

# Animation of Data Refinements

Neil J. Robinson and Colin Fidge

Software Verification Research Centre, The University of Queensland, Australia

njr@svrc.uq.edu.au cjf@svrc.uq.edu.au

## Abstract

*Refinement is the process of deriving verifiably-correct software from its specification. In practice, however, refinement steps are complex and difficult to prove correct. We show how animation can be used to provide insights into the correctness, or otherwise, of refinement steps for the most general form of data refinement in which the whole system design can be changed in a single step.*

## 1 Introduction

The concept of refinement, i.e., formally deriving programs from their specifications, is well understood in the formal methods community. However, refinement is hardly used at all in industry except in the most critical applications. Many refinement steps are accompanied by ‘side conditions’ whose satisfaction involves the same intellectual challenges as theorem proving. To make refinement theory accessible to industry, techniques for helping programmers understand refinement steps are essential.

Animation tools can be effective in the validation of formal specifications [13]. Previously, we proposed the use of animation tools to help validate simple refinements [21, 20]. In this paper we extend that work to consider “whole system” data refinements, in which the entire system design can change in a single refinement step. We illustrate the approach with an action system refinement [1], based on Peterson’s mutual exclusion algorithm [5].

## 2 Comparison with Related Work

Our overall aim is to use animation to support reasoning about the correctness of refinement steps from formal specifications. Relevant previous work includes uses of animation for analysing properties of specifications.

Miller and Strooper investigated the use of animation tools for testing a specification systematically [16]. However, they define a method for systematically deriving test cases from the specification under test, whilst we consider

verifying and validating specifications against more abstract specifications.

In another paper, Miller and Strooper show how the results of animating a specification can be used to test an implementation of the specification [17]. Their method makes use of data refinement techniques, within the limits of the Z specification language. Z does not include guards and its refinement approach relies on explicit matching of each abstract operation to a concrete operation. In our work, guards and preconditions can be refined separately, and the whole system design (including all the operations) can be changed in a refinement step. Also our focus is on interactively exploring the specifications to get a better understanding of the refinement and to allow the user to creatively look for errors.

The Nitpick [14] and the more recent Alcoa [15] tools, check properties of specifications written in Alloy, a subset of Z. Alcoa specialises in producing counter-examples that show cases where the property does not hold and has been optimised to perform this task efficiently. Our primary motivation for using an animation tool is to permit the user to interactively check refinements, in a manner analogous to testing. Such an approach could be complemented by automated tools such as Alcoa.

Waeselynck and Behnia examine the relationship between formal refinement and testing via animation [22]. They propose a process in which an animator is used to test a specification against informal user requirements. Their work differs from ours in that they propose the use of an animator for testing a specification against test cases and test oracles, which are derived from an understanding of informal user requirements. We use the animator to check refinement steps that relate an abstract specification to a concrete one.

Other refinement checking tools exist, such as the Program Refinement Tool [23] and the Refinement Calculator [6]. However these tools are aimed at supporting formal refinement *proofs*. The tools support the user by providing libraries of theorems that are useful in proving refinements and by automating certain routine tasks, including the documentation of the proof process. Our focus is on the more

modest, but practical, goal of helping the programmer visualise refinement steps.

In our previous work [21], we proposed two different approaches to visualising simple refinements using a specification animation tool, one aimed at visualising trace refinements and one at visualising state machine refinements. Subsequently [20], we further developed the approach to animating state machine refinement, by devising a method for checking Z data refinements using an animation tool.

In this paper we focus instead on whole-system trace refinements, using the action system formalism [2]. This is the most general form of refinement, because the entire system design is allowed to change, and consequently involves the most challenging proof obligations. We also develop a general approach to animating refinements that is suitable for both trace refinements and state machine refinements, and is well suited to the type of animation tool we use.

### 3 Background: Data Refinement

If we have a system which conforms to an abstract specification  $A$  and it is replaced by a system which conforms to a concrete specification  $C$ , and if an external observer could never detect that the replacement has occurred, we say that specification  $C$  is a *refinement* of specification  $A$  [10].

Refinements are usually conducted in a series of small steps, moving from an abstract specification towards a concrete specification or code, a process known as *stepwise refinement*. A specification typically consists of some data, a set of operations on that data and a definition of how the operations should be used. In *operation refinements*, individual operations are refined, with the monotonicity of the refinement relation ensuring that the refinement holds for the whole system. In *data refinement*, the local data used by the system operations is refined, and so, as part of the refinement step, the system operations which access that data may need to change. Potentially, in data refinements, the whole system specification can change providing the externally observable behaviour remains the same.

Different formalisms take different approaches to data refinement. For example, they make different assumptions about what parts of system behaviour are externally observable. However, all the forms of data refinement are based on the idea of *simulation*.

We can visualise a system behaviour as a graph, each node of which represents a state of the system, and each arc of which represents an operation which leads from one state to the next. If system  $AS$  is refined by system  $CS$ , then we can visualise their behaviours with a *commuting diagram*, as shown in Figure 1. Corresponding states of the two systems are related via an abstraction relation  $R$ . The basic idea is that for any path through the graph via a concrete operation, there should always be an alternative path

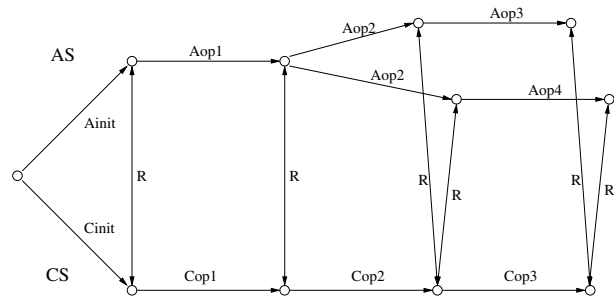


Figure 1. Commuting Diagram

(or *simulation*) to the same state via an abstract operation. This form of simulation is called a *forward simulation*. In certain situations, for example when non-determinism in the abstract specification is postponed in the concrete specification, *backward simulation* is required to prove the refinement. In this paper, we will deal only with forward simulations. De-Roeover and Englehardt provide a comprehensive survey of the various types of simulation in their book on data refinement [7].

## 4 Action Systems

### 4.1 Action System Specifications

Action systems [2] use Guarded Command Language notation to define an interleaving, state-machine simulation of concurrent behaviours. Each action appears as a (multiple) assignment protected by a Boolean guard. An outermost **do...od** loop nondeterministically selects actions with true guards until none remain.

An action system, based on an example by Back [1], is shown in Figure 2. The action system  $SP0$  is intended to specify two concurrent processes, which together update a shared variable  $w$ . One process increases  $w$  by one (via action  $CS.0$ ), whilst the other process increases  $w$  by two (via action  $CS.1$ ). Both processes update  $w$  in two steps using a variable  $y.i$  local to process  $i$ . Actions  $CS.0$  and  $CS.1$  are both executed atomically (i.e., they cannot be interrupted). The motivation for this example is to use it as the starting point for a stepwise refinement to an action system in which the actions  $CS.i$  are not executed atomically, but mutually exclusive access to  $w$  is preserved nevertheless. The variable and action names are chosen with this in mind —  $CS$  represents the critical section,  $NS$  the non-critical section, and  $cr.i$  is a flag which is *true* when process  $i$  is in its critical region.

The action system declares local variables  $y.0, y.1, cr.0$  and  $cr.1$ , with  $cr.0$  and  $cr.1$  being initialised to *false*. The **do...od** loop contains two actions. Each action is specified

```

var  $y.i : \mathbb{N}; cr.i : \mathbb{B}$  for  $i = 0, 1$ 
init  $cr.i := false$  for  $i = 0, 1$ 
do
  ( $\square cr.i \rightarrow y.i := w + i + 1; w := y.i; cr.i := false$  for  $i = 0, 1$ ) [CS.i]
  ( $\square \neg cr.i \rightarrow cr.i := false \square cr.i := true$  for  $i = 0, 1$ ) [NS.i]
od :  $w : \mathbb{N}$ 

```

**Figure 2. Example action system specification  $SP0$**

by a guard before the arrow, and a statement after the arrow. The ‘;’ operator in the statements is sequential composition. Statements do not change the values of variables they do not specifically assign values to. Externally visible variables are listed at the end of the specification. In this case, there is one such variable,  $w$ . In action  $NS.i$ , we use the  $\square$  operator to indicate that this action nondeterministically updates  $cr.i$  to either *true* or *false*.

Action system semantics is defined in terms of *behaviours*, which are sequences of steps, each step consisting of a pair of the local state and the externally visible state. A *private step* is a step in an action system behaviour with the same externally visible state as the previous step. A *trace* of an action system is its externally visible behaviour, and is obtained by removing all *finite* sequences of private steps and then removing all the local states from the sequence [3]. Our example specifies all traces in which  $w$  can only increase by either 1 or 2 at each externally visible step.

## 4.2 Action System Refinement

Informally, an action system  $\mathcal{A}$  is refined by action system  $\mathcal{C}$  if every trace of  $\mathcal{C}$  is also a trace of  $\mathcal{A}$  [3].

Action system refinement allows for changes to guards and statements as well as the addition of *stuttering actions*. A stuttering action is an action of the concrete specification which does not change the externally visible state or the parts of the local state that are related to the abstract state via the abstraction relation. In terms of a simulation, each concrete stuttering action is effectively equivalent to the abstract action system doing nothing.

Since the action system semantics is defined in terms of traces of variables only, the individual actions used in each iteration of the action system loop are not significant. The refinement rules are stated in terms of the whole action system — there is no need to match up the individual actions in the two specifications. It is however necessary to know whether a new action is a stuttering action or not.

The action system refinement rules are defined using the weakest precondition predicate transformer [8]. For a program  $prog$  and a postcondition  $A$ ,  $wp(prog, A)$  is the weakest precondition sufficient to ensure termination in a state described by  $A$  [18].

An action system  $\mathcal{A}$  is defined by its local variables  $x = x_0, x_1, \dots$ , the initial values of its local variables  $x_{init} = x_{0init}, x_{1init}, \dots$ , its externally visible variables  $z = z_0, z_1, \dots$  and its actions  $A = A_0, A_1, \dots$ , where each action  $A_i \stackrel{def}{=} gA_i \rightarrow sA_i$  consists of a guard  $gA_i$  and a statement  $sA_i$  [1].

The alternate actions  $A_0, A_1, \dots$  comprising an action system together make up a single action  $A$ . So,

$$\begin{aligned}
 A &= A_1 \square \dots \square A_m \\
 &= gA_1 \rightarrow sA_1 \square \dots \square gA_m \rightarrow sA_m \\
 &= \bigvee_{i=1}^m gA_i \rightarrow \mathbf{if} A_1 \square \dots \square A_m \mathbf{fi}
 \end{aligned}$$

Let  $gA$  be  $\bigvee_i gA_i$  and  $sA$  be  $\mathbf{if} A_1 \square \dots \square A_m \mathbf{fi}$ . Then the whole action system  $\mathcal{A}$  can be considered to consist of a single action  $A$ , i.e.,

$$\mathcal{A} = \mathbf{begin} \mathbf{var} x \mathbf{do} A \mathbf{od} \mathbf{end} : z$$

This concept is used throughout the definition of the action system refinement rules.

An action  $A$  involving variables  $x : X$  and  $z : Z$  is *data refined* by action  $A'$  involving variables  $x : X, x' : X'$  and  $z : Z$  if there is a relation (called an *abstraction relation*) on these variables  $R : \mathbb{P}(X, X', Z)$  such that for all postconditions  $q$ :

$$RI \wedge wp(A, q) \Rightarrow wp(A', \exists x \bullet RI \wedge q)$$

where  $RI$  is shorthand for  $(x, x', z) \in R$ . If an action  $A$  is data refined by action  $A'$  via the abstraction relation  $R$ , then we write  $A \sqsubseteq_R A'$ .

Action system  $\mathcal{A}$  containing action  $A$ , is refined by action system  $\mathcal{A}'$  containing action  $A'$  that corresponds with the old action  $A$ , and a new stuttering action  $H'$  (which may consist of many individual stuttering actions  $H'_i$ ), if there exists a relation  $R : \mathbb{P}(X, X', Z)$  such that the following proof obligations hold [1].

1. *Initialisation:* For any initial value of  $z$  and any initialisation  $x'_{init}$  of  $\mathcal{A}'$ , there is some initialisation  $x_{init}$  of  $\mathcal{A}$  that establishes the abstraction relation.

$$\forall z, x'_{init} \bullet \exists x_{init} \bullet (x_{init}, x'_{init}, z) \in R$$

2. *Main Action:* Action  $A$  is data refined by action  $A'$ , via the abstraction relation  $R$ .

$$A \sqsubseteq_R A'$$

Note that the ‘A’ here refers to the combination of all the individual actions in the action system (as explained earlier).

3. *Exit Condition*: If action  $A$  is enabled then either the action  $A'$  is enabled, or a new stuttering action is enabled.

$$R \wedge gA \Rightarrow gA' \vee gH'$$

4. *Auxiliary Actions*: The stuttering actions do not affect the externally visible variables  $z$ .

$$SKIP \sqsubseteq_R H'$$

Here  $SKIP$  is an action whose guard is always enabled and which does not change the local state of action system  $\mathcal{A}$  or the externally visible state.

5. *Internal Convergence*: It is not possible to have an infinite sequence of stuttering actions.

$$R = wp([\mathbf{do} H' \mathbf{od}], true)$$

Recall that for some action  $A$ ,  $wp(A, true)$  characterises those states for which  $A$  is guaranteed to terminate [8].

When all these conditions hold, we write  $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ .

### 4.3 An Example Refinement

Back gives an example refinement from the action system shown in Figure 2, to an action system in which the two steps used to update variable  $w$  may be interrupted [1]. The refinement makes use of Peterson’s mutual exclusion algorithm [5].

In this paper, we examine the first two steps of the refinement. In the first step, Back introduces some stuttering actions, and additional variables which are used to control the order of the actions, and provides the necessary variables for implementation of the mutual exclusion algorithm. In the second step, he breaks up the actions in the critical section.

Action system  $\mathcal{SP}1$ , the outcome of the first step of the refinement, is shown in Figure 3.

Each process  $i$  owns a boolean variable  $b.i$  which it sets to *true* to indicate its intention to enter its critical section. To resolve contention when both processes wish to enter their critical sections a shared (atomically accessible) variable  $t$  of type  $\{0, 1\}$  is set to ‘ $i$ ’ to indicate process  $i$ ’s willingness to give way to the other process  $(1 - i)$ . Process  $i$  will enter its own critical section only if the other process  $(1 - i)$  has not indicated its intention to also do so ( $\neg b.(1 - i)$ ) or if the other process has given way to process  $i$  ( $t = 1 - i$ ).

As can be seen in Figure 3, action  $NS.i$  has not changed, but action  $CS.i$  has been refined to action  $CS'.i$ , and there are

$pc.i$	$b.i$	$cr.i$	$t$
0	$F$	$X$	$X$
1	$T$	$F$	$X$
2	$T$	$T$	$X$
3	$T$	$X$	$X$

**Figure 4. Abstraction relation for the first refinement step**

a number of new stuttering actions  $BS.i$ ,  $TS.i$  and  $BR.i$ . For each process  $i$  an additional ‘program counter’ variable  $pc.i$  is introduced to control the sequence in which process  $i$ ’s actions occur.

Back provides a suitable abstraction relation and a proof of this refinement step [1]. The abstraction relation is shown in Figure 4, and is expressed as an invariant on the new variables from action system  $\mathcal{SP}1$ , where  $X$  denotes ‘don’t care’.

The most complex part of the proof is a case analysis performed to prove the *exit condition* proof obligation. We omit the proof here, and focus instead on the second refinement step (see Section 7).

## 5 Representing Action Systems in Possum

We use the Z animation tool Possum [12], a part of the Cogito [11] toolset, to visualise and animate action system refinement.

This extends our previous work on visualising and animating Z refinements [21, 20]. Animators exist for other specification languages, such as B and CSP. However, these lack important features of Possum, such as the ability to execute implicit specifications and the ability to plug-in visualisations.

Possum interprets queries, written in Z, and responds with simplifications of those queries [12]. It can be used either to step through consecutive states of a state machine by ‘executing’ the operations of that machine, or to evaluate arbitrary expressions and predicates.

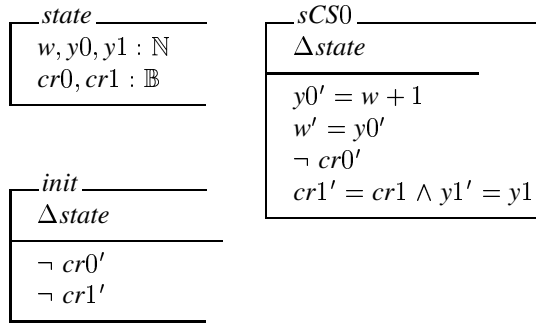
Z schemas are named specification components comprising variable declarations, and predicates relating the variables. By convention, individual Z operations, which we will use for modelling the statement part of actions, are specified by a schema whose declaration part includes  $\Delta state$  to define those variables that the operation may change. Unprimed variable names, e.g.,  $cr0$ , are the values of variables in the before state of the operation, and primed variable names, e.g.,  $cr0'$  are values of variables in the after state. The predicate part of the operation schema specifies the relationship between before and after states.

```

var ( $pc.i, y.i : \mathbb{N}; b.i, cr.i : \mathbb{B}$  for  $i = 0, 1$ );  $t : \{0, 1\}$ 
init  $b.i := false; pc.i := 0; cr.i := false$  for  $i = 0, 1$ 
do
  ( $\square cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1$  for  $i = 0, 1$ )           [BS.i]
  ( $\square pc.i = 1 \rightarrow t := i; pc.i := 2$  for  $i = 0, 1$ )           [TS.i]
  ( $\square pc.i = 2 \wedge (\neg b.(1 - i) \vee t = 1 - i) \rightarrow y.i := w + i + 1; w := y.i;$ 
     $cr.i := false; pc.i := 3$  for  $i = 0, 1$ )           [CS'.i]
  ( $\square pc.i = 3 \rightarrow pc.i := 0; b.i := false$  for  $i = 0, 1$ )     [BR.i]
  ( $\square \neg cr.i \rightarrow cr.i := false \square cr.i := true$  for  $i = 0, 1$ ) [NS.i]
od :  $w : \mathbb{N}$ 

```

**Figure 3. Action system  $SP1$ , the outcome of the first step of the refinement**



**Figure 5. Part of the Z representation of action system  $SP0$**

Whereas action systems have predicate transformer semantics, i.e., functions from predicates to predicates, the Z specification language is interpreted as having a relational semantics, i.e., predicates relating before and after states. For example, consider the operation  $sCS0$  shown in Figure 5, which achieves the same effect as the statement part  $sCS.0$  of action  $CS.0$  from Figure 2.

If we perform operation  $sCS0$  when  $cr0$  and  $cr1$  are *true* and  $w, y0$  and  $y1$  are all 0, then the animation tool responds with:

[...  $cr0' := false, cr1' := true, w' := 1, y0' := 1, y1' := 0$ ]

These bindings to the primed variables become the new current state.

Where there is nondeterminism in a specification, Possum picks the first binding that satisfies the specification. For example, if we call the *init* schema from Figure 5, it responds with

[ $cr0' := false, cr1' := false, w' := 0, y0' := 0, y1' := 0$ ]

Possum does this predictably, i.e., it will always respond as

above. The user of the animator can however tell Possum to pick an alternative binding by entering:

[*init* |  $w' = 1$ ]

This will respond as above but with  $w' := 1$ .

If an operation is called when its precondition is false, then Possum responds with *no solution*, meaning that it cannot find a binding from the current state in which both the pre and post conditions of the operation are *true*.

In order to animate the action system refinement using Possum, it is necessary to represent the whole action system example in Z. In Figure 5, we showed how it is straightforward to represent the statement part  $sA$  of an action  $A$  as a Z operation. However, it is also necessary to represent the guard  $gA$  of each action.

CSP-OZ [9] solves this problem by representing each guard as a separate schema, associated with its corresponding operation by a naming convention. However, for the purposes of animation, we really want the state of the guard predicates to be updated after each operation.

We therefore create a boolean variable for each of the guards, and set their values as part of the state invariant. As shown in Figure 6, which defines the guards for specification  $SP0$ , we also introduce a new variable *terminated*, which defines termination of the whole action system (when no guard is enabled).

Now we can represent all the action system variables, initialisation, guards and statements in Z. Although guards are only matched to operations representing statements by a naming convention, we create a plug in interface to the animator to prevent the user calling an operation when its guard is not enabled. This is achieved by providing a button for calling each operation, which is enabled only when the corresponding guard is *true* (see Section 6).

Note that the main **do...od** loop of the action system model does not appear in the Z representation. Its role is taken by the user's interaction with the animator itself. For

$\begin{array}{l} \text{state} \\ w, y0, y1 : \mathbb{N} \\ cr0, cr1 : \mathbb{B} \\ gCS0, gCS1, gNS0, gNS1, terminated : \mathbb{B} \\ \hline gCS0 = cr0 \wedge gCS1 = cr1 \\ gNS0 = \neg cr0 \wedge gNS1 = \neg cr1 \\ terminated = \\ \neg (gCS0 \vee gCS1 \vee gNS0 \vee gNS1) \end{array}$
---

Figure 6. Specification of guards in Z

the purposes of animation, we allow the user to pick any enabled action. Each execution of an action represents one iteration of the loop. The loop terminates when no actions are enabled — the user can do nothing.

## 6 Animation of Action System Refinements

### 6.1 Overview of Approach

The user’s aim in animating an action system refinement is to check that the refinement relationship is as intended, or to find errors in the refinement. This process is analogous to testing one specification against another, the aim being to find inconsistencies.

To animate a refinement from action system  $SP0$  to action system  $SP1$ , our approach is as follows:

#### 6.1.1 Preparation for the Animation

- We represent the action systems  $SP0$  and  $SP1$  as shown in Figures 5 and 6, but prefixing every variable and schema name with  $sp0$  or  $sp1$ , to make it clear which specification they belong to.
- We specify a combined state  $S$ , which includes both  $sp1state$  and  $sp2state$ , and a schema  $R$  relating the two states. This represents the abstraction relation  $R$ , discussed in Section 4.2, an example of which is shown in Figure 4.
- We add a counter variable  $iter$  for each of the specifications. This represents the current iteration of the action system. All operations increment their respective iteration variable, except  $SKIP$ .
- Using the visualisation capabilities of Possum, we provide a button for each action. We present the buttons in an intuitive layout. For example the buttons for the first example refinement step are shown in Figure 7. In this case, it is natural, for example, to place the  $SP0$

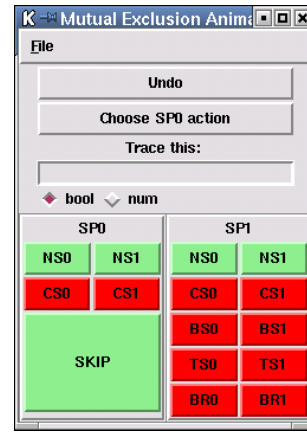


Figure 7. The interface to the animator

buttons for  $NS0$  and  $NS1$  side by side, and to line these buttons up with the similarly named buttons in  $SP1$ . Also note that we have provided one large  $SP0$   $SKIP$  button that lines up with the stuttering actions in  $SP1$ . This is a special  $SKIP$  operation, which increments the  $sp0iter$  iteration variable, but leaves all other variables unchanged. It is needed to keep the two models synchronised, so that  $SP0$  can perform an action whenever  $SP1$  performs one of its stuttering actions.

- We provide the user with a means of tracing variables from the two specifications. The following types of traces are supported:
  - Comparative traces, in which the tool plots a variable from each specification on the same graph.
  - Individual variable traces, in which the tool plots one variable value on a graph.
  - Predicate traces, in which the tool plots the truth of a predicate on a graph. This is a particularly powerful capability, allowing arbitrary properties to be observed.

#### 6.1.2 Performing the animation

The animation of the refinement proceeds in a series of steps. Each step is made up of two *turns*, beginning with the user’s turn, followed by the tool’s turn. The user always controls the concrete actions and the tool always controls the abstract actions.

In the first step, the user initialises the concrete specification, (in this case  $SP1$ ), by calling

$$sp1init \wedge sp0skip$$

and then the tool tries to match  $SP1$ ’s state, using:

$$sp0init \wedge sp1skip \wedge R'$$

Assuming that initialisation is successful, we move to the next step. Now the user can choose from the enabled concrete actions. For example, the user can choose to execute  $SP1$  action  $NS0$ , by clicking on the appropriate button (which is green when enabled). This causes another call to the animator:

$$sp1sNS0 \wedge sp0skip$$

It is the tool's turn again — the user selects the “choose  $SP0$  action” button, which causes the tool to pick from the enabled  $SP0$  actions. For example, the tool could attempt to perform a skip in  $SP0$ , as follows:

$$sp0skip\_special \wedge sp1skip \wedge R'$$

In this case, this will result in Possum responding with *no solution*, since it cannot match the two states using a *SKIP*. The tool will now try another of the enabled actions, and will continue to try enabled actions until either one of them succeeds in matching the two states, or none of them succeed. In this particular case, the tool can successfully match the states using the  $SP0$  action  $NS0$ .

We now explain how this approach to animation supports checking of the refinement rules.

## 6.2 Checking the Refinement Proof Obligations

As stated earlier, the user's main aim is to find errors in the refinement relationship. Such failures can be defined in terms of the proof obligations needed to verify refinement steps. We now consider each of Back's proof obligations [1] in turn and show how the failure of each proof obligation, with the given abstraction relation, manifests itself in the animation process.

1. *Initialisation*:  $\forall z, x'_{init} \bullet \exists x_{init} \bullet (x_{init}, x'_{init}, z) \in R$   
If the user can perform an initialisation that the tool cannot match then this proof obligation does not hold.
2. *Main Action*:  $A \sqsubseteq_R A'$   
If the user can choose a concrete action, and the tool has a choice of enabled abstract actions (other than *SKIP*), but none of these (including *SKIP*) can result in a matching state, then this proof obligation does not hold. This means that either the refinement cannot be shown by forward simulation, or the choice of abstraction relation is wrong.
3. *Exit Condition*:  $R \wedge gA \Rightarrow gA' \vee gH'$   
If the user has no concrete actions enabled, but the tool has abstract actions enabled (other than *SKIP*), then this proof obligation does not hold. This means that the concrete system has terminated prematurely.
4. *Auxiliary Actions*:  $SKIP \sqsubseteq_R H'$   
If the user chooses a concrete stuttering action, and the

$pc.i (SP1)$	$pc.i (SP2)$
0	0
1	1
2	2
2	3
2	4
3	5

**Figure 9. Mapping between values of the old and new program counter  $pc.i$**

tool cannot match it with a *SKIP* in the abstract specification then this proof obligation does not hold. This means that the concrete system can perform internal actions not allowed by the abstract specification.

5. *Internal Convergence*:  $R = wp([\mathbf{do} H' \mathbf{od}], true)$   
If the user enters a concrete state, and then by using only stuttering actions is able to return to the same state, then this proof obligation does not hold. This means that the concrete system can perform internal actions indefinitely, i.e., it has entered an infinite loop.

If one of the above proof obligations does not hold then we can conclude that either we cannot prove the refinement using forward simulation, or the choice of abstraction relation is wrong.

We now illustrate this approach using the second stage of Back's mutual exclusion refinement case study [1], and we show how the method was used to expose an error in Back's abstraction relation.

## 7 Finding an Error in a Data Refinement

In the second stage of the refinement, Back splits action  $CS.i$  into three actions  $CAS.i$ ,  $CBS.i$  and  $CCS.i$ , thus allowing the processes' critical sections to be executed non-atomically. The result of this refinement step is shown in Figure 8. As well as splitting action  $CS.i$ , Back also changes the operation of the program counter  $pc.i$  [1].

The abstraction relation is given as a mapping between values of the old and new program counter  $pc.i$ , as shown in Figure 9, plus an invariant on the new variables as follows:

$$\begin{aligned} pc.i = 1 \vee pc.i = 2 &\Rightarrow b.i \\ \wedge pc.i = 3 &\Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc.j = 1) \\ \wedge pc.i = 4 \vee pc.i = 5 &\Rightarrow b.i \wedge (\neg b.j \vee t = j \vee \\ &pc.j = 1) \wedge y.i = w + i + 1 \end{aligned}$$

where  $j = 1 - i$ .

Back verifies the refinement step by first proving that the above invariant is preserved by each action in  $SP2$ , and then satisfying each of the proof obligations.

```

var ( $pc.i, y.i : \mathbb{N}; b.i, cr.i : \mathbb{B}$  for  $i = 0, 1$ );  $t : 0 \dots 1$ 
init  $b.i := false; pc.i := 0; cr.i := false$  for  $i = 0, 1$ 
do
  ( $\square cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1$  for  $i = 0, 1$ )           [BS.i]
  ( $\square pc.i = 1 \rightarrow t := i; pc.i := 2$  for  $i = 0, 1$ )           [TS.i]
  ( $\square pc.i = 2 \wedge (\neg b.(1 - i) \vee t = 1 - i) \rightarrow pc.i := 3$  for  $i = 0, 1$ ) [CAS'.i]
  ( $\square pc.i = 3 \rightarrow y.i := w + i + 1; pc.i := 4$  for  $i = 0, 1$ ) [CBS'.i]
  ( $\square pc.i = 4 \rightarrow w := y.i; cr.i := false; pc.i := 5$  for  $i = 0, 1$ ) [CCS'.i]
  ( $\square pc.i = 5 \rightarrow pc.i := 0; b.i := false$  for  $i = 0, 1$ ) [BR.i]
  ( $\square \neg cr.i \rightarrow cr.i := false \square cr.i := true$  for  $i = 0, 1$ ) [NS.i]
od :  $w : \mathbb{N}$ 

```

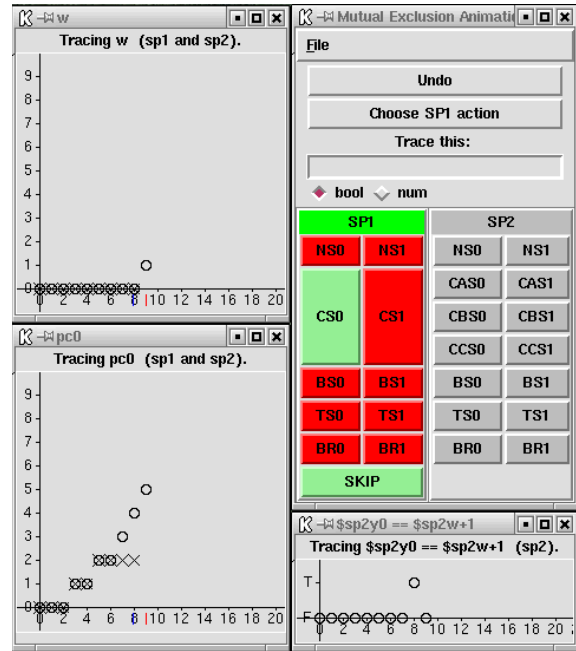
**Figure 8. Action system  $SP2$ , the outcome of the second step of the refinement**

We checked this refinement step using the method described in Section 6.1.2. This quickly revealed an error in Back's abstraction relation via the following sequence of actions:

User's choice ( $SP2$ )	Tool's choice ( $SP1$ )
<i>init</i>	<i>init</i>
NS.0	NS.0
NS.1	NS.1
BS.0	BS.0
BS.1	BS.1
TS.0	TS.0
CAS.0	SKIP
CBS.0	SKIP
CCS.0	Can't match!

The sequence ends when the tool is unable to match  $SP2$ 's state with any of the enabled  $SP1$  actions. The traces generated by the above animation are shown in Figure 10. Three traces are shown:

- Shared variable  $w$ . This shows the values of the externally visible variable  $w$  in both  $SP1$  (denoted with a cross) and  $SP2$  (denoted with a circle). (Note that this is not equivalent to seeing a *trace*, in the sense of the action system semantics defined in Section 4.1. Here we can see iterations of the action systems in which  $w$  does not change — steps which are not considered to be externally visible.)
- Program counter  $pc.0$ . This shows the values of the program counter  $pc.0$  in both  $SP1$  (denoted with a cross) and  $SP2$  (denoted with a circle). This clearly shows the relationship between the program counter variables that Back defined in tabular form (see Figure 9).
- Predicate  $y.0 = w + 1$ . This shows the truth of a part of Back's invariant in  $SP2$ . It should be *true* when-



**Figure 10. Problem - tool cannot match  $SP2$ 's state**

ever  $pc.0 = 4 \vee pc.0 = 5$ , if the invariant is to be preserved. (Its value under other circumstances is irrelevant.)

As shown by the traces in Figure 10, program counter  $pc.0$  equals 4 at time 8 and 5 at time 9. We therefore expect predicate  $y.0 = w + 1$  to be *true* at both these times. However, the trace shows that this predicate is *false* at time 9, thus revealing the error in the invariant.

When we check the concrete action system specification in Figure 8 we see that action  $CCS'.i$  makes the assignments  $w := y.i$  and  $pc.i := 5$  which clearly violate the invariant reproduced above.

Fortunately, this problem can be easily fixed by changing the abstraction invariant to:

$$\begin{aligned} pc.i = 1 \vee pc.i = 2 &\Rightarrow b.i \\ \wedge pc.i = 3 &\Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc.j = 1) \\ \wedge pc.i = 4 &\Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc.j = 1) \wedge \\ & y.i = w + i + 1 \\ \wedge pc.i = 5 &\Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc.j = 1) \wedge y.i = w \end{aligned}$$

where  $j = 1 - i$ . With this correction, we are unable to find any further errors in the refinement.

## 8 Discussion

The error that we found in the refinement does not invalidate the refinement step, but it does show that Back's proof of the step was incorrect. This type of error in the abstraction relation appears to be common in refinement proofs [20]. Using proof alone it can be a very time consuming process to find the right abstraction relation. Our method very quickly revealed the mistake.

Furthermore, our experience indicates that the user of such a tool can quickly get an intuitive understanding of how the refinement step works. The user achieves this by watching the guards being enabled and disabled dynamically, as well as selecting and examining the traces.

The approach is comparable to program testing, and exhibits the same properties. It is effective for finding errors in a refinement but, except in examples with very small state spaces, it is not effective for proving a refinement to be correct.

Our approach does not currently consider backward simulations [7]. If the refinement relation being checked is provable only by backward simulation then our checks may fail, i.e. the user may appear to find an error, even though the refinement relation is valid. This can be solved by allowing the tool to backtrack in order to find a match, but this introduces complexity into an otherwise intuitive approach. We intend to consider backward simulations in future work, but since most practical refinements are forward simulations, our current approach is useful as it stands.

In the examples in this paper, the statements are mostly straightforward executable assignments. However, Possum can animate specifications that are not refinable to code. In future work, it would be interesting to explore cases in which the specifications are not obviously executable.

The animation method we have described is much like a game that the user plays against the tool. The aim of the game from the user's perspective is to find an error in the refinement. The animation tool supports the user in winning the game, by providing an intuitive interface to the two action systems, and by allowing the user to select suitable traces to visualise. The user can take advantage of these visualisations to help him guide the game towards a winning position, in which one of the refinement rules can be shown not to hold (see Section 6.2).

This is a very different kind of game to the game approach developed by Back and Von Wright [4], where they use game theory to provide an operational interpretation of contracts statements (a more general form of specification). It is, in fact, closer to the use of game theory in bisimulations [19], although the mechanisms and aims of the game are quite different. The game analogy seems to be quite general, and can be adapted for all the other styles of data refinement we have examined. It also lends itself to a more general specification/program testing method, which focusses and supports the user in achieving the right goal — that of finding errors. This is a direction for future work.

## 9 Conclusion

We have shown how animation can be used to provide programmers with insights into the correctness, or otherwise, of refinement steps. While such techniques cannot offer the same guarantees as theorem prover-based refinement tools, we have seen that they can quickly uncover major problems in the same way that testing can often detect gross errors more efficiently than formal verification.

Furthermore, we considered the most general and powerful form of refinement, in which the whole design may change in a single step. In doing so, we noted a pleasing correspondence between the five proof obligations needed to verify such a refinement step and distinct errors detectable via the animator.

## Acknowledgements

We would like to thank Graeme Smith for his guidance and for his comments on drafts of this paper. This work was funded in part by Australian Research Council Large Grant A00104650, Verified Compilation Strategies for Critical Computer Programs.

## References

- [1] R.-J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-93, Computer Science Department, California Institute of Technology, 1992.
- [2] R.-J. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8:324–346, 1996.
- [3] R.-J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Proceedings of CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer-Verlag, 1994.
- [4] R.-J. R. Back and J. von Wright. *Refinement calculus: a systematic introduction*. Springer-Verlag, 1998.
- [5] A. Burns and A. J. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1990.
- [6] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 40–61. Springer-Verlag, 1997.
- [7] W.-P. de Roever and K. Engelhardt. *Data refinement: model-oriented proof methods and their comparison*. Cambridge University Press, Cambridge, UK, 1998.
- [8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [9] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99 Proceedings of the 1st international conference on Integrated Formal Methods*, pages 315–334. Springer-Verlag, 1999.
- [10] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5:367–382, 1993.
- [11] N. Hamilton, D. Hazel, P. Kearney, O. Traynor, and L. Wildman. A complete formal development using Cogito. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Science Conference*, pages 319–330. Springer-Verlag, 1998.
- [12] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the Sum specification language. In W. Wong and K. Leung, editors, *Proceedings Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 42–51. IEEE Computer Society Press, December 1997.
- [13] C. Heitmeyer. On the need for practical formal methods. In A. P. Ravn and H. Rischel, editors, *Formal Techniques for Real Time and Fault Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 18–26. Springer-Verlag, 1998.
- [14] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.
- [15] D. Jackson, I. Schechter, and I. Schlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the 2000 International Conference on Software Engineering, Limerick, Ireland*, pages 730–733. ACM, New York, USA, June 2000.
- [16] T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. In D. D. Grant and L. Sterling, editors, *Proceedings of the Australian Software Engineering Conference, ASWEC 2001, Canberra, Australia*, pages 76–85. IEEE Computer Society Press, 2001.
- [17] T. Miller and P. Strooper. Combining the animation and testing of abstract data types. In *Proceedings of the Second Asia-Pacific Conference on Quality Software, APAQS 2001, Hong Kong*, pages 249–258. IEEE Computer Society Press, December 2001.
- [18] C. Morgan. *Programming from specifications*. International series in computer science. Prentice Hall, 1990.
- [19] M. Nielsen and C. Clausen. Bisimulation for models in concurrency. In B. Jonsson and J. Parrow, editors, *Proceedings of CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 385–400. Springer-Verlag, 1994.
- [20] N. J. Robinson. Checking Z data refinements using an animation tool. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B, Proceedings of the 2nd International Conference of B and Z Users*, volume 2272 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2002.
- [21] N. J. Robinson and C. Fidge. Visualisation of refinements. In D. D. Grant and L. Sterling, editors, *Proceedings of the Australian Software Engineering Conference, ASWEC 2001, Canberra, Australia*, pages 244–251. IEEE Computer Society Press, 2001.
- [22] H. Waeselynck and S. Behnia. B model animation for external verification. In J. Staples, M. Hinchey, and S. Liu, editors, *Second International Conference on Formal Engineering Methods, Brisbane, Australia, December 9-11, 1998*, pages 36–45. IEEE Computer Society Press, 1998.
- [23] L. Wildman, C. Fidge, and D. Carrington. Computer-aided development of a real-time program. *Software - Concepts and Tools*, 19(4):190-202, August 2000.