical refinement rules (simulation and cosimulation rules) exist for proving trace refinements between action systems [2]. Since trace refinement implies stream refinement, we may use these to prove stream refinements between continuous action systems. However, because stream semantics are more general than trace semantics, the trace refinement rules alone are incomplete for continuous action systems with a stream semantics. We introduce a new data refinement rule for continuous action systems that is able to prove refinements that are valid in the stream—but not the trace—model. For a subclass of continuous action systems, we demonstrate that our new rule in conjunction with the standard data refinement rules are as complete for continuous action system stream refinement as the action system data refinement rules are for standard action system trace refinement.

The following three sections contain background information relevant to the paper: the structure and semantics of action systems is described in Sections 2 and 3, Sect. 4 describes continuous action systems and their interpretation as action systems. In Sections 5, 6 and 7 we examine the semantics of continuous action systems in detail: an alternative mapping from continuous action systems to action systems is defined, a stream semantics for continuous action systems is given, and we perform a comparison between the stream and trace semantics. In Sect. 8 an algebra for reasoning about the semantics of continuous action systems is constructed and used to develop a new stream data refinement rule, and we discuss the completeness of the data refinement rules.

2 Action Systems

An action system [4, 5] is of the form:

$$|[\ \mathbf{var}\ x_1 : X_1;\ ...;\ x_n : X_n;\ S_0;\ \mathbf{do}\ S\ \mathbf{od}]|{:}< z_1 : Z_1, ..., z_m : Z_m >$$

where each x_i is a local variable, and each z_j is a global variable. S_0 is an *initialisation action* that initialises the local variables without modifying the global variables. S is an *action* that operates on the combined local and global state space. The following syntax is used to represent commands used in actions.

$$S ::= \{g\} \mid [g] \mid x := e \mid S_1;\ S_2 \mid S_1 \sqcap S_2 \mid \sqcap i : T \bullet S_i \mid S^\omega \mid S^* \mid S^\infty$$

Here g is a predicate, x is a variable in the state space, and e is an expression on the state space. The semantics of our actions are described using conjunctive *predicate transformers*. A predicate transformer is a function from predicates on the output state space Γ to predicates on the input state space Σ. Given a predicate transformer $S : (\Gamma \to \mathbb{B}) \to (\Sigma \to \mathbb{B})$ and a predicate q, $S.q$ returns the weakest precondition of S to achieve q. The conjugate of $S.q$ is written $\overline{S}.q$, and is defined as $\neg S.(\neg q)$. Informally, $\overline{S}.q$ specifies the set of states from which S may possibly achieve q (but is not necessarily guaranteed to achieve q). A predicate transformer S, is conjunctive if it distributes over nonempty meets, i.e., if $S.(\bigwedge i : I \bullet q_i) = (\bigwedge i : I \bullet S.q_i)$. Conjunctivity implies monotonicity. A predicate transformer S_2 is said to be a refinement of S_1 if, for all predicates q, the weakest precondition of S_2 to achieve q is implied by the weakest precondition of S_1 to achieve q:

$$S_1 \sqsubseteq S_2 \ \triangleq\ \forall q \bullet S_1.q \Rightarrow S_2.q$$

More detailed information about predicate transformers and program refinement can be found elsewhere [3, 13, 9].

Assertion :	$(\{g\}).q$	$g \wedge q$
Coercion :	$([g]).q$	$g \Rightarrow q$
Assignment :	$(x := e).q$	$q[x \setminus e]$
Sequential composition :	$(S_1;\ S_2).q$	$S_1.(S_2.q)$
Nondeterministic choice :	$(S_1 \sqcap S_2).q$	$S_1.q \wedge S_2.q$
General nondet. choice :	$(\sqcap i : T \bullet S_i).q$	$\bigwedge i : T \bullet S_i.q$
Strong iteration :	$(S^\omega).q$	$(\mu\, T \bullet S;\ T \sqcap \mathbf{skip}).q$
Weak iteration :	$(S^*).q$	$(\nu\, T \bullet S;\ T \sqcap \mathbf{skip}).q$
Infinite iteration :	$(S^\infty).q$	$(\mu\, T \bullet S;\ T).q$
skip :	**skip**	$[\mathit{true}]$
magic :	**magic**	$[\mathit{false}]$
abort :	**abort**	$\{\mathit{false}\}$

Fig. 1. Predicate transformer semantics of actions

In Fig. 1 we give a semantics for commands in which we identify a command with its predicate transformer. Assignment, assertion, coercion, nondeterministic choice, and sequential composition have the usual definitions. The unary operators (*,$^\omega$,$^\infty$) have the highest precedence, followed by ";", and then "$\sqcap$". We use the iteration constructs of Back and von Wright [6, 3]. Informally, weak iteration S^* performs the operation S any finite number of times. Strong iteration S^ω

either performs S any finite or any infinite number of times. Infinite iteration S^∞ performs S an infinite number of times. Strong and infinite iteration are defined using the least fixed point operators, while weak iteration is defined using the greatest fixed point operator. **skip** has no effect on the state. In terms of the refinement lattice, the least predicate transformer is **abort**, while the greatest predicate transformer is **magic**. **abort** is not guaranteed to terminate or produce any particular output. Infinite iterations of predicate transformers are considered to be aborting: for example, $\mathbf{skip}^\infty = \mathbf{skip}^\omega = \mathbf{abort}$. **magic** is miraculous, it can achieve everything, but it cannot be implemented.

We refer to the state space of an action system $\mathbf{A}$ as Σ_A, which is a mapping from the names of variables in $\mathbf{A}$, to the types of the variables (in each $\sigma : \Sigma_A$ each variable name must be mapped to a value in the corresponding type for that variable). The local and global parts of this space are referred to as $local.\Sigma_A$ and $global.\Sigma_A$ respectively, where $local.\Sigma_A$ and $global.\Sigma_A$ must have disjoint domains. For any state $\sigma : \Sigma_A$, we have that

$$\begin{aligned} local.\sigma &\triangleq dom.(local.\Sigma_A) \lhd \sigma \\ global.\sigma &\triangleq dom.(global.\Sigma_A) \lhd \sigma \end{aligned}$$

where "$\lhd$" represents domain restriction. Given an action system $\mathbf{A}$, we refer to the initialisation action of $\mathbf{A}$ as A_0 and the action as A. The guard of action A is denoted by $g.A$, and $t.A$ denotes the states from which action A terminates,

$$\begin{aligned} g.A &\triangleq \neg A.False \\ t.A &\triangleq A.True \end{aligned}$$

We write $A = g_1 \rightarrow S_1 \mathbin{[\!]} ... \mathbin{[\!]} g_m \rightarrow S_m$, to mean that $A = [g_1];\ S_1 \sqcap ... \sqcap [g_m];\ S_m$, where each g_i is a predicate and each S_i is a predicate transformer. For an action A of this form, we also refer to each predicate transformer "$[g_i];\ S_i$" as an action (an action can be viewed as a nondeterministic choice between a finite set of actions). If all predicate transformers S_i are non-miraculous for $A = g_1 \rightarrow S_1 \mathbin{[\!]} ... \mathbin{[\!]} g_m \rightarrow S_m$, then $g.A$ is simply $\bigvee i \bullet g_i$, and $t.A$ is $\bigwedge i \bullet t.([g_i];\ S_i)$.

3 Action System Trace Semantics

Back and von Wright have given a semantics for action systems in terms of traces [2]. The trace semantics of an action system $\mathbf{A}$ is given in terms of sets of *behaviours* that $\mathbf{A}$ may produce, $beh.\mathbf{A} : \mathbb{P}(seq.\Sigma_A)$. Each behaviour is a finite or infinite sequence of states from Σ_A (the state space of $\mathbf{A}$) that may be either terminating, nonterminating, or aborting. Each behaviour $b : beh.\mathbf{A}$ must satisfy the following conditions:

- The first state of b must be reachable by executing A_0 from a global initial state.
- For every pair of adjacent states in b, the second state must be reachable from the first by action A.

– For every state in b other than the final (for infinite behaviours there is no final state), $g.A$ and $t.A$ must hold.
– If b is finite then either $\neg g.A$ or $\neg t.A$ must hold in the final state.

A behaviour is defined to be terminating if it is finite and $\neg g.A$ holds in its final state; it is aborting if it is finite and its last final state satisfies $\neg t.A$, it is nonterminating if it is neither terminating or aborting.

$$\mathit{term}.b \triangleq \mathit{finite}.b \wedge \mathit{last}.b \in \neg g.A \tag{1}$$
$$\mathit{aborting}.b \triangleq \mathit{finite}.b \wedge \mathit{last}.b \in \neg t.A \tag{2}$$
$$\mathit{nonterm}.b \triangleq \neg \mathit{finite}.b \tag{3}$$

Note that action systems are reactive, hence their behaviour differs from that of predicate transformers. In reactive systems, the behaviour of the system up until an aborting action is executed is preserved. This means that nonterminating reactive systems that don't contain aborting actions generate behaviours of infinite length, while nonterminating predicate transformers are considered to be aborting.

An action system **A** is refined by another action system **C**, if the globally visible behaviour of **C** is permitted by **A**. In standard action systems, the globally visible view of a behaviour b is a *trace* $tr.b$ of type $\mathit{seq}.(\mathit{global}.\Sigma_A)$. A trace of a behaviour is simply the behaviour with all finite sequences of *stuttering* steps and local states removed: a stuttering step is a step which does not modify the global state. Formally, the trace refinement relation $\sqsubseteq_{\mathrm{tr}}$ between two action systems **A** and **C** is defined as follows [2]

$$\mathbf{A} \sqsubseteq_{\mathrm{tr}} \mathbf{C} \triangleq \forall\, b_C : \mathit{beh}.\mathbf{C} \bullet (\exists\, b_A : \mathit{beh}.\mathbf{A} \bullet b_A \preceq_{\mathrm{tr}} b_C)$$

where $b_A \preceq_{\mathrm{tr}} b_C$ if, neither $tr.b_A$ nor $tr.b_C$ is aborting and $tr.b_A = tr.b_C$, or $tr.b_A$ is aborting and is a prefix of the sequence $tr.b_C$.

4 Continuous Action Systems

Continuous action systems have the same form as action systems, however all variables are represented as timed streams. For some type VAL we define the set of all timed streams on VAL, $\mathit{Stream}.\mathit{VAL}$, as the set of total functions from Time to VAL:

$$\mathit{Stream} \triangleq \lambda\, \mathit{VAL} \bullet \mathit{Time} \rightarrow \mathit{VAL}$$

where Time is defined to be the set of non-negative real numbers. For any $s : \mathit{Stream}.\mathit{VAL}$, and time interval I, we refer to the stream s over time interval I as $s \downarrow I$. We write $s \ll s'$ to mean that s is a stream prefix of s'.

An implicit variable τ of type Time is used to refer to the current time. Actions that are performed on the continuous state space are atomic and they take no time to execute: time is allowed to pass between the execution of actions. Actions are constrained such that they cannot change the past: they are only allowed to change future values of timed streams. The initialisation command and the action may refer to the implicit variable τ however they may not update it. This variable is implicitly initialised and updated.

Given a variable x and an expression on the state space, the future update statement $x :- e$ [1] is defined as

$$x :- e \ \triangleq\ x := \lambda\, t \bullet \ \mathbf{if}\ t < \tau\ \mathbf{then}\ x.t\ \mathbf{else}\ e.t$$

This assignment statement is used instead of ":=" in order to ensure that actions do not change the past. We express nondeterministic future assignment as

$$x :\in E \ \triangleq\ \sqcap e : E \bullet x :- e$$

where E is a set of expressions on the state.

Any discrete (non-stream) variable, may be given a stream interpretation. For example a variable x of type $\mathbb{N}$ may be interpreted as a variable of type *Stream*.$\mathbb{N}$, the occurrence of x in expressions may be replaced by $x.\tau$, and assignments $x := v$ can be taken to mean $x :-(\lambda\, t : \mathit{Time} \bullet v)$. In later examples, for brevity, we define some continuous action system variables to be of discrete types.

The meaning of a continuous action system is expressed by Back et al. [1] using an equivalent action system.

Definition 1 (actSysOLD). *Given a continuous action system* **CA***, with local variables* $x_i : \mathit{Stream}.X_i$ *for* $i \in [1..n]$*, global variables* $z_j : \mathit{Stream}.Z_j$ *for* $j \in [1..m]$*, initialisation action* A_0*, and action* A*,* *actSysOLD*.**CA** *is defined as*

$$\begin{array}{l} |[\ \mathbf{var}\ \tau : \mathit{Time}, x_1 : \mathit{Stream}.X_1;\ ...;\ x_n : \mathit{Stream}.X_n; \\ \quad \tau := 0;\ A_0;\ N;\ \mathbf{do}\ A;\ N\ \mathbf{od} \\ ]|{:}< z_1 : \mathit{Stream}.Z_1, ..., z_m : \mathit{Stream}.Z_m > \end{array}$$

where

$$N \ \triangleq\ (\tau := \mathit{next}.(g.A).\tau)$$

$$\mathit{next}.gg.t \ \triangleq\ \begin{cases} \mathit{min}\{t' \mid t' \geq t \wedge gg.t'\}, & \mathit{if}\ (\exists\, t' \bullet t' \geq t \wedge gg.t') \\ t, & \mathit{otherwise} \end{cases}$$

In this mapping the variable τ is introduced, and initialised to zero. After the execution of each action τ is advanced to the earliest time the action will be enabled, if such a time exists; τ is not modified if no more commands will ever be enabled, or if a command is currently enabled. Although a continuous action system **CA** may map to a terminating action system, continuous action systems themselves have no termination time. Termination of *actSys*.**CA** merely signifies that from the termination time onwards, the stream variables evolve according to their last assignment.

Apart from satisfying these constraints, continuous action systems are not allowed to contain Zeno-behaviour: only a finite number of iterations are allowed in a finite period of time.

5 Continuous Action System Trace Semantics

Given the mapping from continuous action systems to action systems (Definition 1), we consider the definition of trace refinement to be invalid and overly restrictive for continuous action systems. We interpret it to be invalid because it allows aborting action systems to be refined incorrectly by action systems that modify past behaviours.

Continuous action systems should not modify past values of streams. This means that a behaviour that aborts at time t should not be refined by one that produces different output streams in the interval $[0..t)$, nor should it be refined by one that aborts at an earlier time. However, in Fig. 2 we can see that **CJ** is a trace refinement of **CI**: from initial state $y = y_0$, **CI** produces global trace $\langle y_0, f \rangle$, and then aborts, while **CJ** produces global output trace $\langle y_0, f, f \downarrow [0..1) \frown g \downarrow [1..\infty) \rangle$. (Where "$\frown$" is the stream concatenation operator.) **CI** aborts at time 2. **CJ** does not abort, however, it produces a different output stream in the interval $[0..2)$. This problem arises because the time of program abortion is irrelevant to the definition of trace refinement.

```
CI ≙
|[ var $n : \mathbb{N}$;
      $n := 0$;
      do $(\tau = n = 0) \to y :- f;\ n := n + 2$
       ▯ $(\tau = n = 2) \to abort$
       od
]|:< $y : Stream.\mathbb{N}$ >
```

```
CJ ≙
|[ var $n : \mathbb{N}$;
      $n := 0$;
      do $(\tau = n = 0) \to y :- f;\ n := n + 1$
       ▯ $(\tau = n = 1) \to$
            $y :- g;\ n := n + 2$
       ▯ $(\tau = n = 3) \to n := n + 1$
       od
]|:< $y : Stream.\mathbb{N}$ >
```

Fig. 2. Continuous action systems **CI** and **CJ**. f and g are functions of type $Stream.\mathbb{N}$.

We also consider the definition of trace refinement to be overly restrictive because the global stream variables are defined over all time: therefore the traces that describe the visible behaviour of the action system include information about the future values of output streams after each action. Since the future values of output streams may be modified by further actions, they should not have to be preserved by refinements. For example, we have that **CE** and **CF** (Fig. 3) produce the same overall output stream for y but, according to mapping *actSysOLD* (Definition 1), they are not trace equivalent: **CE** is not a valid trace refinement of **CF**, although **CF** is a valid trace refinement of **CE**. After each action, the set of possible future values of the streams are not the same, even though both of these programs produce the same output streams. (Both systems produce the same global stream $y = f$.) From initial state $y = y_0$, **CE** produces the set of global traces of the form $\langle y_0, g_1, g_2, g_3 ... \rangle$, where $\forall i : \mathbb{N} \bullet g_i \downarrow [0..i] = f \downarrow [0..i]$, while **CF** produces the global trace $\langle y_0, f, f, f, ... \rangle$.

```
CE ≜                                                    CF ≜
|[ var $n : \mathbb{N}$;                                |[ var $n : \mathbb{N}$;
    $n := 0$;                                               $n := 0$;
    do $(\tau = n) \to$                                     do $(\tau = n) \to$
        $n := n + 1$;                                           $n := n + 1$;
        $y :\in \{g : Stream.\mathbb{N} \mid g\downarrow[0..n] = f\downarrow[0..n]\}$        $y :- f$
    od                                                      od
]|:< $y : Stream.\mathbb{N}$ >                          ]|:< $y : Stream.\mathbb{N}$ >
```

Fig. 3. Continuous action systems **CE** and **CF**. f is a function of type *Stream*.$\mathbb{N}$.

A simple modification to the mapping from continuous action systems to action systems alleviates these problems. In our modification the global variables are redefined as partial streams: they are used to describe the output streams that have already been produced (and cannot be modified by further actions). Future values of global variables are stored as local variables. At the end of an action, if no future actions will be enabled then the global variables are defined over all time; if future actions are enabled then the global variables are defined over the half-open interval $[0..\tau)$. A half-open interval is used in this last case because future actions may change the values of variables at time τ.

We define $Stream^*.VAL$ to be the set of all partial streams on VAL defined over both half-open and closed intervals, and $Stream^\omega.VAL$ to be set of all partial and total streams on VAL.

$$\begin{aligned} Stream^* &\triangleq \lambda\, VAL \bullet \\ &\quad \{s : Time \nrightarrow VAL \mid \exists\, r : \mathbb{R} \bullet (dom.s = [0..r) \vee dom.s = [0..r])\} \\ Stream^\omega &\triangleq \lambda\, VAL \bullet Stream.VAL \cup Stream^*.VAL \end{aligned}$$

Definition 2 (actSys). *Given a continuous action system* **CA***, with local variables* $x_i : Stream.X_i$ *for* $i \in [1..n]$*, global variables* $z_j : Stream.Z_j$ *for* $j \in [1..m]$*, initialisation action* A_0*, and action* A*, let* $z \triangleq \langle z_1, ..., z_m\rangle$*,* $z' \triangleq \langle z_1', ..., z_m'\rangle$*. We define actSys.***CA** *as*

```
|[ var $\tau : Time$; $x_1 : Stream.X_1$; ...; $x_n : Stream.X_n$;
    $z_1' : Stream.Z_1$; ...; $z_m' : Stream.Z_m$;
    $\tau := 0$; $A_0[z \setminus z']$; $M$;
    do $A[z \setminus z']$; $M$ od
]|:< $z_1 : Stream^\omega.Z_1, ..., z_m : Stream^\omega.Z_m$ >
```

where

$$\begin{aligned} M &\triangleq \tau := next.(g.A[z \setminus z']).\tau; \\ &\qquad z_1 := znext.(g.A[z \setminus z']).z_1'.\tau;\ ...;\ z_m := znext.(g.A[z \setminus z']).z_m'.\tau \end{aligned}$$

$$next.gg.t \triangleq \begin{cases} min\{t' \mid t' \geq t \wedge gg.t'\}, & if\ (\exists\, t' \bullet t' \geq t \wedge gg.t') \\ t, & otherwise \end{cases}$$

$$znext.gg.z'.t \triangleq \begin{cases} z'\downarrow[0..t), & if\ gg.t \\ z' & otherwise \end{cases}$$

As before, the variable τ is introduced, and initialized to zero, and after the execution of each action τ is advanced to the earliest time an action will be enabled, if such a time exists. For each global variable z_j, a new local variable $z_j' : \mathit{Stream}.Z_j$ is introduced. This variable is used in actions A_0 and A instead of z_j. Each global variable z_j is redefined to be of type $\mathit{Stream}^\omega.Z_j$. After each action the global variables are updated so that they are defined either: over all time if no further actions are enabled, or just up until the current time if future actions are enabled.

Using our new mapping *actSys*, it is trivial to show that **CJ** is not a trace refinement of **CI** (Fig. 2). **CI** produces global trace $\langle f \downarrow [0..0), f \downarrow [0..2)\rangle$, while **CJ** produces global trace $\langle f \downarrow [0..0), f \downarrow [0..1), f \downarrow [0..1) \frown g \downarrow [1..\infty)\rangle$. We are also able to prove that continuous action systems **CE** and **CF** (Figure 3) are trace equivalent. We have that

```
$\mathbf{E} \triangleq \mathit{actSys}.\mathbf{CE} =$
$|[$ var $\tau : \mathit{Time}, n : \mathbb{N}, y' : \mathit{Stream}.\mathbb{N};$
    $\tau, n, y := 0, 0, y' \downarrow [0..0);$
    do $(\tau = n) \rightarrow$
        $n := n + 1;$
        $y' :\in \{g : \mathit{Stream}.\mathbb{N} \mid$
            $g \downarrow [0..n] = f \downarrow [0..n]\};$
        $\tau, y := n, y' \downarrow [0..n)$
    od
$]| :< y : \mathit{Stream}^\omega.\mathbb{N} >$
```

```
$\mathbf{F} \triangleq \mathit{actSys}.\mathbf{CF} =$
$|[$ var $\tau : \mathit{Time}, n : \mathbb{N}, y' : \mathit{Stream}.\mathbb{N};$
    $\tau, n, y := 0, 0, y' \downarrow [0..0);$
    do $(\tau = n) \rightarrow$
        $n := n + 1;$
        $y' :- f;$
        $\tau, y := n, y' \downarrow [0..n)$
    od
$]| :< y : \mathit{Stream}^\omega.\mathbb{N} >$
```

where we have simplified expressions *next* and *znext*. It can be seen that both **E** and **F** produce the nonterminating global trace $\langle f \downarrow [0..0), f \downarrow [0..1), f \downarrow [0..2), ...\rangle$, hence they are trace equivalent.

The parallel composition operator that Back et al. [1] defined for continuous action systems is performed at the continuous action system level (before mapping the continuous action systems to action systems), and hence it remains the same despite our modifications to the action system mapping *actSysOLD*.

6 Continuous Action System Stream Semantics

Trace refinement is valid with respect to our new mapping *actSys* (Definition 2) from continuous action systems to action systems, however it is still overly restrictive: this is because trace refinement requires the timing of actions to be preserved by refinement. This information should not have to be preserved, because it does not influence the set of global output streams that may be produced. In this section we define a stream semantics for continuous action systems that overcomes this problem. The stream semantics that we construct is a better choice of semantics than the trace model because it is more general.

Instead of using discrete traces over the continuous state variables to describe continuous action system semantics, we may describe its semantics in terms of the continuous timed streams that are generated by the program. We can express this alternative semantics in terms of the trace semantics of action systems. We define the set of streams that may be produced by a continuous action system **CA** as *behStreams*.(*actSys*.**CA**), where

$$behStreams.\mathbf{A} \triangleq \{b : beh.\mathbf{A} \bullet behStream.b\} \tag{4}$$

$$behStream.b \triangleq (\lambda\, var : (dom.\Sigma_A - \tau) \bullet \lim getSeq.b.var) \tag{5}$$

$$getSeq.b.var \triangleq \begin{cases} (\lambda\, i : dom.b \bullet b.i.var \downarrow [0..b.i.\tau)) \\ \quad\quad {}^\frown\langle last.b.var \downarrow Time\rangle, & \text{if } term.b \\ (\lambda\, i : dom.b \bullet b.i.var \downarrow [0..b.i.\tau)), & \text{otherwise} \end{cases} \tag{6}$$

Each stream is defined as the limit of a sequence of partial streams. (Back et al. [1] observed that the streams produced by continuous action systems could be defined in this way, although they did not specify that aborted sequences should be treated in the way we have done, nor do they define a refinement relation on sets of streams.) If the non-Zeno property holds (as assumed), then the limit of the sequence of partial streams *getSeq.b.var* is defined over all time if b is not aborting, and is defined up until the time of abortion if b is aborting. Aborted behaviours produce partial timed streams that have an open interval at the end. Aborted streams do not define the value of the stream at the time of abortion because refinements may modify this value. Given a continuous action system **CA** and s : *behStream*.(*actSys*.**CA**),

$$aborting.s \triangleq \exists\, r : Time \bullet (\forall\, var : dom.s \bullet dom.(s.var) = [0..r)) \tag{7}$$

The global behaviour of a stream s : *behStreams*.**CA** is referred to as *tr.s*, where

$$tr.s \triangleq global.s \tag{8}$$

If this semantics is adopted then a suitable notion of stream refinement, $\sqsubseteq_{\mathrm{str}}$, between continuous action systems may be defined. Given two continuous action systems **CA** and **CB**, we say that **CB** is a stream refinement of **CA** if *actSys*.**CA** $\sqsubseteq_{\mathrm{str}}$ *actSys*.**CB**, where

$$\mathbf{A} \sqsubseteq_{\mathrm{str}} \mathbf{B} \triangleq \forall\, s_B : behStreams.\mathbf{B} \bullet (\exists\, s_A : behStreams.\mathbf{A} \bullet s_A \preceq_{\mathrm{str}} s_B) \tag{9}$$

where

$$s_A \preceq_{\mathrm{str}} s_B \triangleq \begin{cases} tr.s_A \ll tr.s_B, & \text{if } aborting.(s_A) \\ tr.s_A = tr.s_B & \text{if } \neg aborting.(s_A) \end{cases}$$

Since our stream semantics is derived from the trace semantics, this definition of refinement is equivalent to the following: given action systems **A** and **B**,

$$\mathbf{A} \sqsubseteq_{\mathrm{str}} \mathbf{B} \triangleq \forall\, b_B : beh.\mathbf{B} \bullet (\exists\, b_A : beh.\mathbf{A} \bullet behStream.b_A \preceq_{\mathrm{str}} behStream.b_B) \tag{10}$$

This definition of refinement is used in later proofs. We write **CA** $\sqsupseteq_{\mathrm{str}}$ **CB** to mean that **CA** is stream equivalent to **CB**.

7 Correspondence Between Trace and Stream Semantics

Simple refinement rules exist for proving trace refinements between action systems. It would be useful if we could use these to prove stream refinements between continuous action systems. In this section, we show that if a refinement is valid using trace semantics then it is also valid using stream semantics.

Lemma 3. *For any continuous action system* **CA**, *and* $b : beh.(actSys.\mathbf{CA})$, *we have that:* $aborting.b \Leftrightarrow aborting.(behStream.b)$.

Proof. This follows from the definition of *aborting* for behaviours (2) and streams (7), the definitions of *behStream* (5), and *actSys* (Definition 2)), and the non-Zeno property for continuous action systems. □

Lemma 4. *For any continuous action system* **CA**, *and* $b : beh.(actSys.\mathbf{CA})$,

$$tr.(behStream.b) = trStream.(tr.b)$$

where

$$trStream.(tr.b) \triangleq \lambda\, var : dom.(global.\Sigma_A) \bullet \lim(\lambda\, i : dom.(tr.b) \bullet (tr.b).i.var)$$

Proof. This follows directly from *behStream* (5), *tr* for both behaviours and streams (8), and *actSys* (Definition 2). □

Lemma 5. *For continuous action systems* **CA** *and* **CB** *with the same global state space, and* $b_A : beh.(actSys.\mathbf{CA})$, $b_B : beh.(actSys.\mathbf{CB})$,

$$(b_A \preceq_{\mathrm{tr}} b_B) \Rightarrow (behStream.b_A \preceq_{\mathrm{str}} behStream.b_B)$$

Proof. We prove this by cases.

Case 1: $aborting.b_A$

$(b_A \preceq_{\mathrm{tr}} b_B)$

$\Rightarrow$ (Definition $\preceq_{\mathrm{tr}}$ and $aborting.b_A$)

$\exists\, n : dom.b_B \bullet (\forall\, var : dom.(global.\Sigma_A) \bullet last.b_A.var = b_B.n.var)$

$\Leftrightarrow$ (*actSys* (Definition 2) and $aborting.b_A$)

$\exists\, n : dom.b_B \bullet (\forall\, var : dom.(global.\Sigma_A) \bullet$
$\quad last.b_A.var \downarrow [0..last.b_A.\tau) = b_B.n.var \downarrow [0..b_B.n.\tau))$

$\Leftrightarrow$ (*getSeq* (6))

$\exists\, n : dom.b_B \bullet (\forall\, var : dom.(global.\Sigma_A) \bullet$
$\quad \lim getSeq.b_A.var = b_B.n.var \downarrow [0..last.b_B.\tau))$

$\Rightarrow$ (*actSys* (Definition 2) and *getSeq* (6). Note that from *actSys* and the constraints on continuous action systems, we have that actions cannot change the past.)

$\forall\, var : dom.(global.\Sigma_A) \bullet \lim getSeq.b_A.var \ll \lim getSeq.b_B.var$

$\Leftrightarrow$ (*behStream* (5) and *tr* (8))

$tr.(behStream.b_A) \ll tr.(behStream.b_B)$

$\Leftrightarrow$ ($aborting.b_A$ and Lemma 3)

$tr.(behStream.b_A) \ll tr.(behStream.b_B) \wedge aborting.(behStream.b_A)$

$\Leftrightarrow$ ($\preceq_{\mathrm{str}}$)

$behStream.b_A \preceq_{\mathrm{str}} behStream.b_B$

Case 2: $\neg aborting.b_A$

$$\begin{array}{ll} & (b_A \preceq_{\mathrm{tr}} b_B) \\ \Leftrightarrow & (\text{Definition } \preceq_{\mathrm{tr}} \text{ and } \neg aborting.b_A) \\ & tr.b_A = tr.b_B \\ \Rightarrow & (\text{Lemma 4}) \\ & tr.(behStream.b_A) = tr.(behStream.b_B) \\ \Leftrightarrow & (\neg aborting.b_A \text{ and Lemma 3}) \\ & tr.(behStream.b_A) = tr.(behStream.b_B) \wedge \neg aborting.(behStream.b_A) \\ \Leftrightarrow & (\preceq_{\mathrm{str}}) \\ & behStream.b_A \preceq_{\mathrm{str}} behStream.b_B \end{array}$$

□

Theorem 6. *For all continuous action systems* **CA** *and* **CB***,*

$$(actSys.\mathbf{CA} \sqsubseteq_{\mathrm{tr}} actSys.\mathbf{CB}) \Rightarrow (actSys.\mathbf{CA} \sqsubseteq_{\mathrm{str}} actSys.\mathbf{CB})$$

Proof. We have that,

$$\begin{array}{ll} & b_B \in beh.(actSys.\mathbf{CB}) \\ \Rightarrow & (actSys.\mathbf{CA} \sqsubseteq_{\mathrm{tr}} actSys.\mathbf{CB}) \\ & \exists\, b_A \in beh.(actSys.\mathbf{CA}) \bullet b_A \preceq_{\mathrm{tr}} b_B \\ \Rightarrow & (\text{Lemma 5}) \\ & \exists\, b_A \in beh.(actSys.\mathbf{CA}) \bullet behStream.b_A \preceq_{\mathrm{str}} behStream.b_B \end{array}$$

Hence, by the definition of stream refinement (10), $actSys.\mathbf{CA} \sqsubseteq_{\mathrm{str}} actSys.\mathbf{CB}$. □

The converse does not hold. That is, it is not true for all continuous action systems **CA** and **CB** that

$$(actSys.\mathbf{CA} \sqsubseteq_{\mathrm{str}} actSys.\mathbf{CB}) \Rightarrow (actSys.\mathbf{CA} \sqsubseteq_{\mathrm{tr}} actSys.\mathbf{CB})$$

For example we have that **CM** and **CN** (Fig. 4) are stream equivalent, but not trace equivalent. In **CN**, the action from **CM** has been decomposed into two steps. Both **CM** and **CN** produce global output stream f, however, **CM** produces global trace $\langle f \downarrow [0..0), f \downarrow [0..1), f \downarrow [0..2), ...\rangle$, while **CN** produces global trace $\langle f \downarrow [0..0), f \downarrow [0..0.5), f \downarrow [0..1), f \downarrow [0..1.5), ...\rangle$. Both traces are not aborting, but they are not equal. However, their limits are the same.

8 Data Refinement

As mentioned in the previous section, trace refinement is incomplete for continuous action systems with a stream semantics: that is, there exist valid stream refinements that are considered to be invalid in the trace model. In this section we derive a new simulation rule for proving stream refinements between continuous action systems. This rule can be used to prove refinements that are valid in the stream semantics, but may not be valid in the trace semantics. We then analyse the completeness of our new rule in conjunction with the action system data refinement rules. For our proofs we make use of algebraic properties of action systems.

```
CM ≜
|[ var n : ℝ;
    n := 0;
    do τ = n → z :∈ {g : Stream | f ↓ [0..τ + 1] = g ↓ [0..τ + 1]};
               n := n + 1
    od
]|:< z : Stream.ℕ >

CN ≜
|[ var n : ℝ, b : 𝔹;
    n, b := 0, true;
    do τ = n ∧ b → z :∈ {g : Stream | f ↓ [0..τ + 1] = g ↓ [0..τ + 1]};
               n, b := n + 1/2, false
    ⫿ τ = n ∧ ¬b → n, b := n + 1/2, true
    od
]|:< z : Stream.ℕ >
```

Fig. 4. Continuous action systems **CM** and **CN**. f is a function of type *Stream*.ℕ.

8.1 An Algebra for Continuous Action Systems

Algebraic theories are frequently used for reasoning about iterative program constructs: for example, Back and von Wright [6] have used results from fixed point theory to derive transformation rules for loop constructs, Hayes has used a similar approach to reason about execution paths in programs [11], and iterative real-time programming constructs [10], Cohen [7] and Kozen [12] performed early work on the Kleene algebra with tests. Here we define an algebra to describe sets of behaviours, and develop transformation rules for manipulating these sets of behaviours. These rules are used to derive a refinement rule for continuous action systems. Our algebra is closest to the concrete predicate transformer algebra of Back and von Wright [6][1] (the approach taken by Cohen [7] is and Kozen [12] is abstract-algebraic).

Behaviour Set Primitives and Composition Operators. Given a state space Σ, we use the primitives in Fig. 5 to describe sets of traces of type Σ. $\langle\!\langle A_0 \rangle\!\rangle$ defines a set of traces of length one such that the first value in the trace is reachable by A_0 from any global state. $\langle\!| A |\!\rangle$ defines a set of traces of length two, where the first element may be any possible state, and the second element is reachable from the first by executing action A (recall from Sect. 2 that $\overline{A_0}$ is the conjugate of A_0). Note that if A aborts, then every possible state is reachable from any initial state, hence $\langle\!|$ **abort** $|\!\rangle$ equals $\{\langle \sigma_1, \sigma_2 \rangle : seq.\Sigma \mid true\}$. A coercion primitive $[g]$ produces a set of traces of length one where the first value in a trace is any input from the state space that satisfies g. $\{g\}$ produces the set of all non-empty traces such that the first element of each trace does not satisfy

[1] Note that von Wright [14] later showed how to work with loop refinement in an abstract-algebraic setting.

Initialisation Action :	$\langle\!\langle A_0 \rangle\!\rangle$	$\{\langle\sigma_1\rangle : seq.\Sigma \mid (\exists\, \sigma_0 : global.\Sigma \bullet \overline{A_0}.(\lambda\, \sigma \bullet \sigma = \sigma_1).\sigma_0)\}$
Action :	$\llparenthesis\, A\, \rrparenthesis$	$\{\langle\sigma_1, \sigma_2\rangle : seq.\Sigma \mid \overline{A}.(\lambda\, \sigma \bullet \sigma = \sigma_2).\sigma_1\}$
Coercion :	$[g]$	$\{\langle\sigma_1\rangle : seq.\Sigma \mid g.\sigma_1\}$
Assertion :	$\{g\}$	$\{s : seq.\Sigma \mid \neg g.(first.s)\} \cup \{\langle\sigma_1\rangle : seq.\Sigma \mid g.\sigma_1\}$
Bottom :	**abort**	$\{false\}$
Top :	**magic**	$[false]$
Skip :	**skip**	$[true]$

Fig. 5. Trace set primitives. A_0 is an initialisation action, A is an action, and g is a predicate.

Nondeterministic choice :	$X \sqcap Y$	$X \cup Y$
General nondet. choice :	$\sqcap i : T \bullet X_i$	$\bigcup i : T \bullet X_i$
Sequential composition :	$X;\ Y$	$\{b : X \mid \neg finite.b\} \cup \{b, b' : seq.\Sigma, s : \Sigma \mid b \frown \langle s\rangle \in X \wedge \langle s\rangle \frown b' \in Y \bullet b \frown \langle s\rangle \frown b'\}$
Strong iteration :	Y^ω	$(\mu X \bullet Y;\ X \sqcap \mathbf{skip})$
Weak iteration :	Y^*	$(\nu X \bullet Y;\ X \sqcap \mathbf{skip})$
Infinite iteration :	Y^∞	$(\mu X \bullet Y;\ X)$

Fig. 6. Trace set composition operators

g, combined with the set of all traces of length one such that the first element of each trace satisfies g. The bottom set of traces is **abort**, which is the set of all possible traces, while the top set of traces is **magic**, which defines the empty set of traces. Note that **abort** is not equal to $\llparenthesis$ **abort** $\rrparenthesis$. Much of the notation we use here is overloaded, i.e., assertions are represented the same way for both actions and sets of traces. The meaning of statements should be clear from the context.

The trace set composition operations are defined in Fig. 6. We reuse the definition of *weak*, *strong* and *infinite* iteration used by Back and von Wright [6, 3]. The definition of *weak* iteration is equivalent to the Kleene star iterator of Kozen and Cohen [12, 7][2]. Informally, Y^* produces the set of traces that are constructed by sequentially composing Y any finite number of times, Y^∞ produces the set of traces that are constructed by sequentially composing Y an infinite number of times, and Y^ω produces the trace set $Y^* \cup Y^\infty$. Note that for programs represented using the predicate transformer semantics from Sect. 2, we have that nonterminating behaviour is equivalent to **abort**; here this is not the case, nontermination generates traces of infinite length. In this sense the meaning of our infinite iteration operator is most similar to that used by Hayes [10]. (Cohen has also constructed an infinite iteration operator (Y^∞) that is applicable to trace-based models [8].)

The set of trace sets satisfies all the properties of an an idempotent semiring under ("$\sqcap$", "; ", **magic**, **skip**), except that X; **magic** $\neq$ **magic** does not hold

[2] This equivalence is described by the Kleene star equivalence property.

in general if X does not terminate. Additionally, the trace composition operators satisfy the following properties.

Theorem 7. *The following properties hold for both conjunctive predicate transformer semantics and the semantics of sets of traces.*

$$\begin{array}{rr}
X^{\omega} = X;\ X^{\omega} \sqcap \mathbf{skip} \textit{ and } X^{\omega} = X^{\omega};\ X \sqcap \mathbf{skip} & \text{(unfold strong iteration)} \\
X^{*} = X;\ X^{*} \sqcap \mathbf{skip} \textit{ and } X^{*} = X^{*};\ X \sqcap \mathbf{skip} & \text{(unfold weak iteration)} \\
X^{\infty} = X;\ X^{\infty} & \text{(unfold infinite iteration)} \\
X^{\omega} = X^{*} \sqcap X^{\infty} & \text{(decompose strong iteration)} \\
X^{\infty} = X^{\omega};\ \mathbf{magic} & \text{(infinite iteration equivalence)} \\
X^{*} = \sqcap i : \mathbb{N} \bullet X^{i} & \text{(Kleene star equivalence)} \\
X;\ (Y;\ X)^{\omega} = (X;\ Y)^{\omega};\ X & \text{(leapfrog)} \\
(X \sqcap Y)^{\omega} = X^{\omega};\ (Y;\ X^{\omega})^{\omega} & \text{(decomposition)} \\
[g1] \sqcap [g2] = [g1 \vee g2] & \text{(guard disjunction)} \\
[g1];\ [g2] = [g1 \wedge g2] & \text{(guard conjunction)}
\end{array}$$

Here $X^0 = \mathbf{skip}$ *and* $X^{i+1} = X;\ X^i$ *for* $i \in \mathbb{N}$*.*

All of the properties in Theorem 7, apart from *Kleene star equivalence*, have been verified by by Back et al. to be correct for conjunctive predicate transformers [6, 3]: they are also applicable to sets of traces. The *Kleene star equivalence* property may be simply verified for both conjunctive predicate transformers and sets of traces. (Induction rules also exist, however they are not required except to prove Theorem 7).

Theorem 8. *The following properties hold for sets of traces:*

$$\begin{array}{rr}
\langle\!\mid [g];\ A \mid\!\rangle = [g];\ \langle\!\mid A \mid\!\rangle & \text{(shift guard)} \\
\langle\!\mid A \sqcap A' \mid\!\rangle = \langle\!\mid A \mid\!\rangle \sqcap \langle\!\mid A' \mid\!\rangle & \text{(shift nondet. choice)} \\
\langle\!\mid \sqcap i : T \bullet A_i \mid\!\rangle = \sqcap i : T \bullet \langle\!\mid A_i \mid\!\rangle & \text{(shift general nondet. choice)}
\end{array}$$

Lemma 9. *Given action A such that A^{ω} is terminating, i.e., $A^{\omega}.True = True$,*

$$\langle\!\mid A \mid\!\rangle^{\omega} = \langle\!\mid A \mid\!\rangle^{*} \textit{ and } A^{\omega} = A^{*}$$

Proof. If A^{ω} is terminating $A^{\infty} = \mathbf{magic}$ (from Theorem 7 (*infinite iteration equivalence*)), and so from Theorem 7 (*decompose strong iteration*) we have that $A^{\omega} = A^{*} \sqcap \mathbf{magic} = \mathbf{A}^{*}$. A similar argument applies for $\langle\!\mid A \mid\!\rangle^{\omega}$. □

Defining Action System Behaviours. The behaviours of an action system may then be expressed using our primitives and composition operators as follows.

$$beh.\mathbf{A} \triangleq \langle\!\langle A_0 \rangle\!\rangle;\ ([t.A];\ \langle\!\mid A \mid\!\rangle)^{\omega};\ [\neg g.A \vee \neg t.A] \qquad (11)$$

We express guarded loops using iteration constructs in the same way as Back and von Wright [6].

For action systems with trace semantics, it well known that we can merge together two actions A and B as long as B is a stuttering action, without changing the set of global traces that are produced by the action system. For continuous action systems (using stream semantics) we are also able to merge together two actions A and B as long as B does not abort (B may be non-stuttering), without changing the set of streams that are produced by the action system. It is this property that enables us to introduce a new data refinement rule for continuous action systems. We write $X =_{\mathrm{str}} Y$, where X and Y are set of traces, to mean that $behStreams.X = behStreams.Y$.

Lemma 10. *For action A, initialisation action A_0, and terminating action B, we have that*

$$\llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis =_{\mathrm{str}} \llparenthesis\, A;\ B \,\rrparenthesis \textit{ and } \langle\!\langle A_0 \rangle\!\rangle;\ \llparenthesis\, B \,\rrparenthesis =_{\mathrm{str}} \langle\!\langle A_0;\ B \rangle\!\rangle$$

Proof. We show that if B is not aborting then $\llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis =_{\mathrm{str}} \llparenthesis\, A;\ B \,\rrparenthesis$. The proof of $\langle\!\langle A_0 \rangle\!\rangle;\ \llparenthesis\, B \,\rrparenthesis =_{\mathrm{str}} \langle\!\langle A_0;\ B \rangle\!\rangle$ is similar. From the definition of the behaviour set primitives and sequential composition we have that

$$\begin{aligned} \llparenthesis\, A;\ B \,\rrparenthesis &= \{\langle \sigma_1, \sigma_3\rangle \mid \overline{(A;\ B)}.(\lambda\,\sigma \bullet \sigma = \sigma_3).\sigma_1\} \\ \llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis &= \{\langle \sigma_1, \sigma_2, \sigma_3\rangle \mid \overline{A}.(\lambda\,\sigma \bullet \sigma = \sigma_2).\sigma_1 \wedge \overline{B}.(\lambda\,\sigma \bullet \sigma = \sigma_3).\sigma_2\} \end{aligned}$$

And from the definition of predicate transformer sequential composition and conjugates, we have

$$\overline{(A;\ B)}.(\lambda\,\sigma \bullet \sigma = \sigma_3).\sigma_1 \Leftrightarrow \exists\,\sigma_2 \bullet \overline{A}.(\lambda\,\sigma \bullet \sigma = \sigma_2).\sigma_1 \wedge \overline{B}.(\lambda\,\sigma \bullet \sigma = \sigma_3).\sigma_2$$

Hence $\langle \sigma_1, \sigma_2, \sigma_3\rangle \in \llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis$, iff $\langle \sigma_1, \sigma_3\rangle \in \llparenthesis\, A;\ B \,\rrparenthesis$. Since all traces from $\llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis$ and $\llparenthesis\, A;\ B \,\rrparenthesis$ are finite, and each trace b from either $\llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis$ or $\llparenthesis\, A;\ B \,\rrparenthesis$, has a corresponding trace b' from the other set in which $last.b' = last.b$, from $behStream$ (5) we have that $behStream.b = behStream.b'$, and hence (from the definition of stream refinement (10)), $\llparenthesis\, A \,\rrparenthesis;\ \llparenthesis\, B \,\rrparenthesis =_{\mathrm{str}} \llparenthesis\, A;\ B \,\rrparenthesis$. □

Lemma 11. $\llparenthesis$ **skip** $\rrparenthesis =_{\mathrm{str}}$ **skip**, *where the first occurrence of* **skip** *is a predicate transformer, and the second occurrence is a set of traces.*

Lemma 12. *Given terminating action A,*
$\llparenthesis\, A \,\rrparenthesis^{*} =_{\mathrm{str}} \llparenthesis\, A^{*} \,\rrparenthesis$

Proof.

$$\begin{array}{ll} & \llparenthesis\, A \,\rrparenthesis^{*} \\ = & \text{(Theorem 7 (\textit{Kleene star equivalence}))} \\ & \sqcap i : \mathbb{N} \bullet \llparenthesis\, A \,\rrparenthesis^{i} \\ =_{\mathrm{str}} & \text{(Lemma 10 and 11)} \\ & \sqcap i : \mathbb{N} \bullet \llparenthesis\, A^{i} \,\rrparenthesis \\ = & \text{(Theorem 8 (\textit{shift general nondet. choice}))} \\ & \llparenthesis\, \sqcap i : \mathbb{N} \bullet A^{i} \,\rrparenthesis \\ = & \llparenthesis\, A^{*} \,\rrparenthesis \end{array}$$

8.2 Stream Refinement Rules

In this section we derive new stream data refinement rules for continuous action systems using the rules developed in the previous section. First we construct an equivalence rule, that shows how possibly non-stuttering actions may be merged with other actions in a continuous action system without changing the output streams that are generated by it. We then combine this rule with existing action system trace refinement rules to generate new useful stream refinement rules.

Given a continuous action system $\mathbf{CA}$, let $\mathbf{A} = \mathit{actSys}.\mathbf{CA}$. For an action decomposition $A = A_\natural \sqcap A_\sharp$, we define action system $\mathbf{A}_{(\mathbf{A}_\natural,\mathbf{A}_\sharp)}$ as follows.

$$\mathbf{A}_{(\mathbf{A}_\natural,\mathbf{A}_\sharp)} \triangleq |[\ \mathbf{var}\ x : X;\ A_0;\ \mathit{DO}_{(A_\natural,A_\sharp)};\ \mathbf{do}\ A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)}\ \mathbf{od}]|{:}< z : Z >$$

where the local and global variables are the same as those of $\mathbf{CA}$. Program $\mathit{DO}_{(A_\natural,A_\sharp)}$ may perform action $A_\natural$ for as long as it is enabled and an aborting action isn't enabled, and it may terminate when either the guard of $A_\sharp$ holds or the guard of $A_\natural$ ceases to hold:

$$\mathit{DO}_{(A_\natural,A_\sharp)} \triangleq ([t.A];\ A_\natural)^\omega;\ [\neg g.A_\natural \vee g.A_\sharp]$$

Theorem 13 (Stream Equivalence). *If $\mathit{DO}_{(A_\natural,A_\sharp)}$ is terminating (note that this implies that $A_\natural$ must not be aborting), we have that*

$$\mathbf{A} \mathrel{\sqsubseteq\!\!\!\sqsupseteq}_{\mathrm{str}} \mathbf{A}_{(\mathbf{A}_\natural,\mathbf{A}_\sharp)}$$

Proof. We show that $\mathit{beh}.\mathbf{A}$ is stream equivalent to $\mathit{beh}.\mathbf{A}_{(\mathbf{A}_\natural,\mathbf{A}_\sharp)}$.

$$\begin{array}{ll}
 & \mathit{beh}.\mathbf{A}_{(\mathbf{A}_\natural,\mathbf{A}_\sharp)} \\
= & (11) \\
 & \langle\!\langle A_0;\ \mathit{DO}_{(A_\natural,A_\sharp)}\rangle\!\rangle;\ ([t.(A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)})];\ \llparenthesis\ A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)}\ \rrparenthesis)^\omega; \\
 & [\neg g.(A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)}) \vee \neg t.(A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)})] \\
= & \langle\!\langle A_0;\ \mathit{DO}_{(A_\natural,A_\sharp)}\rangle\!\rangle;\ ([t.A_\sharp];\ \llparenthesis\ A_\sharp;\ \mathit{DO}_{(A_\natural,A_\sharp)}\ \rrparenthesis)^\omega;\ [\neg g.A_\sharp \vee \neg t.A_\sharp] \\
= & (\mathit{DO}_{(A_\natural,A_\sharp)} \text{ is terminating, and Lemma 9}) \\
 & \langle\!\langle A_0;\ ([t.A];\ A_\natural)^*;\ [\neg g.A_\natural \vee g.A_\sharp]\rangle\!\rangle; \\
 & ([t.A_\sharp];\ \llparenthesis\ A_\sharp;\ ([t.A];\ A_\natural)^*;\ [\neg g.A_\natural \vee g.A_\sharp]\ \rrparenthesis)^\omega;\ [\neg g.A_\sharp \vee \neg t.A_\sharp] \\
=_{\mathrm{str}} & (\text{Lemmas 10 and 12, Theorem 8 } (\mathit{shift\ guard}), \text{ and} \\
 & \text{Theorem 7 } (\mathit{guard\ conjunction})) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*;\ [\neg g.A_\natural \vee g.A_\sharp]; \\
 & ([t.A_\sharp \wedge g.A_\sharp];\ \llparenthesis\ A_\sharp\ \rrparenthesis;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*;\ [\neg g.A_\natural \vee g.A_\sharp])^\omega;\ [\neg g.A_\sharp \vee \neg t.A_\sharp] \\
= & (\text{Theorem 7 } (\mathit{leapfrog})) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*;\ ([\neg g.A_\natural \vee g.A_\sharp];\ [t.A_\sharp \wedge g.A_\sharp];\ \llparenthesis\ A_\sharp\ \rrparenthesis;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*)^\omega; \\
 & [\neg g.A_\natural \vee g.A_\sharp];\ [\neg g.A_\sharp \vee \neg t.A_\sharp] \\
= & (\text{Theorem 7 } (\mathit{guard\ conjunction}),\ \neg g.A = \neg g.A_\natural \wedge \neg g.A_\sharp,\ \neg t.A_\sharp \Rightarrow g.A_\sharp) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*;\ ([t.A_\sharp \wedge g.A_\sharp];\ \llparenthesis\ A_\sharp\ \rrparenthesis;\ \llparenthesis\ [t.A];\ A_\natural\ \rrparenthesis^*)^\omega;\ [\neg g.A \vee \neg t.A_\sharp] \\
= & (\mathit{DO}_{(A_\natural,A_\sharp)} \text{ is terminating, Lemma 9 and Theorem 8 } (\mathit{shift\ guard})) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ ([t.A];\ \llparenthesis\ A_\natural\ \rrparenthesis)^\omega;\ ([t.A_\sharp];\ \llparenthesis\ A_\sharp\ \rrparenthesis;\ ([t.A];\ \llparenthesis\ A_\natural\ \rrparenthesis)^\omega)^\omega;\ [\neg g.A \vee \neg t.A_\sharp] \\
= & (\text{Theorem 7 } (\mathit{decomposition})) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ ([t.A];\ \llparenthesis\ A_\natural\ \rrparenthesis \sqcap [t.A_\sharp];\ \llparenthesis\ A_\sharp\ \rrparenthesis)^\omega;\ [\neg g.A \vee \neg t.A_\sharp] \\
= & (\text{Theorem 8 } (\mathit{shift\ nondet.\ choice}),\ t.A = t.A_\natural \wedge t.A_\sharp \text{ and } t.A_\natural = \mathit{true}) \\
 & \langle\!\langle A_0\rangle\!\rangle;\ ([t.A];\ \llparenthesis\ A_\natural \sqcap A_\sharp\ \rrparenthesis)^\omega;\ [\neg g.A \vee \neg t.A] \\
= & (11) \\
 & \mathit{beh}.\mathbf{A}
\end{array}$$

□

Since trace refinement implies stream refinement (Theorem 6), we are able to combine the stream equivalence rule (Theorem 13) with standard action system trace refinement rules in order to generate new stream refinement rules. From the standard trace simulation rule for action systems [2] we have that **A** is refined by **B** if there exists a *valid* representation program *rep* such that

$$\begin{aligned} A_0;\ rep &\sqsubseteq B_0 \\ A;\ rep &\sqsubseteq rep;\ B \\ g.A \wedge t.A &\Rightarrow rep.gB \end{aligned}$$

We say that a representation program is valid if it does not modify the global state and it is non-miraculous.

Theorem 14 (Stream Simulation). *For any continuous action systems* **CA** *and* **CB** *with the same global state space, let* **A** *be actsys.***CA** *and* **B** *be actsys.***CB***. If there exists a decomposition* $A = A_\sharp \sqcap A_\natural$ *and* $B = B_\sharp \sqcap B_\natural$ *such that programs* $DO_{(A_\natural, A_\sharp)}$ *and* $DO_{(B_\natural, B_\sharp)}$ *terminate and there exists a valid representation program rep, such that*

$$A_0;\ DO_{(A_\natural, A_\sharp)};\ rep \sqsubseteq B_0;\ DO_{(B_\natural, B_\sharp)} \tag{12}$$

$$A_\sharp;\ DO_{(A_\natural, A_\sharp)};\ rep \sqsubseteq rep;\ B_\sharp;\ DO_{(B_\natural, B_\sharp)} \tag{13}$$

$$g.A_\sharp \wedge t.A_\sharp \Rightarrow rep.(gB_\sharp) \tag{14}$$

then $\mathbf{CA} \sqsubseteq_{\mathrm{str}} \mathbf{CB}$.

Proof. This follows directly from Theorems 13 and 6, and the standard action system simulation rule. □

A corresponding stream cosimulation rule exists. The stream simulation rule may be used to prove refinements (using stream semantics) that are not possible using the standard simulation and cosimulation rules. We demonstrate this by showing that **CN** is a valid refinement of **CM** (Fig. 4). Recall from Sect. 7 that in **CM**, one action from **CN** has been decomposed into two separate actions.

Example. $\mathbf{CM} \sqsubseteq_{\mathrm{str}} \mathbf{CN}$ (Figure 4)

Proof. Let **M** and **N** be *actSys*.**CM** and *actSys*.**CN**, respectively. Then

$M_0 = n, \tau := 0, 0;\ z := z' \downarrow [0..0)$

$N_0 = n, b, \tau := 0, \mathit{true}, 0;\ z := z' \downarrow [0..0)$

The proof obligations of the stream simulation rules can easily be shown to hold given the following action decompositions (proof steps elided). Program *intr* introduces the local variable b.

$$\begin{aligned} rep &\triangleq intr;\ b := \mathit{true} \\ M_\sharp &\triangleq [\tau = n];\ z' :\in \{g : \mathit{Stream}.\mathbb{N} \mid f \downarrow [0..\tau + 1] = g \downarrow [0..\tau + 1]\}; \\ &\quad\ n := n + 1;\ \tau := n;\ z := z' \downarrow [0..\tau) \\ M_\natural &\triangleq \mathbf{magic} \\ N_\sharp &\triangleq [\tau = n \wedge b];\ z' :\in \{g : \mathit{Stream}.\mathbb{N} \mid f \downarrow [0..\tau + 1] = g \downarrow [0..\tau + 1]\} \\ &\quad\ n, b := n + \tfrac{1}{2}, \mathit{false};\ \tau := n;\ z := z' \downarrow [0..\tau) \\ N_\natural &\triangleq [\tau = n \wedge \neg b];\ n, b := n + \tfrac{1}{2}, \mathit{true};\ \tau := n;\ z := z' \downarrow [0..\tau) \end{aligned}$$

We have that

$$\begin{aligned}DO_{(M_\natural, M_\sharp)} &\triangleq \textbf{skip}\\ DO_{(N_\natural, N_\sharp)} &\triangleq [\tau = n \wedge \neg b];\ n, b := n + \tfrac{1}{2}, \mathit{true};\ \tau := n;\ z := z' \downarrow [0..\tau)\\ &\quad \sqcap [(\tau = n) \Rightarrow b]\end{aligned}$$

□

8.3 Completeness of Data Refinement Rules

A set of rules is complete with respect to a chosen semantics if all valid refinements in the semantics can be proven using the specified rules. As mentioned earlier, the standard action system data refinement rules alone are incomplete with respect to the stream semantics for continuous action systems. Here we prove a completeness result for our new stream refinement rules in conjunction with the action system data refinement rules, with respect to our stream semantics.

For any continuous action system **CA** such that *actSys*.**CA** is nonterminating, and **CA** only contains terminating actions, we show that it is possible to use our new stream refinement rules to convert *actSys*.**CA** to a stream equivalent canonical form *can*.(*actSys*.**CA**). For any two such continuous action systems, **CA** and **CB**, we show that both trace and stream refinement are equivalent for *can*.(*actSys*.**CA**) and *can*.(*actSys*.**CB**). This means that, for this case, the usual completeness results for action system data refinement apply.

Using our new stream simulation rule (Theorem 14), we can show that any continuous actions system **CA** is stream equivalent to

$$\begin{array}{l}\mathbf{CA_{check}} \triangleq\\ |[\ \textbf{var}\ x : \mathit{Stream}.X, \mathit{check} : \mathit{Time};\\ \quad CA_0;\ \mathit{check} := 0;\ \mathit{CHECK};\\ \quad \textbf{do}\ CA;\ \mathit{CHECK}\ [\!]\ \mathit{check} = \tau \rightarrow \mathit{check} := \mathit{check} + p\ \textbf{od}\\ ]|{:<}\ z : \mathit{Stream}^{\omega}\ {>}\end{array}$$

where

$$\begin{array}{l}\mathit{CHECK} \triangleq\\ \textbf{if}\ (\exists\, t : \mathit{Time} \bullet t \geq \tau \wedge (g.CA).t) \rightarrow \mathit{skip}\\ [\!]\ \neg(\exists\, t : \mathit{Time} \bullet t \geq \tau \wedge (g.CA).t) \rightarrow \mathit{check} := -1\\ \textbf{fi}\end{array}$$

check is a fresh variable and p is a defined non-zero time period. (It is assumed that x and z are the local and global variables of **CA** respectively.) The action "$[\mathit{check} = \tau];\ \mathit{check} := \mathit{check} + p$" occurs every p seconds until the action system terminates or aborts:

$$\begin{array}{l}\mathbf{A_{check}} \triangleq \mathit{actSys}.\mathbf{CA_{check}} =\\ |[\ \textbf{var}\ \tau : \mathit{Time}, x : \mathit{Stream}.X, z' : \mathit{Stream}.Z, \mathit{check} : \mathit{Time};\\ \quad (\tau := 0;\ CA_0;\ \mathit{check} := 0;\ \mathit{CHECK})[z \setminus z'];\ M;\\ \quad \textbf{do}\ (CA;\ \mathit{CHECK})[z \setminus z'];\ M\ [\!]\ \mathit{check} = \tau \rightarrow \mathit{check} := \mathit{check} + p;\ M\ \textbf{od}\\ ]|{:<}\ z : \mathit{Stream}^{\omega}.Z\ {>}\end{array}$$

For a continuous action system **CA** such that action CA terminates and *actSys*.**CA** is nonterminating, we define the canonical form of *actSys*.**CA**, *can*.(*actSys*.**CA**), to be

$$\begin{array}{l} |[\ \mathbf{var}\ \tau : \mathit{Time}, x : \mathit{Stream}.X, z' : \mathit{Stream}.Z, \mathit{check} : \mathit{Time}; \\ \quad A_{\mathit{check}_0};\ DO_{(A_{\mathit{check}_\natural}, A_{\mathit{check}_\sharp})}; \\ \quad \mathbf{do}\ A_{\mathit{check}_\sharp};\ DO_{(A_{\mathit{check}_\natural}, A_{\mathit{check}_\sharp})}\ \mathbf{od} \\ ]|:< z : \mathit{Stream}^{\omega}.Z > \end{array} \tag{15}$$

where

$$\begin{array}{lcl} A_{\mathit{check}_\sharp} & \triangleq & [\mathit{check} = \tau];\ \mathit{check} := \mathit{check} + p;\ M \\ A_{\mathit{check}_\natural} & \triangleq & (\mathit{CA};\ \mathit{CHECK})[z \setminus z'];\ M \end{array}$$

The action of $\mathit{can}.(\mathit{actSys}.\mathbf{CA})$ occurs every p seconds. It performs all the actions in actSys.**CA** that occur between τ and $\tau + p$. The variable *check* is used to regulate the period of the actions. For any two non-aborting, nonterminating action systems in canonical form, the timing of their actions is the same.

Lemma 15. *Given continuous action system* **CA** *such that action CA is terminating,*

$$\mathit{can}.(\mathit{actSys}.\mathbf{CA}) =_{\mathrm{str}} \mathit{actSys}.\mathbf{CA}$$

Proof. The stream equivalence rule (Theorem 13) can be used to verify that $\mathit{actSys}.\mathbf{CA_{check}}$ is equivalent to $\mathit{can}.(\mathit{actSys}.\mathbf{CA})$ (note that because action CA is terminating, $A_{\mathit{check}_\natural}$ is terminating). Theorem 14 can be used to prove the equivalence between $\mathbf{CA_{check}}$ and **CA**. □

Lemma 16. *For any two continuous action systems* **CA** *and* **CB** *such that actions CA and CB are terminating and actSys.***CA** *and actSys.***CB** *are nonterminating,*

$$(\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{str}} \mathit{can}.(\mathit{actSys}.\mathbf{CB})) = (\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{tr}} \mathit{can}.(\mathit{actSys}.\mathbf{CB}))$$

Proof.

1. $(\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{str}} \mathit{can}.(\mathit{actSys}.\mathbf{CB})) \Rightarrow$
   $(\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{tr}} \mathit{can}.(\mathit{actSys}.\mathbf{CB}))$
   This follows from the fact that the semantics of continuous action systems are trace extending (*actSys* (Definition 2)), and that actions in $\mathit{can}.(\mathit{actSys}.\mathbf{CA})$ and $\mathit{can}.(\mathit{actSys}.\mathbf{CB})$ occur at the same regular interval for all time.
2. $(\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{tr}} \mathit{can}.(\mathit{actSys}.\mathbf{CB})) \Rightarrow$
   $(\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{str}} \mathit{can}.(\mathit{actSys}.\mathbf{CB}))$
   There exists a continuous action system $\mathbf{CA_{can}}$ such that $\mathit{actSys}.\mathbf{CA_{can}} = \mathit{can}.(\mathit{actSys}.\mathbf{CA})$, so implication in this direction follows from Theorem 6.

□

Theorem 17. *For any two continuous action systems* **CA** *and* **CB** *such that actions CA and CB are terminating,*

$$(\mathit{actSys}.\mathbf{CA} \sqsubseteq_{\mathrm{str}} \mathit{actSys}.\mathbf{CB}) = (\mathit{can}.(\mathit{actSys}.\mathbf{CA}) \sqsubseteq_{\mathrm{tr}} \mathit{can}.(\mathit{actSys}.\mathbf{CB}))$$

Proof. This follows from Lemma 15 and Lemma 16. □

This theorem demonstrates that, for a restricted class of continuous action systems (those that do not contain aborting behaviours, and that are associated with nonterminating action systems), our new stream refinement rules in conjunction with the action system trace refinement rules are as complete with respect to our stream semantics, as the action system trace refinement rules are with respect to the trace semantics. For action systems with neither infinite stuttering nor terminating behaviours, Back and von Wright have proved that any trace refinement can be proved by a combination of backward and forward simulation [2].

9 Conclusion

We have identified how the mapping from continuous action systems to action systems may be adjusted such that action system trace semantics are valid for continuous action systems, and we have defined a stream semantics that is complementary to the trace semantics, but is more general. Our results indicate that action system data refinement rules are applicable to continuous action systems with stream semantics, but they are not complete. Subsequently, we constructed and verified a new stream refinement rule that is capable of proving stream refinements that are not possible in the more restrictive trace semantics. For a certain subclass of continuous action systems we have shown that our new stream refinement rule, in conjunction with the existing action system data refinement rules, are as complete (with respect to our stream semantics) as the data refinement rules are for action systems with trace semantics. This work enables the continuous action systems formalism to be used to reason about the derivation of hybrid systems.

Acknowledgements. This research was supported by Australian Research Council (ARC) Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems.* We would like to thank Robert Colvin for feedback on earlier drafts of this paper, and the anonymous referees for their suggested improvements.

References

1. Back, R.-J., Petre, L., Porres, I.: Generalizing action systems to hybrid systems. In Joseph, M., ed.: Proc. of 6th Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2000. Vol. 1926 of Lect. Notes in Comput. Sci. Springer-Verlag (2000) 202–213
2. Back, R.-J., von Wright, J.: Trace refinement of action systems. In Jonsson, B., Parrow, J.: Proc. of 5th Int. Conf. on Concurrency Theory, CONCUR '94. Vol. 836 of Lect. Notes in Comput. Sci. Springer-Verlag (1994) 367–384
3. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag (1998)

4. Back, R.-J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In Proc. of 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, PODC '83. ACM Press (1983) 131–142
5. Back, R.-J., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Trans. on Program. Lang. and Syst. **10**(4) (1988) 513–554
6. Back, R.-J., von Wright, J.: Reasoning algebraically about loops. Acta Inform. **36**(4) (1999) 295–334
7. Cohen, E.: Hypotheses in Kleene algebra. Techn. Report TM-ARH-023814. Belcore (1994)
8. Cohen, E.: Separation and reduction. In Backhouse, R. C., Oliveira, J. N., eds.: Proc. of 5th Int. Conf. on Mathematics of Program Construction, MPC 2000. Vol. 1837 of Lect. Notes in Comput. Sci. Springer-Verlag (2000)
9. Dijkstra, E. W.: A Discipline of Programming. Prentice Hall (1976)
10. Hayes, I. J.: Reasoning about real-time repetitions: Terminating and non-terminating. Sci. of Comput. Program. **43**(2–3) (2002) 161–192
11. Hayes, I. J.: Programs as paths: An approach to timing constraint analysis. In Dong, J. S., Woodcock, J., eds.: Proc. of 5th Int. Conf. on Formal Engineering Methods, ICFEM 2003. Vol. 2885 of Lect. Notes in Comput. Sci. Springer-Verlag (2003) 1–15
12. Kozen, D.: Kleene algebra with tests. ACM Trans. on Program. Lang. and Syst. **19**(3) (1997) 427–443
13. Morgan, C.: Programming from Specifications. 2nd edn. Prentice Hall (1994)
14. von Wright, J.: From Kleene algebra to refinement algebra. In Boiten, E. A., Möller, B., eds.: Proc. of 6th Int. Conf. on Mathematics of Program Construction, MPC 2002. Vol. 2386 of Lect. Notes in Comput. Sci. Springer-Verlag (2002) 233–262