An extension of the Interpreter pattern to define domain-parametric rewriting systems

Lorenzo Capra, Vincenzo Stile
Dipartimento di Informatica
Università degli Studi di Milano
Milan, Italy
capra@di.unimi.it,vincenzo.stile@studenti.unimi.it

Abstract—The Interpreter design pattern provides an elegant and natural way of implementing systems based on term-rewriting in a OO fashion. The idea is simply associating each term of a specific language, either terminal or non-terminal, to a corresponding class provided with a suitable simplify() method. Reducing a term to a normal form is thus performed through a series of recursive calls to such a method. The main weakness of this approach is that it does not take into account similarities among different domains, thus enforcing programmers to pollute generic and domain-specific rules, what often results in a wordy, hard to maintain, non-reusable code. In this paper we adapt the Interpreter pattern so that a clean separation between generic (common to different domains) and domain-specific rules is possible. That significantly helps design even complex rewriting systems. A running example which refers to a generic Logical domain will be used throughout the paper. An application to High Level Petri nets analysis will be sketched. Without any loss of generality we shall use Java as reference language.

Keywords—Rewriting systems; Design pattern; Interpreter; generic programming

I. Introduction

Term rewriting is not just a Turing-complete model of computation, but also a very convenient way to write programs in a high-level, algebraic style. According to a general model, programs are simply collections of rules (equations) which are used to rewrite (evaluate) expressions in a symbolic fashion. This process is repeated until no more rules apply, in which case the resulting expression is in "normal form" and is taken to be the "value" of the original expression.

Lots of software systems rely on this model. Let us mention: Obj [1] (the mother of all algebraic languages, based on equational logic), Maude [2] (based on both equational and rewriting logic), Elan [3] (a language based on rewrite rules controlled by strategies), OPAL [4] (an algebraic programming language with support for typical functional programming), Stratego [5] (a tool-set for program transformation), Pure (http://purelang.bitbucket.org/) (a mixture of general term rewriting and key concepts from functional programming), etc., other than well known computer algebra systems such as Mathematica, Maple, Reduce.

We are interested in discussing a design pattern for building software systems based on (symbolic) term rewriting in a OO framework. The first question which arises is: why not using any of the existing system/languages based on term rewriting? Why proposing a pattern? There are several answers. The first -let us say tautological- one is that we are not considering languages or tools, but we are reasoning at methodological/design level. Another important reason is that many programmers, while appreciating rewriting-based approaches, do not like the syntax of algebraic/functional, or even declarative languages (and their variants/blends). It is not so easy for them representing real application domains using a pure algebraic/functional style. The need for implementing structured objects and algorithms reflecting the complexity of real world’s entities is more naturally matched by a traditional OO paradigm.

The Interpreter, one of the design Patterns published in the GoF book [6], suggest a natural way for implementing a rewriting system in a OO fashion. The intent of Interpreter, coming under behavioural patterns, is usually summarized as follows: given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. Interpreter’s structure is identical to Composite’s one. The basic idea is using a concrete type to represent each grammar’s rule/symbol (terminal or non-terminal). Since grammars are usually hierarchical in structure, a hierarchy of classes of a OO programming language maps nicely. Each rule in the grammar is either a composite object (a rule that references other rules) or a terminal (a leaf node in a tree structure). Interpretation of sentences thus relies on the recursive traversal of the underlying Composite pattern. The root of the hierarchy is an abstract type declaring the method interpret(). Each concrete
(sub)class implements interpret() (by possibly accepting an argument representing the computation context) according to its internal structure. A concrete example will be presented in the sequel.

Representing the expression domain through a Composite structure, and viewing the interpret() method (hereafter more conveniently called simplify()) as the implementation of a (structured) rewriting rule, seems to be an intuitive solution for designing rule-based applications. Discussing Interpreter from software engineer’s perspective is out of the scope of this work, so let us just say a few words. As for most of patterns, its actual usefulness has been largely debated. Summarizing, the criticisms against Interpreter are that if one needs to use it, he/she is likely to create something that’s slow, ad-hoc and poorly specified. Anyway there are lots of examples of convenient practical uses of Interpreter: as symbol processing engine, SQL evaluation engine, graphing calculator, etcetera. These are all programs that solve the problem of evaluating words in a language, whatever that language may be.

From our perspective, the main weakness of Interpreter is that similarities among different domains (e.g set- and boolean expressions) are not caught, thus programmers are forced to pollute generic and domain-specific parts of rules, what often results in replicated, wordy, hard to maintain, and non-reusable code. In this paper we propose an Interpreter-based extension that, through a clean separation between generic (i.e., common to different domains) and domain-specific rules, significantly helps implement rewriting systems in a OO fashion. Other original patterns are used, such as Template Method and Factory Method.

A generic "logical" domain will be used as running example. Its original Interpreter-like representation is discussed in Section II. The domain-parametric extension of Interpreter is presented in Section III. A term rewriting based library for structural analysis of Symmetric Petri nets building on this extension is finally overseen in Section V.

II. A simple language of set-expressions

Assume we want to represent a language whose terms represent (sub)sets of an unspecified domain. The language’s terms are the constants Empty-set and Universe, (possibly indexed) variables denoting unknown sets, and the Intersection, Union and Complement operations. The symbols \( \emptyset, U, x, y, x_1, y_1, \ldots \) and \( \cap, \cup, \neg \) will be used for terminals and operators, respectively. A term like \( x_i \), or \( \neg y \) is called literal. Yet, assume we want to build a rewriting-based engine for reducing any expression to the normal form defined next.

**equivalence** Two expressions \( e_1, e_2 \) are said equivalent \( (e_1 \equiv e_2) \) if and only if they can be rewritten to the same normal form.

**normal form** \( E \) is a normal form if and only if it is either \( \emptyset, U \) or a full disjunctive normal form\(^1\) \( E = \bigcup_{i=1}^{n} t_i, n \geq 1, \text{s.t. } E \neq \emptyset, \forall t_k, t_k \neq \emptyset, \forall t_k, t_j, k \neq j \Rightarrow t_k \cap t_j \equiv \emptyset \).

These expressions are normal forms: \( \neg x, x \cap y \cup \neg x, \neg x \cap \neg y \). The following instead are not: \( x \cup y \neg \neg y (y \neg \neg y \equiv \emptyset), \neg x \cup \neg y (\neg x \cap \neg y \not\equiv \emptyset), x \cap y \cup y \neg \neg x \cup \neg y (\equiv U) \).

The grammar of the language, represented according to Interpreter, is specified by the UML class diagram in Figure 1. The root of the hierarchy is the abstract class **SetExpr**, which contains a stub implementation for the method simplify, which returns the reference to the object on which it is invoked. Non-terminal symbols (the operators) are composite objects, recursively defined in terms of **SetExprs** (their operands), while terminal ones are the "leafs" of the logical tree structure. Literal constants are defined as Singleton objects for the sake of efficiency. For the same reason, and for getting more flexibility, union and intersection are defined as n-ary operations. The (abstract) class **N_arySetOperator** just allows some refactoring of subclasses’ code.

Rewriting an expression into a normal form involves using logical equivalences, such as the double complement elimination, De Morgan’s laws, \( n\)-ary operator’s associativity, commutativity, idempotence. The classes representing operators must override the method simplify for implementing such general rules. This task turns out to be complex, especially as concerns Intersection.simplify, in which most of equivalence rules are gathered. **Union.simplify** handles the pairwise disjoin of its operands: this time-consuming operation can be optimized setting a boolean flag in the static factory method of **Union** class that avoids the algorithm to be performed again on a already disjoint form. **Complement.simplify** performs double complement elimination and (together with Intersection.simplify) De Morgan.

New objects representing operators are created during the recursive simplification of a non terminal expression. For memory saving, the simplify method should return just the reference to the current object if no reductions are performed. The equals method (inherited from **Object**) can be redefined for efficiency reasons (this is why it is marked abstract in **SetExpr**), e.g. for recognizing simple equivalences such those due to commutativity of operators. In general however

\(^1\)A disjunction of conjunctive forms of literals in which each of its variables appears exactly once in every conjunctive form.
it should perform a pseudo-syntactical equivalence check, i.e., it should never invoke `simplify`, in order to avoid possibly infinite recursive calls during a simplification. Shared sub-expressions, a well known major inefficiency source, should be referred to by aliasing references. This aspect however cannot be directly tackled at design level, rather it should be addressed at expressions’ parsing (or pre-processing) level.

The considered language of set-expressions exactly matches a "Boolean algebra": the conjunction (hereafter denoted *) maps to Intersection, the disjunction (denoted +) maps to Union, the negation (always denoted ¬) to Complement, and the logical values true, false to Universe and Empty, respectively.

Unfortunately, if one wants to represent logically similar domains (e.g., propositional logic, boolean functions, functions mapping on sets, Cartesian product,..) he/she has to build hierarchies very close to the one we have described (what is very simple to do, at the only price of an acceptable redundancy), without any possibility of reusing generic parts of rewriting. Another important issue concerns domain-specific rules. In the analysed language there are no such rules. Consider now two simple extensions, one structural, the other only behavioural:

a concrete sets (e.g., \{1, 2, 4\}) have to be included in the grammar
b variables with the same name and different indexes denote disjoint sets (x₁ ∩ x₂ ≡ ∅)

The above notion of normal form can be easily adapted considering concrete sets as literals, imposing that in a conjunctive form there are not different variables with the same name or different concrete sets, and that in the final disjunctive form there is at most one term being a concrete-set. As an example x₁ ∩ y ∩ \{1, 2\} ∪ x₂ is a normal form, while x₁ ∩ \{1, 2\} ∪ ¬x₂ is not (≡ ¬x₂).

As for a), a new "leaf" class `ConcreteSet` should be added to the hierarchy in Figure 1. This class will be provided with suitable constructors (factory methods) for building concrete sets. It could be conveniently defined as a generic type `ConcreteSet<E>`: in such a case the whole hierarchy would become generic in turn, at a very low cost. For both cases a) and b) the `simplify` methods of n-ary logical operators need to be modified so that they include ad-hoc rules for handling the language extensions (as concerns b) changes involve only `Intersection.simplify`, disjointness test relying upon intersection). The resulting code, not listed for space reasons, is a hard to maintain mixture of generic and domain-specific parts.

The solution we are going to propose permits programmers to cleanly separate domain-specific rules from generic ones, making possible to apply the pattern in different domains via generic code reuse. The approach relies on generic types and methods, present in most of modern OO languages.

III. A domain-parametric extension of Interpreter

The domain-parametric extension of Interpreter relies on a parallel hierarchy of generic abstract types (Java interfaces) composed by two logically separated parts: one, depicted in Figure 2, describing rewritable terms of an arbitrary language; the other, in Figure 3,
specifying any domain (hereafter denoted as "logical") matching a Boolean algebra. Both hierarchies (each corresponding to a Java package) have a structure which recalls Interpreter, but also contains specific elements, among which a collector of generic methods that manipulate the entities of the associated abstract hierarchy. That is analogous to the relationship between the class java.util.Collections and the interfaces of the Java Collection framework. This approach may be conveniently adopted by any language using single inheritance at class level.

Any rewritable expression (see the root of Figure 2) is simply characterized as an object which is provided with a rewrite rule (method specSymplyfiy) specific of the domain the expression belongs to. The only tricky point is represented by the type parameter (E). It allows the domains over which expressions are built to be given as arguments; for instance the parametrized type Expression<SetExpr> represents rewritable expressions which are built over the concrete domain SetExpr. Note we are not saying "expressions of sort SetExpr". This may be true in most cases (as in our running example), but not always: think, e.g., of terms that are bags of set-expressions. It might be convenient for any reasons to define these entities of type Expression<SetExpr> while their sort is Bag<SetExpr>. Thus the return type of specSymplyfiy is Expression<E>, not just E.

The Expression.convert method allows programmers to disambiguate this point. In the event the domain over which expressions are built and their sort coincide its invocation trivially performs a kind of safe cast operation from Expression<E> to E^2. Otherwise it throws a ClassCastException. The rest of the Expression hierarchy defines an intuitive interface for non-terminal symbols (operators). It could be integrated with other abstract types (e.g., representing operator’s composition), depending on everyone’s needs. The declaration of Factory Methods (build) creating operators of the same type as the objects on which they are invoked, as well as that of a method claiming the (non-)associativity for n-ary operators, stems in the generic interpretation algorithm, whose building blocks are at the bottom of Figure 2.

As said, the Expressions class is a collection of generic methods (expressed in a non-conventional UML) which perform parts of simplification on whatever may be seen as an expression. They are utility methods that could be invoked from everywhere. As an example, assocRule applies associativity in a non-recursive way on the passed collection, by replacing each operator present in the collection matching the token-type argument with the list of its operands.

### Listing 1: fixed-point normalization

```java
static <E> Expression<E> normalize(Expression<E> e, Simplifier s) {
    Expression<E> temp = null;
    while (!e.normalized() || e.equals(temp)) {
        temp = e;
        e = s.simplify(temp).specSimplify();
    }
    return e;
}
```

The core of the normalization algorithm is represented by the normalize method, reported in Listing 1. This Template Method performs a fixed-point iteration in which a generic simplify and a domain-specific simplify are invoked sequentially. The generic simplification step is encapsulated in a Simplifier object provided as a parameter (see Figure 2).

### Listing 2: unary operator’s simplification

```java
static <E> Expression<E> simplifyUnaryOp (UnaryOp<E> op, Simplifier s) {
    Expression<E> arg = op.getArg(),
    s_arg = normalize(arg,s);
    return s_arg.equals(arg) ? op :
    op.build(s_arg);
}
```

The normalize method can be invoked by other generic methods which perform parts of the whole rewriting. Listing 2 contains the method that simplifies an arbitrary unary operator by normalizing its argument. The usage of Factory Methods is illustrated.

Our solution permits programmers to focus on aspects related to the application domain, by decoupling generic reductions from domain-specific ones. Further, despite generic rules might be invoked from inside specific ones, there are no reasons for doing that. Programming efficient rewrite rules thus becomes much easier than with the original Interpreter. A blend of recursion and iteration, more than a purely recursive schema, allows memory saving, and permits writing generic algorithms in a natural fashion, as shown for the assocRule method. Especially, a uniform approach can be used for representing different abstract domains, each with a proper Composite structure and a proper set of generic rules: it is only needed to link the abstract domain types to the base hierarchy of types in Figure 2, and use the fixed-point algorithm as basic rewriting engine. Let us explain how that can be simply done through our running example.

A. An abstract Boolean algebra

The abstract grammar depicted Figure 3 describes an arbitrary "logical" domain. The central part closely
reflects the concrete type structure in Figure 1. Interfaces denoting logical operators play essentially the role of markers. The right-most side specifies the link to the Expression hierarchy. But for some overriding with specialization purposes, the remarkable things concern the methods declared in the root interface LogicalExpr. In particular, Factory Methods are declared for logical operators and constants, so that they can be used in the generic steps of simplification of logical expressions. Consider for instance the conjunctive form (of type AndOp) \( e : x \ast (y + z) \). When applying De Morgan, it is needed to build three new objects: \( t_1 : x \ast y, t_2 : x \ast z \), of type AndOp, and a new OrOp taking \( t_1 \) and \( t_2 \) as operands. This can be done by invoking the corresponding Factory Methods on object \( e \). The other methods declared in LogicalExpr (e.g., isFalse, isTrue) syntactically check for given equivalences/relations. The particular use of optional method disjoined (whose stub implementation returns false) will be illustrated later. In analogy with expr.Expressions, the class LogExpressions gathers generic methods implementing pieces of simplification of logical terms. Almost everything is needed to get a normal form from one such expression is concentrated there. Some of generic rules defined in expr.Expressions (e.g., associativity) are reused. LogExprSimplifier is a Singleton implementation of expr.Simplifier, which just invokes a dispatcher (LogExpressions.simplifyExpr) selecting the appropriate generic rule on the basis of the run-time type of the logical expression taken as an argument. The normalization of a logical expression is started by invoking Expressions.normalize (Listing 1), passing the LogExprSimplifier instance as an argument.

**B. Linking concrete and abstract grammars**

Let us go back to the concrete set-expressions’ grammar (Figure 1). If one wants to deploy the approach just introduced, he/she has simply to link the concrete hierarchy of types to the corresponding parametrized types of the abstract grammar of “logical” expressions (Figure 3). This can be done for whatever concrete domain matches a “Boolean algebra”. The link between the hierarchies is described in Figure 4, considering just the roots and the conjunction operators of either (SetExpr.simplify is renamed specSimplify).

Implementing SetExpr hierarchy now is trivial. In practice, one has to provide the abstract class SetExpr with stub implementations for all of the abstract methods declared in LogicalExpr. In particular SetExpr.simplify just returns the reference to the current object. As for SetExpr.convert, it performs a safe cast. Methods SetExpr.isTrue and SetExpr.isFalse simply check that the run-time type of their argument is Universe and Empty, respectively. The Factory Methods create objects of corresponding types: as an example SetExpr.andFactory returns either a new
Intersection, or the element contained in the passed list of operands, in the event of a singleton.

Some minor overriding may be performed, for efficiency purposes. As said, one might redefine Object.equals for executing more efficient syntactical equivalence checks. For the same reason, method isFalse could be redefined in Intersection class so that it checks for the presence of any empty element in the list of operands. And so on.

With respect to the original Interpreter implementation of set-expressions we have passed from a few hundred lines of code for the overall hierarchy to a very few dozens. This is not surprising, given that most of rewriting is done at generic level. Having isolated generic rules in the abstract grammar for logical-expressions makes it very simple introducing in the concrete hierarchy domain-specific aspects, as the two ones proposed in Section II. As concerns concrete sets, everything is needed (after having introduced the ConcreteSet class as a leaf in the...
hierarchy, provided with a suitable constructor) is to override method SetExpr.specSimplify in classes Intersection and Union: if a java.util.Set were used as internal representation for a concrete set, then the union/intersection operations on concrete sets directly map to java.util.Set.addAll and java.util.Set.retainAll bulk methods, respectively. Finally ConcreteSet.specSimplify should be overridden just to translate an empty concrete set into an Empty literal. Here is its possible encoding.

```java
Listing 3: specific simplify for a ConcreteSet
SetExpr specSimplify() {
    if (this.cs.isEmpty()) { //cs: java.util.Set
        return Empty.getInstance();
    }
    return this;
}
```

The second domain-specific extension is straightforward. It only involves overriding method SetExpr.disjoined in class Var, so that it returns true if and only if the passed argument is a Var with the same name as the current one, but with a different index. The method LogExpressions.simplifyAndOp, invoked during normalization, does the rest. The Java source code of the running example including the abstract hierarchies, and a parser for set-expressions, can be downloaded from: https://bitbucket.org/lorenzoc/interpreter/downloads/interpreter.zip

C. Complexity and performance issues

A theoretical study of time and memory complexity of the proposed approach is out of the scope of this work, for two main reasons. First, we focused on software engineering aspects related to generic code reuse, domain-parametric code writing, separation of concerns. Secondly, complexity strictly depends on the considered abstract domain (and its instances), and the adopted normal form (e.g., bringing an expression to a disjoint normal form may have a complexity exponential in the number of involved literals).

Nevertheless, efficiency should be taken into great consideration when implementing both generic and domain-specific parts. The generic rewriting engine is conceived to minimize creation of new objects at run time, and to support rewriting modulo commutativity/associativity. Yet, it gives the programmer the faculty to declare terms (e.g., literals) as normalized, so that the fixed point iteration immediately ends (listing 1).

An interesting question is: does the design pattern which builds on decoupling generic simplification steps from specific ones incur a significant loss of performance, compared to a more naive simplifier build-

IV. Related work

Generic programming has become a hot research topics. Many efforts are currently devoted to trying to make this technique more and more safe and adaptable. Traversal/modification algorithms in the functional programming world are an example. An approach which is worthy of mention, for some similarity of intents with the domain-parametric Interpreter extension, is presented in the "Scrap your boilerplate" (SYB) series of papers. The SYB approach allows the programmer to write generic functions that can traverse arbitrary data structures, and yet have type-specific cases, by exploiting the Haskell’s rich type system. In origin the approach required all the type-specific cases to be supplied at once, while Haskell’s type classes support open functions that can be extended with new type-specific cases as new data types are defined. In [7] the SYB approach was extended to support this open style. A C++ version of SYB exists [8]. The SYB pattern in Haskell depends on a number of features that are not directly available in C++, most notably rank-2 types, and polymorphic type extension. The C++ implementation of SYB, however, closely resembles the original Haskell version. Not surprisingly, the C++ solution depends heavily on the template feature, function overloading, and other relevant idioms including function objects and template member functions. Instead it does not exploit object-oriented features, hence it cannot be considered a true design pattern. The natural approach to recursive traversal in the object-oriented world is the well-known "Visitor" pattern. The basic formulation has severe limitations. For one, the action to be performed and the traversal strategy are mixed together. This was remedied in part by visitor combinators [9], which allow traversal to be encoded separately from actions.

Even if SYB approaches and, especially, Visitor’s extensions might be interesting alternative design choices, we believe that the easiest and most natural way for implementing rewriting systems in a OO fashion is adhering to Interpreter. The Interpreter ex-
tension we have proposed tries to overcome the limits of the original version in the generic programming perspective. The only features required at language level are generic types/methods and abstract types.

V. An application to HLPN analysis

Symmetric Nets (SN), formerly known as Well-formed coloured Nets [10], are a High Level Petri net formalism with syntax constraints that allow automatic discovery and exploitation of behavioural symmetries in models. SN’s Places and transitions are nodes of a bipartite graph, both associated to finite color domains. They are connected through arcs annotated with expressions taking the form of (guarded) tuples (i.e., Cartesian products) of basic class-functions.

The computation of structural properties (conflict, causal connection, mutual exclusion, invariants..) of SN models, hereafter simply SN’s calculus, requires manipulating terms of a language whose grammar is a small extension of SN’s arc functions’ one [11]. A minimal set of functional operators (intersection, difference, transpose, and composition) are used, with respect to which the language was shown to be close.

A Java library was developed [12] implementing all basic functional operators of the calculus. The whole framework is conceived as a rewriting system: every basic functional operators invokes in sequence domain-specific and generic rules. This approach makes use of parallel abstract hierarchies representing generic domains, to which concrete structures have to be linked. Terms’ normalization is performed through an iterative fixed point algorithm that invokes in sequence domain-specific and generic rules.

Original Factory Method and Template Method, in addition to features such as generic types/methods, are used. A library for structural analysis of Symmetric Petri nets, strictly adherent to the proposed pattern, has been cited as a successfully application case.

References