

TermWare: A Rewriting Framework for Rule-Based Programming Dynamic Applications

A. Doroshenko¹, R. Shevchenko²

¹Institute of Software Systems of NASU,
Glushkov prosp. 40, Kiev 03187, Ukraine, dor@isofts.kiev.ua

²Gradsoft Ltd., Kiev, Ukraine, <http://www.gradsoft.kiev.ua>

Abstract

In recent years light-weighted formal methods are of growing interest in construction and analysis of complex concurrent software system. A new rule-action based term rewriting framework, called TermWare, is proposed and its application to software system analysis are described to provide better cost effectiveness of software maintenance under varied requirements and specifications of operation. The main advantage is light-weighted formal model based not on computation semantics but on particular properties of software system to be analyzed. Such approach eliminates the need in full formal analysis of software system and allows extreme flexibility of applications in two major concerns: high adaptability to changeable environment and easy reengineering and component reuse. The language and formal semantics of the system are defined. A new semantic model, called term system with action, is proposed for TermWare. A case study with some representative examples in source code analysis and software development with TermWare framework is presented.

1. Introduction

Rewriting rule paradigm is an important direction in symbolic computations application in software system development [1]. Today there exist more than one hundred rewriting rules systems which are supported by both research groups and commercial organizations. In its evolution rewriting systems reflect changes in both software development methodologies and software application field. From conventional job of string processing (may be the most perfectly embodied in Refal [2]) that is fundamental for each software, rewriting systems have reached and successfully applied to problems of program analysis and transformations, automatic generation of software documentation and automation of inference in logic theories. A particular class of rewriting rule systems consists of comprehensive ones such as APS [3], Maude [4] or Stratego [5] that appeared recently as an alternative to systems of logic programming. Their reference feature is availability of the rules of representation of a subject domain objects as algebraic terms and universal possibility of these term transformation through policies of application of the production rules based on non-logic principles, often in a form of imperative sequences (or strategies) of looking up and applications of the rules.

The field of symbolic computation has been traditionally elaborated in Cybernetics Center of National Academy of Sciences of Ukraine from early days of “MIR” computers and interpreted-by-hardware symbolic computation language ANALITIK [6] to more contemporary systems of algebraic programming [3]. In this paper another system of symbolic computations is presented (called TermWare) that is based on rewriting term paradigm and intended for efficient solving problem in software engineering for highly dynamic applications.

TermWare belongs to the last mentioned class of comprehensive rewriting systems. Also it is closed to fundamental approaches in contemporary directions software design such as strategic programming [5] and design patterns [7]. In this paper syntax and semantics of the language is described and a method of embedding the system into application is presented. The range of application is especially intended for problems of programming dynamic applications

characterized, on one hand, by high degree of interactions with application environment and, on the other hand, by high variability of the requirements and necessity of reprogramming such systems.

TermWare differs from the majority of other systems of this class in both semantics of used facilities and technology of their implementation. TermWare language is not the universal programming language in the sense that it is not intended for writing full-fledged software system. Instead it is like a coordination shell language [8] and it is aimed to compose domain specific parts of the applications built into an applied system for implementing functions of interaction of the system with its environment.

The semantic features of TermWare language essentially differs from conventional rewriting systems. Usually programming with rewriting rules uses formal model of a subject domain with rewriting rules to be applied to implement classic “closed” computation paradigm in functional style where input/output operations and interactions are not represented explicitly and are rather “side effects” than regular events. Such approach is no more commonly applicable in software industry for a number of reasons. First, modern software systems are often “open” and can hardly be described as algorithms in classic functional sense, i.e. only as input/output transformers. Very often a software system represents in itself some behavior including interaction with an environment. Second, construction of full formal definition of a subject domain is often too labor- and time-intensive process. Moreover, the subject domain often includes many already constructed elements of environment like relational databases (DB), object models, protocols etc. So to use all these components in one declarative model and to give formal semantics to already constructed imperative models seems too tedious and expensive.

Formal transformations are widely recognized as high-level and powerful facility that gives advantages of strong foundation of source code analysis and in many cases can provide full automation of solving problems in program manipulation and software maintenance. On one extreme of a spectrum of formal transformation methodologies there is full-fledged systems based on wide spectrum language. Known example of such kind of systems is FermaT transformation system [9] – an industrial-strength transformation system designed for forward and reverse engineering and program comprehension. However despite knowledgeability of this approach complete utilization of its advantages is very complicated, expensive and needs extensive support at all stages/levels of software evolution. On the other extreme in recent years there are of ever growing interest to “light-weighted” approaches to formal methods for program analysis aimed to reduce costs and improve efficiency of software maintenance [10-12]. They try to focus attention to particular features of programs where formal (symbolic) computation can be both a good base for modeling/specification purposes and, on another hand, an efficient tool for computation of accurate program information avoiding full formalization of program semantics. The latter approach is particularly valuable in context of dominating trend of component-oriented software development that assumes building new software system from pre-existing components with some glue between components and new functionality. To reuse components effectively we need some facilities to capture interface specifications of components and methods of interface compatibility validation. Formalisms like UML widely used in design and documentation appear too informal and heavy for these purposes.

Interaction (e.g. input/output) operations usually understood in declarative programming as the side effects are too important to be ignored in formal model of computation. So in TermWare a new formalism of *term systems with actions* is proposed for the declarative description of programming distributed software. It includes the description of system functionality as a term system and system’s interactions with an environment that is represented as dynamic base of the facts. Thus, instead of hiding imperative style of programming in side effects TermWare offers some kind of immersion of imperative operations into logic of declarative program. The term system is open in the sense that it is interactive and its behavior is determined not only by program code but also by state of environment.

The outline of the paper is as follows. In next section formal model of TermWare language and the system will be presented. In section 3 implementation issues and illustrative example of language usage are given. Two examples of case studies of TermWare application are described in section 4. Concluding remarks are presented in section 5.

2. Formal model

The formal model of TermWare is constructed as an algebra of terms by expressions of the form $f(x_1, \dots, x_n)$ with variables and basic data types similarly to other functional programming languages. The only difference from the common approach is that constructor of an ordered set is the designated term and the operations of substitution on these terms keep the order. Computation is defined by sets of the rewriting rules together with policy of their application like other rewriting system such as APS [3] or Stratego [5].

However unlike languages of algebraic programming (such as OBJ and Maude [4]) the model of computation in TermWare is determined in terms of behavior of the programs, i.e. input actions and reactions. More precisely, in conventional systems of the rewriting rules computation is a transformation of an input term to some canonical form with the help of a set of the rewriting rules. Thereby the language determines some kind of functional relation of the form $y=f(x)$ where x is the input (term) of computation and y is its output (term). In TermWare the set of the rewriting rules determines not such computation as a whole but only one step this computation of the form $f(x,s) \rightarrow (s',y)$, where x is input signal and y is output one, s is an internal state of the transformer. For lack of space in this section there are presented rather simple syntactical features of TermWare formal model only to demonstrate basic ideas and facilities. More details can be found at the project site: <http://www.gradsoft.kiev.ua>.

2.1 Basics of the language

Below everywhere if not specified apart inferior indexes are considered to belong to enumerable set.

The alphabet of the TermWare language comprises constants c_i of primitive types: INT (integers), BOOL (Boolean constants **true** and **false**), STRING (set of strings in Unicode), ATOM (set of atomic noninterpreted values; among them empty atom NIL plays special role). Besides the language includes also set of terminal symbols t_1, \dots, t_N , set of propositional variables x_i , brackets '(' and ')' and sign of comma ',' set of functional symbols f_i , symbol of environment S , set φ of symbols of readings of the information from environment and set δ of symbols of effects on environment (actions).

The terms of the language are constructed under the following scheme. First the set of concrete terms T_c is defined in following several steps: $c_i \in T_c$; and if $x_1 \in T_c, \dots, x_N \in T_c$ and f is a functional symbol of appropriate arity then $f(x_1, \dots, x_N) \in T_c$. Similarly the set of substitutional terms T_v is defined: if $x \in T_c$ then $x \in T_v$; and if $x_1 \in T_v, \dots, x_N \in T_v$ and f is a functional symbol of appropriate arity then $f(x_1, \dots, x_N) \in T_v$. A term from the set of difference $T_v \setminus T_c$ is called the term with free variables and for each term t the set of its nonterminal symbols, i.e. the set of free variable this term, is designated $\nu(t)$.

Further following syntactical functions of term system and their properties are defined:

- function $typename(): T_v \rightarrow \text{STRING}$ returns syntactical type of an argument term; it can be: one of the names "BOOL", "INT", "ATOM" or "STRING" for primitive terms; "x" for proposition variables and "T" for composite terms;
- function $name(): T_v \rightarrow \text{STRING}$ returns name of the head character of argument term;
- function $arity(): T_v \rightarrow \text{STRING}$ returns arity of the character of argument term;

- function $subterm(i,t): \text{INT} \times T_v \rightarrow T_v$ computes i -th subterm of the term t if $t \in T_v$ and $i \leq \text{arity}(t)$; if $i > \text{arity}(t)$ then $subterm(i,t) = \text{NIL}$;
- function $equal(x,y): T_v \times T_v \rightarrow \text{BOOL}$ returns **true** value in case of identical coincidence $x=y$;
- function $less(x,y): T_v \times T_v \rightarrow \text{BOOL}$ returns **true** value if y follows x in the sense of ordering terms.

Thus, the set of terms (to enumeration within propositional symbols) is completely ordered. It allows to have the constructor of an ordered set $set(x_1, \dots, x_N)$, i.e. the term that means the ordered set of N elements. This implies that if $t = set(x_1..x_N)$, $subterm(i,t) = x$, $subterm(j,t) = y$, $0 < i < j \leq \text{arity}(t)$ then $less(x, y)$. Now it is possible to define $v(t)$ at syntactical level as follows: $v(c_i) = \text{NIL}$, $v(x_i) = \{x_i\}$, $v(f(t_1, \dots, t_N)) = \bigcup_{i=1}^n v(x_i)$.

The expression $subst(t, x, s)$ is the substitution s in t instead of free variable x . Equivalent notation for substitution $t[x,s]$ can be also used. The term $bound_unify(t_1, t_2)$ means operation of unification t_1 and t_2 with set of bound proposition variables of terms t_1 and t_2 ; term $free_unify(t_1, t_2)$ designates operation of unification t_1 and t_2 where free variables of t_1 and t_2 are previously renamed so that $v(t_1) \cap v(t_2) = \emptyset$.

Semantics of term rewriting rule without interaction with environment is defined by expression $apply(t, x \rightarrow y) = subst(y, free_unify(x, t))$, where $x \rightarrow y$ is a rule of a rewriting of term x in y . An environment S can be thought as a database (of facts) having defined acts of interactions of a term with environment via a function: $\varphi: S \times T_c \rightarrow T_c$ for obtaining information from S and a function $\delta: S \times T_c \rightarrow S$ for delivering information in S .

The main object of our consideration – term system – can be identified as a pair $\tau = (S, R)$ where S is an environment and R is a set of the rewriting rules presenting interaction with this environment. A rewriting rule from R is a term with four arguments $rule(x, in, y, out)$ which for convenience we shall also designate as $x[in] \rightarrow y[out]$ that stands for "rewrite x in y under condition in having applied to environment operation out ". Taking into account notations introduced above general expression the scheme of rewritings can be given in the form:

$$apply((S, t), x[\varphi] \rightarrow y[\delta]) = (\delta(S, free_unify(z, out)), z),$$

where $z = subst(y, free_unify(x, t, \varphi(S, in)))$ and t is a term. Environment is defined as class in object oriented language (Java in our case), so φ and δ are terms which includes calls for methods of environment class.

In TermWare an important concept is of a name of term system and set of term systems. Term systems usually are thought together with subject domains which can be nested. So name of a term system can be composite one. The name of a term system is an atom or special term $_name(x_1, \dots, x_n)$ where x_1, \dots, x_n are atoms. Interpretation of the term $_name(x_1, \dots, x_n)$ is determined by a name function $\phi: T_c \rightarrow \Phi$ which returns a name of the term system in some (hierarchical) space of names. If the TermWare interpreter meets a composite name it searches in a directory tree of an operating system for the file with a path whose beginning corresponds to problem areas (domains). Term system itself is defined as a term of the form $System(name, ruleset, facts, strategy)$ where $name$ is a name of the system, $ruleset$ is a set of rewriting rules with interaction, $facts$ is a database of the facts of environment, $strategy$ is a policy of application of the rewriting rules that determines some sequence of application of the rules.

As simple examples let us consider some predefined strategies of TermWare system. *FirstTop* strategy is to find the first matching "from top the most left" in term to be a reduced and to return the result. *BottomUp* strategy at first recursively find the most down matching (bottom) and then moving up trying to apply the same policy to the obtained outcome. *Top* strategy is nonrecursive

policy which works only if the upper term is precisely matched to a pattern of any of a rule. For example if we have following set of the rules:

```
ruleset(
    p($x,q) -> q,
    p(q) -> y,
)
```

which must be applied to the term $f(p(p(q),q))$ then *FirstTop* strategy returns $f(q)$ by the first rule, *BottomUp* strategy yields the same applying first the second rule and then the first one. *Top* strategy can not be applied.

The function of reduction of a term in the system is expressed by a term $reduce(s, t,)$ where t is the term to be reduced in a system with the name s .

2.2 Action semantics

The central point that differs TermWare from conventional approaches is the introduction interaction of it with environment in operational semantics of term systems. This extends usual approach to term rewriting (set of rules that transform input term set to output one) to behavior modeling that reflects much more dynamic features of contemporary software systems. This is because in applied problems unlike theoretical vacuum there is always some environment from and which we can obtain/deliver necessary information and often we need to have description of sequences of exchanges with environment. Besides the transformation rules set can also be dynamic. Among the samples of such applications there are set of rules for determination of actions in a business process where the set depends on process variables, code generation process that depends on certain architecture, output rules presented by facts of the deductive database.

Consequently there is a need to define operational semantics in such away to include environment interaction and rules set modification dependent on state. For this purpose more elaborated notion of term rewriting machine is introduced as the transition system $\langle S, E, \varphi, \delta \rangle$. In this quadruple the pair of S and E represent set of states of the machine where $S = \langle S_t, S_r \rangle$ comprises term set S_t and active rewriting rules set S_r ; E is environment representing in the system (database of facts). Transitions of this machine are described by functions $\varphi: S \times X \times E \rightarrow S \times Y$ – the system transformation function and $\delta: E \times Y \rightarrow E$ – the environment reaction function, in following transition rule: $\langle S, E, X \rangle \rightarrow \langle \varphi(X, E, S)|_S, \delta(\varphi(X, E, S)|_Y) \rangle$. Here expression $x|_Y$ denotes projection x onto coordinate Y in the sense of set theory. The function φ realises its operation according to some strategy of rewriting that influences environment as “side effect”. The system is normal if φ defines fixed point of application S_r to S_t .

Thus, TermWare gives a direct mapping of logical model of interactions to the programming language having clones of modern means of support of inheriting and a hierarchical name system. Below we shall show, that such systems more naturally mirror representative problems of programming in a level to the architecture and are suitable not only for the description of computing algorithms, but also for the description of behavior of programmatic complexes consisting of several interacting subsystems.

3. Implementation and examples

Technically TermWare is made as a library of Java classes to be built in applied software system and can be also viewed and used as the software agent seamlessly built into a general infrastructure of an application. To be used in applications some additional means and agreements are adopted in language syntax. Particularly, names of propositional variables begin with dollar sign followed by sequence of characters or digits, for example $\$x10$, $\$saved_args$; constructors of terms have a conventional functional form $f(x_1, \dots, x_n)$; the comments are strings beginning with the character “#”.

For convenience syntax of TermWare language is adopted similar to C syntax. Expression $x[y] \rightarrow z[v]$ is used instead of $rule(x,y,z,v)$, that means a rewriting rule with an input pattern x , condition y , output pattern z and action v . If one or several members of the expression missed they can be omitted. There are also other abbreviations in the language: $\{x:y\}$ is an abbreviation for *set pattern* (x, y); $\{x_1, \dots, x_N\}$ stands for *set*(x_1, \dots, x_N); x in y used as *_in* (x, y); $[x_1, \dots, x_N]$ stands for *cons*($x_1, cons(x_1, cons(\dots cons(x_N) \dots))$); $x? y: z$ – for *ifelse*(x, y, z). Infix notation is used for binary operations like $x+y$ for *plus*(x, y).

The rewriting rules are packaged in sets with the help of term rule sets. For example, set of the rules depicting Boolean algebra is described below where the term system is introduced with the expression *System*(x,y,z,v):

```
System(BooleanAlgebra,
  default,                               # databse of facts
  ruleset(
    $x & ($x => z) -> $z,                 # rule for implication
    not($x & $y) -> not($x) | not($y),    # De Morgan rule
    not($x | $y) -> not($x) & not($y),    # De Morgan rule
    $x & ($y | $z) -> ($x & $y) | ($x & $z), # distributive property
    not(not($x)) -> $x,                   # rule for double negation
  ), FirstTop);                           # police of rewriting
```

Here the first subterm *BooleanAlgebra* is the name of the system and the second one *default* is the name of the database of the facts (empty in this case) followed by the set of the rewriting rules. The last expression *FirstTop* is a policy of rewriting, (will be explained below). The database of the facts and policy of a rewriting are determined at a level of procedural language depending on a subject domain. How it can be done will be seen from an example in following section.

Let us pay attention for other important features of the language. To make possible structuring information into subsystems and to organize application specific domains the name system in TermWare uses concepts of name hierarchies and inheritance like in Java classes to integrate packages. Term systems can reuse the information with the help of import of rules similarly to inheritance in object-oriented languages. Following piece of code gives an example of reusing given above Boolean algebra rewriting rules in Boolean logic unit:

```
System(BooleanLogic,default,
  ruleset( import(BooleanAlgebra),
    true => $x -> $x,
    false => $x -> not($x),
    true | $x -> true,
    false | $x -> $x,
    true & $x -> $x,
    false & $x -> false,
    not(true) -> false,
    not(false) -> true
  ),
  FirstTop)
```

4. Case studies in embedding TermWare in applications

Previous examples of rewriting systems are of limited expressiveness as they show only basic elements of the language. Real significance and novelty of TermWare approach can be seen in applications where dynamic nature of database of the facts and action semantics is exploited. Partly it is called by that TermWare is intended for embedding in the applications, and the

database of the facts and set of actions depends on concrete features of the applications. To embed elements of logic inference into applications and to design a set of the database facts and actions while using TermWare one usually follows a number of steps:

- determining a subsystem in the application where declarative rules would be reasonable to use;
- defining interactions of the subsystem with a rest of the an application and elaborating list of functions for obtaining of the information (conditions) and delivery of information (actions);
- development of obtained set of interactions as Java-classes which should implement the interface *IFacts*;
- development of strategies of application of the rules in such a way that either one of existing strategies is chosen or another implementation of the interface *Istrategy* is coded;
- embedding TermWare in the application either by a direct call for object constructor *ItermSystem* or by definition in TermWare of conformity between names of a database of facts (strategies) and classes realizing them in Java.

The general arrangement of creation of a database of facts by implementation the interface *Ifacts* looks like following:

```
public interface IFacts
{
public String    getDomainName();
public boolean   check(ITerm t) throws TermWareException;
public void      set(ITerm t) throws TermWareException;
}
```

While interpreting rules of a rewriting an inquiry of the information from an environment occurred TermWare invokes a method *check* of the associate database of the facts. If an action occurred the method *set* is invoked. There is also an auxiliary class *DefaultFacts* which independently executes an analysis of arguments by means of the reflective applied Java interface.

Terms are represented as objects realizing the interface *ITerm*. In the *ITerm* signature there are defined operations of a type definition, access to the typed values, unification, equivalence and lexicographic matching. Developer of the application sees term system as a set of copies of the class *ItermSystem* with a possibility to control rewriting rules and reduction of terms. The knowledge base also can contain imperative elements written as Java classes. Thus there is an opportunity to use declarative model of programming in software systems written in imperative style where it is necessary.

Limited space of the paper does not allow giving here the details of the signature and describing real life examples completely. Instead we will demonstrate capabilities of programming by a few small but representative examples.

4.1 Life game

First example is a widely known Convey's game "Life" that is given here as a good example of highly dynamic application that uses database of facts for maintenance information about the game field. The term system for this application looks like following:

```
domain (examples,
  system (Life1, LifeDB,
    ruleset (
      # $T - set of pairs to test.
      {l($i,$j):$T} [existsCell($i,$j)) && (n($i,$j)==2|n($i,$j)== 3) ] ->
      $T [ putCell ($i,$j) ] ,
```

```

    {l($i,$j):$T} [n($i,$j) == 3] -> $T [ putCell($i,$j)],
    {l($i,$j):$T} [n($i,$j)<2 || n($i,$j)>3] -> $T [removeCell($i, $j)],
    {} -> $T [ [ showGeneration(), createNewTestSet($T) ] ]
    ),
    FirstTop)
);

```

As it can be seen from a database of facts the following facts and conditions are requested: 1) *existsCell(i,j)* that stands for “whether there is a cell at the address (i,j) ”; and 2) $n(i,j)$ which is a request for quantity of the neighbours of the cell (i,j) . Actions are following:

- *putCell(i, j)* – to place a cell at the address (i,j) ;
- *removeCell(i, j)* – to eliminate a cage;
- *showGeneration ()*– to show the next breed;
- *createNewTestSet (\$T)* – to generate new test set of cells.

In shorthand form appropriate class of a database can be seen in:

```

class LifeFieldFacts extends DefaultFacts
{
    private int nX _;
    private int nY _;
    private boolean [] [] field _;
    private boolean [] [] nextField _;
    private Canvas drowing _;

    public Term existsCell (int x, int y) throws TermWareException
    {
        return ITermFactory.createBoolean(field_[x%nX][y%nY]);
    }

    public void putCell (int x, int y) throws TermWareException
        //... implementation skipped.

    public void generateNewTestSet (TransformationContext ctx, Term t) throws
    TermWareException
    {
        // implementation skipped.
        // ...
        // fill the value of the propositional variable.
        ctx.getCurrentSubstituion().put(t, retval);
    }
}

```

Let us pay attention to substitution of set instead of a proposition variable in operation *generateNewTestSet*. This example illustrates a transfer of information from an imperative part of the program in its declarative part.

4.2 Source code analysis and refactoring

The second example is a source code analyzer for which input term is a syntax tree of analyzed source code, database of facts consists of the program environment of our analyzer and semantics model of software, rewriting rules with actions are used for extracting semantics information from syntax tree and put one into semantics mode.

Processing abstract syntax trees (AST) as terms we can give an opportunity to perform well defined transformations on quite reach term algebra in declarative form.

Look for example at the following AST tree for Java expression:

```
class X
{
    int x() { return 1; }
}
```

is mapped to term as follows:

```
java_class_declaration(
  java_identifier("X"),
  java_empty_type_properties,
  java_name([java_identifier("java"),
             java_identifier("lang"),
             java_identifier("Object")]),
  empty_list,
  cons(
    java_method_declaration(
      java_identifier("x"),empty_list,java_int,empty_list,empty_list,
      [java_return(java_integer_literal("1"))]
    )
  )
)
```

We can discover code patterns (for example, violations of programming style) just by running rewriting engine over source code of system. For example, next rule is check for empty catch clause in Java code and change one to provide exception logging.

```
java_catch($formal_parameter, java_empty_block ) →
    java_catch($formal_parameter,
java_dot(java_identifier(LOG),
java_identifier("error"), $formal_parameter) ) )
    [violation(EmptyCatchClauses, \" empty
catch clause \",$formal_parameter)"]
```

The next rule is to check for generic exception catch clause:

```
java_catch(
  java_formal_parameter($x,java_name([java_identifier(\"Exception
\")] ),$final),
  $block)
    → PART
    [violation(GenericExceptionCatchClauses,\"generic exception
catch clause\",$x) ]
```

More details and examples on rewriting rules techniques for static analysis of source code which use action semantics for extracting semantics properties from source code to model can be found in [13].

5. Benchmarking and comparison with other systems.

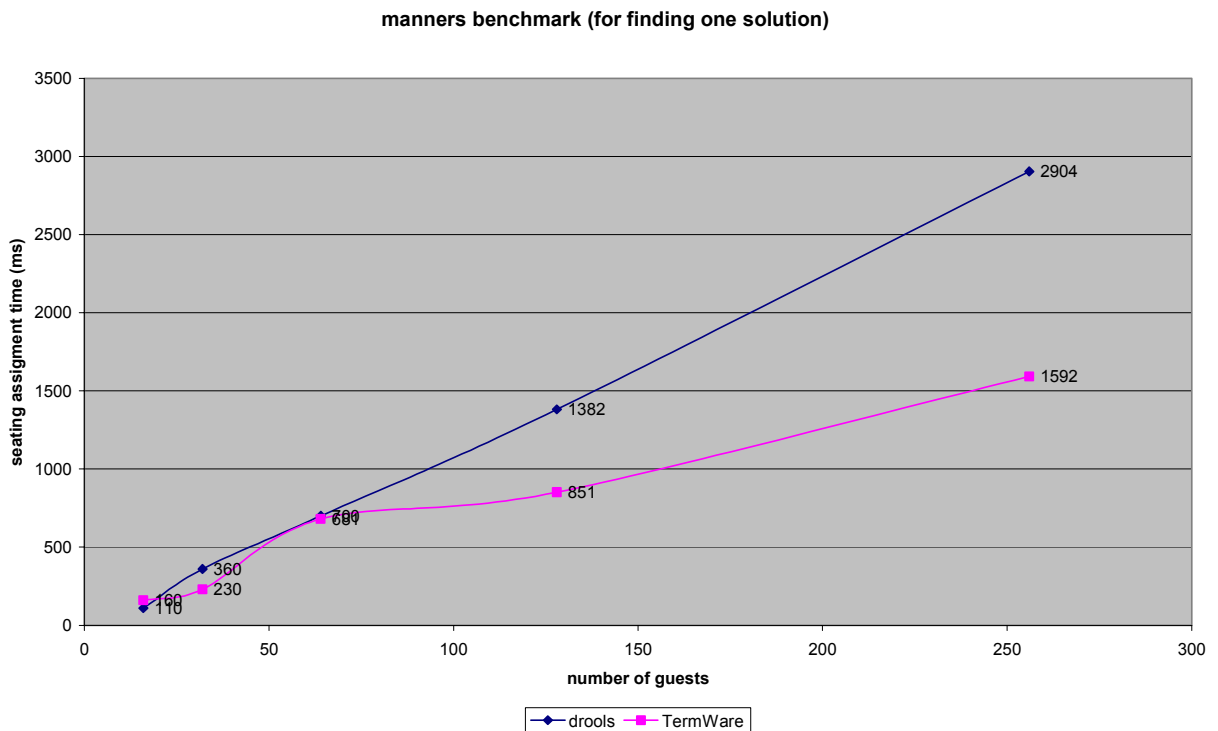
The declarative programming are slowly adopted by industry, one of reasons is an opinion about inefficiency of high-level languages. One of ways to change this situation is to provide industry accepted criteria of execution speed, i.e. set of standard benchmarks.

The most known benchmark for rule-based systems is “Miss Manners” problem, explained in [20],[21]: guests on party must be seated so that neighbors are of different sex and share the same hobby.

So, we implemented Manners benchmark and compare one with implementations of one in Drools [19] on the same datasets.

Note, that direct comparison have limited sense, since Drools use many-objects, many-patterns matching, while TermWare follows one-object, many-patterns approach. So, direct translation of original algorithm is impossible: it was modified to operate with one working term at a time. Also note, that rule set used by Drools' Manners example also is not identical to the "official" Miss Manners benchmark.

Result are summarized on the next diagram:



On X axis we see number of guests in dataset, on Y – time for finding solution in milliseconds. All tests was performed on Celeron 1.1 GHz notebook with 512Mb RAM with Sun JVM 1.4.2 with default options on Windows XP OS. Datasets was identical, Drools example was modified to accept dataset from external source instead generating one.

We see, that both Drools and TermWare implementations are quite fast: rule computation bottleneck is less than typical bottleneck from remote database. TermWare outperform Drools on large datasets, i. e. coordinate Rete net for set of terms is more time consuming, than perform rewriting of one big term.

6. Conclusion

As a programming method rule-based programming is a powerful facility for development of software systems which involve significant decision-making capability. Exploiting rules engine is especially important to provide rules flexibility and to meet needs of code to be maintained over time. The system TermWare is aimed at development of highly dynamic application rule-based systems with strong demands of built-in intelligence, fast interactions, cost effectiveness and reusability of developed software. The model and examples show how powerful and productive can be term rewriting techniques in solving practical issues of software design and engineering.

Development of TermWare system is quite in the mainstream of modern software industry tendencies of integrating different software engineering techniques and introducing light-weighted methodologies in software analysis and design processes [10-12]. Rule-based processing has also growing up rapidly and even is being standardized (see, for instance [14]). There are also some other systems proposing embedding rewriting rules subsystems in Java, like Jess [15], Drools[19] intended to use rewriting rules for construction of expert system, and ASDF [16] for transformations of syntactical trees analysis accordingly.

TermWare system based on rewriting techniques has been successfully applied in industrial and research projects for cost effective reengineering program source code by means of formal program transformations [17-18]. It appeared that Termware can be used also for managing business logic and improvements in interface mechanisms described in languages like CORBA's IDL [17]. Usually, IDL can provide a kind of general guarantee of software component interoperability for customers. But to ensure user confidence in software services in particular context and on particular network platform some additional efforts are needed.

A prototype of the language and system of TermWare is freely available for noncommercial usage at <http://www.gradsoft.kiev.ua>

References

1. Dershowitz N., Jouannaud J.-P. Rewrite Systems, in J. van Leeuwen (ed.) "Handbook of Theoretical Computer Science", Vol. B: Formal Models and Semantics. Boston: Elsevier and MIT Press, 1990.– P. 243-320.
2. Refal-5: Programming Guide and Reference Manual . Holyoke MA: New England Publishing Co., 1989.
3. A.A. Letichevsky, J.V. Kapitonova, A.E. Doroshenko, V.A. Volkov, Parallel Symbolic Simulation Using the Algebraic Programming System, in “Algebraic Engineering”: Proc. Intern. Conf. on Semigroups and Algebraic Engineering. – Singapore:World Scientific, 1999.- P.350-360.
4. Winkler T., Programming in OBJ and Maude // P. Lauer (ed.), Functional Programming, Concurrency, Simulation and Automated Reasoning. - LNCS.- 1993. - Vol. 693.- P. 229-277.
5. Visser E. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5 // A. Middeldorp (ed.), Rewriting Techniques and Applications (RTA'01) / LNCS. - 2001.- Vol. 2051.- P. 357-361.
6. Glushkov V.M., Grinchenko T.A., Dorodnitsyna A.A. et al. ANALITIK-74. Kibernetika, – 1978, №5, pp. 114-147 (translated from Russian).
7. *Design Patterns: Elements of Reusable Object Oriented Software* / E. Gamma, R. Helm R. Jonhson, J. Vlissides. -Reading,MA: Addison-Wesley, 1995.
8. Gelernter D. Carriero N., Coordination Languages and Their Significance. Communications of the ACM, **35**(2), 97-107, 1992.
9. Ward, M., Bennett, K.H.: Formal Methods to Aid the Evolution of Software. Int. Journal of Software Engineering and Knowledge Engineering, 1995, vol. 5(1); pp 25-47.

10. Liang, D.; Harrold, M.J.: Light-Wight Context Recovery for Efficient and Accurate Program Analysis. In Proc. 22-nd Int. Conf. Software Engineering, (June 4-11, 2000, Limerick, Ireland), ACM Press, New York, 2000; pp. 366-406.
11. The Agile Manifesto, www.agilemanifesto.org
12. J. Nawrocki, M. Jasinski, B. Walter and A. Wojciechowski. Extreme programming modified: embrace requirements engineering practices, Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02), Essen, Germany, pp. 303-310, 9-13 September 2002
13. R. Shevchenko. JavaChecker: a static analysis of software by means of rewriting techniques. Problems in Programming, N 4, p. 223-230. 2004. (in Russian).
14. JSR-000094 Java™ Rule Engine API. - <http://www.jcp.org/aboutJava/communityprocess/review/jsr094/>.
15. Jess – the expert system shell for the Java Platform.- http://herzberg.ca.sandia.gov/jess/ri_overview.shtm
16. M. G. J. van den Brand, H. de Jong, P. Klint, P. Olivier. Efficient annotated terms. Software, Practice & Experience. **30**(3) 259-291, 2000.
17. Shevchenko, R., Doroshenko, A.: Techniques to Increasing Performance of CORBA Parallel Distributed Applications, in PACT-2001, Proc. 6-th Int. Conf. on Parallel Computing Technologies. Lect. Notes Comput. Sci., vol. 2127, 2001; pp. 319-328.
18. Shevchenko R., Doroshenko A. Managing Business Logic with Symbolic Computation / in "Information Systems Technology and Applications": Proc.2-nd Int. Conf. ISTA'2003, June 19-21, 2003, Kharkiv, Ukraine (M. Godlevsky, S. Liddle, H. Mayr, eds.)// Lecture Notes in Informatics.- 2003.- Vol. P-30.- P.143-152.
19. drools: Drools rule engine. <http://www.drools.org>
20. Gerald Kiernan, Cristophe de Maindreville, Eric Simon. Making deductive database a practical technology: a step forward. Inria technical report N 1153, 1990
21. D.A. Brant and T. Grose and B. Lofaso and D.P. Miranker, Effects of Database Size on Rule System Performance:Five Case Studies, Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), 1991