УДК 519.686.2

А.А. Летичевский (мл.), М.К. Мороховец, В.С. Песчаненко

Система доказательного программирования

Описаны методы доказательства правильности программ в системе инсерционного моделирования, ее архитектура и функциональные возможности, даны основные сведения о нем. Рассмотрена инсерционная машина метода Флойда, методы проверки выполнимости формул и их использование при доказательстве правильности программ.

The methods are described of proving the programs correctness within the Insertion Modeling System. The architecture of the system and its functional possibilities are described, the basic notions of insertion modeling are presented. An insertion machine for Floyd's method is presented, the methods of the satisfiability checking of the formulae, and their usage for proving the program correctness are described.

Описано методи доведення правильності програм у системі інсерційного моделювання, її архітектура та функціональні можливості, подано основні відомості про нього. Розглянуто інсерційну машину методу Флойда, методи перевірки виконуваності формул та їх використання у доведенні правильності програм.

Введение. Система доказательного программирования – это новая и современная система для поддержания высокого уровня подготовки квалифицированных специалистов в области программирования, созданная на базе систем инсерционного моделирования *IMS* [1], алгебраического программирования *APS* [2], разработанных на рубеже прошлого и нынешнего веков в Институте кибернетики им. В.М. Глушкова НАН Украины с участием авторского коллектива Херсонского государственного университета под руководством академика НАН Украины профессора А.А. Летичевского.

Украина имеет достаточно высокий потенциал по подготовке специалистов в области информационных технологий. Эти специалисты ценятся во всем мире, в первую очередь, благодаря созданной в нашей стране системе фундаментальной подготовки специалистов в области информационных технологий и, прежде всего, программистов. Однако их подготовка отстает от мирового уровня, поскольку сейчас очень важно не только писать программы, но и верифицировать их, повышать их надежность и эффективность. Для этого, например, компания *Microsoft* создает новую систему *Spec#* [3], к разработке которой подключаются специалисты наивысшей квалификации: математики, инженеры и т.д.

В последнее время в мире становится все более популярной идея использования в наукоемких производствах программного обеспечения, корректность которого доказана. Однако на рынках программного обеспечения Украины и других стран ощущается недостаток программных продуктов, способных эффективно проверять корректность программ. Это обусловлено тем фактом, что для создания таких программных продуктов необходимы очень квалифицированные специалисты как в области технологий, так и в области фундаментальных наук. Кроме того, осваивая учебные дисциплины профильных специальностей, студенты не имеют возможности научиться *доказательному* программированию с использованием современного программного обеспечения. Использование такой системы в высших учебных заведениях Украины может существенно улучшить качество фундаментальной подготовки наших программистов.

Таким образом, создание системы доказательного программирования актуально. Данная статья посвящена методам доказательства правильности программ в системе инсерционного моделирования IMS.

Состав системы доказательного программирования

Система доказательного программирования в идеале должна состоять из следующих подсистем:

– предварительной подготовки;
– верификации;
– диалога;
– накопления знаний о верификации.

Функции подсистемы предварительной подготовки: подготовка аннотированной программы (обеспечивается среда для создания аннотированной программы, в частности, предос-

тавляется набор примеров–образцов аннотированных программ разных видов); настройка на предметную область (по виду верифицируемой программы – последовательная, параллельная – и по виду утверждений в аннотациях, подлежащих проверке – утверждения о целых, действительных числах, структурированных объектах – определяется набор средств, необходимых для верификации, и проводится настройка подсистемы верификации).

Функции подсистемы верификации: доказательство утверждений–аннотаций (используются средства самой системы, а также «внешние» средства доказательства и проверки выполнимости утверждений в различных предметных областях).

Функции подсистемы диалога: обеспечение возможности следить за процессом верификации, давать подсказки; подготовка отчета о верификации с той или иной степенью детализации; выделение ошибочных мест в программе; рекомендации по исправлению программы (если в результате верификации программа не признана правильной).

Функции подсистемы накопления знаний о верификации: хранение примеров аннотированных программ различных видов; хранение общих рекомендаций по составлению аннотаций; хранение прочей справочной информации (например, синтаксических правил записи аннотаций).

В статье обсуждается первая версия системы доказательного программирования, содержащая подсистему верификации и некоторые диалоговые средства. В будущем предполагается развитие системы.

Инсерционное моделирование

Инсерционное моделирование – это подход к моделированию сложных распределенных систем, основанный на теории взаимодействия агентов и сред, математические основы которой были изложены в [4]. В течение последнего десятилетия инсерционное моделирование применялось к проверке требований к программному обеспечению [5]. Теория взаимодействия агентов и сред предложена в качестве альтернативы известным теориям взаимодействия, таких как *CCS* Милнера [6] и π-исчисление [7], *CSP* Хоара [8], мобильных амбиентов Карделли [9] и т. д. Идея декомпозиции системы и представления ее в виде композиции среды и агентов, погруженных в эту среду, неявно присутствует во всех теориях взаимодействия. Эти идеи приняты в качестве основы для системы инсерционного моделирования *IMS* [10], разработанной как расширение системы алгебраического программирования *APS* [2]. Первую версию системы *IMS* и некоторые простые примеры ее использования можно найти в [11]. Одному из применений системы *IMS* посвящена эта статья – доказательству правильности программ в системе инсерционного моделирования.

Инсерционная машина системы

Система инсерционного моделирования – это среда для разработки инсерционных машин и проведения экспериментов с ними. Понятие инсерционной машины впервые было введено в [10]. Она использовалась в качестве инструмента для программирования со специальным классом функций погружения. Позднее это понятие было расширено для более широкой области применения (различные уровни абстракции и многоуровневые структуры).

Далее будут использованы такие обозначения. Результат погружения агента, находящегося в состоянии u, в среду, находящуюся в состоянии E, обозначается $E[u]$. Состояния транзиционных систем рассматриваются с точностью до отношения бисимуляции [10].

Общая архитектура инсерционной машины представлена на рис. 1.

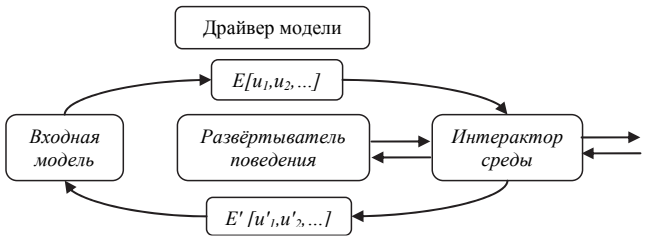

Рис. 1. Общая архитектура инсерционной машины

Основным компонентом инсерционной машины есть «Драйвер модели» – компонент, контролирующий движение машины по дереву поведения модели. Состояние модели представлено в виде текста на входном языке инсерци-

онной машины и рассматривается как алгебраическое выражение. Входной язык включает рекурсивные определения поведений агента, определение функции погружения и композиции для состояний среды. Состояние системы должно быть приведено к виду $E[u_1, u_2, ...]$. Это осуществляется с помощью модуля, который называется «Развертыватель поведения». Чтобы осуществить движение, состояние среды должно быть приведено к нормальной форме $\sum_{i \in I} a_i \cdot E_i + \varepsilon$, где a_i – действия, E_i – среда, ε – завершающая константа. Это осуществляется с помощью модуля «Интерактор среды». Он выполняет вызов функции погружения, а если это необходимо – «Развертывателя поведения» агентов. Если бесконечное множество I индексов в нормальной форме разрешено, тогда используется слабая нормальная форма $a.F+G$, где G – произвольное выражение входного языка.

Входной язык инсерционной машины используется для описания свойств модели и ее поведения. Это описание состоит из следующих частей: описания среды, описания поведения (в том числе поведениz среды и поведения агентов) и функции погружения. Описание поведения имеет следующий простой синтаксис:

<behavior>::= *Delta | bot | 0 | < action > | <action> . <behavior> |*

<behavior> + <behavior>|

<environment state>[*<list of named agent behaviors separated by ,>*]|

<functional expression>

<named agent behavior>::=*<agent name>*:*<behavior>*

Язык алгебры поведений (константы завершения, префиксинг и недетерминированный выбор) распространяется на функциональные выражения и явное представление функции погружения. Синтаксис и семантика действий, состояния среды и функциональные выражения определены в описании среды. При этом используется следующий синтаксис:

<environment_descr>::=*obj*([*<tp>*;]*<attributes_def>*; *<interpreted_def>*;*< axioms_def>*;*< init_def>*);

<tp>::=*types:obj*(*<type_def>*[,*<types>*]);

<type_def>::=*<type_name>*:(*<type_consts>*)

<type_name>::=*<identifier>*

<type_consts>::=*<identifier>*[,*<type_consts>*]

<attributes_def>::=*attributes:obj*(*<idetifier>*:*<type>*[, *<attributes_def>*])

<type>::=*<simple type>* |*<functional type>*|*array* (*<integer const>*) *of <simple type>* | *list of <simple type>*

<simple type>::=*bool*|*int*|*real*|*<type_name>*|*symb*

<functional type>::=(*<functional params>*)–>*<simple type>*

<functional params>::=*<simple type>*[,*<functional params>*]

<interpreted_def>::=*interpreted:*(*<funcs names list>*)

<funcs names list>::=*<identifier>*[,*<funcs names list>*]

<axioms_def>::= *axioms: Forall <list of forall vars> <expr>*

<list of forall vars>::=*<list of vars with types>*[,*<list of vars with types>*]

<list of vars with types>::=*<identifier>*:*<simple type>*

<init_def>::= *initial:<environment state>*[*<behavior>*]

<lists values>::=*<identifier>*:(*<list of consts>*)[,*<lists values>*]

<list of consts>::=*<const>*[,*<list of consts>*]

<list of assigment>::=*<assignment var>*:=*<expr>* [,*<list of assigment>*]

<assignment var>::=*<identifier>*|*<identifier>*(*<expr>*)

В качестве *<identifier>* в IMS используются атомы, *<expr>* – выражения, их точное определение зависит от пользователя (от отметок, определенных в модуле), *<rewriting rule>* – описание системы переписывающих правил. Более подробно о синтаксисе этих структур можно узнать из [2]. Поле *<interpreted_def>* используется для того, чтобы передать типы функциональных атрибутных выражений внешним модулям проверки выполнимости формул. Поле *<axioms_def>* используется для доказательства выполнимости вместе с аксиомами. Более детально о том, что происходит с данными из этих полей, сказано в следующем разделе.

Исследование процессов проверки корректности программ имеет долгую историю, которая началась с работ Хоара [12] и Флойда [13]. Условия корректности программы имеют следующий вид $\alpha \to <P> \beta$ или $\alpha \to [P] \beta$. Формулы α и β (пред- и постусловие) есть формулами языка спецификаций (например, языка исчисления предикатов первого порядка), P – спецификация программы (предполагается, что дано начальное состояние памяти). Первая формула означает, что если условие α истинно и программа P завершается, то условие β тоже истинно на конечном состоянии памяти. Вторая формула выражает полную корректность и означает, что если условие α истинно, то программа завершается и условие β тоже истинно.

В текущей реализации системы мы рассматриваем сокращенный вариант аналитической инсерционной машины (она предназначена для анализа модели, проверки ее свойств и т.д.), основанной на аннотировании программ методом Флойда. На вход этой машины подаются аннотированные программы, записанные с использованием четырех операторов: присваивания, условного оператора, оператора *go to* и оператора s*top*. Аннотации в программе бывают двух видов: предположения и утверждения. Синтаксис аннотированной программы следующий:

<input model>::=*make_model*(*<operator>*[;*<input model>*]);*verify <label name>*

<operator>::=[*<label name>*:]*<operator descr>*

<operator descr>::=*<assignment descr>*|*<if descr>*|*go to <label name>*| *stop*|

|*<assumption descr>*|*<assertion descr>*

<assignment descr>::=*<identifier>*:=*<expr>*

<if descr>::=*<expr>*–>*<list of operators>*[*else <list of operators>*]

<list of operators>::=*<operator decr>*[;*<list of operators>*]

<assumption descr>::=*assumption:<expr>*

<assertion descr>::=*assertion:<expr>*

<label name>::=*<identifier>*

Первое поле *<assumption descr>* определяет предусловие программы, второе поле *<assertion descr>* – ее постусловие. Сама программа рассматривается как атрибутная транзиционная система. Контрольные точки – состояния системы, которые отмечаются аннотациями. Набор действий такой системы состоит из операторов присваивания, утверждений *stop* и *ask(u)*, где *u* – условие. Разметка атрибутов – это разметка программы, помечающая некоторые из ее состояний. Контрольные точки разделяют аннотированную программу на блоки. Каждый блок начинается с размеченного утверждения. Внутри блока отсутствуют размеченные операторы, а его окончательное утверждение – оператор *go to*. Блоки собираются в структуру данных под названием *model*, а доступ к единице, помеченной разметкой *L*, определяется выражением *model.L*. Тело *make_model* содержит текст аннотированной программы, а инструкция *verify <label name>* означает точку входа в программу.

Входные данные системы представляются в соответствии со следующим правилом:

<input data>::=(*program environment: <enviroment_ descr>; <input model>*)

Функция погружения инсерционной машины позволяет осуществить последовательное и параллельное погружение, где состояния среды – формулы исчисления предикатов первого порядка. Отношения транзиционной системы представлены следующим образом:

$$\varphi(p) \xrightarrow{ask\ u} (\varphi \wedge [p]u),\ Sat(\varphi \wedge [p]u)$$
$$\varphi(p) \xrightarrow{x:=y} \varphi[p*(x:=y)]$$
$$\varphi(p) \xrightarrow{go\ to\ L} \varphi[p, model.L]$$
$$\varphi(p) \xrightarrow{assumption\ \psi} (\varphi \wedge [p]\psi)[p]$$
$$\varphi(p) \xrightarrow{assertion\ \psi} \psi[empty],\ \neg Sat(\varphi \rightarrow [p]\psi)$$
$$\varphi(p) \xrightarrow{assertion\ \psi} 0,\ Sat(\varphi \rightarrow [p]\psi)$$
$$\varphi(p) \xrightarrow{stop} \varphi[\Delta]$$

Здесь φ, ψ – формулы исчисления предикатов первого порядка, p – параллельное присваивание или Δ. Функция *Sat u* проверяет выполнимость формулы *u*, используется для разрешения условий. Условный оператор **if** *u* **then** *P else Q* рассматривается как функциональное выражение и транслируется с помощью следующего правила: **if** *u* **then** *P* **else** $Q = ask\ u.P + (ask\ \neg u).Q$. Циклы и другие языковые конструкции могут быть введены аналогичным образом. Такая инсерционная машина может доказать частичную правильность недетерминированных программ, потому что операция недетерминированного выбора определена во входном языке. Если параллельное погружение определено, то такая машина может доказать частичную корректность параллельных программ над общей памятью. Для того чтобы избежать интерливинга, можно использовать разработанный нами алгоритм [14].

Методы проверки выполнимости формул

Проверка выполнимости формулы проходит в четыре этапа:

- элиминация кванторов всеобщности;
- элиминация функций доступа к элементам списков;
- элиминация функциональных атрибутов;
- доказательство замкнутой формулы без атрибутных выражений.

Если входные данные содержат аксиомы, то строится конъюнкция аксиом и формулы–ан-

нотации (при этом осуществляется вынесение кванторов всеобщности) и проверяется ее выполнимость. Алгоритм снятия кванторов всеобщности описан в [15].

Составные функциональные выражения преобразуются путем замены каждого вхождения подвыражения вида $f(x)$ на связанную квантором существования переменную y: $P(g(f(x))) \Rightarrow$ $\Rightarrow \exists y\,(P(g(y)) \wedge y = f(x))$. Для всех атрибутных выражений f массива или функционального выражения все вхождения $f(x_1), f(x_2), \ldots$ с различными параметрами $x_1, x_2, \ldots$ типа рассматриваются следующим образом: $f(x_i)$ заменяется на переменную y_i, связанную квантором существования, и к доказываемой формуле конъюнктивно добавляется выражение $(x_i = x_j) \rightarrow (y_i = y_j)$, а для массива типа $array\,(m)\ of\ <simple\ type>$ дополнительно к формуле добавляется ограничение на значение параметров $0 \le x_i < m$. Простые атрибутные выражения заменяются на переменные, связанные квантором существования.

В результате описанных преобразований получаем замкнутую формулу, где все переменные, связанные квантором существования, могут иметь один из следующих типов: булевый, целый, вещественный, перечислимый или символьный.

Наша система содержит три специализированных подпрограммы поиска доказательства утверждений (три прувера): целочисленный прувер для арифметики Пресбургера, алгоритм Фурье–Моцкина для линейной арифметики над вещественными числами, символьный прувер, включающий алгоритм поиска наиболее общего решения системы символьных равенств (усовершенствованный алгоритм унификации Монтанари–Росси, интегрированный с числовыми пруверами).

Переменные булевого типа рассматриваются как переменные перечислимого типа с двумя константами: 0,1. Для доказательства выполнимости замкнутой формулы она приводится к дизъюнкции конъюнкции литералов, связанных квантором существования. После этого достаточно доказать выполнимость одной из конъюнкций в дизъюнкции, выполнив сначала соответствующую подстановку в формулу констант из перечислимых типов вместо соответствующих переменных.

Алгоритм сужает область действия каждого литерала, связанного квантором существования, и начинает снимать эти кванторы для наиболее вложенных формул, вызывая при этом прувер соответствующего типа: целочисленный, вещественный или символьный. Более детальное описание встроенной системы проверки выполнимости формул будет дано в дальнейших публикациях авторов.

Результатом проверки выполнимости формулы могут быть три константы: *proved(1)*, *refuted(0)*, *not proved*. Если $Sat\ u = 1$, то формула выполнима. Если $Sat\ u = 0$, то формула не выполнима. Если $Sat\ u = not\ proved$, то формула не доказана, т.е. проверка выполнимости замкнутой формулы, которая получается в процессе доказательства, выходит за рамки функциональных возможностей встроенных пруверов. Самой распространенной причиной может быть нелинейность формулы (нелинейными считаются формулы, в которых встречаются переменные в степени выше 1-й). Для того, чтобы решить эту проблему, в нашу дедуктивную систему интегрирован внешний прувер – *cvc3* [16], а также разработан алгоритм подключения внешних пруверов, которым можно воспользоваться для подключения других пруверов.

Если $Sat\ u = not\ proved$, то вызывается функция проверки выполнимости из *cvc*3:*sat*3_*sat*. Эта функция также может вернуть один из трех результатов: *proved*(1), *refuted*(0), *not proved*. Однако в общем случае проверка выполнимости может занять значительное время. Для того, чтобы избежать излишнего ожидания, процесс доказательства формул в *cvc*3 можно ограничить по времени. Процесс доказательства, прерванный по времени, возвращает *not proved.* В некоторых случаях может оказаться, что доказательство того, что формула истинна при заданном состоянии памяти, в *cvc*3 может дать результат. Поэтому далее пробуем это доказать, используя функцию: *cvc*3_*validate* (с ограничением по времени). Если же и эта функ-

ция вернула *not proved*, тогда система спросит у пользователя, выполнима ли заданная формула или нет.

Количество используемых систем для проверки выполнимости можно расширить, например, используя следующие системы: *MathSat* [17], *Vampire* [18] и т.д. Для интеграции с внешними модулями *IMS* предоставляет следующий интерфейс:

1. Три определенных константы *proved*, *refuted*, *not_proved* в классе *Clew* [2] (этот класс представляет собой набор функций для работы с деревом).

2. Функция *put_result* класса *Clew* сообщает интерпретатору, что *APLAN*-процедура возвращает значение. (*APLAN* – язык системы *APS* [2].)

3. Процедура должна установить результат в предопределенное *APLAN* имя *verdict* (одну из трех констант пункта 1).

4. На вход функции поступает формула и описание среды с типами данных. Например, для подключения *cvc*3 использовано только поле *<interpreted def>*, поскольку *cvc*3 реализует алгоритмы в целочисленной арифметике (все переменные целого типа), однако для функциональных атрибутных выражений нужно было задавать количество параметров.

Примеры проверки правильности программ

Рассмотрим простой пример целочисленной функции, определяемой рекурсивно: $f(1)=1$, $f(n+1)=f(n)+(n+1)$. Запишем последовательность шагов вычисления этой функции, пользуясь операторами, принятыми в языках процедурного программирования:

fc:= 0; *k*:= *n*; /* присваивание начальных значений */
*L*1: *k*:= *k*–1; /* выполнение цикла вычислений */
*L*2: (*k*>=1) –> (*fc*:=*fc*+ (*k*+1)) *else*(*fc*:= *fc*+1; *go to L*3); *go to L*1; /* проверка условия выполнения цикла */
*L*3: *stop* /* завершение цикла вычислений */

Здесь n – входная переменная, определяет число слагаемых вычисляемой суммы, *fc* – выходная переменная, содержит результат вычисления суммы, k – переменная цикла.

Для верификации программы средствами системы *IMS* необходимо: аннотировать программу («разметить» ее аннотациями), подготовить среду верификации.

Назовем рассматриваемую функцию *Sumpos*. Аннотированная программа ее вычисления, подготовленная к верификации средствами *IMS*, выглядит следующим образом:

make_model(
*L*0: *assumption*:*n*>=1;
fc:= 0; *k*:= *n*;
*L*1: *k*:= *k*+ (–1);
*L*2: *assertion*:(*Sumpos*(*n*)=*fc*+*Sumpos*(*k*+1)) $\wedge$ 0 ≤ *k* $\wedge$ *k* ≤ *n*;
(*k* ≥ 1)–>(*fc*:=*fc*+(*k*+1)) *else*(*fc*:=*fc*+1; *go to L*3); *go to L*1;
*L*3: *assertion*:(*fc* = *Sumpos* (*n*));
stop);
*verify L*0);

Одним из этапов подготовки среды верификации программы является задание вычисляемой функции в виде системы переписывающих правил. Для функции *Sumpos* эта система имеет вид:

$$Sumpos{:}\mathrm{rs}(n)(Sumpos(n)=((n=1)\text{->}(Sumpos(n)=1))\wedge((n>1)\text{->}(Sumpos(n)=Sumpos(n-1)+n)).$$

Кроме переписывающих правил, среда верификации содержит описание переменных программы вычисления функции *Sumpos* и формулы, используемые при верификации:

(*program environment*:*obj*(*attributes*:*obj*(*fc*:*int*, *k*:*int*, *n*:*int*, *Sumpos*:*int*–>*int*);
simple_attributes:(*fc*:*int*, *k*:*int*, *n*:*int*); *interpreted*: *obj*(*Sumpos*:*int*–>*int*);
axioms: $\forall$ (*w*:int)(
((*w*=1)–>(*Sumpos*(*w*)=1)) $\wedge$
$\wedge$ ((*w*>1)–>(*Sumpos*(*w*)=*Sumpos*(*w*–1)+*w*)));
initial:1[*empty*]);

Процесс верификации сопровождается выдачей на экран сообщений, что дает возможность следить за ходом работы системы. Например, в случае верификации программы, вычисляющей функцию *Sumpo*s, последовательность сообщений выглядит следующим образом (сообщения разделяются запятыми):

trace, *init*, *start verify L*0, *assumption* ($n \geq 1$) *is consistent*, *fc*:=0, *k*:=*n*, *go to L*1, *k*:=*k*–1,
*go to L*2, *assertion* ((*Sumpos n* = *fc* + *Sumpos* (*k* +1)) $\wedge$ $\wedge$ 0 ≤ *k* $\wedge$ *k* ≤ *n*) *proved*, *ask*($\neg(k \geq 1)$),
fc:=*fc*+1, *go to L*3, *assertion* (*fc* = *Sumpos n*) *proved*, *stop*, *running model finished*, *all space covered*

Верификация прошла успешно.

Рассмотрим другую версию программы вычисления функции *Sumpos*. Опишем сначала среду верификации:

(*program environment*:*obj*(*attributes*:*obj*(*fc*:*int*, *fp*:*int*, *k*:*int*, *n*:*int*, *Sumpos*:*int*–>*int*);
simple_attributes:(*fc*:*int*, *fp*:*int*, *k*:*int*, *n*:*int*); *interpreted*:*obj*(*Sumpos:int*–>*int*);
axioms: ∀ (*w*:int)(
((*w*=1)–>(*Sumpos*(*w*)=1)) ∧
∧ ((*w*>1)–>(*Sumpos*(*w*)=*Sumpos*(*w*–1)+*w*)));
initial:1[*empty*]);

Аннотированная программа, подготовленная для обработки *IMS*, выглядит следующим образом:

make_model(
*L*0: *assumption*: $n \ge 1$;
fc:=1; *k*:=1; *fp*:=0;
*L*1: *k*:=*k*+1;
*L*2: *assertion*:(*fc*–*fp*=*k*–1) $\wedge 1 \le k \wedge k \le n+1$;
(*k*≤*n*)->(*fp*:=*fc*; *fc*:=*fp*+*k*) *else*(*go to L*3); *go to L*1;
*L*3: *assertion*:(*fc*=*Sumpos*(*n*));
stop);
*verify L*0);

Последовательность шагов *IMS*:

trace, *init*, *start verify L*0, *assumption* ($n \ge 1$)*is consistent*, *fc*:=1, *k*:=1, *fp*:=0, *go to L*1,

k:= *k*+1, *go to L*2, *assertion* ((*fc* – *fp* = *k* –1) ∧ $\wedge 1 \le k \wedge k \le n+1$) *proved*, *ask*(*k*<= *n*), *fp*:= *fc*,

fc:= *fp*+*k*, *go to L*1, *k*:= *k*+1, *go to L*2, *assertion* (*fc* – *fp* = *k* –1) $\wedge 1 \le k \wedge k \le n+1$) *proved*,

ask(¬(*k* ≤ *n*)), *go to L*3, *assertion* (*fc* = *Sumpos n*) *proved*

Заключительное сообщение данной последовательности было сформировано в результате таких действий. Утверждение–аннотация *fc* = = *Sumpos n* не было доказано с помощью встроенной функции проверки выполнимости *IMS*. В этом случае происходит обращение к внешней программе *cvc*3_*sat*, на вход которой подается формула:

$$\forall(fc:\text{int}, fp:\text{int}, k:\text{int})(k \le 0 \vee \neg(fc - fp = k-1) \vee$$
$$\forall(n:\text{int})((fc = Sumpos(n)) \vee n - k + 2 \le 0 \vee k - n \le 0)) \vee$$
$$\vee\exists(w:\text{int})(\neg(Sumpos(w) = w + Sumpos(w-1)) \wedge w > 1) \vee$$
$$\vee\exists(w:\text{int})(\neg(Sumpos(w) = 1) \wedge (w = 1)).$$

Эта программа возвращает *not_proved*. Далее происходит обращение к следующей функции *cvc*3_*validate*, ответ так же *not_proved*. Система *IMS* вступает в этом случае в диалог с пользователем, спрашивая, верно ли утверждение, которое ей не удалось доказать; в нашем случае пользователь дает утвердительный ответ:

not proved assertion fc = Sumpos n, *is it valid*? (1,0)>1; *assertion fc = Sumpos n proved*

Рассмотренные программы вычисления функции *Sumpos* различаются аннотациями, помещенными в цикл (после метки *L*2): аннотация в первой программе имеет вид *assertion*: ($Sumpos(n) = fc + Sumpos(k+1)) \wedge 0 \le k \wedge k \le n$, а во второй – *assertion*:($fc - fp = k-1) \wedge 1 \le k \wedge k \le n+1$. Формула $Sumpos(n) = fc + Sumpos(k+1)$ из первой аннотации по форме схожа с переписывающим правилом $Sumpos(n) = Sumpos(n-1) + n$ (для случая $n > 1$) в определении функции *Sumpos*, а формула $fc - fp = k - 1$ из второй аннотации отличается от этого переписывающего правила. Отсюда и различия в обработке программы вычисления функции *Sumpos* в *IMS*.

Эксперименты по верификации программ вычисления целочисленных функций, заданных рекурсивно, проводились со следующими функциями:

1. $f(1)=1,\ f(n+1)=f(n)+(n+1)$.
2. $f(1)=1,\ f(n+1)=f(n)+(n+1)^2$.
3. $f(1)=1,\ f(n+1)=f(n)+(n+1)^3$.
4. $f(1)=1,\ f(n+1)=f(n)+(2n+1)$.
5. $f(1)=1,\ f(n+1)=f(n)+(2n+1)^2$.
6. $f(1)=1,\ f(n+1)=f(n)+(2n+1)^3$.
7. $f(1)=2,\ f(n+1)=f(n)+(n+1)(3n+2)$.

Верификация осуществлялась с помощью системы *IMS* и (для контроля) вручную. В случае функций 2, 3, 5–7, определения которых содержат нелинейные выражения; в процессе верификации система обращалась за подсказкой к пользователю.

Работа *IMS* проверялась как на примерах правильно составленных программ (что подтверждалось проверкой вручную), так и программ, составленных неверно. Приведем пример верификации системой *IMS* ошибочной программы (программы вычисления функции 4 из перечня, приведенного выше; далее эта функция называется *Sumodds*).

Среда верификации и размеченная программа, подготовленная для обработки *IMS*, выглядят следующим образом:

(*program environment*: *obj* (*attributes*: *obj* (*fc*:*int*, *fp*:*int*, *k*:*int*, *n*:*int*, *Sumodds*:*int*->*int*);

simple_attributes: (*fc*:*int*, *fp*:*int*, *k*:*int*, *n*:*int*); *interpreted*:*obj* (*Sumodds*:*int*->*int*);

axioms: ∀ (*w*:int)(
((*w*=1)->(*Sumodds* (*w*)=1)) ∧
∧ ((*w* >1)–>(*Sumodds*(*w*) = *Sumodds*(*w* – – 1) + (2*w* –1))));

initial:1[*empty*]);

make_model(
*L*0: *assumption*:*n*>=1;
fc:=1; *k*:= 0; *fp*:=0;
*L*1: *k*:=*k*+1;
*L*2: *assertion*:(*fc*–*fp*=2*k*–1) ∧1 ≤ *k* ∧ *k* ≤ *n* +1;
(*k* ≤ *n*)–>(*fp*:=*fc*; *fc*:=*fp*+2*k*+1) *else*(*go to L*3); *go to L*1;
*L*3: *assertion*:(*fc*=*Sumodds*(*n*));
stop);
*verify L*0);

Программа составлена неверно: она вычисляет *Sumodds*(*n*+1), а не *Sumodds*(*n*). В ходе верификации происходит обращение к функции *cvc3_sat* с входной формулой:

$$\forall(fc:\text{int}, fp:\text{int}, k:\text{int})(k \le 0 \vee \neg(2k-1 = \\ = fc - fp) \vee \forall(n:\text{int})((fc = Sumpos(n)) \vee \\ \vee n - k + 2 \le 0 \vee k - n \le 0)) \vee \exists(w:\text{int}) \times \\ \times(\neg(Sumpos(w) = 2w + Sumpos(w-1) - 1) \wedge \\ \wedge w > 1) \vee \exists(w:\text{int})(\neg(Sumpos(w) = 1) \wedge (w = 1)).$$

Эта функция возвращает *refuted*, поэтому система выводит следующее сообщение: *assertion fc=Sumodds n is not valid.*

Заметим, что ошибочность программы вычисления функции *Sumodds* установлена автоматически: обращения к пользователю за подсказкой в ходе верификации не происходило.

Заключение. Разработанная система успешно используется в учебных процессах Киевского национального университета имени Тараса Шевченко и Херсонского гос. университета при обучении доказательному программированию. В дальнейшем планируется расширить количество внешних модулей для доказательства выполнимости формул, а также продолжение доказательства частичной корректности программ на новых примерах (параллельных программах и др.).

Работа выполнена при поддержке ДФФД в рамках проекта Ф40.1/004.

1. *Letichevsky A.A.*, *Letychevskyi O.A.*, *Peschanenko V.S.* Insertion Modeling System // PSI 2011, Lecture Notes in Comp. Sci. – **7162**. – Springer, 2011. – P. 262–274.
2. *Letichevsky A.A.*, *Kapitonova Ju.V.*, *Konozenko S.V.* Algebraic programming system APS-1 / O.M. Tammepuu (Ed.), Informatics'–89 // Proc. of the Soviet–Franch symp., Tallin, 1989. – P. 46–55.
3. *SPEC* – Standard Performance Evaluation Corporation. – http://www.spec.org
4. *Letichevsky A.A.*, *Gilbert D.R.* A General Theory of Action Languages. – Cybernetics and System Analyses. – 1998. – **1**. – P. 16–36.
5. *Insertion* modeling in distributed system design / A. Letichevsky, Ju. Kapitonova, V. Kotlyarov et al. // Problems of Programming. – 2008. – **4**. – P. 13–39.
6. *Milner R.* Communication and Concurrency. – New Jersey: Prentice Hall, 1989. – 260 p.
7. *Milner R.* Communicating and Mobile Systems: the Pi Calculus. – Cambridge University Press, 1999. – 161 p.
8. *Hoare C.A.R.* Communicating Sequential Processes. – New Jersey: Prentice Hall, 1985. – 260 p.
9. *Cardelli L.*, *Gordon A.* Mobile Ambients // Foundations of Software Science and Computational Structures / Maurice Nivat (Ed.). – 1998. – LNCS 1378. – P. 140–155.
10. *Insertion* programming / A. Letichevsky, Ju. Kapitonova, V. Volkov et al. // Cybernetics and System Analyses. – 2003. – **1**. – P. 19–32.
11. *History* of APS&IMS Systems. – http://apsystem.org.ua
12. *Hoare C.A.R.* An axiomatic basis for computer programming // Communications of the ACM. – 1969. – **12**(10). – P. 576–580.
13. *Floyd R.W.* Assigning meanings to programs // Proc. of the American Mathematical Soc. Symposia on Applied Math., 1967. – **19**. – P. 19–31.
14. *Letichevsky A.*, *Letichevsky O.*, *Peschanenko V.* Efficient algorithm for reachability / V. Ermolayev (Ed.) // Proc. 8-th Int. Conf. ICTERI 2012, Kherson, Ukraine, June 6–10, 2012. – P. 71–81.
15. *Nikitchenko N.S.*, *Tymofieiev V.G.* Satisfiability Problem in Composition-Nominative Logics of Quantifier-Equational / V. Ermolayev (Ed.) // Ibid. – P. 56–70.
16. *CVC3*: The CVC3 User's Manual – http://www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html
17. *The MathSat* 5 SMT Solver – http://mathsat.fbk.eu/
18. *Vampire's* Home Page –http://www.vprover.org/

Тел. для справок: +38 044 526-0058 (Киев)
E-mail: *mmk@incyb*