# Unifying Theories of Programming that Distinguish Nontermination and Abort

Ian J. Hayes[1], Steve E. Dunne[2], and Larissa Meinicke[3]

[1] The University of Queensland, Brisbane, 4072, Australia
[2] School of Computing, University of Teesside, Middlesbrough, TS1 3BA, UK
[3] Macquarie University, Sydney, Australia

**Abstract.** In this paper we focus on the relationship between a number of specification models. The models are formulated in the Unifying Theories of Programming of Hoare and He, but correspond to widely used specification models. We cover issues such as partial correctness, total correctness, and general correctness. The properties we use to distinguish the models are these:

- whether they allow the specification of assumptions about the initial state outside of which no guarantees are given about the behaviour of the program, i.e., the program may "abort";
- whether a specification may allow or even require nontermination as a valid (non-aborting) outcome; and
- whether they allow the expression of *tests* or *enabling conditions*, outside of which the program has no possible behaviour.

When considering termination, we consider both an abstract model, which only distinguishes whether a program terminates or not, as well as models that include a notion of time: either abstract time representing a notion of progress or real-time.

## 1 Introduction

The aim of this paper is to better understand the relationships between a number of models of program specifications. We are interested in whether they can express properties such as total correctness, partial correctness, general correctness, timing properties, and reactive behaviour. As a framework to relate these models we use Hoare and He's Unifying Theories of Programming (UTP) [1], because this theory is general enough to do this succinctly.[4] Section 2 addresses UTP designs (or specifications), which support total-correctness specifications in the form of a precondition and a pre-post relation. These correspond to specifications in VDM [2], the refinement calculus [3–6], and B [7]. Section 3 examines Z specifications [8, 9], which form the least expressive model considered here.

Section 4 introduces a new model that extends designs to distinguish abort and nontermination; this allows specification of both total- and partial-correctness properties.

[4] The UTP models that we consider are based on homogeneous relations between states. These are not rich enough to express both demonic and angelic choice simultaneously, which is possible in predicate transformer models, but such relational models are sufficient for the properties explored in this paper.

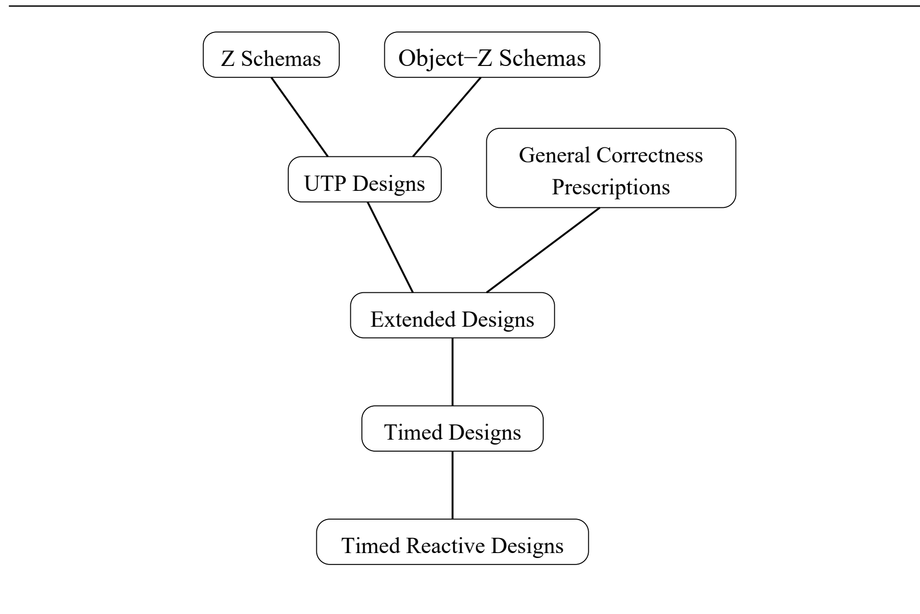


**Fig. 1.** Relationships between models: most general at the bottom

Extended designs can also express general-correctness properties. General correctness allows a termination set to be defined, but does not model a program aborting. General correctness is explored in Section 5, and Section 6 highlights the distinction between an assumption on the initial state and a termination set.

Section 7 generalises extended designs to allow timing properties to be expressed by using an observation of the time, $\tau$, with $\tau' = \infty$ signifying nontermination; this approach is similar to that used by Hehner [10]. Section 8 generalises this further to allow the expression of reactive properties of traces of the program variables over time; this model corresponds to the real-time refinement calculus [11–15]. Figure 1 summarises the relationships between the models graphically.

Before getting into the details of the models, we warn the reader that the literature on the different models uses the term "precondition" in various different ways. It may be any of

- an *assumption*: a condition characterising those initial states from which the program is required to work properly or not be enabled, but outside of which it may "abort" (for example, by crashing), or terminate capriciously with an incorrect result, or fail to terminate at all by executing forever;
- a *termination set*: the initial states from which termination is required;
- an *enabling condition* (or *guard* or *test*): execution can only begin from initial states satisfying the condition (i.e., it is infeasible outside the enabling condition); or
- combinations of the above.

In all cases it is a predicate on the initial state. These different interpretations of the term "precondition" can lead to misunderstandings when moving from one model to another. One aim of this paper is to clarify these distinctions by making it clear how "precondition" is interpreted in each model.

**Syntactic substitution.** We use the notation $r\left[\frac{e}{v}\right]$ to stand for the relation $r$ with every free occurrence of the variable name $v$ replaced by the expression $e$. This can also be generalised so that $v$ is a list of variable names and $e$ a corresponding list of expressions.

## 2 UTP Designs

In Hoare and He's Unifying Theories of Programming (UTP), designs (or specifications) and programs are modelled via relations between the before and after program states [1]. A special boolean observation *okay* is used to model program termination, which in the model is indistinguishable from the program not aborting. Hoare and He [1] also use the term *stable*. A *design* has the syntax $(p \vdash w)$, where $p$ is a single-state predicate on the before-values of the program variables and $w$ characterises a relation between before- and after-values of the program variables.[5] For UTP designs the *precondition*, $p$, represents both an assumption on the initial state and the set of initial states from which termination is required. The semantics of a design is given by a relation characterised by the predicate

$$okay \wedge p \Rightarrow okay' \wedge w \ , \tag{1}$$

where unprimed variable names correspond to the initial values of the variables and primed names to their final values. If the program starts (i.e., *okay* holds) in an initial state in which $p$ holds, the program will terminate (i.e., $okay'$ will hold) and relation $w$ will hold between the initial and final states. Neither $p$ nor $w$ may refer to the observations *okay* and $okay'$.[6] Designs only model total correctness. To simplify the presentation below, we do not distinguish between a relation and the predicate characterising that relation.

A design $(p_0 \vdash w_0)$ *is refined by* another design $(p_1 \vdash w_1)$, written

$$(p_0 \vdash w_0) \sqsubseteq (p_1 \vdash w_1) \ ,$$

provided the semantic relation defined by the latter implies (is included in) the semantic relation defined by the former, that is,

$$[(okay \wedge p_1 \Rightarrow okay' \wedge w_1) \Rightarrow (okay \wedge p_0 \Rightarrow okay' \wedge w_0)] \ , \tag{2}$$

where, as in Hoare and He [1], the notation $[P]$ stands for the universal quantification of $P$ over all variables in the alphabet (including *okay* and $okay'$). Refinement condition (2) holds if and only if

$$[p_0 \Rightarrow (p_1 \wedge (w_1 \Rightarrow w_0))] \ . \tag{3}$$

---

[5] Hoare and He [1] allow $p$ to be a relation in general, but then introduce a constraint (H3) that requires $p$ to be single-state. The designs we describe here are thus their H3-designs.

[6] Allowing $p$ and $w$ to refer to *okay* and $okay'$ doesn't add anything because $(p \vdash w)$ is semantically equivalent to $(p\left[\frac{\mathrm{true}}{okay}\right] \vdash w\left[\frac{\mathrm{true},\mathrm{true}}{okay,okay'}\right])$.

There are three interesting extreme cases of designs:

$$\begin{aligned} \textbf{abort}_{UTP} &\mathrel{\widehat{=}} (\text{false} \vdash \text{true}) \\ \textbf{terminates}_{UTP} &\mathrel{\widehat{=}} (\text{true} \vdash \text{true}) \\ \textbf{magic}_{UTP} &\mathrel{\widehat{=}} (\text{true} \vdash \text{false}) \end{aligned}$$

UTP designs form a complete lattice under the refinement ordering, with least element $\textbf{abort}_{UTP}$ and greatest element $\textbf{magic}_{UTP}$. The design $\textbf{terminates}_{UTP}$ is the specification that guarantees termination but nothing else. Note that the design $(\text{false} \vdash w)$ is semantically equivalent to $\textbf{abort}_{UTP}$ for any relation $w$. A design is *infeasible* in any state, $v$, for which the precondition $p$ holds but the relation $w$ doesn't relate $v$ to any final state, i.e., the set of infeasible states are those satisfying $p \land \neg (\exists v' \bullet w)$. The design $\textbf{magic}_{UTP}$ is everywhere infeasible.

## 3 Z specifications

Let $S$ be a Z schema [8, 9] representing the program state, then the Z schema

can be used as a specification. Here $S'$ stands for $S$ with all components decorated with a prime, and $R$ is a predicate relating the components of $S$ and $S'$. To save space we write the above schema in its equivalent *horizontal* form: $[S;\ S' \mid R]$.

In Z, the precondition of an operation is the predicate $(\exists S' \bullet R)$, i.e., any initial state for which there exists a corresponding final state. The precondition is the domain of the relation $R$. Under the conventional so-called *contract*, or *non-blocking*, interpretation of a Z operation schema [16], a precondition in Z represents, as for a UTP design, both an assumption on the initial state and the set of initial states on which termination is required. Thus the schema $Z$ above *is refined by* a schema $[S;\ S' \mid Q]$, i.e., $[S;\ S' \mid R] \sqsubseteq [S;\ S' \mid Q]$, provided

$$[(\exists S' \bullet R) \Rightarrow ((\exists S' \bullet Q) \land (Q \Rightarrow R))] \ .$$

**Relating Z schemas and designs.** Any Z schema of this form can be uniquely mapped to a UTP design by the function $ZD$ defined as follows.

$$ZD([S;\ S' \mid R]) \mathrel{\widehat{=}} ((\exists S' \bullet R) \vdash R)$$

This mapping preserves the refinement ordering, and the image of the mapping forms a subtheory of designs [1, Chap. 4].

There are two interesting extreme cases:

$$\mathbf{abort}_Z \mathrel{\widehat=} [S;\ S' \mid \mathrm{false}]$$
$$\mathbf{terminates}_Z \mathrel{\widehat=} [S;\ S' \mid \mathrm{true}]$$

where $\mathbf{abort}_Z$ is the least program in the refinement ordering. These schemas correspond to their equivalents using designs, i.e.,

$$ZD(\mathbf{abort}_Z) = \mathbf{abort}_{UTP}$$
$$ZD(\mathbf{terminates}_Z) = \mathbf{terminates}_{UTP}$$

but note that one cannot represent magic by a Z specification. More generally, Z cannot represent infeasible specifications and hence the Z model is not as expressive as UTP designs (or VDM pre/post specifications, or B specifications).

In Object-Z [17, 18] operations are always terminating and the domain of the relation $R$ is treated as the operation's enabling condition[7]. Hence for Object-Z the schema $Z$ above is mapped to a design as follows:

$$OZD([S;\ S' \mid R]) \mathrel{\widehat=} (\mathrm{true} \vdash R) \ .$$

In the Object-Z model one can express infeasible specifications, although all specifications are terminating and non-aborting.

## 4 Distinguishing nontermination and abort

For reactive and real-time programs, it is often desirable to distinguish between abort and nontermination, because nontermination can be a desirable property that one would like to allow or even require in some circumstances. Even in the non-reactive case, one would like to be able to specify heuristic search problems, where if the program terminates, it returns a valid answer to a query, but the program is not guaranteed to terminate for all queries. It is not possible to specify such programs using designs.

An important distinction between nontermination and abortion is that, while the former may sometimes be a desirable property, the latter never is, so while a specification may sometimes tolerate abortion, it should never demand it. Even the specification $\mathbf{abort}_{UTP}$, while everywhere admitting abortion, does not actually demand it. So while some of our new constructs in this paper allow a specification sometimes to demand nontermination, none of them provides a means of demanding abortion.

To allow such specifications to be expressed, we extend the model to allow nontermination and a program aborting to be distinguished. To do this, we use the boolean observation $term'$ to model termination, and the boolean observation $ok'$ to model the program not aborting. Informally $ok' \wedge term'$ in this model is equivalent to $okay'$ for the UTP design model. As with the UTP design model, we also have the observations $ok$, indicating that the program starts in a stable state (i.e., the preceding program has not aborted), and $term$, indicating that the program starts at some finite time, (i.e., the

[7] This is known in the Z and Object-Z literature as the *blocking* interpretation [16].

preceding program terminated). We introduce a new form of design, an *extended design* $(p \vdash_X r)$, that if $p$ holds initially, guarantees to deliver a post-state satisfying $r$ with respect to the pre-state. The syntax of an extended design uses "$\vdash_X$" to distinguish it from a design, which uses just "$\vdash$". The assumption $p$ holding initially does not guarantee termination, but $r$ may explicitly refer to $term'$ to require termination. For example, for a relation $w$ that does not refer to $term'$,

- $(p \vdash_X term' \wedge w)$ requires that if $p$ holds initially, the program should terminate and satisfy $w$ between its pre-state and post-state;
- $(p \vdash_X term' \Rightarrow w)$ requires that if both $p$ holds initially and the program terminates, then it also satisfies $w$;
- $(p \vdash_X (q \Rightarrow term' \wedge w) \wedge (\neg\, q \Rightarrow \neg\, term'))$, where $q$ is a single-state predicate on the pre-state, requires that if $p$ holds initially, then both the following conditions hold: if $q$ holds initially, the program should terminate and satisfy $w$; and if $q$ does not hold initially, the program never terminates.

For (non)termination there are three possibilities for each initial state: termination is required, nontermination is required, or either termination or nontermination is possible.

The semantics of the extended design $(p \vdash_X r)$ as a relation is characterised by the following predicate:

$$(ok \wedge term \wedge p \Rightarrow ok' \wedge r) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term)\ . \tag{4}$$

It says that if the program starts in a stable state at some finite time (i.e., $ok \wedge term$) in an initial state in which $p$ holds, then the program will remain in a stable state (i.e., $ok'$) and relation $r$ will hold between the initial and final states. A nonterminating program is *ipso facto* non-aborting and therefore stable $(\neg\, term' \Rightarrow ok')$, and a program can only terminate if its predecessor terminated $(term' \Rightarrow term)$. For an extended design, the single-state predicate $p$ should not refer to $ok$ or $term$, and the relation $r$ may refer to $term'$ but not $ok$, $ok'$ or $term$.[8] The relation $r(v, v', term')$ is in terms of the initial values of the program variables $v$, their final values $v'$, and the final termination observation $term'$.

An extended design of the form

$$(\mathrm{true} \vdash_X \neg\, term' \wedge x' = 1) \tag{5}$$

is not sensible because it constrains the final value of $x$ to be one, even though it is guaranteed to never terminate. Because one can never observe the final values of the program variables in the case of nontermination, for an extended design to be well formed, we require that $r$ is such that it does not constrain the final values of the program variables in the case of nontermination, i.e.,

$$[p \wedge \neg\, term' \Rightarrow (r \Leftrightarrow (\forall\, v' \bullet r))]\ , \tag{6}$$

where $v'$ is the set of final-state program variables. Note that the program variables $(v)$ do not include the observations $ok$ and $term$. The example (5) does not satisfy this

[8] Again, allowing $p$ and $r$ to refer to $ok$, $ok'$, and $term$ doesn't add anything because $(p \vdash_X r)$ is semantically equivalent to $(p \left[\frac{\mathrm{true},\mathrm{true}}{ok,term}\right] \vdash_X r \left[\frac{\mathrm{true},\mathrm{true},\mathrm{true}}{ok,term,ok'}\right])$.

requirement because the following does not hold for all values of $term'$ and $x'$ :

$$
\begin{aligned}
&[\neg\, term' \Rightarrow (\neg\, term' \wedge x' = 1 \Leftrightarrow (\forall\, x' \bullet \neg\, term' \wedge x' = 1))] \\
\equiv\ &[\neg\, term' \Rightarrow (x' = 1 \Leftrightarrow \mathrm{false})] \\
\equiv\ &[\neg\, term' \Rightarrow (x' \neq 1)]\ .
\end{aligned}
$$

An extended design $(p_0 \vdash_X r_0)$ *is refined by* another extended design $(p_1 \vdash_X r_1)$ provided the semantic relation defined by the latter implies the semantic relation defined by the former, that is,

$$
\begin{aligned}
&[(ok \wedge term \wedge p_1 \Rightarrow ok' \wedge r_1) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term) \Rightarrow \\
&\ (ok \wedge term \wedge p_0 \Rightarrow ok' \wedge r_0) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term)]\ ,
\end{aligned}
$$

which holds if and only if

$$[p_0 \Rightarrow (p_1 \wedge (r_1 \Rightarrow r_0))]\ . \tag{7}$$

This is similar to the condition for refining UTP designs (3), except that $r_0$ and $r_1$ may refer to $term'$ to specify termination behaviour.

**Relating designs and extended designs.** To see that an extended design generalises a design, we show that we can map any design into a unique extended design. For any design, $(p \vdash w)$, we have

$$DX(p \vdash w) \mathrel{\widehat{=}} (p \vdash_X term' \wedge w)\ .$$

It is straightforward to show that this mapping preserves the refinement ordering, and that the image of this mapping is a subtheory of extended designs [1, Chap. 4].

For extended designs, we have the following interesting extreme cases:

$$
\begin{aligned}
\mathbf{abort}_X &\mathrel{\widehat{=}} (\mathrm{false} \vdash_X \mathrm{true}) \\
\mathbf{chaos}_X &\mathrel{\widehat{=}} (\mathrm{true} \vdash_X \mathrm{true}) \\
\mathbf{terminates}_X &\mathrel{\widehat{=}} (\mathrm{true} \vdash_X term') \\
\mathbf{forever}_X &\mathrel{\widehat{=}} (\mathrm{true} \vdash_X \neg\, term') \\
\mathbf{magic}_X &\mathrel{\widehat{=}} (\mathrm{true} \vdash_X \mathrm{false})
\end{aligned}
$$

where $\mathbf{abort}_X$, $\mathbf{terminates}_X$, and $\mathbf{magic}_X$ correspond to their UTP design equivalents (via the mapping $DX$). The other two commands do not have equivalent UTP designs: $\mathbf{chaos}_X$ does not abort but it may or may not terminate, and if it terminates then any final state is possible; and $\mathbf{forever}_X$ does not abort but also never terminates. The extended design $\mathbf{chaos}_X$ is refined by both $\mathbf{terminates}_X$ and $\mathbf{forever}_X$. Extended designs form a complete lattice under the refinement ordering, with least element $\mathbf{abort}_X$ and greatest element $\mathbf{magic}_X$.

**Total and partial correctness.** To show that a program $s$ is totally correct with respect to the precondition $p$ (interpreted as both an assumption on the initial states and a termination set) and relation $w$, we must show

$$(p \vdash_X term' \wedge w) \sqsubseteq s$$

and to show partial correctness with respect to the same precondition (this time interpreted as just an assumption on the initial state) and relation, we must show

$$(p \vdash_X term' \Rightarrow w) \sqsubseteq s \ .$$

## 5 General correctness

Parnas [19, 20] introduced the notion of a limited domain (LD) relation to describe termination sets and pre-post relations (of a control structure that generalised Dijkstra's guarded command control structures [21]). Jacobs and Gries [22] introduced a similar idea called *general correctness*, which has been further explored by Nelson [23] and Dijkstra and Scholten [24]. Dunne has studied general correctness [25, 26] and incorporated general correctness into a UTP setting [27]. He makes use of a *prescription* of the form $(p \Vdash w)$, which is guaranteed to terminate from initial states in which $p$ holds, and if it does terminate (whether or not it was guaranteed to do so) then relation $w$ holds on termination. Note that the syntax of a prescription uses a "$\Vdash$" in place of a "$\vdash$" to distinguish it. We can model the semantics of the prescription $(p \Vdash w)$ as a relation by making use of the observation *term*, which represents termination[9]:

$$(term \wedge p \Rightarrow term') \wedge (term' \Rightarrow w \wedge term) \ . \qquad (8)$$

The following examples illustrate the expressive versatility of prescriptions:

- $(\mathrm{true} \Vdash w)$ guarantees termination from any state and that $w$ holds;
- $(p \Vdash p \Rightarrow w)$ requires that in any initial state in which $p$ holds, the program terminates and satisfies $w$, and if $p$ does not hold initially, there is no guarantee of termination and no guarantee about the final state (although it never aborts — see Section 6 for further explanation);
- $(\mathrm{false} \Vdash w)$ corresponds to a partial correctness specification — although no guarantee of termination is given, if it does terminate, $w$ holds; and
- $(\mathrm{false} \Vdash \mathrm{false})$ guarantees to never terminate.

A prescription $(p_0 \Vdash w_0)$ *is refined by* another prescription $(p_1 \Vdash w_1)$ provided the semantic relation defined by the latter implies (is included in) the semantic relation defined by the former, that is,

$$[(term \wedge p_1 \Rightarrow term') \wedge (term' \Rightarrow w_1 \wedge term) \Rightarrow$$
$$(term \wedge p_0 \Rightarrow term') \wedge (term' \Rightarrow w_0 \wedge term)] \ ,$$

which holds if and only if $[p_0 \Rightarrow p_1] \wedge [w_1 \Rightarrow w_0]$ .

[9] We use the observation name "*term*" to be consistent with the terminology in the rest of this paper, although the name "*ok*" is used by Dunne [27].

**Relating prescriptions and extended designs.** To see that an extended design generalises a prescription, we show that we can map any prescription into a unique extended design. For any prescription $(p \Vdash w)$ we have

$$PX(p \Vdash w) \mathrel{\widehat{=}} (\mathrm{true} \vdash_X (p \Rightarrow term') \wedge (term' \Rightarrow w)) \ .$$

It is straightforward to show that this mapping preserves the refinement ordering, and that the image of this mapping is a subtheory of extended designs.

For prescriptions we have the following interesting extreme cases:

$$\begin{aligned} \mathbf{chaos}_P &\mathrel{\widehat{=}} (\mathrm{false} \Vdash \mathrm{true}) \\ \mathbf{terminates}_P &\mathrel{\widehat{=}} (\mathrm{true} \Vdash \mathrm{true}) \\ \mathbf{forever}_P &\mathrel{\widehat{=}} (\mathrm{false} \Vdash \mathrm{false}) \\ \mathbf{magic}_P &\mathrel{\widehat{=}} (\mathrm{true} \Vdash \mathrm{false}) \end{aligned}$$

where these all correspond to their extended design equivalents, but note that there is no equivalent of $\mathbf{abort}_X$. Prescriptions form a complete lattice under the refinement ordering, with least element $\mathbf{chaos}_P$ and greatest element $\mathbf{magic}_P$. We expand on the distinction between general correctness and extended designs in the next section.

## 6 Assumptions on the initial state versus termination sets

In both the UTP design, $(p \vdash w)$, and the extended design, $(p \vdash_X r)$, the predicate $p$ acts as an assumption the implementor can make about the initial state. If $p$ doesn't hold initially then the implementation is free to do anything, even abort. The UTP design has the requirement that the program must also terminate whenever $p$ holds initially. Hence for a UTP design, $p$ is both an assumption on the initial state and a termination set. For extended designs, $p$ is only an assumption on the initial state. Modulo $p$, the termination set is specified within $r$. This is because any behaviour is allowable if $p$ does not hold initially, so termination is only guaranteed from those initial states where both $p$ holds and $r$ requires termination.

In the general correctness prescription $(p \Vdash w)$, the predicate $p$ specifies the termination set. There is no way to specify an assumption on the initial state (in the above sense) in general correctness, because general correctness has no notion of abortion. One can get close with a prescription of the form $(p \Vdash q \Rightarrow w)$, where $q$ is a single-state predicate on the initial state. If $q$ does not hold initially, then any non-aborting behaviour is allowed. However, this has a subtle difference in behaviour when prescriptions are sequentially composed. In all our models, sequential composition is defined as the relational composition of the semantic relations of the two commands. For general correctness we have that

$$(p \Vdash \mathrm{true});\ (\mathrm{true} \Vdash x' = 1) \tag{9}$$

guarantees that, even if $p$ does not hold initially, if the first prescription terminates, then the whole terminates and the final value of $x$ will be one. Hence, if the whole terminates, then $x$ is guaranteed to be one. If we replace the prescriptions in (9) with UTP designs

or extended designs of the same form, no such guarantee about the final value of $x$ is given if $p$ does not hold initially. With the UTP design $(p \vdash \mathrm{true})$, if $p$ doesn't hold initially, then its semantic relation allows $okay'$ to be false, in which case $okay$ may be false for the second command $(\mathrm{true} \vdash x' = 1)$ and hence it can do anything, and no guarantee can be given about the final value of $x$. However, for the prescription $(p \Vdash \mathrm{true})$, if $p$ doesn't hold initially, this prescription isn't required to terminate, but if it does terminate, $(\mathrm{true} \Vdash x' = 1)$ is then required to terminate and set $x$ to one.

Note that for extended designs, we have the law

$$\mathbf{abort}_X;\ s = \mathbf{abort}_X\ ,$$

but the following law does not hold in general

$$\mathbf{chaos}_X;\ s = \mathbf{chaos}_X\ .$$

In summary, the implementor can rely on the assumption, $p$, on the initial state holding. Nothing can be assumed about an implementation, $I$, when it is executed from an initial state not satisfying $p$, and furthermore nothing can be assumed about the behaviour of any component executing after $I$ if the execution of $I$ happens to terminate. In contrast (non)termination represents an allowed or required behaviour of any implementation. The reason these are often confused is that both assumptions on the initial state and termination sets are defined in terms of a condition on the initial state.

## 7 Timed designs

To discuss timing issues one can introduce an observation representing the current time, as done by Hehner [10, 28, 29] and Abadi and Lamport [30]. An extended design can be generalised to a timed design by replacing the observations $term$ and $term'$ by the observations $\tau$ and $\tau'$, representing the initial and final times, respectively. The two most interesting choices for representing time are the natural numbers and the non-negative reals, in both cases augmented with the value $\infty$ to represent nontermination. For our discussion here either representation is valid. Hehner [10, 28] uses natural numbers to represent *abstract time*, that is, they represent a notion of progress rather than real time. The real-time refinement calculus [13] uses real numbers to represent real time.

A *timed design*, $(q \vdash_T r)$, has a semantics given by the following relation:

$$(ok \land \tau \neq \infty \land q \Rightarrow ok' \land r) \land (\tau' = \infty \Rightarrow ok') \land \tau \leq \tau'\ . \qquad (10)$$

The form is similar to that for an extended design (4), except that we require that time does not go backwards, i.e., $\tau \leq \tau'$. We allow $q$ to refer to the before-values of the program variables as well as $\tau$ and $\tau'$, and $r$ can to refer to both the before- and after-values of the program variables as well as $\tau$ and $\tau'$, but neither $q$ nor $r$ can refer to $ok$ or $ok'$. Because we allow it to refer to $\tau'$ the precondition $q$ – which specifies the states in which the program is guaranteed not to abort – is no longer a condition on the initial state only, unlike the preconditions of each of our previous designs. To emphasise this we have used the name $q$ rather than $p$, which we reserve for predicates on a single state.

We allow $q$ to refer to $\tau'$ so that we may express constraints regarding *when* the program may abort. But note that $q$ may not refer to $v'$ because, unlike $\tau'$, the final values $v'$ of the program variables cannot be constrained in the event of the program aborting.

The inclusion of a time variable makes it possible to express, not just if the program terminates, but when it terminates. Such execution time constraints may be included in $r$, e.g., $\tau' - \tau \leq 1$ requires execution to take at most one time unit. Execution time constraints may be used to define a deadline command [31], that requires that the time is at most $D$ when the deadline command is reached, as the timed design $(\mathrm{true} \vdash_T \tau = \tau' \leq D \wedge \mathrm{id})$, where $\mathrm{id}$ is the identity relation on program variables. The deadline command is a specification construct; it cannot be directly implemented.

As already mentioned, as well as being used to specify termination time constraints, time may also be used to specify when the program may abort. Program abortion time constraints can be included in $q$. If $q$ is taken to be $\mathrm{false}$, as in the extreme program

$$\mathbf{abort}_T \mathrel{\widehat{=}} (\mathrm{false} \vdash_T \mathrm{true}) \ ,$$

we have that the program may become unstable immediately at the initial time $\tau$. Since $q$ is able to reference the final time, $\tau'$, it is also possible to specify that a program may abort at least $t$ time units *after* the start time $\tau$. For example, design

$$(\tau' - \tau < 10 \vdash_T r)$$

may either terminate within 10 time units satisfying $r$, or it may do anything as long as the $\tau'$ is greater than or equal to $\tau + 10$. A special case of this is the timed design

$$(\tau' - \tau < 10 \vdash_T \mathrm{false})$$

which guarantees to run for 10 time units, after which it may become unstable. It cannot terminate within 10 time units because in doing so it would incur the impossible obligation of satisfying $\mathrm{false}$. This program may also be expressed as the sequential composition

$$(\mathrm{true} \vdash_T \tau' - \tau \geq 10);\ \mathbf{abort}_T$$

but note that this sequential composition could not be expressed as a single timed design if we did not allow the precondition to refer to $\tau'$.

We consider a timed design such as $(\tau' - \tau > t \vdash_T r)$ for some non-negative time $t$ not to be reasonable since it would put an upper bound on the time at which the program may abort. Since we would like to specify that a program that may abort at time $t$ may be implemented by one which aborts at some later time (that is, a program that delays the occurrence of a catastrophic event), we impose a condition on the assumption $q$ of a timed design $(q \vdash_T r)$ that $\neg\, q$ must not impose an upper bound on $\tau'$, i.e.,

$$\left[\neg\, q \Rightarrow \left(\forall\, \tau'' \bullet \tau' < \tau'' \Rightarrow \neg\, q\left[\frac{\tau''}{\tau'}\right]\right)\right] \ . \tag{11}$$

We also need a timed-design version of condition (6) ensuring that the final values of the program variables are not constrained under nontermination:

$$[\tau \neq \infty \wedge q \wedge \tau' = \infty \Rightarrow (r \Leftrightarrow (\forall\, v' \bullet r))] \ . \tag{12}$$

*Refinement* of timed designs,

$$(q_0 \vdash_T r_0) \sqsubseteq (q_1 \vdash_T r_1) ,$$

is defined in terms of reverse implication of the equivalent semantic relations, and hence holds provided

$$[\tau \neq \infty \wedge \tau \leq \tau' \wedge q_0 \Rightarrow ((\tau' \neq \infty \Rightarrow q_1) \wedge ((q_1 \Rightarrow r_1) \Rightarrow r_0))] . \qquad (13)$$

This condition is similar to that for UTP designs (3) and extended designs (7), except that it adds the implicit precondition that the start time is finite, and the healthiness condition that no command can allow time to go backwards. The consequent is also expressed differently because $q_1$ may refer to the finish time $\tau'$. In the common special case that $q_1$ is independent of $\tau'$, the consequent simplifies to $(q_1 \wedge (r_1 \Rightarrow r_0))$. In the more general case, satisfaction of the antecedent $\tau \neq \infty \wedge \tau \leq \tau' \wedge q_0$ need only imply that $q_1$ holds when $\tau'$ is finite, since programs may not abort at time infinity, however it must always guarantee that $((q_1 \Rightarrow r_1) \Rightarrow r_0)$ .

### 7.1 Relating extended designs and timed designs

Timed designs are richer than extended designs and hence we can simulate an extended design $(p \vdash_X r)$ by the timed design in which within $r$ the observation $term'$, representing termination, is replaced by the observation that the final time is finite, i.e., $\tau' \neq \infty$. Hence we can map any extended design into a unique timed design. For any extended design $(p \vdash_X r)$ we have

$$XT(p \vdash_X r) \mathrel{\widehat{=}} (p \vdash_T r \left[\frac{\tau' \neq \infty}{term'}\right]) .$$

It is straightforward to show that $XT$ preserves the refinement ordering, and that the image of this mapping is a subtheory of timed designs.

One can define extreme cases in a similar fashion to those for extended designs, except that $\mathbf{terminates}_T$ and $\mathbf{forever}_T$ make use of $\tau'$ rather than $term'$. We only give the definition of these two:

$$\mathbf{terminates}_T \mathrel{\widehat{=}} (\mathrm{true} \vdash_T \tau' \neq \infty)$$
$$\mathbf{forever}_T \mathrel{\widehat{=}} (\mathrm{true} \vdash_T \tau' = \infty) .$$

Timed designs form a complete lattice under the refinement ordering, with least element $\mathbf{abort}_T$ and greatest element $\mathbf{magic}_T$.

## 8 Timed reactive designs

To model the interactions of a real-time program with its environment, one can use a trace of the values of the program variables over time, i.e., a mapping, $\sigma$, from times to the values of the program variables at those times. As with timed designs, time can either be natural numbers or real numbers, in both cases extended with infinity. The

domain of a trace, $\mathrm{dom}(\sigma)$, never includes the time $\infty$. A program relation then relates an initial trace, $\sigma$, to an extension of that trace $\sigma'$. For a *timed reactive design*, $(q \vdash_R r)$, both $q$ and $r$ are relations between the initial trace, $\sigma$, of the values of the program variables up to the start time of the command, and the final trace $\sigma'$. The start time $\tau$ is then an abbreviation for $\sup(\mathrm{dom}(\sigma))$ and the final time $\tau'$ is an abbreviation for $\sup(\mathrm{dom}(\sigma'))$, where $\sup$ stands for supremum, i.e., least upper bound. For a nonterminating computation, the domain of the final trace $\sigma'$ has no finite bound, and hence $\sup(\mathrm{dom}(\sigma')) = \infty$. A timed reactive design $(q \vdash_R r)$ has a semantics given by the following relation:

$$(ok \wedge \tau \neq \infty \wedge q \Rightarrow ok' \wedge r) \wedge (\tau' = \infty \Rightarrow ok') \wedge \sigma \subseteq \sigma' \ . \tag{14}$$

The significant change from the timed design semantics (10) is that $\tau \leq \tau'$ is replaced by the stronger requirement that $\sigma$ is a prefix of $\sigma'$, i.e., $\sigma \subseteq \sigma'$, which implies $\sup(\mathrm{dom}(\sigma)) \leq \sup(\mathrm{dom}(\sigma'))$, i.e., $\tau \leq \tau'$. The initial state of a timed reactive design corresponds to $\sigma(\tau)$ and the final state (if there is one) to $\sigma'(\tau')$. Note that if $\tau' = \infty$, $\mathrm{dom}(\sigma')$ is the complete range of all finite times, but does not include infinity. Hence we don't need a version of condition (12) in this case.

Another change from the timed design semantics is that precondition $q$ – which specifies the conditions under which the program is guaranteed to not abort – may refer to the final trace $\sigma'$. To illustrate, consider the following timed reactive design interpreted using abstract time (i.e., the domain of $\sigma$ is natural numbers):

$$(\sigma' \neq \sigma \frown \langle x \rangle \vdash_R \text{false}) \ .$$

This may become unstable immediately after it has set the program state to the value $x$ at time $\tau + 1$ .

For the reactive timed design $(q \vdash_R r)$ we impose a condition on $q$ that is analogous to (11) for timed designs:

$$\left[\neg\, q \Rightarrow \left(\forall\, \sigma'' \bullet \sigma' \subset \sigma'' \Rightarrow \neg\, q \left[\frac{\sigma''}{\sigma'}\right]\right)\right] \ . \tag{15}$$

It requires that a reactive design that may abort after behaving like trace $\sigma'$, i.e., if $q$ is false for $\sigma'$, may be implemented by one that aborts at some later time, i.e., $q$ with $\sigma'$ replaced by $\sigma''$ is false for all traces $\sigma''$ that are extensions of $\sigma'$. Note that with $\tau' = \sup(\mathrm{dom}(\sigma'))$ and $\tau'' = \sup(\mathrm{dom}(\sigma''))$, (15) implies (11). The reactive design

$$(\neg\, (\exists\, \sigma'' \bullet x \notin \mathrm{ran}(\sigma'') \wedge \sigma' = \sigma \frown \sigma'') \vdash_R \text{false}) \ ,$$

for instance, does *not* satisfy (15), since it may abort at time $\tau$, but it may not delay the abortion time to time $\tau + 1$ and extend the final trace with a state that takes the value $x$.

*Refinement* of timed reactive designs,

$$(q_0 \vdash_R r_0) \sqsubseteq (q_1 \vdash_R r_1) \ ,$$

is defined in terms of reverse implication of the equivalent semantic relations, and hence holds provided

$$[\tau \neq \infty \wedge \sigma \subseteq \sigma' \wedge q_0 \Rightarrow ((\tau \neq \infty \Rightarrow q_1) \wedge ((q_1 \Rightarrow r_1) \Rightarrow r_0))] \ .$$

This condition is similar to that for timed designs (13), except that the predicates now refer to the initial and final traces, $\sigma$ and $\sigma'$, and the healthiness constraint $\tau \leq \tau'$ is strengthened to ensure that the initial trace is a prefix of the final trace, i.e., $\sigma \subseteq \sigma'$.

For abstract time (natural numbers) the timed reactive model corresponds closely to models based on sequences of states as used in, for example, action systems [32] and TLA [33], while for real-time (real numbers) a timed reactive design corresponds closely to a *real-time specification* as used in the real-time refinement calculus [34, 35, 15, 13]. Hoare and He [1, Chap. 8] introduce reactive processes, which consider traces of events, *tr*. Their processes satisfy the property that a process only ever extends a trace, i.e., $tr \leq tr'$, similar to our constraint on traces of states.

**Relating timed designs and timed reactive designs.** Timed reactive designs generalise timed designs. Each timed design $(q \vdash_T t)$ can be mapped to a unique timed reactive design.

$$TR(q \vdash_T t) \mathrel{\widehat{=}} \left( \begin{pmatrix} \exists \tau, \tau', v \bullet q \wedge \\ \quad \tau = \sup(\mathrm{dom}(\sigma)) \wedge \\ \quad \tau' = \sup(\mathrm{dom}(\sigma')) \wedge \\ \quad v = \sigma(\tau) \end{pmatrix} \vdash_R \begin{pmatrix} \exists \tau, \tau', v, v' \bullet t \wedge \\ \quad \tau = \sup(\mathrm{dom}(\sigma)) \wedge \\ \quad \tau' = \sup(\mathrm{dom}(\sigma')) \wedge \\ \quad v = \sigma(\tau) \wedge \\ \quad (\tau' \neq \infty \Rightarrow \\ \qquad v' = \sigma'(\tau')) \end{pmatrix} \right)$$

This mapping preserves the refinement ordering, and the image of this mapping is a subtheory of timed reactive designs.

One can define extreme cases in a similar fashion to those for extended designs and timed designs, except that $\mathbf{terminates}_R$ and $\mathbf{forever}_R$ make use of $\sigma'$ rather than *term*$'$ or $\tau'$. We only give the definition of these two:

$$\begin{aligned} \mathbf{terminates}_R &\mathrel{\widehat{=}} (\mathrm{true} \vdash_R \sup(\mathrm{dom}(\sigma') \neq \infty) \\ \mathbf{forever}_R &\mathrel{\widehat{=}} (\mathrm{true} \vdash_R \sup(\mathrm{dom}(\sigma') = \infty) \;. \end{aligned}$$

Timed reactive designs form a complete lattice under the refinement ordering, with least element $\mathbf{abort}_R$ and greatest element $\mathbf{magic}_R$ .

## 9 Conclusions

The purpose of this paper has been to help formalise the relationships between a number of different relational models of programs. These relationships are summarised graphically in Figure 1. We introduced extended designs to allow nontermination and abort to be distinguished. This allows both partial and total correctness concerns to be modelled, as well as allowing requirements like "a program must not terminate from certain initial states" to be specified. Z specifications [8, 9], UTP designs [1], VDM pre-post specifications [2], refinement calculus specifications [3, 4], B specifications [7], and general-correctness prescriptions [27] can be seen as special cases of extended designs. Extended designs allow the distinction between an assumption on the initial state and a termination set to be made.

One interesting consequence of formalising the relationships between these models is that it has highlighted the fact that these different approaches to specification use the word "precondition" to mean different things: it can mean an assumption, a termination set, an enabling condition (or guard or test), or combinations of these (as described in Section 1). By embedding all these approaches in the more general extended design model, we can separate out these concepts and hence determine which of them applies in each case. We hope that this better understanding of the relationships between the models and the different interpretations of the meaning of "precondition" will lead to less confusion when comparing or switching between these different models.

Extended designs can be seen as an abstraction of timed designs. In a timed design one can place specific requirements on the final time $\tau'$, whereas in an extended design one can only refer to termination ($term'$), which effectively abstracts all finite restrictions of $\tau'$ in a timed design simply to $\tau' \neq \infty$. Timed designs making use of abstract time are closely related to Hehner's timed models [10, 28, 29].

Timed reactive designs provide a richer model than timed designs, in which initial and final states are replaced by initial and final traces, $\sigma$ and $\sigma'$, where $\sigma$ is a prefix of $\sigma'$. With abstract time this model corresponds to those used for action systems [32] and TLA [33], and with real time to the real-time refinement calculus [13, 15].

In developing the real-time refinement calculus, it was observed that one needed to distinguish abort and nontermination, unlike in existing pre-post specifications in UTP designs, VDM, the refinement calculus, and B. It was in order to reconcile these models that the extended-design model was invented. It generalises the existing pre-post specification models, while simultaneously being a specialisation of both the timed and timed reactive models.

The relationship between the various models is given by the mappings between models. Each downward link in Figure 1 corresponds to a mapping from a sparser model to a richer one. In addition, one can compose these mappings to create a mapping from a model to any richer one that can be reached by following downward links. For example, one can compose the mapping $DX$ from UTP designs to extended designs with the mapping $XT$ from extended designs to timed designs to get a mapping $XT \circ DX$ from designs to timed designs.

Because the mappings between models embed one model as a subtheory of another, this allows properties proved in the richer model, that apply to the elements of the subtheory, to be used in the simpler model. Investigation of these uses of the mappings and extending the mappings to program constructs other than designs are avenues for future research.

**Acknowledgements.** This research was supported, in part, by the EPSRC-funded *Trustworthy Ambient Systems (TrAmS)* Platform Project and Australian Research Council (ARC) Discovery Grants DP0987452 and DP0879529. We would like to thank Brijesh Dongol for feedback on earlier drafts of this paper.

## References

1. Hoare, C.A.R., He Jifeng: Unifying Theories of Programming. Prentice Hall (1998)

2. Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall (1986)
3. Back, R.J.R.: On correct refinement of programs. Journal of Computer and System Sciences **23**(1) (February 1981) 49–68
4. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
5. Morgan, C.C.: The specification statement. ACM Trans. on Prog. Lang. and Sys. **10**(3) (July 1988)
6. Morgan, C.C.: Programming from Specifications. second edn. Prentice Hall (1994)
7. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
8. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, London (1989)
9. Hayes, I.J., ed.: Specification Case Studies. second edn. Prentice Hall (1993)
10. Hehner, E.C.R.: Termination is timing. In van de Snepscheut, J., ed.: Mathematics of Program Construction. Volume 375 of Lecture Notes in Computer Science., Springer (June 1989) 36–47
11. Utting, M., Fidge, C.J.: A real-time refinement calculus that changes only time. In He Jifeng, ed.: Proc. 7th BCS/FACS Refinement Workshop. Electronic Workshops in Computing, Springer (July 1996)
12. Hayes, I.J., Utting, M.: Coercing real-time refinement: A transmitter. In Duke, D.J., Evans, A.S., eds.: BCS-FACS Northern Formal Methods Workshop (NFMW'96). Electronic Workshops in Computing, Springer (1997)
13. Hayes, I.J., Utting, M.: A sequential real-time refinement calculus. Acta Informatica **37**(6) (2001) 385–448
14. Hayes, I.J.: A predicative semantics for real-time refinement. In McIver, A., Morgan, C.C., eds.: Programming Methodology. Springer Verlag (2003) 109–133
15. Hayes, I.J.: Reasoning about real-time repetitions: Terminating and nonterminating. Science of Computer Programming **43**(2–3) (2002) 161–192
16. Derrick, J., Boiten, E.: Refinement in Z and Object-Z. Springer (2001)
17. Duke, R., Rose, G., Smith, G.: Object-Z: A specification language advocated for the description of standards. Computer Standards and Interfaces **17** (1995)
18. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers (2000)
19. Parnas, D.L.: A generalized control structure and its formal definition. Commun. ACM **26**(8) (1983) 572–581
20. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1) (1995) 41–61
21. Dijkstra, E.W.: Guarded commands, nondeterminacy, and a formal derivation of programs. CACM **18** (1975) 453–458
22. Jacobs, D., Gries, D.: General correctness: a unification of partial and total correctness. Acta Informatica **22** (1985) 67–83
23. Nelson, G.: A generalisation of Dijkstra's calculus. ACM Trans. on Prog. Lang. and Sys. **11**(4) (1989)
24. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag (1990)
25. Dunne, S.E., Stoddart, W.J., Galloway, A.J.: Specification and refinement in general correctness. In Evans, A., Duke, D., Clark, A., eds.: Proceedings of the 3rd Northern Formal Methods Workshop, BCS Electronic Workshops in Computing (1998)
26. Dunne, S.E.: Abstract commands: a uniform notation for specifications and implementations. In Fidge, C., ed.: Computing: The Australasian Theory Symposium (CATS 2001). Volume 42 of Electronic Notes in Theoretical Computer Science., Elsevier Science BV (2001) 104–123
27. Dunne, S.E.: Recasting Hoare and He's unifying theory of programs in the context of general correctness. In Butterfield, A., Strong, G., Pahl, C., eds.: Proceedings of the 5th Irish Workshop in Formal Methods, IWFM 2001. Workshops in Computing, British Computer Society (2001)

28. Hehner, E.C.R.: Abstractions of time. In Roscoe, A., ed.: A Classical Mind. Prentice Hall (1994) 191–210
29. Hehner, E.C.R.: Retrospective and prospective for unifying theories of programming. In Dunne, S., Stoddart, B., eds.: UTP. Volume 4010 of Lecture Notes in Computer Science., Springer (2006) 1–17
30. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Trans. on Prog. Lang. and Sys. **16**(5) (September 1994) 1543–1571
31. Fidge, C.J., Hayes, I.J., Watson, G.: The deadline command. IEE Proceedings—Software **146**(2) (April 1999) 104–111
32. Back, R.J., von Wright, J.: Trace refinement of action systems. In Jonsson, B., Parrow, J., eds.: Proc. of CONCUR'94: Concurrency Theory. Volume 836 of LNCS., Springer-Verlag (1994) 367–384
33. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison Wesley (2003)
34. Hayes, I.J.: Procedures and parameters in the real-time program refinement calculus. Science of Computer Programming **64**(3) (February 2007) 286–311
35. Hayes, I.J.: Termination of real-time programs: definitely, definitely not or maybe. In Dunne, S.E., Stoddart, W.J., eds.: UTP 2006: First Int. Symp. on Unifying Theories of Programming. Volume 4010 of LNCS., Springer Verlag (2006) 141–154