Applications of Summation and Product Operators in the Refinement Calculus

Ralph-Johan Back Michael Butler
Dept. of Computer Science, Åbo Akademi, Finland
{backrj,mbutler}@abo.fi

30 November 1994

Abstract

Product and summation operators for predicate transformers were introduced by Naumann [20] and by Martin [15] using category theoretic considerations. In this paper, we formalise these operators in the higher order logic approach to the refinement calculus of [6], look at various algebraic properties of these operators, and examine several of their applications. We look at how the product operator provides a model of simultaneous execution of statements, while the summation operator provides a simple model of late binding. We also generalise the product operator slightly to form an operator that corresponds to conjunction of specifications. We show how the product operator may be used to model extension and modification operators for programs, and how a combination of the product and summation operators may be used to model inheritance in an object-oriented programming language.

1 Introduction

Dijkstra introduced weakest-precondition predicate transformers as a means of verifying total correctness properties of sequential programs [9]. In the *refinement calculus* of Back, Morgan, and Morris [2, 16, 18], specifications and programs are regarded uniformly as predicate transformers, and refinement laws are derived from properties of predicate transformers.

The refinement calculus provides various choice and assignment operators that are generalisations of Dijkstra's operators, and the applications of these operators are well-known. However, the applications of an operator representing simultaneous execution of program statements are less well developed in the refinement calculus. Such an operator was introduced by Naumann [20] and by Martin [15] using category theoretic considerations. This *product* operator combines predicate transformers by forming the cartesian product of their state spaces. In this paper, we examine the product operator using the higher-order logic formalisation of the refinement calculus of Back & von Wright [6]. We examine various distributivity and refinement preserving properties of the operator and show that it can be used to model simultaneous execution and to extend the state spaces of statements so they can be more easily matched with other statements. We also generalise the definition of the product operator slightly to form what we call a *fusion* operator and show that the product operator is a special case of the fusion operator. The fusion operator can also be applied to conjoining or amalgamating specification statements.

The *summation* (or *co-product*[1]) operator, which is the categorical dual of the product operator and combines statements by forming the disjoint union of their state spaces, is also described in [20] and [15]. The summation operator is a form of choice operator and we show that it is a special case of the existing choice operators of the refinement calculus. We show that this operator provides a simple yet powerful model of late binding, and that when combined with the product operator, provides a model of inheritance in an object-oriented programming language.

The paper is organised as follows. First the refinement calculus basics are described. Then Section 3 examines properties of the summation operator, while Section 4 examines properties of the product and fusion operators. In Section 5 we look briefly at the categorical properties of the summation and product operators. Section 6 looks at various applications of the operators including simultaneous execution, extending state spaces, late binding, and inheritance. Note that proofs are omitted here, but may be found in [3].

2 Refinement Calculus Basics

A *predicate* over a set of states Σ is a boolean function $p : \Sigma \to \mathbf{Bool}$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\,\Sigma$:

$$\mathcal{P}\,\Sigma \;\;\widehat{=}\;\; \Sigma \to \mathbf{Bool}.$$

We define the *entailment ordering* on predicates by pointwise extension: for $p, q : \mathcal{P}\,\Sigma$,

$$p \le q \;\;\widehat{=}\;\; (\forall \sigma : \Sigma \cdot p\ \sigma \;\Rightarrow\; q\ \sigma).$$

The identically false predicate is denoted $\bot$, and the identically true predicate is denoted $\top$. Negation, conjunction, and disjunction of (similarly-typed) predicates are defined by pointwise extension, so that, e.g.,

$$(p \wedge q)\ \sigma \;\;\widehat{=}\;\; (p\ \sigma \;\wedge\; q\ \sigma).$$

Predicates form a complete lattice under the entailment ordering. Conjunction and disjunction represent meet and join respectively, while $\top$ and $\bot$ represent top and bottom respectively.

A *relation* from Σ to Γ is a function $P : \Sigma \to \mathcal{P}\,\Gamma$ that maps each state σ to a predicate on Γ. We write

$$\Sigma \leftrightarrow \Gamma \;\;\widehat{=}\;\; \Sigma \to \mathcal{P}\,\Gamma.$$

This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The domain and range of relation P, are denoted $dom\,P$ and $ran\,P$ respectively. Conjunction and disjunction of relations is defined pointwise, so that, e.g., $(P \wedge Q)\ \sigma\ \gamma \;=\; (P\ \sigma\ \gamma) \wedge (Q\ \sigma\ \gamma)$. We write $P;Q$ for the relational composition of $P : \Sigma \leftrightarrow \Gamma$ and $Q : \Gamma \leftrightarrow \Lambda$, and P^{-1} for

[1] The direction of arrows in [20] and [15] are the reverse of what we use, so our product operator corresponds to their co-product operator, and vice versa.

the inverse relation. For function $f : \Sigma \rightarrow \Gamma$, we write f^{-1} for the relation $(\lambda\gamma \cdot (\lambda\sigma \cdot \gamma = f\sigma)) : \Gamma \leftrightarrow \Sigma$. A partial function $g : \Sigma \rightarrow_p \Gamma$ is a relation from Σ to Γ such that $g\ \sigma\ \gamma\ \wedge\ g\ \sigma\ \gamma' \ \Rightarrow\ \gamma = \gamma'$.

A *predicate transformer* is a function $S : \mathcal{P}\,\Gamma \rightarrow \mathcal{P}\,\Sigma$ from predicates to predicates. We write,

$$\Sigma \longmapsto \Gamma \quad \widehat{=} \quad \mathcal{P}\,\Gamma \ \rightarrow\ \mathcal{P}\,\Sigma.$$

Note the reversal of Γ and Σ: program statements in the refinement calculus are identified with weakest-precondition predicate transformers that map a postcondition $q : \mathcal{P}\,\Gamma$ to the weakest precondition $p : \mathcal{P}\,\Sigma$ such that the program is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. For program statement $S : \Sigma \longmapsto \Gamma$, we say that S has *source* Σ and *target* Γ. Programs need not have identical initial and final state spaces, though if they do, we write $S : \Delta(\Sigma)$ instead of $S : \Sigma \longmapsto \Sigma$. Note that, in order to aid intuition, we sometimes discuss commands from an operational viewpoint, though our formal reasoning is always in terms of predicate transformers.

The *refinement ordering* on predicates is defined by pointwise extension: for $S, T : \Sigma \longmapsto \Gamma$,

$$S \leq T \quad \widehat{=} \quad (\forall q : \mathcal{P}\,\Gamma \cdot S\ q\ \leq\ T\ q).$$

The refinement ordering on predicates models the notion of total-correctness preserving program refinement. A total-correctness specification is typically given as a precondition-postcondition pair $(pre, post)$, and program (i.e., predicate transformer) S satisfies $(pre, post)$ if $pre \leq (S\ post)$. Now, for programs S and T, $S \leq T$ holds if and only if T satisfies any specification satisfied by S.

Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the predicate transformer *abort* that maps each postcondition to $\bot$, and the top element is the predicate transformer *magic* that maps each postcondition to $\top$. The *abort* statement is never guaranteed to terminate, while the *magic* statement is *miraculous* since it is always guaranteed to establish any postcondition. A miraculous statement cannot be implemented. For statement S, $halt\ S \ \widehat{=}\ S\ \top$ describes those initial states under which S is guaranteed to terminate, while $gd\ S \ \widehat{=}\ \neg\ (S\ \bot)$ (called *guard* of S) describes those initial states under which S behaves non-miraculously.

Conjunction and disjunction of (similarly-typed) predicate transformers is defined pointwise, so that, e.g., $(S \wedge T)\ q = (S\ q) \wedge (T\ q)$. Conjunction of statements models *demonic nondeterministic choice* between executing S and T (i.e., each alternative must establish the postcondition), whereas disjunction models *angelic nondeterministic choice* (i.e., some alternative must establish the postcondition). If $halt\ S = \neg\ halt\ T$, then $S \vee T$ becomes a deterministic choice, while if $gd\ S = \neg\ gd\ T$, then $S \wedge T$ becomes a deterministic choice.

Sequential composition of program statements is modelled by functional composition of predicate transformers, i.e., for $S : \Sigma \longmapsto \Gamma,\ \ T : \Gamma \longmapsto \Lambda,\ \ p : \mathcal{P}\,\Lambda, \qquad S;T\ p \ \widehat{=}\ S\ (T\ p)$. The program statement $skip_\Sigma$ is modelled by the identity predicate transformer on $\mathcal{P}\,\Sigma$.

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update statement* $\{P\} : \Sigma \longmapsto \Gamma$ and *demonic update statement* $[P] : \Sigma \longmapsto \Gamma$ are defined by

$$\{P\}\ q\ \sigma \quad \widehat{=} \quad (\exists \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \wedge (q\ \gamma))$$

$$[P]\ q\ \sigma \quad \widehat{=} \quad (\forall \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \Rightarrow (q\ \gamma)).$$

When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P\sigma\gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P\sigma\gamma$ holds. If no such new state exists then $\{P\}$ aborts, while $[P]$ behaves as *magic*.

For predicate $p : \mathcal{P}\,\Sigma$, let $\overline{p} : \Sigma \leftrightarrow \Sigma$ be the corresponding test relation for p, i.e., $(\lambda\sigma \cdot \lambda\sigma' \cdot p\ \sigma\ \wedge\ \sigma = \sigma')$. Then we write $\{p\}$ for $\{\overline{p}\}$ and $[p]$ for $[\overline{p}]$. $\{p\}$ models the *assert* statement that behaves as *skip* if p holds, otherwise it aborts. $[p]$ models the *guard* statement that behaves as *skip* if p holds, otherwise it behaves as *magic*.

Given a function $f : \Sigma \rightarrow \Gamma$, the *deterministic update statement* $\langle f \rangle : \Sigma \longmapsto \Gamma$ is defined by

$$\langle f \rangle\ q\ \sigma \quad \widehat{=} \quad q\ (f\ \sigma).$$

Let $\tilde{f}$ be the the deterministic relation corresponding to function f, i.e., $(\lambda\sigma \cdot \lambda\gamma \cdot f\ \sigma = \gamma)$. Then we have $\{\tilde{f}\} = \langle f \rangle = [\tilde{f}]$.

Ordinary program constructs such conditionals, recursions, and assignments may be modelled using the basic operators presented above. For example, **if** g **then** S **else** T **fi** may be modelled by $[g]; S\ \wedge\ [\neg\, g]; T$ or alternatively by $\{g\}; S \vee \{\neg\, g\}; T$. Program variables may be modelled as cartesian components of the state space, and a multiple assignment $x, y := e, f$ in a state space with three components, representing program variables (x, y, z), may be modelled by the deterministic update:

$$\langle \lambda x, y, z \cdot (e, f, z) \rangle.$$

It is easy to show, for predicates *pre* and *post*, that $\{pre\}; [post]$ is the least statement satisfying the specification pair $(pre, post)$. Thus, implementing the specification $(pre, post)$ involves constructing a statement S such that $\{pre\}; [post] \leq S$. If *post* is a relation rather than a predicate, we can model postconditions on the before and after states in the manner of VDM [14] and Z [22]. Rules for the stepwise refinement of specification statements of the form $\{pre\}; [post]$ into more familiar program constructs may be found in [2, 16, 18].

All predicate transformers S constructed using the operators described above will be *monotonic*, i.e., $p \leq q \ \Rightarrow\ S\ p \leq S\ q$. A predicate transformer S is *bottom homomorphic* if $S\ \bot = \bot$, and *top homomorphic* if $S\ \top = \top$. *Conjunctive* predicate transformers form a subset of the monotonic predicate transformers satisfying $S\ (\forall i \in I \cdot q_i) = (\forall i \in I \cdot S\ q_i)$, for non-empty I. S is *universally conjunctive* if it is conjunctive and top homomorphic. *Disjunctive* and *universally disjunctive* predicate transformers are defined dually. A predicate transformer is *continuous* if it is disjunctive over non-empty chains of predicates. Demonic update statements are universally conjunctive, angelic update statements are universally disjunctive, and deterministic update statements are both. The operators $\wedge$ and ; preserve the conjunctivity of their operands, while $\vee$ and ; preserve the disjunctivity of their operands. Also, each of $\wedge$, $\vee$, and ; preserve refinement of monotonic predicate transformers, e.g., $S \leq S' \ \Rightarrow\ S; T \leq S'; T$.

In Dijkstra's original calculus, all statements were conjunctive and non-miraculous. Also, for conjunctive statements, disjunctivity corresponds to determinism, while continuity corresponds to bounded nondeterminism [9].

3 Summation Operators

In this section, we describe two summation operators for predicate transformers: the first only sums the operand sources, while the second sums both the operand sources and the operand targets.

The sum (or disjoint union) of two state spaces Σ_1 and Σ_2 is written $\Sigma_1 + \Sigma_2$. Associated with the summation are two *injections* ϕ_1 and ϕ_2 which map elements of the base type to elements of the summation ($A \rightarrow_\iota B$ represents the set of injective functions from A to B):

$$\phi_1 : \Sigma_1 \rightarrow_\iota \Sigma_1 + \Sigma_2 \tag{1}$$

$$\phi_2 : \Sigma_2 \rightarrow_\iota \Sigma_1 + \Sigma_2. \tag{2}$$

Since any element of $\Sigma_1 + \Sigma_2$ must either come from Σ_1 or Σ_2, but not both, the injections also satisfy:

$$(ran\, \phi_1, ran\, \phi_2) \text{ partition } \Sigma_1 + \Sigma_2. \tag{3}$$

We shall assume that all injection pairs (ϕ_1, ϕ_2) associated with summations satisfy (1), (2), and (3).

Since ϕ_1 and ϕ_2 are injective, their inverses ϕ_1^{-1} and ϕ_2^{-1} are partial functions:

$$\phi_1^{-1} : \Sigma_1 + \Sigma_2 \rightarrow_p \Sigma_1 \qquad \phi_2^{-1} : \Sigma_1 + \Sigma_2 \rightarrow_p \Sigma_2.$$

For $\sigma \in \Sigma_1 + \Sigma_2$, $(\phi_1^{-1}\sigma)$ is only defined if $\sigma \in (dom\, \phi_1^{-1})$.

Summation of predicates is given by:

Definition 1 *For $p_1 : \mathcal{P}\,\Sigma_1$, $p_2 : \mathcal{P}\,\Sigma_2$, $\quad p_1 + p_2$ is of type $\mathcal{P}\,(\Sigma_1 + \Sigma_2)$, where for $\sigma : \Sigma_1 + \Sigma_2$,*

$$(p_1 + p_2)\ \sigma \ \ \widehat{=}\ \ \sigma \in (dom\, \phi_1^{-1}) \wedge p_1(\phi_1^{-1}\sigma) \ \ \vee \ \ \sigma \in (dom\, \phi_2^{-1}) \wedge p_2(\phi_2^{-1}\sigma).$$

Given an injection $\phi_1 : \Sigma_1 \rightarrow_\iota \Sigma_1 + \Sigma_2$, the deterministic update

$$\langle \phi_1 \rangle : \Sigma_1 \longmapsto \Sigma_1 + \Sigma_2 \quad (\ =\ \mathcal{P}\,(\Sigma_1 + \Sigma_2) \rightarrow \mathcal{P}\,\Sigma_1)$$

maps predicates in the summation type to predicates in the first base type. The updates can be used to select the components of a summation of predicates:

Theorem 1 *For $p_1 : \mathcal{P}\,\Sigma_1$, $p_2 : \mathcal{P}\,\Sigma_2$,*

$$\langle \phi_1 \rangle\ (p_1 + p_2) \ =\ p_1 \quad \textit{and} \quad \langle \phi_2 \rangle\ (p_1 + p_2) \ =\ p_2.$$

The following theorem shows that a predicate in the summation type can always be separated into a summation of predicates:

Theorem 2 *For $p : \mathcal{P}\,(\Sigma_1 + \Sigma_2)$, $\quad p \ =\ \langle \phi_1 \rangle\ p \ +\ \langle \phi_2 \rangle\ p.$*

3.1 Summation of Predicate Transformers

The summation of predicate transformers S_1 and S_2 is written $S_1 \oplus S_2$, where S_1 and S_2 have the same target, but possibly distinct sources:

Definition 2 *For $S_1 : \Sigma_1 \longmapsto \Gamma$, $\ S_2 : \Sigma_2 \longmapsto \Gamma$, $S_1 \oplus S_2$ is of type $\Sigma_1 + \Sigma_2 \longmapsto \Gamma$, where for $q : \mathcal{P}\,\Gamma$:*

$$(S_1 \oplus S_2)\ q \ \ \widehat{=}\ \ (S_1\ q)\ +\ (S_2\ q).$$

The selection and separation properties of the summation operator follow directly from Theorems 1 and 2:

Theorem 3 *For* $S_1 : \Sigma_1 \longmapsto \Gamma,\ S_2 : \Sigma_2 \longmapsto \Gamma,\ S : \Sigma_1 + \Sigma_2 \longmapsto \Gamma,$

$$\langle\phi_1\rangle; (S_1 \oplus S_2) \ = \ S_1 \quad \textit{and} \quad \langle\phi_2\rangle; (S_1 \oplus S_2) \ = \ S_2$$
$$S \ = \ \langle\phi_1\rangle; S \ \oplus \ \langle\phi_2\rangle; S.$$

Summation of predicate transformers is a form of choice operator since the effect of executing $S_1 \oplus S_2$ in some initial state σ depends on the base type of σ: if it is of type Σ_1, then S_1 is executed, while if it is of type Σ_2, then S_2 is executed. This may be seen more clearly in the following theorem where the summation operator is characterised purely in terms of the existing choice operators. The theorem uses the following angelic and demonic updates formed from the inverse injections:

$$\{\phi_1^{-1}\} : \Sigma_1 + \Sigma_2 \longmapsto \Sigma_1 \qquad\qquad [\phi_1^{-1}] : \Sigma_1 + \Sigma_2 \longmapsto \Sigma_1.$$

Theorem 4 *For* $S_1 : \Sigma_1 \longmapsto \Gamma,\ S_2 : \Sigma_2 \longmapsto \Gamma,$

$$S_1 \oplus S_2 \ = \ \{\phi_1^{-1}\}; S_1 \ \vee \ \{\phi_2^{-1}\}; S_2 \ = \ [\phi_1^{-1}]; S_1 \ \wedge \ [\phi_2^{-1}]; S_2.$$

It is also clear from Theorem 4 that the choice between S_1 and S_2 is deterministic since ϕ_1^{-1} and ϕ_2^{-1} have disjoint domains, so that

$$\textit{halt}\ \{\phi_1^{-1}\} = \neg\ \textit{halt}\ \{\phi_2^{-1}\} \quad \text{and} \quad \textit{gd}\ [\phi_1^{-1}] = \neg\ \textit{gd}\ [\phi_2^{-1}].$$

3.2 Derived Summation Operator

Next we define a summation operator on predicate transformers with distinct sources and distinct targets. Let ψ_1 and ψ_2 be the injections for the summation $\Gamma_1 + \Gamma_2$ (thus, e.g., $\langle\psi_1\rangle$ has type $\Gamma_1 \longmapsto \Gamma_1 + \Gamma_2$).

Definition 3 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1,\quad S_2 : \Sigma_2 \longmapsto \Gamma_2,\quad S_1 + S_2$ *is of type* $\Sigma_1 + \Sigma_2 \longmapsto \Gamma_1 + \Gamma_2$, *where*

$$S_1 + S_2 \ \widehat{=} \ S_1; \langle\psi_1\rangle \ \oplus \ S_2; \langle\psi_2\rangle.$$

The components of $S_1 + S_2$ may be selected as follows:

Theorem 5 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1,\ S_2 : \Sigma_2 \longmapsto \Gamma_2,$

$$\langle\phi_1\rangle; (S_1 + S_2); \{\psi_1^{-1}\} \ = \ \langle\phi_1\rangle; (S_1 + S_2); [\psi_1^{-1}] \ = \ S_1$$
$$\langle\phi_2\rangle; (S_1 + S_2); \{\psi_2^{-1}\} \ = \ \langle\phi_2\rangle; (S_1 + S_2); [\psi_2^{-1}] \ = \ S_2.$$

Summation distributes through sequential composition as follows:

Theorem 6

$$\textbf{(i)} \ \ (S_1 + S_2) \ ; \ (T_1 \oplus T_2) \ = \ (S_1; T_1) \oplus (S_2; T_2).$$
$$\textbf{(ii)} \ \ (S_1 + S_2) \ ; \ (T_1 + T_2) \ = \ (S_1; T_1) + (S_2; T_2).$$

In order to consider the summation of angelic and demonic updates, we require the following definition of summation of relations:

Definition 4 *For* $P_1 : \Sigma_1 \leftrightarrow \Gamma_1,\ P_2 : \Sigma_2 \leftrightarrow \Gamma_2,\ \sigma : \Sigma_1 + \Sigma_2,\ \gamma : \Gamma_1 + \Gamma_2,$

$$\begin{aligned}(P_1 + P_2)\ \sigma\ \gamma \ \ \widehat{=} \ \ & \sigma \in (\textit{dom}\,\phi_1^{-1}) \ \wedge \ \gamma \in (\textit{dom}\,\psi_1^{-1}) \ \wedge \ P_1\ (\phi_1^{-1}\sigma)\ (\psi_1^{-1}\gamma) \ \ \vee \\ & \sigma \in (\textit{dom}\,\phi_2^{-1}) \ \wedge \ \gamma \in (\textit{dom}\,\psi_2^{-1}) \ \wedge \ P_2\ (\phi_2^{-1}\sigma)\ (\psi_2^{-1}\gamma).\end{aligned}$$

Summation distributes through updates, assert and guard statements as follows:

Theorem 7 *For* $P_1 : \Sigma_1 \leftrightarrow \Gamma_1,\ P_2 : \Sigma_2 \leftrightarrow \Gamma_2,\ p_1 : \mathcal{P}\,\Sigma_1,\ p_2 : \mathcal{P}\,\Sigma_2,$

$$\{P_1\} + \{P_2\} = \{P_1 + P_2\} \ \ \textit{and} \ \ [P_1] + [P_2] = [P_1 + P_2] \tag{4}$$
$$\{p_1\} + \{p_2\} = \{p_1 + p_2\} \ \ \textit{and} \ \ [p_1] + [p_2] = [p_1 + p_2]. \tag{5}$$

Both the summation operators preserve refinement allowing us to refine summands separately:

Theorem 8

$$\begin{array}{lll} S_1 \leq S_1' \ \wedge \ S_2 \leq S_2' & \Rightarrow & (S_1 \oplus S_2) \leq (S_1' \oplus S_2') \\ T_1 \leq T_1' \ \wedge \ T_2 \leq T_2' & \Rightarrow & (T_1 + T_2) \leq (T_1' + T_2'). \end{array}$$

4 Product Operators

The cartesian product of state spaces Σ_1 and Σ_2 is denoted $\Sigma_1 \times \Sigma_2$. Given two predicates q_1 and q_2 their product is denoted $q_1 \times q_2$ and is defined as follows:

Definition 5 *For $q_1 : \mathcal{P}\,\Sigma_1$, $q_2 : \mathcal{P}\,\Sigma_2$, $\quad q_1 \times q_2$ is of type $\mathcal{P}\,(\Sigma_1 \times \Sigma_2)$ where for $\sigma_1 : \Sigma_1$, $\sigma_2 : \Sigma_2$,*

$$(q_1 \times q_2)\ (\sigma_1, \sigma_2) \quad = \quad (q_1\ \sigma_1) \wedge (q_2\ \sigma_2).$$

Note that, in contrast to summation of predicates where any predicate of type $\mathcal{P}\,(\Sigma_1 + \Sigma_2)$ may be represented by the summation of two predicates of types $\mathcal{P}\,\Sigma_1$ and $\mathcal{P}\,\Sigma_2$ respectively, predicates of the form $q_1 \times q_2$ only form rectangular subsets of $\mathcal{P}\,(\Sigma_1 \times \Sigma_2)$. Also, the components of a product of predicates cannot always be selected, e.g., p_1 cannot be retrieved from $p_1 \times \perp$. As we shall see, these facts mean that the product operators for predicate transformers do not satisfy as many properties as the summation operator.

Consider the following two predicate transformers which have a common source but distinct targets:

$$S_1 : \Sigma \longmapsto \Gamma_1, \qquad S_2 : \Sigma \longmapsto \Gamma_2.$$

The product operator combines these two commands to form a command with target $\Gamma_1 \times \Gamma_2$ as follows:

Definition 6 *For $S_1 : \Sigma \longmapsto \Gamma_1$, $S_2 : \Sigma \longmapsto \Gamma_2$, $S_1 \otimes S_2$ is of type $\Sigma \longmapsto \Gamma_1 \times \Gamma_2$, where for $q : \mathcal{P}\,(\Gamma_1 \times \Gamma_2)$:*

$$(S_1 \otimes S_2)\ q \quad \widehat{=} \quad (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \,|\, q_1 \times q_2 \leq q \cdot S_1 q_1 \ \wedge \ S_2 q_2).$$

We shall see later that execution of $S_1 \otimes S_2$ has the same effect as simultaneous execution of S_1 and S_2.

Before investigating properties of the product operator further, we first look at two related operators. The first is derived from the product operator. Let π_1 and π_2 be the projections from $\Sigma_1 \times \Sigma_2$ to Σ_1 and Σ_2 respectively:

$$\begin{array}{ll} \pi_1 : \Sigma_1 \times \Sigma_2 \rightarrow \Sigma_1, & \pi_2 : \Sigma_1 \times \Sigma_2 \rightarrow \Sigma_2, \\ \pi_1(\sigma_1, \sigma_2) = \sigma_1, & \pi_2(\sigma_1, \sigma_2) = \sigma_2. \end{array}$$

The projections give the following deterministic updates:

$$\langle \pi_1 \rangle : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_1 \quad \text{and} \quad \langle \pi_2 \rangle : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

The statement $\langle \pi_1 \rangle$ starts in a state (σ_1, σ_2) and ends in the state σ_1 simply discarding the second component. The derived product operator combines commands with distinct sources and distinct targets, and is defined as follows:

Definition 7 *For $S_1 : \Sigma_1 \longmapsto \Gamma_1$, $S_2 : \Sigma_2 \longmapsto \Gamma_2$, $S_1 \times S_2$ is of type $\Sigma_1 \times \Sigma_2 \longmapsto \Gamma_1 \times \Gamma_2$, where*

$$S_1 \times S_2 \quad \widehat{=} \quad \langle \pi_1 \rangle; S_1 \ \otimes \ \langle \pi_2 \rangle; S_2.$$

We shall see later that the derived product operator models simultaneous execution of two commands with distinct initial states and distinct final states.

Definition 6 may be generalised to define what we term a *fusion* operator that combines commands with common sources and common targets as follows:

Definition 8 *For* $S_1, S_2 : \Sigma \longmapsto \Gamma$, $S_1 \odot S_2$ *is of type* $\Sigma \longmapsto \Gamma$*, where for* $q : \mathcal{P}\,\Gamma$*:*

$$(S_1 \odot S_2)\ q \quad \widehat{=} \quad (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot S_1 q_1 \ \wedge \ S_2 q_2).$$

Neither product operator is commutative, since $\Sigma_1 \times \Sigma_2$ is different to $\Sigma_2 \times \Sigma_1$ when Σ_1 and Σ_2 are different. However, it is clear from Definition 8 that the fusion operator is commutative. We proceed by investigating further properties of the fusion operator and showing that the product operators are special cases of the fusion operator.

4.1 Properties of Fusion Operator

The fusion operator preserves refinement:

Theorem 9 $\qquad S_1 \leq S_1' \ \wedge \ S_2 \leq S_2' \quad \Rightarrow \quad (S_1 \odot S_2) \leq (S_1' \odot S_2')$.

A program specification takes the form $\{p\};[P]$, where p is a state predicate and P is a state relation. The following theorem provides a simple way of calculating the fusion of two such specifications:

Theorem 10 *For relations* $P, Q : \Sigma \leftrightarrow \Gamma$*, and predicates* $p, q : \mathcal{P}\,\Sigma$*,*

$$\{p\};[P] \ \odot \ \{q\};[Q] \quad = \quad \{p \wedge q\};[P \wedge Q].$$

The theorem illustrates that the effect of the fusion operator is to reduce the (demonic) nondeterminism of the terminating behaviour of both commands. The theorem also illustrates that the fusion operator can be used as a (refinement preserving) way of combining program specifications.

Given a predicate transformer $S : \Sigma \longmapsto \Gamma$, we derive a relation, called *rel* S, of type $\Sigma \leftrightarrow \Gamma$ as follows:

$$(rel\ S)\ \sigma\ \gamma \quad \widehat{=} \quad \neg\ S(\lambda\gamma' \cdot \gamma \neq \gamma')\ \sigma.$$

This definition is from [10]. It can be shown that any conjunctive predicate transformer may be characterised as a specification involving *halt* S and *rel* S:

Lemma 1 *For conjunctive predicate transformer* S, $\qquad S \ = \ \{halt\ S\};[rel\ S]$.

Using this lemma and Theorem 10, we arrive at the following theorem:

Theorem 11 *For conjunctive predicate transformers* S *and* T*,*

$$S \odot T \quad = \quad \{\ (halt\ S) \wedge (halt\ T)\ \}\ ;\ [\ (rel\ S) \wedge (rel\ T)\ ].$$

Abrial [1] describes a parallel operator for statements whose definition is similar to the right-hand side of Theorem 11. Thus our fusion of statements is similar to Abrial's parallel composition, for conjunctive statements.

We say that two specifications $\{p\};[P]$ and $\{q\};[Q]$ are *contradictory* in some initial state σ if σ has a successful outcome in each specification, but does not have a successful outcome in their fusion, i.e., σ holds in p, q, *dom* P, and *dom* Q, but not in *dom* $(P \wedge Q)$. For those contradictory initial states, the fusion of both specifications behaves miraculously. Ward [25] has defined a slight variant of Abrial's parallel combinator where the combination behaves as *abort* when both specifications are contradictory. Thus, in Ward's case, a contradictory combination may be refined by any statement whereas, in our case, it is unimplementable. Furthermore, Ward's combinator is not refinement preserving.

4.2 Properties of Product Operator

Now we show that the product operator is a special case of the fusion operator. Let ϖ_1 and ϖ_2 be the projections from $\Gamma_1 \times \Gamma_2$ to Γ_1 and Γ_2 respectively:

$$\varpi_1 : \Gamma_1 \times \Gamma_2 \to \Gamma_1, \qquad \varpi_2 : \Gamma_1 \times \Gamma_2 \to \Gamma_2.$$

The inverses of ϖ_1 and ϖ_2 are relations:

$$\varpi_1^{-1} : \Gamma_1 \leftrightarrow \Gamma_1 \times \Gamma_2, \qquad \varpi_2^{-1} : \Gamma_2 \leftrightarrow \Gamma_1 \times \Gamma_2,$$

so that the demonic update $[\varpi_1^{-1}]$ transforms predicates in $\Gamma_1 \times \Gamma_2$ to predicates in Γ_1:

$$[\varpi_1^{-1}] : \Gamma_1 \longmapsto \Gamma_1 \times \Gamma_2, \qquad (= \mathcal{P}(\Gamma_1 \times \Gamma_2) \to \mathcal{P}\,\Gamma_1),$$

and similarly for $[\varpi_2^{-1}]$. The command $[\varpi_1^{-1}]$ starts in a state $\gamma_1 : \Gamma_1$ and ends in a state $(\gamma_1', \gamma_2') : \Gamma_1 \times \Gamma_2$ such that $\gamma_1 = \gamma_1'$ and γ_2' is chosen nondeterministically. Given $S_1 : \Sigma \longmapsto \Gamma_1$, the predicate transformer $S_1; [\varpi_1^{-1}] : \Sigma \longmapsto \Gamma_1 \times \Gamma_2$ is a 'lifted' version of S_1 that behaves as S_1 on the first component of the final state, and nondeterministically writes any value to the second component. Similarly for $S_2; [\varpi_2^{-1}]$. Now we have:

Theorem 12 *For $S_1 : \Sigma \longmapsto \Gamma_1,\ S_2 : \Sigma \longmapsto \Gamma_2$,*

$$S_1 \otimes S_2 \quad = \quad S_1; [\varpi_1^{-1}] \ \odot \ S_2; [\varpi_2^{-1}].$$

Preservation of refinement follows from Theorems 9 and 12. The product operator does not, in general, satisfy selection and separation properties, though it does satisfying the following:

Theorem 13 *For monotonic $S_1 : \Sigma \longmapsto \Gamma_1,\ S_2 : \Sigma \longmapsto \Gamma_2,\ q : \mathcal{P}\,\Gamma_1$,*

$$(S_1 \otimes S_2); \langle \varpi_1 \rangle\ q \quad = \quad (S_1 q\ \wedge\ S_2 \top)\ \vee\ (S_1 \top\ \wedge\ S_2 \perp).$$

The separation property does not hold for the product operator, though we do have the following inequality for conjunctive S:

Theorem 14 *For conjunctive $S : \Sigma \longmapsto \Gamma_1 \times \Gamma_2, \quad S; \langle \varpi_1 \rangle \otimes S; \langle \varpi_2 \rangle \ \leq\ S.$*

Assume that for functions $f_1 : \Sigma \to \Gamma_1$, $f_2 : \Sigma \to \Gamma_2$ and relations $P_1 : \Sigma \leftrightarrow \Gamma_1$, $P_2 : \Sigma \leftrightarrow \Gamma_2$ we have:

$$f_1 \otimes f_2\ \ \sigma \quad \widehat{=} \quad (f_1\ \sigma, f_2\ \sigma)$$

$$P_1 \otimes P_2\ \sigma\ (\gamma_1, \gamma_2) \quad \widehat{=} \quad P_1\ \sigma\ \gamma_1 \quad \wedge \quad P_2\ \sigma\ \gamma_2.$$

The product operator then combines with specification statements and deterministic updates in the following manner:

Theorem 15 *For $p_1, p_2 : \mathcal{P}\,\Sigma$, and $P_1 : \Sigma \leftrightarrow \Gamma_1,\ P_2 : \Sigma \leftrightarrow \Gamma_2$,*

$$\{p_1\}; [P_1]\ \otimes\ \{p_2\}; [P_2] \quad = \quad \{p_1 \wedge p_2\}; [P_1 \otimes P_2].$$

Theorem 16 *For $f_1 : \Sigma \to \Gamma_1,\ f_2 : \Sigma \to \Gamma_2, \quad \langle f_1 \rangle \otimes \langle f_2 \rangle \quad = \quad \langle f_1 \otimes f_2 \rangle.$*

4.3 Properties of the Derived Product Operator

Theorem 12 shows that the derived product operator is also a special case of the fusion operator:

Theorem 17 *For* $S_1 : \Sigma_1 \longmapsto \Gamma_1,\ \ S_2 : \Sigma_2 \longmapsto \Gamma_2$,

$$S_1 \times S_2 \;=\; \langle \pi_1 \rangle; S_1; [\varpi_1^{-1}] \;\odot\; \langle \pi_2 \rangle; S_2; [\varpi_2^{-1}].$$

Here, the command $\langle \pi_1 \rangle$ starts in a state $(\sigma_1, \sigma_2) : \Sigma_1 \times \Sigma_2$ and ends simply in the state σ_1. Given $S_1 : \Sigma_1 \longmapsto \Gamma_1$, the predicate transformer $\langle \pi_1 \rangle; S_1; [\varpi_1^{-1}] : \Sigma_1 \times \Sigma_2 \longmapsto \Gamma_1 \times \Gamma_2$ is a 'lifted' version of S_1 that behaves as S_1 on the first components of the before and after state, and behaves nondeterministically on the second components. Similarly for $\langle \pi_2 \rangle; S_1; [\varpi_1^{-1}]$.

The derived product operator preserves refinement and it is easy to show that it combines with specification statements and deterministic updates in a similar manner to the product operator.

5 Applications

In Section 4.1 (c.f. Theorem 10), we saw that the fusion operator corresponds to a form of conjunction operator for specifications. Here we look more closely at applications of the other operators.

5.1 Simultaneous Assignment

As mentioned in Section 2, variables may be modelled as cartesian components of the state space. We take the view that the predicate transformer $S_1 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_1$ models a statement that reads from program variables x and y, and assigns a value to program variable x, while $S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2$ reads from the same variables and assigns a value to y. The product $S_1 \otimes S_2$ has type $\Delta(\Sigma_1 \times \Sigma_2)$ and models simultaneous assignment to x and y. For example, we have

$$\begin{array}{rcl} x := f(x,y) \;\otimes\; y := g(x,y) & = & \langle \lambda x, y \cdot f(x,y) \rangle \;\otimes\; \langle \lambda x, y \cdot g(x,y) \rangle \\ & = & \qquad \{\text{Theorem 16}\} \\ & & \langle \lambda x, y \cdot f(x,y), g(x,y) \rangle \\ & = & x, y := f(x,y), g(x,y). \end{array}$$

If S_1 and S_2 read from distinct state components, i.e., S_1 has type $\Delta(\Sigma_1)$ and S_2 has type $\Delta(\Sigma_2)$, then their simultaneous execution is modelled by the derived product $S_1 \times S_2$. With both product operators we know that if either assignment is nonterminating, then the combined assignment is nonterminating, e.g., $S_1 \otimes \mathit{abort} \;=\; \mathit{abort}$.

The *action system* formalism of Back & Kurki-Suonio [4] uses predicate transformers to model parallel programs. An action system consists of an initialisation predicate transformer and a set of action predicate transformers. Execution of an action system proceeds by firstly executing the initialisation, then, repeatedly, executing an enabled action (an action A is enabled when $gd\ A$ holds). Two action systems are composed in parallel by composing their initialisations such that they are executed simultaneously and forming the union of their actions. Conventionally, the initialisations are demonic updates $[I_1]$, $[I_2]$, and their composition is simply $[I_1 \wedge I_2]$. The product operator provides a way of composing more general initialisations achieving the same effect.

In [8], the actions of an action system are given labels and a correspondence with Hoare's CSP is established. A version of parallel composition of action systems is described in which commonly labelled actions from the respective action systems are composed such that they are executed simultaneously. This composition is defined by the properties that it should satisfy. One such property is that the composition should only be enabled when both actions are enabled. The product operator almost satisfies all the required properties: it fails if one of the actions can abort. However, by guarding the product of the actions as follows, we get an operator that satisfies the required properties:

$$A_1 \| A_2 \;\;\widehat{=}\;\; [(gd\ A_1) \times (gd\ A_2)]\ ;\ (A_1 \times A_2).$$

5.2 Superposition

Consider the following two predicate transformers:

$$S_1 : \Delta(\Sigma_1), \qquad\qquad S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

S_1 only operates on Σ_1, while S_2 reads from $\Sigma_1 \times \Sigma_2$ and writes to Σ_2. Superposition of S_2 on to S_1 allows S_2 to add extra program variables which it writes to, as well as allowing S_2 to read the program variables of S_1. Such an operation may be defined as a special case of the product operator:

Definition 9 $\qquad S_1\,\mathbf{sup}\,S_2 \;\;\widehat{=}\;\; \langle\pi_1\rangle; S_1 \;\otimes\; S_2.$

Here, $\langle\pi_1\rangle$ reads from $\Sigma_1 \times \Sigma_2$ though it discards the Σ_2 component. For example, we have:

$$x := f(x) \;\;\mathbf{sup}\;\; y := g(x, y) \;\;=\;\; x, y := f(x), g(x, y).$$

Superposition refinement of action systems is described in [5], where superposition on individual actions is described in terms of sequential composition. Our superposition operator could be used instead.

5.3 Modification

Consider the following two predicate transformers:

$$S_1 : \Delta(\Sigma_1 \times \Sigma_2), \qquad\qquad S_2 : \Sigma_1 \times \Sigma_2 \longmapsto \Sigma_2.$$

S_1 operates on $\Sigma_1 \times \Sigma_2$, while S_2 reads from $\Sigma_1 \times \Sigma_2$ and writes to Σ_2. For example, we could have

$$S_1 \;=\; x, y := f(x, y), g(x, y), \qquad\qquad S_2 \;=\; y := h(x, y).$$

We wish to modify S_1 with S_2 so that S_1 writes to Σ_1 as before, but the value written to Σ_2 is determined instead by S_2. Again, such an operation may be defined as a special case of the product operator:

Definition 10 $\qquad S_1\,\mathbf{mod}\,S_2 \;\;\widehat{=}\;\; S_1; \langle\pi_1\rangle \;\otimes\; S_2.$

Here, $\langle\pi_1\rangle$ discards the Σ_2 component written to by S_1 preserving only the Σ_1 component. For example,

$$x, y := f(x, y), g(x, y) \;\;\mathbf{mod}\;\; y := h(x, y) \;\;=\;\; x, y := f(x, y), h(x, y).$$

5.4 Rearranging and Extending State Components

Sometimes when combining statements, their state components may not match in the required way. For example, the statements $S_1 : \Delta(\Sigma_1 \times \Sigma_2 \times \Sigma_3)$ and

$S_2 : \Delta(\Sigma_2 \times \Sigma_3 \times \Sigma_1)$ may not be combined directly as $S_1; S_2$, (even if Σ_1, Σ_2, and Σ_3 are the same sets, constructing $S_1; S_2$ directly will result in the state components being mismatched). Instead, the state components may be rearranged using an update statement before combining the statements:

$$S_1; \langle \delta \rangle; S_2,$$

where $\delta(\sigma_1, \sigma_2, \sigma_3) = (\sigma_2, \sigma_3, \sigma_1)$.

The same technique can be used to match statements modulo associativity. For example, although

$$(S_1 \times S_2) \times S_3 \quad \neq \quad S_1 \times (S_2 \times S_3),$$

we do have (let $\kappa((\sigma_1, \sigma_2), \sigma_2) \quad = \quad (\sigma_1, (\sigma_2, \sigma_2))$):

$$(S_1 \times S_2) \times S_3 \quad = \quad \langle \kappa \rangle; (S_1 \times (S_2 \times S_3)); \langle \kappa^{-1} \rangle.$$

Sometimes the state components of a statement need to be extended in order to combine it with another statement. For example, suppose we wish to combine $S_1 : \Delta(\Sigma_1 \times \Sigma_2)$ and $S_2 : \Delta(\Sigma_2)$. Here, S_2 is independent of Σ_1, so when combining it with S_1 we would most likely expect it to leave the Σ_1 component unchanged. To achieve this, we first combine S_2 with *skip*:

$$S_1; (skip_{\Sigma_1} \times S_2).$$

Extension of the state space is required, for example, when composing action systems: the state space of all actions must be extended to the state space of the combined action system.

5.5 Shared Variable Assignment

The fusion operator may be viewed as a way of combining statements such as that they both assign a value that they agree on to the same state component. In the case that two statements have only partially overlapping state components, then they need to be extended firstly with the *havoc* statement, before being combined using the fusion operator. For example, $S_1 : \Delta(\Sigma_1 \times \Sigma_2)$ and $S_2 : \Delta(\Sigma_2 \times \Sigma_3)$ may be combined as follows:

$$(S_1 \times havoc_{\Sigma_3}) \odot (havoc_{\Sigma_1} \times S_2).$$

Since *havoc* acts as a unit for the fusion operator, the effect of this statement on the Σ_1 component will be determined purely by S_1, and the effect on the Σ_3 component will be determined by Σ_3. The effect on the Σ_2 component will be the intersection of the nondeterministic possibilities allowed by S_1 and S_2. If that intersection is empty, then, provided S_1 and S_2 terminate, the combination behaves as magic.

Note that the above operands of the fusion operator, do not match exactly: the first is of type $\Delta((\Sigma_1 \times \Sigma_2) \times \Sigma_3)$, while the second is of type $\Delta(\Sigma_1 \times (\Sigma_2 \times \Sigma_3))$. However, these can easily be made to match using the function $(\lambda((\sigma_1, \sigma_2), \sigma_3) \cdot (\sigma_1, (\sigma_2, \sigma_3)))$ as described in Subsection 5.4.

5.6 Late Binding of Procedures

We model a procedure by simply associating a statement with a procedure name. Calling a procedure then involves executing the statement associated with the procedure name. The following example shows the concrete syntax used to describe procedures:

$$\begin{array}{l} \textbf{proc}\ Increment \\ \quad \textbf{var}\ x : \textbf{Int} \\ \quad x := x + 1 \end{array}$$

The concept of *late binding* of procedures is important in object-oriented programming. Late binding means that the effect of executing a procedure depends on the value of the state on which it is to be executed. For example, Utting & Robinson [23] model late binding in the refinement calculus by regarding a procedure as being a function from values to statements:

$$proc\ :\ \Sigma \rightarrow \Delta(\Sigma).$$

The effect of executing procedure *proc* in state σ is then determined by the statement $(proc\ \sigma)$. This is sometimes referred to as *instance-centered* late binding [12].[2]

A simpler notion of late binding uses types rather than values to determine which statement is selected when a procedure is called (so-called *class-centered* late binding [12]). Here we show how the summation operator may be used to model this form of late binding.

Firstly, the following operator lifts predicate transformers in either base type of a summation to the summation type:

Definition 11 *Let $S_1 : \Sigma_1 \longmapsto \Gamma_1$ and $S_2 : \Sigma_2 \longmapsto \Gamma_2$ be predicate transformers. Then,*

$$S_1^+ \mathrel{\widehat{=}} S_1 + abort \qquad\qquad S_2^+ \mathrel{\widehat{=}} abort + S_2.$$

Now, given predicate transformers

$$\begin{array}{ll} S_1 : \Sigma_1 \longmapsto \Gamma_1 & \qquad S_2 : \Sigma_2 \longmapsto \Gamma_2 \\ T_1 : \Gamma_1 \longmapsto \Lambda_1 & \qquad T_2 : \Gamma_2 \longmapsto \Lambda_2, \end{array}$$

we have the following result:

Theorem 18

$$S_1^+;(T_1+T_2) \quad = \quad (S_1;T_1)^+ \qquad\qquad S_2^+;(T_1+T_2) \quad = \quad (S_2;T_2)^+.$$

That is, the effect of (T_1+T_2) depends on whether S_1 or S_2 is executed beforehand.

Consider the following two procedures:

$$\begin{array}{ll} \textbf{proc}\ Invert_1 & \qquad \textbf{proc}\ Invert_2 \\ \quad \textbf{var}\ x : \textbf{Int} & \qquad \quad \textbf{var}\ x : \textbf{Bool} \\ \quad x := -x & \qquad \quad x := \neg\ x \end{array}$$

Let $Invert \mathrel{\widehat{=}} Invert_1 + Invert_2$. Then

$$\begin{array}{rcl} (x := 10)^+; Invert & = & (x := 10)^+; (Invert_1 + Invert_2) \\ & = & (x := 10; Invert_1)^+ \end{array}$$

[2]Sometimes the terms "late binding" and "dynamic binding" are used interchangeably in the literature. We prefer to reserve the term "dynamic binding" for the case where the procedure associated with an instance/class may change during execution. Such an effect may be achieved, for example, by using stored procedures as provided in Oberon [19] where procedures are themselves values. A predicate transformer model of stored procedures may be found in [21].

$$= \quad (x := -10)^{+},$$

while

$$\begin{aligned} (x := true)^{+}; Invert &= (x := true)^{+}; (Invert_1 + Invert_2) \\ &= (x := true; Invert_2)^{+} \\ &= (x := false)^{+}. \end{aligned}$$

Thus the effect of $Invert$ depends on whether an integer or boolean value has been assigned to x beforehand.

5.7 Inheritance

Consider the following object-oriented definition of a procedure:

proc R
inherits S
extended by E
modified by M

Here, R inherits the behaviour of S, extends the behaviour of S by E and modifies the behaviour of S by M.

Suppose S, E and M have the following types:

$$\begin{aligned} &S : \Delta(\Sigma_1 \times \Sigma_2) \\ &E : \Sigma_1 \times \Sigma_2 \times \Sigma_3 \longmapsto \Sigma_3 \\ &M : \Sigma_1 \times \Sigma_2 \times \Sigma_3 \longmapsto \Sigma_2. \end{aligned}$$

That is, E introduces a new component of type Σ_3, while M modifies how S behaves on the Σ_2 component. One way to define R would be as follows:

$$R \quad \widehat{=} \quad (S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M.$$

However, in object-oriented programming, as well as performing the extended and modified behaviour on the extended type of R, we would expect R to behave as S on values of the type of S. But some of the behaviour of S has been discarded in $(S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M$. So instead we define R as follows:

$$R \quad \widehat{=} \quad S + ((S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M).$$

Now, R uses late binding when choosing whether to perform the inherited behaviour or the extended/modified behaviour.

This form of inheritance preserves refinement:

Theorem 19

$$\begin{aligned} &S \le S' \;\wedge\; M \le M' \;\wedge\; E \le E' \;\Rightarrow \\ &S + ((S \,\mathbf{sup}\, E) \,\mathbf{mod}\, M) \;\le\; S' + ((S' \,\mathbf{sup}\, E') \,\mathbf{mod}\, M'). \end{aligned}$$

It should also be possible to define multiple inheritance in a similar way, with the fusion operator being used in the case where two inherited procedures act on the same state component.

6 Conclusions

We have investigated summation and product operators for predicate transformers and shown that they satisfy a rich set of algebraic laws. These are mostly distributivity laws and refinement preservation laws that experience with other operators in the refinement calculus has shown to be useful.

We examined possible applications of these operators such as simultaneous execution, conjunction of specifications, late binding and inheritance. Simultaneous execution is important, for example, for various forms of action system parallel composition. The fusion operator provides a refinement-preserving way of composing requirements (specifications) which could also be viewed as a form of multiple inheritance. Late binding and inheritance are important features of object-oriented programming languages, and although our approach is limited in that we only model class-centered late binding, we believe it to be sufficiently powerful and find the rich set of associated laws encouraging.

Acknowledgements

The work reported here was carried out within the IRENE-project supported by the Academy of Finland. We are grateful to David Naumann and Xu Qiwen for useful comments.

References

[1] J.R. Abrial. *B-Technology Technical Overview.* B-Core(UK) Ltd., 1993.

[2] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[3] R.J.R. Back and M.J. Butler. *Exploring Summation and Product Operators in the Refinement Calculus.* Åbo Akademi reports on Computer Science and Mathematics, Series A, No.152, 1994.

[4] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In K.A. Parker and G.A. Rose, editors, *FORTE'91.* North–Holland, 1992.

[6] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 5:247–272, 1990.

[7] R.J.R. Back and J. von Wright. *Refinement Calculus.* Book in preparation, Åbo Akademi, 1994.

[8] M.J. Butler. *Refinement and Decomposition of Value-Passing Action Systems.* In E. Best, editor, *CONCUR'93*, volume LNCS 715. Springer–Verlag, 1993.

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

[10] E.W. Dijkstra. *From Predicate Transformers to Predicates.* Manuscript EWD821, April 1982.

[11] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[12] J. Gutknecht. *Object-Oriented Programming with Oberon.* Lecture Notes from Eastern Finland Universities International Summer School on Novel Computing, Lapeenranta University of Technology, Finland, 1994.

[13] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *European Symposium on Programming*, volume LNCS 213. Springer–Verlag, 1986.

[14] C.B. Jones. *Systematic Software Development using VDM – Second Edition*. Prentice–Hall, 1990.

[15] C.E. Martin. *Preordered Categories and Predicate Transformers*. D.Phil. Thesis, Programming Research Group, Oxford University, 1991.

[16] C.C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[17] C.C. Morgan. The cuppest capjunctive capping, and Galois. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice–Hall, 1994.

[18] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[19] H.P. Mossenboeck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1994.

[20] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers*. Ph.D. Thesis, University of Texas at Austin, 1992.

[21] D.A. Naumann. *On the Essence of Oberon*. Southwestern University, Georgetown, Texas, 1993.

[22] J.M. Spivey. *The Z Notation - A Reference Manual*. Prentice–Hall, 1989.

[23] B.M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In *Mathematics of Program Construction, 1992*, volume LNCS 669. Springer–Verlag, 1993.

[24] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31(2):105–135, 1994.

[25] N. Ward. Adding specification constructs to the refinement calculus. In *FME'93*, volume LNCS 670. Springer–Verlag, 1993.