Formal Program Development with Approximations

Eerke A. Boiten[1] and John Derrick[2]

[1] Computing Laboratory, University of Kent at Canterbury
[2] Department of Computer Science, University of Sheffield
E.A.Boiten@kent.ac.uk,J.Derrick@dcs.shef.ac.uk

Abstract. We describe a method for combining formal program development with a disciplined and documented way of introducing realistic compromises, for example necessitated by resource bounds. Idealistic specifications are identified with the limits of sequences of more "realistic" specifications, and such sequences can then be refined in their entirety. Compromises amount to focusing the attention on a particular element of the sequence instead of the sequence as a whole.
This method addresses the problem that initial formal specifications can be abstract or complete but rarely both. Various potential application areas are sketched, some illustrated with examples. Key research issues are found in identifying metric spaces and properties that make them usable for refinement using approximations.

Keywords: Refinement, approximations, metric spaces.

1 Introduction

In formal program development, one starts with a complete formal statement of the problem to be solved, and then develops a program by gradually adding detail to the solution. In practical program development, one starts with an incomplete informal problem statement, and then develops a program by adding detail to the solution (and implicitly also to the specification). The difference between the two approaches thus appears immediately in two aspects: *formality* and *completeness* of the initial specification. Just using a *formal* initial specification is a definite[1] improvement on the practical development process; however, insisting that the initial specification be *complete* requires a radical change in the process.

Indeed, it is sometimes argued that this assumption of completeness makes formal development inadequate in practice. Initial specifications often *cannot* be complete and abstract at the same time. Consider, for example, a garbage collector for a programming language. An idealistic specification would say that it collects all memory cells that have become inaccessible whenever they become

[1] Disregarding issues of communication – assume a specification language is used which transliterates to English or the diagrammatic notation *du jour*.

H. Treharne et al. (Eds.): ZB 2005, LNCS 3455, pp. 375–393, 2005.

so. As this is unrealistic and probably not desirable, one might specify it as: "*periodically* collects *some* memory cells which have become inaccessible". Unfortunately, without quantifying "some" and "periodically", such a specification is likely to be inadequate, also satisfied by a program which collects no cells on a yearly cycle. Replacing "some" and "periodically" by explicit values makes the specification less abstract; worse, it may only be evident much later in the development process which values are realistic or optimal. An informal incomplete specification which can *avoid* using either "all" or "some" appears to be at an advantage here.

This paper presents an approach which addresses this issue through using *chains* of specifications, which are considered equivalent to their limits. Different examples may be based on different limits for the chain, based on measures of distance between specifications (i.e., metrics), or orderings such as degree of fairness or randomness, machine integer size, floating point precision, memory space, etc. The limits represent idealised behaviour, such as unbounded integers, infinite floating point precision, etc. Elements in the chain represent approximations to the limits, which for sensible orderings behave "correctly" within boundaries which can be characterised precisely. Metrics measure the distance between specifications, and thus provide a notion of convergence to idealised behaviour from approximations which might be realisable as implementations.

The corresponding development process starts with an initial "optimistic" specification. At some stage, this is replaced by a chain of specifications whose limit it represents. Further development then refines[2] the chains uniformly in their entirety for as long as possible. This implies that "the position in the chain" – often a resource constraint – is treated as a parameter of the specification. The, a specific member of the chain is selected for further development – this step represents a *compromise* with respect to the limit specification, and will generally *not* preserve correctness. However, by postponing this step for as long as possible, any further development which relies on the specific choice of compromise is isolated from the more "generic" development up to that point, which could later be reused for a *different* choice. In this way, the development trail clearly documents where resource constraints were first taken into consideration, and most importantly: at which point the resource constraint's impact on the specification forced a compromise. One might call this process "approximate refinement", but it is actually refinement interspersed with isolated and highlighted approximation steps.

This process addresses the "incompleteness vs. abstraction" issue by allowing initial specifications to be optimistic, assuming for example unbounded resources. For as long as any *mention* of resource bounds can be avoided, it *is*; then, we introduce the resource bound but continue to develop independently of its value for as long as possible. This may go all the way to a code level constant,

[2] This paper will use the term "refinement" to describe any formal development process, without implying the post-hoc verification bias sometimes associated with "refinement".

that is later instantiated on the basis of empirical information such as testing or profiling.

Apart from the issue of limited resources, a number of other areas which traditional refinement does not cover in a satisfactory way are addressed by this approach. One of these is *probabilistic algorithms* which can come arbitrarily close to the exact solution but are not guaranteed to ever reach it. Floating point precision was already mentioned above; in general, any system which should be dealing with real numbers (e.g. hybrid systems) at some stage should be dealing with the approximation by floating point numbers. Also, the notion of *urgency* in real-time formalisms, i.e., actions that happen instantaneously once they are enabled, is often an unsatisfactory way of describing "as soon as possible"; approximation is useful in this context as well.

No particular specification notation or refinement relation has been assumed so far – indeed, apart from the specific notions of metrics and limits, the approach described here applies independently of such choices. However, examples in this paper will mostly use relational refinement [5, 6] using Z [14] as a concrete syntax. This is briefly described in Section 2.

Section 3 presents sequences of specifications, identified with their limits, and describes a formal development method using sequences. The rich mathematics of chains and convergence leads to a number of foundational questions. Some of these are discussed in Section 4 which considers limits defined by metrics over the space of specifications. We discuss possible approaches to defining metrics for use with refinement, and sketch a number of these. Section 4.3 discusses open issues concerning metrics, limits and their induced topology. Section 5 considers the relationship between refinement of limits and element-wise refinement of chains.

Various ways of addressing limitations of refinement have been published previously; these are discussed in Section 6, and we conclude in Section 7.

2 Refinement

Although the approach described in this paper is independent of the choice of specification notation, we fix a notation and notion of refinement for use in examples. Our basic formalism is that of alphabetised relations [9], characterised by predicates, using Z as a concrete syntax.

Where we consider a single operation on a fixed state, we may give its frame directly, e.g., $\Delta [x : \mathbb{N}]$, rather than define an explicit state schema.

Refinement for Z is described in the monograph [6]; we omit many of the more general definitions and technical details here. For the examples in this paper, algorithmic or *operation* refinement mostly suffices. We first recap on the basic definition of refinement.

A Z specification defines a data type consisting of partial relations, $\mathsf{D} = (\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i \in I}, \mathsf{Fin})$, where I is the alphabet of the data type (its set of operations). Each such data type induces a number of potential programs, each program being a relation over some global state G (the final element Fin is the finalisation which details what is observable, its technicalities need not concern

us here). In the context in which we are working, a program will be characterised by a sequence of operation indices (i.e., elements of I), and each such sequence defines a program. E.g., if $\mathsf{p} = \langle \mathsf{p}_1, ..., \mathsf{p}_n \rangle$ then $\mathsf{p}_D = \mathsf{Init} \fatsemi \mathsf{Op}_{\mathsf{p}_1} \fatsemi ... \fatsemi \mathsf{Op}_{\mathsf{p}_n} \fatsemi \mathsf{Fin}$.

We use the standard definition of refinement [5], defined in terms of these potential programs. The relational inclusion ensures that the observations produced by data type C must be consistent with those produced by data type A for the same sequence of inputs.

Definition 1 (Data Refinement). *For data types* A *and* C, C refines A, *written* $\mathsf{A} \sqsubseteq_{data} \mathsf{C}$, *iff for each program* p *over* I, $\mathsf{p}_C \subseteq \mathsf{p}_A$. *Further, we write* $=_{data}$ *to denote data refinement in both directions.* □

In general two methods are used to verify such refinements: downward and upward simulations. In this paper they coincide because we will use retrieve relations which are the identity. Thus we use the following.

Definition 2 (Operation Refinement). *Operation COp is an operation refinement of the operation AOp on the same state, using the same inputs and outputs, iff*

$$\text{pre } AOp \Rightarrow \text{pre } COp$$
$$\text{pre } AOp \wedge COp \Rightarrow AOp$$

hold for all states, inputs and outputs. pre *Op denotes the "domain" of Op: those before-states and inputs for which related after-states and outputs are defined by Op.*

This definition is extended operation-wise to abstract data types of the form $(State, Init, \{Op_i\}_{i \in I})$ (both using the same state) with sets of operations and initialisation (as an operation with an irrelevant before-state). The same conditions hold for operation refinement when the abstract state is a subset of the concrete one.

However, in a real development it could be argued that this ideal is not always obtained. Consider the following two examples.

2.1 Example - Bounded Buffer

Part of an abstract specification might specify an abstract buffer:

Buffer

$cont : \text{seq } Item$

EmptyBuf

$Buffer'$

$cont' = \langle\,\rangle$

Remove

$\Delta Buffer$

$out! : Item$

$cont = \langle out! \rangle \frown cont'$

$$
\begin{array}{|l}
\hline
\mathit{Insert}_\infty \\
\Delta \mathit{Buffer} \\
\mathit{in}? : \mathit{Item} \\
\hline
\mathit{cont}' = \mathit{cont} \frown \langle \mathit{in}? \rangle \\
\hline
\end{array}
$$

However, this might be implemented by a bounded buffer of a particular size, e.g., $n = 256$. In this implementation, Insert_∞ is replaced by Insert_n, where this is defined as:

$$
\begin{array}{|l}
\hline
\mathit{Insert}_n \\
\Delta \mathit{Buffer} \\
\mathit{in}? : \mathit{Item} \\
\hline
(\#\mathit{cont} < n \land \mathit{cont}' = \mathit{cont} \frown \langle \mathit{in}? \rangle) \lor (\#\mathit{cont} \geq n \land \mathit{cont}' = \mathit{cont}) \\
\hline
\end{array}
$$

This bounded buffer is only an approximate refinement (in some sense) of the abstract infinite buffer – it certainly does not meet the requirements of the definition of operation refinement.

2.2 Example - Add

We could also imagine a development involving several steps that include an approximation at the end. One that starts with Add_∞, replaces it by Add_n, refines that to ModAdd_n and finally instantiates n to `maxint`. Here the specifications are:

$$
\begin{array}{|l}
\hline
\mathit{Add}_\infty \\
\Delta [\, x : \mathbb{N} \,];\ \mathit{add}? : \mathbb{N} \\
\hline
x' = x + \mathit{add}? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\mathit{Add}_n \\
\Delta [\, x : \mathbb{N} \,];\ \mathit{add}? : \mathbb{N} \\
\hline
x' = x + \mathit{add}? \land x' < n \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{ModAdd}_n \\
\Delta [\, x : \mathbb{Z} \mid -n \leq x < n \,] \\
\mathit{add}? : \mathbb{N} \\
\hline
x' \bmod (2 * n) = (x + \mathit{add}?) \bmod (2 * n) \\
\hline
\end{array}
$$

The final result is not a correct implementation of the original specification, but it can be formally traced back to it, recording the approximations needed in its development.

3 Refinement and Approximation with Chains

To model these types of approximate refinements we consider chains of specifications (S_n), where we will identify a chain with its limit:

Definition 3. *A sequence of specifications* $(S_n)_{n\in\mathbb{N}}$ *is considered equivalent to its limit.*

Sequences which do not have a limit are not considered meaningful. Different approximations give different notions of a limit, and we consider examples of these in section 4. In this section, we describe an envisaged formal development process using chains of specifications. (For simplicity, we do not mention sequences with multiple indices, although they are definitely not excluded.)

The development process contains four kinds of steps:

Element-wise Refine. A refinement step is a normal development step of the notation at hand, applied to the current specification. When we are dealing with a sequence (S_n), we apply such a step uniformly to each of its elements.

Introduce Sequence. At any time, we may replace a specification S with a sequence of specifications (S_n) such that S is the limit of (S_n).

Replace Sequence. A sequence (S_n) may be replaced by a sequence (T_n) whose limit is identical to, or a refinement of, the limit of (S_n).

Compromise. We replace a sequence (S_n) with one of its elements S_n.

Sequence introduction and replacement are correct steps, this follows directly from our interpretation of sequences. Conditions for the correctness of refinement steps applied element-wise are discussed below, where we also explain what we mean by the "uniform" application of such steps. A compromise step generally does not preserve correctness, and for that reason forms a central part of the development documentation.

Example 2.2 uses sequence introduction when we replace Add_∞ by the sequence (Add_n) and element-wise refinement in the refinement to $(ModAdd_n)$, and the final step is a compromise, i.e., when we instantiate n to `maxint`.

A variation on the bounded buffer example given previously might start development from the unbounded buffer, replace it by (Buf_n) (sequence introduction), then by $(Buf_{2*n})_{n\in\mathbb{N}}$ (sequence replacement, the limit of even-sized buffers is still the infinite buffer), implement that by serial composition of two copies of Buf_n (element-wise refinement), and finally instantiate n to 256 (compromise).

The definition of refinement between sequences is inherited from their identification with limits. Thus, we have:

Theorem 1. *The sequence* (S_n) *with limit* S_∞ *is refined by the sequence* (T_n) *with limit* T_∞ *iff* S_∞ *is refined by* T_∞.

Clearly our main objective in refining chains should be to use element-wise refinement as much as possible. Sequence replacement is complete by definition for sequence refinement (assuming a complete rule for refinement of their limits), but defeats the purpose of moving to sequences. Its use is probably best limited to replacing sequences which converge "at different speeds", such as in the even-sized buffer example above. However, element-wise refinement does *not* necessarily lead to refinement between sequences; this is discussed further in Section 5 below.

4 Metrics and Limits

The above discussion on orders and limits is the precursor to a discussion on alternative approaches to limits in terms of metrics (and hence topologies). We first recap on the standard definitions. Limits are defined in terms of convergence, which is relative to a particular distance function: a *metric*.

Definition 4 (Metric). *A metric on a space A is a function $d : A \times A \to \mathbb{R}$ such that $\forall\, x, y, z \in A$:*

$$
\begin{array}{l}
d(x, y) \geq 0 \\
d(x, y) = d(y, x) \\
d(x, y) = 0 \textit{ iff } x = y \\
d(x, y) \leq d(x, z) + d(z, y)
\end{array}
$$

□

The limit of a sequence is defined as the point of convergence with respect to a metric.

Definition 5 (Limit of a Sequence). *A sequence s_n converges to s, denoted $s_n \to s$, whenever:*

$$\forall\, \epsilon > 0 \bullet \exists\, N \bullet \forall\, n > N \bullet d(s_n, s) \leq \epsilon$$

□

There are several approaches to defining metrics and topologies for specifications, to understand what is relevant in terms of refinement we return to its definition. This was defined as consisting of program observation, expressed as $\mathsf{p_C} \subseteq \mathsf{p_A}$, where $\mathsf{p_C}$ is a finite sequential composition of operations from C. From this there are two immediate parameters which can be used to define metrics: the consistency of observations as represented by $\subseteq$, and the programs themselves. We discuss each of these in turn.

4.1 Program Length

Data refinement asks for consistency of observations for all programs. We can define a metric by assigning a distance to specifications which agree on observations up to a certain length. This is easiest if phrased in terms of equivalence (i.e., data refinement in both directions). Thus we define

Definition 6 (Program Length Metric). *We define the metric d_l on specifications as follows:*

$$d_l(A, C) = \begin{cases} 0 & \textit{if } A =_{data} C \\ 2^{-n} & \textit{if } n = min\{m : \mathbb{N} \mid \exists\, \mathsf{p} \bullet \mathsf{p_C} \neq \mathsf{p_A} \wedge \#\mathsf{p} = m\} \end{cases}$$

where the length of a program is the number of operations plus one (for the initialisation). □

It should be clear that this defines a metric on the set of equivalence classes (with respect to $=_{data}$) of specifications. The basis of this metric, see [4], is the idea that two specifications are close if it takes a long time to tell them apart, where a 'long time' is the length of the program before the difference is observed.

Limits with respect to this metric are characterised as follows. The sequence S_n converges to S whenever, $d_l(S_n, S) \to 0$, i.e., $2^{-n} \to 0$ where n is the minimum length of program needed to distinguish S_n from S.

Buffer Example. This metric works well on the buffer example. The shortest program that can observe that Buf_n, a buffer of size n, does not have infinite capacity has size $2n + 2$: first $n + 1$ elements are inserted (the last of which is the first one to be ignored), then n *Remove* operations are successful, and the next *Remove* operation fails[3].

Thus $d_l(Buf_n, Buf) = 2^{-(2n+3)}$ and so $Buf_n \to Buf$.

This metric has thus here correctly formalised our intuition that Buf_n gets closer to its idealised behaviour as n gets larger. The metric quantifies this closeness numerically.

Notice that the definition of the metric is, as one would hope, not sensitive to small changes. For example, if we consider finite and infinite stacks (i.e., inserting and removing from the same end) we can observe the difference more quickly than in the buffer example (since we do not have to remove all the elements first). However, the distance is $2^{-(n+1)}$, and thus we still get convergence to the infinite stack as one would expect.

Although its definition seems to assume observations being characterised by outputs, it also works for specification styles that use other notions of observability [7]. This is because there will always be a minimum length program where any difference can be observed – whether that be due to an output of an operation, or an explicit finalisation after the last operation.

Add Example. Applying this metric to the second example results in the following. First, note that as it stands the specifications are data refinement equivalent (for any n) since no difference can be *observed*. So let us add an observer to both:

Obs

$\Xi [x : \mathbb{N}]; \ out! : \mathbb{N}$

$out! = x$

Denoting the two specifications by A_n and A_∞, we find that $d_l(A_n, A_\infty) = 2^{-3}$ since the sequence *Init*; *Add*; *Obs* will observe a difference for any input

[3] The standard semantics of applying *Remove* outside its precondition allows for any result including the "correct" one and a completely undefined one; however, we are looking for *equality* of semantics of programs rather than *inclusion*.

bigger than n. Thus, with respect to this metric, we do not get convergence. This is despite the sequence (Add_n) being ordered by refinement:

$$\text{pre}\, Add_n = x + add? < n \Rightarrow \text{pre}\, Add_{n+1} = x + add? < n + 1$$

and

$$\begin{array}{l} \text{pre}\, Add_n \wedge Add_{n+1} \\ \equiv \\ x + add? < n \wedge x' = x + add? \wedge x' < n + 1 \\ \equiv \\ x + add? < n \wedge x' = x + add? \wedge x' < n \\ \equiv \\ Add_n \end{array}$$

The use of this metric on this example could be criticised because, although $d_l(A_n, A_\infty) = 1/2^3$, the behaviour is correct for some inputs. That is, this definition stresses quantification over programs at the expense of quantification over inputs and outputs. Consider, for example,

Op_N

$\Delta [x : \mathbb{N}]$
$x? : \mathbb{N}$
$y! : \mathbb{B}$

$x? \neq N \Leftrightarrow y!$

Op

$\Delta [x : \mathbb{N}]$
$x? : \mathbb{N}$
$y! : \mathbb{B}$

$y!$

then (assuming each specification is completed in an obvious way) $d_l(Op_N, Op) = 1/4$ despite the fact that Op and Op_N have identical behaviour for all but one input. The metric in Section 4.2 tackles this issue.

Metrics via Probability Distributions. It should be noted that the program length metric is a worst case analysis. The shortest program that one can observe the difference on is used to determine the difference, irrespective of whether that program is likely to appear in practice. Such a worst case analysis is useful in, for example, safety analysis. However, it might well be that other analyses are useful on occasion, and this would involve the use of a different metric. To determine the correct measure, a probability distribution would have to be assigned to the possible programs occurring, and this probability would be reflected in the distance calculated.

For example, let $\pi : \mathsf{P} \to [0, 1]$ be a probability distribution on the space of all finite programs P (which is countable). Thus $\pi(\mathsf{p})$ represents the probability that p will occur. At this moment we abstract away from any discussion of time intervals over which programs may be invoked, but assume they occur in some unspecified interval, kept finite (along with the program length) to avoid any issues of fairness.

Definition 7. *We say a probability distribution π is observationally consistent over a set of specifications S if, for all $A, C \in S$ we have $A \neq_{data} C \Rightarrow \exists\, \mathsf{p}.\pi(\mathsf{p}) > 0 \wedge \mathsf{p_A} \neq \mathsf{p_C}$.* □

This is needed to ensure that we can observe the non-equivalent specifications. It is a natural requirement to seek, since $A \neq_{data} C$ means we can observe a difference in behaviour, and this will only be the case if there is a non-zero probability of a program being invoked which exhibits that difference. We can now define a metric with respect to such a probability distribution as follows.

Definition 8. *Let π be an observationally consistent probability distribution over a set of specifications S. Define d_π by*

$$d_\pi(A, C) = \begin{cases} 0 & \text{if } A =_{data} C \\ p & \text{if } p = \Sigma\{\pi(\mathsf{p}) \mid \mathsf{p} : \mathsf{P} \wedge \mathsf{p_C} \neq \mathsf{p_A}\} \end{cases}$$

□

Thus d_π measures the distance in terms of a probability that the non data-equivalence will be observed by one of the programs.

Theorem 2. *d_π is a metric on the set of equivalence classes (with respect to $=_{data}$) of specifications.*

Proof:
1. Non-negativity and symmetry are obvious.

2. $d_\pi(A, C) = 0$ iff $A =_{data} C$ follows from π being observationally consistent.

3. For the triangle equality, given non-equivalent specifications A, B, C. Let p be a program with $\mathsf{p_C} \neq \mathsf{p_A}$ and $\pi(\mathsf{p}) = p$; then either $\mathsf{p_B} \neq \mathsf{p_A}$ or $\mathsf{p_C} \neq \mathsf{p_B}$ (or indeed both), thus p will be in the sum of probabilities in the measure $d_\pi(A, B)$ or $d_\pi(B, C)$. This holds for all elements in the sum $\Sigma\{\pi(\mathsf{p}) \mid \mathsf{p} : \mathsf{P} \wedge \mathsf{p_C} \neq \mathsf{p_A}\}$, and thus

$$\Sigma\{\pi(\mathsf{p}) \mid \mathsf{p} : \mathsf{P} \wedge \mathsf{p_C} \neq \mathsf{p_A}\} \leq \\ \Sigma\{\pi(\mathsf{p}) \mid \mathsf{p} : \mathsf{P} \wedge \mathsf{p_B} \neq \mathsf{p_A}\} + \Sigma\{\pi(\mathsf{p}) \mid \mathsf{p} : \mathsf{P} \wedge \mathsf{p_C} \neq \mathsf{p_B}\}$$

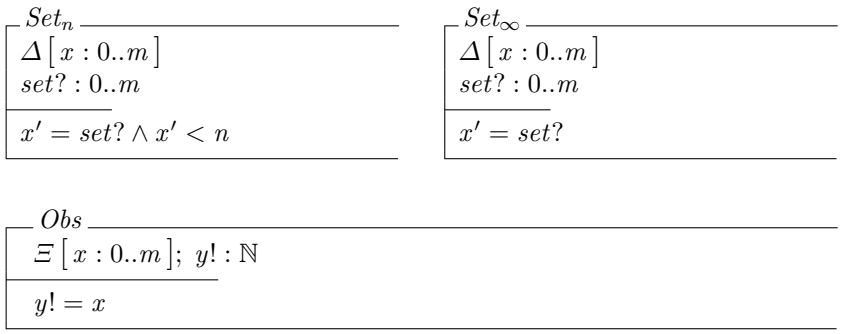
as required. □

Other variants of such a metric are clearly feasible and this approach needs to be assessed against practical as well as theoretical relevance.

4.2 Input/Output Metrics

An alternative to a metric based around program length is one that 'counts' the inputs/outputs for which the concrete specification correctly refines the abstract one. Due to the issue of counting over an infinite domain such as $\mathbb{N}$, we first consider bounded data types before generalising to unbounded ones.

Bounded Data Types. Consider a simplification of the *Add* example given as follows.



We want to define a metric which counts the values for which $(Init, Set_n, Obs)$ differs from $(Init, Set_\infty, Obs)$. This time we will base our metric on the simulation rules (i.e., operation refinement as in Definition 2), in contrast to the metric in Section 4.1, which used the basic definition of data refinement.

First note that the temptation to just count the outputs (i.e., observations) is not sufficient: *Obs* in the concrete is clearly a correct refinement of *Obs* in the abstract. Thus if we are to base a metric on the simulation rules we will clearly need to consider both inputs and outputs. This means we will consider refinement of both preconditions and postconditions. Again we will define a metric d that is zero on data refinement equivalent specifications, this will be defined in terms of the maximum distance between the constituent operations:

$$d(A, C) = max_{i \in I}\, d(AOp_i, COp_i)$$

and the distance between two operations in terms of an asymmetrical distance based on applicability and correctness:

$$d(AOp, COp) = max\{\rho(AOp, COp), \rho(COp, AOp)\}$$
$$\rho(AOp, COp) = \rho_a(AOp, COp) + \rho_c(AOp, COp)$$

Here ρ_a will measure distance in preconditions, and ρ_c distance in correctness. Both will count values where failure occurs, and return the ratio of this to the size of the input/output domain as the distance. Thus we define (with suitable generalisation):

Definition 9 (Input/Output Metric). $\rho_a(AOp, COp) = \frac{(\#Y - \#T)}{\#Y}$ *where* Y *is the type of the input* $x?$ *and* T *is the largest set* $T \subseteq Y$ *such that*

$$\forall\, x? : T \bullet \forall\, State \bullet \text{pre}\, AOp \Rightarrow \text{pre}\, COp$$

$\rho_c(AOp, COp) = \frac{(\#Z - \#T)}{\#Z}$ *where* Z *is the type of the output* $y!$ *and* T *is the largest set* $T \subseteq Z$ *such that*

$$\forall\, y! : T \bullet \forall\, State;\ State';\ x? : Y \bullet \text{pre}\, AOp \land COp \Rightarrow AOp \qquad \square$$

Limits, and convergent sequences, with respect to this metric are characterised as follows. The sequence S_n converges to S whenever the number of inputs and outputs for which S_n and S are not equivalent tends to zero.

We can calculate the distance between $A_n = (Init, Add_n, Obs)$ and $A_\infty = (Init, Add_\infty, Obs)$:

$$\begin{aligned}
&d(A_n, A_\infty) = d(Add_n, Add_\infty) && \text{since } Obs \text{ is identical in } A_n \text{ and } A_\infty \\
&= \rho(Add_\infty, Add_n) && \text{since } Add_n \sqsubseteq Add_{n+1} \\
&= \rho_a(Add_\infty, Add_n) && \text{since there are no outputs} \\
&= (m + 1 - \#T)/(m + 1)
\end{aligned}$$

where T is the largest $T \subseteq 0..m$ for which $add? : T \bullet add? < n$, hence

$$d(A_n, A) = \begin{cases} (m + 1 - (n - 1))/(m + 1) & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$$

Hence $A_n \to A_\infty$ with respect to this metric. In a similar way with the *Add* example from Section 2.2 we also get $A_n \to A_\infty$. The following proposition follows directly.

Proposition 1. *d is a metric on the set of equivalence classes (with respect to refinement) of specifications.*

Proposition 2. *If $S_i \sqsubseteq S_{i+1}$ and S is the least upper bound in the refinement ordering (i.e., $S_i \sqsubseteq S$ and no other $S_i \sqsubseteq T \sqsubseteq S$ with $T \neq_{data} S$), then $d(S_i, S) \to 0$.*

Proof
It suffices to consider one operation Op_i. Since $S_i \sqsubseteq S_{i+1}$ we have

$$\begin{aligned}
&\text{pre } S_i \Rightarrow \text{pre } S_{i+1} \\
&\text{pre } S_i \wedge S_{i+1} \Rightarrow S_i
\end{aligned}$$

and the distance will depend upon convergence of the following

$$\begin{aligned}
&\text{pre } S_{i+1} \Rightarrow \text{pre } S_i \\
&\text{pre } S_{i+1} \wedge S_i \Rightarrow S_{i+1}
\end{aligned}$$

Now since the sequence (S_i) is bounded above and the types are bounded, the preconditions must eventually converge, i.e., $\exists N \bullet \forall n > N \bullet \text{pre } S_n = \text{pre } S$.

Similarly, correctness will allow reduction of non-determinism in output and after-state. With the bounded output type, once the preconditions have converged, the correctness must do so also, i.e., $\exists M > N \bullet \forall m > M \bullet S_m =_{data} S$. □

Notice that this proposition is also true for the metric in Section 4.1.

In order to calculate a ratio $\frac{\#Y - \#T}{\#Y}$ the input and output types must obviously be bounded, however, the state does not have to be bounded. However, a

non-finite state can lead to differences in the distance calculated as the following example shows.

$State_1$
$x : 0..m$

$State_2$
$x : \mathbb{N}$

$Add_n[S]$
ΔS
$add? : 0..m$

$x' = x + add? \land x' < n$

$Add_\infty[S]$
ΔS
$add? : 0..m$

$x' = x + add?$

We can calculate $d(Add_n[State_1], Add_\infty[State_1]) = 1 - \#T/(m+1)$, where, similarly to before, we find that

$$T = \begin{cases} \varnothing & \text{if } n \leq m \\ 0..m & \text{if } n > 2m \\ 0..(n-m) & \text{if } m < n \leq 2m \end{cases}$$

so again $d(Add_n[State_1], Add_\infty[State_1]) \to 0$, indeed the distance is zero after $n = 2m$.

However, if we calculate $d(Add_n[State_2], Add_\infty[State_2])$, this also comes to $1 - \#T/(m+1)$, but $T = \varnothing$ since for no input values can we guarantee that $x' < n$ for all state. And this sequence does not converge, correctly reflecting our intuition that we can never force Add_n to behave like Add_∞ no matter what input values are chosen.

A similar situation occurs when we apply this metric to the buffer example. In *Insert*, the value of the input chosen is immaterial thus $d(Buf_n, Buf) = 1$ since it is the size of the state that forces convergence or otherwise.

Arbitrary Input/Output Types. How do we generalise the above metric to arbitrary input/output types such as $add? : \mathbb{N}$? The approach we take is to avoid the problem and recognise the nature of approximation of implementation. The observation is that in any real implementation approximations will be made to data types such as $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ etc. For example, $\mathbb{N}$ will usually be implemented as $0..maxint$ and so forth, $\mathbb{R}$ as a certain precision of float.

As an example, consider our original addition example:

Add_n
$\Delta[x : \mathbb{N}]$
$add? : \mathbb{N}$

$x' = add? + 1 \land x' < n$

Add_∞
$\Delta[x : \mathbb{N}]$
$add? : \mathbb{N}$

$x' = add? + 1$

Add_n is a correct refinement of Add_∞ whenever $x' < n$. Now, the maximum range of a realistic implementation for $\mathbb{N}$ is $0..maxint$, so when n exceeds $maxint$, Add_n should correctly refine the implementation of Add_∞.

We thus use the same definition for d and ρ, and adapt the definition of ρ_a and ρ_c to take into account the implementation range. Then to adapt Definition 9 we take Y_{imp} (and resp. Z_{imp}) to be the actual implementation of Y (and resp. Z), and then find $\frac{(\#Y_{imp}-\#T)}{\#Y_{imp}}$ where T is the largest set $T \subseteq Y_{imp}$ such that

$$\forall x? : T \bullet \forall State \bullet \mathrm{pre}\, AOp \Rightarrow \mathrm{pre}\, COp$$

and similarly for ρ_c. (Notice we do not change the calculation of the precondition.)

With this metric should be a description of how each infinite type has been implemented, e.g., $\mathbb{N}$ as $0..maxint$.

Applying this to the example above gives us:

$$\begin{aligned} \mathrm{pre}\, Add_\infty &= true \\ \mathrm{pre}\, Add_n &= (add? + 1 < n) \\ &= add? : 0..n-2 \end{aligned}$$

So we find the largest T with $\forall add? : T \bullet \forall State \bullet add? : 0..n-2$, we then calculate $\frac{(\#Y_{imp}-\#T)}{\#Y_{imp}}$ which is

$$\begin{cases} \frac{(maxint+1-(n-1)}{maxint+1} & \text{if } n < maxint+2 \\ 0 & \text{otherwise} \end{cases}$$

which tends to zero as $n \to \infty$.

Discussion. Although this metric has the pleasing characteristic that it is defined via the simulation rules, and this is tractable, it is open to criticism in a number of respects. Firstly, it is somewhat ad hoc, and one must wonder whether a better characterisation can be obtained by beginning with the definition of downward simulation instead of the definition of data refinement. Second, the fudge to deal with unbounded data types is indeed a fudge. Whilst it deals satisfactorily with an infinite data type such as $\mathbb{N}$, it is less clear how effective it would be with $\mathbb{R}$ where the subset of values actually represented in a programming language varies more wildly depending on implementation strategy.

There are a number of ways these issues could be tackled. For example, one could embed the input/output in the state (in the standard fashion [6]) in order to derive the simulation rules from the data refinement definition. The metric could also take account of the complexity of constructing a particular input as its characterisation of how often this would occur (as opposed to a ratio as above). Such a complexity measure is akin to using a probability distribution, and this perhaps is the most promising avenue to explore. Instead of returning a ratio of failures to all possible values, we should construct a distance in terms of the probability of a particular input/output occurring that is a witness to the non-equivalence of A and C. This could then be combined with the approach to probability distribution discussed in Section 4.1. How a simulation based characterisation could be derived from this definition would be a challenging problem.

4.3 Open Questions

The purpose of this paper was to provide an initial articulation of the problem of approximate refinement, and sketch an approach based on metrics, chains and their limits. The use of metrics as a semantic basis in computer science is not new [4, 1, 11], however, the emphasis here has been on using the measure of distance and characterisation of limits rather than interest in the induced topological structure. In addition to the questions raised in the discussions above, there are a number of open questions which need addressing, including:

What is the Relationship of the Metrics Outlined in this Paper to Work on Metric Space Approaches to Semantics? Metric spaces used for denotational semantics have principally been used where concurrency is an issue, however, the metric defined in Section 4.1 is based upon that in [4], and this could be the starting point for such an exploration. It is less obvious what the relation is between the metric in Section 4.2 and those used as the basis for the semantics of concurrency.

What are the Topological Characteristics of the Metrics? Some will be inherited from their derivation, e.g., the metric used in [4]. For others, such as that in Section 4.2, an understanding of completeness, compactness and when they induce known topologies would be interesting.

What is their Basis in Terms of Data Refinement? The metric defined in Section 4.1 is based upon the definition of data refinement, how does this interact with the normal definition of simulation rules? On the one hand, the metric from Section 4.2 adapts the definition of operation refinement, rather than going back to the basic definition (Definition 1). This is rather unsatisfactory, and a better characterisation would derive the definition from that of data refinement. How should this be done, and how do these metrics relate to the finalisations (which determine what is visible). How do the results generalise to non-identity retrieve relations, and what topological properties do the retrieve relations induce?

What Alternative Metrics are there? Do the ones defined here capture all the intuitive properties of approximation? Which, if any, is the most attractive from a theoretical or practical viewpoint? How do notions of approximate refinement relate to work on implementing programming language data types such as [10]?

5 Metrics and Chains

The previous section has discussed possible metrics at some length. We now briefly discuss how they fit into the use of chains as an approach to development. Four kinds of development steps were proposed. The metric chosen will have a direct relevance to the limits in *introduce sequence* and *replace sequence*. It is also

worth noting that our discussion above has revealed there is no single canonical notion, thus the choice of metric is down to practical considerations: it depends on what aspects one considers important for a particular application. Similarly the choice of *compromise* depends on practical considerations, that is, how close an approximation is needed in particular circumstances.

In *element-wise refinement*, metrics and refinement are closely intertwined. What is needed is a way of refining each element in such a way that the refined sequence converges (hopefully to a refinement of the original limit).

A simple example shows that arbitrary refinement does not always have this property. Consider the following specifications together with the program length metric of Section 4.1.

Consider the sequence of ADTs consisting of an observer operation together with Inc_n, where each Inc_n is defined by

Inc_n

$$\Delta [x : \mathbb{N}]$$

$$x \leq x' \leq x + 2$$

This sequence is constant (its elements are independent of n), so it converges. However, the individual operation Inc_n is refined by the operation $EvenInc_n$, defined as

$EvenInc_n$

$$\Delta [x : \mathbb{N}]$$

$$\textbf{if } even(n) \textbf{ then } x' = x + 2 \textbf{ else } x' = x$$

but the sequence $(EvenInc_n)$ does not converge.

This proves the following (counter-)theorem:

Theorem 3. *Element-wise refinement does not guarantee sequence refinement, i.e., a refinement relation* $\sqsubseteq$ *and sequences* (S_n) *and* (T_n) *exist such that*

$$\forall\, n : \mathbb{N} \bullet S_n \sqsubseteq T_n$$

and (S_n) *converges, but* (T_n) *does not.*

This example shows that non-determinism cannot, in general, be resolved without losing uniformity. To preserve it one would have to ensure that the non-determinism was resolved in the same way in each element of the sequence. Further, with this metric, preconditions cannot be weakened, even if the weakening is the same in each element in the sequence.

Consider the following diagram, which shows the effect of one operation after another. There is no discernible difference, however, this does not hold if we weaken the precondition of the first operation (this is represented by the dotted line), and the distance has increased with this element-wise refinement.

In order to achieve element-wise refinement we clearly need the refinement to be uniformly convergent in the following (usual) sense. Letting $f(x)$ denote a refinement of x, we need

$$\forall \epsilon > 0 \bullet \exists \delta > 0 \bullet d(x, y) < \delta \Rightarrow d(f(x), f(y)) < \epsilon$$

which might be achieved, for example, in the following ways.

One possibility is to use the refinement calculus to refine each element in the sequence uniformly. This would involve not weakening the preconditions and resolving the non-determinism in a uniform way.

Another similar idea is to use calculational approaches. In particular, one can calculate the weakest downward or upward simulation of a specification (with respect to a retrieve relation which might change the state), and the result is equivalent to the original. Thus calculation could be applied element wise to a sequence, and convergence would be preserved (as required by element-wise refinement).

An alternative is to perform the refinement independently of n in such a way that element-wise refinement is obtained. Details of this are left for the future.

6 Related Work

The observation that idealised specifications correspond to "realistic" specifications with resource bounds tending to infinity is not new. In particular, in his PhD thesis [12], Neilson defined ∞-refinement $\sqsubseteq_\infty$ in terms of ordinary refinement $\sqsubseteq$ as follows:

$$A \sqsubseteq_\infty B$$
$$\Leftrightarrow$$
$$\exists\, c_1, c_2, \ldots, c_n : \mathit{ResourceLimit} \bullet \lim_{c_1, c_2, \ldots, c_n \to \infty, \infty, \ldots, \infty} (A \sqsubseteq B)$$

where the resource constraints c_i appear free in B and not at all in A. Such a refinement step establishes resource limits as constants in the specification; Neilson implicitly indicates that subsequent refinement may fix the values of these constants. Thus, these constants act as existential rather than universal parameters of the specification [3]: they may be arbitrarily constrained as long as the specification remains satisfiable. In our view, the use of underspecified constants obscures the distinction between the introduction of resource constraints, refinement, and the actual approximation that occurs by specialising their values. Another difference to our approach is that after the ∞-refinement step, there is no further mention (let alone development) of chains.

Banach and Poppleton have defined and investigated a generalisation of refinement called "retrenchment" [2]. They add "within" and "concedes" relations to every refinement step, indicating where preconditions are strengthened and postconditions weakened. These allow for developments which are not quite correctness preserving to be documented. However, this documentation refers to the internals of a specification at any given point in the development trace, and is thus hard to relate to external behaviour. Clearly, by taking a strong enough "within" relation, retrenchment holds between any pair of specifications – its value is in the documentation of where and how refinement has been relaxed. Taking that interpretation, our main objection to retrenchment as a development relation is that it encourages inexact development steps throughout, rather than localising them as we do here. Similar ideas are explored by Smith [13] for real-time specification – importantly, this work concentrates on the properties that are preserved by development steps which are not quite refinement steps but so-called "realisations".

Approximate refinement has been listed in a number of contexts as being desirable. In the UK Grand Challenge for Non-Classical Computation [15], it is mentioned as necessary for non-classical models such as quantum computation. Researchers at Berkeley [8] suggested its use for hybrid systems, although offered no means to do so.

7 Concluding Comments

This paper has set out an approach which we believe might be useful for combining formal program development in a disciplined way with the inevitable compromises required by the bounded resources of implemented programs. The underlying mathematics is very rich, and we have hardly begun to explore it, even in the context of relational specification languages – but we hope that the examples presented give a flavour of what might be possible, and some indication that a further exploration of these ideas would be worthwhile.

Acknowledgements. Thanks are due to members of the EPSRC RefineNet network (www.refinenet.org.uk), whose feedback on an earlier presentation substantially improved some of the above ideas, to Dan Grundy who commented on the draft, and to the reviewers for their useful suggestions.

References

1. P. America and J. Rutten. Solving reflexive domain equations in a category of complete metric spaces. In J. W. de Bakker and J. J. M. Rutten, editors, *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, pages 131–163. World Scientific, Singapore, 1992.
2. R. Banach and M. Poppleton. Retrenchment, refinement and simulation. In J.P. Bowen, S. King, S. Dunne, and A. Galloway, editors, *ZB 2000*, volume 1878 of *Lecture Notes in Computer Science*, pages 304–323. Springer-Verlag, 2000.

3. 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.
4. J. W. de Bakker and J.-J. C. Meyer. Metric semantics for concurrency. In J. W. de Bakker and J. J. M. Rutten, editors, *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, pages 104–130. World Scientific, Singapore, 1992.
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, 1998.
6. J. Derrick and E.A. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT. Springer Verlag, May 2001.
7. J. Derrick and E.A. Boiten. Relational concurrent refinement. *Formal Aspects of Computing*, 15(2):182–214, 2003.
8. A. Ghosal, M. Jurdzinski, R. Majumdar, and V. Prabhu. Approximate refinement for hybrid systems. Berkeley EECS Research Summary for 2003, http://buffy.eecs.berkeley.edu/ResearchSummary/03abstracts/vinayak.1.html.
9. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
10. B. Jacobs. Java's integral types in PVS. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *FMOODS'03*, pages 1–15, Paris, November 2003. Springer.
11. Marta Kwiatkowska and Gethin Norman. A fully abstract metric-space denotational semantics for reactive probabilistic processes. In Abbas Edalat, Achim Jung, Klaus Keimel, and Marta Kwiatkowska, editors, *Electronic Notes in Theoretical Computer Science*, volume 13. Elsevier, 2000.
12. D.S. Neilson. *From Z to C: Illustration of a Rigorous Development Method*. PhD thesis, Oxford University Computing Laboratory, 1990.
13. G. Smith. From ideal to realisable real-time specifications. In N. Leslie, editor, *Fifth New Zealand Formal Program Development Colloquium*, number 99-1 in IIMS Technical Report. Institute of Information and Mathematical Sciences, Massey University at Albany, 1999.
14. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.
15. Susan Stepney, John A. Clark, Colin G. Johnson, Derek Partridge, and Robert E. Smith. Artificial immune systems and the grand challenge for non-classical computation. In Jon Timmis, Peter Bentley, and Emma Hart, editors, *Proceedings of the 2003 International Conference on Artificial Immune Systems*, LNCS 2787, pages 204–216. Springer, September 2003.