УДК 51:681.03

*А.Е. Дорошенко, Р.С. Шевченко*

# СИСТЕМА СИМВОЛЬНЫХ ВЫЧИСЛЕНИЙ ДЛЯ ПРОГРАММИРОВАНИЯ ДИНАМИЧЕСКИХ ПРИЛОЖЕНИЙ

Описана система символьных вычислений, предназначенная для встраивания в промышленные приложения с целью обеспечения их гибкости и настраиваемости соответственно изменяющимся требованиям и условиям функционирования. Предложен метод погружения настраиваемых программных систем в промышленные комплексы, основанный на использовании декларативных средств программирования при описании взаимодействий программных компонент со своим окружением. Метод позволяет значительно уменьшить стоимость разработки сложного программного обеспечения, а также решать задачи анализа и обеспечения надежности программного кода.

## *Введение*

Системы символьных вычислений имеют богатую историю и применяются для различных целей с первых лет существования компьютерной науки как самостоятельной области. Одним из направлений в разработке систем символьных вычислений являются системы переписывающих правил (rewriting rules systems) [1]. На сегодня в мире существует более сотни систем переписывающих правил, которые поддерживаются как исследовательскими группами, так и коммерческими организациями и используются в различных областях приложений. В Кибернетическом центре НАН Украины имеются большие традиции построения таких систем: одной из первых была система "Аналитик" [2], широко известна также система алгебраического программирования АПС [3, 4]. В этой статье представлена еще одна система символьных вычислений, основанная на парадигме переписывания TermWare и предназначенная для решения задач программной инженерии.

Существует некоторая типология систем переписывания, отражающая как эволюцию программного обеспечения (ПО), так и области применения. Традиционной сферой применения является обработка строк, в качестве примера можно привести Рефал [5]. Здесь фундаментальной математической моделью являются алгорифмы Маркова, а вычисления — это последовательное применение правил к начальной цепочке строк. Подобное ПО встроено в почти каждую существующую систему морфологического анализа. С развитием программной инженерии появилась потребность автоматического анализа и преобразования программ, где объектами преобразований являются синтаксические деревья конструкций языка программирования. Спектр применения таких систем — от автоматического генерирования документации до систем анализа ПО и автоматического перевода программ с одного языка программирования на другой. Автоматизация построения логических доказательств это одно из наиболее естественных применений символьных вычислений. Здесь в основе математической модели лежит логическая редукция: шаг вычислений соответствуют шагу логического вывода в некоторой системе аксиом в соответствии с определенными правилами вывода. Известными формальными моделями в этом направлении являются комбинаторная логика [6] и $\pi$-исчисление [7]. На этой основе построен целый класс языков программирования и разработаны классы промышленных применений, в частности системы проектирования логических микросхем, высокоуровневые языки программирования, экспертные системы.

Особый класс составляют универсальные системы переписывающих правил, который появился в последние годы как альтернатива систем логического программирования. Характерной его особенностью является наличие

правил представления либо преобразования объектов предметной области произвольной структуры в виде алгебраических термов, а также стратегий применения правил вывода, основанных на внелогических принципах, часто в виде императивного алгоритма последовательности поиска и применений правил.

TermWare принадлежит к последнему типу систем наряду с APS [3], Maude [8] и Stratego [9]. В данной статье описывается синтаксис и семантика языка, демонстрируется применение системы к задачам программирования динамических приложений, характеризуемых, с одной стороны, высокой степенью взаимодействий со средой, а с другой — высокой изменчивостью требований и необходимостью перепрограммирования таких систем.

### *1. Назначение и особенности системы TermWare*

Система TermWare нацелена на разработку высокодинамичных прикладных систем, к которым предъявляются повышенные требования как к встроенному интеллекту для обеспечения интерактивности разрабатываемых систем, так и к удешевлению разработки, сокращению сроков проектирования и улучшения характеристик повторного их использования (реинжиниринга) и сопровождаемости. Она отличается от большинства других систем этого класса как назначением и семантикой используемых средств, так и технологией их реализации. Язык TermWare не является универсальным языком программирования в том смысле, что он не предназначен для написания полнофункциональных программных систем. Это координационный язык-оболочка [10], предназначенный для написания предметно-ориентированных частей приложений, встраиваемых в прикладную систему для реализации функций взаимодействия этой системы с программным окружением.

Существенны также семантические особенности языка TermWare, отличающие его от других разработок в данной области. Традиционное декларативное программирование с помощью парадигмы переписывания основано на таком образце применения для написания программной системы, при котором строится формальная модель предметной области, в рамках этой формальной модели описываются основные понятия и структуры данных предметной области, переписывающие правила применяются для реализации классического алгоритма, преобразующего вход в выход, и операции ввода/вывода и взаимодействия с другими программами не представлены в декларативном описании в явном виде, а представляют собой побочные эффекты. Такой устоявшийся подход малоприменим в индустрии программирования по следующим причинам.

Во-первых, построение полного формального определения предметной области часто представляет собой слишком трудоемкий процесс для практического применения. Как правило, предметная область включает в себя много уже построенных элементов окружения, например реляционные базы данных (БД), объектные модели, протоколы межпрограммного взаимодействия. Для того чтобы использовать декларативное программирование, нужно представить все эти компоненты в одной формальной модели, придавая формальную семантику уже построенным императивным моделям. Это делает процесс построения формальной модели слишком сложным и дорогостоящим. Во-вторых, современные программные комплексы не могут быть прямо описаны как алгоритмы в классическом смысле, т.е. как преобразователи входной информации в выходную. Очень часто программная система представляет собой некоторое поведение, включающее в себя взаимодействие с внешней средой. И в-третьих, операции ввода/вывода, обычно представленные в декларативном программировании как побочные эффекты, имеют слишком большое значение чтобы быть проигнорирован-

ными в формальной модели вычислений. Отсутствие ввода/вывода в модели означает невозможность анализировать и преобразовывать механизмы взаимодействия программ с окружением, что делает традиционные системы логического программирования маломощными и непригодными для важных классов приложений.

В TermWare предложен новый формализм для декларативного описания предметной области программирования распределенных программных комплексов — термальные системы. Он включает в себя описание в виде термов как функциональности частей программных комплексов, подобно традиционным системам, так и их взаимодействия со внешней средой. Последнее представляет собой динамическую базу фактов во внутренней логике системы, например состояние внешней среды в системе управления либо перечень текущих бизнес-процессов в системе управления потоками работ. Таким образом, вместо сокрытия императивного стиля программирования в побочных эффектах в TermWare происходит погружение императивных операций в логику декларативной программы. Термальная система является открытой в том смысле, что она интерактивна и ее поведение определяется не только программным кодом, но и состоянием среды.

Технологические отличия TermWare связаны с тем, что в современной индустрии разработки ПО уже не рассматривается отдельно от таких элементов технологической платформы программного окружения, как базы данных, системы мониторинга транзакций и унаследованные библиотеки. Однако большинство существующих систем предоставляют свое окружение, подразумевая, что конечный пользователь работает в оболочке системы символьного программирования.

TermWare оформлена как библиотека языка Java, встраиваемая в программное приложение. Для разработчика система термов выглядит как совокупность экземпляров класса ITermSystem, с возможностью управления набором правил и редуцирования термов. База знаний также может содержать императивные элементы, описанные как Java-классы. Таким образом, имеется возможность использовать декларативную модель программирования в программных комплексах, основанных на Java-платформе в тех случаях, когда это необходимо. Разработчик может использовать термальную систему как программный агент, встраиваемый в общую инфраструктуру программной системы.

Встраивание систем обработки правил в Java-платформу в последнее время быстро развивается и даже стандартизируется. Существует предложение к стандарту языка Java (jsr94 [11]) и несколько систем, встраиваемых в Java по тому же принципу, что и TermWare. Среди них можно выделить Jess [12] (порт CLISP) и ASDF [12], представляющие собой системы переписывающих правил, предназначенные для построения экспертных систем и преобразований деревьев синтаксического разбора соответственно. Поэтому разработка TermWare своевременна и необходима.

### *2. Формальная модель*

Формальная модель TermWare построена как обычная алгебра термов, где объектами языка являются термы, т.е. выражения вида $f(x_1, ..., x_n)$, переменные и элементарные типы данных, подобно другим языкам функционального программирования. Единственное отличие от стандартного подхода состоит в том, что конструктор упорядоченного множества является выделенным термом и операции подстановки на этих термах сохраняют порядок. Вычисления определяются наборами переписывающих правил вместе со стратегией применений по такой же схеме, как и в APS [3] или Stratego[9].

Основное отличие TermWare от языков алгебраического программирования (таких, как OBJ и Maude [8]) состоит в том, что модель вычислений в TermWare определяется в терминах

поведения программ, т.е. входных воздействий и реакций, а не в терминах входа и выхода, как это происходит при традиционном подходе. Иными словами, в традиционных системах переписывающих правил вычисления являются преобразованиями входного терма в некоторую каноническую форму с помощью набора переписывающихся правил. Тем самым язык определяет функциональные зависимости вида $f(x) = y$, где $x$ — вход алгоритма; $y$ — выход в терминах классической теории алгоритмов. В TermWare набор переписывающих правил определяет не вычисления целиком, а один шаг вычислений, в терминах теории динамических систем [14] — вида $f(x, s) \to \to (s', y)$, где $x$ — входной сигнал; $y$ — выходной, $s$ — внутреннее состояние.

В этом разделе представлена относительно простая синтаксическая формальная модель и показано, как на ее основе можно организовать работу с более общими структурами.

Считаем, что *алфавит* языка включает (везде, если это не оговорено особо, нижние индексы принадлежат перечислимому множеству):

- множество констант $c_i$, принадлежащих одному из примитивных типов (INT, BOOL, STRING, ATOM); с каждой константой связано значение на множестве соответствующего примитивного типа;
- INT — множество целых чисел, представляемое константами 1, 2, ....;
- BOOL — множество булевых значений, представляемое константами **true** и **false**;
- STRING — множество строк в алфавите Unicode;
- ATOM — множество атомарных неинтерпретируемых значений; среди атомов выделим NIL, играющий специальную роль.

Кроме того, язык включает также множество терминальных символов $t_1, ..., t_N$, множество пропозициональных переменных $x_i$, скобки '**(**' и '**)**' и знак зяпятой ',' множество функциональных символов $f_i$, символ среды $S$, множество $\varphi$ модельных символов операций чтения информации из среды, а также множество $\delta$ модельных символов воздействия на среду.

*Термы языка* строятся по следующей схеме. Сначала определяется множество конкретных термов $T_c$:

- $c_i \in T_c$;
- если $x_1 \in T_c, ..., x_N \in Tc$, то и $f$ есть функциональный символ, $f(x_1, ..., x_N) \in T_c$,

а также множество подстановочных термов $T_v$:

- $x \in T_c \Rightarrow x \in T_v$;
- $x_i \in T_v$;
- $x_1 \in T_v, ..., x_N \in T_v \Rightarrow f(x_1, ..., x_N) \in T_v$.

Терм, принадлежащий разности множеств $T_v \backslash T_c$, назовем термом со свободными переменными, а для каждого терма t множество входящих в него нетерминальных символов — множеством свободных переменных этого терма и обозначим $v(t)$.

Далее на системе термов определим следующие синтаксические функции и их свойства:

- функция *typename0*: $T_v \to$ STRING возвращает синтаксический тип терма; это может быть:
  - наименование одного из типов "BOOL", "INT", "ATOM" или "STRING" для примитивных термов;
  - "*x*" для пропозициональных переменных;
  - "T" для сложных термов;
- функция *name*: $T_v \to$ STRING возвращает имя головного символа терма;
- функция *arity*: $T_v \to$ STRING возвращает арность символа терма;
- функция *subterm*: $T_v \times$ INT $\to T_v$ вычисляет индекс подтерма начиная с нуля:
  - если $t \in T_v$, $i < arity(t)$, то *subterm*($t$) возвращает $i$-й субтерм $t$;
  - если $i > arity(t)$, то *subterm*($t$) = = NIL;
- функция *equal*: $T_v \times T_v \to$ BOOL возвращает истинное значение в случае тождественного совпадения **x = y**;

• функция *less*: $T_v \times T_v \rightarrow$ BOOL определяется такими случаями:

- если $x \neq$ NIL, то *less*(NIL, *x*);
- если *typename0*(*x*) < *typename0*(*y*), то *less*(NIL , *x*);
- если *x*, *y* — константы, то *less*(*x*, *y*) представляет собой естественное упорядочение;
- если *x*, *y* — атомы, то *less*(*x*, *y*) представляет собой лексикографическое упорядочение их имен;
- если *x*, *y* — пропозициональные переменные, то имеет место $\neg less(x, y) \wedge \neg less(y, x)$;
- если *x*, *y* — составные термы и *arity*(*x*) < *arity*(*y*), то *less*(*x*, *y*) = = **true**;
- если *x*, *y* — составные термы и *arity*(*x*) = *arity*(*y*) $\wedge$ *less*(*name*(*x*), *name*(*y*)), то *less*(*x*, *y*) = **true**;
- если *x*, *y* — составные термы и *arity*(*x*) = *arity*(*y*) $\wedge$ *name*(*x*) = = *name*(*y*), то *less*(*x*, *y*) представляет собой лексикографический порядок кортежей элементов.

Таким образом, множество термов (с точностью до перенумерации пропозициональных символов) является полностью упорядоченным. Это позволяет выделить конструктор упорядоченного множества *set*($x_1$, …, $x_N$), т. е. терм, обозначающий упорядоченное множество из *N* элементов. Отсюда можно полагать, что если *t* = *set*($x_1$, …, $x_N$), *subterm*(*i*, *t*) = *x*, *subterm*(*j*, *t*) = *y*, *o* < *i* < *j* < < *arity*(*t*), то *less*(*x*, *y*).

Теперь $\nu(t)$ можно определить уже на синтаксическом уровне: $\nu(c_i)$ = NIL, $\nu(x_i) = \{x_i\}$, $\nu(f(t_1, ..., t_N)) = \bigcup_{i=1}^{n} \nu(x_i)$.

Выражение *subst*(*t*, *x*, *s*) есть подстановка *s* в *t* вместо свободной переменной *x*. Будем также употреблять для подстановки обозначение *t*[*x*, *s*]. Терм *bound_unify*($t_1$, $t_2$) означает операцию унификации $t_1$ и $t_2$ со множеством связанных пропозициональных переменных, входящих в $t_1$ и $t_2$, *free_unify*($t_1$, $t_2$) — операцию унификации $t_1$ и $t_2$, при которой свободные переменные $t_1$ и $t_2$ предварительно переименовываются таким образом, чтобы $\nu(t_1) \cap \nu(t_2) = \varnothing$.

Семантика правила переписывания без взаимодействия со средой определяется в виде *apply*(*t*, $x \rightarrow y$) = *subst*(*y*, *free_unify*(*x*,*t*)), где $x \rightarrow y$ — правило переписывания терма *x* в *y*. Любую среду *S* можно представить как БД фактов, определив акты взаимодействия терма со средой с помощью функций $\varphi$: $S \times T_c \rightarrow$ $\rightarrow T_c$ (получение информации из *S*) и $\delta$: $S \times T_c \rightarrow S$ (передача информации в *S*).

Наконец, определим основной объект нашего рассмотрения — термальную систему. *Термальная система* $\tau$ есть пара $\tau = (S, R)$, где *S* — модель среды, а *R* — набор переписывающих правил, представляющий взаимодействие со средой. Переписывающее правило из *R* — это терм с четырьмя аргументами вида *rule*(*x*, *in*, *y*, *out*), которое для удобства обозначим $x[in] \rightarrow$ $\rightarrow y//out$. Такое выражение можно прочитать как "переписать *x* при условии *in* в *y*, применив к среде операцию *out*". С использованием введенных выше обозначений схему выполнения правила переписывания можно записать в виде

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

где $z = subst(y, free\_unify(x, t, \varphi(S, in)))$.

Наконец, последний шаг — добавим в модель понятие имени системы и множества систем. Имя системы — это атом либо специальный терм *_name*($x_1$, ..., $x_n$), где $x_1$, …, $x_n$ — атомы. Считаем, что интерпретация терма *_name*($x_1$, ..., $x_n$) определяется функцией именования $\phi$: $T_c \rightarrow \Phi$, которая возвращает определение термальной системы в некотором (иерархическом) пространстве имен Φ. Саму терминальную систему можно определить как терм вида *System*(*name*, *ruleset*, *facts*, *strategy*), где *name* — имя системы; *ruleset* — множество правил переписывания со взаимодействием; *facts* — база данных фактов среды; *strategy* — стратегия

применения переписывающих правил, определяющая последовательность применения правил. Функция редуцирования терма в системе выражается в виде терма *reduce*(*s*, *t*), где *t* — терм, редуцируемый в системе с именем *s*.

Таким образом, TermWare — это непосредственное отображение логической модели взаимодействий в язык программирования, обладающее аналогами современных средств поддержки наследования и иерархической системы имен. Ниже покажем, что такие системы более естественно отражают типичные проблемы программирования на уровне архитектуре и пригодны для описания не только вычислительных алгоритмов, но и поведения программных комплексов, состоящих из нескольких взаимодействующих подсистем.

### 3. Реализация языка

Формальная модель, изложенная выше, отображается в языке программирования с помощью следующих средств и соглашений.

*Терминальные символы*:

- константы — это числа, логические значения и строки Unicode с обычной семантикой;
- пропозициональные переменные начинаются знаком доллара и продолжаются последовательностью букв или цифр, например **$x10**, **$saved_args**;
- идентификаторы — последовательности из букв и цифр;
- конструкторы термов имеют традиционный функциональный вид $f(x_1, ..., x_n)$;
- в языке используются знаки →, [| |], &, |, &&, ||, +, -, *, /, //, ^, <<, >>, <, <=, >, =>, ==, !=, !;
- комментарии — это строки, начинающиеся знаком #.

В целях удобочитаемости в TermWare определен синтаксис, подобный языку C. Выражение $x$ [| $y$| ] → → $z$ // $v$ является сокращением для *rule*($x$, $y$, $z$, $v$) и обозначает переписывающее правило с входным образцом $x$, условием $y$, выходным образцом $z$ и действием $v$. В случае, если один или несколько элементов отсутствуют, они могут быть опущены. Далее установлено соответствие для других выражений языка и их сокращений: $\{x \mid y\}$ — является сокращением для *set_pattern*($x$, $y$); $\{x_1, ..., x_N\}$ — для *set*($x_1$, ..., $x_N$); $x$ *in* $y$ — для *_in*($x$, $y$); $[x_1, ...., x_N]$ — для *cons*($x_1$, *cons*(... *cons*($x_N$)...); $x$ ? $y$: $z$ — для *ifelse*($x$, $y$, $z$); $x + y$ — для *plus*($x$, $y$) и аналогично для всех инфиксных операторов языка C.

Переписывающие правила объединяются в наборы с помощью терма *ruleset*, например, набор правил, описывающий булеву алгебру, описывается так:

```
ruleset(
  $x & ($x => z) -> $z,
  not($x & $y) -> not($x) | not($y),
  not($x | $y) -> not($x) & not($y),
  $x & ($y | $z) -> ($x & $y) | ($x & $z),
  not(not($x)) -> $x,
).
```

Термальная система описывается с помощью выражения *System*($x$, $y$, $z$, $v$), например:

```
System(BooleanAlgebra, default,
  ruleset(
  $x & ($x => z) -> $z,
  not($x & $y) -> not($x) | not($y),
  not($x | $y) -> not($x) & not($y),
  $x & ($y | $z) -> ($x & $y) | ($x & $z),
  not(not($x)) -> $x,
  ),
  FirstTop);
```

Здесь первый подтерм — это имя системы, второй — имя базы данных фактов, после чего следуют набор правил переписывания и стратегия применения. В приведенной выше схеме использована пустая баз данных и стратегия переписывания *FirstTop*.

База данных фактов и стратегия переписывания определяются на уровне процедурного языка в зависимости от предметной области. Как это происходит, покажем в следующем разделе,

а пока обратим внимание еще на две особенности языка. Система имен в TermWare представляет собой иерархию с целью структуризации информации: подсистемы можно организовывать в области (domains) так же, как в Java классы объединять в пакеты (packages). Термальные системы могут повторно использовать информацию с помощью механизма импорта правил аналогично механизму наследования в объектно-ориентированных языках.

*System*(*BooleanLogic*, *defaut*,
*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. Встраивание в приложения*

Предыдущие примеры систем несколько ограничены и не используют основные элементы новизны динамической модели — БД фактов и действия. Отчасти это вызвано тем, что TermWare предназначается для встраивания в приложения, а БД фактов и набор действий зависит от конкретных особенностей приложений. Встраивание элементов логического вывода в приложения и проектирование набора БД фактов и действий при использовании TermWare обычно осуществляются по следующей стратегии:

- в программе выделяется подсистема, где уместно использовать декларативные правила;
- определяется взаимодействие с остальной частью комплекса в виде набора функций получения информации (условий) и передачи информации (действий);
- полученный набор взаимодействий представляется в виде Java-класса (который должен реализовать интерфейс IFacts);
- определяется стратегия применения правил: при этом либо выбирается одна из существующих, либо программируется реализация интерфейса IStrategy;
- происходит встраивание TermWare в приложение либо с помощью прямого вызова конструктора объекта ItermSystem, либо путем определения на языке TermWare соответствия между именами БД фактов (стратегий) и реализующими их классами Java.

Общая схема создания БД фактов путем реализации интерфейс Ifacts имеет следующий вид:

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

Когда при интерпретации правил переписывания встречается запрос информации из внешней среды, TermWare вызывает метод `check` ассоциированной базы данных фактов; если встречается действие, вызывается метод `set`. Существует вспомогательный класс DefaultFacts, который с помощью рефлективного прикладного интерфейса Java самостоятельно осуществляет разбор аргументов.

Собственно термы представляются разработчику как объекты, реализующие интерфейс ITerm. В сигнатуре ITerm определены операции определения типа, доступа к типизированным значениям, унификации, эквивалентности и лексикографического сравнения. Объем статьи не позволяет привести здесь эту сигнатуру полностью. Для демонстрации возможностей программирования приведем в качестве примера описание широко извест-

ной игры Конвея "жизнь" [15], используя БД фактов для представления информации о поле. Соответствующая термальная система выглядит следующим образом:

```
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)
|] -> $T // putCell($x,$y),
{ l($i,$j) |$T } [| n($i,$j) == 3
|] -> $T // putCell($x,$y),
{ l($i,$j) |$T } [| n($i,$j) <2 ||
n($i,$j)>3 |] -> $T // remove-
Cell($x,$y),
{ } -> $T // showGeneration() &&
createNewTestSet($T)
),
FirstTop)
);
```

Как видно, из БД запрашиваются следующие факты и условия: 1) *existsCell*$(i, j)$ — существует ли клетка по адресу $i, j$ и 2) $n(i, j)$ — количество соседей клетки. К действиям относятся:

- *putCell*$(i, j)$ — поместить клетку в ячейку $i, j$;
- *removeCell*$(i, j)$ — удалить клетку;
- *showGeneration*() — показать на экране следующее поколение;
- *createNewTestSet*($\$T$) — сгенерировать новое тестовое множество клеток.

В сокращенном виде соответствующий класс БД выглядит следующим образом:

```
class LifeFieldFacts exstends
      DefaultFacts
{
 private int nX_;
 private int nY_;
 private boolean[][] field_;
 private boolean[][] nextField_;
 private Canvas drowing_;
 public ITerm existsCell(ITerm tx,
   ITerm ty) throws TermWare-
   Exception
 {
 int x = tx.getInt();
 int y = ty.getInt();
 return ITermFactory.create-
   Boolean(field_[x%nX][y%nY]);
 }

public void putCell(ITerm tx,
 ITerm ty) throws TermWareException
 {
 int x = tx.getInt();
 int y = ty.getInt();
 nextField_[x%nX][y%nY] = true;
 }
public void generateNewTest-
 Set(ITerm t) throws
 TermWareException
 {
if (!t.isX()) {
 throw new TermWareException (”ar-
  gument of generateNewTestSet mus
  be propositional variable”);
 }
 SetTerm retval = ITermFac-
  tory.createSet();
 field_ = nextField_;
 nextField_ = createNewField();
 for(int i=0; i<nX; ++i) {
 for(int j=0; j<nY; ++j) {
 if (field_[i][j]) {
 retval.insert(createComplex-
  Term2(”l”, (i-1)%nX,(j-1)%nY));
 retval.insert(createComplex-
  Term2(”l”,(i-1)%nX,(j )%nY));
 retval.insert(createComplex-
  Term2(”l”,(i-1) %nX,(j+1)%nY));
 retval.insert(createComplex-
  Term2(”l”,(i ) %nX,(j-1 )%nY));
 retval.insert(createComplex-
  Term2(”l”,(i ) %nX,(j )%nY));
 retval.insert(createComplex-
  Term2(”l”,(i ) %nX,(j+1)%nY));
```

```
 retval.insert(createComplex-
  Term2("l",(i+1) %nX,(j-1)%nY));
 retval.insert(createComplex-
  Term2("l",(i+1) %nX,(j )%nY));
 retval.insert(createComplex-
  Term2("l",(i+1) %nX,(j+1)%nY));
 }
 }
 }
 // fill the value of the proposi-
    tional variable.
 tx.subst(tx.minFv(),retval);
 }
}
```

Обратим внимание на подстановку вместо пропозициональной переменной множества в действии generateNewTestSet. Этот пример иллюстрирует передачу информации из императивной части программы в ее логическую часть.

### Выводы

Таким образом, разработанная система дает возможность проектировать интеллектуальную составляющую приложений с учетом существующих технологических платформ и естественно сочетать императивный и логический стили программирования с помощью общей объектно-ориентированной среды. Это облегчает как проектирование приложений, так и модификацию бизнес-логики, которую можно задавать независимо от внешней среды.

В настоящее время TermWare применяется в системах организации потока работ, где с помощью логических правил описываются бизнес-процессы, а в качестве БД фактов выступает взаимодействие с исполнителями и данными о текущем состоянии бизнес-процессов [16]. Еще одна область применения — это анализ структур программ, позволяющая выделять шаблоны проектирования [17] и проводить анализ корректности кода относительно известного множества типичных погрешностей проектирования [18]. Особенный интерес представляет семантический анализ моделей UML[19] в распределенных программных комплексах [20], [21].

Система TermWare реализована в ООО Град-Софт. Более детальное описание и некоторые приложения доступны на Web-странице: http://-www.gradsoft.kiev.ua


1. *Dershowitz N., Jouannaud J.-P.* Rewrite Systems // Handbook of Theoretical Computer Science. Vol. B: Formal Models and Semantics / Ed. J. van Leeuwen. — Boston: Elsevier and MIT Press, 1990. — P. 243–320.
2. *АНАЛИТИК–74* / В.М. Глушков, Т.А. Гринченко, А.А. Дородницына и др. // Кибернетика. — 1978. — № 5. — С. 114–147.
3. *Капитонова Ю.В., Летичевский А.А.* Методы и средства алгебраического программирования // Кибернетика и системный анализ. — 1993. — № 3. — С. 7–12.
4. *Parallel* Symbolic Simulation Using the Algebraic Programming System / A.A. Letichevsky, J.V. Kapitonova, A.E. Doroshenko, V.A. Volkov // Algebraic Engineering: Proc. Intern. Conf. on Semigroups and Algebraic Engineering. — Singapore: World Scientific, 1999. — P. 350–360.
5. *Refal-5:* Programming Guide and Reference Manual. — Holyoke MA: New England Publishing Co., 1989. — 342 p.
6. *Semantics* of Constraint Logic Programs / J. Jaffar, M.J. Maher, K.G. Marriott, P.J. Stuckey // J. of Logic Programming, — 1998. — **37**. — P. 1–46.
7. *Milner R.* Communicating and Mobile Systems: the π-calculus. — Cambridge: University Press, 1999. — May. — 174 p.
8. *Winkler T.* Programming in OBJ and Maude // Functional Programming, Concurrency, Simulation and Automated Reasoning / Ed. P. Lauer. — LNCS. — 1993. — **693**. — P. 229–277.
9. *Visser E.* Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5 // Rewriting Techniques and Applications (RTA'01) / Ed. A. Middeldorp, Ibid. — 2001. — **2051**. — P. 357–361.
10. *Gelernter D. Carriero N.* Coordination Languages and Their Significance // Comm. ACM. — 1992. — **35**, № 2. — P. 97–107.
11. *JSR-000094 JavaTM Rule Engine API.* — http://www.jcp.org/aboutJava/communityprocess/ review/jsr094/.
12. *Jess* — the expert system shell for the Java Platform. — http://herzberg.ca.sandia.gov/-jess/ri_overview.shtm.
13. *Efficient annotated terms* / M.G.J. van den Brand, H. de Jong, P. Klint, P. Olivier // Software, Practice & Experience. — 2000. — **30**(3). — P. 259–291.
14. *Капитонова Ю.В., Летичевский А.А.* Математическая теория проектирования вычис-

лительных систем. — М.: Наука, 1988. — 296 с.
15. *Эндрюс Г.Р.* Основы многопоточного, параллельного и распределенного программирования. — М.: Вильямс, 2003. — 506 с.
16. *Shevchenko R., Doroshenko A.* Managing Business Logic with Symbolic Computation // Information Systems Technology and Applications: Proc. 2-nd Intern. Conf. ISTA'2003, June 19—21, 2003, Kharkiv, Ukraine. — Kharkiv, 2003. — P. 143—152.
17. *Design Patterns*: Elements of Reusable Object Oriented Software / E. Gamma, R. Helm R. Jonhson, J. Vlissides. — Reading, MA: Addison-Wesley, 1995. —329 p.
18. *Liang, M., Harrold J.* Light-Weight Context Recovery for Efficient and Accurate Program Analysis // ICSE 2000: Proc. 22-nd Intern. Conf. Software Engineering, June 4—11, 2000, Limerick, Ireland. — New York: ACM Press, 2000. — P. 366—406.
19. *UML 2.0* Specification. — http://www.omg.-org/technology/uml/.
20. *Shevchenko R., Doroshenko A.* Evolution of CORBA Framework: An Experience Study, Case studies of CSMR 2002, Workshop Proc. 6-th European Conf. on Software Maintenance and Reengineering, March 11—13, 2002, Budapest, Hungary. — Budapest, 2002. — P. 3—9.
21. *Shevchenko R., Doroshenko A.* A Method of Mediators for Building Web Interfaces of CORBA Distributed Enterprise Applications // Information Systems Technology and its Applications: Proc. Intern. Conf. ISTA-2001, June 13—15, 2001, Kharkiv, Ukraine. — Kharkiv, 2001. — 4. — P. 53—63.

*Получено 23.11.03*

## *Об авторах*

*Дорошенко Анатолий Ефимович,*
доктор физ.-мат.наук, профессор,
заместитель директора по научной работе

*Шевченко Руслан Сергеевич,*
аспирант

*Место работы авторов*
Институт программных систем НАН Украины,
просп. Академика Глушкова, 40,
Киев-187, 03680, Украина
Тел. 266 1538
E-mail: dor@isofts.kiev.ua,
Ruslan@Shevchenko.kiev.ua