Specification of Required Non-determinism

J Fiadeiro, A Lopes
Dept. of Informatics, University of Lisbon, Campo Grande, 1700 Lisbon.
K Lano, J Bicarregui
Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ

Abstract. We present an approach to the specification of *required external non-determinism*: the willingness of a component to respond to a number of external action requests, using a language, COMMUNITY, which provides both permission and willingness guards on actions.

This enables a program-like declaration of required non-determinism, in contrast to the use of a branching-time temporal logic. We give a definition of parallel composition for this language, and show that refinement is compositional with respect to parallel composition. We use the concepts developed for COMMUNITY to identify extensions to the B and VDM^{++} model-based specification languages to incorporate specification of required non-determinism. In particular, we show that preconditions may be considered as a form of willingness guard, separating concerns of acceptance and termination, once module contracts are re-interpreted in a way suitable for a concurrent environment.

1 Introduction

Non-determinism is usually regarded as an aspect of abstract specification which is to be eliminated during the refinement process. Indeed this view equates non-determinism in specifications with *under-specification*: the incomplete description of a value or operation which leaves open the choice of several deterministic implementations[1]. For example, in B [1] we could write:

CONSTANTS **ff**
PROPERTIES
$\quad \mathbf{ff} \in \mathbb{N} \rightarrow \mathbb{N} \ \wedge$
$\quad \forall\ \mathbf{xx}.(\mathbf{xx} \in \mathbb{N} \ \Rightarrow\ \mathbf{ff}(\mathbf{xx}+1) > \mathbf{ff}(\mathbf{xx}))$

This is an underspecified description of **ff** – we know that **ff** must be strictly increasing, but no other constraint is provided. An implementation of this function **ff** must be deterministic – one possible choice would be the successor function on $\mathbb{N}$.

Thus if we had defined an operation using **ff**:

$\mathbf{yy} \longleftarrow \mathbf{op}(\mathbf{xx}) =$
$\quad$ PRE $\mathbf{xx} \in \mathbb{N}$
$\quad$ THEN

[1] Both B and VDM-SL take the interpretation that loosely-specified values are actually deterministic, just unknown, whilst operations may be internally non-deterministic.

```
    yy := ff(xx)
  END
```

we would expect the same result from calling **op** with a particular argument value **xx** each time this call is made.

In contrast an operation specified as

```
yy ⟵ random =
  ANY vv
  WHERE vv ∈ ℕ
  THEN yy := vv
  END
```

could, in principle, be implemented in a non-deterministic manner: successive calls of **random** could yield different elements of $\mathbb{N}$ as their results.

An example where such genuinely non-deterministic operation implementations are useful is a random number generator [9]. Applications in the field of security – where it is important that some clients of an operation cannot use its result to deduce certain secure information – also arise. However, the B language, or similar model-based languages such as VDM or Z cannot be used for such specification: the operation **random** above can validly be implemented by an operation which always returns the answer 5, for instance.

The property that all possible non-determinism in the effects of an operation is actually observable will be termed *required internal non-determinism* in this paper. Our main focus will be on a related form, termed *required external non-determinism*. This refers to the willingness of a component to answer a range of operation requests at a given time. This is particularly important in a concurrent execution context, where we need some guarantees that a parallel composition of two components, where one requests services from another, will not deadlock.

The use of *permission* guards for operations has become a common mechanism for concurrent object-based and object-oriented languages [5, 8]. A permission guard **G** for an operation **op** of a server object **obj** expresses that **obj** will refuse to provide the service **op** to external callers unless **G** holds. The caller will be "blocked", ie, suspended in its thread of execution, if it attempts to call **op** at a time when **G** does not hold, and will only be freed to complete the call if **obj** changes state (as a result of other calls from other clients) so that **G** becomes true.

Such guards allow a passive shared server object to protect its internal state. For example, a buffer with several clients would need to block clients that wish to execute a **get** method until there are some elements in the buffer. In VDM^{++} notation [10] this could be specified as:

```
class Buffer
instance variables
  contents : ℕ*;
init objectstate == contents := []
methods
  put(x : ℕ) ==
      contents := contents ⌢ [x];
```

```
  get() value y : ℕ ==
      (y := hd(contents);
       contents := tl(contents);
       return y)
sync
  per get   ⇒   len(contents) > 0
end Buffer
```

The permission guard $\mathbf{len}(\mathbf{contents}) > 0$ for **get** in the **sync**hronisation clause asserts that **get** can only initiate execution if this condition holds.

Permission guards are used rather than preconditions because the presence of concurrency requires a change in the usual interpretation of module contracts [5, Chapter 11]: instead of producing an arbitrary result or behaviour if it is called outside the stated assumptions of its contact, a supplier operation such as **get** must suspend the client until the operation assumptions hold.

In terms of the semantics of classes [10], a permission guard **G** for **op** must be true at each time $\uparrow(\mathbf{op}, \mathbf{i})$ which is the initiation of the **i**-th invocation of **op**:

$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{G} \circledcirc \uparrow(\mathbf{op}, \mathbf{i})$$

$\varphi \circledcirc \mathbf{t}$ asserts that φ holds at time **t**.

Hence, permission guards can be *strengthened* as development or specialisation proceeds. This ensures internal consistency, at the risk of system deadlock. In terms of theory extension, this is valid, as subtypes or refinements will then have stronger theories (the $\circledcirc$ operator is monotonic in the RAL formalism [10] used as a semantics for VDM^{++}). This is not totally satisfactory however, as it allows a class to be "implemented" by a class with **false** permission guards for each of its methods, ie, whose objects refuse to execute any methods. The alternative, to leave these guards essentially unchanged through refinement [3], is not adequately flexible if we wish to combine subtyping and synchronisation [12].

We propose therefore a means of specifying an upper bound on the strength of permission guards, at specification time, via the use of "willingness" guards which provide a guarantee that implementations of a class will answer requests for services under certain conditions. The willingness guards imply the permission guards (if an object is willing to execute a method, then certainly it must permit itself to do so) and may be *weakened* during refinement. We have the situation shown in Figure 1: the grey area, where the permission guard is true but the willingness guard is not, may be eliminated during refinement or specialisation. It represents a form of under-specification whereby it is not known whether an object of the class will accept or refuse a request for the particular service under these conditions.

Such predicates can also be of use in B. A frequent style of abstract specification in B is to leave unspecified how a choice between error and normal behaviour is to be made:

```
  add_data(dd) =
```

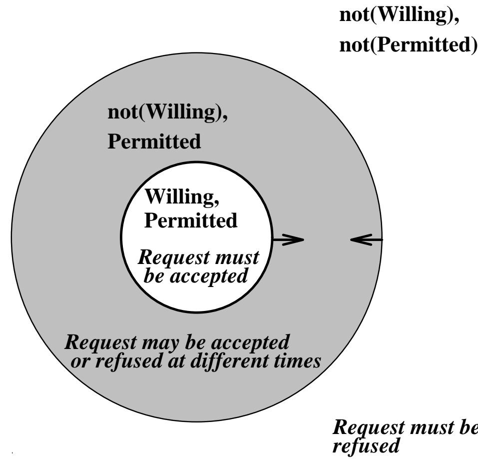

Fig. 1. Permission and Willingness Conditions

```
PRE dd ∈ Data
THEN
    CHOICE
        memory := memory ⌢ [dd]
    OR
        SKIP
    END
 END
```

In this case the SKIP statement is meant only to be executed if the data cannot be stored because memory is exhausted – however in principle a developer could implement the entire operation by a SKIP. Using permission and willingness guards we could express precisely under what conditions the addition of data is *guaranteed* to be carried out by an implementation (perhaps if memory usage is below 70% of capacity), under what conditions it *cannot* be performed (eg., if memory capacity has been reached), and (the remaining states) when we are uncertain as to the behaviour of the eventual implementation (the request may be refused or accepted). Similar examples concern the withdrawal of money from a bank account, etc.

In Sections 2 and 3 we describe the COMMUNITY language, its semantics and relation to branching temporal logic. We give the definitions of refinement and parallel composition in this language, and show that refinement is compositional with respect to parallel composition. Section 4 identifies suitable extensions of the B language to cover concurrent specification with required non-determinism. Section 5 considers the extension of VDM^{++} with specification of required non-determinism.

2 COMMUNITY

COMMUNITY [7] is based on the UNITY language of [4]. It extends UNITY with the specification of *willingness* guards in addition to the usual permission guards. A COMMUNITY program is a pair (Θ, Δ) where Θ is a *program signature*: a pair $(\mathbf{V}, \Gamma)$ of sets of attribute symbols $\mathbf{V}$ and action symbols Γ. Each action symbol $\mathbf{g} \in \Gamma$ has a *write frame* (the set of attribute symbols that it may change) $\mathbf{D}(\mathbf{g}) \subseteq \mathbf{V}$.

The second component Δ of a program is a tuple $(\mathbf{I}, \mathbf{F}, \mathbf{P}, \mathbf{W})$ where $\mathbf{I}$ is an initialisation for the program (module): a predicate over the attributes $\mathbf{V}$. $\mathbf{F}$ gives for each action $\mathbf{g}$ and attribute $\mathbf{a}$ an effect statement $\mathbf{F}(\mathbf{g}, \mathbf{a})$ which describes how $\mathbf{a}$ is changed by $\mathbf{g}$. $\mathbf{P}$ gives for each action a permission guard $\mathbf{P}(\mathbf{g})$: when $\mathbf{P}(\mathbf{g})$ is false, $\mathbf{g}$ cannot be executed. $\mathbf{W}$ gives a willingness guard $\mathbf{W}(\mathbf{g})$ for each action: when $\mathbf{W}(\mathbf{g})$ holds the program is willing to execute $\mathbf{g}$ if the environment requests it.

We can also attach an invariant predicate to a program. The program will be *internally consistent* if the initialisation implies the invariant and the invariant is preserved by the execution of each possible combination of actions.

As a simple example, consider the task of maintaining a bank account. This has an invariant that the account **balance** is always above the overdraft **limit**, and actions to deposit and withdraw money:

var
  $\mathbf{balance} : \mathbb{Z};$
  $\mathbf{limit} : \mathbb{Z}$
initialisation
  $\mathbf{balance} = 0 \wedge \mathbf{balance} \geq \mathbf{limit}$
invariant
  $\mathbf{balance} \geq \mathbf{limit} \wedge 0 \geq \mathbf{limit}$
do
  $\mathbf{deposit}(\mathbf{amt} : \mathbb{N}) : [\mathbf{true}, \mathbf{true} \rightarrow \mathbf{balance} := \mathbf{balance} + \mathbf{amt}] \;[]$
  $\mathbf{withdraw}(\mathbf{amt} : \mathbb{N}) : [\mathbf{balance} - \mathbf{amt} \geq \mathbf{limit},$
    $\mathbf{balance} - \mathbf{amt} \geq 0 \rightarrow$
      $\mathbf{balance} := \mathbf{balance} - \mathbf{amt}]$

An action specification has the form $\mathbf{g}(\mathbf{p}) : [\mathbf{P}(\mathbf{g}), \mathbf{W}(\mathbf{g}) \rightarrow \mathbf{F}(\mathbf{g})]$: the permission guard $\mathbf{P}(\mathbf{g})$ is written before the willingness guard $\mathbf{W}(\mathbf{g})$. $\mathbf{F}(\mathbf{g})$ is the effect statement of the action, using the abstract generalised substitution notation of B. The write frame $\mathbf{D}(\mathbf{g})$ is calculated as the set of attribute symbols occurring as the target of assignments in $\mathbf{F}(\mathbf{g})$.

The above specification asserts that **deposit** is always available for clients. However the **withdraw** action will definitely not be accepted for execution if $\mathbf{balance} - \mathbf{amt} < \mathbf{limit}$: the permission guard for **withdraw** is false. It is guaranteed to be accepted if $\mathbf{balance} - \mathbf{amt} \geq 0$: the willingness guard for **withdraw** is true. In other cases the bank may use its discretion in accepting or rejecting the request.

In general, a COMMUNITY program may execute several actions in the same time interval (but the effects of such actions must not conflict – so $\mathbf{deposit}(\mathbf{x})$

cannot occur with $\mathbf{withdraw}(\mathbf{x})$ unless $\mathbf{x} = 0$). Over a time interval, a given attribute **att** may only change its value if there is some action $\mathbf{g}$ executing in that interval with **att** in the write frame of $\mathbf{g}$.

We can define a semantics $\models$ for programs using transition systems termed *Θ-interpretation structures* (see Appendix A).

Definition A program $\mathbf{P} = (\Theta, \Delta)$ is *realisable* iff (i) $\mathbf{I}$ is satisfiable, and (ii) every Θ-interpretation structure $\mathbf{S}$ that satisfies conditions 1 – 4 of the definition of model has that $\mathbf{S} \models \mathbf{W}(\mathbf{g}) \Rightarrow \mathbf{P}(\mathbf{g})$ for every $\mathbf{g} \in \Gamma$.

A realisable program that has a model of the permission and functionality constraints also has a model of its willingness constraints:

Proposition If $\mathbf{P}$ is realisable, then every Θ-interpretation structure $\mathbf{S}_1$ that satisfies conditions 1 – 4 has an extension (ie, with more states and transitions) $\mathbf{S}_2$ which is a Θ-interpretation structure that is a model of $\mathbf{P}$.

Usually we will write programs with $\mathbf{W}(\mathbf{g})$ implying $\mathbf{P}(\mathbf{g})$ in any case. Henceforth in this paper we will omit the part of $\mathbf{W}(\mathbf{g})$ which simply repeats $\mathbf{P}(\mathbf{g})$ and only explicitly write the additional conditions.

2.1 Parallel Composition

Program signatures and morphisms define a category $\mathcal{SIG}$:

Definition Given program signatures $\Theta_1 = (\mathbf{V}_1, \Gamma_1)$ and $\Theta_2 = (\mathbf{V}_2, \Gamma_2)$, a *signature morphism* σ from Θ_1 to Θ_2 consists of a pair $(\sigma_\alpha : \mathbf{V}_1 \rightarrow \mathbf{V}_2, \sigma_\gamma : \Gamma_1 \rightarrow \Gamma_2)$ of functions such that for every $\mathbf{a} \in \mathbf{V}_1$:

$$\{\mathbf{g} \in \Gamma_2 \mid \sigma(\mathbf{a}) \in \mathbf{D}_2(\mathbf{g})\} \;\subseteq\; \sigma(\!|\; \{\mathbf{g} \in \Gamma_1 \mid \mathbf{a} \in \mathbf{D}_1(\mathbf{g})\} \;|\!)$$

Given a signature morphism σ we can translate a predicate φ in the language of Θ_1 into a predicate $\sigma(\varphi)$ in the language of Θ_2 by applying σ to all the attribute and action symbols of φ.

Definition A *superposition* (component-of) morphism $\sigma : (\Theta_1, \Delta_1) \rightarrow (\Theta_2, \Delta_2)$ is a signature morphism $\sigma : \Theta_1 \rightarrow \Theta_2$ such that:

1. for all $\mathbf{g}_1 \in \Gamma_1$, $\mathbf{a}_1 \in \mathbf{D}_1(\mathbf{g}_1)$,

   $$\models_2 \; \mathbf{P}_2(\sigma(\mathbf{g}_1)) \;\Rightarrow\; \mathbf{F}_2(\sigma(\mathbf{g}_1), \sigma(\mathbf{a}_1)) = \sigma(\mathbf{F}_1(\mathbf{g}_1, \mathbf{a}_1))$$

   in the case the effects are assignments $\mathbf{a_i} := \mathbf{F_i}(\mathbf{g_i}, \mathbf{a_i})$; other cases are similar.
2. $\models_2 \mathbf{I}_2 \Rightarrow \sigma(\mathbf{I}_1)$
3. for every $\mathbf{g}_1 \in \Gamma_1$, $\models_2 \; \mathbf{P}_2(\sigma(\mathbf{g}_1)) \Rightarrow \sigma(\mathbf{P}_1(\mathbf{g}_1))$
4. for every $\mathbf{g}_1 \in \Gamma_1$, $\models_2 \; \mathbf{W}_2(\sigma(\mathbf{g}_1)) \Rightarrow \sigma(\mathbf{W}_1(\mathbf{g}_1))$.

Programs and superposition morphisms constitute a finitely co-complete category **c**-$\mathcal{PROG}$ [11].

We can define interconnections between COMMUNITY programs using these morphisms and *channels*, which are programs with a single action $\mathbf{c} : [\mathbf{true}, \mathbf{true} \rightarrow \mathbf{skip}]$ which allow synchronisation of actions of other programs (see Figure 2). The co-limit of such a diagram of programs $\{\mathbf{P_i} : \mathbf{i} \in \mathbf{Ind}\}$ is a program $|| \{\mathbf{P_i} : \mathbf{i} \in \mathbf{Ind}\}$ defined as follows:

1. Its actions are all those actions from any $\mathbf{P_i}$ which are not synchronised by any channel between programs, together with all the composite actions induced by these synchronisations
2. Its initialisation is the conjunction of the initialisations of the $\mathbf{P_i}$
3. The write frame of a composite action is the union of the write frame of its parts
4. The effect of a composite action is the $||$ combination (in the sense of B) of the individual effects
5. The permission guard of a composite action is the conjunction of the individual permission guards (compare with the definition of $||$ for actions in [3])
6. The willingness guard of a composite action is the conjunction of the individual willingness guards.

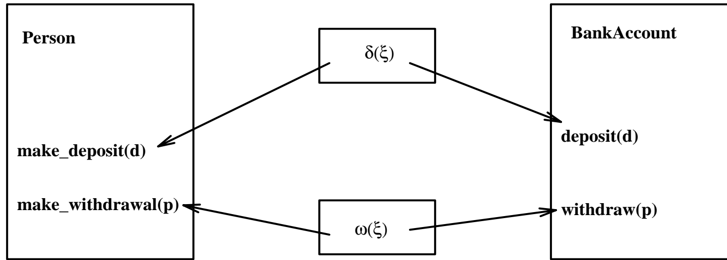

Fig. 2. Concurrent Composition via Synchronisation

This composition is similar to the synchronised interleaving of traces in VDM^{++} [10], or the $||$ operator of CSP. A parallel composition should only be willing to perform a composite action **g** if every component involved in the execution of **g** is willing to accept it for execution: hence the use of $\wedge$ in the final clause of the above definition.

Channels with attributes can be used to represent the form of $||$ composition of actions with input and output parameters defined in [3].

In the example of Figure 2, if **Person** has the form:

var
  cash : $\mathbb{N}$
do
 $\textbf{make_deposit}(\textbf{amt} : \mathbb{N}) : \ [\textbf{amt} > 0 \wedge \textbf{amt} \leq \textbf{cash}, \textbf{true} \rightarrow \textbf{cash} := \textbf{cash} - \textbf{amt}]$ []
 $\textbf{make_withdrawal}(\textbf{amt} : \mathbb{N}) : \ [\textbf{amt} > 0, \textbf{true} \rightarrow \textbf{cash} := \textbf{cash} + \textbf{amt}]$

(that is, the willingness guards for the two actions are equivalent to the permission guards), then the concurrent composition of the two modules has two parameterised actions $\textbf{dep}(\mathbf{x} : \mathbb{N})$ which is the composite

$$\textbf{make_deposit}(\mathbf{x}) \,||\, \delta(\mathbf{x}) \,||\, \textbf{deposit}(\mathbf{x})$$

and $\textbf{with}(\mathbf{x} : \textbf{nat})$ which is the composite

$$\textbf{make_withdrawal}(\mathbf{x}) \,||\, \omega(\mathbf{x}) \,||\, \textbf{withdraw}(\mathbf{x})$$

The first has the derived definition

$$\textbf{dep}(\textbf{amt} : \mathbb{N}) : \ [\textbf{amt} > 0 \wedge \textbf{amt} \leq \textbf{cash}, \textbf{true} \rightarrow \textbf{cash} := \textbf{cash} - \textbf{amt} \,||\, \textbf{balance} := \textbf{balance} + \textbf{amt}]$$

and the second the definition

$$\textbf{with}(\textbf{amt} : \mathbb{N}) : \ [\textbf{amt} > 0 \wedge \textbf{balance} - \textbf{amt} \geq \textbf{limit}, \textbf{balance} - \textbf{amt} \geq 0 \rightarrow \textbf{cash} := \textbf{cash} + \textbf{amt} \,||\, \textbf{balance} := \textbf{balance} - \textbf{amt}]$$

2.2 Subtyping and Refinement

We will define subtyping of COMMUNITY programs in terms of morphisms which allow permission guards to be *strengthened* and willingness guards to be *weakened* (see Figure 1). This means that the degree of uncertainty about the behaviour of a subtype object will be lower than for a supertype object – it may both definitely refuse more requests and definitely accept more requests.

Definition A *subtyping morphism* $\sigma : (\Theta_1, \Delta_1) \rightarrow (\Theta_2, \Delta_2)$ is a signature morphism $\sigma : \Theta_1 \rightarrow \Theta_2$ such that:

1. for all $\mathbf{g}_1 \in \Gamma_1$, $\mathbf{a}_1 \in \mathbf{D}_1(\mathbf{g}_1)$,
$$\models_2 \ \mathbf{P}_2(\sigma(\mathbf{g}_1)) \ \Rightarrow \ \mathbf{F}_2(\sigma(\mathbf{g}_1), \sigma(\mathbf{a}_1)) = \sigma(\mathbf{F}_1(\mathbf{g}_1, \mathbf{a}_1))$$
2. $\models_2 \mathbf{I}_2 \Rightarrow \sigma(\mathbf{I}_1)$
3. for every $\mathbf{g}_1 \in \Gamma_1$, $\models_2 \ \mathbf{P}_2(\sigma(\mathbf{g}_1)) \Rightarrow \sigma(\mathbf{P}_1(\mathbf{g}_1))$
4. for every $\mathbf{g}_1 \in \Gamma_1$, $\models_2 \ \sigma(\mathbf{W}_1(\mathbf{g}_1)) \Rightarrow \mathbf{W}_2(\sigma(\mathbf{g}_1))$.

Programs and subtyping morphisms constitute a category $\mathbf{s}\text{-}\mathcal{PROG}$. σ is a *refinement* morphism if σ_γ is surjective: ie., no new external actions are introduced.

A central result is that refinement is compositional with respect to parallel composition:

Proposition If there are subtyping morphisms $\eta_1 : \mathbf{P}_1 \to \mathbf{P}'_1$ and $\eta_2 : \mathbf{P}_2 \to \mathbf{P}'_2$ then there is a unique subtyping morphism $\eta : \mathbf{P}_1 \parallel \mathbf{P}_2 \to \mathbf{P}'_1 \parallel \mathbf{P}'_2$ for any parallel composition of $\mathbf{P}_1$ and $\mathbf{P}_2$ (and corresponding composition of their subtypes or refinements).

The same applies with regard to refinement morphisms.

Informally this is clear because parallel composition is a monotonic operator in terms of the logical and functional elements of its components. The full proof is given in [11].

3 Temporal Logic Specification of Required Non-Determinism

We can more abstractly and generally specify the required availability of actions by using a *branching time* temporal logic [13]. Specifically we will use the CTL* language which contains a branch quantifier $\mathbb{E}\varphi$ "on some path φ holds" and the derived $\mathbb{A}\varphi$ quantifier "on all paths φ holds".

Definition The computational tree logic institution CTL* is defined as follows:

- Its category of signatures is $\mathcal{SET} \times \mathcal{SET}$
- The grammar functor defines, for every signature $\Theta = (\mathbf{V}, \Gamma)$, the set of *state formulas* $\mathbf{CTL}^*(\Theta)$:

$$\phi_{\mathbf{S}} ::= \mathbf{a} \mid \neg\, \phi_{\mathbf{S}} \mid \phi_{\mathbf{S}} \Rightarrow \psi_{\mathbf{S}} \mid \mathbf{beg} \mid \mathbb{E}\phi_{\mathbf{P}}$$

  and the set $\mathbf{CTL}^*_{\mathbf{P}}(\Theta)$ of *path formulas*:

$$\begin{aligned}\phi_{\mathbf{P}} ::= &\ \mathbf{g} \mid \phi_{\mathbf{S}} \mid \neg\, \phi_{\mathbf{P}} \mid \phi_{\mathbf{P}} \Rightarrow \psi_{\mathbf{P}} \\ \mid &\ \bigcirc\phi_{\mathbf{P}} \mid \phi_{\mathbf{P}} \mathcal{U} \psi_{\mathbf{P}}\end{aligned}$$

The specification of the bank account given in Section 2 can be alternatively presented as a $\mathbf{CTL}^*$ theory with data and initialisation axioms:

$$\begin{aligned}&\mathbf{balance} \in \mathbb{Z} \\ &\mathbf{limit} \in \mathbb{Z} \\ &\mathbf{beg} \ \Rightarrow\ \mathbf{balance} = 0 \wedge \mathbf{balance} \geq \mathbf{limit} \\ &\mathbf{balance} \geq \mathbf{limit}\end{aligned}$$

locality axioms:

$$\begin{aligned}&\bigcirc\mathbf{limit} \ = \ \mathbf{limit} \\ &\bigcirc\mathbf{balance} \ = \ \mathbf{balance} \ \vee\ \exists\,\mathbf{amt} : \mathbb{N} \cdot \mathbf{deposit(amt)} \ \vee \\ &\qquad\quad \exists\,\mathbf{amt} : \mathbb{N} \cdot \mathbf{withdraw(amt)}\end{aligned}$$

permission/effect axioms:

$$\forall\, \mathbf{amt} : \mathbb{N} \cdot \mathbf{deposit}(\mathbf{amt}) \;\Rightarrow\; \bigcirc \mathbf{balance} \;=\; \mathbf{balance} + \mathbf{amt}$$

$$\forall\, \mathbf{amt} : \mathbb{N} \cdot \mathbf{withdraw}(\mathbf{amt}) \;\Rightarrow\; \mathbf{balance} - \mathbf{amt} \geq \mathbf{limit} \;\wedge\; \bigcirc \mathbf{balance} = \mathbf{balance} - \mathbf{amt}$$

and willingness axioms:

$$\forall\, \mathbf{amt}_1, \mathbf{amt}_2 : \mathbb{N} \cdot \mathbb{E}(\neg\, \mathbf{deposit}(\mathbf{amt}_1) \wedge \neg\, \mathbf{withdraw}(\mathbf{amt}_2))$$

$$\mathbf{balance} \geq 0 \;\Rightarrow\; \mathbb{E}(\mathbf{deposit}(0) \wedge \mathbf{withdraw}(0))$$

$$\forall\, \mathbf{amt}_1, \mathbf{amt}_2 : \mathbb{N} \cdot \mathbf{balance} - \mathbf{amt}_2 \geq 0 \;\Rightarrow\; \mathbb{E}(\mathbf{withdraw}(\mathbf{amt}_2) \wedge \neg\, \mathbf{deposit}(\mathbf{amt}_1))$$

$$\forall\, \mathbf{amt}_1, \mathbf{amt}_2 : \mathbb{N} \cdot \mathbb{E}(\mathbf{deposit}(\mathbf{amt}_2) \wedge \neg\, \mathbf{withdraw}(\mathbf{amt}_1))$$

There are concepts of morphism and refinement for such theories, and a mapping from programs to theories: the above theory is an example of application of this mapping.

Notice that the only essential path quantifier needed to represent the semantics of programs is of the form $\mathbb{E}\gamma$ where γ is a set of actions. We could therefore work in a LTL language extended with formulae $\mathbf{enabled}(\gamma)$ representing such path constraints.

4 Extending B with Required Non-Determinism

A model of action-based concurrency for B has been developed in [3]. In this approach the operations of B machines are viewed as actions similar to those of COMMUNITY programs, and operations of different machines may be synchronised under certain conditions. This language already possesses a form of *permission* guard, since the SELECT **G** statement of B may be interpreted as asserting that "there are no possible executions unless **G** holds". However it has no separate willingness guard – effectively this is taken to be equivalent to the permission guard since the permission guard cannot be essentially strengthened during refinement.

The wp semantics $[\mathbf{S}]\mathbf{P}$ of a statement **S** gives a predicate **Q** for which every execution of **S** started from a state satisfying **Q**, will result in a post-state satisfying **P**. In the case of SELECT we have

$$[\textsc{select}\ \mathbf{G}\ \textsc{then}\ \mathbf{S}\ \textsc{end}]\mathbf{P} \;=\; \mathbf{G} \Rightarrow [\mathbf{S}]\mathbf{P}$$

In the case that **G** does not hold initially, this says that every execution of the SELECT will achieve **P**, even if **P** is **false**. This can only be true if "every" is a null quantifier, ie, there are no executions of the statement if **G** fails.

However in B there is no explicit willingness guard, so that it is always possible to refine a system by strengthening permission guards to **false**, since

$$(\mathbf{G}_1 \Rightarrow \mathbf{G}_2) \;\Rightarrow\; \textsc{select}\ \mathbf{G}_2\ \textsc{then}\ \mathbf{S}\ \textsc{end} \;\sqsubseteq\; \textsc{select}\ \mathbf{G}_1\ \textsc{then}\ \mathbf{S}\ \textsc{end}$$

where $\sqsubseteq$ is the refinement relation between substitutions.

We can however interpret *preconditions* as a form of willingness guard. Consider the usual "design by contract" meaning of a precondition **P** of an operation **op**. This asserts that if **P** holds when an attempt is made to execute **op**, then:

1. execution of **op** will be accepted
2. execution of **op** will terminate in a valid state, as specified by the postcondition.

In a sequential environment the first property is assumed to always hold, so the focus is on the guarantee of termination. Nevertheless, since

$$\textsc{pre}\ \mathbf{G}\ \textsc{then}\ \mathbf{S}\ \textsc{end}\ \sqsubseteq\ \textsc{select}\ \mathbf{G}\ \textsc{then}\ \mathbf{S}\ \textsc{end}$$

one possible implementation of the preconditioned substitution is the corresponding guarded command, and therefore one possible behaviour of the operation outside its precondition is a refusal to execute.

We can reinterpret the design by contract use of the precondition to represent just the first kind of guarantee to the environment – that execution of the operation will be accepted. Proof of termination will be performed separately.

Given this interpretation, we can write the bank account example of Section 1 as:

```
MACHINE Account
CONSTANTS limit
PROPERTIES limit ∈ ℤ ∧ limit ≤ 0
VARIABLES balance
INVARIANT
    balance ∈ ℤ ∧ balance ≥ limit
INITIALISATION balance := 0
OPERATIONS
    deposit(amt) =
        PRE amt ∈ ℕ
        THEN
            balance := balance + amt
        END;

     withdraw(amt) =
        SELECT amt ∈ ℕ  ∧  balance − amt ≥ limit
        PRE balance − amt ≥ 0
        THEN
            balance := balance − amt
        END

END
```

A "double guard" statement SELECT **P(op)** PRE **W(g)** THEN **S** END is used in the definition of **withdraw**. This extends the suggestion of [2] that guards are made "first-class citizens". We can see the two guards as successive filters – if the

first guard fails then execution of the operation is not *permitted*, so the second guard need not be tested. If the first guard is passed successfully then the second guard is tested to see if acceptance of this execution is *obliged*.

A similar approach works for the memory management example. We can relate this modified B to COMMUNITY as follows.

4.1 Relationship of B to COMMUNITY

Taking the above interpretation of guards and preconditions, B can be interpreted/implemented in COMMUNITY, provided a facility for hiding data and actions was added to COMMUNITY. The results of Section 3 show that a definition of parallel composition can be given for this extended language which is monotonic with respect to refinement.

A specification-level B machine can be viewed as a COMMUNITY module:

1. Machine parameters can be defined as variables in a module which is then made a component (via a superposition morphism) of the machine module;
2. Machine constants and variables can be defined as attributes of the machine module, with their properties and invariant expressed as an invariant;
3. The machine initialisation $\mathbf{T}$ can be re-expressed as a predicate $(\neg\, [\mathbf{T}](\mathbf{x} \neq \mathbf{x}'))[\mathbf{x}/\mathbf{x}']$ in the initialisation of the machine module;
4. Operations $\mathbf{y} \longleftarrow \mathbf{op}(\mathbf{x}) \;=\; \text{SELECT } \mathbf{P} \text{ PRE } \mathbf{W} \text{ THEN } \mathbf{S} \text{ END}$ can be expressed as actions

   $$\mathbf{op}(\mathbf{x}) : [\mathbf{P}, \mathbf{W} \;\rightarrow\; \mathbf{S}]$$

   where the output parameters $\mathbf{y}$ have been converted into new attributes (unique to $\mathbf{op}$).

The locality axiom is true for modules derived from B machines, because B allows variables to be modified only via operations of the machine in which this data is declared. Additionally, at most one of the operations declared in a particular machine can execute at any time – this must be expressed by a specific axiom in the translated module.

Machine inclusion mechanisms can be expressed in terms of colimits wrt suitable superposition morphisms:

1. If machine $\mathbf{B}$ SEES machine $\mathbf{A}$, then we can express the meaning of $\mathbf{B}$ as the colimit of a diagram (Figure 3) where $\mathbf{B}'$ is $\mathbf{B}$ with the seen data $\mathbf{x}$ renamed to $\mathbf{y}'$, and $\mathbf{C}$ just contains the seen data (therefore, this data must be constant). σ' is the identity morphism;
2. If machine $\mathbf{B}$ USES machine $\mathbf{A}$, the same construction can be used, however now $\mathbf{B}$ may refer to the shared data in its invariant;

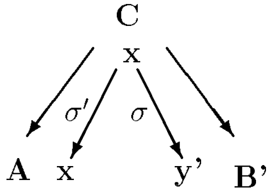

Fig. 3. Diagram of Modules for SEES

3. If machine **B** INCLUDES machine **A**, then **C** will have an action symbol α for every pair of operations **op** of **B** and $\mathbf{op}_1$ of **A** which is called by **op**. For example, if **op** was defined in **B** as

   $$\mathbf{op} \quad = \quad \text{PRE } \mathbf{P} \text{ THEN } \mathbf{S} \;||\; \mathbf{op}_1 \text{ END}$$

   then the two superposition morphisms would map α to **op** and α to **op**$'$. In **B**$'$ **op** is defined without any subordinate calls.

   More complex cases of calling, involving conditional behaviour, can also be expressed via such unifying theories.

   C must also contain a copy of all the data of **A**, and use this copy to identify the data of **A** with the included version of this data in **B**$'$.

   The operations included from **A** must be hidden in the co-limit, except for PROMOTEd operations.

Internal consistency conditions for a machine now include additionally that the explicit willingness guards given in the PRE clause, together with the permission guards given in the SELECT clause, ensure the termination of the body of the operation.

5 Extending VDM^{++}

VDM^{++} is an extension of the VDM-SL notation to cover object-oriented structuring, concurrency and real-time specification elements.

We noted in Section 1 that the absence of willingness guards in VDM^{++} leads to the danger of introducing deadlock during refinement, because permision guards may be arbitrarily strengthened in implementations. The same applies to subtyping.

We propose the use of a predicate **enabled(m)** which asserts that **m** is available for execution, ie, that a request for **m** will be accepted (in terms of the RAL semantics of VDM^{++} this only means that **m** may be the next executed action of an object of the class, with no intermediate interruption by other

actions). Implementations of specifications containing such assertions therefore cannot strengthen the permission guards of **m** further than these asserted willingness conditions: the willingness conditions give an upper bound on how far the permissions can be strengthened.

In fact, we will interpret a permission guard

```
per m  ⇒  G
```

by the formula **enabled(m)** $\Rightarrow$ **G** in the semantics of VDM^{++} in the RAL formalism [10].

A willingness guard could be simply written as

```
enabled m  ⇐  W
```

in the **sync** clause of a class, and is interpreted by the formula **W** $\Rightarrow$ **enabled(m)**.

Interpreting refinement by theory extension, it is clear that willingness guards can be weakened (**m** can be asserted to be accepted under more conditions) and permission guards can be strengthened, provided we do not assert non-permission in cases where willingness has been asserted.

A natural example making use of these extra conditions would be a general buffer specification. This would express only a very loose form of synchronisation, whereby update methods such as **put** and **get** would be asserted to be self and mutually exclusive, and to exclude enquiry methods such as **full**, but the latter could co-execute. The permission guards would therefore have the form:

```
sync
  per get  ⇒
         #active(get) + #active(put) + #active(full) = 0;
  per put  ⇒
         #active(get) + #active(put) + #active(full) = 0;
  per full ⇒
         #active(get) + #active(put) = 0
```

where #**active**(**M**) counts the number of active instances of the method **M**, ie, the number of invocations of this method that have been initiated but not yet terminated. However, we do want to assert also that every subtype must guarantee acceptance of methods whenever possible. Since we may strengthen the synchronisation policy to one which is fully mutex, and in which **get** can only be executed if there are elements to get, ie, if **len**(**contents**) > 0, we can only make the following guarantees:

```
enabled put   ⇐  #active(get) + #active(put) + #active(full) = 0
enabled full  ⇐  #active(get) + #active(put) + #active(full) = 0
enabled get   ⇐  #active(get) + #active(put) + #active(full) = 0 ∧
                                                len(contents) > 0
```

Subtypes of this buffer can strengthen the permissions to define fully mutex behaviour and to prevent **get** executing on an empty buffer – in this case the permissions and willingness conditions would be equivalent and could not be further changed.

Alternatively, we could implement an internally concurrent buffer that obeyed the readers/writers protocol with **get** guarded by the empty buffer condition. In this case the willingness for **full** could be weakened to assert its availability if neither **put** or **get** are executing:

$$\mathsf{enabled}\ \mathbf{full} \ \Leftarrow \ \#\mathbf{active}(\mathbf{get}) + \#\mathbf{active}(\mathbf{put}) = 0$$

In this case again the permission and willingness guards would coincide.

6 Required Internal Non-Determinism

A similar treatment can be given to internal non-determinism. We could specify upper and lower bounds on the possible refinements of an operation effect by a pair POSS $\mathbf{S_l}$ REQD $\mathbf{S_u}$ END where the $\mathbf{S_i}$ are B-like substitutions with a wp semantics, such that $\mathbf{S_l} \sqsubseteq \mathbf{S_u}$. This pair is refined by POSS $\mathbf{H_l}$ REQD $\mathbf{H_u}$ END iff $\mathbf{S_l} \sqsubseteq \mathbf{H_l}$ and $\mathbf{H_u} \sqsubseteq \mathbf{S_u}$ in the usual sense.

The intuitive meaning of such a pair is that all its "executions" obey the specification $\mathbf{S_l}$, and that every execution that obeys $\mathbf{S_u}$ is an execution of the pair. There is underspecification if $\mathbf{S_u}$ is not equivalent to $\mathbf{S_l}$, in that it is not known which executions of specifications "between" $\mathbf{S_l}$ and $\mathbf{S_u}$ are included in its implementation. This underspecification can be resolved in different ways in distinct subtypes or alternative refinements.

Thus we could specify a random number generator which must at least provide a non-deterministic choice between the numbers $0, \ldots, 10$ by:

$$\text{POSS } \mathbf{x} :\in \mathbb{N} \text{ REQD } \mathbf{x} :\in 0\,..\,10 \text{ END}$$

This can be refined by POSS $\mathbf{x} :\in 0..20$ REQD $\mathbf{x} :\in 0..20$ END or by POSS $\mathbf{x} :\in 0\,..\,10$ REQD $\mathbf{x} :\in 0\,..\,10$ END for example, but not by POSS $\mathbf{x} := 0$ REQD $\mathbf{x} := 0$ END which is equivalent to the single statement $\mathbf{x} := 0$.

If $\mathbf{S}$ is non-deterministic[2] then POSS $\mathbf{S}$ REQD $\mathbf{S}$ END cannot be simplified to $\mathbf{S}$ – because then subsequent refinement could remove the non-determinism – but must be implemented by a program element which genuinely exhibits the required non-determinism.

In terms of semantics, we can represent the "executions" of a statement by actions in the logics of Section 3. Requiring a certain level of internal non-determinism can then be expressed as asserting that a certain collection of these actions are possible next steps, ie, $\mathbb{E}\gamma$ holds for each such execution action γ.

Specifically, let $\alpha_{\mathbf{x},\mathbf{n}}$ be the action with effect $\mathbf{x} := \mathbf{n}$ where $\mathbf{x}$ is a list of attributes, and $\mathbf{n}$ a list of corresponding values. Then to assert that the full non-determinism expected from a statement $\mathbf{S}$ such as POSS $\mathbf{x} :\in 0\,..\,10$ REQD $\mathbf{x} :\in 0\,..\,10$ END occurs, we write:

$$\mathbb{E}\alpha_{\mathbf{x},0} \wedge \mathbb{E}\alpha_{\mathbf{x},1} \wedge \ldots \wedge \mathbb{E}\alpha_{\mathbf{x},10}$$

[2] That is, $\exists\,\mathbf{x}' \cdot [\mathbf{S}](\mathbf{x} = \mathbf{x}')$ fails where $\mathbf{x}$ is the write frame of $\mathbf{S}$.

In other words, there *are* executions of the statement **S** for each of the specified behaviours. In terms of Kripke models, there are paths starting from the current state which begin with some $\alpha_{\mathbf{x},\mathbf{i}}$ for each choice of $\mathbf{i} \in 0\,..\,10$. Notice that the logic does not distinguish between choices that are made externally from those that are made internally – $\mathbb{E}\alpha$ simply asserts that *some* behaviour starting with an execution of α will occur.

For an action definition

α = POSS **L** REQD **U** END

we have the axioms

$$\alpha \Rightarrow [\![\, \mathbf{L} \,]\!]$$

where $[\![\, \mathbf{L} \,]\!]$ is the temporal logic semantics of **L** as given in Appendix B, and

$$\alpha \Rightarrow \bigwedge_{\alpha_{\mathbf{x},\mathbf{n}}} \mathbb{E}\alpha_{\mathbf{x},\mathbf{n}}$$

where the $\alpha_{\mathbf{x},\mathbf{n}}$ are all those assignment actions which have $\mathbf{U} \sqsubseteq \alpha_{\mathbf{x},\mathbf{n}}$.

7 Discussion

We have proposed adding one extra guard to a B, VDM^{++} or UNITY specification, and subsuming the usual precondition under a willingness guard. There may be justification in retaining three separate guards: a precondition **Pre**, a permission guard **G** and a willingness guard **W**. The meaning of these guards for an action α with postcondition **Post** is expressed by the axioms:

1. $\alpha \wedge \mathbf{Pre} \Rightarrow \bigcirc\mathbf{Post}$ "If α executes when **Pre** holds, then it terminates, and **Post** is achieved at termination".
2. $\alpha \Rightarrow \mathbf{G}$ "α can only execute if **G** holds".
3. $\mathbf{W} \Rightarrow \mathbb{E}\alpha$ "If **W** holds, α must be accepted if called".

If **G** contains only attributes, and no temporal operators or action symbols, then $\mathbb{E}\mathbf{G} \Rightarrow \mathbf{G}$ and hence $\mathbf{W} \Rightarrow \mathbf{G}$. There are thus 6 possible behaviours for various combinations of the guards being true or false (Table 1). The final case would be eliminated if we required that $\mathbf{W} \Rightarrow \mathbf{Pre}$, which would be the case if we adopted the suggestion of Section 4 that **W** ensures termination of the operation statement.

Internal actions are used in [3, 2] in order to decompose an operation into a sequence of subordinate operations. The same approach could be taken in our version of B/COMMUNITY if internal actions and action hiding were included in the language.

Required non-determinism can be expressed in the traces/failures semantics of CSP. If **s** is a trace and **a** an action, then

$$(\mathbf{s} \frown \langle \mathbf{a} \rangle, \mathbf{X}) \in \mathbf{F} \quad \wedge \quad (\mathbf{s}, \{\mathbf{a}\}) \in \mathbf{F}$$

Pre	*G*	*W*	
$\surd$	$\times$	$\times$	No execution
$\surd$	$\surd$	$\times$	Execution is permitted
			Executions that occur will be valid
$\surd$	$\surd$	$\surd$	Acceptance of execution guaranteed
			Executions will be valid
$\times$	$\times$	$\times$	No execution
$\times$	$\surd$	$\times$	Execution permitted, but non-termination
			or arbitrary behaviour may occur
$\times$	$\surd$	$\surd$	Execution guaranteed, but non-termination
			or arbitrary behaviour may occur

Table 1. Behaviour Depending on Guards

for a failures set $\mathbf{F}$ indicates that $\mathbf{a}$ is permitted but not required to be available in 'state' $\mathbf{s}$. This corresponds to $\mathbf{s}$ being in the 'grey area' in Figure 1.

$$(\mathbf{s} \frown \langle \mathbf{a} \rangle, \mathbf{X}) \in \mathbf{F} \quad \wedge \quad (\mathbf{s}, \{\mathbf{a}\}) \notin \mathbf{F}$$

indicates that $\mathbf{a}$ is permitted and required, whilst

$$(\mathbf{s}, \{\mathbf{a}\}) \in \mathbf{F} \;\wedge\; (\mathbf{s} \frown \langle \mathbf{a} \rangle, \mathbf{X}) \notin \mathbf{F}$$

indicates that $\mathbf{a}$ is not permitted.

Refinement in CSP terms then means the first kind of non-determinism may be eliminated.

8 Conclusion

We have introduced a means of specifying required external non-determinism in model-based languages: providing the environment with a guarantee of acceptance of one of a choice of possible actions, by means of *willingness guards*. We have defined languages for specifying and implementing systems with required external non-determinism, and shown how these languages can be related to specification languages such as B and VDM^{++}. Similar extensions could be made to the Syntropy [5] or $\pi \mathbf{o} \beta \lambda$ [8] languages. We have also described how required internal non-determinism can be treated in this context.

References

1. J-R Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J-R Abrial. *Extending B Without Changing it (for Developing Distributed Systems)*, B Conference, IRIN, Nantes, November 1996.
3. M Butler. *Stepwise Refinement of Communicating Systems*, Southampton University, 1997.

4. K M Chandy and J Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.
5. S Cook and J Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Sept 1994.
6. J Fiadeiro and T Maibaum. *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing 4(3), pp. 239–272, 1992.
7. J Fiadeiro and T Maibaum. *Categorical Semantics of Parallel Program Design*, Science of Computer Programming, in print, 1997.
8. C B Jones. *Accommodating Interference in the formal design of concurrent object-based programs.* Formal Methods in System Design, 8(2): 105–122, March 1996.
9. R Kuiper. *Enforcing Nondeterminism via Linear Time Temporal Logic Specification using Hiding*, in B Banieqbal, H Barringer and A Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 295–303.
10. K Lano, S Goldsack, J Bicarregui and S Kent. *Integrating VDM^{++} and Real-Time System Design*, Z User Meeting, 1997.
11. A Lopes. *COMMUNITY and Required Non-determinism*, Department of Informatics, University of Lisbon, 1996.
12. C McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD Thesis, University of Dublin, 1995.
13. C Stirling. Comparing linear and branching time temporal logics. In B Banieqbal, H Barringer and A Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989.

A Semantics of COMMUNITY

The semantics of a program is expressed by a form of transition system (we give the case for propositional logic, the case of first-order logic is similar [11]).

Definition A *Θ-interpretation structure* for a signature $\Theta = (\mathbf{V}, \Gamma)$ is a pair $(\mathcal{T}, \mathcal{A})$ where

- $\mathcal{T}$ is a transition system $(\mathcal{W}, \mathbf{w}_0, 2^{\Gamma}, \rightarrow)$
- $\mathcal{A}$ is a map $\mathcal{A} : \mathbf{V} \rightarrow (\mathcal{W} \rightarrow \{\mathbf{t}, \mathbf{f}\})$

Definition Given a signature $\Theta = (\mathbf{V}, \Gamma)$ and Θ-interpretation structure $\mathbf{S} = (\mathcal{T}, \mathcal{A})$, the truth $(\mathbf{S}, \mathbf{w}) \models \varphi$ of a formula φ at a world $\mathbf{w} \in \mathcal{W}$ of $\mathbf{S}$ is recursively defined by:

$$(\mathbf{S}, \mathbf{w}) \models \mathbf{a} \quad \text{iff} \quad \mathcal{A}(\mathbf{a})(\mathbf{w})$$
$$(\mathbf{S}, \mathbf{w}) \models (\mathbf{t}_1 = \mathbf{t}_2) \quad \text{iff} \quad [\![\, \mathbf{t}_1 \,]\!]^{\mathbf{S}} = [\![\, \mathbf{t}_2 \,]\!]^{\mathbf{S}}$$
$$(\mathbf{S}, \mathbf{w}) \models (\phi_1 \Rightarrow \phi_2) \quad \text{iff} \quad (\mathbf{S}, \mathbf{w}) \models \phi_1 \;\text{ implies }\; (\mathbf{S}, \mathbf{w}) \models \phi_2$$
$$(\mathbf{S}, \mathbf{w}) \models \phi_1 \vee \phi_2 \quad \text{iff} \quad (\mathbf{S}, \mathbf{w}) \models \phi_1 \;\text{ or }\; (\mathbf{S}, \mathbf{w}) \models \phi_2$$
$$(\mathbf{S}, \mathbf{w}) \models \neg\, \phi \quad \text{iff} \quad \neg\, ((\mathbf{S}, \mathbf{w}) \models \phi)$$

Definition A formula ϕ is *true* in a Θ interpretation structure $\mathbf{S}$, written $\mathbf{S} \models \phi$, iff $(\mathbf{S}, \mathbf{w}) \models \phi$ at every state $\mathbf{w}$ of $\mathbf{S}$.

Definition Given a program $\mathbf{P} = (\Theta, \Delta)$ where $\Theta = (\mathbf{V}, \Gamma)$ and $\Delta = (\mathbf{I}, \mathbf{F}, \mathbf{P}, \mathbf{W})$, a *model of* P is a Θ-interpretation structure $\mathbf{S} = (\mathcal{T}, \mathcal{A})$ such that:

1. $(\mathbf{S}, \mathbf{w}_0) \models \mathbf{I}$. The initialisation is true in the first state;
2. for every set $\mathbf{e} \subseteq \Gamma$ of actions, action $\mathbf{g} \in \mathbf{e}$ and attribute $\mathbf{a} \in \mathbf{D}(\mathbf{g})$, if $\mathbf{w}, \mathbf{w}' \in \mathcal{W}$ have $\mathbf{w} \rightarrow^{\mathbf{e}} \mathbf{w}'$ then $\mathcal{A}(\mathbf{a})(\mathbf{w}') = [\![\mathbf{F}(\mathbf{g}, \mathbf{a})]\!]^{\mathbf{S}}(\mathbf{w})$. The effect of $\mathbf{F}(\mathbf{g})$ is achieved on $\mathbf{a}$;
3. for every attribute $\mathbf{a}$, $\mathbf{w}, \mathbf{w}' \in \mathcal{W}$, and $\mathbf{e} \subseteq \Gamma$, if $\mathbf{w} \rightarrow^{\mathbf{e}} \mathbf{w}'$ and no $\mathbf{g} \in \mathbf{e}$ has $\mathbf{a} \in \mathbf{D}(\mathbf{g})$, then $\mathcal{A}(\mathbf{a})(\mathbf{w}) = \mathcal{A}(\mathbf{a})(\mathbf{w}')$. Attributes outside the frame of every action currently executing are unchanged;
4. for every action $\mathbf{g} \in \Gamma$, $\mathbf{w} \in \mathcal{W}$ and $\mathbf{e} \subseteq \Gamma$ with $\mathbf{g} \in \mathbf{e}$, if there exists some $\mathbf{w}' \in \mathcal{W}$ with $\mathbf{w} \rightarrow^{\mathbf{e}} \mathbf{w}'$, then $(\mathbf{S}, \mathbf{w}) \models \mathbf{P}(\mathbf{g})$. Any action that occurs from $\mathbf{w}$ must be permitted in this state;
5. for every $\mathbf{e} \subseteq \Gamma$ and $\mathbf{w} \in \mathcal{W}$, if $(\mathbf{S}, \mathbf{w}) \models \mathbf{W}(\mathbf{g})$ for every $\mathbf{g} \in \mathbf{e}$ and $[\![\mathbf{F}(\mathbf{g}, \mathbf{a})]\!]^{\mathbf{S}}(\mathbf{w}) = [\![\mathbf{F}(\mathbf{g}', \mathbf{a})]\!]^{\mathbf{S}}(\mathbf{w})$ for each $\mathbf{g}, \mathbf{g}' \in \mathbf{e}$ and $\mathbf{a} \in \mathbf{D}(\mathbf{g}) \cap \mathbf{D}(\mathbf{g}')$, then there exists some $\mathbf{w}'$ such that $\mathbf{w} \rightarrow^{\mathbf{e}} \mathbf{w}'$. A program must be capable of executing a set of actions all of whose willingness conditions are true, and whose effects are consistent.

Implicitly, only the actions of a program can possibly change the attributes of that program – all concurrency in COMMUNITY is achieved by the sharing of actions rather than sharing of data.

B Temporal Logic Specifications

A Θ model of a CTL* theory is a tuple $(\mathcal{W}, \mathcal{R}, \mathcal{V}, \mathcal{E}, \mathbf{w}_0)$ where $\mathcal{R} \subseteq \mathcal{W} \times \mathcal{W}$ is a transition relation over the set $\mathcal{W}$, $\mathcal{V} : \mathbf{V} \rightarrow \mathbb{P}(\mathcal{W})$ and $\mathcal{E} : \Gamma \rightarrow \mathbb{P}(\mathcal{W} \times \mathcal{W})$ give the interpretation of attributes and actions. $\mathbf{w}_0$ is the initial state.

Each Θ-model $\mathbf{M}$ generates a set of *paths* through $\mathcal{R}$, $\mathbf{path}(\mathbf{M})$, defined by

$$\{\pi \in \mathbb{N} \rightarrow \mathcal{W} : \forall \mathbf{i} : \mathbb{N} \cdot \mathcal{R}(\pi(\mathbf{i}), \pi(\mathbf{i} + 1))\}$$

The satisfaction relation $\models$ is defined by: $(\mathbf{M}, \mathbf{w}) \models \phi$ for state formulae ϕ iff

- for all $\mathbf{a} \in \mathbf{V}$, $(\mathbf{M}, \mathbf{w}) \models \mathbf{a}$ iff $\mathbf{w} \in \mathcal{V}(\mathbf{a})$
- $(\mathbf{M}, \mathbf{w}) \models \neg\, \phi$ iff $\neg\, ((\mathbf{M}, \mathbf{w}) \models \phi)$
- $(\mathbf{M}, \mathbf{w}) \models \phi \Rightarrow \psi$ iff $(\mathbf{M}, \mathbf{w}) \models \phi$ implies $(\mathbf{M}, \mathbf{w}) \models \psi$
- $(\mathbf{M}, \mathbf{w}) \models \mathbf{beg}$ iff $\mathbf{w} = \mathbf{w}_0$
- $(\mathbf{M}, \mathbf{w}) \models \mathbb{E}\phi$ iff there exists a path $\pi \in \mathbf{path}(\mathbf{M})$ such that $\pi(0) = \mathbf{w}$ and $(\mathbf{M}, \pi) \models \phi$

For path formulae $\models$ is defined by:

- for all $\mathbf{g} \in \Gamma$, $(\mathbf{M}, \pi) \models \gamma$ iff $(\pi(0), \pi(1)) \in \mathcal{E}(\gamma)$
- for all state formulae ϕ, $(\mathbf{M}, \pi) \models \phi$ iff $(\mathbf{M}, \pi(0)) \models \phi$
- $(\mathbf{M}, \pi) \models \neg\, \phi$ iff $\neg\, ((\mathbf{M}, \pi) \models \phi)$
- $(\mathbf{M}, \pi) \models \phi \Rightarrow \psi$ iff $(\mathbf{M}, \pi) \models \phi$ implies $(\mathbf{M}, \pi) \models \psi$
- $(\mathbf{M}, \pi) \models \bigcirc\phi$ iff $(\mathbf{M}, \pi^1) \models \phi$ where $\pi^{\mathbf{j}}(\mathbf{i})$ is $\pi(\mathbf{j} + \mathbf{i})$
- $(\mathbf{M}, \pi) \models \phi \mathcal{U} \psi$ iff there exists $\mathbf{j} > 0$ such that $(\mathbf{M}, \pi^{\mathbf{j}}) \models \psi$, and such that for all $\mathbf{k}$ with $0 < \mathbf{k} \leq \mathbf{j}$, $(\mathbf{M}, \pi^{\mathbf{k}}) \models \phi$.

We can relate programs in COMMUNITY to specifications in **CTL*** by defining a mapping from a program to a specification which gives an axiomatic expression of its semantics.

Definition Given a program $\mathbf{P} = (\Theta, \Delta)$ where $\Delta = (\mathbf{I}, \mathbf{F}, \mathbf{P}, \mathbf{W})$ we define:

1. $\mathbf{Safe}(\mathbf{P})$ is the following set of formulae in $\mathbf{LTL}(\Theta)$:
   - $\mathbf{beg} \Rightarrow \mathbf{I}$
   - For every action $\mathbf{g} \in \Gamma$, the proposition $\mathbf{g} \;\Rightarrow\; \mathbf{P}(\mathbf{g}) \wedge \bigwedge_{\mathbf{a} \in \mathbf{D}(\mathbf{g})} \bigcirc \mathbf{a} = \mathbf{F}(\mathbf{g}, \mathbf{a})$
   - For every $\mathbf{a} \in \mathbf{V}$, the proposition $(\bigvee_{\mathbf{a} \in \mathbf{D}(\mathbf{g})} \mathbf{g}) \vee \bigcirc \mathbf{a} = \mathbf{a}$.
2. $\mathbf{ND}(\mathbf{P})$ is the following set of formulae in $\mathbf{CTL}^*(\Theta)$:
   - for every set $\gamma \subseteq \Gamma$ of actions: $\mathbb{A}(\mathbf{W}(\gamma) \wedge \mathbf{sync}(\gamma) \Rightarrow \mathbb{E}\gamma)$ where

$$\begin{aligned}
&\mathbf{W}(\gamma) \;=\; \textstyle\bigwedge_{\mathbf{g} \in \gamma} \mathbf{W}(\mathbf{g}) \\
&\gamma \;=\; \textstyle\bigwedge_{\mathbf{g} \in \gamma} \mathbf{g} \;\wedge\; \bigwedge_{\mathbf{g} \notin \gamma} \neg\, \mathbf{g} \\
&\mathbf{sync}(\gamma) \;=\; \textstyle\bigwedge \{\mathbf{F}(\mathbf{g}, \mathbf{a}) = \mathbf{F}(\mathbf{g}', \mathbf{a}) : \mathbf{g}, \mathbf{g}' \in \gamma \wedge \mathbf{a} \in \mathbf{D}(\mathbf{g}) \cap \mathbf{D}(\mathbf{g}')\}
\end{aligned}$$

$\mathbf{Safe}(\mathbf{P})$ gives the axioms for the effects and permissions of the actions, together with typing axioms (for COMMUNITY with general variables). $\mathbf{ND}(\mathbf{P})$ gives the axioms for willingness.

We can generalise the semantics to deal with B substitutions other than $\mathbf{a} := \mathbf{F}(\mathbf{g}, \mathbf{a})$ by defining a temporal logic semantics $[\![\, \mathbf{S} \,]\!]$ for substitutions $\mathbf{S}$:

$$\begin{aligned}
&[\![\, \mathbf{x} \;:=\; \mathbf{e} \,]\!] &&= \bigcirc \mathbf{x} = \mathbf{e} \\
&[\![\, \mathbf{x} \;:\in\; \mathbf{s} \,]\!] &&= \bigcirc \mathbf{x} \in \mathbf{s} \\
&[\![\, \mathbf{S}_1 \,||\, \mathbf{S}_2 \,]\!] &&= [\![\, \mathbf{S}_1 \,]\!] \wedge [\![\, \mathbf{S}_2 \,]\!] \\
&[\![\, \textsc{if}\ \mathbf{E}\ \textsc{then}\ \mathbf{S}_1\ \textsc{else}\ \mathbf{S}_2\ \textsc{end} \,]\!] &&= (\mathbf{E} \Rightarrow [\![\, \mathbf{S}_1 \,]\!]) \wedge (\neg\, \mathbf{E} \Rightarrow [\![\, \mathbf{S}_2 \,]\!]) \\
&[\![\, \mathbf{op}(\mathbf{x}) \,]\!] &&= \mathbf{op}(\mathbf{x})
\end{aligned}$$

with similar clauses for PRE, ANY and SELECT.

This article was processed using the LaTeX macro package with LLNCS style