A Calculus of Program Transformations and Its Applications *

Rahma Ben Ayed
School of Engineering
University of Tunis II
Belvedere, 1002 Tunisia
rbenayed@stardent.cck.tn

Jules Desharnais
Department of Informatics
Laval University,
Quebec City, PQ G1K 7P4, Canada
Jules.Desharnais@ift.ulaval.ca

Marc Frappier
Dept of Maths and Informatics
University of Sherbrooke
Sherbrooke, PQ J1K 2R1 Canada
marc.frappier@dmi.usherb.ca

Ali Mili†
The Institute for Software Research
1000 Technology Drive
Fairmont, WV 26554 USA
amili@csee.wvu.edu

June 27, 1998

Abstract

Traditional programming calculi focus on transforming a specification into a program. With the advent of such recent paradigms as white box reuse, component based software development, and reengineering, and with the continued predominance of software maintenance, less and less software engineering activity nowadays deals with straightforward development. More and more, software products are derived by combining, merging, or adapting existing components. In this paper we present a calculus that models several aspects of these paradigms.

Keywords: Formal specifications, Programming calculi, Program construction, Software merging, Software incrementation, Software adaptation.

1 Relational Specifications

We represent specifications by relations; without much loss of generality, we consider homogeneous relations, and we denote by S the space on which relations are defined. As a specification, a relation contains all the (input, output) pairs that are considered correct. We have discussed in [6] how to use (heterogeneous) relations to represent specifications of objects, in the sense of object oriented programming. Hence even though the discussions of this paper deal with specifications of simple input/output functions, they can in fact be applied to state-bearing modules, such as objects.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

†Correspondence Author.

Constant relations include the *universal* relation, denoted by L, the *identity* relation, denoted by I, and the *empty* relation, denoted by $\emptyset$. Operations on relations include the *complement*, denoted by $\overline{R}$, the *inverse*, denoted by $\widehat{R}$, and the *product*, denoted by $R \circ R'$ or RR'. We say that R is *deterministic* if and only if $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$.

We define an ordering relation on relational specifications under the name *refinement ordering*: A relation R is said to *refine* a relation R' if and only if

$$RL \cap R'L \cap (R \cup R') = R'.$$

We then write $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit without proof that this relation is a partial ordering. We also admit that, modulo some definition of total correctness, the following propositions hold:

- A program P is correct with respect to a specification R if and only if $[P] \sqsupseteq R$, where $[P]$ is the function defined by P.
- $R \sqsupseteq R'$ if and only if any program correct with respect to R is correct with respect to R'.

The refinement ordering has some lattice-like properties which, for the sake of parsimony, we present briefly without proof (details can be found in [2]).

- Two relations R and R' have a join (denoted by $R \sqcup R'$) if and only if they satisfy the condition

  $$RL \cap R'L = (R \cap R')L.$$

  Under this condition, the join is given by

  $$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup (R \cap R').$$

  The specification $R \sqcup R'$ captures all the information of R and all the information of R'. It is satisfied by any implementation that is known to satisfy both R and R'.

- Any two relations R and R' have a meet, which is denoted by $R \sqcap R'$ and defined by

  $$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

  The specification $R \sqcap R'$ captures the requirements information that R and R' have in common. It is satisfied by any implementation that is known to satisfy either R or R'.

The join operator can be used to structure complex specifications as aggregates of simpler specifications; hence most of the manipulations we discuss in this paper can be made arbitrarily simple by by considering arbitrarily simple component specifications.

2 A Calculus of Programming by Parts

Programming by parts rests on the premise that complex specifications can be structured as joins of simpler specifications, and proceeds by solving each component of the join in turn, then combining the partially defined solutions so obtained to produce a program that satisfies all the component subspecifications simultaneously. To support this program derivation method, we provide the following tools:

- A specification/programming notation that supports specification structuring (by means of joins, meets) as well as program structuring (by means of traditional programming constructs).
- A set of guidelines for deriving partially determined programs from component subspecifications.
- A set of rules for combining partially determined programs into (more) completely determined programs.

Due to space restrictions, we can only present a flavor of this method; details of this method can be found in [2].

2.1 A Specification Notation

Specifications are represented by structured relational expressions. We use two elementary specifications, in addition to *closed form relations*, which are:

- *Establish*, which we denote by $\mathbf{est}(t)$, and define as:

  $$\mathbf{est}(t) = \{(s, s') | t(s')\}.$$

- *Preserve*, which we denote by $\mathbf{prs}(t)$, and define as:

  $$\mathbf{prs}(t) = \{(s, s') | t(s) \wedge t(s')\}.$$

For the sake of stepwise refinement, we decide that all our compound statements must be monotonic with respect to the refinement ordering; we present some of these statements, for illustrative purposes.

- *Join.* When two requirements must be satisfied simultaneously, they are combined with the join ($\sqcup$).
- *Meet.* When any one of two requirements may be satisfied, they are combined with the meet ($\sqcap$).
- *Composition.* Because the traditional relational product is not monotonic with respect to refinement, we define a monotonic sequence-like operator, which we denote by $R \circ R'$ and define as

  $$R \circ R' = RR' \cap \overline{R\overline{R'L}}.$$

- *Closure.* When a specification must be applied an arbitrary number of times, we apply the closure operator to it, to produce

  $$R^{\star} = \sqcup_{i \geq 0} R^i,$$

  where R^i is the monotonic composition of R by itself i times.

2.2 Refinement Rules: Basis of Induction

We present a sample of refinement rules that produce a programming construct or eliminate a specification construct (join, meet).

- $R \sqcup R$ is refined by R.
- $R \sqcap R'$ is refined by R (and by R').
- If R is deterministic or R' is total then $R \circ R'$ is refined by `R; R'`.

2.3 Refinement Rules: Merging

Given two components of the join that have the same structure, these rules merge the components into a single specification that has the common structure, and propagate the joins deeper into the nesting structure. A sampling of these rules is:

- If all the joins are defined, then $(R \circ R') \sqcup (Q \circ Q')$ is refined by

  $$(R \sqcup Q) \circ (R' \sqcup Q').$$

- If all the joins are defined, then $R^* \sqcup Q^*$ is refined by

  $$(R \sqcup Q)^*.$$

2.4 Refinement Rules: Structure Unification

While the rules of the previous section exploit the common structure of two subspecifications to merge them into a single specification (that has the common structure), the rules that we discuss in this section attempt to unify the structure of two components. Typically, they do so by imposing the structure of one component on the other. For the sake of illustration, we present two such rules, which impose a sequence structure on a (structureless) relation.

- R is refined by $(R /\!\!/ R) \circ R$,
- R is refined by $R \circ (R \backslash\!\!\backslash R)$,

where $(R /\!\!/ R)$ is the least refined solution to the equation $R \sqsubseteq X \circ R$ and $(R \backslash\!\!\backslash R)$ is the least refined solution to the equation $R \sqsubseteq R \circ X$. Note that for $R = \mathbf{est}(t)$, for some predicate t, the first rule yields the decomposition

$$\mathbf{est}(t) \sqsubseteq \mathbf{est}(true) \circ \mathbf{est}(t),$$

and the second rule yields the decomposition

$$\mathbf{est}(t) \sqsubseteq \mathbf{est}(t) \circ \mathbf{prs}(t).$$

2.5 Brief Illustration

For the sake of illustration, we consider the specification of a sorting program and show how we can apply some refinement steps using the method of program construction by parts.

$$
\begin{array}{ll}
 & Sort \\
\sqsubseteq & \qquad \{ \text{ join structure } \} \\
 & Perm \sqcup Ord \\
\sqsubseteq & \qquad \{ \text{ definition of } Ord \text{ } \} \\
 & Perm \sqcup \mathbf{est}(sorted) \\
\sqsubseteq & \qquad \{ \text{ } sorted \equiv psorted \wedge k = N \text{, where } psorted \text{ means partially sorted } \} \\
 & Perm \sqcup \mathbf{est}(psorted) \sqcup \mathbf{est}(k = N) \\
\sqsubseteq & \qquad \{ \text{ structure unification rules } \} \\
 & Perm \,\square\, Perm \\
 & \sqcup\ \mathbf{est}(psorted) \,\square\, \mathbf{prs}(psorted) \\
 & \sqcup\ \mathbf{est}(\mathbf{true}) \,\square\, \mathbf{est}(k = N) \\
\sqsubseteq & \qquad \{ \text{ merging rule } \} \\
 & (Perm \sqcup \mathbf{est}(psorted) \sqcup \mathbf{est}(\mathbf{true})) \\
 & \square\, (Perm \sqcup \mathbf{prs}(psorted) \sqcup \mathbf{est}(k = N)) \\
\sqsubseteq & \qquad \{ \text{ common refinement of the first factor } \} \\
 & (k := 1 \sqcup k := 1 \sqcup k := 1) \\
 & \square\, (Perm \sqcup \mathbf{prs}(psorted) \sqcup \mathbf{est}(k = N)) \\
\sqsubseteq & \qquad \{ \text{ basis of induction rule: eliminating joins } \} \\
 & (k := 1) \,\square\, (Perm \sqcup \mathbf{prs}(psorted) \sqcup \mathbf{est}(k = N)). \\
\sqsubseteq & \qquad \{ \text{ basis of induction rule: replacing } \square \text{ with } ; \text{ } \} \\
 & (k := 1)\ ;\ (Perm \sqcup \mathbf{prs}(psorted) \sqcup \mathbf{est}(k = N)).
\end{array}
$$

The first term of the sequence is the initialization segment; the second term is the specification of the while statement of an insertion sort.

3 Application: Software Incrementation

We consider a software system C and a feature F that we wish to add to C, and we are interested in how to augment C so that it has feature F. Our position is that this problem amounts to refining the specification

$$C \sqcup F.$$

Note that although C is a program, we can write it using our specification notation (we have rules for doing that, given in [2]). For the sake of argument, we assume that C and F do indeed have a join, and that C is structured as a sequence of three components, say

$$C = C_0 \,\square\, C_1 \,\square\, C_2.$$

We assume further that after analyzing C and F, we have determined that modifications of C to augment it with F are localized in component C_1. In light of this analysis, we deccompose F as

$$F \sqsubseteq (F /\!\!/ F) \,\Box\, F \,\Box\, (F \backslash\!\!\backslash F),$$

using the structure unification rules given above. It may be worthwhile at this stage to analyze the intuitive meaning of $(F /\!\!/ F)$ and $(F \backslash\!\!\backslash F)$: the term $(F /\!\!/ F)$ represents the minimal (with respect to refinement) specification that must be satisfied upstream of F in order not to destroy any information that F may need; for example, if F is the specification of a sort routine, $(F /\!\!/ F)$ is specification $Perm$, which prescribes that no cell of the array to be sorted is lost. Also, the term $(F \backslash\!\!\backslash F)$ represents the minimal (with respect to refinement) specification that must be satisfied downstream of F in order not to destroy the work achieved by F; if F is the specification of a sort routine, then $(F \backslash\!\!\backslash F)$ is the specification that prescribes that array a (which has been sorted by F) is kept intact.

Using the sequential decomposition of C and F into three terms, we find the following refinement:

$$
\begin{array}{ll}
 & C \sqcup F \\
\sqsubseteq & \qquad \{ \text{ substitution } \} \\
 & (C_0 \,\Box\, C_1 \,\Box\, C_2) \sqcup \\
 & (F /\!\!/ F) \,\Box\, F \,\Box\, (F \backslash\!\!\backslash F) \\
\sqsubseteq & \qquad \{ \text{ merging } \} \\
 & (C_0 \sqcup (F /\!\!/ F)) \\
 & \Box\, (C_1 \sqcup F) \\
 & \Box\, (C_2 \sqcup (F \backslash\!\!\backslash F)).
\end{array}
$$

We consider the first term of the product, and analyze the possibilities that may arise (the same discussion holds for the third term):

- C_0 and $(F /\!\!/ F)$ have no join; this means that the decompositions of F and C have not been lined up, and must be revised. To illustrate what it means for decompositions to be lined up, consider the following relations:

  $$R' = \{(s, s') | s' = s + 3\},\ R'' = \{(s, s') | s + 7 \leq s' \leq s + 8\}.$$
  $$Q' = \{(s, s') | s + 3 \leq s' \leq s + 4\},\ Q'' = \{(s, s') | s' = s + 7\}.$$
  $$R = Q = \{(s, s') | s + 10 \leq s' \leq s + 11\}.$$

  Relations R and Q do have a join since they are identical; if we decompose R as $R' \,\Box\, R''$ and decompose Q as $Q' \,\Box\, Q''$ then we obtain a lined up decomposition, since R' and Q' have a join (R'), and R'' and Q'' have a join (Q''); but if we decompose R as $R' \,\Box\, R''$ and Q as $Q'' \,\Box\, Q'$ then we do not obtain a lined up decomposition, because neither R' and Q'' nor R'' and Q' have a join.

- C_0 and $(F /\!\!/ F)$ do have a join, and C_0 does not refine $(F /\!\!/ F)$. This means that C_0 must be modified to accomodate specification $(F /\!\!/ F)$, hence enable the application of function F in the next component $(C_1 \sqcup F)$. The modification of C_0 is carried out using the refinement rules of the calculus of programming by parts.

- C_0 and $(F /\!\!/ F)$ do have a join; furthermore, C_0 refines $(F /\!\!/ F)$. This is the ideal case: it means that the modification of C does not involve component C_0, since C_0 preserves all the information required to compute function F. The verification condition provided for this case, which is

  $$C_0 \sqsupseteq (F /\!\!/ F)$$

  (along with the homologous condition, $C_2 \sqsupseteq (F \backslash\!\!\backslash F)$) consitutes the necessary condition under which the modification of C to augment it with F is localized to component C_1.

4 Application: Software Merging

We consider two versions, say V and W, of some software system, and we are interested in merging them into a single version that has all the features of V and all the features of W. We view this problem as that of refining the expression

$$V \sqcup W.$$

Because V and W stem presumably from a common original system by incrementation, it is reasonable to expect that large portions of V and W are common, and that their differences are localized. For the sake of argument, we assume that both V and W are structured as sequences, which we write as:

$$\begin{array}{ll} & V \sqcup W \\ \sqsubseteq & \qquad \{ \text{ substitutions } \} \\ & (V_0 \,\square\, V_1 \,\square\, V_2) \\ & \sqcup (W_0 \,\square\, W_1 \,\square\, W_2) \\ \sqsubseteq & \qquad \{ \text{ merging } \} \\ & ((V_0 \sqcup W_0) \,\square\, (V_1 \sqcup W_1) \,\square\, (V_2 \sqcup W_2)). \end{array}$$

We imagine, for the sake of argument, that

- The feature of V that W does not have is localized in V_1; hence $V_1 \sqsupseteq W_1$, whence $V_1 \sqcup W_1 = V_1$.
- The feature of W that V does not have is localized in W_2; hence $W_2 \sqsupseteq V_2$, whence $V_2 \sqcup W_2 = W_2$.
- The components V_0 and W_0 have not been changed from the original version of the software system, hence are identical; whence $V_0 \sqcup W_0 = V_0$.

Under the condition hypothesized above, the new version of the software system become, after substitution:

$$V_0 \,\square\, V_1 \,\square\, W_2.$$

5 Application: Software Adaptation

We consider a software component C and a specification K; we assume that C does not satisfy specification K, but we have reasons to believe that it can be modified economically to satisfy it. This problem does look like the refinement of $K \sqcup C$, since it proceeds by considering information from two specifications/programs, but differs in two significant ways:

- Whereas the refinement of $K \sqcup C$ seeks to satisfy both K and C, the modification of C to satisfy K really seeks to satisfy K only —but expects to use C towards this end.
- Whereas structure unification rules of program construction by parts consider the two arguments as interchangeable, and may impose any one argument's structure on the other, software adaptation rules will always attempt to impose the structure of C on K. In effect, what happens in software adaptation is that the refinement proceeds by drawing syntactic information from C and semantic information from K, to produce a component that acts (semantics) like K but looks (syntax) like C.

To acknowledge the asymmetry between the roles played by K and C in the adaptation of C to satisfy K, we write the expression to refine as

$$K \rhd C,$$

and we derive slightly different refinement rules for the $\rhd$ operator than we had for the $\sqcup$ operator. Space limitations prohibit us from presenting these rules, and from giving an illustrative example.

6 Conclusion

In this paper we have briefly presented a calculus of program construction by parts, and have discussed its use for recent software development paradigms, such as software incrementation, software merging, and software adaptation.

Several authors have advocated that the join operator is useful in the refinement of complex specifications [2, 4, 1]. Hoare *et al* [1] identifies some basic properties of join, like idempotence, commutativity, associativity and absorption. They also mention distributivity over usual programming constructs like sequential composition, alternation, iteration and nondeterministic choice, but for a very limited case of join (*directed set* of subspecifications with bounded nondeterminacy). Recent work of Hehner [4] includes laws of refinement for join-structured specifications. In Hehner's calculus, specifications are predicates, termination is prescribed using a time variable, and join is logical conjunction. Morgan and Gardiner [3] use a join statement for data refinement and for the definition of logical constants. They do not use join as a specification structuring device nor do they study the refinement of join-structured specifications. Finally, the work of von Wright also mentions a join operator. In [7], he reconstructs the refinement calculus of Back from elementary primitives. The emphasis of this work is more on the definition of a language than on the definition of a refinement calculus. The semantics of the language is given by predicate transformers, and miraculous specifications are allowed.

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* [1] by using partial relations and demonic operators, by not using a fictitious state to represent nontermination, 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, by not allowing miraculous specifications, and by studying the transformation of join-structured specifications. We share with the work of Sekerinski [5] (and Z) the same specification model where the focus is on input-output pairs for which a program must terminate. Most importantly, the work presented in this paper differs from other programming calculi (including our own [2]) by the fact that it deals with new software development paradigms rather than traditional *program construction*.

References

[1] C.A.R. Hoare et al. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[2] M. Frappier, J. Desharnais, and A. Mili. A calculus of program construction by parts. *Science of Computer Programming*, 6:237–254, 1996.

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

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

[5] E. Sekerinski. A calculus for predicative programming. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of program construction : second international conference*, number 669 in Lecture Notes in Computer Science, Oxford, UK, 1992. Springer-Verlag.

[6] D.R. Skuce and A. Mili. Behavioral specifications in object oriented programming. *Journal of Object Oriented Programming*, pages 41–49, January 1995.

[7] J. Von Wright. A lattice theoretical basis for program refinement. Technical report, Dept. of Computer Science, Åbo Akademi, Finland, 1990.