CONTROL SYSTEMS AS ACTION SYSTEMS

M. Rönkkö [1] E. Sekerinski[2] K. Sere[3]

[1] Department of Computer Science and Applied Mathematics, University of Kuopio, P.O.Box 1627, FIN-70211 Kuopio, Finland, Mauno.Ronkko@uku.fi
[2] Department of Computer Science, Åbo Akademi, FIN-20520 Turku, Finland, Emil.Sekerinski@abo.fi
[3] Department of Computer Science and Applied Mathematics, University of Kuopio, P.O.Box 1627, FIN-70211 Kuopio, Finland, Kaisa.Sere@uku.fi

Keywords
action systems, discrete event systems, formal verification, PVS, weakest preconditions

Abstract

We discuss how the action systems formalism can be used in constructing controllers for discrete event systems. Action systems are based on predicates and predicate transformers. Our approach is exemplified through applying action systems into a real-world control problem. We show, how the reachability of safe states in infinite unpredictable system, the safety conditions, can be stated as simple predicates. Verification is carried out in a standard manner with the weakest precondition calculus of Dijkstra.

1 Introduction

Action systems, originally proposed by Back and Kurki-Suonio [2], are predicate transformer based systems that operate on some state base. Actions resemble events, but are more precise, as they express algorithmically, how the state base is changed. In action systems, actions are executed in a do-loop which iterates as long as any of the actions is enabled. In this loop, actions are connected with a choice operator which models parallel nondeterministic choice between enabled actions. This allows compact description of unpredictable behaviour. Therefore, action systems have been successfully applied in many non-trivial applications when modelling reactive and concurrent behaviour, see e.g.[3, 5].

The use of predicate transformers in discrete event systems is not new, see e.g. [9]. The novelty in our approach is that both the controller and its environment are modelled uniformly by action systems. This allows reasoning of combined behaviour with simple predicates.

1.1 Background of the work

This work is part of ongoing activity where formal methods are tried out in several practical examples together with industrial partners. Applications are mainly concurrent, safety-critical systems from the fields of medicine and health care. The example presented in this paper is carried out in co-operation with a Finnish company manufacturing diagnostics devices and laboratory equipments for hospitals and clinics.

1.2 Overview

We start by briefly describing actions and action systems in section 2. A general approach of how to model a controller and its environment with action systems is given in section 3. Safety condition verification and directions towards its mechanical verification are also presented in that section. The approach is then applied to a conveyor belt system in section 4.

2 Actions and action systems

2.1 Actions

An action is any statement in an extended version of Dijkstra's guarded command language [8]. It includes abort and skip actions, (multiple) assignment, sequential composition, nondeterministic choice, conditional, and iteration. The language is defined using the *weakest precondition* predicate transformer $\mathrm{wp}(A, q)$, which for any action A and postcondition q yields the weakest precondition such that A terminates and establishes q. Pre- and postconditions are predicates over the state variables. Function $q[e/x]$ denotes the substitution of free occurrences of variables x with expressions e in predicate q. For actions A, B we define:

$$\begin{array}{lcl} \mathrm{wp}(\mathit{abort}, q) & = & \mathit{false} \\ \mathrm{wp}(\mathit{skip}, q) & = & q \\ \mathrm{wp}(x := e, q) & = & q[e/x] \\ \mathrm{wp}(A;\ B, q) & = & \mathrm{wp}(A, \mathrm{wp}(B, q)) \\ \mathrm{wp}(A \parallel B, q) & = & \mathrm{wp}(A, q) \ \wedge \ \mathrm{wp}(B, q) \end{array}$$

This language is extended by allowing pure guarded commands of the form $g \to A$, where g is a predicate, with the meaning:

$$\mathrm{wp}(p \to A, q) \quad = \quad p \ \Rightarrow \ \mathrm{wp}(A, q)$$

proc. of the workshop on Discrete Event Systems

Operationally, this action waits till guard g holds and then executes the body A.

The *guard* of an action A is the condition $g\,A$, defined by:

$$g\,A \;=\; \neg \mathrm{wp}(A, \mathit{false}).$$

An action A is said to be *enabled* when its guard is true. For example, we have:

$$\begin{array}{lcl} g\,(x := e) & = & \mathit{true} \\ g\;\mathit{abort} & = & \mathit{true} \\ g\;\mathit{skip} & = & \mathit{true} \\ g\,(A \parallel B) & = & g\,A \;\vee\; g\,B \\ g\,(p \to A) & = & p \;\wedge\; g\,A \end{array}$$

Having guarded actions allows us to define conditional and iteration concisely by

$$\begin{array}{lcl} \mathrm{wp}(\mathbf{if}\ A\ \mathbf{fi}, q) & = & \mathrm{wp}(A, q) \;\wedge\; g\,A \\ \mathrm{wp}(\mathbf{do}\ A\ \mathbf{od}, q) & = & (\forall\, n.\mathrm{wp}(A^n, g\,A \;\vee\; q)) \;\wedge \\ & & (\exists\, n.\neg\, g\,A^n) \end{array}$$

where $A^0 = \mathit{skip}$ and $A^{n+1} = A^n;\ A$. In the definition of iteration, conjuncts correspond to partial correctness and termination respectively.

2.2 Action systems

An *action system* $\mathcal{A}$ has the form

$$\mathcal{A} \;=\; |[\ \mathbf{var}\ x := x0; \mathbf{do}\ A\ \mathbf{od}\ ]| : z$$

First, local variables x are initialized to $x0$, and then action A is repeatedly executed. This execution terminates only when A is no longer enabled, and aborts when A aborts.

Local and *global* variables of $\mathcal{A}$ are variables x and z respectively. These variables form the *state variables* of $\mathcal{A}$. All of them are distinct and associated with an explicit type. Actions are allowed to refer to any state variable. In the following, we use the keywords **global** for global and **var** for local variables.

The action A is typically of the form:

$$A \;\widehat{=}\; A_1 \parallel \ldots \parallel A_m$$

Operationally, any of the enabled actions A_i in A is selected nondeterministically for execution. The execution of an action is always atomic. If two actions refer to disjoint variables, their execution can be in any order or in parallel. Hence, this models parallelism by interleaving. Action systems are similar to the Unity logic of Chandy and Misra [6] and are also related to Lamports Temporal Logic of Actions [10]. However, contrary to these approaches there are no assumptions about the fairness in the selection of actions; fairness is not needed here.

2.3 Composition of action systems

Consider two action systems $\mathcal{A}$ and $\mathcal{B}$

$$\begin{array}{lcl} \mathcal{A} & = & |[\ \mathbf{var}\ x := x0; \mathbf{do}\ A\ \mathbf{od}\ ]| : z \\ \mathcal{B} & = & |[\ \mathbf{var}\ y := y0; \mathbf{do}\ B\ \mathbf{od}\ ]| : v \end{array}$$

where x is disjoint from y. We define the *parallel composition* $\mathcal{A} \parallel \mathcal{B}$ to be the action system

$$\begin{array}{lcl} \mathcal{C} & = & |[\ \mathbf{var}\ x, y := x0, y0; \\ & & \quad \mathbf{do}\ A \parallel B\ \mathbf{od}\ ]| : z \cup v. \end{array}$$

Thus, it combines the state spaces of the two constituent action systems, merging the global variables and keeping the local variables distinct. The behaviour of a parallel composition depends on how the individual action systems, the *reactive components*, interact with each other via the global variables that they reference. For instance, a reactive component does not terminate by itself: termination is a global property of the composed action system [1].

The modelling of control systems is based on the *prioritizing composition* of action systems [15]. Action system $\mathcal{C} = \mathcal{A} \;//\; \mathcal{B}$ combines the action systems $\mathcal{A}$ and $\mathcal{B}$ so, that preference is given to the actions of $\mathcal{A}$. Therefore the action B can only be selected for execution when the action A is not enabled, otherwise the action A is executed:

$$\begin{array}{lcl} \mathcal{C} & = & |[\ \mathbf{var}\ x, y := x0, y0; \\ & & \quad \mathbf{do}\ A \parallel \neg\, g\,A \to B\ \mathbf{od}\ ]| : z \cup v. \end{array}$$

3 Specification approach

3.1 Overall framework

We start by defining a uniform framework for the entire system. It is modelled as a combined action system, where the controller has priority over its environment. When a controller action is enabled, it will be immediately taken, similarly to an interrupt procedure. This behaviour is captured by *Controller // Environment*.

Unobservable actions, i.e. unobservable events in [13], can be presented as stuttering steps in the environment. Stuttering steps are actions that modify the local variables of the action system. This kind of activity does not change the values of any externally visible variables, and thus, is completely invisible to other action systems. The amount of stuttering steps in any action system is assumed to be finite.

Similarly, uncontrollable actions can be introduced as actions in the environment. Since any action may abort the entire action system through *abort* statement, we can even model aborting behaviour in the entire system, including machine breakdowns and other external disturbances.

Sensors observed by the controller and actuators that control the physical environment are modelled as variables which can be read by all of the actions. Sensors can be modified by the environment actions and the actuators by the controller actions. Sensors and actuators are initialized by an action *Init*, which is assumed to be executed before the main loop. Thus, our model is formalized as:

$$System \;\widehat{=}\; Init;\,(Controller \;//\; Environment)$$

When modelling with action systems, any variables can be made unobservable by hiding them. Technically hiding means making the variables local to some action system. Similarly, variables can be revealed by declaring them global.

3.2 Safety condition verification

System can be seen controllable, if the controller guarantees the *safety*. The safety condition is expressed as a predicate over the state space of the whole control system. For proving purposes, we also need an *invariant*. It gives a bound on the set of all reachable states, both safe and unsafe ones. Let predicates *safety*, *inv* and action systems

$$\begin{array}{lcl} Controller & = & |[\ \mathbf{var}\ x := x0;\ \mathbf{do}\ C\ \mathbf{od}\ ]|\ :\ z \\ Environment & = & |[\ \mathbf{var}\ y := y0;\ \mathbf{do}\ E\ \mathbf{od}\ ]|\ :\ v \end{array}$$

be given. We know that an environment action may lead to an unsafe state, and that the controller must re-establish the safety. This allows us to state the safety condition as following proof obligations [16]:

1. Initialization establishes *inv*:

   $$\mathrm{wp}((Init;\, x, y := x0, y0), inv)$$

2. *Environment* preserves *inv*, provided that *safety* holds:

   $$inv \;\wedge\; safety \quad\Rightarrow\quad \mathrm{wp}(E, inv)$$

3. *Controller* eventually establishes *safety* and preserves *inv*:

   $$inv \quad\Rightarrow\quad \mathrm{wp}(\mathbf{do}\ C\ \mathbf{od}, inv \;\wedge\; safety)$$

This also shows that controllability is bound to the detail level of the environment specification. An environment specification with unobservable and uncontrollable actions requires more robust controller specification or less binding safety condition than a fully observable environment specification.

3.3 Mechanical verification of safety

The verification of the proof obligations above can be carried out mechanically. A theorem prover called HOL has been successfully used for this purpose [16]. Also, other verification systems can be used. We have used the PVS prover [7] for verifying the obligations in this paper [14]. PVS, Prototype Verification System, has been developed at SRI International. Its use is aimed at specifying and verifying digital systems, and has been succesfully used e.g. in verification of fault-tolerant architectures [12].

3.4 Refinement into an executable program

Action systems can be further refined by using *refinement calculus* [1]. Shortly described, refinement of an action system means either incorporation of new internal activity or modification of old activity under strict guidelines, which are stated in *refinement relation*. Refinement relation, in turn, must fulfill obligations set by *simulation* characteristics, which essentially state that all observable behaviour of a refined action system must be a part of the behaviour of the original action system as well. The refinement calculus can be used for example in formal translation of action system into executable program code.

Even though our framework combines both the controller and the environment into one system, it is still an open model. This is provided by the state space partitioning in the action systems. Therefore, independent refinement of either the controller or the environment specification using refinement calculus is well encouraged [5].

4 The conveyor belt system

4.1 The conveyor belt control system

Our example is part of an autonomous system making laboratory tests. In style, it is similar to the production cell in [11]. Three belts carry cassettes containing test samples. *Inbelt* brings in new cassettes which are processed on the *main lane* and transported away on the *outbelt*. The belts are operated via motors controlled by microprocessors in a local network. We leave out the outbelt for brevity, as it works very much like the inbelt. Therefore, there are only two motors used as actuators to control the belts.

A sensor called *entry* is at the end of the inbelt. It marks that a cassette is ready to enter the main lane. There are three sensors on the main lane: *entered* marks when the cassette has entered the processing part, *exit* marks when it has been processed, and the purpose of *slant* is to mark that a cassette is in a position where a new incoming cassette could cause slanting. The danger of jamming exists, if there is not enough space between the cassettes on the main lane. Figure 1 gives a schematic picture of the con-

veyor belt system. Figure 2 illustrates slanting and jamming of cassettes.

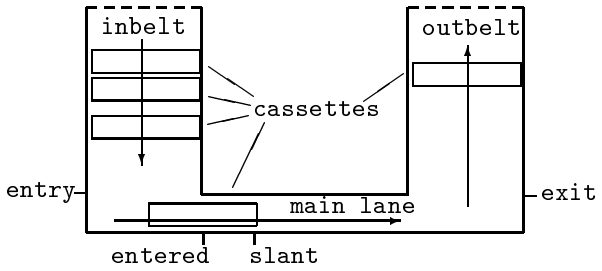

Figure 1: A schematic view of the conveyor belt system.

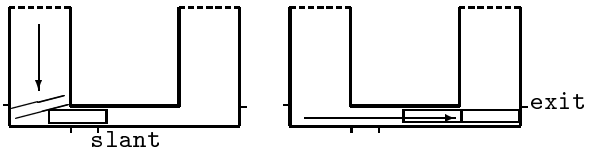

Figure 2: Left: Slanting of cassettes; Right: Jamming of cassettes.

We aim at showing that, with a suitable level of abstraction, a formal specification of the system can be developed and stated concisely. Thus, certain real time and capacity requirements are not considered.

4.2 Sensors and actuators

We start by specifying the *Init* action, whose purpose is to initialize all the sensors and the actuators in the system. Therefore, we have to first introduce them as variables.

The sensor readings are given by variables:

global $entry,\ entered,\ slant,\ exit$: Bool

Following variables represent the belt actuators, where *inbelt* corresponds to the motor status of the inbelt and *running* to that of the main lane:

global $inbelt,\ running : on \mid off$

Initially, we assume that both belts are on, and there are no cassettes on them:

$$\begin{array}{rl} & inbelt, running := on, on; \\ Init \mathrel{\widehat{=}} & entry, entered := false, false; \\ & slant, exit := false, false \end{array}$$

4.3 Environment specification

The second phase in constructing the specification is to define the *Environment* action system. For this, we first need to identify possible local variables and then formalize the behaviour in separate actions, which are later bound by the *Environment*.

4.3.1 Local variables

The environment has two local variables *ready* and *processing* which tell the position of a cassette on the main lane:

var $ready,\ processing$: Bool

The value of *ready* equals to a cassette being at the beginning of the main lane, and the value of *processing* equals to a cassette being processed.

4.3.2 Environment actions

The physical behaviour of the environment is captured by following actions. The general condition in these actions is that belts are running.

Firstly, a cassette appears eventually to the inbelt:

$$E1 \mathrel{\widehat{=}} \begin{array}{l} inbelt = on \wedge \\ entry = false \end{array} \rightarrow\ entry := true$$

Cassettes are registered to be on the main lane one at the time:

$$E2 \mathrel{\widehat{=}} \begin{array}{l} inbelt = on \wedge \\ entry = true \wedge \\ ready = false \end{array} \rightarrow \begin{array}{l} entry := false; \\ ready := true \end{array}$$

A cassette at the beginning of the main lane enters the processing part, provided that there are no cassettes at the slant sensor:

$$E3 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ ready = true \wedge \\ entered = false \wedge \\ slant = false \end{array} \rightarrow \begin{array}{l} entered := true; \\ ready := false \end{array}$$

An entered cassette reaches the slant position, provided that there are no cassettes being processed:

$$E4 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ entered = true \wedge \\ slant = false \wedge \\ processing = false \end{array} \rightarrow\ slant := true$$

After passing the slant position, cassette enters the processing part completely:

$$E5 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ entered = true \wedge \\ slant = true \wedge \\ processing = false \end{array} \rightarrow \begin{array}{l} entered := false; \\ processing := true \end{array}$$

During its processing, the cassette fully passes the slant sensor:

$$E6 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ slant = true \wedge \\ processing = true \end{array} \rightarrow\ slant := false$$

The processed cassette reaches the exit position:

$$E7 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ slant = false \wedge \\ exit = false \wedge \\ processing = true \end{array} \rightarrow \begin{array}{l} exit := true; \\ processing := false \end{array}$$

A cassette at the exit position finally disappears from the system (outbelt is sometimes on):

$$E8 \mathrel{\widehat{=}} exit = true \rightarrow exit := false$$

Now, the environment is given as:

$$\begin{array}{lll} Environment & \mathrel{\widehat{=}} & |[\ \mathbf{var}\ ready := false; \\ & & \quad processing := false; \\ & & \quad \mathbf{do}\ E1 \parallel \ldots \parallel E8\ \mathbf{od} \\ & &]|\ : entry, entered, slant, exit, \\ & & \quad inbelt, running \end{array}$$

4.4 Controller specification

Controller actions control the movement of cassettes on the two conveyor belts by switching the respective motors on and off. In the following, controller actions are already worked out. However, the formal definition of the environment and the safety condition can be used in the derivation of the controller actions [16].

In order to prevent jamming, the main lane must be turned off if there is a cassette at exit position:

$$C1 \mathrel{\widehat{=}} \begin{array}{l} running = on \wedge \\ exit = true \end{array} \rightarrow running := off$$

The main lane can be turned on again when the cassette at exit position has moved away:

$$C2 \mathrel{\widehat{=}} \begin{array}{l} running = off \wedge \\ exit = false \end{array} \rightarrow running := on$$

The inbelt should be switched off in case there is a danger of slanting:

$$C3 \mathrel{\widehat{=}} \begin{array}{l} inbelt = on \wedge \\ entry = true \wedge \\ entered = true \wedge \\ slant = true \end{array} \rightarrow inbelt := off$$

As soon as the cassette has completely entered the processing part, the inbelt can be turned on again:

$$C4 \mathrel{\widehat{=}} \begin{array}{l} inbelt = off \wedge \\ entered = false \end{array} \rightarrow inbelt := on$$

Hence, the controller is as follows:

$$\begin{array}{lll} Controller & \mathrel{\widehat{=}} & |[\ \mathbf{do}\ C1 \parallel C2 \parallel C3 \parallel C4\ \mathbf{od} \\ & &]|\ : entry, entered, slant, exit, \\ & & \quad inbelt, running \end{array}$$

4.5 Verification of the safety condition

Before the verification can be done using the presented verification rule, *safety* condition and invariant *inv* have to be defined. The safety condition in the conveyor system is that the controller should not allow the cassettes to slant or cause jamming.

$$safety \mathrel{\widehat{=}} \neg slanting \wedge \neg jamming$$

More precisely, there is a danger of jamming, if there is a cassette is at the end of the running main lane. Then, another cassette moving on the main lane could bump into it:

$$jamming \mathrel{\widehat{=}} running = on \wedge exit = true$$

Slanting could occur, if a cassette on the main lane has reached the slant position, the inbelt is on, and another cassette is at the entry position:

$$slanting \mathrel{\widehat{=}} \begin{array}{l} inbelt = on \wedge entry = true \wedge \\ slant = true \wedge entered = true \end{array}$$

For verifying the safety condition in our example the invariant *true* is sufficient (note that in other examples more complicated invariants are necessary [16]). This leads to the following proof obligations:

1. $\mathrm{wp}\ (\ (Init;\ ready, processing := true, true),\ true)$
2. $safety \Rightarrow \mathrm{wp}(E1 \parallel \ldots \parallel E8, true)$
3. $\mathrm{wp}(\mathbf{do}\ C1 \parallel \ldots \parallel C4\ \mathbf{od}, safety)$

As mentioned earlier, we used PVS for verifying that our specification fulfills these obligations [14].

5 Conclusions

We studied the problem of constructing discrete event system controllers within the action system framework.

That there are three major advantages in using this approach: (1) Both the system and its environment can be modelled within the same unifying framework, making it feasible to carry out reasoning about the systems behaviour as a whole. (2) Action systems are originally designed for the derivation of parallel and distributed systems, which makes it natural to derive and reason about distributed control of a discrete event system in the sense of [4] for instance. (3) Action systems are intended to be stepwise developed within the *refinement calculus* [1]. This gives us a formal framework to transform the controller specification into an efficient control program.

The example was taken from an implementation project. In a closely related paper, Butler, Sekerinski, and Sere [5] show how a high level specification of a steam boiler is created within the action systems framework using several levels of abstractions and the refinement calculus.

5.1 Acknowledgements

This research was partly supported by the Technology Development Centre of Finland, Tekes. Discussions with Petri Luostarinen concerning the conveyor belt system were helpful.

References

[1] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*. Springer–Verlag, 1990.

[2] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[3] R. J. R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17-30, 1991.

[4] A. Bergeron. Sharing out control in distributed processes. *Theoretical Computer Science* 139 (1995) 163–186. Elsevier.

[5] M. Butler, E. Sekerinski, and K. Sere. Steam boiler specification and refinement. To be published, preliminary version presented at the Seminar on *Methods for Semantics and Specification*, Schloss Dagstuhl, June 1995.

[6] K. M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addision–Wesley, 1988.

[7] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas. A Tutorial Introduction to PVS. Presented at *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

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

[9] R. Kumar, V. Garg, and S. I. Marcus. Predicates and predicate transformers for supervisory control of discrete event dynamical systems. *IEEE Transactions on Automatic Control*, 38, No. 2, February 1993.

[10] L. Lamport. *A Temporal Logic of Actions*. Research Report No. 57, DEC System Research Center, 1990.

[11] C. Lewerentz, Th. Lindner (Eds). *Formal Development of Reactive Systems - Case Study Production Cell*. volume 891 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995

[12] S. Owre, J. Rushby, N. Shankar, F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, vol. 21, No. 2, February 1995.

[13] P. J. Ramadge and W. N. Wonham. The control of discrete event systems. *Proc. of IEEE: Special lssue on Discrete Event Systems: Models and Applications*, 77:81–98, 1989.

[14] M. Rönkkö. *Using PVS for Verifying Safety in Action Systems*. Technical Report A/1996/2, Department of Computer Science and Applied Mathematics, University of Kuopio, 1996.

[15] E. Sekerinski and K. Sere. *A Theory of Prioritizing Composition*. TUCS Technical Report, No 5, May 1996.

[16] E. Sekerinski. *Deriving Control Programs using Weakest Preconditions*. TUCS Technical Report, No 4, May 1996.