УДК 004.827, 004.832.22, 004.832.25

Система программирования в ограничениях Nemo

Сидоров В.А. (Институт систем информатики СО РАН)

Метод недоопределённых вычислений является одним из подходов для решения задачи удовлетворения ограничений. На его основе разработан ряд программных систем, в том числе система программирования в ограничениях Nemo. В статье рассматриваются принципы, используемые для разработки системы Nemo, их реализация на практике и возможности, предоставляемые пользователю для решения задач.

Ключевые слова: *задача удовлетворения ограничений, метод недоопределённых вычислений, язык описания модели.*

1. Введение

Задача удовлетворения ограничений (**ЗУО**) является одним из способов формулировки и решения нестандартных и сложных вычислительных задач. Неформально ЗУО можно описать следующим образом:

- Решаемая задача состоит из набора переменных и набора ограничений.
- С каждой переменной связана собственная область определения (множество допустимых значений), которая может изменяться в процессе решения ЗУО.
- Ограничения связывают переменные и ограничивают множества их значений.
- Решением ЗУО является такой набор значений переменных, для которого удовлетворяются все ограничения.
- Целью решения ЗУО может быть как поиск одного решения (возможно, с заданными свойствами), так и поиск всех решений.

Впервые задача удовлетворения ограничений была сформулирована в начале 70х годов [13, 15].

В начале 80х годов А.С. Нариньяни был предложен *метод недоопределённых вычислений* [5, 8], который является одним из методов решения задачи удовлетворения ограничений. Метод обладает следующими особенностями:

- Значения переменных являются *недоопределёнными*, представленными в виде множества допустимых значений.
- Алгоритм вычислений фиксированный. На каждом шаге удовлетворяется (вычисляется) одно активное ограничение; выделяются переменные, значение

которых изменилось на данном шаге; активируются ограничения, связанные с данными переменными.

Отметим важную особенность метода недоопределённых вычислений — универсальность. Алгоритм не зависит от типа обрабатываемых данных; в частности он позволяет решать задачи, содержащие как непрерывные, так и дискретные численные значения переменных.

На основе метода недоопределённых вычислений было создано нескольких линеек программных систем, таких как математические вычислители UniCalc [1], экспертные системы Semp [3], мультиагентные системы ТАО [2], электронные таблицы FinPlan [17]. В статье рассматривается система программирования в ограничениях Nemo, относящаяся к семейству систем NeMo [4, 11], характерными особенностями которых является возможность работать с различными типами данных и специфицировать задачу на высоком уровне.

Система Nemo изначально проектировалась как промышленная система с сохранением широких возможностей по расширяемости и универсальности, характерных для семейства NeMo. Рассмотрим её свойства в следующем порядке:

- Вначале сформулируем общие принципы (и вытекающие из них свойства) используемые при разработке системы;
- Далее опишем реализацию указанных свойств в архитектуре системы;
- В конце опишем некоторые возможности предоставляемые пользователям системы для формулировки и решения различных типов задач.

2. Принципы создания системы Nemo

Предназначение системы Nemo определяется следующим образом:

1. Система предназначена для использования в роли универсального встраиваемого вычислителя. Главными характеристиками в этом случае являются **производительность** и **технологичность**.
2. Система предназначена для исследования различных задач удовлетворения ограничений и алгоритмов их решения. Ключевой характеристикой при этом является **универсальность** - возможность настраивать систему для решения различных классов задач.

На первый взгляд, эти характеристики противоречат друг другу, но это не так. Например, для достижения максимальной производительности требуется использовать различные алгоритмы для различных классов задач, что, в свою очередь, определяется степенью универсальности системы.

Для реализации этих характеристик был определен ряд свойств, которые использовались как при разработке архитектуры системы, так и были доступны далее на уровне пользователя:

1. **Универсальность** означает способность формулировать и решать максимально широкий список задач. На степень универсальности влияют следующие свойства системы Nemo:
   - Декларативное представление модели: описание модели содержит только описание задачи, а не алгоритм её решения. Это имеет ряд преимуществ и недостатков:
     - Упрощается формулировка задачи пользователем.
     - Порядок задания ограничений (описания задачи) не влияет на результат вычислений.
     - Декларативность ограничивает круг решаемых задач – система Nemo позволяет решать только статические (не изменяющиеся в процессе решения) задачи.
   - Расширяемость системы позволяет использовать дополнительные специализированные типы данных и ограничений, и, соответственно, настраивать систему для решения конкретных классов задач.
   - Поддержка объектно-ориентированного строения модели повышает уровень описания задачи, что упрощает формулировку сложных задач.
   - Использование концепции «недоопределённость» позволяет работать с неточными и не полностью определёнными данными.
   - Единый универсальный алгоритм решения задачи позволяет решать любые задачи удовлетворения ограничений.
2. Под **производительностью** мы понимаем не только собственно скорость решения задачи, но и возможность описывать и решать большие и сложные задачи.
   - Скорость вычислений. Для решения ЗУО используется универсальный встроенный в систему алгоритм достижения совместности и различные методы перебора с откатами [12, 16]. В худшем случае, скорость перебора экспоненциально зависит от числа переменных, но в среднем её можно улучшить за счёт использования различных эвристик, специализированных ограничений и более эффективного представления данных.

- Возможность формулировать и решать большие задачи закладывалась на этапе проектирования системы.
- Высокоуровневый язык описания модели позволяет быстро описывать решаемую задачу.

3. **Технологичность**. Простота интеграции в системы сторонних производителей достигается за счёт:
   - Модульного строения системы.
   - Мощного внешнего программного интерфейса (API) системы.
   - Использования парадигмы «интерфейс-реализация» (COM-технология) при создании API.

Далее рассмотрим реализацию этих принципов с точки зрения внутренней архитектуры системы (инструментальный уровень) и с точки зрения средств, предоставляемых пользователю для решения конкретных задач (пользовательский уровень).

3. Архитектура системы Nemo. Инструментальный уровень

Система Nemo состоит из:

- Загружаемых модулей (библиотек), содержащий типы данных и ограничения, используемые при создании и обработке вычислительной модели.
- Ядро системы, включающего в себя вычислительную модель, реализацию метода недоопределённых вычислений и менеджер загружаемых модулей.
- API системы.
- Набор утилит для тестирования и отладки системы, трансляторы с языка описания модели Nemo.

Архитектура проектировалась с учётом поддержки различных вычислителей и различных API, хотя на практике был реализован всего один вычислитель и один API для него.

3.1. Библиотеки типов данных и ограничений

Ядро системы реализует базовый алгоритм метода недоопределённых вычислений и не содержит никакой информации о существующих типах данных и ограничениях – вся информация о них хранится во внешних библиотеках. Соответственно, тип данных (или ограничение) становится доступными только после загрузки соответствующей библиотеки с помощью менеджера модулей. Это является результатом последовательного следования принципу расширяемости.

Разработан и реализован достаточно большой список дополнительных модулей — в стандартной конфигурации система включает 12 библиотек, содержащих большой набор (порядка 700) типов данных и отношений.

3.2. Ядро системы

Внутренняя реализация описания задачи (*вычислительная модель*) строится на основе 2 понятий: объект и дескриптор объекта.

- Все переменные задачи являются объектами. Сам объект содержит только своё текущее значение, которое может изменяться в процессе вычислений.
- Дескриптор объекта содержит всю информацию об объекте (полное описание типа данных, доступные интерфейсы и свойства). Также он используется для создания, удаления и копирования объекта.
- Ограничение также является объектом. Связь между ограничением и его аргументами вынесена на уровень выше, в составной объект (см. далее).
- Дескриптор ограничения хранит имя и типы аргументов ограничения.
- Составной объект представляет собой контейнер для объектов.
- Дескриптор составного объекта хранит описание структуры составного объекта (список дескрипторов составляющих его объектов и ограничений, их имена) и таблицу связей между внутренними ограничениями и объектами.
- Отдельная сущность - обобщённый (generic) дескриптор, который создаёт дескрипторы, а не объекты.
- Вычислительная модель является обычным составным объектом.

Ядро также содержит менеджер модулей, список загруженных дескрипторов и реализацию алгоритма недоопределённых вычислений, работающего с вычислительной сетью.

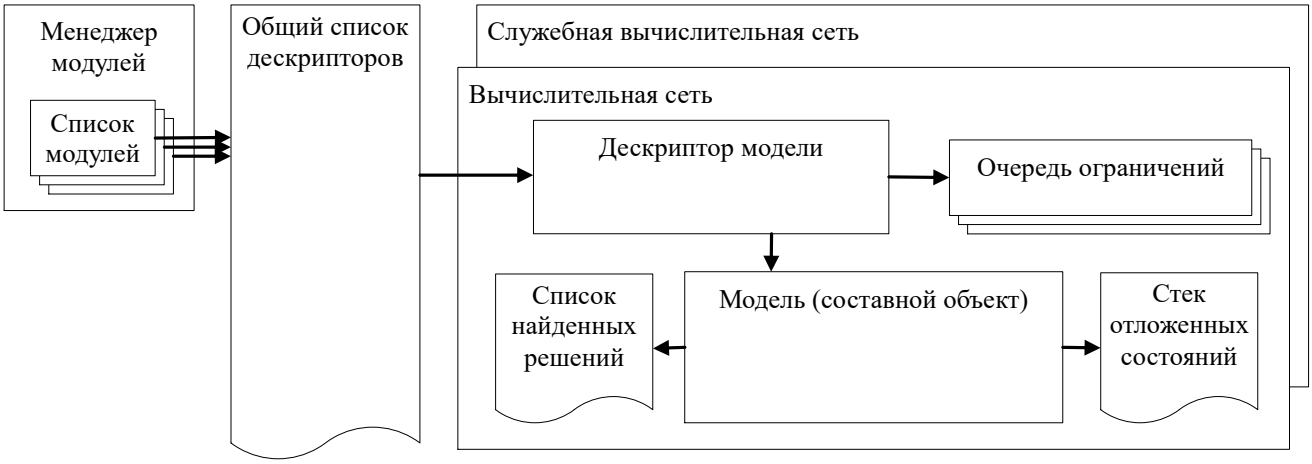

Рис. 1. Строение ядра системы Nemo.

Компоненты вычислителя взаимодействуют следующим образом:

- Менеджер модулей позволяет загружать библиотеки (внешние модули), извлекать из них дескрипторы и добавлять их в общий список дескрипторов.
- Общий список дескрипторов содержит все доступные в конкретный момент времени дескрипторы.
- Внутреннее представлением решаемой задачи (вычислительная сеть) создаётся на основе дескриптора модели, который является обычным дескриптором составного объекта.
- Дескриптор модели используется для создания самой модели (которая является составным объектом), очереди ограничений, и для хранения связей между ограничениями и их аргументами.
- Во время вычислений текущий набор значений объектов модели может сохраняться:
  - в списке найденных решений – если это решение;
  - в стеке отложенных состояний – для обработки этого набора значения в дальнейшем, на следующих шагах алгоритма.
- В некоторых специальных случаях (таких как решение задачи иерархического удовлетворения ограничений, вычисление невязки и т.д.) создаётся служебная вычислительная сеть.

3.3. API

Программный интерфейс системы (API) позволят пользователю формулировать и решать задачи с помощью более высоко-уровневых понятий. Интерфейс API создан с помощью языка описания интерфейсов (idl), реализация написана на C++. API создан в соответствии с моделью «интерфейс-реализациия» объектно-ориентированного парадигмы программирования. API системы состоит из следующих интерфейсов:

Фабрика. Объект этого класса всегда присутствует в единственном экземпляре и служит для создания объектов всех остальных классов.

Модуль. Модуль представляет собой контейнер, содержащий списки типов данных и отношений. Модуль может быть записан в файл и загружен из файла.

Тип данных. Служит для доступа к типам данных модели. Пользователь может создать новый типы данных.

Отношение. Служит для доступа к функциям, операциям и предикатам модели. Пользователь может создать новое отношение (новую функцию или операцию).

Переменная. Переменная модели. При создании переменной указываются её имя и тип данных. Переменная также содержит методы для установки и получения её значения, пометки переменной как константы, для доступа к внутренним слотам переменной (если тип данных переменной - составной).

Терм. Терм является внутренней вершиной дерева выражения. При создании терма указываются отношение и список аргументов (которыми могут быть либо переменные, либо термы). Также как и переменная, терм имеет собственное имя и тип данных.

Ограничение. Элементарный элемент, с помощью которого строится модель. Все факты (уравнения, неравенства, предикаты), которые добавляются в модель, должны быть ограничениями. Ограничение создаётся по терму, являющемуся корнем дерева выражения.

Решатель. Служит для создания модели и управления процессом вычислений.

Таблица. Таблица служит для хранения данных в собственном формате и обеспечивает создание и доступ к строкам и столбцам данных. Таблица используется при создании табличного отношения.

Строка таблицы. Одна строка данных таблицы. Используется для заполнения таблицы данными.

3.4. Сервисный уровень

Сервисный уровень содержит программы и модули для работы с системой Nemo. Из них можно выделить:

- **Компилятор языка** описания модели Nemo является основным инструментом для формулировки задач и тестирования системы.
- **Интерпретатор языка** описания модели Nemo позволяет в интерактивном режиме работать с моделью: загружать модули, создавать переменные, модифицировать модель и выполнять вычисления. Интерпретатор используется для тестирования методов динамического управления моделью через API.
- **Отладчик** является основным средством отладки алгоритма вычислений. Он позволяет выполнять вычисления по-шагово, отображая текущие значения переменных и состояние очереди ограничений.
- Утилита для просмотра содержимого библиотек.
- **Компилятор таблиц** позволяет создавать табличные ограничения на основе текстового представления данных.

- **Декомпиллятор** восстанавливает описание модели на языке Nemo.

Все указанные утилиты и модули взаимодействуют с системой исключительно через её API.

4. Архитектура системы Nemo. Пользовательский уровень

Для пользователя, система, в первую очередь, является инструментом для решения задач. С этой точки зрения её описание представляет собой набор методов и приёмов, используемых для формулировки и решения различных задач. Система Nemo предоставляет 2 способа, с помощью которых возможно сформулировать решаемую задачу:

- Низкоуровневый – с помощью API системы, на языке программирования (C++, Basic).
- Высокоуровневый – с помощью языка описания модели Nemo, в текстовом виде.

Далее мы будем использовать второй способ (описание на языке Nemo), так как язык поддерживает все возможности, предоставляемые API и при этом оперирует более высокоуровневыми понятиями. Отметим, что язык Nemo является развитием языка описания модели системы HeMo+ [7] при сохранении его базовых принципов и свойств.

Напомним базовые свойства системы, которые можно использовать при описании задачи:

- Декларативное представление модели.
- Модульное строение системы.
- Расширяемость (возможность создания новых типов данных и новых отношений).
- Объектно-ориентированное строение модели.
- Поддержка концепции недоопределённости.
- Поддержка различных классов решаемых задач.

4.1. Декларативное представление модели

Так как алгоритм для решения задач встроен в систему, то пользователю достаточно специфицировать задачу (задать объекты, их значения и связывающие их отношения) в терминах языка системы и запустить механизм вычислений.

```
uses "NemNumbers";
main
    x, y, d, z : interval real;
    x + y = 12.0;
```

```
    2 * x = y;
    z^3 + y^5 = d;
    x^2 + 4 * z - 6.33 = 0.0;
end;
```

Пример 1. Простая система уравнений.

В примере:

- загружается библиотека «NemNumbers», которая содержит реализацию основных числовых типов данных и операций над ними (в том числе типа данных «interval real»);
- структурные скобки «main … end» ограничивают описание задачи;
- декларируются 4 переменных: x, y, d, z;
- задаются ограничения задачи в виде системы уравнений.

Как уже было отмечено, декларативность с одной стороны упрощает формулировку задачи, с другой – ограничивает класс решаемых задач. В частности, система не позволяет решать динамические задачи удовлетворения ограничений. Что не мешает использовать систему Nemo как компоненту для решения ЗУО в системах с динамически изменяемой моделью (например, система Semp-ТАО [6]).

4.2. Модульность

Модульное строение системы позволяет наращивать систему новыми типами данных (классами) и отношениями путём подключения дополнительных внешних модулей (библиотек).

Вычислитель системы Nemo реализует базовый алгоритм вычислений и содержит системные классы и методы. При этом он не имеет никакой информации ни о каких конкретных типах данных (классах) и отношениях – все они загружаются из внешних модулей динамически. Перед языком, и частично перед API, это ставит несколько проблем:

- Представление констант. Действительно, так как не существует предопределённых типов данных, и так как число типов данных не ограничено, то транслятор не должен иметь встроенных средств разбора констант. Для решения этой проблемы были приняты следующие соглашения:
  - Сделано исключения для распространённых типов данных: заданы форматы представления значений для типов с именами ***bool***, ***real***, ***int*** и ***string***.
  - Заданы форматы представления значений для видов недоопределённости ***Interval***, ***Enum*** и ***Multiinterval***.

- Для каждого типа данных реализован (непосредственно в дескрипторе объекта) метод получения значение объекта из текстовой строки. При этом формат текстовой строки определяется внутри реализации этого метода. Соответственно, в язык введена конструкция «**const**(***TypeName***, "Value")», которая создаёт константный объект типа TypeName со значением, загруженным из текстовой строки "Value".

- Отношения в языке задаются в виде операций и функций. Наличие большого (потенциально - неограниченного) количества отношений требует от языка и API поддержки возможности свободного задания имён и сигнатур операций и функций. А также возможность их переопределения.
- В связи с большим количеством операций возникает проблема указания их ассоциативности и приоритета, используемых при разборе выражений. Для её решения введены два соглашения:
  - Приоритеты операций с одним именем совпадают. Ассоциативности операций с одним именем совпадают.
  - Приоритеты и ассоциативности операций загружаются из внешнего текстового файла.

Заметим, что для API этой проблемы не существует, так как выражения собираются из термов по-одному, вне зависимости от приоритетов, ассоциативности и префиксной/инфиксной формы вызова отношения.

```
class Point;
    slots (x, y, z: real);
end;

function Norm( P : Point ) : positive;
    Result^2 = (P.x^2 + P.y^2 + P.z^2);
end;

operator - ( P1, P2 : Point ) : Point;
    Result.x = P1.x - P2.x;
    Result.y = P1.y - P2.y;
    Result.z = P1.z - P2.z;
end;
```

```
operator + ( P1, P2 : Point ) : Point;
    Result.x = P1.x + P2.x;
    Result.y = P1.y + P2.y;
    Result.z = P1.z + P2.z;
end;

function Distance( P1, P2: Point ): positive;
    Result = Norm(P1 - P2);
end;
```

Пример 2. Фрагмент модуля геометрической библиотеки.

В примере:

- создаётся новый тип данных «Point», содержащий 3 вещественных поля: x, y, z;
- создаются функции «Norm» и «Distance», а также операции «-» и «+» для работы с этим типом данных.

4.3. Виды недоопределённости

Вид недоопределённости задаёт способ представления множества допустимых значений переменной [10]. В язык введена поддержка видов недоопределённости: ***Single***, ***Interval***, ***Enum***, ***MultiInterval***. При описании переменной модели, можно явно указывать вид недоопределённости; по умолчанию используется ***Interval***.

```
uses "NemNumbers";
uses "NemEnumIntegers";
uses "NemAbstractTypes";

main
 // qi is the queen in the i-th column
    q1, q2, q3, q4, q5, q6, q7, q8 : enum int;
    q1 = [1, 8]; q2 = [1, 8]; q3 = [1, 8]; q4 = [1, 8];
    q5 = [1, 8]; q6 = [1, 8]; q7 = [1, 8]; q8 = [1, 8];

    alldiff( q1, q2, q3, q4, q5, q6, q7, q8 );
    alldiff( q1-3, q2-2, q3-1, q4, q5+1, q6+2, q7+3, q8+4 );
    alldiff( q1+3, q2+2, q3+1, q4, q5-1, q6-2, q7-3, q8-4 );
end;
```

%exact = **On**;

Пример 3. Задача о расстановке ферзей.

В примере:

- Для решения ЗУО в дискретных областях удобно использовать представление значений переменных в виде перечисления. Таким типом данных является «enum int», реализация которого загружается из библиотеки «NemEnumIntegers».
- В данной задаче удобно устанавливать начальные значения переменных с помощью интервалов: «q1 = [1,8]».
- Ограничение «alldiff» задаёт попарное неравенство всех своих аргументов. Ограничение имеет произвольное количество аргументов и реализовано в библиотеке «NemAbstractTypes».
- Параметр алгоритма «%exact» указывает, что выполняется поиск точных решений (см. далее подраздел 4.9. Классы решаемых задач).

4.4. Типы данных

Тип данных представляет собой описание, содержащее информацию, общую для некоторой группы переменных. Типы данных системы Nemo (и языка) можно разделять по различным критериям.

По роли в модели:

- **Абстрактные типы данных** служат для создания иерархии типов данных.
- **Конкретные типы данных** используются для создания объектов модели.

По строению:

- **Простые типы данных.** Объекты таких типов являются неделимым сущностями**.**
- **Структурные типы данных**. Переменные таких типов имеют внутреннее строение, которое можно использовать при описании модели.

По способу создания:

- **Инструментальные типы данных** создаются разработчиками системы. Такие типы реализуются на языке программирования (C++) и могут быть абсолютно произвольными. При описании модели такие типы данных становятся доступными только после загрузки соответствующего модуля.
- **Составные типы данных** могут создаваться пользователем различными способами:

1. Создать тип данных, явно описав его структуру (для записей – это список слотов; для массивов – тип элементов и размер).
2. Наследовать тип данных от уже существующего типа.
3. Добавить к типу данных собственную подмодель (дополнительный набор ограничений на внутренние поля объектов).

- Типы данных, созданные с помощью **обобщённых** (родовых) **классов**.

Отметим что, так как система Nemo работает с недоопределёнными данными, то для каждого (не абстрактного) типа данных должен указываться вид недоопределённости. Иерархия типов данных, реализованных в стандартной конфигурации, представлена на следующей схеме:

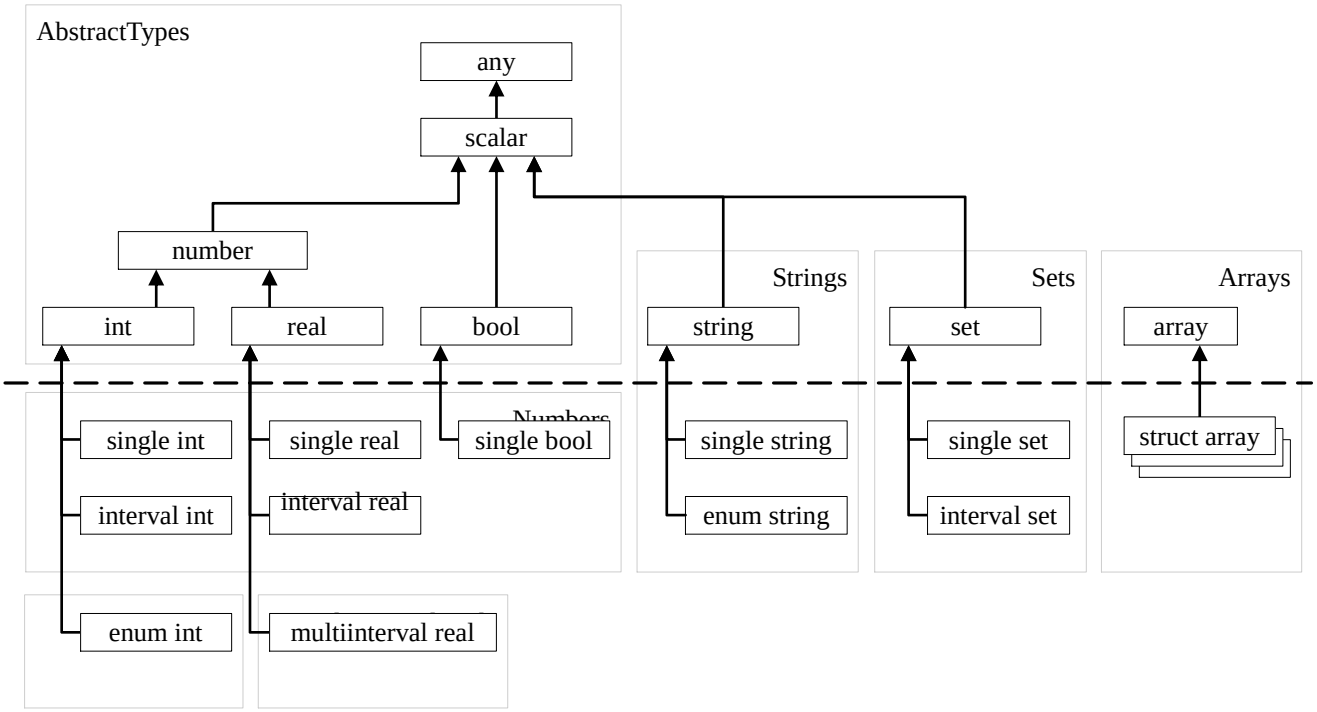

Рис. 2. Иерархия типов данных

Все указанные типы данных могут использоваться сразу же после подключения соответствующих библиотек.

4.5. Отношения

Также как и типы данных (по отношению к переменным), отношения обобщают информацию, общую для некоторой группы ограничений. Отношения можно разделять по различным критериям:

С точки зрения синтаксиса языка:

- **Функции**. Имя функции является идентификатором. В ограничениях функция используется только в префиксной форме.

- **Операции**. Имя операции состоит из специальных символов. В ограничениях операция может использоваться как в инфиксной, так и в префиксной формах.
- **Методы** типов данных. Метод связан с конкретным типом данных и задаёт ограничения, связывающие объект данного типа с внешними объектами.

```
function Norm( P : Point ) : positive;
    Result^2 = (P.x^2 + P.y^2 + P.z^2);
end;

operator + ( P1, P2 : Point ) : Point;
    Result.x = P1.x + P2.x;
    Result.y = P1.y + P2.y;
    Result.z = P1.z + P2.z;
end;

method angle::deg() : real;
    Pi*Result = 180*This;
end;
```

Пример 4. Фрагмент модуля геометрической библиотеки.

В примере создаются отношения:

- функция «Norm»;
- операция сложения для объектов типа данных «Point»;
- метод «deg» для инструментального типа данных «angle».

Заметим, что на уровне API такого разделения не существует – ограничения создаются с явным указанием имени отношения и списка аргументов.

По способу создания:

- **Инструментальные отношения** создаются разработчиками системы. Такие отношения реализуются на инструментальном языке программирования (C++) и могут быть абсолютно произвольными. При описании модели такие отношения становятся доступными сразу после загрузки соответствующего модуля.
- **Составные отношения** могут создаваться пользователем (точнее – инженером знаний). При этом он должен явно указать имя отношения, описать сигнатуру (типы аргументов и результата) и набор ограничений, из которых состоит данное отношение.

- Отношения, созданные с помощью механизма **обобщённых ограничений**.

4.6. Объектно-ориентированная парадигма описания модели

Парадигма объектно-ориентированного программирования (**ООП**) естественным образом вытекает из поддержки типизации и модульного строения системы. Понятие ООП также можно разложить на несколько составляющих:

1. **Типизация**. Это свойство означает наличие понятия «тип данных» («класс»), возможность прямо указывать класс каждого элемента модели и возможность создавать новые классы и отношения. API и язык поддерживает иерархию классов, как реализованных на инструментальном уровне, так и создаваемых пользователем.
2. **Модульность**. Система работает с двумя типами модулей: инструментальный модуль (динамически загружаемая библиотека со стандартизованным интерфейсом классов и процедур) и пользовательский (созданный самим пользователем) модуль. С точки зрения пользователя, они полностью эквивалентны.
3. **Инкапсуляция**. Инкапсуляция означает скрытие информации: данных и реализацию методов взаимодействия с объектом, - внутри самого объекта. Реализация этого понятия в системе Nemo имеет свои особенности:
    - В системе нет прав доступа к внутренним полям объекта – они все доступны (в терминах C++ - «публичные»).
    - Для типа данных можно указать **собственную подмодель** (см. подраздел 4.4. Типы данных), которая позволяет централизовать и скрывать функциональную информацию о связях между внутренними данными объекта.
    - Для типа данных можно описать набор методов для доступа к внутренним полям объекта этого типа (см. подраздел 4.5. Отношения).
4. **Наследование**. Система Nemo предоставляет методы построения новых классов при помощи механизма наследования. При этом имеются некоторые ограничения: запрещено множественное наследование, и базовый тип может быть только составным (массивом или записью).
5. **Полиморфизм**. Поддерживается возможность перегружать слоты и методы классов при наследовании, а также перегружать операции и функции для новых типов аргументов. Но так как нет виртуального наследования и виртуальных методов классов, то нельзя сказать, что полиморфизм реализован в полном объёме.

```
class Point2D; slots(x, y : real); end;
```

```
class Point3D: Point2D;
    slots(z : real);
end;
class Sphere: Point3D;
    slots(R : real);
    R >= 0.0;
end;
class Circle: Sphere;
    z = 0.0;
end;

method Point2D::Norm() : positive;
    Result^2 = This.x^2 + This.y^2;
end;
method Point3D::Norm() : positive;
    Result^2 = This.Norm() + This.z^2;
end;
```

Пример 5. Наследование в языке Nemo.

В примере создается иерархия типов данных: «Point2D» -> «Point3D» -> «Sphere» -> «Circle». Метод «Norm» создаётся для типа данных «Point2D» и перегружается для типа данных «Point3D».

Отметим, что конструкция «This.Norm()» внутри описания метода «Point3D::Norm()» указывает на метод родительского класса «Point2D::Norm()», так как для типа «Point3D» этот метод ещё не определён и, соответственно, не перегружен.

4.7. Механизм обобщённых классов и отношений

Существуют группы типов данных, отличающиеся друг от друга одним или несколькими параметрами. Такие типы данных можно сгруппировать в «метаклассы», которые мы будем называть **обобщёнными классами**. При этом, конкретный тип данных строится на основе обобщённого класса и набора параметров. Характерными примером такой конструкции является массив - для создания конкретного типа массива достаточно:

- Использовать обобщённый класс **generic array**.
- Указать тип элементов будущего массива.
- Указать размер будущего массива.

Тип элементов и размер задаются в виде параметров.

```
uses "NemNumbers";
uses "NemArrays";
uses "NemEnumIntegers";
uses "NemAbstractTypes";
main
    x,y: generic array <5, enum int>;
    i0, i1, i2, i3, i4 : enum int;
    i0=[0,4]; i1=[0,4]; i2=[0,4]; i3=[0,4]; i4=[0,4];
    x[0] = 3; x[1] = 1; x[2] = 7; x[3] = 5; x[4] = 0; // x = {3, 1, 7, 5, 0};
    x[i0] <= x[i1]; x[i1]<= x[i2]; x[i2] <= x[i3]; x[i3] <= x[i4];
    alldiff(i0,i1,i2,i3,i4);

    y[0] = x[i0];
    y[1] = x[i1];
    y[2] = x[i2];
    y[3] = x[i3];
    y[4] = x[i4];
end;
%exact = On;
```

Пример 6. Задача сортировки массива.

В примере создаются 2 массива (x и y). В результате вычислений не отсортированные значения из первого массива будут записаны в заданном порядке во второй массив.

В стандартной конфигурации системы реализованы несколько обобщённых классов:

- **generic array** служит для создания типов данных, описывающих одномерные массивы фиксированной длины.
- **generic scale** используется для создания перечислений.
- **generic grid** создаёт числовой тип данных с указанным шагом значений.
- **generic precision** создаёт вещественный числовой тип данных, все операции над которым выполняются с указанной точностью.

Все вышесказанное справедливо и для отношений: существуют группы отношений, отличающиеся друг от друга некоторыми параметрами; такие группы можно естественным образом сгруппировать в обобщённые классы отношения. В стандартной конфигурации системы реализованы следующие обобщённые классы отношений:

- **generic zero_partially** создаёт ограничения для решения задачи иерархического удовлетворения ограничений (РЗУО).
- **generic table** используется для создания табличных ограничений.
- **generic blackbox** используется для создания BlackBox-ограничений.

Механизм обобщённых классов и отношений поддерживается на уровне вычислителя системы Nemo, и далее в API и в языке Nemo.

4.8. Таблицы и BlackBox-ограничения

Отдельно отметим два вида универсальных отношений, с помощью которых можно смоделировать любое ограничение:

- **Табличные ограничения**. Как правило, ограничения на значения числовых объектов (целых, вещественных, логических) задаются интенсионально (посредством формул). Но в реальных задачах часто приходится иметь дело с ограничениями, заданными экстенсионально (посредством перечисления всех наборов значений, которые удовлетворяют данному ограничению). Каждое такое экстенсиональное ограничение удобнее всего представлять в виде таблицы, число столбцов которой совпадает с числом аргументов ограничения, а в каждой строке записан допустимый данным ограничением набор значений аргументов этого ограничения. На практике, табличные ограничения используются для задания нормативных параметров, для поддержки сложных перечислений и для взаимодействия с базами данных [14].
- **BlackBox-ограничения** (чёрный ящик). Существует класс задач, при описании и решении которых используются внешние, по отношению к системе Nemo, специализированные модули. Такие модули могут появляться в различных случаях:
  - Если для решения задачи необходимо использовать внешний модуль, реализующий некоторый закрытый алгоритм, взаимодействие с которым осуществляется только через заданный публичный интерфейс.
  - Если у пользователя есть готовые модули для решения подзадач и он хочет их использовать для задачи, решаемой системой Nemo. В этом случае пользователю может быть известна некоторая дополнительная информация, касающаяся поведения алгоритма. Например монотонность или дифференцируемость функции, вычисляемой алгоритмом.

  С точки зрения системы Nemo такие ограничения являются «чёрными ящиками» и для работы с ними создан специальный вид ограничений – BlackBox-ограничения [9, 18].

При использовании языка Nemo табличные и BlackBox-ограничения создаются с помощью механизма обобщённых классов:

```
function AB( abs1, abs2: int );
    table<"./", // path
            "Trains_from_a_to_b.nbt", // table name
            "1", // memory or file
            binary table, // table descriptor
            interval int, interval int, // column descriptors
            interval int, interval int> // argument descriptors
    ( abs1, abs2 );
end;

function Volume(x, y, z :real) : real;
    blackbox<
            "x1 * x2 * x3", "BlackBoxTest", "Calculator", 10000,
            "input single real[3]", "output single real", 0, 0,
            "NGradient exact", 1.e-5, 1, 0>
    (x, y, z ) = Result;
end;
```

Пример 7. Описание табличного и BlackBox отношений.

В примере создаются 2 ограничения:

- таблица, загружаемая из файла «Trains_from_a_to_b» (расписание поездов) и содержащая 2 целочисленных столбца;
- BlackBox-ограничение с основе функции «Calculator» из библиотеки «BlackBoxTest» с 3 вещественными аргументами и вещественным результатом.

В обоих случаях для создания конкретного ограничения с помощью обобщённого отношения, приходится указывать множество дополнительных константных параметров, которые не влияют на дальнейшее использование созданных ограничений. Поэтому, для упрощения их использования, ограничения создаются внутри составных отношений «AB» и «Volume».

4.9. Классы решаемых задач

Базовый алгоритм, реализованный в системе Nemo, служит для решения задачи достижения локальной совместности. С точки зрения пользователя, он находит внешнюю

оценку множества решений ЗУО. Тем не менее, с помощью специализированных ограничений, система позволяет решать следующие задачи:

- Поиск точного решения ЗУО с заданной точностью.
- Поиск точного решения, ближайшего к текущему значению аргументов.
- Поиск нескольких/всех точных решений.
- Поиск частичного решения.
- Решение задач оптимизации.
- Решение задач иерархического удовлетворения ограничений (Partial Constraint Satisfaction Problem).

Кроме того, пользователю предоставляется доступ к некоторым параметрам базового алгоритма, которые позволяют управлять процессом поиска решений. В частности можно задавать следующие параметры:

- Задавать начальное значение переменной (**%***var***=**...).
- Задавать точность найденного решения (**%eps=**...).
- Задавать невязку ограничений для решения (**%discrepancy=**...).
- Задавать число требуемых решений (**%solutions=**...).
- Сохранять лучшее частичное решение (**%partial=**...).
- Включать все переменные модели в механизм перебора с откатами (**%exact=**...).
- Задавать целевое значение переменной (**%***var***~**...).
- Задавать шаг решения по переменной (**%eps**(*var*)**=**...).

```
uses "NemNumbers";
uses "NemAbstractTypes";
main
    x1,x2,x3,x4,x5 : real;
    3*x1*(x2 - 2*x1) + 0.25*x2^2 = 0.0;
    3*x5*(20 - 2*x5 + x4) + 0.25*(20 - x4)^2 = 0.0;
    3*x2*(x3 – 2*x2 + x1) + 0.25*(x3 - x1)^2 = 0.0;
    3*x3*(x4 – 2*x3 + x2) + 0.25*(x4 - x2)^2 = 0.0;
    3*x4*(x5 – 2*x4 + x3) + 0.25*(x5 - x3)^2 = 0.0;
end;
//
%solutions=10
%x1 ~ 0.0
```

```
%x2 ~ 0.0


%partial=On
%discrepancy = 1e-8


%eps(x1) = 0.1
%eps(x2) = 0.1
%eps(x3) = 0.1
%eps(x4) = 0.1
%eps(x5) = 0.1
```

Пример 8. Управление механизмом поиска решений.

В примере заданы следующие параметры алгоритма:

- вычисления останавливаются после нахождения 10 решений («%solutions=10»);
- решения ищутся последовательно, начиная от окрестности целевой точки {x1, x2} = {0.0, 0.0};
- сохраняется лучшее частичное решение («%partial=On»);
- все ограничения удовлетворяются с невязкой не более 1e-8 («%discrepancy = 1e-8»);
- расстояние между найденными решениями будет не менее 0.1 по каждой из переменных.

4. Заключение

В статье описывается система программирования в ограничениях Nemo и её особенности, позволяющие эффективно описывать и решать ЗУО.

Система Nemo предоставляет пользователю широкие возможности:

- для декларативного описания задачи;
- для настройки на предметную область с помощью создания собственных типов данных и ограничений;
- для повышения уровня представления данных с помощью методов объектно-ориентированной и обобщённой парадигм программирования;
- для расширения классов решаемых задач с помощью собственных специализированных ограничений и типов данных;
- для интеграции в состав сторонних систем.

Несмотря на долгую историю развития, возможности дальнейшей модернизации системы далеко не исчерпаны. Благодаря заложенным на этапе проектирования принципам система

Nemo может использоваться как инструмент для решения сложных задач, как среда для разработки новых алгоритмов для решения задач в области ЗУО (например: нахождение 3В-совместности, быстрое решение подсистем линейных уравнений, использование производных при поиске точного решения), как вычислительный модуль в составе сторонних систем.

Список литературы

1. Ботоева Е.Ю., Костов Ю.В., Петров Е.С. Универсальный решатель UniCalc // Информационный бюллетень рабочего семинара «Наукоемкое программное обеспечение», Новосибирск: ИСИ СО РАН, 2006, С.42-45.
2. Булгаков. С.В. Подход к построению мульти-агентной системы содержательного поиска во множестве разнородных структурированных источников данных // КИИ’2004. Труды 9-й национальной конференции по искусственному интеллекту с международным участием. Том 2. Москва: Физматлит, 2004. С. 706-714.
3. Гончар А.М., Загорулько Г.Б., Рубан М.Н., Рябков А.Н. Экспертная система поддержки диагностики, профилактики и лечения элементозов на основе коррекции питания // КИИ’2006. Труды 10-й национальной конференции по искусственному интеллекту с международным участием. Том 3. Москва: Физматлит, 2006. С. 849-857.
4. Загорулько Г.Б., Сидоров В.А., Телерман В.В. и др. НеМо+: Объектно-ориентированная среда программирования в ограничениях на основе недоопределённых моделей // КИИ'98. Шестая национальная конференция с международным участием. Сборник научных трудов в трёх томах. Том I. Пущино, 1998. С. 524–530.
5. Загорулько Г.Б., Сидоров В.А., Телерман В.В. и др. Обстановка для программирования в ограничениях на основе недоопределённых моделей NeMo+ (язык, архитектура, интерфейс). // Научно-техн. отчёт N 7 / Российский НИИ искусственного интеллекта, Институт систем информатики им. А. П. Ершова СО РАН. Москва, Новосибирск, 1998. 107 с.
6. Загорулько Ю.А., Попов И.Г., Костов Ю.В. Интеграция технологии баз знаний с агентной технологией и методами программирования в ограничениях. // Материалы международной научно-технической конференции "Информационные системы и технологии" (ИСТ '2000). Новосибирск: Издательство НГТУ, 2000. Т.3. С. 508.
7. Нариньяни А.С. Недоопределенность в системе представления и обработки знаний // Изв. АН СССР. Техническая кибернетика. 1986. № 5. С. 8-11.
8. Нариньяни А.С. Недоопределённые модели и операции с недоопределёнными значениями // Новосибирск, 1982. 33 с. (Препринт / АН СССР. Сиб. отд-ние. ИЦ; № 400).
9. Сидоров В.А. Программирование в ограничениях с чёрными ящиками. Новосибирск, 2003. 39 с. (Препринт / ЗАО Ледас; №2).

10. Сидоров В.А. Сравнение способов представления неточных значений в методе недоопределённых вычислений. // Системная информатика. 2015. № 6. С. 53-66.
11. Телерман В.В., Ушаков Д.М. Недоопределенные модели: формализация подхода и перспективы развития // Проблемы представления и обработки не полностью определенных знаний / Под ред. И.Е. Швецова. Москва-Новосибирск: РосНИИ ИИ, 1996. С. 7–30.
12. Golomb S.W. , Baumert L.D. Backtrack programming // J. of the ACM. 1965. Vol 12, №. 4. P. 516–524.
13. Huffman D.A. Impossible objects as nonsence sentences // Machine Intelligence. 1971. Vol. 6. P. 295–323.
14. Lipski S., Sidorov V., Telerman V., Ushakov D. Database Processing in Constraint Programming Paradigm Based on Subdefinite Models // Joint Bulletin of NCC&IIS,. 12 (1999), NCC Publiher. Novosibirsk, 1999.
15. Mackworth A.K. Consistency in Networks of Relations. // Artificial Intelligence. 1977. № 8. P. 99–118.
16. Nilsson N.J. Principles of artificial intelligence. // Numerical Computation Guide. Mountain View, USA, November, 1995.
17. Shvetsov I., Kornienko V., Preis S.. Interval spreadsheet for problems of financial planning // PACT`97, England, London, 1997. P. 373-385
18. Sidorov V., Telerman V. Industrial Application of External Black-Box Functions in Constraint Programming Solver. // Perspectives of System Informatics: Proc./ Ed. by M.Broy, A.Zamulin. Berlin a.o.: Springer-Verlag, 2003. P. 415 – 422. (Lect. Notes Comput. Sci.; 2890).