The Timed Predicative Calculus as a Framework for Comparative Semantics

Jonathan Ostroff and Richard Paige*
Department of Computer Science, York University,
4700 Keele St., Toronto, ON M3J 1P3, Canada.

April 13, 2000

Abstract

Predicates are used in a variety of formal specification languages, though a predicate does not always mean the same thing in each approach. For example, the predicate *false* in Z means the same thing as *true* in the predicative calculus of Hehner.

In this paper, we compare the specification languages Z, Morgan's Refinement Calculus, and Parnas's Limited Domain relations, using the timed predicative calculus (TP) of Hehner as an underlying framework. In particular, we show that TP is more expressive than the other languages. We also show that refinement in TP is strictly weaker than the refinement relations associated with the other languages. As a result, refinement laws from the other languages can be reused in TP.

We conject that the simplicity of TP makes it a good candidate as a specification language and program development method for the refinement of procedural specifications, as well as real-time object oriented specifications.

*Email: {jonathan, paige}@cs.yorku.ca. Supported by a grant from NSERC.

Contents

1 Introduction **3**

2 The predicative calculus (UP and TP) **5**
2.1 The timed predicative calculus (TP) 7

3 Other programming calculi **9**
3.1 Morgan Refinement Calculus (MRC) 10
3.2 Z 13
3.3 Limited domain relations (LD) 15

4 Comparison **16**
4.1 Refinement comparison 16
4.2 Specification and refinement by parts 17
4.3 Expressiveness 20

5 Discussion and Conclusions **22**

List of Figures

1 Expressiveness of the various calculi 21

List of Tables

1 Refinement relations for $(S_1, P_1) \sqsubseteq (S_2, P_2)$ 16
2 Standard and extreme specifications in the calculi 22

1 Introduction

Predicates are used in a variety of formal specification methods though a predicate does not always mean the same thing in each approach. For example, the predicate *false* in Z means the same thing as *true* in the predicative calculus of Hehner (as explained in the sequel). The meaning of a predicate has a substantial impact on the complexity of a method's refinement rule and the expressiveness of its specification language.

Our original motivation for this paper is work we have recently done on a predicative refinement calculus for object-oriented software development [23, 24]. We would like the simplest possible *semantics* and *refinement relation* for such a calculus, while allowing for the calculus to be easily extended to real-time reactive systems. Since refinement is what we must prove at each step when we are programming, it is best to make refinement as simple as possible. It is also advantageous to make the semantics of such a language as simple as possible, if we desire to use supporting automated tools to abet the specification and refinement process, and if we want the language to be explainable and understandable.

The purpose of this paper is to briefly review the *timed predicative calculus* (TP) of Hehner [11] and to show that it is more expressive than other refinement calculi while having a strictly weaker (and hence easier to prove and simpler) refinement rule. We also demonstrate that specification and refinement by parts is simpler in TP than in other specification and refinement calculi. Thus, a key conclusion of this paper is that TP is an appropriate basis to use for the formal development of procedural software, and we also conject that it is appropriate to develop real-time, object-oriented software as well.

The paper is organised as follows:

- We describe how TP is built from the untimed predicative calculus (UP) of Hehner [11] by adding to it a time variable $t \in \mathbb{R} \cup \{\infty\}$. The notion of an execution, a specification, and an execution satisfying a specification can all be expressed as predicates in TP.

- We show how specifications in Z [27], Morgan's refinement calculus (MRC) [22], and Parnas's Limited Domain relations (LD) can be translated to TP without loss of expressiveness. Thus, TP can be used as an underlying framework for expressing the semantics of the other calculi, and for comparing the calculi.

- We show that the TP refinement relation is strictly weaker (and hence easier to prove) than simple refinement (as used in Eiffel [21]) and Z refinement. LD refinement has the same form as that of Z. MRC refinement is defined in terms of weakest preconditions. We prove that MRC refinement also has the same form as that of Z and LD (Theorem 1); to our knowledge, such a proof has not previously appeared in the literature. Thus, we show that TP refinement is strictly weaker than the others. A consequence of the simpler TP notions of specification and refinement is that specification and refinement by parts is simpler in TP.

- We show that TP is more expressive than other calculi (Theorem 2). In particular, we use extreme specifications to illustrate the differences between the calculi (Table 2).

- We discuss our work in relation to other comparisons in the literature.

Notation

To avoid unnecessary brackets, we apply the following operator precedence from highest (level 0) to lowest precedence (level 8):

0. unary prefix operators: $+$, $-$, $\neg$, pre
1. $*$, $/$
2. $+$, $-$, $\cap$, $\cup$ etc.
3. relations: $=$, $\neq$, $<$, $\leq$ etc. and $\in$
4. logical operators: $\wedge$, $\vee$
5. $\rightarrow$, $\leftarrow$, $\overset{t,t'}{\rightarrow}$, $\sqsubseteq$, $\sqsubseteq_s$, $\sqsubseteq_m$
6. $\equiv$, $\not\equiv$
7. assignment: $:=$, and quantification: $\forall$, $\exists$
8. definition: $\widehat{=}$

We use $\mathtt{x}, \mathtt{y}, \ldots$ for program variables. We let $\sigma \mathrel{\widehat{=}} x, y, \cdots$ where $x, y, \ldots$ represent the initial state of the program variables respectively, and we let $\sigma' \mathrel{\widehat{=}} x', y', \ldots$ where $x', y', \ldots$ represent the final state of the program variables respectively. P, P_1, P_2, Q, Q_1, Q_2 are predicates (representing specifications) with free variables in σ and σ'. We use S, S_1, S_2 for single state predicates, and D, D_1, D_2 for double state predicates (defined in the sequel).

2 The predicative calculus (UP and TP)

The untimed predicative calculus (UP) was introduced in [11] as a specification and refinement calculus for sequential, parallel, interactive, or functional programs. It is the precursor to TP. In UP, a single predicate such as

$$x' < x \wedge y' = y \tag{1}$$

is used to specify a required program. The predicate asserts that program variable x should be decreased while keeping y the same (notation will be described below). Assuming that $x, x', y, y' : \mathbb{Z}$, the specification is satisfied by many different programs including program `x := x - 1`, program `x := x - 2`, program `x := x - 3` etc. If we add a precondition to the specification so that the specification now reads $x > 0 \rightarrow x' < x \wedge y' = y$, we thereby enlarge the number of satisfying programs, e.g. `if x >= 0 then x := x - 1 else x := 100`.

In order to define the relationship between programs, their executions, and specifications we proceed as follows:

- We first identify the quantities of interest (e.g. memory locations in the computer, termination, etc.) and introduce a variable for each quantity of interest. This allows us to describe program execution (behaviour) in terms of the variables.

- An *observation* is a predicate that precisely describes the pre-state and post-state of a program. For example, the predicate

  $$x = 4 \wedge y = 4 \wedge x' = 3 \wedge y' = 4 \tag{2}$$

  is an observation where the program variables x, y denote the value of the memory locations in the pre-state, and x', y' denote their values in the post-state. An observation ignores all the intermediate states of an execution and specifies only the initial and final states.

- A *specification* is a set of observations that describe the required behaviour of the program. We use predicates in the program variables that relate the pre-state to the post-state for specifications. Thus, the predicate

  $$(x = 4 = y) \wedge (1 \leq x - x' \leq 2) \wedge (y' = y)$$

is a specification that corresponds to two observations viz. $x = 4 \wedge y = 4 \wedge x' = 3 \wedge y' = 4$ and $x = 4 \wedge y = 4 \wedge x' = 2 \wedge y' = 4$, given that $x, y : \mathbb{Z}$.

- A *program* is a specification that has been implemented. We include programs in our specification language as follows

$$\begin{array}{rcl}
\texttt{skip} & \widehat{=} & x' = x \wedge y' = y \wedge \ldots \\
\texttt{x := e} & \widehat{=} & x' = e \wedge y' = y \wedge \ldots \\
\texttt{if b then P else Q} & \widehat{=} & (b \rightarrow P) \wedge (\neg\, b \rightarrow Q) \\
\texttt{P; Q} & \widehat{=} & \exists\, \sigma'' \bullet P[\sigma' := \sigma''] \wedge Q[\sigma := \sigma'']
\end{array}$$

  P and Q can themselves be specifications or programs; hence, we mix programs and specifications because they are all described by predicates. A full range of program constructs is defined in [11].

- From the program definitions many useful laws of programming can be derived. For example, a simple proof using the definitions justifies

$$\texttt{if b then P else P} \quad \equiv \quad \texttt{P}$$

- If we have a particular observation *Obs* of program execution and we want to know if it *satisfies* a specification *Spec*, then we must show that

$$Obs \rightarrow Spec \tag{3}$$

  is a theorem. The above definition of satisfaction provides a criterion for distinguishing between observations that satisfy the specification and those that do not. An observation of the states of an executing program can never be equivalent to *false*, so the satisfaction criterion will never be vacuous.

- A specification $Spec1$ is refined by a specification $Spec2$ (written $Spec1 \sqsubseteq Spec2$) if all the observations represented by $Spec2$ are also observations of $Spec1$. This leads to a very simple definition of refinement

$$Spec1 \sqsubseteq Spec2 \;\; \widehat{=} \;\; (\forall\, \sigma, \sigma' \bullet Spec2 \rightarrow Spec1) \tag{4}$$

The idea behind refinement is that we gradually transform an abstract program (possibly a specification) into a concrete implementation. If the initial abstract program is correct, and the transformation steps preserve correctness, then the resulting steps will be correct by construction. An abstract program is usually easier to prove correct – i.e., that it satisfies its requirements – than an implementation. Our definition of refinement above captures the essential notion that we want, viz., no client of the more concrete program can observe that it is not using the more abstract version of the program. The notion of abstractness is a relative term, i.e. the direction of refinement is from a higher level (the specification on the left of $\sqsubseteq$) to a lower level (the one on the right); the higher level is called the abstract specification.

In UP, we do not need to distinguish between the specification language and its semantics; a single predicate fulfills the role of both. All the fundamental definitions and notions such as observations (behaviour), specifications, programs, satisfaction (3), and refinement (4) are defined in terms of predicates with free variables in σ and σ'.

2.1 The timed predicative calculus (TP)

UP only deals with partial correctness. This is because we have not introduced any variables for the main quantity of interest in total correctness — termination. The UP specification

$$x > 0 \;\rightarrow\; x' < x \wedge y' = y \tag{5}$$

simply means that if a computation starts in a state satisfying $x > 0$ then the terminating state, if there is one, will satisfy $x' < x \wedge y' = y$. If x does not have a positive value in the prestate then any behaviour is allowed. Since there is no mention of time t in the specification, nothing is asserted about the time taken by the computation.

To obtain a timed predicative calculus (TP) we just add a time variable t of type $\mathbb{R} \cup \{\infty\}$. We use t for the time at which execution starts, and t' for the time at which execution ends. To allow for non-termination, we allow the domain of time to be a number system that includes ∞. Axioms such as $\infty + 1 = 1$ and $-\infty < x < \infty \rightarrow x * 0 = 0$ are needed to deal with infinity — see [11] for the complete list. No changes to the UP notions of satisfaction and refinement are needed. We do need to make the following two stipulations:

1. $\sigma = t, x, y, \ldots$, i.e. t must be added to the state space (e.g. when defining sequential composition we need to existentially quantify over a state space that includes t). t may appear in the text of a program (e.g. to refer to the current time), but any changes that are made to t in the program must be according to a coherent timing policy. Two such policies — real-time and recursive time — are described in [11].

2. A specification can be any predicate with free variables in σ and σ' including the predicate *false*. However, not every specification is implementable. We say that a specification *Spec* is *implementable* if

$$(\forall \sigma \, \exists \sigma' \bullet \mathit{Spec} \wedge t' \geq t) \tag{6}$$

   The definition asserts two important facts. First, *every* prestate must have at least one corresponding well-defined post-state. This means that *false* cannot be implementable, but it also means that $x \geq 0 \wedge y' = 0$ is not implementable. If the initial value of x is non-negative, then the specification can be satisfied by setting y to zero. But if the initial value of x is negative, there is no way to set y in such a way as to satisfy the specification. Perhaps the specifier has no intention of providing a negative input, but to the implementor, every input is a possibility. The specifier should have written

$$x \geq 0 \rightarrow y' = 0$$

   which means that for negative input the implementor is free to do anything. He can provide an error message or just let the program crash.

   The second important assertion in the definition of implementability is that time cannot decrease between the input and the output. This makes the specification $Q : x' = 2 \wedge t' < \infty$ unimplementable, because $Q \wedge t' \geq t$ is unsatisfiable for an initial value of $t = \infty$. It may seem strange to reject the specification just because it won't work at time $t = \infty$; after all, no actual implementation will start at an infinite time in the future. But consider the sequential composition `infiniteloop` ; Q. In this case Q starts at infinite time (i.e. it never starts). The theory has to cover this runaway case as well. Thus Q must be checked for a non-decrease in time at infinity as well.

The following example illustrates how we equip specifications with termination.

$$x > 0 \wedge t = 0 \rightarrow x' < x \wedge y' = y \wedge t' = 6 \tag{7}$$

Any observation in which initially $t = 0$ with $x > 0$ will terminate in exactly 6 ticks. Any other observation can behave arbitrarily, including never terminating.

In Hoare [18], termination is described by the introduction of a new variable *ok* which is true of a program that has started in a fully defined state, and ok' which is true of a program that has stopped in a fully defined state. To indicate termination, specifications are written: $pre \wedge ok \rightarrow post \wedge ok'$. This allows for termination but does not describe real-time.

Hehner argues that total correctness, without explicit time, is wasteful [12]. In a total correctness formalism, one does a lot of work with variants or least fixpoints (including an indirect calculation of a time bound which is later thrown away) to gain one bit of information (termination ok') which is of dubious value. This is because termination without a specified bound is unobservable. How will you know that termination has happened? If nothing has happened for 5 minutes can you report termination? No; perhaps the machine is computing $Ackermann(6, 6)$ and still has a century to go. In this paper we will not hesitate to write total correctness or liveness properties such as

$$S \rightarrow D \wedge t' \geq t \wedge t' \neq \infty \tag{8}$$

which asserts that there will be termination at some unspecified future time. The above predicate is equivalent to

$$S \wedge t \neq \infty \rightarrow D \wedge t' \geq t \wedge t' \neq \infty$$

because $t' \geq t \wedge t' \neq \infty \equiv t' \geq t \wedge t' \neq \infty \wedge t \neq \infty$. The abbreviation $S \stackrel{t,t'}{\rightarrow} D$ defined by

$$S \stackrel{t,t'}{\rightarrow} D \;\;\widehat{=}\;\; S \rightarrow D \wedge t' \geq t \wedge t' \neq \infty \tag{9}$$

will be used in the sequel to describe timed behaviour in TP.

3 Other programming calculi

The well-known Hoare calculus [17] uses the triple $\{S\}PROG\{D\}$ to assert that: "an execution of $PROG$ begun in a prestate satisfying the predicate S

must terminate in a post-state satisfying the predicate D". As an example, consider program $PROG1$ described by

$\{S_1 : x > 0\}$
```
x := x - 1
y := y
```
$\{D_1 : x' < x \wedge y' = y\}$

Postcondition D_1 is a *double-state* predicate because it has occurrences of initial and final variables. Precondition S_1 is a *single-state* predicate because it only has initial variables.

Suppose program $PROG2$ has precondition S_2 and postcondition D_2, and let $PROG1 \sqsubseteq PROG2$ denote that $PROG1$ is refined by $PROG2$. A simple minded definition of refinement is

$$PROG1 \sqsubseteq_s PROG2 \quad \widehat{=} \quad (\forall \sigma, \sigma' \bullet (S_1 \rightarrow S_2) \wedge (D_2 \rightarrow D_1)) \qquad (10)$$

i.e. in $PROG2$ we may weaken the precondition and/or strengthen the postcondition of $PROG1$. This is the method used for *redefining* the contract of an inherited feature in the Eiffel programming language [21]. In Eiffel, classes possess procedures and functions (also called features), which can be given contracts (pre- and postconditions). A contract of a feature can be changed when the feature is inherited. Particularly, the inherited precondition can be weakened and the postcondition can be strengthened. Such a requirement permits refinement of specifications and implementations in a class, without changing a client's view of the class[1].

3.1 Morgan Refinement Calculus (MRC)

The above Hoare triple mixes the implementation (programs) with specifications. It is useful to be able to deal with pure specifications, as well as code, in refinement, so that specifications can be reasoned about, and programs can be developed in a piece-by-piece and step-by-step manner. Morgan [22] introduced the notation $x : \llparenthesis\, S,\ D \,\rrparenthesis$ to denote specifications[2], where x is the

[1]Eiffel uses a syntactic convention to ensure that if a contract is changed when it is inherited, it is guaranteed to be a refinement of the original. This eliminates the need for theorems to be proved at compile-time.

[2]Morgan's syntax is actually $x : [S, D]$, but we use the square brackets for substitution, e.g. $P[x := e]$ means replace all free occurrences of x in P by e. Also, we use primed vari-

frame (the list of variables that may change between the pre-state and post-state), S is the precondition, and D is the postcondition. A specification $x : \llparenthesis\, S,\ D \,\rrparenthesis$ may be combined with other specifications or programs, using program combinators like sequencing. $PROG1$ is thus specified in MRC by

$$x : \llparenthesis\, x > 0,\ x' < x \,\rrparenthesis$$

The predicate $y' = y$ is not needed in the postcondition as the frame takes care of this constraint. The specification $x : \llparenthesis\, S,\ D \,\rrparenthesis$ thus means "if the prestate satisfies the precondition S then change only the variables listed in the frame so that there is a terminating poststate that satisfies D".

The quantities of interest in the above definition are the prestate of the memory (S), the poststate of the memory (D), and termination. No statement of termination appears explicitly in the specification statement $x : \llparenthesis\, S,\ D \,\rrparenthesis$, so the formal semantics will have to somehow take it into account.

The semantics of MRC specifications is given using the weakest precondition calculus [5] which can be used to justify the consistency of the transformation laws used in the process of refinement. The semantics of the specification statement is given by [22][3]

$$wp(x : \llparenthesis\, S,\ D \,\rrparenthesis,\ P) \ \ \widehat{=}\ \ S \wedge (\forall\, x' \bullet D \wedge y' = y \ \rightarrow\ P) \tag{11}$$

given that $\sigma = x, y$. The meaning of refinement is justified by

$$P_1 \sqsubseteq_m P_2 \ \ \widehat{=}\ \ (\forall\, \sigma, \sigma', P \bullet wp(P_1, P) \rightarrow wp(P_2, P)) \tag{12}$$

where $\sqsubseteq_m$ is the Morgan refinement relation. This seems to be the only reasonable way to define refinement using weakest preconditions.

We claimed in the introduction that TP can be used as an underlying foundation for the various calculi. We can transform MRC specifications to TP as follows:

$$x : \llparenthesis\, S,\ D \,\rrparenthesis \ \ \widehat{=}\ \ S \overset{t,t'}{\rightarrow} D \wedge y' = y \tag{13}$$

This TP semantics explicitly mentions timing and is much simpler than the corresponding wp-semantics (no quantifiers are needed). Furthermore, there

ables for poststates and unprimed variables for prestates, whereas Morgan uses unprimed variables for poststates.

[3]The formulation in Morgan is more complicated, and initial variables are not allowed to occur in the predicate P. We do not see the need for this restriction.

is no need to separate the specification language from its semantics: all specifications and programs are, as argued earlier, given in terms of predicates.

The semantics of Morgan refinement can also be expressed more easily in terms of a predicate in TP (the proof format and justifications follow the approach used in [7]).

Theorem 1 (MRC refinement in TP) *Without loss of generality, let* $\sigma = x, y$. *Then*

$$x : \llparenthesis\, S_1,\ D_1 \rrparenthesis \ \sqsubseteq_m \ x : \llparenthesis\, S_2,\ D_2 \rrparenthesis \quad \equiv \quad (\forall\, \sigma, \sigma' \bullet S_1 \;\rightarrow\; S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1))$$

where $\tilde{D}_1 \mathrel{\widehat{=}} (D_1 \wedge y' = y)$ *and* $\tilde{D}_2 \mathrel{\widehat{=}} (D_2 \wedge y' = y)$.

Proof (sufficiency)

$x : \llparenthesis\, S_1,\ D_1 \rrparenthesis \ \sqsubseteq_m \ x : \llparenthesis\, S_2,\ D_2 \rrparenthesis$

$=$ ⟨ definition of $\sqsubseteq_m$ (12) ⟩

$(\forall\, \sigma, \sigma', P \bullet wp(x : \llparenthesis\, S_1,\ D_1 \rrparenthesis, P) \rightarrow wp(x : \llparenthesis\, S_2,\ D_2 \rrparenthesis, P))$

$=$ ⟨ definition of wp (11) ⟩

$(\forall\, \sigma, \sigma', P \bullet S_1 \wedge (\forall\, x' \bullet \tilde{D}_1 \rightarrow P) \rightarrow S_2 \wedge (\forall\, x' \bullet \tilde{D}_2 \rightarrow P))$

$\Rightarrow$ ⟨ instantiation (9.13) with $P := \tilde{D}_1$ ⟩

$(\forall\, \sigma, \sigma' \bullet S_1 \wedge (\forall\, x' \bullet \tilde{D}_1 \rightarrow \tilde{D}_1) \rightarrow S_2 \wedge (\forall\, x' \bullet \tilde{D}_2 \rightarrow \tilde{D}_1))$

$=$ ⟨ reflexivity of $\rightarrow$ (3.71); identity of $\wedge$ (3.39) ⟩

$(\forall\, \sigma, \sigma' \bullet S_1 \rightarrow S_2 \wedge (\forall\, x' \bullet \tilde{D}_2 \rightarrow \tilde{D}_1))$

$=$ ⟨ $\neg\, occurs(`x'', S_1, S_2)$; distributivity of $\forall$ over $\rightarrow$ as in (9.5) ⟩

$(\forall\, \sigma, \sigma', x' \bullet S_1 \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1))$

$=$ ⟨ σ' already contains x' by assumption ⟩

$(\forall\, \sigma, \sigma' \bullet S_1 \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1))$

(necessity)

$(\forall\, \sigma, \sigma' \bullet S_1 \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1))$

$=$ ⟨ $(\forall\, P \bullet pred) \equiv pred$ by (9.5) if P is a fresh dummy ⟩

$(\forall\, \sigma, \sigma', P \bullet S_1 \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1))$

$\Rightarrow$ ⟨ prop. logic $(p \rightarrow q) \rightarrow (p \wedge r \rightarrow q \wedge r)$ and monotonicity ⟩

$(\forall\, \sigma, \sigma', P \bullet S_1 \wedge (\tilde{D}_1 \rightarrow P) \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow \tilde{D}_1) \wedge (\tilde{D}_1 \rightarrow P))$

$\Rightarrow$ ⟨ transitivity of $\rightarrow$ (3.82a) and MON ⟩

$(\forall\, \sigma, \sigma', P \bullet S_1 \wedge (\tilde{D}_1 \rightarrow P) \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow P))$

$=$ ⟨ by assumption $\sigma' = x', y'$ ⟩

$(\forall\, \sigma, \sigma', P, x' \bullet S_1 \wedge (\tilde{D}_1 \rightarrow P) \rightarrow S_2 \wedge (\tilde{D}_2 \rightarrow P))$

$$
\begin{array}{ll}
\Rightarrow & \langle \text{ monotonicity of } \forall \text{ (9.12); distributivity of } \forall \text{ over } \wedge \text{ (8.15) } \rangle \\
& (\forall\, \sigma, \sigma', P \bullet S_1 \wedge (\forall\, x' \bullet \tilde{D}_1 \rightarrow P) \rightarrow S_2 \wedge (\forall\, x' \bullet \tilde{D}_2 \rightarrow P)) \\
= & \langle \text{ definition of } wp \text{ (11) } \rangle \\
& (\forall\, \sigma, \sigma', P \bullet wp(x : \llparenthesis S_1,\ D_1 \rrparenthesis, P) \rightarrow wp(x : \llparenthesis S_2,\ D_2 \rrparenthesis, P)) \\
= & \langle \text{ definition of } \sqsubseteq_m \text{ (12) } \rangle \\
& x : \llparenthesis S_1,\ D_1 \rrparenthesis \quad \sqsubseteq_m \quad x : \llparenthesis S_2,\ D_2 \rrparenthesis \qquad \blacksquare
\end{array}
$$

3.2 Z

Z is based on typed set theory, and provides a structuring mechanism: the schema. The schema is used to introduce a named collection of variables and provides a predicate to show how the variables are related. $PROG1$ from Section 3 may be specified by

Decrease
$x, x', y, y' : \mathbb{N}$
$x > 0$ $x' < x \wedge y' = y$

The predicate P corresponding to the schema *Decrease* is just the conjunction of the type declarations, the precondition, and the postcondition, i.e. $P \mathrel{\widehat{=}} (x, x', y, y' : \mathbb{N}) \wedge x > 0 \wedge x' < x \wedge y' = y$. In general, the precondition of a Z schema need not be explicitly given in the predicate part; it may have to be calculated.

Z also provides a *schema calculus* that can be used to combine schemas. Operators like schema conjunction, schema disjunction, and schema composition have been defined that allow large specifications to be constructed by parts. The schema calculus operators all combine the declarations and predicates of the schema operands.

Strangely, the interpretation of Z predicates as specifications of behaviour is just a convention and we are free to make our own convention if we like. That being said, what are the usual conventions for these matters? According to Spivey [27] on page 129: "if the (initial) state is related to at least one possible state after the operation, then the operation must terminate successfully ... If the predicate relates the state before the operation to no possible state afterwards, then nothing is guaranteed ... the operation may

fail to terminate, may terminate abnormally, or may terminate successfully in any state at all." This convention, in essence, is the same as MRC, except that we will need to extract the precondition associated with the predicate in order to talk about the initial states.

The precondition (pre P) of a Z predicate P is defined as follows[4]: $\mathrm{pre}\, P \mathrel{\widehat=} (\exists\, \sigma' \bullet P)$. The semantics of the Z schema in TP can then be given as

$$(\mathrm{pre}\, P) \;\stackrel{t,t'}{\rightarrow}\; P \tag{14}$$

Given two schemas with predicates P_1 and P_2, respectively, the Z refinement rule is [30]:

$$P_1 \sqsubseteq_z P_2 \;\;\mathrel{\widehat=}\;\; (\forall\, \sigma, \sigma' \bullet \mathrm{pre}\, P_1 \rightarrow \mathrm{pre}\, P_2 \wedge (P_2 \rightarrow P_1)) \tag{15}$$

Z refinement is equivalent to MRC refinement, once we split the Z predicate into two predicates (the precondition pre P and the original predicate P).

We can now begin to contrast Z and MRC refinement with simple refinement (a more thorough comparison is presented in Section 4). Consider the following example. The schema

DownOne

$x, x', y, y' : \mathbb{N}$

$x' = x - 1 \wedge y' = y$

refines the schema *Decrease* whether we use the simple refinement rule (10), or the Z refinement rule (15). The schema

DownInt

$x, x', y, y' : \mathbb{Z}$

$x' = x - 1 \wedge y' = y$

does refine *Decrease* under Z refinement. However, the Z version of simple refinement (10) which is

$$(\mathrm{pre}\, \mathit{Decrease} \rightarrow \mathrm{pre}\, \mathit{DownInt}) \wedge (\mathit{DownInt} \rightarrow \mathit{Decrease})$$

[4] If the schema also has an included state schema with a corresponding predicate *inv*, then the precondition is defined as $(\exists\, \sigma' \mid \mathit{inv} \bullet P)$

does not hold, as we cannot prove $DownInt \rightarrow Decrease$, i.e. $x' = x - 1 \wedge y' = y \rightarrow x > 0 \wedge x' < x \wedge y' = y$ is not a theorem, as can be seen by using the substitutions $x := 0, x' := -1, y := 2, y' := 2$.[5] We seek the weakest possible refinement rule that preserves the essential notions of refinement, which is that no client of the more concrete program can observe that it is not using the more abstract version of the program that it refines. By that criterion, MRC/Z refinement is superior to simple refinement. We can now compare Z and MRC.

- MRC specifications contain two predicates. Z specifications consist of a single predicate.
- Z specifications also have a precondition like MRC, but the precondition is fully determined by the Z predicate. This means that Z is less expressive than MRC. For example, by (14) we see that Z cannot be used to express miracles (i.e. the TP predicate *false*).
- The Z refinement rule (15) has the same form as the MRC refinement rule by Theorem 1.
- Z does not have a frame to facilitate writing postconditions that need to assert that a set of variables do not change. However, the Ξ convention can be used to achieve the same thing.

3.3 Limited domain relations (LD)

A Limited Domain (LD) relation is a pair (R, C) where R is a binary relation and C (the competence set) is a subset of the domain of R. LD relations are described using predicate logic, i.e. an LD relation can be described by the characteristic predicate of the relation, the domain and a competence set. A logic for partial functions [25] is used to ensure that specifications are well-defined in every state.

The semantics of LD relations is different from MRC and Z. We may use a pair of predicates (S, D) to represent the characteristic predicates of the competence set and relation respectively. The meaning of such an LD

[5] In the case of $DecreaseOne$, the type information viz., $x, x' : \mathbb{N}$ can be used to guarantee $x > 0$, i.e. $(x, x' : \mathbb{N} \wedge x' = x - 1 \wedge y' = y) \rightarrow (x, x' : \mathbb{N} \wedge x > 0 \wedge x' < x \wedge y' = y)$. We thank John Wordsworth for pointing out these simple but illustrative counter-examples.

Simple Refinement $\sqsubseteq_s$	$(\forall \sigma, \sigma' \bullet (S_1 \to S_2) \wedge (D_2 \to D_1))$
Standard refinement $\sqsubseteq_m$ (used by MRC, Z and LD)	$(\forall \sigma, \sigma' \bullet S_1 \to (S_2 \wedge (D_2 \to D_1)))$
TP refinement $\sqsubseteq$	$(\forall \sigma, \sigma' \bullet (S_2 \to D_2) \to (S_1 \to D_1))$

Table 1: Refinement relations for $(S_1, P_1) \sqsubseteq (S_2, P_2)$

description is as follows: (a) if execution terminates in state σ', starting from some initial state σ, then (σ, σ') must satisfy D, and (b) if the program starts in a state satisfying S, then it must terminate. We can rephrase this as follows:

- if execution starts in initial state σ satisfying S, then it must terminate in a state σ' so that (σ, σ') satisfies D.
- if execution starts in a state σ not satisfying S, then it will either fail to terminate or it will terminate in a state σ' so that (σ, σ') satisfies D.

The semantics of $(S, D$ can thus be translated to TP as follows

$$(S \overset{t,t'}{\to} D) \wedge (\neg S \to (D \wedge t' \geq t \wedge t' \neq \infty) \vee t' = \infty) \qquad (16)$$

The refinement relation is the same as for MRC.

4 Comparison

4.1 Refinement comparison

The simplest way to compare refinement rules for the various calculi is, without loss of generality, to use a pair of predicates (S, D), where S is a precondition and D a postcondition, to represent specifications in various the calculi. MRC and LD are already based on a pair of predicates, and the single predicate formalism Z (with predicate P) can be translated into the pair $(\mathrm{pre}\, P, P)$, using the precondition operator provided in the Z toolkit and in (14). The three refinement rules of the specification languages are expressed as predicates in TP in Table 1.

We now classify the refinement relations in terms of their complexity. In particular, we provide a hierarchy of refinement relations, based on the

expression of the relations in TP. Before classifying the relations, we define the notion of *strictly weaker* predicates.

Definition 1 (Strictly weaker) *A predicate P_2 is* strictly weaker *than a predicate P_1 if (a) $P_1 \to P_2$ is a theorem, and (b) $P_2 \to P_1$ is not a theorem.*

The notion of one predicate being strictly weaker than another can be used to produce a refinement hierarchy. This hierarchy will be useful in classifying the refinement relationships, so that users of refinement can best select the most appropriate refinement relationship for their work. Given the TP expressions of the refinement relations, we can prove the following.

Theorem 2 (Refinement Hierarchy) *(i) Standard Refinement is strictly weaker than Simple Refinement. (ii) TP refinement is strictly weaker than Standard Refinement.*

Proof: (i) Proof of (a) in Definition (1) is by simple predicate calculus. For the proof of (b) in Definition (1) just use the counter-example $D_2 := \mathit{true}, S_1 := \mathit{false}, D_1 := \mathit{false}$. (Note that we already saw in Section 3.2 that *DownInt* refines *Decrease* under standard refinement but not under simple refinement.)

(ii) Proof of (a) in Definition (1) is again by predicate calculus. For the proof of (b) in Definition (1) use the counter-example $D1 := \mathit{true}, S_1 := \mathit{true}, S_2 := \mathit{false}$. ■

Corollary (a) The TP refinement rule is weaker (and hence, is easier to prove) than MRC, Z, LD, and simple refinement. (b) We can use all the Z and MRC refinement rules (appropriately translated) for refinement in TP.

4.2 Specification and refinement by parts

It is often convenient to construct specifications by parts. Suppose we have an array a where elements are numeric and thus comparable. We may want to write a sort specification as follows.

$$\mathit{sort}(a', a) \quad \widehat{=} \quad \mathit{permutation}(a', a) \wedge \mathit{ordered}(a')$$

It is convenient to use ordinary conjunction $P_1 \wedge P_2$ when you want behaviour that simultaneously satisfies specifications P_1 and P_2. We can easily

show from the definition of TP refinement that the rule

$$\frac{P_1 \sqsubseteq P_2, \quad Q_1 \sqsubseteq Q_2}{P_1 \wedge Q_1 \sqsubseteq P_2 \wedge Q_2} \quad [\text{ TPRP – TP refinement by parts }]$$

is valid. Thus, we can do refinement by parts in the expected fashion.

There is no standard definition for specification by parts in MRC, in part because MRC only provides program (and not specification) combinators. It is also not immediately obvious how to put $x : \langle\!|\, S_1,\ D_1 \,|\!\rangle$ and $y : \langle\!|\, S_2,\ D_2 \,|\!\rangle$ together within the framework of MRC.

The definition of specification by parts in [29] is

$$x, y : \langle\!|\, S_1 \wedge S_2 \wedge (\exists\, x', y' \bullet S_1 \wedge S_3),\ D_1 \wedge D_2 \,|\!\rangle$$

The problem with this definition is that MRC refinement ($\sqsubseteq_m$) is not monotonic with respect to specification by parts (i.e. the MRC equivalent to rule TPRP above is not valid). However, using the translation of MRC specifications to TP and (17), below,

$$(S_1 \overset{t,t'}{\to} D_1) \wedge (S_2 \overset{t,t'}{\to} D_2) \ \equiv\ S_1 \vee S_2 \overset{t,t'}{\to} (S_1 \to D_1) \wedge (S_2 \to D_2) \qquad (17)$$

(which can be shown to be true by a simple argument), we get the following definition for MRC specification by parts

$$\begin{array}{ll} & x : \langle\!|\, S_1,\ D_1 \,|\!\rangle \wedge y : \langle\!|\, S_2,\ D_2 \,|\!\rangle \\ \widehat{=} & x, y : \langle\!|\, S_1 \vee S_2,\ (S_1 \to D_1) \wedge (S_2 \to D_2) \,|\!\rangle \end{array}$$

TP has given us the right definition because now MRC refinement by parts is monotonic in $\sqsubseteq_m$, i.e.

$$\frac{P_1 \sqsubseteq_m P_2, \quad Q_1 \sqsubseteq_m Q_2}{P_1 \wedge Q_1 \ \sqsubseteq_m\ P_2 \wedge Q_2} \quad [\text{ MRCRP – MRC refinement by parts }]$$

is valid, where P_1, P_2, Q_1, Q_2 are MRC specifications.

MRC refinement by parts is more complex than the corresponding TP rule, but at least it works in the expected fashion using the right definition of specification by parts. But things do not work so well in Z.

In Z, two schemas (with predicates P_1 and Q_1 respectively) with type compatible signatures can be combined with the schema conjunction operator

to give a new schema with corresponding predicate $P_1 \wedge Q_1$ and signature the result of joining the two component signatures (see [27], p32). The Z refinement rule (15) has the same basic form as the MRC refinement rule, and hence we would like to use MRCRP above for Z refinement by parts. However, using the obvious choice of schema conjunction for specification by parts means that MRCRP is not valid for Z, as the following counterexample shows.

schema with predicate P_1

$$x, x', y, y' : \mathbb{Z}$$

$$x' = y' + 3$$

schema with predicate Q_1

$$x, x', y, y' : \mathbb{Z}$$

$$y' = 7$$

where we want to refine P_1 by P_2 and Q_1 by Q_2 with P_2 and Q_2 defined by

schema with predicate P_2

$$x, x', y, y' : \mathbb{Z}$$

$$x' = 10 \wedge y' = 7$$

schema with predicate Q_2

$$x, x', y, y' : \mathbb{Z}$$

$$y' = 7 \wedge x' = 9$$

For the above examples, the Z preconditions of P_1, P_2, Q_1, Q_2 and $P_1 \wedge Q_1$ are all *true*; however, $\mathrm{pre}(P_2 \wedge Q_2)$ is *false*. Thus, $P_1 \sqsubseteq_m P_2$ and $Q_1 \sqsubseteq_m Q_2$ are *true*, but $P_1 \wedge Q_1 \ \sqsubseteq_m \ P_2 \wedge Q_2$ evaluates to

$$\mathit{true} \rightarrow (\mathit{false} \wedge (P_2 \wedge Q_2 \rightarrow P_1 \wedge Q_1))$$

which is *false*. In TP, the "miracle" *false* (although not implementable) refines every program. However, in Z, *false* means "chaos", i.e. arbitrary possibly non-terminating behaviours (Table 2). Thus, $P_2 \wedge Q_2$ (which is false) cannot refine $P_1 \wedge Q_1$.

Because of the unusual meaning that Z gives to *false*, the definition in [27] of schema conjunction does not work in the intuitive fashion. Consider two schemas, one with predicate `skip` and the other with predicate *false*. According to schema conjunction [27], the predicate of the combined schema is `skip` $\wedge$ *false* which evaluates to *false*, i.e. chaos. This is clearly wrong because the conjunction must behave in a way that is consistent simultaneously with chaos and `skip`. Thus the conjoined predicate should evaluate to `skip` (which is also one of the possible behaviours permitted by chaos). This is in fact what happens in TP where

$$\begin{aligned} \texttt{skip} \wedge \texttt{chaos} &= \texttt{skip} \wedge \mathit{true} \\ &= \texttt{skip} \end{aligned}$$

To make specification by parts work in Z, Hehner [12] suggests that you can put P_1 and Q_1 together if and only if

$$\text{pre}(P_1 \wedge Q_1) \vee \neg\, (\text{pre}\, P_1) \vee \neg\, (\text{pre}\, Q_1)$$

holds, and the joined specification is then

$$(\text{pre}(P_1 \wedge Q_1) \rightarrow P_1 \wedge Q_1) \wedge (\neg\ \text{pre}\, P_1 \rightarrow Q_1) \wedge (\neg\ \text{pre}\, Q_1 \rightarrow P_1)$$

This makes specification and refinement by parts quite complex, whereas it as best to make specification and refinement as simple as possible.

4.3 Expressiveness

We need a formal definition of when one specification language $\mathcal{L}_1$ is more expressive than another $\mathcal{L}_2$. Consider specification $spec \in \mathcal{L}$ where $\mathcal{L}$ is one of the specification languages treated in this paper (UP, TP, MRC, Z and LD). We can always translate *spec* to a predicate $translate(spec)$ in TP using the various translation formulas (13), (14), and (16) — UP and TP specifications can be used as is.

In (2), we defined an *observation* as a program execution expressed as a predicate. Let $OBSERVATION_\sigma$ be the set of all observations corresponding to a bunch of program variables σ. We can then define the set of observations $obs(spec)$ of a specification $spec \in \mathcal{L}$ over some state space σ by

$$obs(spec) \ \widehat{=}\ \{o \in OBSERVATION_\sigma \mid o \rightarrow translate(spec)\} \tag{18}$$

We can think of a language $\mathcal{L}$ as defining possible sets of observations, as described by $\underline{\mathcal{L}} \mathrel{\widehat{=}} \{obs(spec) \mid spec \in \mathcal{L}\}$. Not all possible sets of observations can necessarily be expressed in a given language [9].

Definition 2 (Language Expressiveness) *(a) A language $\mathcal{L}_1$ is* more expressive *than $\mathcal{L}_2$ precisely when $\underline{\mathcal{L}_1}$ is a superset of $\underline{\mathcal{L}_2}$, i.e. $\underline{\mathcal{L}_1} \supset \underline{\mathcal{L}_2}$. (b) Two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are* incomparable *precisely when*

$$(\exists\, s_1 \in \mathcal{L}_1, s_2 \in \mathcal{L}_2 \bullet obs(s_1) \notin \underline{L_2} \wedge obs(s_2) \notin \underline{L_1})$$

Table 2 summarizes the relationship between the various specification languages particularly by looking at extreme specifications. We showed in previous sections that TP can be used to describe all the specifications of the

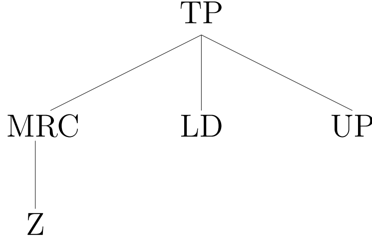

Figure 1: Expressiveness of the various calculi

other languages. However, each of the other language cannot express some property that TP can. MRC and Z cannot express *abort*. LD and UP cannot express the standard notion of total correctness (LD has its own notion of total correctness). In addition, none of the other languages can express real-time (without further modification) . Hence TP is more expressive than the other languages.

A Z specification with predicate P can always be translated to an equivalent MRC specification via $\sigma : \llparenthesis\, \mathrm{pre}\, P,\ P\, \rrparenthesis$, where σ does not include time t. However, Z cannot express a *miracle*, whereas MRC can. Thus, MRC is more expressive than Z (MRC is more flexible than Z because its precondition can be chosen arbitrarily whereas in Z, the precondition is determined from the predicate).

LD can describe *abort* whereas MRC and UD cannot. MRC can describe the standard notion of total correctness, whereas LD and UD cannot. UP can describe partial correctness whereas MRC and LD cannot. We thus have the following theorem.

Theorem 3 (Expressiveness) *(a) TP is more expressive than MRC, Z, LD and UP. (b) MRC is more expressive than Z. (c) MRC, LD and UP are incomparable with each other.*

The expressiveness hierarchy is shown in Figure 1 where the calculus at the top of a vertical line is more expressive than the one at the bottom. LD has the interesting property that it can describe all the extreme specifications.

	TP	MRC[a]	Z	LD	UP
chaos[b]	$true$	$\sigma : \langle\!| false,\ D \,|\!\rangle$	$false$	$false, true$	$true$
choose[c]	$t' \neq \infty \wedge t' \geq t$	$\sigma : \langle\!| true,\ true \,|\!\rangle$	$true$	$true, true$	No
abort[d]	$t' = \infty$	No	No	$false, false$	No
miracle[e]	$false$	$\sigma : \langle\!| true,\ false \,|\!\rangle$	No	$true, false$	$false$
PC[f]	Yes	No	No	No	Yes
STC[g]	$S \overset{t,t'}{\rightarrow} D$	$\sigma : \langle\!| S,\ D \,|\!\rangle$	Yes	No	No
real-time	Yes	No	No	No	No

Table 2: Standard and extreme specifications in the calculi

[a] In this column, σ excludes t
[b] Totally arbitrary behaviour including non-terminating executions.
[c] Only the terminating executions; no result guaranteed.
[d] Only the non-terminating executions.
[e] No executions, i.e. no computer can execute it.
[f] Partial correctness, i.e. if the execution starts in S and terminates then D holds.
[g] Standard Total Correctness where S and D are contingent (neither $true$ nor $false$).

5 Discussion and Conclusions

This paper was inspired by the surveys and semantic comparisons performed in [13] and [8]. Both these papers treat the untimed predicative calculus precursor to TP discussed in [10], and not the version of TP treated in this paper.

[13] compares six specification methods with respect to extreme specifications and properties such as skip, assignment, sequential composition, deterministic and nondeterministic choice, specification by parts, ordering (refinement) and recursion. The methods treated include Jones's VDM, Dijkstra's weakest preconditions, partial relations, LD and an early version of UP. The comparison did not include TP, MRC and Z. In this paper we had the advantage of Hehner's TP that could serve as an underlying framework for the semantic treatment of other languages which made the comparison easier and allowed us to see that TP is more expressive, and its refinement rule simpler than the other methods.

Grundy [8] surveys and compares three simple styles of programming based on single predicates: the relational model (e.g. Z), partial models (i.e. correctness under the assumption of termination), and total models (e.g. the precursor to UP). Grundy shows that there is no commonly held understand-

ing of what a predicate means as a specification of behaviour. Grundy does not treat MRC and LD. Grundy's conclusion was that the predicative models all had difficulties, which is why single predicate methods "have been all but abandoned in favour of specifications phased as pairs of predicates for work in program refinement". His criticisms of the precursor to TP were that sequential composition was not associative and nondeterministic choice of two terminating behaviours can allow for non-termination. These deficiencies are not present in TP. Our conclusion in this paper is that the single predicate TP method is in fact superior to other methods when it comes to refinement and specification by parts.

It is a surprising fact that a standard Z logic has only been proposed recently [28], given the attention Z has garnered as a formal method. Early attempts at a Z logic contained many ambiguities and even inconsistencies [14, 19]. We refer the reader to [15, 16] for further discussion of the issues. Some researchers are of the opinion that Z should remain a pure specification language unrelated to questions of implementation. Others (ourselves included) feel that Z's lack of a standard development method is a problem. What use is a pure specification without knowing what system behaviour is being required? At the very least, we need a theory that links specifications to the observations (or behaviours) of the system under description. We refer the reader to ZRC [4] for a discussion of these issues; this paper also provides a theory of refinement for transforming Z schemas into programs akin to that of MRC.

Back [2] presents a refinement calculus for reasoning about total correctness and refinement of programs using lattice theory as the underlying semantic foundation in which one can look at partial functions, relations and predicate transformers as complementary methods for program derivations.

We have focussed on procedural refinement and the resulting notions of termination. The authors of [26] compare various methods of data refinement. They show that refinement proofs reduce to proving simulation, and they compare 13 methods including TP, VDM, Z, and MRC for incompleteness (the refinement holds but cannot be proved in the method). The authors of [26] write about Hehner's method (TP): "The introduction of data invariants and/or initialization predicates then yields a stunningly simple and elegant method for data refinement, which is complete". By contrast, the data refinement methods for VDM and Z are incomplete.

There have been extensions of Z [6] and MRC [20] to deal with program refinement in the context of real-time systems. In both cases, the refinement

rules have to be changed to accomodate the passage of time. In TP, the refinement rules themselves remain the same. A further advantage of TP is that refinement by parts works in the expected fashion. Thus, the timing and partial correctness results can each be treated in their own right (e.g. see page 57 in [11]).

To what extent can we extend the use of TP as an underlying framework for other model-oriented methods? VDM is based on a three-valued logic; we would thus have to extend the bunch *bool* in [11] with a third value which would result in changes to many of the refinement rules. Thus VDM would not be a suitable candidate for comparison with the methods of this paper.

The B method [1] has a notion of refinement based on first order logic. The definition of refinement given in Chapter 11 of [1] is in terms of the set-transformer semantics rather than weakest preconditions, although it appears to be equivalent to the definition used by others who work with a weakest-precondition semantics. The definition is higher order but Abrial uses the neat trick of showing that it is equivalent to a first order definition [3]. It would appear to be useful to extend the comparisons of this paper to B. But B also provides a specification structuring mechanism (the abstract machine) that goes beyond the structuring mechanisms provided by the methods in this paper: abstract machines can group operations and state, whereas there is nothing equivalent in Z, TP, MRC, or LD.

We now summarise the main conclusions of this paper:

- TP is a single predicate method that can be used to specify and refine sequential and concurrent or real-time programs. It is more expressive than the other methods treated in this paper, can serve as an underlying semantic framework for them, and has the simplest refinement rule. As a consequence, refinement rules of other methods such as MRC can be used in TP. Specification and refinement by parts is simple and can be done without the constraints of other methods.

- Predicates can have very different meanings in the various methods and care must thus be taken in interpreting them. In particular, we have some concerns with Z: there is no fixed standard for how the specifications relate to programs; specification and refinement by parts is complex; and refinement rules need to be changed when dealing with real-time.

Our original motivation for this paper was work we have recently carried out on developing a predicative refinement calculus for object-oriented software development [23, 24] using BON/Eiffel. Our conclusion is that TP provides the simplest basis for an object-oriented refinement calculus. And because of the timed nature of TP, it has the built-in techniques that we will need in extending the calculus to real-time object-oriented program refinement.

References

[1] Abrial, J.-R. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[2] Back, R.J. and J.v. Wright. *Refinement Calculus*. Graduate Texts in Computer Science, Ed: D. Gries and F. Schneider, Springer-Verlag, New York, 1998.

[3] Butler, M.J. Review of "The B-Book" by J.-R. Abrial. *The Computer Journal*, 40(1): p59-61, 1997.

[4] Cavalcanti, A. and J. Woodcock. ZRC - A Refinement calculus for Z. *Formal Aspects of Computing*, 10: p267-289, 1998.

[5] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[6] Fidge, C. Proof Obligations for Real-Time Refinement. In *6th Refinement Workshop*, edited by D. Till, BCS, 279-306, 1994. http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?93-16

[7] Gries, D. and F.B. Schneider. *A Logical Approach to Discrete Math.* Springer Verlag, 1993.

[8] Grundy, J. Predicative Programming - A Survey. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, Springer-Verlag, LNCS 735, 8-25, 1993.

[9] Hayes, I.J. Expressive Power of specification Languages. *Formal Aspects of Computing*, 10: p187-92, 1998.

[10] Hehner, E.C.R. Predicative Programming. *Communications of the ACM*, 27(2): p134-152, 1984.

[11] Hehner, E.C.R. *A Practical Theory of Programming*. Springer-Verlag, New York, 1993.

[12] Hehner, E.C.R. Specifications, Programs and Total Correctness. *Science of Computer Programming*, 34: p191-205, 1999.

[13] Hehner, E.C.R. and A.J. Malton. Termination Convention and Comparitive Semantics. *Acta Informatica*, 25: p1-14, 1988.

[14] Henson, M.C. The Standard Logic of Z is Inconsistent. *Formal Aspects of Computing*, 10: p243-247, 1998.

[15] Henson, M.C. and S. Rees. Revising Z: Part I - logic and semantics. *Formal Aspects of Computing*, 11: p359-380, 1999.

[16] Henson, M.C. and S. Reeves. Revising Z: Part II - logical development. *Formal Aspects of Computing*, 11: p381-407, 1999.

[17] Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*, 12: p576-583, 1969.

[18] Hoare, C.A.R. Unified Theories of Programming. In *Mathematical Methods in Program Development*, ed. M. Broy and B. Schieder. 313–365. NATO ASI Series. Series F: Computer and Systems Sciences. Vol. 158. Springer-Verlag, 1997.

[19] King, S. ‘The Standard Logic for Z’ - A Clarification. *Formal Aspects of Computing*, 11: p472-473, 1999.

[20] Mark Utting, M. and C. Fidge. A Real-Time Refinement Calculus that Changes only Time. In *Proc. of the 7th Refinement Workshop*, BCS-FACS, 266-281, 1996. svrc.it.uq.edu.au/Bibliography/svrc-tr.html?95-48

[21] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[22] Morgan, C. *Programming from Specifications*. International Series in Computer Science, Prentice Hall, 1994.

[23] Paige, R.F. and J.S. Ostroff. Developing BON as an Industrial-Strength Formal Method. In *Proc. World Congress on Formal Methods (FM'99)*, Springer-Verlag, LNCS 1708, 1999.

[24] Paige, R.F. and J.S. Ostroff. An Object-Oriented Refinement Calculus. Department. of Computer Science, York University, Toronto. CS-1999-07, 1999. http://www.cs.yorku.ca/techreports/1999/CS-1999-07.html

[25] Parnas, D.L. Mathematical Descriptions and Specification of Software. In *Proceedings of IFIP World Congress 1994*, Volume I, 354-359, 1994.

[26] Roever, W.-P.d. and K. Engelhardt. *Data Refinement: Model Oriented Proof Methods and their Computations*. Vol:47, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[27] Spivey, J.M. *The Z Notation: A Reference Manual (2nd edition)*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

[28] Toyn, I. Z Notation: Final Committee Draft, CD 13568.2. 1999. ftp://ftp.cs.york.ac.uk/hise_reports/cadiz/ZSTAN/fcd.pdf.

[29] Ward, N. Adding Specification Constructors to the Refinement Calculus. In *Proc. FME'93*, Springer-Verlag, LNCS 670, 1993.

[30] Wordsworth, J. *Software Development with Z*. Addison-Wesley, 1994.