

CLAIRE Rules: From a Predicate Logic to an Event-Based Logic for Objects

Yves Caseau

BOUYGUES e-Lab, 1 avenue Eugène Freyssinet
78061 St-Quentin en Yvelines cedex, FRANCE

`yca@challenger.bouygues.fr`

Abstract. This paper presents an extension of the logic rule engine that is included into the CLAIRE system [CJL99] to handle events. We show how the predicate-based logic language that we use to define rule (an extension of Datalog) can be enriched with a few primitives to capture statements about transitions in the object state and external events. The main contribution of the paper is to show that this syntactical extension can be matched with an extension of the relational algebra that CLAIRE uses to compile rule into demons. Wirth very little effort it is possible to maintain a compilation strategy based on differentiation of relational terms [Cas94] that yields state-of-the-art performance.

1. Introduction

CLAIRE [CJL99] is a high-level object oriented language that incorporates features from many paradigms (functional programming, set-based programming, logic programming, etc. – the combination is very similar to LIFE [AiK93]), which was designed to express, in an easy and elegant manner, complex hybrid algorithms for combinatorial optimization. CLAIRE includes a production rule engine, based on an extension of DATALOG (with sets, like LDL [NT89] and with methods – close to [KW89]). The goal of this rule system was to be able to express constraint propagation strategies more easily. Rules are compiled into procedural demons, with a very small overhead, and we have demonstrated in the previous years that the use of rules instead of procedural code is a viable solution, even from a performance point of view.

However, the use of rules inside CLAIRE applications has been quite limited, mostly to problems where the propagation strategy is very complex, such as cumulative scheduling [CL96], or they are restricted to a marginal role, where they are used as “business rules” to tune heuristic parameters for ad-hoc algorithms (for instance, changing the selection ordering within a greedy algorithm). The main reason is that CLAIRE rules describe conditions about objects’ states, whereas many propagation strategies can be described in term of transitions, where both the “new” and the “old” state of an object must be accessed.

In a more general context, the internal logic of applications is more and more based on event models, with tools such as workflow or message buses. To use rules to describe the behavior of such complex systems, one must be able to reason about events and state transition.

We have built a new extended object logic that captures events and state transition and used it to extend the rule language for CLAIRE. We were equally able to extend the compilation scheme that is based on the algebraic representation of rules. We have introduced two new operators in our relational algebra [Cas94] and found the associated differentiation and code generation formulas. The result is a new release of ClaireRules (v3.0) that is significantly more expressive and powerful, while retaining the performance advantages of the previous system.

The paper is organized as follows. Section 2 describes the new object-oriented logic that we used to define rules. Section 3 presents the extension to the relational algebra that is necessary to capture the increase in expressive power. We show how to extend the symbolic calculus that is used to process rules. The last section is dedicated to the code generation aspects.

2. Claire Rules

2.1 Overview

A rule in CLAIRE is obtained by associating a logical condition to an expression. Each time the condition becomes true for some objects, the expression will be evaluated for these objects (represented by the rule's free variables). A condition is defined with a logical language, which includes predicate logic to describe the state of one or many objects and event logic to capture the occurrence of a precise event.

To define a logical condition, we use the logic language defined by the following grammar.

```
<assertion>    <expression> <comp> <expression> | <variable> :: <class> |
                <value> := <expression> | (<expression> <- <variable>) > |
                exists(<variable>, <assertion>) | not(<assertion>) |
                if (<expression> <comp> <expression>) <assertion> else <assertion> |
                <assertion> & <assertion> | <assertion> | <assertion>

<expression>   <variable> | <entity> | <function>(<expression>) |
                <expression> <operation> <expression> |
                {<var> in <expression> | <assertion>} |
                list{<var> in <expression> | <assertion>}

<value>        <table>[<expression>] | <expression>.<property>
```

The value of a logical expression is a CLAIRE entity. Logical assertions are made by combining expressions. The most common type of assertion is obtained by comparing two expressions with a comparison operation. A comparison operation is an operation that returns a Boolean value. Here are some logical assertions:

```
Paul.age = 10,
size({x in person | Paul % x.friends}) < 2,
y % integer
```

The novelty in CLAIRE 3.0 is the introduction of event logic. There are two events that can be matched precisely: the update of a slot or a table, and the instantiation of a class. The expression $x.r := y$ is an event expression that says both that $x.r = y$ and that the last event is actually the update of $x.r$ from a previous value. Thus, the introduction of event expressions serves two purposes:

- (a) control the triggering of a rule more precisely by telling which event should be considered. Consider the following example:

```
x.age = y & z.age = 2 * y
```

This logical expression may become true (and trigger an associated rule) from an update of the age slot in two manners, either because $x.age$ has changed or because $z.age$ has changed. In most cases, this behavior is precisely what is expected. In contrast, the following expression :

```
x.age := y & z.age = 2 * y
```

adds to the previous expression the explicit constraint that the triggering update is applied to the object x .

- (b) use the “previous value” of the updated expression $x.r$ in the logic expression. This is achieved with the expression $x.r := (y <- z)$ which says that the last event is the update of $x.r$ from z to y . For instance, here is the event expression that states that $x.salary$ crossed the 100000 limit:

```
x.salary := (z <- y) & y < 100000 & z >= 100000
```

The other event that can be described with the event logic is the instantiation of a class. The expression $x :: C$ means that x is a new instance of the class C . As previously, the event expression is the combination of the statement $x \% C$ and the additional constraint that the last event is precisely the creation of the new instance x of the class C . We use object instantiation to represent external events when writing an application that must be integrated with other components using a publish & subscribe event model, such as the Java Bean model (cf. Section 5.3).

Free variables within logical expressions are assumed to be quantified universally, as we shall see in the next section. A rule conditions(x,y,z,\dots) \Rightarrow actions(x,y,z,\dots) means that the actions should be executed for all values x,y,z,\dots once the conditions become true. In addition, existential quantification (there exists a ... such that ...) is introduced with the exists construction. For instance, here is how we say that there exists a common friend (z) to two persons x and y :

```
exists(z, x % z.friend & y % z.friend )
```

Existential variables can be typed, as in

```
exists(z:woman, x % z.friend & y % z.friend)
```

Variables that are not introduced in the logical condition by an "exist" or an "{.. in .. | ...}" are called free variables, they can be used within their scope to form expressions. The semantic of `exists(z, C(..))` is as expected, it is true if there exists one `z` such that the condition `C` is true. Notice that the scope of the variable is only the block between parentheses in the `exists(...)`. Therefore, this variable cannot be used in the action part of the rule.

Last, assertions can be formed as a conditional expression using `if`. The test in a logic conditional is necessarily of the form (expression comp expression) and it must have two branches. Here is an example that one could use to compute taxes:

```
if (r < 10000) x = 0
else if (r < 20000) x = r * 0.1
else x = r * 0.2 - 2000
```

The value of a logical assertion is always a Boolean, thus logical assertions can be combined with `&` (and) and `|` (or). In addition to pre-defined operations such as `<=` or `+`, it is possible to use new properties inside logical assertions but they need to be described using the description table and a set of keyword including comparison, binary_operation, monoid, group_operation. By declaring the algebraic characterization of property we are telling the rewriting system that produces the algebraic representation how to handle the logic expression.

To define a rule, we define a list of free - universally quantified - variables, that are introduced as parameters of the rule, a condition, which is given as an assertion using the previously defined variables and a conclusion that is preceded by `=>`. Here is a classical transitive closure example:

```
r1(x:person, y:person) :: rule(
exists(z, x % z.friends & z % y.friends )
=> y.friend :add x)
```

Rules are named (for easier debugging) and can use any CLAIRE expression as a conclusion, using the set of free variables, such as in:

```
r2(x:person, y:person, z:person) :: rule(
exists(z, x.age + y.age = z.age ) => printf("~S ~S ~S\n",x,y,z))
```

2.2 Event-based Rules

We may now focus on understanding how a rule works. Rules are checked (efficiently as we shall later see) each time that an event occurs. Events come in two kinds. The first kind are updates to slots or tables that are declared as event generating with the event statement.

```
<events definition> <event | noevent>( <<table> | <property>>seq )
```

The second kind of events are instantiations of classes. These include the instantiation of "event objects" classes that represent "external events" used in a larger-scale application, but also the instantiation of local classes. Note that the "instantiation event" occurs once the initialization of the object (with default or given values for the slots).

A rule considers as events all slots or tables that have been declared as such before the rule has been declared, so the event declaration needs to precede the rule declaration, in order to create the "demons" that will watch over the given relation and fire the rule when needed. The declaration `noevent(...)` may be used to prevent explicitly a rule to react to some relations that were declared previously as events for other rules. By spreading the event declarations (and occasionally `noevent` declarations) before and between the rules, one has a precise control over the triggering of the rules. This level of control is further refined with the use of event expressions within the condition of the rule.

Updates that are considered as events are:

- `x.r := y`, where `r` is a slot of `x` and `event(r)` has been declared.
- `a[x] := y`, where `a` is a table and `event(a)` has been declared.
- `x.r :add y`, where `r` is a multi-valued slot of `x` (with range bag) and `event(r)` has been declared.
- `a[x] :add y`, where `a` is a multi-valued table and `event(a)` has been declared.
- The instantiation of a class `C`, whether static (`x :: C(...)`) or dynamic (`new(C,..)`).

The handling of mono- vs. multi- valuation is an important aspect of this logic. CLAIRE makes a difference between a multi-valued slot (for instance, a slot `friends` with range `set<person>`) and a mono-valued slot with range set (we can use the same example). The management of inverses is different: in the first case, the inverse of `friends` is a relation from person to person; in the second case, it is a relation from `set<person>` to

person. Similarly, the atomic events are different: in the first case it is the addition of a new friend; in the second case, it is the replacement of a set of friend by another set. It is our experience that both modeling options are needed to capture a large set of situations.

2.3 Examples

Here are three (simplified) examples taken from the field of constraint propagation. The first one tells that if two tasks are consecutive (attached means without idle time in between), we can propagate the increase in minStart (earliest starting time) directly. The second tells how to propagate a duration increase.

```
R1(x:task, y:task, z:integer, u:integer) :: rule(
  x.attached = y & x.minstart := (z <- u) => y.minstart :+ (z - u) )

R2(x:task, I:taskInterval, z:integer, u:integer) :: rule(
  x.minStart >= I.minstart & x.maxEnd <= I.maxEnd & x.duration := (z <- u)
=> I.duration :+ (z - u) )

R3(x:task, I:taskInterval, z:integer, u:integer) :: rule(
  x.minStart := (z <- u) & z >= I.minstart & u < I.minStart &
x.maxEnd <= I.maxEnd
=> I.duration :+ x.duration )
```

The first two rules are similar and illustrate the use of the event pattern to capture a transition and propagate an incremental change. The second rule also shows that the event pattern forces the rule to trigger when a change is made on the duration attribute, which is what we intend here, since a different rule (R3) handles the case when a change is made on the minStart attribute. In R3, the event pattern is used to capture the state transition (x now belongs to the task interval and did not before).

The next two examples illustrate the use of event-rules to model state-transition systems. They are taken from an “artificial life” scenario where units move from cell to cell. The first rule says that if a unit moves to a position where another unit is already installed, a conflict occurs which will result into a failure of the incoming unit if its strength is less than the strength of the residing unit. The second re-computes the visibility graph for a unit that gets “out-of-the-woods”.

```
RS1(x:unit, y:unit) :: rule(
  x.location := y.location & x.strength <= y.strength => failure(x))

RS2(x:unit, z:position, u:position) :: rule(
  x.location := (z <- u) & u.forest? = true & z.forest? = false =>
computeVisibility(x) )
```

The use of event logic enables us to represent the dissymmetry between x and y in the first rule, which would not be possible with sole predicate logic. The second rule is another example of using the event pattern to capture a state transition (from being in-the-woods to out-of-the-woods).

3. Algebraic Representation of Rules

3.1 Principles of Algebraic Rules

CLAIRE rules are compiled into procedural demons in three phases. First, the condition part of the rule is rewritten into an algebraic expression, which is next differentiated into many functions that are finally transformed into if-write demons. The process of rule compilation relies on the use of a relational algebra. This algebra $A(R)$ contains relational terms (elements of R) that represent binary relations; it is generated from a set of constants (Cartesian products of types), of variables (CLAIRE slots and tables) and a set of operators (similar to [McL81]). These operators can define unions, intersections and compositions over relational terms, but also more advanced algebraic functions such as the composition of a binary operation or a comparison operation with two binary relations, etc. (see [Cas94] for details, a summary of the regular algebra will be added in the full-size paper).

We will briefly describe each step of the compilation process, and follow the transformation of a rule, from its logical expression towards the generated procedural demons. Here, we use the closure rule, which performs a transitive closure over a graph, described through the binary relations `edge` and `path` (& and | are usual Boolean conjunction and disjunction):

```
closure(x:point, y:point) :: rule(
  edge(x,y) | exists(z, edge(x,z) & path(z,y))
=> x.path :add y)
```

- The first phase of the compilation process is the transformation of the conditional part of the rule into a relational formula that belongs to $A(R)$. Translation into the algebraic form is based on rewriting and involves a lot of knowledge about object methods. The principle is to solve the equation `assertion(x,y)`, while considering that `x` is known and that `y` is sought. The result is the relational algorithm that explains how to get `y` from `x`, which is represented as a term in $A(R)$. The conditional part of the rule `closure` is transformed into the following algebraic term `t1`:

$$t1 = edge \quad (path \quad edge)$$

where `and` and `and` are respectively the union operator and the composition.

- Next, a phase of symbolic differentiation [SZ88] computes the effect of adding a pair of values into a table or a slot. The differentiation is a function that is applied to the algebraic term `t`, with respect to each relation R occurring in the condition, and that produces a term `t / R` in $A(R \{0,1\})$. `0` and `1` are two constant functions that represent respectively the empty function (`0 : (x,y) {}`) and the identity function (`1 : (x,y) {(x,y)}`). A set of differentiation rules (one for each operation on binary relations) is used to compute the term `t / R`. The term `t1` computed above is differentiated with respect to the relation `edge` and yields the following function `f1`:

$$f1 = t1 / edge = 1 \quad (path \quad 1)$$

or in other terms:

$$f1 : (a,b) \quad \{ (a,b) \} \quad \{(a,y) \mid (b,y) \quad [path]\}$$

- The last phase of the compilation of a rule consists of associating the set of functions (derived from each relational term occurring in the condition) and the CLAIRE expression in the conclusion, so as to build one if-write demon per function. The demon is used to propagate each update so that rules are triggered in an incremental manner, yielding an approach similar to [SKG87].

3.2 A Relational Algebra

The relational algebra contains terms that represent binary relations. It is generated from a set of constants (cartesian products of types), a set of variables (the relations that are described with rules) and a set of relational operators. This set may be described as follows :

- `+`, `*`, `o` are the usual operators for union, intersection and composition (binary join).
- `[+](r1, r2)` represents the composition of a binary operation (`+`) with two binary relations r_1 and r_2 .
- `[<](r1, r2)` is the composition of a comparison operation (`<`) with two binary relations r_1 and r_2 .
- `if[<](r1, r2)](r3, r4)` is the conditional composition of a test (`[<](r1, r2)`) with two binary relations r_3 and r_4 which represent the two branches of the conditional (r_3 if the test succeeds and r_4 otherwise).
- `{z:r1, r2}` represents a relation that associates to an object the set of objects `z` bound through r_1 which also "satisfy" r_2 (hence, r_2 is a filtering relation - a `+` or a composition of `'s`). This operator is introduced to handle set creation in logical formulae.
- `Set_expansion(r)`, written r^* , is the reciprocal operator which is used to treat a relation r with range set as a multi-valued relation.
-

Here is the formal definition of the relational operators:

$$\begin{aligned}
& (x,y) \quad r_1 \quad r_2 \quad (x,y) \quad r_1 \quad (x,y) \quad r_2 \\
& (x,y) \quad r_1 \quad r_2 \quad (x,y) \quad r_1 \quad (x,y) \quad r_2 \\
& (x,y) \quad r_1 \circ r_2 \quad z, (x,z) \quad r_2 \quad (z,y) \quad r_1 \\
& (x,y) \quad [+](r_1, r_2) \quad z_1, z_2, (x,z_1) \quad r_1, (x,z_2) \quad r_2 \quad y = z_1 + z_2 \\
& (x,y) \quad [<](r_1, r_2) \quad x = y \quad z_1, z_2, (x,z_1) \quad r_1, (x,z_2) \quad r_2 \quad z_1 < z_2 \\
& (x,y) \quad \text{if} [<](r_1, r_2)(r_3, r_4) \\
& \quad \quad \quad \{ (x,y) \quad r_3 \quad z_1, z_2, (x,z_1) \quad r_1, (x,z_2) \quad r_2 \quad z_1 < z_2 \} \\
& \quad \quad \quad \{ (x,y) \quad r_4 \quad z_1, z_2, (x,z_1) \quad r_1, (x,z_2) \quad r_2 \quad \neg(z_1 < z_2) \} \\
& (x,y) \quad \{ z : r_1, r_2 \} \quad y = \{ z \mid (x,z) \quad r_1 \quad (z,z) \quad r_2 \} \\
& (x,y) \quad r^* \quad z, (x,z) \quad r \quad y \quad z
\end{aligned}$$

To achieve the same expressive power as the logical fragment that we use to define conditions on objects, we need to introduce the ability to share a sub-term, which will correspond to the introduction of existential variables. Let us now define $A(R)$ the term algebra generated by the previous set of operators, the constant cartesian products of types (such as $\text{person} \times \{0,1,2\}$) and the set R of relational variables (that represent relations from CLAIRE, which are tables or properties). We extend our definition as follows:

- If x is a relation variable that does not belong to R , if $t_2(R_1, \dots, R_n, x)$ is a term from $A(R \setminus \{x\})$ and $t_1(R_1, \dots, R_m)$ is a term from $A(R)$, then $(x:t_1).t_2$ is a term from $A(R)$ which represent the relation denoted by t_2 in an new “object system” where x is bound to the relation denoted by t_1 .

Notice that an “object system” is simply defined by the valuation function q that associates to each relation name in R the actual binary relation in the current system. We write $[t]_q$ the binary relation denoted by the term t for a given valuation q . A key result in [Cas94] is that the logic fragment that we use and the relational algebra $A(R)$ have the same expressive power. This means that for any condition C with at most two free variables x and y there exists a term t of the algebra such that $C(x,y) \iff (x,y) \in [t]_q$. The term is not necessarily unique and the difficulty is to produce an “optimized” term, which represents the straightest relational computation that is necessary to produce the set of y starting from the value of x .

The transformation of a logic rule into its algebraic representation is based on rewriting, as explained in [Cas94]. This is achieved with rewriting rules that incorporate the algebraic knowledge about objects’ methods such as:

$$\begin{aligned}
& \text{group}(+,0,-) \ \& \ x + y = z \Rightarrow x = z - y \\
& \text{monoid}(*,|,/) \ \& \ x * y = z \Rightarrow (y \mid z) \ \& \ (x = z / y)
\end{aligned}$$

In the previous years, we have added the concept of simplification rules, which implement optimizing strategies (replacing an algebraic term by another one with the same denotation but which is more efficient to compute) in a manner similar to the optimization of SQL queries. The rules are applied recursively while algebraic terms are being built. They are mostly based on three ideas:

- (1) filtering terms should be ordered from simpler to more complex, and applied as soon as possible from an evaluation order point of view,
- (2) constants sub-terms must be recognized, simplified, and re-organized. For instance, linear sub-term factorization $(2 * x + 3 * x \Rightarrow 5 * x)$ is applied through this simplification step.
- (3) inversible sub-terms must be detected because they can be used (in their inverse form) to avoid costly joins.

The motivation for this translation is that we are able to represent propagation as a formal computation over algebraic terms, called differentiation. Suppose that t is a term for $A(R)$, R_i a relation variable from R and q a valuation function representing the current object system. We now consider an atomic update, that is the addition of a pair (x,y) to the relation $q(R_i)$. The term t now represents a new relation in the new object system q' . The goal of differentiation is to compute $[t]_{q'} - [t]_q$, that is the set of object pairs that “appeared” in the relation represented by t “because” of the update (since we use a monotonic algebra, we have $[t]_{q'} \supseteq [t]_q$).

Let us define the functional algebra $F(R)$ as $A(R \setminus \{\emptyset, \uparrow\})$, where each relation of R is mapped to a constant function. Differentiation is a formal operation on $A(R)$ which associates to a pair $(t, R_i) \in A(R) \times R$ a term $\partial t / \partial R_i$ from another functional algebra $F(R)$ (which represents a function that associates a binary relation to an object pair) such that :

$$[t]_{q'} - [t]_q = [\partial t / \partial R_i(a,b)]_{q'} \cup [t]_q \quad (1)$$

To guarantee the interest of differentiation we also impose that $\partial t / \partial R_i(a,b)$ is the smallest term from $F(R)$ which satisfies (1) according to the partial order induced by inclusion. However, (1) is a necessary and sufficient condition to prove the validity of using differentiation to model propagation. We use a canonical functional algebra $F(R)$ generated by $A(R)$ and two constants \emptyset, \uparrow that represent respectively the “empty function”

$((x,y) \rightarrow z)$ and the unit function $((x,y) \rightarrow \{(x,y)\})$. To represent the propagation of an update in a set-valued expression, we also need to extend $F(R)$ with a new projection operation $\pi(t)$ defined as follows.

$$(x,y) \rightarrow [\pi(t)(a,b)]_q \quad x = y \quad z, (x,z) \rightarrow [t(a,b)]_q$$

Thus, $\pi(t)$ is the first projection of the relation represented by t .

3.3 Extending the Algebra to Handle Event-based Logic

The algebra is extended with two operators:

- (R) is an “event-filter” : it represents the same relation as R , but it will behave differently when the term is differentiated.
- R' represents the “previous” version of the relation R before an atomic update. It cannot be used freely in the relational algebra, which would require a much more complex implementation, but only in the following pattern:

$$(z:R').t\{ (R) \}$$

where $t\{ (R) \}$ is a term which contains (R) as a sub-term.

The differentiation rules are straightforward:

$$\begin{aligned} (R) / R &= \mathbb{1} \quad , \quad (R') / R = \mathbb{0} \\ ((z:R').t\{ (R) \}) / R &= (z:R'). \quad t / R \end{aligned}$$

The compilation rules (cf. [Cas94]) are two methods, expR and expF , that represent the semantic of the operators through code generation. Both are defined by structural induction for each operator.

- $\text{expR}(t,x,y,e(y))$ is a code fragment that applies the expression $e(y)$ to each object y that is linked to x according to the binary relation represented by the term t . Here is a simple example for a binary composition:

$$\text{expR}(t_1 \circ t_2, x, y, e(y)) = \text{expR}(t_2, x, z, \text{expR}(t_1, z, y, e(y)))$$

- $\text{expF}(t/R, x, y, u, v, e(u, v))$ is a code fragment that applies the expression $e(u, v)$ for each pair (u, v) that belongs to the relation which is the value of $[t/R](a, b)$, where we recall that t/R is a term of a functional algebra that represent a function that associates a binary relation to each pair of objects. Here is another example for compiling the composition of a derivative and a relation:

$$\text{expF}(t/R \circ t, x, y, u, v, e(u, v)) = \text{expR}(t/R, x, y, a, v, \text{expR}((t)^{-1}, a, u, e(u, v)))$$

This last example shows that although differentiation rules are simple and intuitive, their actual implementation through code generation may be subtler. The ability to inverse any term from the algebra is not easy to reach, although many terms may be inverted through recursive formulas. For the few but annoying exceptions, we actually revert to the logical expression and re-apply a translation step (rewriting) while changing the variable ordering (each translation step takes one free variable as the “base” and the other one as the target). To achieve this, we need to ensure that for each composition sub-term $(r_1 \circ r_2)$, the second relation r_2 is always applied to this “base” variable. We use the associativity of binary join to generate terms like $(r_1 \circ (r_2 \circ r_3))$ as opposed to $((r_1 \circ r_2) \circ r_3)$.

Similarly, the rule for differentiating a “lambda-term” $((z:R).t)$ (i.e., the binding of a sub-term t to a variable z) looks simple:

$$\partial((z:t_1).t_2) / \partial R = (z:\partial t_1 / \partial R).t_2 \quad (z:t_1). \partial t_2 / \partial R$$

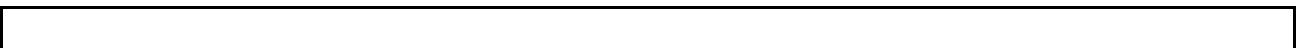
However, the actual evaluation of the second part $(z:t_1). \partial t_2 / \partial R$ is subtle, since we need to know the value of this base variable before being able to apply the sub-term t_1 to find z (and compute the whole expression). This requires a recursive traversal of the derived term to find out exactly when the base variable can be computed.

These compilation methods, expR and expF , are extended to the new operators as follows:

$$\begin{aligned} \text{expR}((R), x, y, e(y)) &= \text{nil} \\ \text{expF}((z:[R]).(t/R), x, y, u, v, e(u, v)) &= \text{let } z = \text{LAST in } \text{expF}(t/R, x, y, u, v, e(u, v)) \end{aligned}$$

The first formula tells that (R) will be ignored in a rule if it is not differentiated precisely according to R . Thus it acts as a filter. The second formula shows that we make a direct reference to the previous value LAST of the attribute that is being updated. This value is stored when we perform the update as we shall later see.

Figure 2 is a table containing the set of derivation rules for this new logic. These rules completely specify differentiation as a formal computation. The analogy with rules for numerical differential calculus is no coincidence and can be proven using a matrix model for binary relations [Cas94].



(i,j), $\partial R_j / \partial R_i = \mathbb{1}$ if $i = j$ and $\mathbb{0}$ otherwise.

(i,j), $\partial R_j^{-1} / \partial R_i = \mathbb{1}^{-1}$ if $i = j$ and $\mathbb{0}$ otherwise.

p P, $\partial p / \partial R_i = \mathbb{0}$.

If S_1 and S_2 are two types, $\partial(S_1 \times S_2) / \partial R = \mathbb{0}$

t_1, t_2 A(R), $\partial(t_1 \circ t_2) / \partial R_i = (\partial t_1 / \partial R_i \circ t_2) \quad (t_1 \circ \partial t_2 / \partial R_i)$

t_1, t_2 A(R), $\partial(t_1 \quad t_2) / \partial R_i = (\partial t_1 / \partial R_i \quad t_2) \quad (t_1 \quad \partial t_2 / \partial R_i)$

t_1, t_2 A(R), $\partial(t_1 \quad t_2) / \partial R_i = (\partial t_1 / \partial R_i \quad \partial t_2 / \partial R_i)$

t_1, t_2 A(R), $f \quad F \circ, \partial [f](t_1, t_2) / \partial R_i = [f](\partial t_1 / \partial R_i, t_2) \quad [f](t_1, \partial t_2 / \partial R_i)$

t_1, t_2 A(R), $F \circ, \partial [\quad](t_1, t_2) / \partial R_i = [\quad](\partial t_1 / \partial R_i, t_2) \quad [\quad](t_1, \partial t_2 / \partial R_i)$

t_1, t_2 A(R), $z \quad V, \partial ((z:t_1).t_2) / \partial R_i = (z:\partial t_1 / \partial R_i).t_2 \quad (z:t_1).\partial t_2 / \partial R_i$

t_1, t_2, t A(R) | $t = [\quad](\dots)$,

$\partial \text{if}[\quad](t_1, t_2) = \text{if}[\partial t_1 / \partial R_i][\quad](t_1, t_2) \quad (\partial t_1 / \partial R_i \circ t) \quad (\partial t_2 / \partial R_i \circ \neg t)$

t A(R), $\partial\{z:t_1, t_2\} / \partial R_i = \{z:t_1, t_2\} \circ \pi(\partial t_1 / \partial R_i \quad \partial t_2 / \partial R_i)$

j , $\partial (R_j^*) / \partial R_i = (R_j^*) \circ \pi(\partial R_j / \partial R_i)$

r R, $\partial \neg r / \partial R = \mathbb{0}$.

(i,j), $\partial (R_j) / \partial R_i = \partial R_j / \partial R_i$

t A(R), $((z.[R]) t) / R = (z.[R]) (t / R)$

Figure 2: Differentiation rules.

3.4 Examples

For each rule given in Section 4.1.3, we give the equivalent algebraic term t and its derivation $\partial t / \partial R$ according to the appropriate relation. Since these rules use an event pattern to capture one precise pattern, there is only one relation R such that the derivative $\partial t / \partial R$ is not empty.

R1 [t1]: $(u:\text{minstart}'). (z: (\text{minstart}).\text{attached})$

R2 [t2]: $(I:\text{task} \times \text{taskInterval}). (z:\text{duration}). ((\text{duration}) \circ [\Rightarrow](\text{minStart}, \text{minStart} \circ I) \circ [\Leftarrow](\text{maxEnd}, \text{maxEnd} \circ I))$

R3 [t3]: $(I:\text{task} \times \text{taskInterval}). ([\Leftarrow](\text{maxEnd}, \text{maxEnd} \circ I) \circ (u:\text{minstart}') . ([\Leftarrow](u, \text{minStart} \circ I) \circ [\Rightarrow](\text{minstart}, \text{minStart} \circ I)))$

RS1 [t4]: $(y:(\text{location}^{-1} \circ (\text{location}))). [\Rightarrow](\text{strength} \circ y, \text{strength})$

RS2 [t5]: $(u:\text{location}'). ([\Rightarrow](\text{forest?} \circ u, \text{false}) \circ [\Rightarrow](\text{forest?} \circ (\text{location}), \text{true}))$

Here are some of the derived terms:

t1/ $\text{minStart} = (u:\text{minstart}'). ((z: \mathcal{I}) \text{attached}^{-1})$

t2/ $\text{minStart} = (I:\text{task} \times \text{taskInterval}). (z:\text{duration}'). (\mathcal{I} \circ [\Rightarrow](\text{minStart}, \text{minStart} \circ I) \circ [\Leftarrow](\text{maxEnd}, \text{maxEnd} \circ I))$

t4/ $\text{location} = (y:(\text{location}^{-1} \circ \mathcal{I})). [\Rightarrow](\text{strength} \circ y, \text{strength})$

t5/ $\text{location} = (u:\text{location}'). ([\Rightarrow](\text{forest?} \circ u, \text{false}) \circ [\Rightarrow](\text{forest?} \circ \mathcal{I}, \text{true}))$

Each derived term represents a function, which meaning is obtained from the semantic of the functional algebra. For instance, the first term is a function that takes a pair of objects (a,b), binds z to the value b (from the update $a.\text{minstart} := b$), binds u to the previous value of $a.\text{minstart}$, and computes the set of objects that are attached to a . The use of this function will become more evident in the next section, where we generate a demon that applied the conclusion of the rule to all pairs (a,y) produced by this derived term.

4. Code Generation

4.1 Generating Demons from Rules

When the compiler is applied to a set of rules, all rules are translated into an algebraic representation and all derivatives are computed. An algebraic rule is represented as $(t \Rightarrow e)$, where t is the term from the relational algebra that is equivalent to the logical condition of the rule and e is the “conclusion” of the rule (the action part). A code fragment is then produced for each rule $(t \Rightarrow e)$ and each derivative t / R by applying the `exp2` method as follows:

```
expF( T / R,x,y,u,v,e)
```

The result is a fragment of CLAIRE code that applies the conclusion of the rule to all pairs (u,v) of objects that satisfies the condition of the rule because of the event “ $x.R := y$ ”.

These fragments are combined, for each relation R , into a if-write demon that will be used to change the value of $x.R$, or $R[x]$ if R is a table (in CLAIRE, tables and slots are abstracted into a binary relations and can be used indifferently in the logic language). The demon generated for R is based on the following pattern:

```
if_write[R](x, y)
-> let LAST := get(R,x) in      // the previous value is stored
  (if (LAST != y)             // checks that this is indeed an update
    (x.R := y,                 // performs the update
     ...                       // propagation fragment from 1st rule
    if (x.R = y) ...          // propagation fragment from 2nd rule
    if (x.R = y) ...          // propagation fragment from 3rd rule
    ...))
```

Note that the propagation fragment obtained from the rules other than the first one are guarded with a test $(x.R = y)$ to make sure that the rules are only applied if the update is still valid.

CLAIRE makes a difference between mono- and multi-valued relations. For multi-valued relations, the event model is simpler since the only event that is detected is the addition of a new member in one object’s set of values. Thus the generation of the demon is slightly simpler in the multi-valued case.

For instance, the last phase of the compilation of the `closure` rule, which we gave as an example in the previous section, consists of associating the set of functions (derived from each relational term occurring in the condition) and the CLAIRE expression in the conclusion, so as to build one if-write demon per function. For instance, the demon associated to the rule `closure` and the relation `edge` is compiled as follows (it is evaluated each time a pair (x,y) is added to the `edge` relation):

```
if_write[edge](x, y)
-> if (x.edge add? y)          // tests if y is not already here
  (x.path :add y,
   for z in y.path x.path :add z)
```

The handling of instantiation is straightforward since each instance is attached to its class through an inversible relationship (`isa/instances`). Therefore the representation of the event

```
x :: C
```

is made with the term `(minstart)`. On the other hand, generating a large collection of if-write demons for this “`isa`” relation would slow instantiation down for every class; thus, we associated directly the if-write demon with the class C (this is why C must be a constant in the expression $x :: C$).

4.2 Code Generation Techniques

The code generation phase is mostly the application of the `expR` and `expF` methods, which embodies the application of various optimisation schemes, such as the generation of simpler and faster code if a sub-term represents a mono-valued relation. Moreover, there are two other techniques that are used to produce procedural code, which is as close as possible to what the user would write herself to propagate the rules. The first technique is the use of precise type inference and the second one is the use of modes to distinguish between different semantics for applying the rules’ conclusions.

Type Inference within the rule compiler is defined by associating a second-order type to each subterm, either from the relational algebra or the derived functional algebra. These second-order types are functions that infer an output type from any input type or reciprocally. For instance, if y is bound to x through the relation represented by t , this function will return the type of y depending on the type of x . The case of a derivative term is slightly more complex since there are two input type parameters, namely the type of x and y where the update is

“ $x.R := y$ ”. These functions are obtained in a straightforward manner through the abstract interpretation of the relational operators on the CLAIRE type domain, which is an inclusion lattice [CC77]. This precise type

inference is key to producing efficient code, and it is more powerful than the type inference performed by CLAIRE on the fragments generated by the rule compiler.

CLAIRE supports some additional tuning of its rules through the mode declaration. These modes are defined as follows, and enable the user to remedy some of the problems that occur with non-monotonic and non-commutative rules.

```
<modes> mode( default | once | set | <integer> )
```

Rules should not, in general, be written in such a way that the result is order-dependent. The order in which they are triggered depends on the events (the propagation pattern) and the order in which the rules were entered. If it becomes necessary to have a more precise control over this order, priorities may be used. A priority is an integer attached to the rule using the mode declaration. CLAIRE will ensure that rules with higher priority will be triggered first for each new event. Notice that there is no implicit stack structure for triggering rules: the events generated by the application of a first rule may cause a new rule to be evaluated before a second rule is applied to the original event.

The conclusion is applied to any pair of objects that is obtained through a logical derivation of the conclusion and the update. This assumes that the conclusion can be fired more than once when the logical expression is redundant (multiple derivation paths for the same pair). However, it may be wrong to apply the conclusion twice even if a pair is obtained from two different paths. Consider the following example:

```
strange1(x:Person) :: rule( x.age = 18 | x.age > 10 => give(x, $1000) )
```

An update "John.age := 18" may cause the rule to be fired twice. The solution in CLAIRE is to use the mode(set) declaration before the rule, which will force the computation of the set of pairs before firing the conclusion (thus eliminating duplicates). Note that the use of an event pattern is precisely another way to get rid of this problem.

Similarly, rules should have a monotonous behavior, which means that their conclusion should not invalidate the condition. CLAIRE supports non-monotonous rules but the difficulties linked to the absence of a clear semantic remain. Let us consider a second example:

```
strange2(x:Person, y:Person) :: rule(
  x.age = 18 & y % x.friends => (x.age := 19, invite(x,y)) )
```

Should the rule invite one friend (which one) or all friends when the age of John is set to 18? The default behavior as well as the "set" behavior will invite all friends. The mode(once) declaration will ensure that the conclusion is fired only once for each update event. Using non-monotonic rules is tricky, but it is sometimes very useful.

These modes are implemented as an additional layer in the generation of the code fragments for the demons. The priorities are used to order the code fragments. The "set" mode builds a temporary set of pairs to be updated before applying the conclusion. The "once" mode uses an exception handler to make sure that the conclusion is only fired once.

4.3 Application

Here are two examples of demons generated by CLAIRE from the rules that were proposed as examples. The first one is associated to the minStart slot and represents the two first rules. It is obtained by applying the code generation method expF to the two derived terms that we have shown in Section 3.3.

```
if_write[minStart](t, y)
-> let LAST := t.minStart in           // the previous value is stored
  (if (LAST != y)                     // checks that this is indeed an update
   (t.minStart := y,                  // performs the update

    // propagation fragment from 1st rule (R1)
    when Y := get(attached,t) in Y.minStart := (y - LAST),

    // propagation fragment from 2nd rule (R3)
    if (t.minStart = y)
    for I in taskInterval
      if (y >= I.minStart & LAST < I.minStart & t.maxEnd <= I.maxEnd)
        I.duration := x.duration )
```

The second example is obtained from rules RS1 and RS2 for the location attribute:

```
if_write[location](x, y)
-> let LAST := x.location in           // the previous value is stored
  (if (LAST != y)                     // checks that this is indeed an update
   (x.location := y,                  // performs the update
```

```

// propagation fragment from 1st rule (RS1)
for Y in of~(location,y) // reads the inverse of location
if (Y.strength >= x.strength) failure(x)
// propagation fragment from 2nd rule (RS2)
if (x.location = y)
if (y.forest? & not(LAST.forest?) computeVisibility(x) )

```

These two examples illustrate our claim that the rule compiler is able to produce procedural code for propagation that is very close to what a user would write.

4.4 Performance Results

We have not tested the new features of ClaireRules from a performance point of view due to the absence of established benchmarks. On more classical problems, such as those reported in Table 3, the code generation strategy of CLAIRE is very efficient (between 3 to 50 Mips = Million inferences per second), even when solving complex problems. The CPU time is measured on a Pentium III laptop at 500Mhz, using MSVC++ 6.0. Since CLAIRE can also generate Java code, we also include the results using Symantec's Visual Café 3.0. CLAIRE is between 2 or 3 orders of magnitude faster than what we have measured with RETE-based inference engines [For82], both in the C++ or Java category. More empirically, we have noticed that CLAIRE programs written with production rules yield only about a 10 to 50 % penalty compared with hand-optimized procedural code.

Note that the two benchmarks that yield lower numbers of inferences per second are hybrid programs that also use methods and backtracking, thus the CPU time does not reflect the rule propagation activity.

	CPU time Java version	CPU time C++ version	Rules/s C++ version
Simple Filtering	0.01s	0.02s	50 Mips
Planning (Monkey & Banana)	0.03s	0.015s	13 Mips
Constraint Propagation (triplet)	0.36s	0.05s	3 Mips
Rules & methods (Airline)	0.05s	0.01s	100Kips
Problem solving (Zebra)	0.043s	0.011	2.7 Mips
Problem solving (Dinner)	0.18s	0.08s	250Kips

Table 3: Benchmarks on rules.

The demons are produced using CLAIRE, taking advantage of CLAIRE high-level of abstraction, which simplifies the handling and the iteration of set expressions, as well as the reflective nature of the CLAIRE system, which makes generating new programs much easier. The CLAIRE program is then compiled into C++ or Java. Our next step is to generate directly Java beans that exploit the event/listener model. Because we can generate Java code with no overhead or central control algorithm, this technique is ideally suited for compiling a set of rules into a Java bean component [Eng97]. This approach can be applied to all other publish & subscribe message architecture, which makes this technique very interesting for Enterprise Application Integration (EAI). For instance, we plan to use this approach to implement distributed workflow engines where a set of process rules is compiled into a group of distributed agents.

5. Conclusion

We have shown how to extend the object-oriented logic that was used for CLAIRE to capture the handling of events. Although the extension is quite simple, the expressive power of the logic is much improved and the rules based on this logic can be used to describe almost any type of constraint propagation strategies, which was not the case with a simpler logic based on predicates. The main technical result of this paper is the fact that the algebraic approach towards rules compiling can also be extended easily, yielding a rule system that is both powerful and efficient.

The task of capturing reasoning about events and objects with rules is a complex one. The approach presented in this paper is only a first step. A more ambitious project is a language called open ROCE (Rules, Objects, Control and Events), which is more concerned with expressive power than compilation techniques. ROCE aims to be a standard language for expressing rules about objects.

A future goal for ClaireRules is to incorporate new features from ROCE, such as the definition of rule sets and their use for explicitly controlling the triggering of rules. When rule sets represent different constraint propagation techniques, the order in which rules must be applied becomes critical, and fix-points of families of rule sets must be reached before more complex strategies (corresponding to global redundant constraints) may be applied.

Acknowledgement

The work on the new rule compiler for CLAIRE is greatly influenced by the design of the ROCE language[Mo00]. I am grateful to the members of the ROCE Team in our lab: Pierre-Etienne Moreau, François Laburthe and François-Xavier Josset.

References

- [AK93] H. Aït-Kaci. An Introduction to LIFE: Programming with Logic, Inheritance, Functions and Equations. Proceedings of ILPS'93, p.117, 1993.
- [AS97] K.R. Apt, A. Schaerf. Search and Imperative Programming. Proceedings of POPL'97, ACM Press, 1997.
- [Cas94] Y. Caseau. Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. Applied Intelligence, Vol. 4, no. 2, May 1994.
- [CL96] Y. Caseau, F. Laburthe. Cumulative Scheduling with Tasks Intervals. Proceedings of JICSLP'96, M. Maher ed., The MIT Press, 1996.
- [Eng97] R. Englander. Developing Java Beans. O'Reilly, 1997.
- [For82] C.L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem Artificial Intelligence, 19, p. 17-37, 1982.
- [KW89] M. Kifer, J. Wu. A logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited). Proceeding of PODS-89, Philadelphia, 1989.
- [McL81] B.J. MacLennan. Programming with A Relational Calculus. Rep N° NPS52-81-013, Naval Postgraduate School, September 1981.
- [Mo00] P.-E. Moreau. Open ROCE: handling Rules, Objects Control and Events. Working paper, Bouygues, June 2000.
- [NT89] S. Naqvi, S. Tsur. A Logical Language for Data and Knowledge Bases. Computer Science Press, 1989.
- [SKG87] H. Schmidt, W. Kiessling, V. Guntzer, R. Bayer. Compiling Exploratory And Goal-Directed Deduction into Sloppy Delta-Iteration. Proc. of the Symposium on Logic Programming, San Francisco, 1987.
- [SS94] C. Schulte, G. Smolka. Encapsulated Search for Higher-order Concurrent Constraint Programming, Proceedings of ILPS'94, M. Bruynooghe ed., MIT Press, p. 505-520, 1994.
- [Smo95] G. Smolka. The Oz Programming Model, Proceedings of Computer Science Today, J. Van Leeuwen ed., LNCS 1000, p. 324-343, Springer, 1995.
- [SZ88] D. Sacca, C. Zaniolo. Differential Fixpoint Methods and Stratification of Logic Programs. MCC Technical Report ACA-ST-032-88, January 1986.