Working Paper Series
ISSN 1170-487X

State- and Event-based refinement

Steve Reeves and David Streader

Working Paper: 09/2006
September 20, 2006

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

State- and Event-based refinement

Steve Reeves and David Streader
University of Waikato,Hamilton, New Zealand
{dstr,stever}@cs.waikato.ac.nz

September 20, 2006

Abstract

In this paper we give simple example abstract data types, with atomic operations, that are related by data refinement under a definition used widely in the literature, but these abstract data types are not related by singleton failure refinement. This contradicts results found in the literature. Further we show that a common way to change a model of atomic operations to one of value passing operations actually changes the underlying atomic operational semantics.

Keywords: data refinement, process refinement, singleton failures

1 Introduction

We will consider a natural notion of data refinement and apply it to abstract data types (ADT) with atomic operations. Similar formalisms appear widely in the literature. We will show that data refinement (on ADT with atomic operations) is not singleton failure refinement. But using ADTs with value passing operations (i.e. non-atomic ADTs), data refinement, as commonly defined in the literature, *is* equivalent to singleton failure refinement. Thus the common way to model value passing actually changes the underlying atomic semantics.

We define data refinement in Section 2 and singleton failure refinement in Section 3: both are definitions that appear widely in the literature. Our result Lemma 3 in Section 4 is somewhat unexpected. It states that for ADTs with atomic operations, data refinement and singleton failure refinement are not the same.

In Section 5 we give an informal description of who observes what of an ADT. To do this we find it useful to split interfaces/programs into two types: transactional and interactive. Whether an informal discussion is insightful is quite naturally a matter of personal taste, but for us it has been useful in explaining the unexpected result of the previous section.

In Section 6.1 we discuss value passing operations as dealt with in the literature and in Section 7 we conclude.

2 Data Refinement

Let an ADT D have a set of operations $\mathsf{D}_N \triangleq \{\mathsf{D.n_i} \mid \mathsf{n_i} \in N\}$ where N is a set of operation names, plus $\mathsf{D.init}$ a definition of the initial state and an operation $\mathsf{D.final}$ to terminate the ADT, a *finalisation*, that defines what can be observed of the ADT. Initialisation means mapping from a global state to the initial state of the ADT, and finalisation means mapping from the final state of the ADT back to the global state. Since we are here dealing with *abstract* data types, we need consider only one designated value in the global state to denote a successful use of the ADT.

We will use Z to define the ADTs but the results in no way depend upon this.

The Z schema $\mathsf{State_D}$ defines the state space, $[\![\mathsf{State_D}]\!]_Z$, of the ADT D . Operation schemas of Z have a well-known partial relational semantics $[\![\mathsf{D.n}]\!]_Z \subseteq [\![\mathsf{State_D}]\!]_Z \times [\![\mathsf{State_D}]\!]_Z$. We will lift and totalise these partial relations, as in [1], to give the operations the *guarded outside of precondition* interpretation.

Let $\mathit{State}_{\mathsf{D}_\perp} \triangleq [\![\mathsf{State_D}]\!]_Z \cup \{\perp_\mathsf{D}\}$ and $\mathit{domain}([\![\mathsf{D.n}]\!]_Z) \triangleq \{x \mid \exists y, (x,y) \in [\![\mathsf{D.n}]\!]_Z\}$.

$$[\![\mathsf{D.n}]\!] \triangleq [\![\mathsf{D.n}]\!]_Z \cup \{(x, \perp_\mathsf{D}) \mid x \in \mathit{State}_{\mathsf{D}_\perp} \setminus \mathit{domain}([\![\mathsf{D.n}]\!]_Z)\}$$

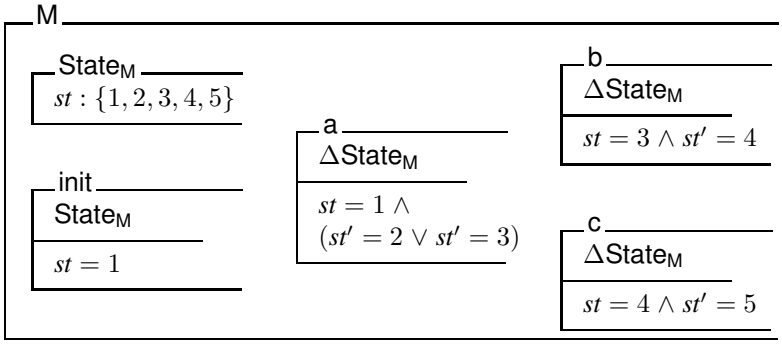

Figure 1: ADT M

So for example using the operation $\mathsf{M.a}$ in Fig. 1:

$[\![\mathsf{M.a}]\!] = \{(\{\!|\ st \rightsquigarrow 1\ |\!\}, \{\!|\ st \rightsquigarrow 2\ |\!\}), (\{\!|\ st \rightsquigarrow 1\ |\!\}, \{\!|\ st \rightsquigarrow 3\ |\!\}), (\{\!|\ st \rightsquigarrow 2\ |\!\}, \perp_\mathsf{M}), (\{\!|\ st \rightsquigarrow 3\ |\!\}, \perp_\mathsf{M}), (\{\!|\ st \rightsquigarrow 4\ |\!\}, \perp_\mathsf{M}), (\{\!|\ st \rightsquigarrow 5\ |\!\}, \perp_\mathsf{A}), (\perp_\mathsf{M}, \perp_\mathsf{M})\}$.

The initialisation of an ADT D is defined as a relation from a singleton $\bullet$ in the global state to the state given by $\mathsf{D.init}$:

$$[\![\mathsf{D.init}]\!] \triangleq \{(\bullet, x) \mid x \in [\![\mathsf{D.init}]\!]_Z\}$$

The finalisation of an ADT D is defined as a relation from states of the ADT to the lifted global state space:

$$[\![\mathsf{D.final}]\!] \triangleq \{(x, \bullet) \mid x \in [\![\mathsf{State_D}]\!]_Z\} \cup \{(\perp_\mathsf{D}, \perp)\}$$

A program is a sequence of operations $\mathsf{D.i1;D.i2;\ldots D.in}$. The relational semantics of a program is the relational composition of the initialised and finalised sequence of relations:

$$[\![\mathsf{i1;i2;\ldots in(D)}]\!] \triangleq [\![\mathsf{D.init}]\!];[\![\mathsf{D.i1}]\!];[\![\mathsf{D.i2}]\!];\ldots [\![\mathsf{D.in}]\!];[\![\mathsf{D.final}]\!].$$

From the definitions it can be seen that if a program fails to terminate the "final state" is $\bot$. Hence all that is known is that the program has failed: how many operations were successfully called is not known. For an example program, using the ADT M in Fig. 1, it is easy to see that:

$$[\![\mathsf{a;b;c(M)}]\!] = \{(\bullet, \bullet), (\bullet, \bot)\} .$$

Definition 1 *Data refinement. Let* $\mathsf{A} \triangleq (\mathsf{A.init}, \mathsf{A}_N, \mathsf{A.final})$ *and* $\mathsf{C} \triangleq (\mathsf{C.init}, \mathsf{C}_N, \mathsf{C.final})$ *be compatible ADTs, built from the same set of operation names N. Let* $P_N(\_) \in N^*$ *be a program calling operations* $\_.\mathsf{n}$ *where* $\mathsf{n} \in N$.

$$\mathsf{A} \sqsubseteq_D \mathsf{C} \triangleq \forall P_N . [\![P_N(\mathsf{C})]\!] \subseteq [\![P_N(\mathsf{A})]\!] \qquad \bullet$$

Essentially the same definition of data refinement can be found in [1, 2, 3, 4, 5].

3 Singleton failure Refinement

We define the singleton failure semantics of an ADT by first mapping the Z relational semantics to a labelled transition system (LTS) semantics and then applying the standard definitions, from the event-based process literature, to the LTS. The standard process definitions are based on the guarded outside of precondition interpretation.

Definition 2 *Labelled transition system* $\mathsf{L} \triangleq (Nodes_{\mathsf{L}}, Tran_{\mathsf{L}}, \{s_{\mathsf{L}}\})$ *where* $s_{\mathsf{L}} \in Nodes_{\mathsf{L}}$ *and* $Tran_{\mathsf{L}} \subseteq \{(n, \mathsf{a}, m) \mid n, m \in Nodes_{\mathsf{L}} \wedge \mathsf{a} \in N\}$, *N a set of operation names.* $\bullet$

The nodes of the LTS associated with ADT D are the states of D, given by bindings of the (private) observations of the ADT, i.e. the observations in schema $\mathsf{State_D}$, to their values. The start state s_{D} is the state in D.init and is marked in the figures with $\bullet\longrightarrow$. The transitions take their names from the names of the operation schemas and the (pre-state,post-state) pair of a transition is an element of the relational semantics of the operation schema.

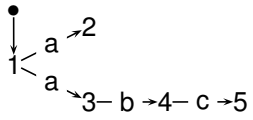

Figure 2: $lts(\mathsf{M})$

Definition 3 *Let* D *be a Z definition of an ADT.*
$lts(\mathsf{D}) \triangleq ([\![\mathsf{State_D}]\!]_Z, \{(x, \mathsf{n}, y) \mid (x, y) \in [\![\mathsf{D.n}]\!]_Z\}, [\![\mathsf{D.init}]\!]_Z)$ $\bullet$

In Fig. 1 we give a Z definition of ADT M and in Fig. 2 we show $lts(\mathsf{M})$. As our example M has only the one observation *st* we can avoid notational clutter and write $\langle\!|\ st \rightsquigarrow v\ |\!\rangle$ simply as *v*.

We write $(\mathsf{a}, \mathsf{b}, ...)$ for the sequence starting with a, followed by b and so on. Further we write $\rho\,|_n$ for the n^{th} element of sequence ρ, $\frown$ for sequence concatenation and $()$ for the empty sequence. Where L is obvious from context, we write: $n \xrightarrow{\mathsf{a}} m$ for $(n, \mathsf{a}, m) \in Tran_{\mathsf{L}}$; $m \xrightarrow{(\mathsf{a})\frown\rho} n$ for $\exists k . m \xrightarrow{\mathsf{a}} k \wedge k \xrightarrow{\rho} n$, where for any k, $k \xrightarrow{()} k$; and we write $m \xrightarrow{\rho}$ for $\exists n . m \xrightarrow{\rho} n$.

The traces of L are $Tr(\mathsf{L}) \triangleq \{\rho \mid s_{\mathsf{L}} \xrightarrow{\rho}\}$, $\pi(s) \triangleq \{\mathsf{a} \mid s \xrightarrow{\mathsf{a}}\}$ and finally we have singleton refusal sets: $Sref(\rho, \mathsf{L}) \triangleq \{\{\mathsf{a}\} \mid s_{\mathsf{L}} \xrightarrow{\rho} s \wedge \mathsf{a} \in Act - \pi(s)\}$

For example $\{\mathsf{c}\} \notin \mathit{Sref}(\mathsf{a};\mathsf{b},\mathsf{M})$ because if operation a followed by operation b have both been observed to succeed then the ADT, now in state 4, cannot refuse to perform a c operation. If only the a operation has been observed to succeed then the ADT, now in state 2 or 3, can refuse to perform a b operation, hence $\{\mathsf{b}\} \in \mathit{Sref}(\mathsf{a},\mathsf{M})$.

Singleton failure refinement: for LTS A and C,

$$\mathsf{A} \sqsubseteq_{sF} \mathsf{C} \triangleq \forall \rho.\mathit{Sref}(\rho,\mathsf{C}) \subseteq \mathit{Sref}(\rho,\mathsf{A}).$$

4 Example

In Fig. 1 we give a Z definition of ADT M and its LTS semantics $\mathit{lts}(\mathsf{M})$ is in Fig. 2. Similarly in Fig. 3 we give a Z definition of ADT N and its LTS semantics $\mathit{lts}(\mathsf{N})$. We will show that M can be data refined into N but not singleton failure refined.

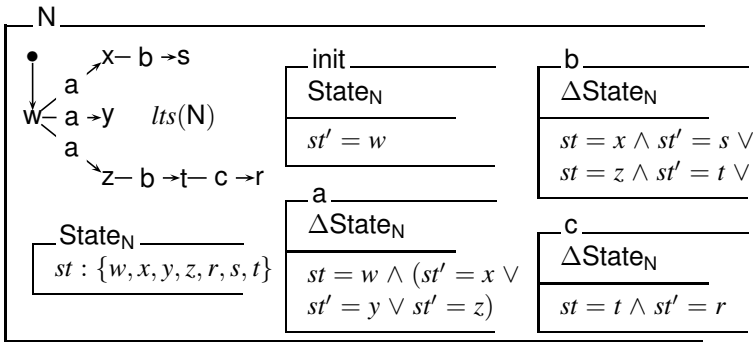

Figure 3: ADT N and LTS

It is easy to see that no program can observe the difference between M and N: the example is so small that a search of all programs is feasible.

Lemma 1 $\mathsf{M} \sqsubseteq_D \mathsf{N} \wedge \mathsf{N} \sqsubseteq_D \mathsf{M}$

Proof by construction of the relations $P(\mathsf{M})$ for all P:
$[\![\epsilon(\mathsf{M})]\!] = [\![\mathsf{a}(\mathsf{M})]\!] = \{(\bullet,\bullet)\}$,
$[\![\mathsf{a};\mathsf{b}(\mathsf{M})]\!] = [\![\mathsf{a};\mathsf{b};\mathsf{c}(\mathsf{M})]\!] = \{(\bullet,\bullet),(\bullet,\bot)\}$,
$\forall \mathsf{x} \notin \{\epsilon,\mathsf{a},\mathsf{a};\mathsf{b},\mathsf{a};\mathsf{b};\mathsf{c}\}.[\![\mathsf{x}(\mathsf{M})]\!] = \{(\bullet,\bot)\}$
By inspection the same results can be seen using N in place of M. •

Lemma 2 $\mathsf{M} \not\sqsubseteq_{sF} \mathsf{N}$

Proof. From Fig. 2 it is easy to see that $\{\mathsf{c}\} \notin \mathit{Sref}(\mathsf{a};\mathsf{b},\mathsf{M})$ and from Fig. 3 it is easy to see that $\{\mathsf{c}\} \in \mathit{Sref}(\mathsf{a};\mathsf{b},\mathsf{N})$ and hence $\mathsf{M} \not\sqsubseteq_{sF} \mathsf{N}$. •

Lemma 3 $\mathsf{M} \sqsubseteq_{sF} \mathsf{N} \not\Rightarrow \mathsf{M} \sqsubseteq_D \mathsf{N}$

Proof. From Lemma 1 and Lemma 2. •

5 General model

In this section we give a standard natural notion of refinement, based on observation, and investigate both what is doing the observing and what is being observed. We describe observation as occurring at an interface between the observer and what is being observed. We use the word *machine* to mean either an ADT or a process, and in Definition 4 formalise the natural notion of machine refinement.

We will refer to an interface as *transactional* if the interaction occurs only at no more than two distinct points: at initialisation and finalisation of the thing being observed. If termination is successful then a value can be observed at finalisation, but if termination is unsuccessful then only $\perp$ is observed. In contrast we refer to an interface as *interactive* when interaction and observation can occur at many points throughout the execution. Hence with interactive interfaces observations can be made prior to termination and even prior to nontermination.

We will use the following **natural notion of refinement** that appears in many places in the literature [1, 2, 3, 4, 5, 6]:

> The concrete machine $\mathsf{C} \in \mathbb{B}$ is a refinement of an abstract machine $\mathsf{A} \in \mathbb{B}$ where no *user* of A could observe if they were given C in place of A.

An ADT is a machine that is used by programs. We need to define what is observed by these users. To do this we first must decide if the observing user is the **program** that uses the ADT, or a **separate machine** that interacts with (observes) the program (which uses the ADT).

If the user is the program then there is only one interface: that between the ADT and the program. We describe the program/ADT interface as interactive and the user (the program) can observe when each individual operation succeeds even if the program never terminates. This is not commonly how data refinement of ADT with atomic action is defined. For many definitions of ADT (machine) refinement [2, 3, 6, 5, 4, 1] observations are only made initially and, if the program terminates, at the point of termination. Consequently we do not pursue the user-as-program view.

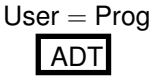

Figure 4:

If the user is not the program but a third machine that observes the program then there are two interfaces: an interactive interface between the ADT and the program and a separate interface between the program and the user. What the user can observe depends upon the second interface.

Figure 5:

We assume the program/ADT interface to be interactive and private, i.e.program/ADT interaction cannot be observed by the user. We further assume that the successful termination of the program can be achieved only if the ADT operations used never fail to terminate. Thus if an ADT operation fails to terminate then the program must also fail to terminate .

Data refinement with a transactional program/user interface allows observation only at the start and end of the program and, as previously stated, such definitions appear in many places in the literature.

With an interactive program/user interface the user can be informed of each successful operation of the ADT, even if the program subsequently fails to terminate.

Our result Lemma 3 shows that applying the different notions of observation that each of these interfaces defines results in our previously stated "natural notion of refinement" giving distinct refinements.

Definition 4 *Let Ξ be a set of contexts in which the machines A and C can be placed. Let Obs be a function defining what can be observed.*

$$\mathsf{A} \sqsubseteq_{(\Xi,Obs)} \mathsf{C} \triangleq \forall\, [\_]_x \in \Xi.Obs([\mathsf{C}]_x) \subseteq Obs([\mathsf{A}]_x) \qquad \bullet$$

Definition 5 *Parallel composition of LTS: for LTS D and E we have,*
$\mathsf{D} \,||_N\, \mathsf{E} = (Nodes_{\mathsf{D}} \times Nodes_{\mathsf{E}}, T_{\mathsf{D}||_N\mathsf{E}}, (s_{\mathsf{D}}, s_{\mathsf{E}}))$ *where $T_{\mathsf{D}||_N\mathsf{E}}$ defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_{\mathsf{D}} l, \mathsf{x} \notin N}{(n,m) \xrightarrow{\mathsf{x}}_{\mathsf{D}||_N\mathsf{E}} (l,m)} \qquad \frac{n \xrightarrow{\mathsf{x}}_{\mathsf{E}} l, \mathsf{x} \notin N}{(m,n) \xrightarrow{\mathsf{x}}_{\mathsf{D}||_N\mathsf{E}} (m,l)} \qquad \frac{n \xrightarrow{\mathsf{a}}_{\mathsf{D}} l, m \xrightarrow{\mathsf{a}}_{\mathsf{E}} k, \mathsf{a} \in N}{(n,m) \xrightarrow{\tau}_{\mathsf{D}||_N\mathsf{E}} (l,k)}$$

To make the interface between our machines and their contexts interactive and private we let $\Xi \subseteq \{(\_ \parallel_N x) \mid x \text{ is an LTS}\}$, where N is the set of operations in the machine/context interface. When the machine is an ADT we restrict the contexts to programs thus, where U is the set of events observable by the user: $\Xi = \{(\_ \parallel_N x) \mid x \in (N \cup U)^*\}$.

Because we are interested in total correctness (live) semantics we allow *Obs* to return a complete trace Tr^c. If the interface between program and user is interactive we allow the program to have any number of operations in this interface. If the interface between program and user is transactional we must restrict the operations of the program/user interface to be in $\{\bullet, \perp\}$. Further, these operations must be the last operation that any program performs.

Interactive programs are different from processes as in CSP/CCS. Processes are prepared to perform an operation from a whole set of operations whereas programs are prepared only to perform one specific operation at a time. For example, a program can perform some sequences of push and pop operations on a stack. But a process, not a program, can offer the stack the ability to perform either push or pop and allow the stack to select which.

Thus we have three separate types of contexts for machines: processes, transactional programs and interactive programs.

6 Enriching the atomic model

We consider an enrichment to our simple atomic model that appears in the literature: the use of value passing operations.

6.1 Value passing

The formal models we have defined apply only to atomic operations, but it is interesting to consider operations that may receive and return values, *value passing operations*.

Clearly the semantics of ADTs with value passing operations will be different to the semantics of ADTs with atomic operations. We will refer to an ADT with operations that all input and output the same fixed value $*$ as a *fixed value ADT*. We claim that it

is intuitive to expect that data refinement of fixed value ADTs would be the same as data refinement of ADT with atomic operations. But this is only true for some models of value passing [3, 5] but not for others [1]. The data refinement of ADTs with value passing operations in [1] when applied to fixed value ADT is the same as singleton failure semantics.

It is also common to model programs that use value passing ADT operations by *winding* the inputs into a sequence observed at the initial state and the outputs into a sequence observed at the final state. For details of how to change the semantics of operations to perform the winding see [3, 1, 7, 4]. The first reference we can find to winding is [3, Chapter 16] and on [3, p258] it is made clear that an operation has type $(State \times in)_\perp \rightarrow (State \times out)_\perp$ and there the relational semantics of a program $[\![\_]\!]_{WoD}$ is the sequential composition of these relations.

Thus using ADT M in Fig. 1 and ADT N in Fig. 3 (and interpreting them as fixed value ADT) we can see:

$$(\bullet, * * *, \perp) \in [\![\mathsf{a};\mathsf{b};\mathsf{c}(\mathsf{M})]\!]_{WoD} \wedge (\bullet, * * *, \perp) \in [\![\mathsf{a};\mathsf{b};\mathsf{c}(\mathsf{N})]\!]_{WoD}$$

In [5, 4] unlifted partial relations are used and nontermination is represented by a pre-state related to no post-state. So in both [3] and [5, 4] all that can be observed of a nonterminating program is that it has failed to terminate.

We will refer to the wound semantics of [1, 7] as $[\![\_]\!]_{BoD}$. It has type:

$$[\![\_]\!]_{BoD} : State_\perp \times input^* \times State_\perp \times output^*$$

so (as stated in [7]) the output sequence is observed even it the program fails to terminate. Hence this semantics is different to that in [3, 5]. Using ADT M in Fig. 1 and ADT N in Fig. 3 we can see:

$$(\bullet, * * *, \perp, **) \notin [\![\mathsf{a};\mathsf{b};\mathsf{c}(\mathsf{M})]\!]_{BoD} \wedge (\bullet, * * *, \perp, **) \in [\![\mathsf{a};\mathsf{b};\mathsf{c}(\mathsf{N})]\!]_{BoD}$$

By examining our example it is easy to see that ADT refinement with value passing operations as defined in [1, 7] is different to ADT refinement with value passing operations as defined in [3, 5, 4].

7 Conclusion

For ADTs with atomic operations data refinement is not singleton failure refinement.

For ADTs with value passing operations, whether data refinement of fixed value ADTs is singleton failure refinement, or the same as data refinement of ADTs with atomic operations, depends upon the details of the definition used.

We have classified programs into two types, which leads to two distinct definitions of data refinement. With transactional programs ADT refinement is what the literature commonly calls data refinement. With interactive programs ADT refinement is singleton failures refinement.

References

[1] Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Research Report PRG-RR-01-11, Oxford University Computing Laboratory (2001)

[2] de Roever, W.P., Engelhardt, K.: Data Refinement: Model oriented proof methods and their comparison. Cambridge Tracts in theoretical computer science 47 (1998)

[3] Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall (1996)

[4] Derrick, J., Boiten, E.: Relational concurrent refinement. Formal Aspects of Computing **15** (2003) 182–214

[5] Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Formal Approaches to Computing and Information Technology. Springer (2001)

[6] Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)

[7] Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Formal Aspects of Computing **18** (2006) 181–210