



ELSEVIER

Science of Computer Programming 26 (1996) 237–254

Science of
Computer
Programming

A relational calculus for program construction by parts¹

M. Frappier^{a,*}, A. Mili^b, J. Desharnais^c

^a*Département de Mathématiques et Informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada, J1K 2R1*

^b*Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada, K1N 6N5*

^c*Département d'Informatique, Université Laval, Ste-Foy, Québec, Canada, G1K 7P4*

Abstract

Given a specification that includes a number of user requirements, we wish to focus on the requirements in turn, and derive a partly defined program for each; then combine all the partly defined programs into a single program that satisfies all the requirements simultaneously. In this paper we introduce a mathematical basis for solving this problem, and we illustrate it by means of a simple example.

1. Introduction and motivation

We propose a program construction method whereby, given a specification that includes a number of user requirements, we focus on the requirements in turn, and derive a partly defined program for each; then combine all the partly defined programs into a single program that satisfies all the requirements simultaneously. In this paper we discuss this programming paradigm, which we call *program construction by parts*, and introduce a mathematical foundation for it.

Our paradigm is based on the premises that programs and specifications are represented by homogeneous binary relations, and that program construction proceeds by correctness-preserving stepwise transformations of relations. In Section 2, we briefly introduce the mathematical background that is needed for our discussions, in terms of relation algebras. In Section 3, we introduce a number of relational operators that are useful for structuring specifications; some of these operators are original, others have been introduced by other researchers. In Section 4, we introduce our programming

* Corresponding author. E-mail: frappier@dm1.usherb.ca.

¹ This research is supported by NSERC (Natural Sciences and Engineering Research Council) of Canada and by FCAR (Fonds pour la Formation de Chercheurs et l'Aide à la Recherche) of Québec.

paradigm by discussing in turn how to solve subspecifications and then how to combine the partially defined programs that are so obtained. Section 5 illustrates our discussions by means of a simple example, and Section 6 summarizes our findings and sketches our prospects for future work.

2. Mathematical background

2.1. Relation algebra

Relations have been used for specification and design in various domains (e.g., [1, 3, 13, 14]). We adopt the definitions provided in [17] for most relational notions used in this paper.

A *concrete* (homogeneous) relation R on a set S (state space) is a subset of the Cartesian product $S \times S$. We denote the *universal relation* ($S \times S$) by L , the *empty relation* by $\emptyset$ and the *identity relation* ($\{(s, s') | s' = s\}$) by I . Also, the complement $\bar{R}$ is the set difference $L - R$; the relational product $R \circ R'$ is defined by $\{(s, s') | \exists t : (s, t) \in R \wedge (t, s') \in R'\}$; finally, the converse $\hat{R}$ is $\{(s, s') | (s', s) \in R\}$.

Our spaces are typically structured as Cartesian products of elementary sets, and are represented in a Pascal-like fashion. Hence, the declaration

space $a : \mathbf{Nat}, b : \mathbf{Int}$

defines a space S as: $S = \mathbf{Nat} \times \mathbf{Int}$. For the sake of brevity, instead of writing our relations in the form

$$\{(\langle a, b \rangle, \langle a', b' \rangle) \in (\mathbf{Nat} \times \mathbf{Int}) \times (\mathbf{Nat} \times \mathbf{Int}) | p(a, b, a', b')\} ,$$

we simply write $\{p(a, b, a', b')\}$.

We say that a relation t is a *left vector* if and only if there exists a subset T of the space S such that $t = T \times S$; the converse $\hat{t}$ of a left vector t is called a *right vector*. To *prerestrict* a relation R to a subset T of the space, we use $t \cap R$, where t is the left vector corresponding to T . Similarly, to *postrestrict* relation R to T , we use $R \cap \hat{t}$. We also use left vectors to represent conditions in programming constructs like **if** and **while**. Moreover, vectors arise naturally in specifications and designs of programs, as we shall see in Section 5.

As a convention, we use symbols P, Q, R (possibly subscripted) for relations, we use t, u, v for left vectors, and we use $\mathcal{A}$ for an arbitrary non-empty set of relations.

The precedence of the relational operators, from highest to lowest, is the following: $\bar{}$ and $\hat{}$ bind equally, followed by $\circ$, followed by $\cap$ and finally by $\cup$. We usually omit the composition operator symbol $\circ$ in expressions (that is, we write QR for $Q \circ R$). The precedence of logical operators, from highest to lowest, is: $\neg, \wedge, \vee, \{\Leftarrow, \Rightarrow\}, \Leftrightarrow, \{\forall, \exists\}$.

2.2. Refinement ordering

We let a *specification* be defined by a set S , called the *space*, and a binary homogeneous relation R on S , called the *relation* of the specification. When the space is implicit from the context, we equate the specification with the relation R . A specification R is interpreted as follows: if a computation starts in a state s in the domain of relation R , then it must terminate in a state s' such that $(s, s') \in R$. If a computation starts in a state outside the domain of R , then any outcome is acceptable, including non-termination.

We wish to define an ordering $\sqsubseteq$ on specifications such that the interpretation of $P \sqsubseteq Q$ is that the specification P is *stronger* than the specification Q , in the sense that any program totally correct with respect to P is totally correct with respect to Q .

Definition 1. We say that a relation P refines a relation Q , denoted by $P \sqsubseteq Q$, if and only if $QL \subseteq PL \wedge QL \cap P \subseteq Q$.

Expressions PL and QL are left vectors representing the domain of P and the domain of Q , respectively. Relation P refines relation Q if and only if the domain of Q is included in the domain of P , and the preresstriction of P to the domain of Q is included in Q . We leave it to the reader to check that this defines a partial ordering (reflexive, antisymmetric and transitive); we refer to this as the *refinement ordering*. To illustrate this definition, we consider the declaration **space** $s : \mathbf{Int}$ and we let P and Q be defined as follows:

$$P \triangleq \{s' \leq s + 1\}, \quad Q \triangleq \{s' \leq s + 3\}.$$

Then P refines Q because they have the same domain and P is a subset of Q . If we now consider the following definition of R :

$$R \triangleq \{s \leq 10 \wedge s' \leq s + 1\},$$

then we find that P refines R because R has a smaller domain, and they both have the same image sets.

3. Structuring relational specifications

With the introduction of the refinement ordering, we can now be more explicit as to what represents a *correctness-preserving* transformation. In the process of transforming a specification into a program, we must ensure that each step maps a relation R into a relation R' such that $R' \sqsubseteq R$. Also, as a measure of separation of concerns, we wish to ensure that whenever a component of a specification is refined, the whole specification is refined; to do so, we take the position that all the operators that we use to structure our specifications must be monotonic with respect to the refinement ordering. In this section, we introduce a number of relational operators that we use to

structure our specifications; for each such operator, we give a simple illustration and a formal semantic definition.

3.1. Demonic meet

In [5], it was found that the refinement ordering confers a lattice-like structure on the set of relations, where the meet (greatest lower bound) exists conditionally. The demonic meet operator is defined if and only if the following condition is satisfied:

$$PL \cap QL \subseteq (P \cap Q)L. \quad (1)$$

Whenever the condition is satisfied, the meet is defined by

$$P \sqcap Q \triangleq (\overline{QL} \cap P) \cup (P \cap Q) \cup (\overline{PL} \cap Q). \quad (2)$$

We assign to $\sqcap$ the same precedence as $\cap$. Condition (1) can be interpreted as: P and Q are mutually consistent, i.e., they do not contradict each other; we call this the *consistency condition*, which we denote by $cs(P, Q)$. Two relations contradict each other when they provide disjoint sets of outputs for some input where they are both defined. The meet can be interpreted as the specification that carries all the requirements of P and all the requirements of Q and nothing more; hence, the meet is indeed adding up the specifications. The formula defining meet may be understood as follows: the term $\overline{QL} \cap P$ provides that the meet of P and Q behaves like P for inputs where only P is defined; similarly, the term $\overline{PL} \cap Q$ provides that the meet behaves like Q for inputs where only Q is defined; the term $P \cap Q$ provides that the meet behaves like P and Q for inputs where both P and Q are defined. The domain of $P \sqcap Q$ is the union of the domains of P and Q .

To illustrate this operator, we consider the case where we want a program that merges two sorted lists l_1 and l_2 into a list l_3 . The following is a possible specification:

$$\{prm(l_1 \bullet l_2, l'_3)\} \sqcap \{std(l_1) \wedge std(l_2) \wedge std(l'_3)\}.$$

This specifications requires a program to produce a list l_3 which is a permutation of the concatenation of the input lists l_1 and l_2 . Moreover, when the input lists are sorted, the final list l_3 is also sorted.

3.2. Demonic join

In [5], it was found that any pair of relations has a join (least upper bound) in the refinement ordering. It is defined as follows:

$$P \sqcup Q \triangleq PL \cap QL \cap (P \cup Q). \quad (3)$$

We assign to $\sqcup$ the same precedence as $\cup$. Demonic join offers a choice between two specifications: a computation satisfies a specification $P \sqcup Q$ if it satisfies P or Q . As

an illustration of this operator, consider the following equalities, where the space is **space** $a : \mathbf{Real}$:

$$\begin{aligned} \{s'^2 = s \wedge s' \geq 0\} \sqcup \{s'^2 = s \wedge s' \leq 0\} &= \{s'^2 = s\} , \\ \{s'^2 = s\} \sqcup \{-(s'^2) = s\} &= \{s' = s = 0\} . \end{aligned}$$

The domain of an expression $P \sqcup Q$ is the intersection of the domains of P and Q ; on this domain, $P \sqcup Q$ is the union of P and Q .

Under the refinement ordering, relations form a complete $\sqcup$ -semilattice [8], i.e., a least upper bound exists for any non-empty set of relations. The definitions of meet and join can be generalized to the infinitary case. When the meets involved are defined and $\mathcal{A}$ is not empty, the following identities hold [8].

$$\begin{aligned} (a) \quad P \sqcap \prod_{Q \in \mathcal{A}} Q &= \prod_{Q \in \mathcal{A}} P \sqcap Q, & (b) \quad t \cap (Q \sqcup R) &= (t \cap Q) \sqcup (t \cap R), \\ (c) \quad P \sqcup \bigsqcup_{Q \in \mathcal{A}} Q &= \bigsqcup_{Q \in \mathcal{A}} P \sqcup Q, & (d) \quad t \cap (Q \sqcap R) &= (t \cap Q) \sqcap (t \cap R), \\ (e) \quad P \sqcap Q &\subseteq P \cap Q, & (f) \quad P \sqcap Q &\subseteq P \cup Q, \\ (g) \quad R &= t \cap R \sqcap \bar{t} \cap R. \end{aligned} \tag{4}$$

3.3. Demonic composition

The traditional relational product is not monotonic with respect to the refinement ordering. Hence, we introduce the *demonic composition* operator [2, 4], which captures the idea of sequential composition and is monotonic with respect to the refinement ordering:

$$P \sqcap Q \triangleq PQ \cap \overline{PQL} . \tag{5}$$

We assign to $\sqcap$ the same precedence as $\circ$. The set-theoretic interpretation of demonic composition is more intuitive:

$$P \sqcap Q \triangleq \{(s, s') \mid (s, s') \in PQ \wedge s.P \subseteq \text{dom}(Q)\} ,$$

where $s.P$ stands for the set of images of s by relation P , and $\text{dom}(Q)$ stands for the domain of relation Q . It is noteworthy that when P is deterministic (i.e., $\widehat{P}P \subseteq I$) or when Q is total, the demonic product of P by Q is the same as the traditional relational product. In [7], it is shown how the definition of demonic composition can be obtained from *relational flow diagrams* [17], which provide a general definition of programming constructs. As an example of demonic composition, we have the following equalities, where $s : \mathbf{Real}$:

$$\begin{aligned} (i) \quad \{s'^2 = s\} \sqcap \{s'^2 = s\} &= \{s = s' = 0\} , \\ (ii) \quad \{s'^2 = s \wedge s' \geq 0\} \sqcap \{s'^2 = s\} &= \{s'^4 = s\} . \end{aligned}$$

In (i), the first square root relation may return negative outputs for which the second square root is not defined; hence the demonic composition is only defined for $s = 0$. In

(ii), the range of the first square root relation is included in the domain of the second relation; hence, the demonic composition is equal to the relational product. Demonic composition satisfies the following laws when the meets involved exist.

$$\begin{aligned}
 (a) \quad & P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R, & (b) \quad & P \sqcap I = I \sqcap P = P, \\
 (c) \quad & P \sqcap \bigsqcup_{Q \in \mathcal{A}} Q = \bigsqcup_{Q \in \mathcal{A}} P \sqcap Q, & (d) \quad & (\bigsqcup_{P \in \mathcal{A}} P) \sqcap Q = \bigsqcup_{P \in \mathcal{A}} P \sqcap Q, \\
 (e) \quad & P \sqcap \bigsqcap_{Q \in \mathcal{A}} Q \sqsubseteq \bigsqcap_{Q \in \mathcal{A}} P \sqcap Q, & (f) \quad & (\bigsqcap_{P \in \mathcal{A}} P) \sqcap Q \sqsubseteq \bigsqcap_{P \in \mathcal{A}} P \sqcap Q.
 \end{aligned} \tag{6}$$

3.4. Demonic closure

If we consider that specification Q can be satisfied by a sequential combination of specification P an arbitrary number of times, we use the *demonic closure* operator, which is defined as follows.

Definition 2. Let $P^{\Box} \triangleq I$ and let $P^{[n+1]} \triangleq P \sqcap P^{\Box}$. The *demonic reflexive transitive closure* (or simply *demonic closure* for short) of a relation P is defined as $P^{\Box} \triangleq \bigsqcup_{i \geq 0} P^{[i]}$.

We assign to $\Box$ the same precedence as $\wedge$ and $\neg$.

As an illustration of demonic closure, we consider the specification of a permutation. If we observe that a natural way to define a permutation of an array is by an arbitrary number of swaps of pairs of cells, then we can write a permutation Prm as

$$Prm = Swap^{\Box}.$$

Because it is based on demonic product and demonic meet, demonic closure may differ significantly from the usual closure (given by $P^* \triangleq \bigcup_{i \geq 0} P^i$, with $P^0 \triangleq I$ and $P^{i+1} \triangleq PP^i$), as the following example illustrates:

$$\{s'^2 = s\}^* = \{\exists n \geq 0 : s'^{2^n} = s\},$$

$$\{s'^2 = s\}^{\Box} = \{s = s' = 0\},$$

where $s : \mathbf{Real}$. In the particular case where P is total, the two closures are the same. Our demonic closure can be understood as an iteration without an exit condition. Typically, a demonic closure will be combined (by the demonic meet) with another specification which provides the exit condition. An illustrative example of this pattern is the specification of the sort program, which can be written as

$$Sort \triangleq Swap^{\Box} \sqcap Sorted,$$

where *Sorted* indicates that the output array is sorted.

The demonic closure satisfies the following properties:

$$\begin{aligned}
 (a) \quad & P \sqsubseteq Q \Rightarrow P^{\Box} \sqsubseteq Q^{\Box} & (b) \quad & I \sqsubseteq P \wedge P \sqcap P \sqsubseteq P \Leftrightarrow P = P^{\Box} \\
 (c) \quad & P^{\Box} = P^{\Box} \sqcap P^{\Box} & (d) \quad & (t \cap P \cap \bar{t} \cap I)^{\Box} \sqsubseteq P^{\Box}
 \end{aligned} \tag{7}$$

3.5. Prerestriction

Whenever we want to limit the domain of a specification (relation) P to a particular subset defined by the left vector t , we use the *prerestriction* operator, which we write as follows:

$$t \rightarrow P \triangleq t \cap P . \quad (8)$$

While intersection is not monotonic with respect to the refinement ordering, this particular form of intersection (by a left vector) is. We refer the reader to the left vector laws of (4), which can be interpreted as properties of $\rightarrow$. We assign to $\rightarrow$ the same precedence as $\cap$ and $\sqcap$.

3.6. Preserve

In program construction by parts, we may sometimes want to keep the program state within some set in order to satisfy a particular requirement. This occurs quite typically in loops: in order to satisfy the loop invariant, we must keep the program state within a particular set from one iteration to the next. The specification that provides that states originating in set T (represented by left vector t) remain in T is defined by

$$t^\diamond \triangleq t \rightarrow \hat{t} . \quad (9)$$

We assign to the *preserve* operator $^\diamond$ the same precedence as $\wedge$, $-$ and $\sqcap$. The preserve operator is an instance of the prerestriction operator; it is not monotonic wrt $\sqsubseteq$, because of the occurrence of $\wedge$ in its definition. For simplicity, if a is a right vector, we write $a^\diamond$ instead of $\hat{a}^\diamond$ (in fact, both forms are equivalent if we extend the definition of $^\diamond$ to take right vectors as operands). The preserve operator satisfies the following laws, where t is a left vector.

$$(a) \ t^\diamond = t^{\diamond\sqcap}, \quad (b) \ \hat{t} = \hat{t} \sqcap t^\diamond . \quad (10)$$

3.7. Local variables

In the process of refining a specification, it is sometimes necessary to introduce a new variable in order to find suitable refinements. The expression

$$\mathbf{var} \ x : T \ P$$

introduces a local variable x of type T . The initial value of x is arbitrary in type T ; its final value is lost when the execution of P is completed. The *scope* of a declaration $\mathbf{var} \ x : T$ extends as far as possible on the right, and is limited by parentheses if necessary.

To define the semantics of declarations precisely we need an auxiliary notion. The *context* of a relation R is the set of declarations whose scope covers R . The context of a relation at the outermost nesting level is the **space** declaration. Assume now a

declaration $\text{var } x : T \ P$ with context w . Then the context of P is defined to be w extended by variable x ; we denote the space of P by S_{w+x} . The space of $\text{var } x : T \ P$ is denoted by S_w . Variable x must not already be declared in w .

Let $\Pi \triangleq \{(s, s') \in S_{w+x} \times S_w \mid \exists t : s = \langle s', t \rangle\}$ be the *projection relation* from S_{w+x} to S_w . Then we define the relation $\text{var } x : T \ P$ as the *demonic projection* of relation P on space S_w . We express this formally as

$$\text{var } x : T \ P \triangleq \widehat{\Pi} \sqcap P \sqcap \Pi .$$

The following laws hold.

$$(a) \ P \sqsubseteq Q \Rightarrow \text{var } x : T \ P \sqsubseteq \text{var } x : T \ Q, \quad (b) \ \text{var } x : T \ \Pi P \widehat{\Pi} \sqsubseteq P. \quad (11)$$

Law (11 a) provides that variable introduction is monotonic. Law (11 b) highlights how local variables are introduced: a relation P is embedded in the enlarged space, through $\Pi P \widehat{\Pi}$.

4. Program construction by parts

4.1. A programming paradigm

We consider a specification R that is structured as a meet of simpler specifications, say R' and R'' : $R \triangleq R' \sqcap R''$. Specifications R' and R'' are typically simpler than R , because they capture partial requirements. As a matter of separation of concerns, we propose to solve R' and R'' in turn, then combine their solutions to find a solution for R . This is the paradigm of *program construction by parts*. In the sequel we discuss in turn: how to refine subspecifications; then how to combine partial solutions to form a solution for the whole specification.

4.2. Refining subspecifications

Given a specification R structured as the meet of two subspecifications R' and R'' , we consider the question of how to refine R' and R'' in turn. The refinement of a subspecification (say, R') involves a tradeoff between two requirements: refine R' sufficiently to register some progress in the refinement process; do not refine R' too much, for fear that it becomes irreconcilable with R'' (i.e., R' and R'' no longer satisfy the consistency condition (Eq. (1))).

To illustrate this tradeoff, we consider again the specification of a sort program:

$$\text{Sort} \triangleq \text{Prm} \sqcap \text{Sorted} .$$

Relation Prm is refined by the identity relation, I . Yet, Sort is not refined by $I \sqcap \text{Sorted}$ because I and Sorted do not satisfy the consistency condition. This is a case where excessive refinement leads to a dead end in the refinement process, even though the

original specification does have a refinement. A more reasonable refinement of *Prm* would be to rewrite *Prm* as the demonic closure of specification *Swap*, which we presented in Section 3.4. Then *Sort* can be refined as follows:

$$\text{Sort} \sqsubseteq \text{Swap}^{\square} \sqcap \text{Sorted}.$$

We leave it to the reader to check that the components of the meet do satisfy the consistency condition.

This example raises the issue of whether we must check the consistency of two subspecifications whenever we perform a refinement. The proposition below provides that the programmer may check consistency arbitrarily seldom in the refinement process – provided that he checks it at the end.

Proposition 3. *Let P, P', Q, Q' be relations such that $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. If specifications P and Q satisfy the consistency condition (Eq. (1)), then so do specifications P' and Q' .*

We will discuss further the issue of checking the consistency condition in Section 4.3.2, Proposition 10.

4.3. Combining subspecifications

The problem that we address in this section is the following: given that we have refined the arguments of a demonic meet, we must now combine them into a single program. In this section we give the rewriting rules that allow us to carry out this transformation. One of the objectives of the rules is to eliminate meet operators from specifications, because the demonic meet cannot be implemented in a compiler, to the extent of our knowledge.

4.3.1. Eliminating meets

A trivial way to get rid of a meet is when the arguments of the meet are identical.

Proposition 4. *A specification of the form $P \sqcap P$ is refined by P .*

This proposition stems readily from the idempotence of lattice operators. It represents the *bottom of recursion* of a recursive process which consists in pushing meets deeper and deeper into the nesting structure of the specification. The inductive step of this recursive process is discussed in Section 4.3.2.

Another way to eliminate meets is to refine a specification by a Pascal-like relation. Although Pascal is a deterministic language, we may provide relational nondeterministic definitions of the basic constructs of Pascal.

Definition 5. Let $x_1, \dots, x_n$ be the variables of a specification, let e be an expression on the corresponding space, let P and Q be relation, let t be a left vector, and let F

be a function in X on relations given by $\bar{t} \cap I \cup (t \cap P) \sqcap X$. Then

$$x_i := e \triangleq \{x'_i = e \wedge \forall j \in 1..n : i \neq j \Rightarrow x'_j = x_j\} ,$$

$$P; Q \triangleq P \sqcap Q ,$$

$$\text{if } t \text{ then } P \text{ else } Q \text{ end} \triangleq t \cap P \cup \bar{t} \cap Q ,$$

$$\text{if } t \text{ then } P \text{ end} \triangleq \text{if } t \text{ then } P \text{ else } I \text{ end} ,$$

$$\text{while } t \text{ do } P \text{ end} \triangleq \text{greatest fixpoint of } F \text{ wrt } \sqsubseteq .$$

Note that these constructs are relational expressions (this is also the case of the **var** construct introduced in Section 3.7). They are a means of structuring complex relational specifications. In the particular case where P and Q are the semantics of programs, these definitions are consistent with traditional relational semantics [15]. The following propositions identify several patterns of meet-structured specifications which can be refined by the above constructs. These propositions are given without proof; their proofs can be found in [9].

Proposition 6. *When condition $P \cap QL \subseteq Q$ is satisfied, the following equality holds, where t is defined by $t = PL$:*

$$P \sqcap Q = \text{if } t \text{ then } P \text{ else } Q \text{ end} .$$

Proposition 7. *The following equality holds for any left vector t and specifications P and Q :*

$$t \rightarrow P \sqcap \bar{t} \rightarrow Q = \text{if } t \text{ then } P \text{ else } Q \text{ end} .$$

In the next proposition, we use the following concept to represent loop termination: a relation R is *progressively finite* [17] if and only if there exists no infinite sequence $s_0, s_1, \dots, s_i, \dots$ such that $(s_i, s_{i+1}) \in R$ for all i .

Proposition 8. *When the conditions*

- (i) $t \rightarrow P$ *is progressively finite,*
 - (ii) $t \subseteq PL$,
- are satisfied the following equality holds:*

$$\text{if } t \text{ then } P \text{ end}^{\square} \sqcap \widehat{t} = \text{while } t \text{ do } P \text{ end} .$$

4.3.2. Propagating meets

If two relational expressions combined with a meet share the same structure, then the meet can be pushed inside the structure. The resulting meets have smaller operands, and hopefully they become easier to eliminate. We express this law for generic monotonic operators in the following proposition.

Proposition 9. Let $\Phi(P_1, \dots, P_n)$ be a monotonic operator in all its operands with respect to $\sqsubseteq$. Then the following holds if the meets are defined:

$$\Phi(P_1 \sqcap Q_1, \dots, P_n \sqcap Q_n) \sqsubseteq \Phi(P_1, \dots, P_n) \sqcap \Phi(Q_1, \dots, Q_n) .$$

As an example, we instantiate this proposition for demonic composition and demonic closure. Of course, these laws hold only if the meets are defined.

$$(P_1 \sqcap Q_1) \sqcap (P_2 \sqcap Q_2) \sqsubseteq P_1 \sqcap P_2 \sqcap Q_1 \sqcap Q_2 \quad (12)$$

$$(P \sqcap Q)^\square \sqsubseteq P^\square \sqcap Q^\square \quad (13)$$

A typical use of the refinement rule (12) arises in cases where the specification at hand has the form $(P_1 \sqcap P_2) \sqcap Q$. In order to apply the refinement, we may decompose Q as $Q \sqcap X$, if we wish to match Q with P_1 , or as $X \sqcap Q$, if we wish to match Q with P_2 . In both cases, X should be chosen in such a way as to be maximal (least refined) in the refinement ordering. We now introduce two operators to compute these greatest solutions.

We denote by $R \parallel Q$ (*demonic left residue*) the greatest solution in X , wrt $\sqsubseteq$, of the inequation $X \sqcap Q \sqsubseteq R$. Similarly, we denote by $Q \backslash R$ (*demonic right residue*) the greatest solution in X , wrt $\sqsubseteq$, of the inequation $Q \sqcap X \sqsubseteq R$. Operators $\parallel$ and $\backslash$ are partial, since some inequations do not have a solution. We refer the interested reader to [8] for more details. Among the properties of $\parallel$ and $\backslash$, we mention the following, because they proved useful in the construction of loops:

$$(a) P \parallel P = (P \parallel P)^\square \quad (b) P \backslash P = (P \backslash P)^\square \quad (14)$$

The set-theoretic interpretation of $P \parallel P$ is $\{(s, s') \mid s' \in \text{dom}(P) \wedge s'.P \subseteq s.P\}$. Hence, $P \parallel P$ strengthens or preserves (in the typical case where $s.P = s'.P$) properties of the state related to the application of P . It is thus an expression of the properties kept invariant by P . Similar comments can be made for $P \backslash P$, after noting that $P \backslash P = (\hat{P} \parallel \hat{P})^\wedge$.

The following proposition states that it is not possible to propagate an undefined meet and obtain defined meets. Recall that $cs(P, Q)$ represents the consistency of P and Q (Section 3.1).

Proposition 10. Let P_1, P_2, Q_1, Q_2 be relations; let Φ be a monotonic operator wrt $\sqsubseteq$ on relations. Then, we have

$$cs(P_1, Q_1) \wedge cs(P_2, Q_2) \Rightarrow cs(P_1 \Phi P_2, Q_1 \Phi Q_2) .$$

A consequence of this proposition is that the programmer may delay the verification of the consistency condition, even if meets are propagated. In fact, it is seldom required to prove consistency using Eq. (1): if one is successful in removing a meet using Propositions 4, 6, 7 or 8 then consistency is proved as a by-product. One would need to prove consistency using the definition only if it seems impossible to eliminate the meet, as a means of determining if components were refined up to a point where they can no longer be satisfied simultaneously.

5. The Naur problem

In this section, we apply our program construction paradigm to the Naur problem. Note that the purpose of this section is not to solve the Naur problem as much as it is to illustrate our method. First, we present the English version of this problem, derived from [16], and then we present the formal relational specification.

5.1. The user requirements

Given are a non-negative integer M and a character set which includes two *break* characters, viz., $\mathfrak{b}$ (*blank*) and $\mathfrak{n}$ (*newline*). For a sequence s of characters, we define a *word* as a non-empty sequence of consecutive non-break characters embedded between break characters or the endpoints of s (i.e., the beginning and end of sequence s). The program shall accept as input a finite sequence of characters and produce as output a sequence of characters satisfying the following conditions.

- (i) If the input includes at least one break character for any consecutive $M + 1$ characters, then all the words of the input appear in the output, in the same order; all the words of the output appear in the input.
- (ii) Furthermore, the output must meet the following conditions:
 - (a) It contains no leading or trailing breaks, nor does it have two consecutive breaks.
 - (b) Any sequence of $M + 1$ consecutive characters includes a newline.
 - (c) Any sequence made up of no more than M consecutive characters and embedded between (the head of the output or a newline on the left) and (the tail of the output or a break on the right) does not contain a newline character.

5.2. The relational specification

Before we proceed with writing a formal specification for this problem, we introduce the following notations: we denote by chr the set of characters under consideration, including the *newline* symbol; also, we let this set be partitioned into the set of *text* characters, which we denote by txt and the set of *breaks*, which we denote by brk ; we let $x \bullet y$ stand for the concatenation of sequences x and y ; we let ε denote the empty sequence; we let X^+ denote the closure under $\bullet$ of the set of sequences X ; we let X^* denote $X^+ \cup \{\varepsilon\}$; we let $\#x$ denote the length of sequence x ; we let $x \preceq y$ be the predicate “ x is a subsequence of consecutive characters of y ”; we let $fw.x$ (first word) denote the first word of sequence x , provided that x contains a word; we let $rw.x$ (remaining words) denote the sequence beginning after the first word of x and terminating at the end of x , provided that x contains a word; we let $lw.x$ (long word) be the predicate “sequence x contains a word of length $> M$ ”; we let $lfw.x$ (long first word) be the predicate “the length of the first word of sequence x is $> M$ ”; we let sw (sequence of words) be a function that maps

a sequence of characters x to the sequence of words of x (e.g., $sw.\langle babcd \rangle = \langle \langle ab \rangle \langle cd \rangle \rangle$).

The space of our specification is given by the following declaration:

space $in, out : seq\ chr$.

We let relation *binary* represent clause (i) of the user requirements. Note that clause (i) does not require *binary* to terminate on input sequences containing a long word. Relation *binary* can be written as

$$binary \triangleq \{ \neg lw.in \wedge sw.in = sw.out' \} .$$

Clauses (a), (b) and (c) are conditions on the output only. They are represented, respectively, by the following right vectors:

$$\begin{aligned} noexb &\triangleq \{ out' \in txt^+ \bullet (brk \bullet txt^+)^* \cup \{ \varepsilon \} \} , \\ sht &\triangleq \{ \forall x : x \preceq out' \wedge \#x = M + 1 \Rightarrow \mathbf{d} \preceq x \} , \\ lng &\triangleq \{ \forall x, y, z : out' = x \bullet y \bullet z \wedge x \in chr^* \bullet \mathbf{d} \cup \{ \varepsilon \} \wedge \\ &\quad \#y \leq M \wedge z \in brk \bullet chr^* \cup \{ \varepsilon \} \Rightarrow \mathbf{d} \not\preceq y \} . \end{aligned}$$

Given *noexb*, *sht* and *lng*, we let *unary* be defined as

$$unary \triangleq noexb \sqcap sht \sqcap lng .$$

The complete specification of the Naur problem is given by

$$Naur \triangleq binary \sqcap unary .$$

We leave it to the reader to check that the components are consistent, hence the meet is defined. The structure of the Naur specification is typical of meet-structured specifications: some components represent the relationship between initial states and final states (i.e., *binary*), whereas others, defined by right vectors, represent a condition on the output (i.e., *unary*). Note that, given the definition of demonic meet, a computation satisfying *Naur* must terminate for an input sequence containing a long word, because *unary* is a total relation. However, since *binary* is not defined for an input sequence containing a long word, the computation needs only satisfy relation *unary*.

5.3. The solution

Our strategy is to solve *binary* and *unary* independently, and then to progressively combine their solutions to come up with a program satisfying the complete specification.

5.3.1. Solving unary

We propose an iterative solution for *unary*, which is general enough to preserve consistency with *binary*, but highlights the structure of a solution to *unary*.

$$\begin{aligned}
 & \text{unary} \\
 = & \quad \{ \text{law (10 b)} \} \\
 & \text{unary} \sqcap \text{unary}^\diamond \\
 = & \quad \{ \text{law (10 a)} \} \\
 & \text{unary} \sqcap \text{unary}^{\diamond\Box}
 \end{aligned}$$

We stop the refinement process at this point in order not to overcommit ourselves.

5.3.2. Solving binary

We are targeting an iterative solution for *binary*. Our strategy consists in decomposing *binary* into a format such that one component allows the introduction of a demonic closure. Considering the rules provided in this paper, it means that this component satisfies law (7 b); in particular this component can be a preserve (law (10 a)) or a demonic residue (laws (14)). The latter form is more appropriate for the solution of *binary*, as we shall see in the sequel.

$$\begin{aligned}
 & \text{binary} \\
 = & \quad \{ \text{def. of binary} \} \\
 & \{ \neg lw.in \wedge sw.in = sw.out' \} \\
 \sqsupseteq & \quad \{ \varepsilon \text{ is the identity of } \bullet \} \\
 & \{ \neg lw.in \wedge sw.in = sw.out' \bullet sw.in' \wedge sw.in' = \varepsilon \}
 \end{aligned}$$

Let us introduce the following abbreviations which will be used in the sequel:

$$\begin{aligned}
 \text{init} & \triangleq \{ \neg lw.in \wedge sw.in = sw.out' \bullet sw.in' \} , \\
 mw & \triangleq \{ sw.in \neq \varepsilon \} .
 \end{aligned}$$

Abbreviation *init* corresponds to the first and second terms of the conjunction in the previous refinement. Abbreviation *mw* is the unprimed negation of the third term of the conjunction; it stands for “more words”, i.e., *in* contains at least one word. We pursue the refinement of *binary*.

$$\begin{aligned}
 & = \quad \{ \text{abbreviations above, set theoretic definition of } \cap \} \\
 & \text{init} \cap \overline{mw}^\wedge \\
 \sqsupseteq & \quad \{ \text{law (4 e)} \} \\
 & \text{init} \cap \overline{mw}^\wedge \\
 \sqsupseteq & \quad \{ \text{definition of } \setminus \text{ (Section 4.3.2)} \} \\
 & \text{init} \sqcap (\text{init} \setminus \text{init}) \cap \overline{mw}^\wedge
 \end{aligned}$$

Let us introduce another abbreviation:

$$pw \triangleq \text{init} \setminus \text{init} = \{ \neg lw.in \wedge \neg lw.out \wedge sw.out \bullet sw.in = sw.out' \bullet sw.in' \} .$$

Abbreviation *pw* stands for “preserve word sequence”. We pursue the refinement of *binary*.

$$\begin{aligned}
 &= \{ \text{abbreviation } pw \} \\
 &\quad init \sqcap pw \sqcap \overline{mw}^\wedge \\
 &= \{ \text{law (14 b)} \} \\
 &\quad init \sqcap pw^\square \sqcap \overline{mw}^\wedge
 \end{aligned}$$

We are close to a relational expression equivalent to a **while** loop (Proposition 8). The next steps will bring us up to that point.

$$\begin{aligned}
 &= \{ \text{for any left vector } t : \hat{t} = L \sqcap \hat{t} \} \\
 &\quad init \sqcap pw^\square \sqcap L \sqcap \overline{mw}^\wedge \\
 &\sqsupseteq \{ \text{law (12)} \} \\
 &\quad (init \sqcap L) \sqcap (pw^\square \sqcap \overline{mw}^\wedge)
 \end{aligned}$$

The second meet clearly has a structure appropriate for a conversion into a **while**, given some additional refinements. We postpone these refinements to the next paragraph.

5.3.3. Merging binary with unary.

We now merge the solutions of *binary* and *unary*. The process is almost a mechanical application of laws, except for the selection of a progressively finite relation in the second-to-last step.

$$\begin{aligned}
 &\quad binary \sqcap unary \\
 &\sqsupseteq \{ \text{refinement of unary and binary above} \} \\
 &\quad unary \sqcap unary^\diamond \sqcap (init \sqcap L) \sqcap (pw^\square \sqcap \overline{mw}^\wedge) \\
 &\sqsupseteq \{ \text{law (12)} \} \\
 &\quad (unary \sqcap init \sqcap L) \sqcap (unary^\diamond \sqcap pw^\square \sqcap \overline{mw}^\wedge)
 \end{aligned}$$

The first component of the sequence in the last step has a simple refinement: $\{out' = \varepsilon \wedge in' = in\}$. We pursue the refinement for the second component of the sequence

$$\begin{aligned}
 &\quad (unary^\diamond \sqcap pw^\square \sqcap \overline{mw}^\wedge) \\
 &\sqsupseteq \{ \text{law (13)} \} \\
 &\quad (unary^\diamond \sqcap pw)^\square \sqcap \overline{mw}^\wedge \\
 &\sqsupseteq \{ \text{law (7 d), Definition 5} \} \\
 &\quad \text{if } mw \text{ then } unary^\diamond \sqcap pw \text{ end}^\square \sqcap \overline{mw}^\wedge \\
 &\sqsupseteq \{ pf \triangleq \{in' = rw.in\} \} \\
 &\quad \text{if } mw \text{ then } unary^\diamond \sqcap pw \sqcap pf \text{ end}^\square \sqcap \overline{mw}^\wedge \\
 &= \{ \text{Proposition 8} \} \\
 &\quad \text{while } mw \text{ do } unary^\diamond \sqcap pw \sqcap pf \text{ end}
 \end{aligned}$$

Let us now pause for a moment and see how the merge was done. The solutions of *unary* and *binary* share the same structure, therefore it is possible to push the meet inside the structure. The result is a sequence of two meets. The first one being quite easy to refine, we focus our attention on the second component. Its structure is close

to a pattern transformable into a **while** loop. We apply two rules to refine it into a pattern equivalent (under the conditions of Proposition 8) to a **while-do**.

We can also solve the loop body using the construction by parts approach. In that case, the operators $\parallel$ and $\backslash$ play a key role in the derivation of a solution. We also need to introduce a local variable, k , which keeps track of the length of the last line of *out*. The reader is referred to [9] for a detailed derivation. We obtain the final program shown below.

```

space in, out : seq chr
var k : Int
out :=  $\varepsilon$ ; k := 0;
while mw do
  if  $\neg \text{!fw.in}$  then
    if out =  $\varepsilon$  then k :=  $\# \text{fw.in}$ 
    else if  $k + \# \text{fw.in} \geq M$  then k :=  $\# \text{fw.in}$ ; out := out •  $\mathbf{u}$ 
    else k :=  $k + \# \text{fw.in} + 1$ ; out := out •  $\mathbf{b}$  end
    end;
    out := out • fw.in
  end;
  in := rw.in
end

```

We may also consider other solutions to the Naur problem – our paradigm is not restrictive in this respect. For instance, it is possible to envisage other loop conditions, or to simplify the loop body by removing the first two **if** conditions and by modifying the loop initialization.

6. Conclusion

In earlier work [5] we had shown that specifications are naturally structured as aggregates of component specifications, using the lattice based meet operator. In this paper we discuss how this structure can be used as a basis for separation of concerns in program construction: we can refine each argument of the meet separately then combine the refinements to get a solution to the overall specification. To be sure, the refinements of the individual components are not totally independent: while refining a component, we must keep an eye on the neighboring components, with a view to remaining consistent with them (keeping the consistency condition true). This could be viewed as a constraint; we prefer to view it as a means to guide the programmer, in the sense that the structure of one component is hinted at by the structure of its neighboring component(s). The mathematics we have developed for our programming paradigm appears to be satisfactory: first, they enable us to produce arbitrarily non-deterministic solutions to the component specifications; second, they faithfully reflect the stepwise negotiation process whereby two component specifications negotiate a

common refinement; third, they do show how the refinement of individual components may lead to failure if it is carried out too independently. We have illustrated our method on a simple example, where our emphasis is more on illustrating the method than on solving the example.

Program construction by parts is not new: Hehner discusses it in [11], in the case where specifications are represented by predicates. Taking the greatest lower bound of programs is not new: Hoare et al. talk about it in [12], where they argue that meet is the most useful operator for structuring large specifications. Von Wright [19] defines a language based on join and meet. Gardiner and Morgan [10] use join and meet for data refinement.

Our work differs from Hehner's work by the representation of specifications (predicates vs. relations) and by their interpretations: termination is implicit in our specifications, whereas it is expressed as timing constraints in Hehner's. Our work differs from that of Hoare et al. [12] by using partial relations and demonic operators, by not using a fictitious state to represent non-termination, and by providing rules to eliminate meets in a specification. Our work differs from the work of von Wright, Gardiner and Morgan by using a different semantics (denotational semantics based on relations, versus axiomatic semantics based on predicate transformers), by not allowing miraculous specifications, and by studying the transformation of meet-structured specifications. We share with the work of Sekerinski [18] (and Z) the same specification model where the focus is on input–output pairs for which a program must terminate.

Finally, we have found that our method for program construction is also very well suited for program *modification*: adding a feature F to a specification P is presented as solving the specification $P \sqcap F$. The construction by parts approach allows an efficient reuse of the solution to P in the solution of $P \sqcap F$. We are pursuing investigations in this area.

Acknowledgements

The authors are grateful to the anonymous reviewers of the Third International Conference on the Mathematics of Program Construction and E.C.R. Hehner for helpful comments on previous versions of this paper.

References

- [1] R.C. Backhouse, et al., A relational theory of data types, Dept. of Math. and Comp. Sc., Eindhoven University of Technology, The Netherlands, 1992.
- [2] R.C. Backhouse and J. van der Woude, Demonic operators and monotype factors, *Math. Struct. Comput. Sci.* **3** (1993) 417–433.
- [3] R. Berghammer and G. Schmidt, Relational Specifications, in: C. Rauszer, ed., *XXXVIII Banach Center Semesters on Algebraic methods in Logic and their Computer Science Applications* (1993) 167–190.
- [4] R. Berghammer and H. Zierer, Relational algebraic semantics of deterministic and nondeterministic programs, *Theoret. Comput. Sci.* **43** (1986) 123–147.

- [5] N. Boudriga, F. Eloumi and A. Mili, On the lattice of specifications: applications to a specification methodology, *Formal Aspects Comput.* **4** (1992) 544–571.
- [6] J. Desharnais et al., A relational division operator: the conjugate kernel, *Theoret. Comput. Sci.* **114** (1993) 247–272.
- [7] J. Desharnais, F. Tchier and R. Khédri, Demonic relational semantics of sequential programs, Research Report DIUL-RR-9406, Département d'informatique, Université Laval, Québec City, Canada, 1994.
- [8] J. Desharnais et al., Embedding a demonic semilattice in a relation algebra, *Theoret. Comput. Sci.*, to appear.
- [9] M. Frappier, *A Relational Basis for Program Construction by Parts*, Dept. of Computer Science, University of Ottawa, 1995.
- [10] P. Gardiner and C.C. Morgan, Data refinement of predicate transformers, *Theoret. Comput. Sci.* **87** (1991) 143–162.
- [11] E.C.R. Hehner, *A Practical Theory of Programming* (Springer, Berlin 1993).
- [12] C.A.R. Hoare et al., Laws of programming, *Comm. ACM* **30** (1987) 672–686.
- [13] G. Jones and M. Sheeran, Designing arithmetic circuits by refinement in ruby, in: R.S. Bird, C.C. Morgan and J.C.P. Woodcock, eds., *Mathematics of program Construction: 2nd Internat. Conf.*, Oxford, 1992, Lecture Notes in Computer Science, Vol. 669 (Springer, Berlin, 1993).
- [14] B. Möller, Relations as a program development language. in: Möller, B., ed., *Constructing Programs from Specifications*, *Proc. IFIP TC 2/WG 2.1*, Pacific Grove, CA, 13–16 May (North-Holland, Amsterdam, 1991) 373–397.
- [15] H.D. Mills et al., *Principles of Computer Programming: A Mathematical Approach* (Allyn and Bacon, Newton, MA, 1987).
- [16] P. Naur, Programming by action clusters *BIT* **9** (1969) 250–258.
- [17] G. Schmidt and T. Ströhlein, *Relations and Graphs* (Springer, Berlin, 1993).
- [18] E. Sekerinski, A calculus for predicative programming, in: R.S. Bird, C.C. Morgan and J.C.P. Woodcock, eds., *Mathematics of program construction: 2nd Internat. Conf.*, Oxford, 1992, Lecture Notes in Computer Science Vol. 669 (Springer, Berlin, 1993).
- [19] J. Von Wright, A lattice theoretical basis for program refinement, Ph.D. Thesis, Dept. of Computer Science, Åbo Akademi, Finland, 1990.