Using State Machines for the Visualisation of Specifications via Refinement

Colin Pilbrow
Department of Computer Science
University of Waikato
Hamilton, New Zealand
colinpilbrow@gmail.com

Steve Reeves
Department of Computer Science
University of Waikato
Hamilton, New Zealand
stever@waikato.ac.nz

ABSTRACT

We talk in this paper about using state machines and refinement to characterise the visualisation of a computation.

We use Z specifications to give examples of systems in the usual way, and then use Z schemas to also represent states and transitions in state machines, which we consider to be a particular kind of visualisation of a specified system.

We have investigated the principle of substitutivity and the idea of downward simulation to check whether or not a refinement relation exists between the specification and the state machine. We are looking at this because we believe that the soundness of the visualisation can be captured by such a refinement relationship.

1. INTRODUCTION

Refinement first appeared as an informal process in [10] where via "stepwise refinement" we move in a series of steps from a high-level specification to a lower-lever implementation. These steps typically involve replacing high-level descriptions of properties with lower-level descriptions that preserve the intended properties given by the specification but which are closer and closer to being expressed completely in the target programming language. This idea was formalised into what we call refinement by defining both the specification language and also what counts as a valid step, and this in turn allows us to prove that the steps preserve properties. Also, via transitivity, the final lower-level version (usually an implementation) preserves the intended properties of the original specification.

Refinement is used to prove correctness when developing systems from specifications, and we will argue here that it can also be used to prove the acceptability (or otherwise) of visualisations of specifications. We will talk in this paper about using state machines as visualisations and the idea of refinement to characterise the *sound* visualisation of a computation. We write both machines and specifications in Z. We check to see whether the Z that represents the state machines is a refinement of the Z that expresses the initial specification. If it is, we argue, then the resulting state machine is both an acceptable visualisation of the original system and also one that it in some sense does not mislead us relative to the initial specification.

In the rest of the paper, we introduce Z very briefly, and then introduce some of the ideas and formalisation concerning refinement. We go on to give examples of state machines that may or may not be sound visualisations, and give examples of checking, via trying to prove that a refinement relation of the right sort exists, whether or not these are in fact acceptable or not.

We are, in this paper, going to ignore concerns around aesthetics, graphic design, usability and so on. There are clearly important when considering visualisation of a computational systems, but they are not what this paper is about.

ASWEC ' 15 Vol. II, September 28-October 01, 2015, Adelaide, SA, Australia

ISBN 978-1-4503-3796-0/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2811681.2811702

2. Z SPECIFICATIONS

2.1 Schemas, bindings, states and operations

The state space of a system is described by Z state schemas. These give the names and types of the parts of the state that can be observed, which we call *observations*, together with predicates which express properties of and constraints on and between these observations. Each schema describes a set of allowable states, and formally each state is represented by a binding, which is simply a record-like structure which, for each of the observations in the state schema, tells us what that observation's value is in that state. Bindings are only part of a state space if the relationship between its observation values actually satisfy the predicates in the state schema.

Further, we have operation schemas which describe, in terms of changes to observations, how the state of the system changes from one state to another. Again, the operation schemas denote sets of bindings. This time instead of each binding representing an allowable state of the system, the bindings in an operation schema show, for each observation, what its value is before and after the operation takes place. So, each binding in this case is rather like an element in a relation between before and after states.

We illustrate these ideas in the next section with two examples. More details are readily available (*e.g.* [3, 11]) since Z has been extensively used for a number of decades now to specify computational systems.

2.2 Birthday Book

This example is the old standard, but one that we will adapt in our examples later in this paper: Spivey's Birthday Book example [8, 9]. The Birthday Book is a system for recording people's birthdays. There are operations for adding or removing people, or finding the names of people with a given birthday.

The sets $NAME$ of people's names, and $DATE$, of dates, are taken as given; their structure is of no concern for this level of detail of specification:

$$[NAME, DATE]$$

We make clear what can be seen of a system via the *observations* possible in the state. The Birthday Book uses an observation of a partial function *birthday* to record the birthdays of known people, and the observation of the set *known* that contains the names of the people whose birthdays are recorded:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook}} \\ \hline
known : \mathbb{P}\, NAME \\
birthday : NAME \nrightarrow DATE \\ \hline
known = \operatorname{dom} birthday \\ \hline
\end{array}$$

So this *schema* defines the state space of the system. Initially, there are no recorded birthdays:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitBirthdayBook}} \\ \hline
BirthdayBook \\ \hline
birthday = \emptyset \\
known = \emptyset \\ \hline
\end{array}$$

The *AddBirthday* operation registers a new birthday, given a name and a date:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddBirthday}} \\ \hline
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\ \hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\ \hline
\end{array}$$

Note that the prime on an observation means, by convention, that this is the observation *after* the operation has taken place.

In a complete specification there would also be schemas for further operations like removing someone's birthday, or looking up a birthday given a name, and so on. Finally, there is also a *schema calculus* which has operations like $\wedge$ and $\vee$ for putting schemas together to make larger schemas. These essentially put the predicate parts of schemas into conjunction or disjunction respectively.

3. REFINEMENT

3.1 Principle of Substitutivity

Essentially what the Principle of Substitutivity (PoS) says is: if we substitute your ideal, specified system with our actual system and you cannot tell the difference, or our system at least does everything your ideal would, then our system must be acceptable to you.

Morgan's book [5] (especially its introductory chapters) has a very good discussion of all this with compelling examples.

This principle, for us, is the basic conceptual foundation upon which we base our search for ways to formally capture what it means to be a sound visualisation of a computational system. So, we will not consider personal or aesthetic aspects of visualisation; we will concentrate on trying to formalise what a *sound* visualisation is, *i.e.* one that does not *mislead* the user, with the understanding that whatever else a visualisation is, it must be sound. If your visualisation is not sound, then however fancy and impressive is looks, it is worthless.

For this much more modest goal, the PoS will serve us well. It also leads on, in its formalisation, to the idea of (formal) *refinement*, which we will take as our basic foundation for soundness of visualisation. We say more about this near the end of the paper.

We now formalise the ideas behind the PoS. If we are given the specification of some operation which changes the state of a system and we are asked to implement that operation then the implementation counts as correct under the following conditions: (a) it can be used in all the situations that the specification of the operation says it can be used (and perhaps more situations); and (b) the results of its use are amongst the results specified for the operation (that is, it does not do anything to the state outside of the possibilities of what can be done to the state as allowed by the specification).

Later we also use *data refinement.* This is based on the PoS, but also features a relation between the specification state space and the visualisation state space.

4. STATE MACHINES USING Z

Here we provide methods for writing state machines using schemas, as a prelude for formally checking for refinement.

4.1 State Machines using Schemas

Each state and each collection of transitions with the same name is written using schemas.

For example, the state space of a state machine with two states can be defined by

$$StateSM ::= State0 \veebar State1$$

where $State0$ and $State1$ are both schemas representing states in the state machine. The schema calculus being used here is simply stating that the system is either in $State0$ or $State1$, but not both (so we have the idea of an exclusive-or being expressed in the schema calculus).

A transition representing operation $Op1$ going from $State0$ to $State1$, for example, can be defined as simply by

$$Op1 :== State0 \wedge State1'$$

If there are several $Op1$ transitions in the state machine, they can be collected using more schema calculus, for example

$$Op1 :== (State0 \wedge State1') \vee (State1 \wedge State0')$$

would be a suitable definition if there were transitions from $State0$ to $State1$ and from $State1$ to $State0$.

5. SOUNDNESS OF A VISUALISATION

We have repeatedly claimed that we are using refinement to check the *soundness* of visualisations, and here we address that claim.

The visualisations that we prove are refinements of specifications have the following properties:

- every trace and transition in the visualisation is also in the specification;
- if an operation is enabled in the specification, it must also be in the visualisation;
- the visualisation can use observations of different types than those in the specification (via a retrieve relation) to help visualise different parts of the specification.

Visualisations are often considered to be an abstraction of the system being visualised, so why are we not refining the visualisation into the specification? If we have a visualisation that refines to the specification, we will know that every trace in the specification is in the visualisation, *i.e.* is being visualised. However, the visualisation may also include traces and transitions that are not in the specification. We consider such visualisations to be misleading. Additionally, it is possible to have operations enabled in a state that are not visualised as transitions from the respective state in the state machine.

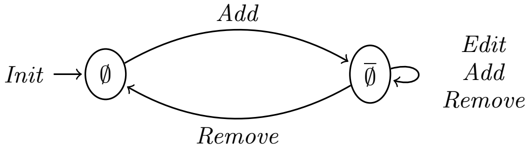
Hence, we are checking soundness of our state machine visualisations by checking that a refinement relation exists from specification to visualisation.

6. STATE MACHINE REFINEMENT

Using the above formalisation of state machines, we will provide two example visualisations to investigate refinement between the visualisations and the birthday book specification. We have chosen these examples because they relate to the "archetypal" birthday book example, and also because they seem to cover some obvious choices that might be made when visualising this example.

6.1 Empty/NonEmpty State Machine

The first visualisation considers only the states where the book is empty or nonempty to be important to the user.



The state machine is initialised in the state where the birthday book is empty. The only enabled operation in this state is the *Add* operation, which changes the state to the state where the birthday book is not empty. We can also *Add* when the birthday book is not empty, and the book stays nonempty. *Edit* is enabled only when the book is nonempty, and does not change the state. *Remove* is also only enabled when the book is nonempty, and will either result in an empty book or a nonempty book.

So, we have removed the information that lets us know if we are removing the last friend from the book, which has created some nondeterminism in the state machine. We can also add a friend that may have already been added, or edit a friend that may not have been added.

The following Z schemas describe the state space and transitions of the state machine.

The *State*1 and *State*2 schemas remain abstract throughout, but can still be manipulated using schema calculus to give the desired schemas.

$$\begin{array}{l} StateSM ::= State1 \veebar State2 \\ InitSM ::= State1' \\ Add ::= State2' \\ Remove ::= State2 \\ Edit ::= State2 \wedge State2' \end{array}$$

Note that *Add* and *Remove* have been simplified. *Add* will always end up in *State*2 from either state, and *Remove* will always start in *State*2 and end in either state.

The relation between the specification state and the state machine state shows how the specification and the state machine are related: when the specification is in a state where $known = \emptyset$ the state machine is in *State*1, otherwise the state machine is in *State*2. (This relation is important when we consider refinement, when it is usually called the *retrieve relation*. It is the same relation that we talked about in section III(b) when discussing data refinement and its simulation relation.) It can be written:

$$\begin{array}{|l} R \\ \hline State \\ StateSM \\ \hline (known = \emptyset \wedge State1) \\ \vee \\ (known \neq \emptyset \wedge State2) \\ \hline \end{array}$$

We can check whether a refinement relation holds by seeing whether a downwards simulation exists (and by considering the principle of substitutivity). We can prove that, in fact, the visualisation is not a downwards simulation of the specification, so a refinement relation does not exist and so, we say, the visualisation is not sound. For example, the trace

$$\begin{array}{l} InitSM, Add(Alan\ Turing, 010112), \\ \qquad\qquad\qquad Remove(Alan\ Turing) \end{array}$$

can end in either the empty or nonempty state, however the specification will always end with an empty book. This example shows that this method can help reveal that a visualisation is unsound.

6.2 Alan Turing Birthday Book

The second example involves a slightly more complicated visualisation, focusing on the status of a particular name in the book.

In figure 1 we have used the following abbreviations:

StateT: states in which birthday books contain

$$Alan\ Turing \mapsto 230612$$

i.e. this maplet is part of the function *birthday* in such states.

StateF: states in which birthday books contain the name *Alan Turing*, but with some date other than 230612.

*State*0 : states in which birthday books do not contain the name *Alan Turing*.

The operations are separated out into the cases where the name and date are equal to or different from *Alan Turing* and 230612, respectively.

In the state machine the operation names have been given subscripts to represent the inputs in order to save space. For example, $Add_{A,T}$ is the add operation with inputs *Alan Turing* and 230612, $Edit_O$ is the edit operation where the name is not *Alan Turing* and the date is any date. $Remove_{A,F}$ is the

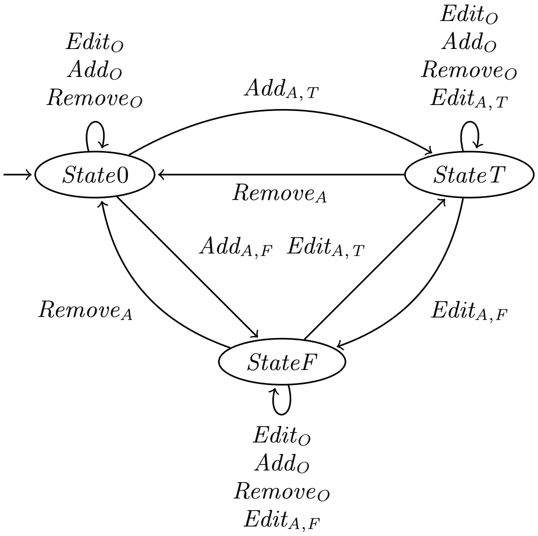


Figure 1: Visualising the Alan Turing birthday book

remove operation where the name is *Alan Turing* and the date is not 230612.

So, the transitions have been separated into transitions that have *Alan Turing* as input, and those that do not. The transitions that do not have *Alan Turing* as input do not change the state, but are still present in the state machine to prevent these operations from behaving chaotically. The transitions that have *Alan Turing* as input have the same preconditions as the original specification.

We need the transitions in schema form to check for a refinement. These concrete schemas are the combination of each of the appropriately named transitions.

$$\begin{array}{|l} A : \mathit{NAME} \\ T : \mathit{DATE} \end{array}$$

$$\mathit{StateSM} ::= \mathit{State}0 \veebar \mathit{StateT} \veebar \mathit{StateF}$$

$$\mathit{InitSM} ::= \mathit{State}0'$$

$$\begin{array}{l} \mathit{Add}_0, \mathit{Edit}_0, \mathit{Remove}_0 ::= \\ ((\mathit{State}0 \land \mathit{State}0') \lor (\mathit{StateT} \land \mathit{StateT}') \lor \\ (\mathit{StateF} \land \mathit{StateF}')) \land \mathit{name}? \neq \mathrm{A} \end{array}$$

$$\begin{array}{l} \mathit{Add} ::= \mathit{Add}_0 \lor \\ (\mathit{State}0 \land \mathit{StateF}' \land \mathit{name}? = \mathrm{A} \land \mathit{date}? \neq \mathrm{T}) \lor \\ (\mathit{State}0 \land \mathit{StateT}' \land \mathit{name}? = \mathrm{A} \land \mathit{date}? = \mathrm{T}) \end{array}$$

$$\begin{array}{l} \mathit{Remove} ::= \mathit{Remove}_0 \lor \\ (\mathit{StateF} \land \mathit{State}0' \land \mathit{name}? = \mathrm{A}) \lor \\ (\mathit{StateT} \land \mathit{State}0' \land \mathit{name}? = \mathrm{A}) \end{array}$$

$$\begin{array}{l} \mathit{Edit} ::= \mathit{Edit}_0 \lor \\ ((\mathit{StateT} \lor \mathit{StateF}) \land \mathit{StateF}' \land \mathit{name}? = \mathrm{A} \\ \qquad\qquad\qquad\qquad \land\ \mathit{date}? \neq \mathrm{T}) \lor \\ ((\mathit{StateT} \lor \mathit{StateF}) \land \mathit{StateT}' \land \mathit{name}? = \mathrm{A} \\ \qquad\qquad\qquad\qquad \land\ \mathit{date}? = \mathrm{T}) \end{array}$$

The simulation relation between the states of the specification and the state machine is given by:

$$\begin{array}{|l} \multicolumn{1}{l}{R} \\ \hline \mathit{State} \\ \mathit{StateSM} \\ \hline (A \notin \mathit{known} \land \mathit{State}0) \\ \lor\ (A \in \mathit{known} \land \mathit{birthday}\ A \neq T \land \mathit{StateF}) \\ \lor\ (A \in \mathit{known} \land \mathit{birthday}\ A = T \land \mathit{StateT}) \\ \hline \end{array}$$

We can prove that this visualisation is a downwards simulation of the specification. So, we can say that this visualisation is sound. Every sequence of operations performed in the specification will end with the same status for *Alan Turing* when the same sequence if performed in the visualisation.

7. MORE "VISUAL" VISUALISATIONS—AN ALTERNATIVE TO STATE MACHINES

An example of this, which also allows us to illustrate more pictorial examples of visualisation beyond state machines, is given by typical uses of ProB [6]. Here we present the simple jars visualisation.

We have two jars, $j3$ has a volume of 3 litres, whereas $j5$ can contain 5 litres.

$$\mathit{Jars} ::= j3 \mid j5$$

$$\begin{array}{|l} \mathit{max\_fill} : \mathit{Jars} \rightarrow \mathbb{N} \\ \hline \mathit{max\_fill} = \{j3 \mapsto 3, j5 \mapsto 5\} \end{array}$$

Each jar currently contains a certain amount of liquid.

$$\begin{array}{|l} \multicolumn{1}{l}{\mathit{Level}} \\ \hline \mathit{level} : \mathit{Jars} \rightarrow \mathbb{N} \\ \hline \forall j : \mathit{Jars} \bullet \mathit{level}(j) \leq \mathit{max\_fill}(j) \\ \hline \end{array}$$

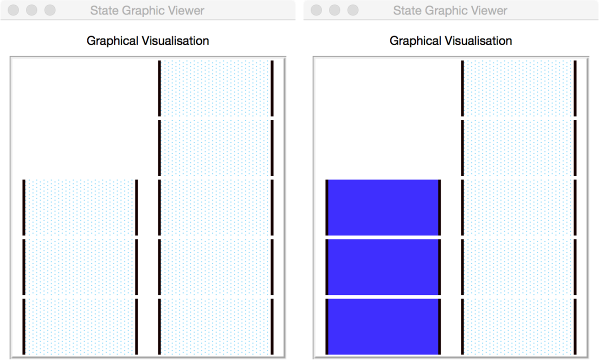
In the beginning, all jars are empty.

$$\mathit{Init} \mathrel{\widehat=} [\, \mathit{Level} \mid \operatorname{ran} \mathit{level} = \{0\} \,]$$

A (not full) jar can be filled completely.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Fill\_Jar}} \\
\Delta \mathit{Level} \\
j? : \mathit{Jars} \\
\hline
\mathit{level}(j?) < \mathit{max\_fill}(j?) \\
\mathit{level}' = \mathit{level} \oplus \{j?\} \lhd \mathit{max\_fill} \\
\hline
\end{array}$$

There are several further operations. The important thing here, though, is that fact that this example is the standard one used in an already existing notion of visualisation in a formal framework, namely the idea of animation (as they call it) in ProB. In ProB there is the facility for representing states by pictures that change when operations are performed, thus animating the specification. So, if we started in a state where a jar was not full and then applied the *Fill_Jar* operation the ProB animation would show this as a change in pictures from a non-full jar to a full jar:



This example and method is in contrast to previous work in this paper, where we have used state machines as our visualisation choice. However, our refinement ideas can easily be used on this sort of visualisation as well, to check for soundness in the same sense as we checked the state machine visualisations.

We should also note here that the visualisations via state machines that we talk about in the rest of the paper are not animated in the sense that ProB's are. That is, as operations are used and the system moves from state to state then ProB animates this process by showing visually that the state has changed (like the before and after pictures above). We, by contrast, show the whole state space and the possible changes between its states all together. However, we could animate our visualisations by, for example, making the state that the system is currently in a different colour from the rest. In the future we will investigate to see whether or not an aesthetic difference such as this can affect the idea of soundness propounded in this paper.

8. CONCLUSIONS AND FUTURE WORK

We have started, in this paper, to explore using the well-established formal notion of refinement as a way of telling whether or not a state machine visualisation of a specification is sound, by which we mean that a presentation of the system via the visualisation would not be misleading—anything that the visualisation showed us the system doing would be something the original specification allowed.This would be important if we were using the visualisation to present a system to a non-programmer for example, where all the messy details of implementation would be unwelcome, but where "what the system does" has to be demonstrated.

One area where a more general notion of visualisation of a computation has been long and widely explored is in the teaching of programming. The work of Rogers *et al.* [1], Stasko *et al.* [4] and even our own (from decades ago) in [2,7] might be relevant here.

Also, visualisations are commonly used to validate a specification. That is, early design decisions can be checked with clients to ensure that those design decisions, which will later become cast into the software of an implementation, are still valid ones as far as the client (and their specification of what they want) is concerned.

We have, of course, barely scratched the surface here, and there is much further work and investigation to be carried out.

Acknowledgments

Thanks for Judy Bowen and Robi Malik for their ideas and encouragement concerning this work. We also thank the anonymous referee who suggested the teaching of programming world might have something to teach us—which we had evidently forgotten!

9. REFERENCES

[1] A. Akingbade, T. Finley, D. Jackson, P. Patel, and S. H. Rodger. Jawaa: Easy web-based animation from cs 0 to advanced cs courses. In *In Proceedings of the 34 th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003*, pages 162–166. ACM Press, 2003.

[2] D. Goldson. A symbolic calculator for non-strict functional programs. *The Computer Journal*, 37(3):177–187, 1994.

[3] M. C. Henson and S. Reeves. Investigating Z. *Journal of Logic and Computation*, 10(1):1–30, 2000.

[4] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[5] C. Morgan. *Programming From Specifications (2nd ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1998.

[6] ProB Project. http://stups.hhu.de/prob/.

[7] S. Reeves, D. Goldson, P. Fung, T. O'Shea, M. Hopkins, and R. Bornat. The Calculator Project-formal reasoning about programs. In M. Purvis, editor, *Proceedings of Software Education Conference (SRIG-ET'94)*, pages 166–173. IEEE Computer Society Press, 1995.

[8] J. Spivey. An Introduction to Z and Formal Specification. *IEE Software Engineering Journal*, 4(1):40–50, 1989.

[9] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.

[10] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[11] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.