Computer Science at Kent

Input/Output Abstraction of State Based Systems

Eerke Boiten

Technical Report No. 12-04
June 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Input/Output Abstraction of State Based Systems

Eerke Boiten

June 2004

Abstract

Abstraction of specifications is a method of making verification and validation of specifications and implementations more tractable. This paper considers the special case where the abstraction is defined by eliding input or output variables in state based specifications – in particular, conditions for such abstractions to be sound and complete with respect to a refinement semantics. Output abstractions turn out to be unconditionally sound, and combinations of output abstractions are complete in certain circumstances. Concrete results are developed in the state-based notation Z, and then considered in the underlying semantic framework and for similar languages.

1 Introduction: Abstraction and Verification

The complexity and size of specifications and implementations form a serious impediment to verification. Model checking is able to deal with ever increasing state spaces, through advances in technology and implementation methods; however, there will always be larger or even infinite spaces of interest to be explored. Full verification, for example using refinement [7, 9], of a candidate implementation with respect to a specification, is only sparingly applied to "real" systems.

Abstraction makes these kinds of verification efforts more tractable. In model checking, where one checks whether a "model" satisfies a "property" by exhaustive search of the model's state space, one may abstract the model. This reduces the state space to be searched, and preserves positive results: if the abstract model can be shown to satisfy a property, so does the original model. In verification, one may abstract the *specification* instead, leading to a preservation of *negative* results. Such abstractions may be viewed as *tests* [1, 16], describing a subset of the originally required properties. Of course, if a candidate implementation fails the test (i.e., it produces a result inconsistent with the abstraction), then it will also fail the original specification. For appropriate definitions of refinement and abstraction (namely: set inclusion over "properties of interest"), they are each others' converse.

This paper considers a particular kind of abstraction: namely, removing *input* and *output* variables from *state based systems*. The notation used is Z [15], which has a well-developed theory of refinement [9, 17].

In Section 2 we describe how Z is used as a state-based specification notation, its standard notion of refinement, and existing approaches to "abstraction as testing" in Z. Section 3 presents syntactic and semantic methods of abstracting over input and output variables. Section 4 then describes the a generalised notion of refinement, necessary to verify these abstractions, viz., IO refinement. The subsequent sections investigate "soundness": whether abstractions are indeed converse IO refinements. For input variables, conditions for this are derived in Section 5. For output variables, an unconditional soundness result is given in Section 6. Section 7 shows that combinations of output abstractions may also be jointly "complete" in certain situations. The final section concludes, discussing how the paper's results transfer to the underlying semantic model and other specification languages.

2 Z, Refinement, Abstraction, Testing

We first describe how abstract data types (ADTs) are specified in Z, then present the standard notion of refinement for such ADTs, and finally discuss existing approaches to abstraction and testing based on this.

2.1 Abstract Data Types in Z

State-based systems are commonly described in Z using the "states-and-operations" style. An ADT is given by a state space, an initialisation, and a collection of operations. All of these are described by Z "schemas", which describe sets of "bindings", essentially labelled products. The labels may be viewed as names of variables, which are indicative of their roles: primed variables represent "after-states", inputs end in question marks, outputs in exclamation marks.

Example 1 The financial affairs of a traditional monarchy may be represented by the data type $(\mathit{Treasury}, \mathit{Init}, \{\mathit{Tax}, \mathit{Spend}\})$ where

$$\begin{array}{|l}\hline \textit{Treasury} \\ m : \mathbb{N} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Init} \\ \mathit{Treasury}' \\ \hline m' > 0 \\ \hline \end{array}$$

Initially, the treasury is non-empty. Taxing the citizens by a value $in?$ results in a corresponding increase of the treasury; the new balance is reported to the treasurer (output $m!$); the king only observes that the balance has increased (output $inc!$). If there are sufficient funds, the king may request to spend an amount $req?$, which leads to the (identical) amount $out!$ being spent.

$$\begin{array}{|l}\hline \textit{Tax} \\ \Delta \mathit{Treasury} \\ in? : \mathbb{Z} \\ m! : \mathbb{N} \\ inc! : \mathbb{B} \\ \hline in? > 0 \\ m' = m + in? \\ m! = m' \\ inc! = (m' > m) \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Spend} \\ \Delta \mathit{Treasury} \\ req? : \mathbb{N} \\ out! : \mathbb{N} \\ \hline out! = req? \\ m' = m - out! \\ \hline \end{array}$$

$\square$

The *signature* ΣS of a schema S is defined as a schema containing all its declarations (normalised[1]) with the predicate "true". Formally, $\Sigma S = S \vee \neg S$. Any schema S such that $S = \Sigma S$ will be called a signature. The subsignature relation on signatures is defined by $S \sqsubseteq T == (S \wedge T) \equiv T$.

Important subsignatures for an operation Op are $?Op$ which returns the signature of the inputs, and $!Op$ which gives the signature of the outputs. The precondition pre Op (in general not a signature) returns only the before-state and inputs, existentially quantifying over after-state and outputs.

2.2 Refinement

Refinement of Z ADTs is normally defined on two levels. Operation refinement, or "algorithmic refinement", which leaves the state unchanged, is defined at the level of individual operations.

[1] Normalisation replaces any declaration $x : S$ by $x : X$ where X is the maximal set containing S, and an extra predicate $x \in S$.

Definition 1 (Operation refinement) An operation COp is an operation refinement of an operation AOp over the same state space $State$ iff

Correctness

$$\forall\, State;\ State';\ ?AOp;\ !AOp \bullet \operatorname{pre} AOp \land COp \Rightarrow AOp$$

Applicability

$$\forall\, State;\ ?AOp \bullet \operatorname{pre} AOp \Rightarrow \operatorname{pre} COp$$

□

The two conditions implicitly represent two ways in which an operation can be refined: by reduction of non-determinism, and by widening the area where the operation is guaranteed to be well-behaved, respectively. As the operations in our example are already deterministic, they can only be refined by weakening their preconditions, e.g., *Tax* may be refined by removing the restriction $in? > 0$.

Data refinement is a generalisation of operation refinement which exploits the data type being *abstract*, i.e., the state may be changed provided the externally visible behaviour is preserved, and in general ADTs need to be refined in their entirety. Based on the relational refinement of He, Hoare and Sanders [11], this refinement theory for Z is described in full detail in the monograph [9]. The standard method of verifying data refinement is through *upward* and *downward simulations*, which are sound and jointly complete. As the former do not contribute to this paper, we only give:

Definition 2 (Downward simulation) Given ADTs $A = (AState, AInit, \{AOp_i\}_{i \in I})$ and $C = (CState, CInit, \{COp_i\}_{i \in I})$, where corresponding operations have identical input and output signatures. The relation R on $AState \land CState$ is a *downward simulation* from A to C if

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R'$$

and for all $i \in I$:

$$\begin{array}{l} \forall\, AState;\ CState;\ ?AOp_i \bullet \operatorname{pre} AOp_i \land R \Rightarrow \operatorname{pre} COp_i \\ \forall\, AState;\ CState;\ CState';\ ?AOp_i;\ !AOp_i \bullet \\ \qquad \operatorname{pre} AOp_i \land R \land COp_i \Rightarrow \exists\, AState' \bullet R' \land AOp_i \end{array}$$

□

2.3 Refinement and Testing

Methods for test case generation based on refinement techniques often employ "horizontal" decompositions of the state space, inputs and outputs, i.e., considering partitions. For horizontal decomposition of the *Spend* operation one might look at the variable $m : \mathbb{N}$ which may or may not be zero, i.e., use the disjunction

$$m = 0 \vee m > 0 \tag{1}$$

as the basis of a decomposition. In the PROST-Objects testing method described by Stepney [16], a test case might be derived by using one of these disjuncts to weaken the operation, e.g.,

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{SpendZero}} \\
\Delta \mathit{Treasury} \\
\mathit{req}? : \mathbb{N} \\
\mathit{out}! : \mathbb{N} \\
\hline
m = 0 \Rightarrow (\mathit{out}! = \mathit{req}? \wedge m' = m - \mathit{out}!) \\
\hline
\end{array}$$

It is clear that *SpendZero* is an abstraction of *Spend*: it is only required to behave like *Spend* on part of its domain.

Dick and Faivre [10] described a method of test case generation based on "disjunctive normal forms" (DNFs). Properties like (1) are used to decompose operations into disjunctions of partial operations, with each such partial operation leading to a test case. Using property (1) leads to three sub-operations, distinguishing whether $m = 0$, either before or after the operation. (The fourth case, $m = 0 \wedge m' > 0$, cannot arise as *Spend* never increases the treasury.)

$$\mathit{Spend} == \mathit{Spend}_1 \vee \mathit{Spend}_2 \vee \mathit{Spend}_3$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Spend}_1} \\
\hline
\Delta \mathit{Treasury} \\
\mathit{req}? : \mathbb{N} \\
\mathit{out}! : \mathbb{N} \\
\hline
\mathit{out}! = \mathit{req}? \\
m' = m - \mathit{out}! \\
m' > 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Spend}_2} \\
\hline
\Delta \mathit{Treasury} \\
\mathit{req}? : \mathbb{N}_1 \\
\mathit{out}! : \mathbb{N}_1 \\
\hline
\mathit{out}! = \mathit{req}? \\
m' = m - \mathit{out}! \\
m' = 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Spend}_3} \\
\hline
\Delta \mathit{Treasury} \\
\mathit{req}? : \mathbb{N} \\
\mathit{out}! : \mathbb{N} \\
\hline
\mathit{out}! = \mathit{req}? = 0 \\
m' = m = 0 \\
\hline
\end{array}$$

By construction, such sub-operations will be disjoint, and together cover the original operation. They will not, in general, be abstractions of the original operation, because they impose restrictions on after-states.

In previous work [8], we explored the interaction between data refinement and DNF-based test case generation, in particular how disjointness and covering of calculated concrete test cases may be preserved.

3 Input and Output Abstractions in Z

Complementary to the "horizontal" decomposition approaches described in the previous section, this paper explores a "vertical" decomposition, concentrating on hiding inputs and outputs only. Recall that Z ADTs are *abstract*, in the sense that the state variables are *not* directly observable. Thus, projection on IO variables rather than on state variables is more pertinent, as it refers to directly observable behaviour. In terms of testing, an abstraction over an output variable represents disregarding the value of that output; an abstraction over an input variable represents the use of an arbitrary ("randomly" generated) input value. Both of these represent simpler tests than those where all inputs should be provided and the values of all outputs should be checked, and thereby potentially a useful simplification of the testing process.

In general, removal of certain input or output variables cannot be done purely syntactically. For example, if we remove $out!$ from $Spend$, with all the predicates that refer to it, we obtain

$$\begin{array}{|l}
\Delta \mathit{Treasury} \\
\mathit{req}? : \mathbb{N} \\
\hline
\end{array}$$

which is *not* an abstraction of $Spend$: it removes the constraint that $req? \leq m$. In other words, it guarantees a well-defined result in an area where $Spend$ did not.

A more appropriate way of hiding variables is semantically based, although it can be expressed syntactically in Z, viz. through existential quantification. For example, the output $out!$ is hidden in $Spend$ as[2]:

[2] In fact, an explicit hiding operator $\setminus$ exists in Z with the same semantics.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, out! : \mathbb{N} \bullet \mathit{Spend}} \\
\Delta \mathit{Treasury} \\
req? : \mathbb{N} \\
\hline
\exists\, out! : \mathbb{N} \bullet out! = req? \\
\qquad\qquad\quad m' = m - out! \\
\hline
\end{array}
$$

whose predicates simplify to $m' = m - req?$.

Although such “abstractions” over input and output variables are based on the standard semantics (logic and set theory), they are not therefore guaranteed to lead to converse refinements. As described in [4, 9], refinement essentially provides a second layer of semantics on top of the standard semantics. In fact, these abstractions are even guaranteed *not* to satisfy the refinement conditions, as they fail the condition that abstract and concrete operations have the same input and output signatures. This gap is bridged by the notion of IO refinement presented in the next section.

4 IO Refinement

IO refinement [5, 3] allows changes of inputs and outputs, and thereby changes the boundaries of the system. It is a strict generalisation of traditional Z refinement [15, 17], which does *not* allow such changes. Input and output form part of the observable behaviour. Thus, when performing IO-refinement we need to keep track of all changes to the inputs and outputs; this book-keeping is elided here, and for the technicalities (concerning “original input and output transformers”) we refer to [9].

Before defining IO refinement, we present the method used for modifying inputs and outputs: through composition with “IO transformers”, which are degenerate operations which have no state, just inputs and outputs.

Definition 3 (IO transformer) A Z schema S is an IO transformer iff $\Sigma S = ?S \wedge !S$, i.e., the signature of S contains only input and output components. □

For example, the schema

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{AnIT}} \\
req?, req! : \mathbb{Z} \\
\hline
req! = req? + 1 \\
\hline
\end{array}
$$

is an IO transformer: $?AnIT == [\, req? : \mathbb{Z} \,]$, $!AnIT == [\, req! : \mathbb{Z} \,]$.

Sometimes the converses of IO transformers need to be used; they are defined by swapping input and output roles. This is indicated by overlining, in analogy with CCS [13].

Definition 4 (IO decorations) For all component names x, let $\overline{x?}$ be the name $x!$, and let $\overline{x!}$ be the name $x?$. This definition is extended to IO transformers, analogous to the normal Z schema decoration conventions. □

The converse of $AnIT$ above is

$$
\begin{array}{|l}
\hline \multicolumn{1}{l}{\overline{AnIT}} \\
req?, req! : \mathbb{Z} \\
\hline
req! = req? - 1 \\
\hline
\end{array}
$$

An IO transformer is an input transformer for an operation if its outputs exactly match the operation's inputs, and analogously for output transformers. Particular IO transformers act as identities on the input and output side.

Definition 5 (Input and output transformers and identities) An IO transformer T is an *input transformer* for an operation Op iff $?Op = \overline{!T}$ and it is an *output transformer* for Op iff $!Op = \overline{?T}$.

For a schema S its *input identity* is defined by $\mathrm{IId}\ S == [\, ?S;\ \overline{?S} \mid \theta ?S = \theta\overline{?S} \,]$ and its *output identity* by $\mathrm{OId}\ S == [\, !S;\ \overline{!S} \mid \theta !S = \theta\overline{!S} \,]$. □

An input transformer IT is applied to operation Op in $IT \gg Op$. In the absence of name capture, the meaning of this is the conjunction of Op and IT, equating and hiding the matching inputs of Op and outputs of IT; an output transformer OT is applied in $Op \gg OT$.

Example 2 The IO transformer $AnIT$ above is an input transformer for $Spend$, its application leads to

$$
\begin{array}{|l}
\hline \multicolumn{1}{l}{AnIT \gg Spend} \\
\Delta Treasury \\
req? : \mathbb{N} \\
out! : \mathbb{N}_1 \\
\hline
out! = req? + 1 \\
m' = m - out! \\
\hline
\end{array}
$$

An output transformer for Tax is for example

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DelInc}} \\
\hline
m?, m! : \mathbb{N} \\
inc? : \mathbb{B} \\
\hline
m? = m! \\
inc? = (m? > 0) \\
\hline
\end{array}
$$

and its application to Tax leads to the removal of the output $inc!$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Tax} \gg \textit{DelInc}} \\
\hline
\Delta \textit{Treasury} \\
in? : \mathbb{Z} \\
m! : \mathbb{N} \\
\hline
in? > 0 \\
m' = m + in? \\
m! = m' \\
\hline
\end{array}
$$

□

Derivations of conditions for IO refinement are given in [9], using the standard relational model for Z. The rules derived generalise data refinement, with rules for both upward and downward simulation. In this paper we will only need the case where concrete and abstract state spaces coincide, which is covered by the accordingly restricted downward IO simulation rule below.

Definition 6 (Downward IO simulation) Consider ADTs $A = (State, Init, \{AOp_i\}_{i \in I})$ and $C = (State, Init, \{COp_i\}_{i \in I})$. Let IT be an input transformer for COp_i which is total on $?AOp_i$. Let OT be a total injective output transformer for AOp_i. C is a downward IO simulation of A iff for all $i \in I$

$$
\begin{array}{l}
\forall\, State;\ ?COp_i \bullet \mathrm{pre}(\overline{IT} \gg AOp_i) \Rightarrow \mathrm{pre}\, COp_i \\
\forall\, State;\ ?AOp_i;\ State';\ !COp_i \bullet \\
\qquad\qquad \mathrm{pre}\, AOp_i \wedge (IT \gg COp_i) \Rightarrow (AOp_i \gg OT)
\end{array}
$$

□

Example 3 For any suitable operation Op, using $AnIT$ above, $AnIT \gg Op$ or $Op \gg AnIT$ is a downward IO simulation, as $AnIT$ represents a total bijection.

Example 4 Due to the predicate on *inc?*, *DelInc* is injective, and thus $Tax \gg DelInc$ is a downward IO simulation of Tax. Intuitively, this shows that an output variable may be removed provided its value can be derived from the other outputs. □

Particular consequences of the downward IO simulation rule are embodied in the following theorem.

Theorem 1 (Simple input and output refinement) For any ADT, adding a declaration of a new output (from a non-empty set) to one of the operations constitutes a valid downward IO simulation.

For any ADT, adding a declaration of a new input (from a non-empty set) to one of the operations constitutes a valid downward IO simulation. □

5 Soundness of Input Abstraction

In this section we investigate the circumstances in which hiding of input variables constitutes a (converse) IO refinement. We consider two data types[3] with a single operation each[4]:

$$D = (State, Init, \{DOp\})$$
$$E = (State, Init, \{EOp\})$$

where Inp is a signature such that $Inp \sqsubseteq ?DOp$ and the operation in E is obtained by abstraction over Inp in D, i.e.,

$$EOp = \exists\, Inp \bullet DOp$$

as a consequence, $?EOp$, the input signature of EOp, is the schema containing the remaining inputs, and we have that[5] $?DOp = Inp \land ?EOp$. Another way of expressing EOp, using an input transformer, is

$$EOp = (\overline{Inp} \land \mathrm{IId}\ (?EOp)) \gg DOp$$

In proofs, we use the fact that (partial) IO identities have no effect in piping, in particular it is also the case that

$$EOp = \overline{Inp} \gg DOp$$

[3]They are, for once, not called A and C as we will consider possible refinement in both directions.

[4]There are situations, for example when considering *refusals*, where the restriction to a single operation allows stronger results. However, in this case the more general treatment would still be operation-by-operation, leading to a cluttered presentation.

[5]Schema conjunction for disjoint signatures is really a Cartesian product.

The case where we might have expected refinement to hold is where the "concrete" specification has input variables that the abstract one does not have. However, there is a proviso.

Theorem 2 D is a downward IO simulation of E if

$$\forall\, State;\ ?EOp \bullet (\exists\, Inp \bullet \operatorname{pre} DOp) \Rightarrow (\forall\, Inp \bullet \operatorname{pre} DOp)$$

(Informally: if DOp is enabled in any state for particular input from Inp, it is enabled in that state for *all* inputs from Inp.)

Proof

The relevant input transformer is $\overline{Inp} \wedge \mathrm{IId}\ (?EOp)$, which is total. The only non-trivial condition is applicability, which reduces to

$$\operatorname{pre}(Inp \gg \overline{Inp} \gg DOp) \Rightarrow \operatorname{pre} DOp$$

which is equivalent to the stated condition. $\square$

We might also state this condition as: the preconditions of the operations are independent of the values of the variables in Inp. A corresponding theorem can be proved for upward IO simulation, with the same condition.

Example 5 Consider the Tax operation, whose precondition includes the condition $in? > 0$. Thus, it does not satisfy the condition of Theorem 2, and indeed the following is *not* a converse refinement of Tax:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, in? : \mathbb{Z} \bullet Tax} \\
\Delta Treasury \\
m! : \mathbb{N} \\
inc! : \mathbb{B} \\
\hline
\exists\, in? : \mathbb{Z} \bullet in? > 0 \\
\qquad\qquad m' = m + in? \\
\qquad\qquad m! = m' \\
\qquad\qquad inc! = (m' > m) \\
\hline
\end{array}$$

We earlier observed that Tax allowed a refinement dropping the condition $in > 0$, and consequently also the postcondition that $m' > m$. The above operation does not allow such a refinement. $\square$

For completeness' sake, we also state the following.

Theorem 3 E and D are downward IO simulation equivalent if

$$\begin{array}{l}\forall\, State;\ State';\ ?DOp;\ !DOp \bullet \\ \qquad\qquad (\operatorname{pre} DOp \wedge Inp \gg \overline{Inp} \gg DOp) \Leftrightarrow DOp\end{array}$$

$\square$

The informal interpretation of this condition is that the input is irrelevant in DOp (note that $Inp \gg \overline{Inp}$ represents the full relation on Inp). Obviously this implies that the input is also irrelevant in the *precondition* of DOp. The downward IO simulation between E and D requires the $\Rightarrow$-part of this condition to guarantee correctness; the reverse implication always holds.

6 Soundness of Output Abstraction

In this section we investigate when abstraction over output variables constitutes a converse IO refinement. We consider data types

$$
\begin{aligned}
D &= (State, Init, \{DOp\})\\
F &= (State, Init, \{FOp\})
\end{aligned}
$$

with some signature $Outp \sqsubseteq\, !DOp$ where the operation in F is obtained by abstraction over $Outp$ in D, i.e.,

$$FOp = \exists\, Outp \bullet DOp$$

Thus, $!FOp$ characterises the outputs of DOp that remain present in FOp. An alternative characterisation is

$$FOp = DOp \gg \overline{Outp}$$

Observe that the preconditions of linked operations coincide in this case:

$$
\begin{aligned}
&\mathrm{pre}\, FOp\\
&= \exists\, State';\ !FOp \bullet FOp\\
&= \exists\, State';\ !FOp \bullet \exists\, Outp \bullet DOp\\
&= \exists\, State';\ !DOp \bullet DOp\\
&= \mathrm{pre}\, DOp
\end{aligned}
$$

Whenever $Outp$ has more than one possible value, the output transformer $\mathrm{OId}\,(!FOp) \wedge \overline{Outp}$ is not injective, and thus F can never be an IO downward (or upward) simulation of D using that output transformer. For that reason, we only need to consider whether D is an IO refinement of F. Observe that Theorem 1 does not apply, as that requires $DOp \gg \overline{Outp} \gg Outp = DOp$, which is only the case if, whenever one output is possible in any particular state, all other outputs are possible there, too.

However, there is still an unconditional positive result in this case:

Theorem 4 D is a downward IO simulation of F.

Proof
The required output transformer is $\mathrm{OId}\,(!FOp) \wedge Outp$, taking the "abstract" outputs for F and adding $Outp$ to those as "concrete" outputs for D. This is clearly injective.

Initialisation is unaffected. Due to equality of preconditions, applicability is guaranteed. Finally, correctness requires that $DOp \Rightarrow EOp \gg Outp$ which does indeed hold. $\square$

Example 6 In Example 4 we showed that $\exists\, inc! : \mathbb{N} \bullet Tax = Tax \gg DelInc$ was an IO refinement of Tax; in combination with the above result we can now conclude that they are even equivalent with respect to IO refinement. $\square$

7 Completeness of Output Abstraction

The previous sections have investigated situations where abstractions of input and output variables were valid tests of a specification with respect to IO refinement. One might view these as *soundness* conditions: passing such a test is a necessary condition for any implementation to be correct with respect to the specification.

In this section, we consider situations in which passing such a test is also a *sufficient* condition. As the input abstractions were only conditionally sound, we concentrate here on completeness of output abstractions.

Clearly, in general, we cannot expect to be able to test a system by *never* checking a particular output. Example 6 gave an exception: where one output is fully determined by another, we might as well *not* check the first output.

Viewing these abstractions as "projections" of a specification invites a geometric analogy: we might wonder what the requirements would be for the sets of variables projected onto to be a *basis*, i.e., when the projections together determine the specification as a whole.

We cannot characterise this directly by refinement: the specification is a refinement of each of its projections, but the reverse only holds if the specification and *each* projection are all equivalent. However, we can identify a specification with the set of all its refinements; if all *joint* refinements of *all* projections are also refinements of the original specification, then the projections and the original specification are "equivalent". (Compare the construction of least common data refinements in [6].) This is characterised in the following definition.

Definition 7 (Basis) Consider an ADT $D = (State, Init, \{DOp\})$. Let $\{NF_j\}_{j \in J}$ be a collection of subsignatures of $!DOp$, and let $F_j = \exists\, NF_j \bullet DOp$ be the corresponding collection of output abstractions.

Then, $\{NF_j\}_{j \in J}$ is a *basis* of D iff for all ADTs $H = (State, Init, \{HOp\})$ with input and output signatures identical to D, whenever H is a downward IO simulation of each of F_j with respect to output transformer NF_j, HOp is an operation refinement of DOp. □

Because NF_j is an output *signature*, the operation $F_j \gg NF_j$ has the same behaviour as DOp, except for producing arbitrary values for outputs in NF_j. Thus, F_j and $F_j \gg NF_j$ are equivalent when interpreted as predicates in a context where ΣDop is defined.

Note that a basis is represented by sets of variables that are *hidden* rather than their complements, the variables projected onto. In particular, including the empty subsignature (i.e., projecting onto the full set of variables) will always lead to a basis.

Using the particular relation between DOp and F_j, we can eliminate the quantification over all H in Definition 7:

Theorem 5 (Basis condition) For nonempty J, $\{NF_j\}_{j \in J}$ is a basis for D if

$$\forall\, State;\ State';\ ?DOp;\ !DOp \bullet (\forall\, j : J \bullet \exists\, NF_j \bullet DOp) \Rightarrow DOp$$

Proof We need to prove that HOp is an operation refinement of DOp, using the fact that H is a downward IO simulation of each of F_j. Recall that the conditions of operation refinement are

$$\begin{aligned} \forall\, State;\ ?DOp \bullet \operatorname{pre} DOp \Rightarrow \operatorname{pre} HOp \\ \forall\, State;\ State';\ ?DOp;\ !DOp \bullet HOp \wedge \operatorname{pre} DOp \Rightarrow DOp \end{aligned}$$

The first ("applicability") condition requires that J is non-empty. From IO-refinement from F_j to H (with the identity input transformer), we get

$$\operatorname{pre} F_j \Rightarrow \operatorname{pre} HOp$$

and we also have that

$$\operatorname{pre} F_j = \operatorname{pre} DOp$$

by the construction of F_j. Together these prove the first condition.

For the correctness condition, we have from IO-refinement that

$$\forall j : J \bullet \operatorname{pre} F_j \land HOp \Rightarrow F_j \gg NF_j$$

which, using $\operatorname{pre} F_j = \operatorname{pre} DOp$ is equivalent to

$$\forall j : J \bullet \operatorname{pre} DOp \land HOp \Rightarrow F_j \gg NF_j$$

and thus to

$$\operatorname{pre} DOp \land HOp \Rightarrow \forall j : J \bullet F_j \gg NF_j$$

Assuming the condition of the theorem, this proves the correctness condition.

□

The interpretation of the basis condition is: outputs can be verified independently, provided that all constraints on their values can be defined in terms of the values of state variables and inputs *only*.

Example 7 As we already showed that Tax and $Tax \gg DelInc$ are equivalent, it follows that projections on $inc!$ and $m!$ separately or jointly, or on $m!$ only, are bases for Tax.

Note that the values of these two outputs are *not independent* of each other; however, in any given state, the value of each can be determined independently from the other from the state only. □

Example 8 For an operation which has no output projections which form a basis apart from itself, consider the operation where the king's treasury is non-deterministically split between his sons.

$$\begin{array}{|l}
\multicolumn{1}{l}{Succession} \\
\hline
\Delta Treasury \\
william!, harry! : \mathbb{N} \\
\hline
william! + harry! = m \\
m' = 0 \\
\hline
\end{array}$$

The projection on one of the outputs (the other is symmetric) is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\exists\, william! : \mathbb{N} \bullet Succession} \\
\hline
\Delta Treasury \\
harry! : \mathbb{N} \\
\hline
harry! \leq m \\
m' = 0 \\
\hline
\end{array}$$

and the conjunction of the two projections is weaker than *Succession*: it contains $harry! \leq m \wedge william! \leq m$ rather than $harry! + william! = m$. Thus, the collection of projections on individual outputs fails the basis condition. □

8 Concluding Comments

The results in this paper were developed in Z, using a non-standard refinement relation. The outcomes were reasonably intuitive: hiding an output is always sound, hiding an input may not be; outputs can sometimes be verified independently. In the underlying semantic framework [9, 11, 17], inputs and outputs are included in sequences, which are both part of the "hidden" local state and the "visible" global state. As such, they play a similar role to "normal" [7] or visible variables in guarded command language based refinement, and our results should carry over to such a context. In the context of (deterministic) programming languages, projections over such sets of visible variables are a special case of *slicing* [14], using a statement that exhibits their values. Slicing is an accepted basis for program verification and testing in that context [2, 12].

Finally, note that most of our results were developed for data types with a single operation. This is no fundamental restriction, as the simple kinds of IO refinement considered lead to conditions per individual operation. Analogous results for ADTs with multiple operations would be obscured by the abundance of i-indexed sets of projections and IO transformers.

Acknowledgement

John Derrick provided useful comments on an earlier version of this paper.

References

[1] B.K. Aichernig. Test-case calculation through abstraction. In J.N. Oliveira and P. Zave, editors, *FME 2001*, volume 2021 of *Lecture Notes in Computer Science*, pages 571–589. Springer-Verlag, 2001.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 384–396. ACM, 1993.

[3] E. A. Boiten and J. Derrick. Liberating data refinement. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 144–166. Springer-Verlag, 2000.

[4] E.A. Boiten. Loose specification and refinement in Z. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2002.

[5] E.A. Boiten and J. Derrick. IO-refinement in Z. In A. Evans, D. Duke, and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, September 1998. http://www.ewic.org.uk/.

[6] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, September 1999.

[7] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[8] J. Derrick and E. A. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification & Reliability*, 9:27–50, 1999.

[9] J. Derrick and E.A. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT. Springer Verlag, May 2001.

[10] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[11] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[12] R.M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, March 2002.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[14] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, 1996.

[15] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[16] S. Stepney. Testing as Abstraction. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 137–151, Limerick, September 1995. Springer-Verlag.

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

Note

Section 7 can probably be simplified by first proving that every HOp in Definition 7 is an operation refinement of $\forall j : J \bullet F_j \gg NF_j$, which I suspect is the least common IO refinement of all F_j. Note that the reverse implication of the basis condition is universally true. All in all this should lead to "iff" in Theorem 5.