ЛОГИЧЕСКИЕ МЕТОДЫ В ПРОГРАММИРОВАНИИ
(Вычислительные системы)
1990 год Выпуск 133

УДК 519.685

СИГМА-ЯЗЫК

Д.И.Свириденко, С.В.Котов

В в е д е н и е

Данная работа выполнена в рамках проекта СИГМА [4], одной из главных целей которого является создание языковых, методических и инструментальных средств, способных эффективно поддерживать процесс доказательного решения сложных и масштабных логических задач. Напомним, что согласно концепции семантического программирования, данный процесс сводится к установлению истинности логических утверждений на моделях, формально представляющих проблемные области (см. [2]). Синтаксические конструкции системы СИГМА, предназначенные для представления всех необходимых объектов,возникающих в процессе решения логических задач, назовем СИГМА-языком. Как и в любом языке, в его структуре, синтаксисе, семантике и прагматике должна найти свое отражение специфика той проблемной области, на которую ориентирована данная среда. При этом естественно придерживаться следующей последовательности приоритетов указанных выше аспектов: прагматика > семантика > синтаксис.

Прагматика СИГМА-языка естественно определяется указанной выше целью проекта. Таким образом, в нашем случае прагматика языка складывается как бы из двух составляющих. С одной стороны, язык должен предоставлять средства для записи логических спецификаций задач и результатов проектирования их решений в

виде исполнимых спецификаций и/или программ инструментального языка. С другой стороны, он должен содержать языковые средства поддержки самого процесса специфицирования задач и доказательного проектирования их решений. Первый аспект обязывает предусмотреть в языке конструкции, совокупность которых естественно называть языком спецификаций задач (точнее, речь будет идти о нескольких языках спецификаций). Второй же аспект вынуждает ввести в язык конструкции,способные представлять историю воз - никновения тех или иных объектов в процессе решения задач.Данное обстоятельство поставило перед разработчиками нетривиаль - ную проблему совмещения обоих видов конструкций в рамках еди - ного языка. Для ее решения предлагается использовать теорети - ко-модельное виденье тех проблемных областей, для которых предназначен СИГМА-язык. В последующем мы будем говорить об этих проблемных областях как об объектах "мира", а о синтаксических конструкциях, предназначенных для представления этих объектов в СИГМА-языке, как об объектах языка. Таким образом, первый методологический тезис, составляющий концептуальную основу нашего языка,может быть кратко сформулирован так: "объекты "мира" есть многосортные модели языка исчисления предикатов 1-го порядка". Другими словами, предлагается мыслить объекты интере - сующего нас "мира" как многосортные модели. Используя хорошо известный в математической логике механизм элементарной опре - делимости моделей [1,3], можно единообразным образом представлять различные модели, выделив одну из них в качестве основной (базисной), а другие модели описывать в ее терминах. В этом и заключается основная идея спецификации объектов в СИГМА-языке: базисная модель мыслится как совокупность встроенных предика - тов и функций, а другие объекты "мира" определяются в СИГМА-языке с их помощью в виде конечного набора определений предикатов и функций, используя для этой цели определенным образом модифицированные формулы исчисления предикатов 1-го порядка.

Второй тезис нашей методологии звучит следующим образом: "все объекты СИГМА-языка должны быть устроены единообразно". Согласно этому тезису, каждая конструкция, представляющая ("изображающая", специфицирующая) в языке тот или иной объект "мира", должна иметь одинаковую структуру. Более точно, такая конструкция СИГМА-языка будет представлять собой пару: < заголовок, тело > . Тело - это и есть собственно спецификация объекта, а заголовок содержит информацию, доступную (видимую) для других объектов и необходимую для работы с конструкцией. Таким образом, заголовок играет роль оболочки,или, как еще говорят, капсулы, скрывающей содержимое объекта и оставляющей видимым и доступным только то, что разрешено. Средства СИГМА-языка,ориентированные на унифицированную запись заголовков (и тем самым на классификацию спецификаций объектов), образуют язык заголовков.

Существенную часть языка заголовков составляют конструкции, предназначенные для описания взаимодействия объектов. Такие конструкции носят название отношений. Отношения позволяют описывать как процесс конструирования из существующих объектов новых объектов, так и процесс построения сложных иерархических спецификаций, выступающих как единое целое (отношения использования). Кроме того, отношения могут описывать историю возникновения и развития объектов в процессе решения задачи, начиная с момента ее постановки (проектировочные отношения). Нужно отметить, что классификация отношений на отношения использования и проектировочные отношения преследует чисто реализационные цели - с общеметодологической точки зрения они неразличимы.По существу,описание отношения в языке заголовков представляет некоторый механизм взаимодействия объектов.В самом же СИГМА-языке такие механизмы,помимо отношений,могут быть представлены и более непосредственным способом - в виде объектов, называемых стратегиями. Между отношениями и стратегиями существует тесная

связь. С одной стороны, отношения можно рассматривать как описание результатов исполнения механизмов (алгоритмов),представленных соответствующими стратегиями: стратегия (или,более точно, соответствующий ей механизм), будучи примененная к объекту, порождает другой объект (объекты) и тем самым устанавливает определенное отношение между ними. С другой стороны, отношение в момент "исполнения" объекта играет роль команды, запускающей соответствующую стратегию. Данное обстоятельство играет важную системообразующую роль, поскольку совокупность этих команд, вынесенная за пределы СИГМА-языка, позволяет придать ему статус системы.

Механизмов взаимодействия в языке несколько видов. Первый вид механизмов взаимодействия - это элементарная определимость одного объекта в другом. Второй вид, к которому относятся параметризация и импорт, позволяет проектировать спецификации объектов, имеющих сложную иерархическую структуру. Третий вид механизмов предоставляет возможность манипулирования объектами. Тем самым в системе как бы задается своеобразная алгебра объектов, отражающая возможность эффективного использования уже имеющихся спецификаций для конструирования новых. Особенность всех этих трех видов механизмов, представленных в языке заголовков отношениями использования, заключается в том, что все они являются встроенными. Так, например, указание в заголовке объекта того или иного отношения использования при "исполнении" объекта автоматически влечет за собой исполнение соответствующего механизма взаимодействия. Заметим также,что при этом третий и второй виды механизмов выступают в роли управляющих по отношению к элементарной определимости.

Наличие в языке,помимо отношений использования,проектировочных отношений объясняется, как уже говорилось выше, желанием иметь языковую поддержку самого процесса решения логической задачи, начиная с ее постановки и кончая созданием либо

исполнимых спецификаций, либо соответствующей этим спецификациям программы. Как и в случае отношений использования, описания данных отношений представляют в заголовках объектов, с одной стороны, результаты исполнения соответствующих механизмов взаимодействия, а с другой - выступают в роли команд, их запускающих. Эти механизмы определяют конкретные семантические отношения между объектами, возникающими в процессе решения задачи, т.е. в процессе специфицирования, проектирования и программирования. Отметим, что важную часть при проектировании таких отношений составляют конструкции, предназначенные для записи и верификации обоснований (доказательств) правильности проектировочных решений или, другими словами, описывающими свойства возникающих семантических отношений между объектами. Это подмножество языка СИГМА естественно назвать языком обоснований, понимая под обоснованиями широкий класс конструкций от неформальных комментариев до строгих формальных доказательств.

Подытоживая все вышесказанное, можно сказать, что СИГМА-язык, ориентирован на:

- обеспечение единообразности представления возникающих в процессе решения задач конструкций и, в частности, спецификаций задач и программ;

- эффективную реализацию механизма элементарной определимости объектов,составляющего основу исполнимости спецификаций, что, в частности, обеспечивается развитостью базового уровня (т.е. многообразием встроенных конструкций);

- обеспечение широких возможностей параметризации объектов и поддержку связанного с этим свойства модульности объектов и понятия типа объекта;

- предоставление средств работы с библиотекой объектов системы как с иерархической и реляционной базами данных (в частности, реализация механизмов импорта, использования и проектирования);

- предоставление средств для отражения процесса доказательного пошагового решения задач, в частности, средств, базирующихся на механизме проектирования,и средств фиксации и верификации обоснованности проектных решений.

Несколько слов о семантике СИГМА-языка. Из вышесказанного следует, что под математической семантикой данного языка (во всяком случае той его части, которая ориентирована непосредственно на спецификацию задач) естественно понимать соответствующим образом модифицированную теоретико-модельную семантику языка исчисления предикатов 1-го порядка (см.,например, [1]). В таком контексте заголовок объекта языка,представляющего объект "мира", фактически определяет сигнатуру соответствующей модели и ее роль в спецификации задачи и в процессе проектирования,а тело - элементарное определение этой модели в базовой модели.

Поскольку решение задачи в конечном счете мыслится как получение исполнимых спецификаций, то решающее значение для СИГМА-языка приобретает операционная семантика. Для ее пост - роения, во-первых, необходимо указать тот подкласс (конструктивных) спецификаций, которые потенциально могут быть исполнены, а во-вторых, ввести в рассмотрение специальные логико-математические конструкции, в терминах которых будет описан операционный смысл спецификаций. Для того чтобы сохранить концеп - туальную целостность языка,эти конструкции должны,с одной стороны, обладать удовлетворительной теоретико-модельной семантикой,а с другой - иметь отчетливую процедурную интерпретацию.Естественно, что включение этих конструкций в СИГМА-язык существенно увеличивает его выразительную силу.

Цель настоящей публикации - конкретизация высказанных выше положений. Работа состоит из введения, четырех параграфов, заключения и приложения, в котором приводится первая версия синтаксиса СИГМА-языка. При описании синтаксических конструк - ций СИГМА-языка и в статье, и в приложении используются следую-

щие соглашения:

{...} - необязательная конструкция;

$\{...\}^{+}$ - повторение конструкции не менее одного раза;

$\{...\}^{*}$ - повторение конструкции;

| - символ альтернативного определения ;

обычный шрифт - нетерминальные конструкции;

подчеркнутый текст - терминальные конструкции.

§1. Язык заголовков

Как уже было сказано во введении, любая спецификация может быть оформлена как объект СИГМА-языка. Для этого ей сопоставляется заголовок, а собственно спецификация становится телом объекта. Заголовок объекта включает в себя всю информацию, которую автору или самой системе необходимо, а также желательно зафиксировать для работы с данной спецификацией. Следовательно, система при работе со спецификацией не обращается непосредственно к телу объекта (т.е. к тексту спецификации), что защищает его от каких-либо несанкционированных автором изменений. Тем самым фактически обеспечивается модульность объектов языка. В дальнейшем спецификации в явном виде, т.е. в виде текстов, хранящихся в библиотеке системы, мы и будем называть модулями.

Далее мы будем придерживаться положения, согласно которому любая конструкция, хранящаяся или возникающая в системе, или необходимая для ее функционирования, является объектом СИГМА-языка. Поэтому мы должны помимо а) модулей включить в число объектов СИГМА-языка также и следующие классы конструкций:

б) конструкции, представляющие отношения ("отношения"),которые существенно отличаются от модулей тем, что их тела всегда фиктивны;

в) конструкции, возникающие в процессе исполнения спецификаций и программ ("объекты исполнения");

г) конструкции, реализующие программную поддержку процесса проектирования, различные способы тестирования и исполнения спецификаций ("стратегии");

д) объекты операционной системы ЭВМ и инструментальных языков реализации, элементы архитектуры ЭВМ и хранящихся в них данных ("объекты окружения");

Специфика большинства этих объектов в том, что они не могут быть модифицированы пользователем системы (в отличие от модулей). С другой стороны, ясно, что все указанные объекты также можно считать спецификациями (так, например, объекты класса "д" "специфицируют" себя посредством выполнения своих функций). Именно в этом смысле и утверждается, что в системе СИГМА "все есть объект". Рассмотрим теперь подробнее устройство объектов языка, и, в частности, их заголовков.

1.1. Заголовки объектов. Как следует из введения, заголовки объектов являются сферой применения языка заголовков. Мы будем говорить о заголовке (любого!) объекта как о тексте на этом языке. Каковы же основные разделы заголовка объекта?

Во-первых, очевидно, необходимо указать к какому классу ("а"-"д") относится данный объект, а также его персональную идентификацию.

Идентификатор есть слово СИГМА-языка, допускающее однозначное прочтение. Идентификатор может быть связан с некоторым объектом или набором (множеством) однотипных объектов, которые естественно называть значениями идентификатора. Предполагается, что в процессе решения задачи с использованием системы СИГМА значения идентификаторов могут изменяться. Конкретное состояние системы фиксируется совокупностью задействованных в системе идентификаторов и их значений в данный момент времени.

Идентификаторы обладают типом, таким же как и их значения (смотри ниже). Простейшим идентификатором является имя.

Следующие разделы заголовка содержат определение параметризации объекта и указание взаимодействия данного объекта с "окружающим миром". Подробное описание и примеры использования этих разделов приведены в нижеследующих пунктах.

1.2. Параметризация объектов. Параметризация объектов играет двоякую роль. Во-первых, это параметризация спецификаций; такие параметры можно вычислить как "входные", т.е. такие, от заданных значений которых зависит смысл (семантика) спецификации. Во-вторых, мы предполагаем, что объект может определять не только семантику идентификатора объекта, но и вводить новые объекты в поле зрения системы программирования. Такие параметры мы будем называть выходными. Кроме того, зачастую некоторые параметры будут попеременно исполнять как первую,так и вторую роль.

ПРИМЕР 1. Мы хотим описать новый абстрактный тип данных (АТД). Естественно специфицировать его некоторым объектом.Идентификатор объекта будет соотноситься со всем АТД как описанием, а функции и предикаты, которые, собственно, и определяют содержание АТД, должны быть выходными параметрами объекта. Это позволит использовать их как самостоятельные объекты системы, т.е. как отдельные спецификации.

Итак, в СИГМА-языке каждый объект обладает фиксированным количеством параметров (этот набор параметров может оказаться и пустым) и для каждого параметра определена его роль. Следующие служебные слова уточняют эту роль: IN - параметр является входным; OUT - параметр является выходным; UN - роль параметра неопределена и может быть или IN, или OUT в зависимости от контекста использования объекта.

Важное значение в СИГМА-языке имеют ограничения на возможности параметризации объектов. Варианта ограничений два: син-

таксический и семантический. Первый вариант ограничений может быть легко (автоматически) проверен при попытке использования ("вызова") объекта с некоторыми фиксированными объектами в качестве параметров ("данных"). Суть этого ограничения отражена в понятии типа объекта, аналогичного типам данных, процедур и функций в языках программирования.

Ограничения второго варианта ориентированы на семантическую согласованность объектов (вызываемого и данного), являются более сложными и оформляются в виде требований, записываемых в подразделе свойств объекта. Отличие свойств от типов состоит в том, что свойства, в общем случае, не могут быть проверены с использованием только синтаксиса заголовков объектов, а требуют рассмотрения их тел, т.е. собственно спецификаций. Так как в СИГМА-языке допускается неконструктивность спецификаций, то естественно допустить и неконструктивность задания свойств. Мы предполагаем, что в настоящее время наиболее распространенным будет способ использования раздела свойств, как раздела формальных комментариев к спецификации и объекту в целом. В этом смысле интерес представляет также случай,когда тело проектируемого объекта еще пусто, но раздел свойств уже спроектирован. В данном случае естественно рассматривать свойства проектируемого объекта как его формальные спецификации.Заметим, что для объектов, являющихся проектировочными отношениями, содержание раздела свойств можно рассматривать как обоснование правильности соответствующих проектных решений.

Параметричность объекта подразумевает возможность обращения к нему с набором идентификаторов в качестве параметров. Пусть A есть объект с k параметрами и $B_1, \ldots, B_k$ есть идентификаторы, удовлетворяющие требованиям на параметры. Запись $A(B_1, \ldots, B_k)$ ("обращение") обозначает собой спецификацию частного случая объекта A при заданных значениях входных параметров. Заметим сразу, что смысл (семантика) этой

записи зависит от текущего состояния системы. Одновременно это выражение означивает и идентификаторы, сопоставленные выходным параметрам. В случае, когда мы не собираемся использовать какой-то выходной параметр и, следовательно, не заинтересованы в означивании им какого-либо идентификатора, можно использовать фиктивный идентификатор, обозначаемый _. Сопоставление этого идентификатора входному параметру недопустимо.

1.3. Типы объектов. Предполагается следующий подход к типизации объектов системы СИГМА. Будем считать, что каждый тип выделяет из класса всех объектов некоторый подкласс, элементам которого и приписан данный тип. Необходимо сразу заметить, что согласно такому предположению всякому объекту может быть приписано несколько типов. Но поскольку требуется, чтобы мы могли легко проверить согласованность типов непосредственно при обращении к объекту, то крайне невыгодно (а зачастую и невозможно) хранить при объекте все допустимые для него (т.е. содержащие его) типы. Естественным решением выглядит введение частичного упорядочения $\sqsubseteq$ на классе всех типов $\mathbb{T}$, такое что $\langle \mathbb{T}, \sqsubseteq \rangle$ есть полная верхняя полурешетка. Это позволяет для каждого подмножества типов выделить один, наиболее общий тип (наименьшая верхняя грань данного подмножества). Таким образом, каждому объекту можно сопоставить один, наиболее общий для него, тип.

Класс типов $\mathbb{T}$ СИГМА-языка строится конструктивно, начиная с простейших типов, посредством фиксированных конструкторов типов. Для изложения основных идей СИГМА-языка нам достаточно наличия в нем шести простейших и одного базового типа:

а) простейшие типы:

object - тип, приписываемый всем объектам системы (наибольший тип в $\mathbb{T}$); им удобно пользоваться, когда нам неизвестна или безразлична природа какого-то объекта;

module - приписывается всем модулям системы;

data - приписывается всем объектам исполнения;

strategy - приписывается всем стратегиям системы;

relation - приписывается объектам, представляющим отношения;

environment - приписывается объектам из окружения системы (операционная система ЭВМ, инструментальные языки и аппаратная часть);

б) базовый тип:

boolean - тип истинностных значений; в системе есть две стандартные константы, которым он приписан - true и false.

Типы module, data, strategy, relation и environment назовем типами классов.

Из определений вытекает:

$$\text{boolean} \sqsubseteq \text{data} \sqsubseteq \text{object}; \quad \text{module} \sqsubseteq \text{object};$$
$$\text{strategy} \sqsubseteq \text{object}; \quad \text{relation} \sqsubseteq \text{object}; \quad \text{environment} \sqsubseteq \text{object}.$$

Определим теперь два конструктора типов, из описания которых можно извлечь следующую информацию об объекте данного типа: указание класса объекта, количество параметров объекта, роли параметров объекта, требования (ограничения) на параметры объекта. Первый конструктор определяется следующим образом.

Пусть $t_1, \ldots, t_n \in T$ - уже построенные типы, тогда $l[l_1, \ldots, l_n]$ и $[l_1, \ldots, l_n]$ также есть типы, если l есть один из типов классов и каждый l_i есть запись вида $\underline{IN}.t_i$, $\underline{OUT}.t_i$, $\underline{UN}.t_i$. Такой тип может быть приписан объекту класса l (или любого класса, если l опущен) с n параметрами. Роли параметров, как уже было сказано ранее, определяются префиксами IN, OUT и UN. Типы t_i определяют ограничения на параметры.

Из этого определения непосредственно вытекает, что

$$l[l_1, \ldots, l_n] \sqsubseteq [l_1, \ldots, l_n] \sqsubseteq \text{object};$$
$$l[l_1, \ldots, l_n] \sqsubseteq l \sqsubseteq \text{object}.$$

Кроме того, если $t_1^1,\ldots,t_n^1,t_1^2,\ldots,t_n^2 \in T$ и для всех i имеет место $t_i^1 \subseteq t_i^2$, а l_i^1,l_i^2 получены соответственно из t_i^1 и t_i^2 таким образом, что если их префиксы не совпадают, то l_i^2 имеет префикс UN, тогда

$$[l_1^1,\ldots,l_n^1] \subseteq [l_1^2,\ldots,l_n^2],$$

$$l[l_1^1,\ldots,l_n^1] \subseteq l[l_1^2,\ldots,l_n^2].$$

Содержательный смысл типа $l[l_1,\ldots,l_n]$ таков: l указывает только на природу (т.е. класс) объекта и, когда это не имеет значения, l опускается; l_i характеризует i-й параметр объекта. Тип t_i определяет допустимость подстановки некоторого объекта в качестве параметра. Подстановка допустима, если тип t подставляемого объекта находится в отношении $\subseteq$ с типом t_i. Префиксы параметров имеют смысл: если у объекта в описании типа нет префиксов вида UN, то его можно понимать как процедуру со входными параметрами, помеченными IN, и выходными параметрами, помеченными OUT. Выходные значения гарантированно принадлежат указанным типам параметров. Префикс UN означает, что роль соответствующего параметра изначально не зафиксирована и спецификация (тело) объекта позволяет использовать параметр и как входной, и как выходной. Можно сказать, что такой объект имеет несколько процедурных интерпретаций, а какая из них будет использоваться при конкретном обращении к объекту, зависит от контекста обращения (вызова) к объекту.

Поскольку каждый объект есть некоторая спецификация, то разумно потребовать, чтобы некоторый параметр всегда был выходным, однако наше определение допускает и случай, когда такого параметра нет. Для решения этого вопроса устанавливается правило: *если среди префиксов типа объекта не встречается* OUT *и контекст вызова доопределил все параметры с ролью* UN *как входные, то считается, что объект*

обладает (в данном контексте) еще одним параметром типа boolean, *не указанным явно в типе объекта.*

Второй конструктор сложных типов имеет вид $1\langle 1_1,\ldots,1_n\rangle$ и незначительно отличается от первого. В первом случае предлагалось смотреть на спецификацию как на процедуру, которая по набору входных параметров выдавала набор выходных (традиционный для программирования вариант). Однако для работы с такими понятиями, как предикат, нам удобно работать с процедурой, которая может порождать набор (в том числе пустой) результатов (выходных спецификаций) для одного фиксированного набора входных параметров. Это свойство - единственное отличие второго конструктора от первого. Поэтому естественно, что все факты об отношении $\subseteq$ на типах вида $1[1_1,\ldots,1_n]$ переносятся и на типы вида $1\langle 1_1,\ldots,1_n\rangle$. Кроме этого, верно:

$$[1_1,\ldots,1_n]\subseteq\langle 1_1,\ldots,1_n\rangle,$$

$$1[1_1,\ldots,1_n]\subseteq 1\langle 1_1,\ldots,1_n\rangle.$$

ПРИМЕР 2. Константа типа t может быть задана как объект типа $[\underline{OUT}.t]$.

ПРИМЕР 3. Некоторое множество констант типа t можно задать объектом типа $\langle \underline{OUT}.t\rangle$.

ПРИМЕР 4. Функция типа t от n аргументов типов $t_1,\ldots,t_n\in T$ представима объектом типа $[\underline{IN}.t_1,\ldots,\underline{IN}.t_n,\ \underline{OUT}.t]$. Булевозначную функцию можно представить и как объект типа $[\underline{IN}.t_1,\ldots,\underline{IN}.t_n]$.

ПРИМЕР 5. Тип n-местного предиката может быть задан как $\langle \underline{UN}.t_1,\ldots,\underline{UN}.t_n\rangle$, где $t_1,\ldots,t_n\in T$ - типы параметров предиката.

ПРИМЕР 6. Процедуру обращения к внешнему устройству вывода можно определить как объект без выходных параметров. В этом случае значение неявного булева параметра логично интерпретировать как сообщение об успешности выполнения процедуры обращения.

ПРИМЕР 7. Семейство предикатов $\{P_i(x) | t \in I\}$ можно рассматривать как объект типа $\langle \underline{IN}.t_1, \underline{UN}.t_2 \rangle$, где тип t_1 задает возможные значения индекса i, а область предиката задается типом t_2.

Таким образом, два конструктора типов ("функциональный" [...] и "предикатный" $\langle ... \rangle$) позволяют строить иерархию типов из простейших и базовых. Построенные подобным образом типы будем называть конструируемыми. Однако в действительности мы имеем еще один мощный способ построения типов. Рассмотрим объект типа $\langle \underline{UN}.\underline{object} \rangle$. Как указано в примере, это одноместный предикат, область задания которого состоит из всех объектов системы. Следовательно, этот объект задает некоторый новый тип (как совокупность приписанных ему объектов). Если в нашей системе присутствует конструктивная спецификация такого объекта, то можно считать, что соответствующий тип построен и может быть включен в класс T как элемент, меньший типа object. Этот тип может быть вновь использован для определения новых типов с помощью описанных конструкторов. В дальнейшем тип $\langle \underline{UN}.\underline{object} \rangle$ будем обозначать type и называть его специфицированным (в отличие от описанных выше конструируемых типов).

1.4. Отношения и взаимодействия объектов. Взаимодействие объектов в системе может осуществляться для многих целей и несколькими способами. Факт взаимодействия объектов удобно отражать в языке в виде отношений. Многообразие, развитость и удобство работы с отношениями во многом определяет, как станет ясным из нижеследующего, уровень и выразительность СИГМА-языка.

Как уже было сказано во введении, объекты в системе вступают в два вида отношений: отношения использования и проектировочные отношения. К первому виду относятся отношения, которые представляют собой отношения между спецификациями, способствующими исполнению спецификаций и организации более сложных спецификаций, т.е. использованию других объектов языка. К от-

ношениям второго вида - отношения, описывающие историю проектирования (возникновения, происхождения) того или иного объекта языка. Заметим, что, поскольку часто отношение использования фактически соответствует и генезису объекта, провести четкой границы между отношениями использования и проектировочными отношениями трудно. Гораздо проще все отношения в СИГМА-языке делить на встроенные и определяемые, а иногда даже различать встроенный вариант отношения от определяемого. То обстоятельство, что в языке СИГМА имеется возможность доопределять нужные отношения, существенно увеличивает его выразительные возможности как технологического языка, но не языка исполнимых спецификаций. Определение такого отношения оформляется как объект языка типа relation. В качестве значений параметров этого объекта будут выступать те объекты, которые предполагается связать данным отношением. Подобная возможность заставляет нас ввести в язык два варианта описания отношения. Первый вариант описания отношений выглядит так: тот факт, что описываемый объект находится в отношении R с объектами $B_1, \ldots, B_k$, может быть записан как

$$R \ \underline{OF} \ B_1\{\underline{FROM}\ C_1\},\ldots,B_k\{\underline{FROM}\ C_k\}.$$

Другой вариант описания отношений ориентирован как раз на тот случай, когда отношение R представляет собой самостоятельный объект языка. В данном случае описание выглядит так:

$$\underline{IS\_ARG\_OF}\ R(arg1\ \{\underline{FROM}\ C_1\},\ldots,\ argK\ \{\underline{FROM}\ C_k\}).$$

Данная запись утверждает, что описываемый объект является аргументом отношения, представленного объектом с именем R. Заметим, что некоторые из аргументов R могут быть обоснованиями, которые также могут быть оформлены как объекты языка (см. ниже). Напомним, что во втором варианте обоснования могут быть оформлены как раздел свойств отношения R. Смысл конструкции FROM C_1 будет объяснен позже.

Отношения описываются в заголовках объектов: за ключевым словом RELATIONS должна стоять последовательность указаний отношений объекта, разделенных точкой с запятой.Ниже будет более подробно описаны оба вида отношений - отношения использования и проектировочные отношения. Напомним еще раз,что подобное разделение отношений весьма условно и носит чисто реализацион - ный характер.

1.4.1.Проектировочные отношения.Как отмечалось ранее,проектировочные отношения предназначены для языковой поддержки самого процесса решения задач,включая этапы формулировки задач, их специфицирования,последующей конструктивизации спецификаций (если это необходимо) и их анализ, этап исполнения специфика - ций и,наконец, этап получения программ, соответствующих спецификациям . Весь процесс решения задачи представляется в виде специальной конструкции Р-схемы, в которой отражена история появления спецификаций и программ. Р-схему можно мыслить в виде графа, вершинами которого являются объекты, возникающие в процессе решения задачи, а дуги представляют собой те отношения,в которых находятся эти объекты (заметим, что сами отношения также могут быть оформлены как объекты языка,но поскольку их роль заключается именно в фиксации отношений, те естественно их мыслить как дуги Р-схемы).Другими словами, на дуги Р-схем можно смотреть как на объекты языка, представляющие собой фактически запись проектировочных решений. Поскольку в проекте СИГМА [4] реализуется методология доказательного программирования,то помимо записи проектных решений зачастую необходимо фиксировать и обоснования их правильности. Поэтому дуги в Р-схемах могут быть нагруженными или "помеченными", где "метки" - это объекты языка, представляющие обоснования (более правильно смотреть на "метки" все же как на вершины Р-схем). Таким образом, в проектировочных отношениях содержится вся информация о происхожде -

нии данного объекта, о его роли и месте в процессе решения задачи. Используя проектировочные отношения, можно восстановить всю историю возникновения данного объекта, узнать то место,которое он занимает в Р-схеме,представляющей процесс решения задачи,восстановить всю Р-схему или нужные ее фрагменты.Заметим, что при исполнении спецификаций,проектировочные отношения чаще всего игнорируются транслятором языка исполнимых спецификаций.

Поясним теперь смысл конструкции FROM C_i в описаниях отношений:

$$R \text{ OF } B_1\{\text{FROM } C_1\},\ldots,B_k\{\text{FROM } C_k\};$$

и

$$\text{IS\_ARG\_OF } R(\text{arg1 } \{\text{FROM } C_1\},\ldots,\text{ argK } \{\text{FROM } C_k\}).$$

Она указывает на то, что объект B_i принадлежит Р-схеме C_i. Отсутствие в описании данной конструкции означает, что объект берется из текущей Р-схемы и если его там нет, то он ищется в системной библиотеке. Предполагается, что базисные объекты принадлежат системной библиотеке со стандартным именем base.

Несколько слов о встроенных отношениях. Первое отношение - это отношение копирования: IS_COPY OF B {FROM C}. Данная запись утверждает, что описываемый объект есть копия объекта B (из Р-схемы C). Данное отношение может быть как отношением использования, так и проектировочным отношением.

Отношение IS_SCHEME является унарным (т.е. фактически свойством) и означает, что описываемый объект является Р-схемой. Это чисто проектировочное отношение. Р-схемы при желании могут рассматриваться как объекты системы типа environment.

Отношение IS_INITIAL означает, что описываемый объект является новым, что ему в процессе решения задачи, т.е. в Р-схеме, не предшествует никакой другой объект: IS_INITIAL {OF B}. Здесь B - имя Р-схемы. Если в записи данного отношения имя

Р-схемы не указывается, то это означает создание объекта в текущей Р-схеме. Заметим, что если описываемый объект имеет тип environment, то наличие в его разделе RELATIONS записи IS_INITIAL означает создание новой Р-схемы. Заметим, что с точки зрения библиотечной организации системы Р-схема соответствует именованному набору файлов, а вышеприведенная запись - созданию нового файла в данном наборе.

Большую группу отношений образуют отношения модификации. Они подразделяются на две подгруппы. Главным критерием подразделения отношений является семантическая эквивалентность исходного и модифицированного объектов. Первая подгруппа отношений должна гарантировать сохранность семантики исходного объекта и в настоящее время состоит из единственного отношения "быть версией": IS_VERSION OF B {FROM C}. Другая запись этого отношения, используемая в том случае, когда целесообразно помимо указания того объекта B (из Р-схемы C), версией которого является описываемый объект A, указать дополнительную информацию, раскрывающую цель создания версии (скажем, отладочная версия модуля), обосновывающую семантическую эквивалентность A и B и оформленную в виде модуля D, выглядит следующим образом: IS_ARG_OF VERSION имя модуля (A, B, C, D).

Вторая группа модификационных отношений, объединяемая общим названием вариант, более многочисленна. В нее входят такие отношения, как:

а) переименование

IS_RENAMIHG OF B {FROM C} WHERE список переименований;

или

IS_ARG_OF RENAME имя модуля (A,B{FROM C},...);

б) упрятывание

IS_HIDING OF B {FRON C} {WHERE список упрятываний};

или

IS_ARG_OF HIDE имя модуля (A,B{FROM C},...);

в) расширение

IS_EXTENSION_OF B {FROM C};

или

IS_ARG_OF EXTENSION_имя модуля(A,B{FROM C},...);

г) сужение

IS_RESTRICTION OF B {FROM C};

или

IS_ARG_OF RESTRICTION_имя модуля(A,B{FROM C},...);

д) объединение

IS_UNION OF B_1{FROM C_1},...,B_n{FROM C_n};

или

IS_ARG_OF UNION_имя модуля(A,B_1{FROM C_1},...,B_n{FROM C_n};

е) сумма

IS_SUMM OF B_1{FROM C_1},...,B_n{FROM C_n};

или

IS_ARG_OF SUMM_имя модуля(A,B_1{FROM C_1},...,B_n{FROM C_n});

ж) конкретизация

IS_CONCRETIZATION OF B {FROM C};

или

IS_ARG_OF CONCRETIZATION_имя модуля(A,B{FROM C},...);

з) обобщение

IS_GENERALIZATION OF B {FROM C};

или

IS_ARG_OF GENERALIZATION_имя модуля(A,B {FROM C},...).

Содержательный смысл этих отношений достаточно прост и фактически определяется названием. Отметим только разницу между отношениями объединения и суммы, а также между отношениями сужения и упрятывания. Если **A** объявляется как результат объединения **B** и **C**, то заголовок объекта **A** должен мыслится как простое слияние текстов заголовков **B** и **C** (заметим, что при этом могут возникнуть коллизии с одинаковыми именами; предполагается,что по умолчанию приоритет имеют имена объекта **B**).Отно-

шение суммы похоже на отношение объединения, но требуется,чтобы множество имен, используемых в заголовках B и C, не пересекались; если же имеет место обратное, то соответствующие имена объекта C предварительно переименовываются так, чтобы не было коллизии имен (это должно делаться автоматически).

Если A и B находятся в отношении упрятывания, то это означает, что A и B практически идентичны, но в объекте A некоторые конструкции (в сравнении с B) являются "невидимыми" для других объектов. Если же A и B находятся в отношении сужения, то это означает, что некоторые конструкции из B удалены и результатом является объект A .

Несколько слов об отношениях конкретизации и обобщения. В первом случае речь идет о том, что вместо некоторых параметров объекта B подставлены более конкретные выражения и результатом является объект A . А во втором, наоборот, некоторые конструкции из B объявляются параметрами. Отметим, что все модификационные отношения являются в том числе и отношениями использования.

Укажем еще два чисто проектировочных отношения:

и) утончение

IS_REFINING OF B {FROM C};

или

IS_ARG_IN RIFINE_имя модуля(A,B,C,...);

к) абстракция

IS_ABSTRACT OF B {FROM C};

или

IS_ARG_IN ABSTRACT_имя модуля(A,B,C...).

Их смысл достаточно очевиден и соответствует названиям.Заметим, что эти отношения не являются отношениями использования.

И наконец, еще два проектировочных отношения:

л) декомпозиция

IS_ARG_IN DECOMP_имя модуля(A{FROM B};

A_1{FROM B_1},...,A_k{FROM B_k}));

м) композиция

IS_ARG_IN COMP_имя модуля(A{FROM B};

A_1{FROM B_1},...,A_k{FROM B_k}).

Первое отношение говорит, что описываемый объект является одной из декомпозиционных частей объекта A (из Р-схемы B), а второе - что описываемый объект A есть результат композиции объектов $A_1,...,A_1$ (другие объекты могут быть обоснованиями и условиями композиции).

1.4.2. Отношения использования. Отношения использования играют в языке исключительную роль. Главное обстоятельство,заставляющее нас выделить эти отношения в отдельный вид, следующее: отношения использования составляют неотъемлемую часть спецификаций задач (в том числе и исполнимых), поскольку представляют собой средство определения одной спецификации в терминах других; тем самым они должны быть встроенными; более того, они должны быть реализуемыми, т.е. им должны соответствовать конкретные трансляционные алгоритмы, позволяющие исполнять те спецификации, в которых указаны эти отношения.

Простейшим отношением использования и соответствующим ему взаимодействием являются отношение и взаимодействие, называемые обращением к объекту A с согласованными по типам и свойствам аргументами $A_1,...,A_n$. Это взаимодействие уже было описано ранее. Описанный там же механизм параметризации представляет собой средство большой выразительности и гибкости.Однако если мы ограничим себя только этим способом организации сложных спецификаций, нам не избежать ряда трудностей. Например, для использования спецификации A в более сложной спецификации B нам придется передавать первый объект второму как аргумент при обращении. А это ведет к тому, что количество

параметров объекта, каждый из которых необходимо всегда явно указывать при обращении, увеличивается на число объектов, используемых для составления данной спецификации.

Кроме этого, "всеобщность" типа object легко порождает противоречия. Например, пусть тип B (объект типа type) включает в себя объект A , построенный одним из конструкторов и имеющий входной параметр типа type. Тогда обращение A(B) может оказаться неэффективным рекурсивным определением. Описанный ниже механизм импортирования позволяет сократить интенсивность использования обращений, уменьшив тем самым вероятность подобной коллизии.

Отношение импортирования указывает, что объект A импортирует конструкции, специфицированные другими (и обязательно не использующими A , т.е. "более простыми") объектами. Как уже указывалось ранее, объекты, как правило,

а) могут рассматриваться непосредственно как спецификации, и

б) могут специфицировать неявно несколько объектов в качестве своих выходных параметров.

Импортирование спецификации, формализованной в виде "а", т.е. импортирование всего объекта назовем прямым, а по типу "б" косвенным. Далее, необходимо заметить, что мы допускаем параметрическое косвенное импортирование, когда объект косвенного импорта имеет один или несколько входных параметров. Эти параметры могут быть зафиксированы двумя способами: во-первых, некоторым входным параметром объекта A и, во-вторых, объектом, в свою очередь,импортированным.

Описание отношения импортирования в разделе RELATIONS начинается со служебного слова IS_IMPORTING. Далее, через точку с запятой помещается последовательность (порядок существен!) выражений вида:

идентификатор | идентификатор FROM вызов.

Первый вариант соответствует прямому импорту. Идентифицируемый объект должен быть известен системе: модуль, отношение или стратегия из библиотеки системы, известное операционной системе стандартное имя, стандартные объекты инструментальных языков.

Во втором случае требуется, чтобы вызвался объект конструируемого типа вида [...]. При этом в вызове идентификатор должен фигурировать как выходной аргумент, а все входные аргументы должны идентифицироваться именами, импортированными выше, или входными аргументами модуля.

ПРИМЕР 8. Запись

IS_IMPORTING b; c FROM d(c,x); e FROM c(y,e,_);

является допустимым описанием импорта для объекта a , если

1) имена b и d известны системе в данный момент времени;

2) x и y есть входные параметры модуля a ;

3) вызовы объектов c и d выполнены правильно;

4) объекты b,c,d и e не импортируют объект a прямо, косвенно, через третьи объекты и т.д.

Преимущества импортирования, как минимум, таковы:

а) Отношение импортирования зафиксировано в системе, а не возникает лишь в момент работы с объектами, как это имеет место для отношения обращения к объектам. Это позволяет проверить допустимость импортирования до того момента, когда импортирование будет практически использоваться при прототипировании.

б) Жесткие (непараметрические) отношения между объектами позволяют строить иерархические спецификации, причем при проектировании объекта "выше" данного не нужно знать, что находится "ниже" данного.

§2. Язык спецификаций

Главной целью деятельности пользователя системы СИГМА является создание исполнимых спецификаций решаемой задачи. Есте-

ственно, что центральное место в СИГМА-языке должны занимать средства спецификаций задач. Естественно также желание пользователя иметь в своем распоряжении достаточно широкий набор спецификационных конструкций, позволяющий наиболее полно учиты - вать особенности решаемых задач. Именно по этой причине речь идет не об одном языке спецификаций, а о целом их семействе - выбор конкретных спецификационных средств определяется контекстом их использования. Так,например, на этапе постановки за - дачи может использоваться либо проблемно-ориентированный, либо формальный логический язык, либо тот и другой. На этапе формализации постановки задачи также могут использоваться логиче - ские средства различной природы - функциональные, алгебраиче - ские и т.п. В этом смысле СИГМА-язык является открытым языком. Но для того, чтобы сохранялась концептуальная целостность языка,нужно, чтобы выполнялось главное требование - теоретико-модельное "видение мира".

На данном этапе развития СИГМА-языка мы ограничиваемся тремя видами спецификационных конструкций, выбор которых про - диктован особенностями СИГМА-метода [4]:

а) на этапе постановки задачи допускается пользоваться проблемно-ориентированными фрагментами естественного языка:

б) на этапе формализации постановки задачи используется расширенный некоторыми процедурными конструкциями (см. [5]) многосортный язык исчисления предикатов в полном объеме; сама процедура спецификации ориентирована, главным образом, на механизм элементарной определимости одной многосортной модели в терминах другой;

в) на этапе конструктивизации спецификаций используется процедурное расширение языка Σ-определений [5]; в основе процедуры специфицирования лежит механизм элементарной определи - мости.

В перспективе стоит проблема интеграции теоретико-доказательных средств спецификации в СИГМА-язык.

2.1. Исполняемые спецификации. Наряду с задачей обеспечения пользователя средствами для спецификационной деятельности, в системе СИГМА необходимо должна решаться и задача исполнения (практической проверки, прототипирования, тестирования) построенных спецификаций. Понятно, что это выполнимо далеко не для всякой спецификации. Необходимо, как минимум, чтобы удовлетворялись следующие требования:

а) текст спецификации, оформленной в отдельный объект, должен быть написан на одном из строго определенных формальных языков спецификации; желательно, кстати, чтобы этот язык хотя бы частично поддерживал предложенную систему типов и предложенный способ идентификации объектов;

б) должен существовать по крайней мере один алгоритм машинной интерпретации текстов на этом языке, или алгоритм трансляции текстов в некоторый язык, для которого проблема интерпретации решена; суть интерпретации/трансляции должна строго соответствовать интуитивному и математическому смыслу языковых конструкций;

в) все, что импортируется объектом или подается ему в качестве входных параметров при обращении, также должно обладать точным операционным смыслом;

г) транслирующий алгоритм должен уметь обращаться со сложными спецификациями, т.е. с отношениями между объектами.

Требования "а" и "в" обращены непосредственно к пользователю, однако требуют и системной поддержки, а "б" и "г" целиком относятся к разработчикам системы. Системная поддержка требования "а" может заключаться в предоставлении пользователю совокупности объектов, реализующих синтаксический анализ текстов на основных предполагаемых языках спецификации. Это значительно облегчит подготовку как исполнимых, так и неисполнимых спе-

цификаций. Чтобы выполнить условие "в", пользователю по крайней мере необходимо иметь в своем распоряжении набор объектов, заведомо исполняемых и имеющих явный операционный смысл. Для этого в системе должен существовать некоторый базовый уровень, включающий в себя совокупность объектов, достаточную для решения задач в определенной проблемной области. Базовый уровень должен быть выполнен на определенном базовом языке спецификаций, на котором разработчики системы смогут выполнить свои обязанности. Естественно, что система должна содержать в себе транслятор этого языка, равно как и трансляторы из требования "б" и синтаксические анализаторы из "а". Объекты, представляющие трансляторы различного назначения, условно назовем стратегиями. Не забудем и то, что СИГМА-язык одновременно является языком проектирования. Поэтому все средства языка, поддерживающие процесс проектирования, также являются стратегиями.

Следующий пункт кратко конкретизируют наше представление о базовом уровне.

2.2. Базовый уровень системы. Исполняемые спецификации суть те, которые составлены из элементарных конструктивных единиц при помощи определенных конструкторов. Достаточную для работы совокупность таких объектов пользователь не может создать сам, они должны существовать к моменту начала его общения с системой. Такую совокупность объектов мы называем базовым уровнем системы. Он включает в себя:

- средства обращения к операционной системе ЭВМ и, возможно, непосредственно к внешним устройствам ЭВМ;

- набор модулей, реализующих основные типы данных и операции над ними;

- трансляторы с зафиксированных основных языков реализации, например, "Ассемблера", C, "Modula-2" и т.д.

- трансляторы с языков спецификаций (стратегии), и в том числе стратегии, поддерживающие проектировочную деятельность, а

также средства редактирования и манипулирования объектами системы, организации библиотеки модулей и т.д.

Понятно, что базовый уровень должен быть ориентирован на определенную проблемную область и доступные средства реализации. Давать какие-либо универсальные рецепты по организации базового уровня на данном этапе развития языка и в данной статье нам представляется неверным.

§3. Стратегии

Описание содержательных преобразований языковых объектов в СИГМА-языке разрешается оформлять в виде объектов, называемых стратегиями. Таким образом, стратегии - это конструктивные объекты системы, представляющие собой описание алгоритмов:

а) реализации отношений и одновременно исполнения соответствующих команд;

б) трансляции одного языка спецификации в другой;

в) исполнения спецификаций на базовых языках;

г) интерпретационное исполнение спецификаций на различных языках спецификации;

д) преобразования объектов и отношений между ними в процессе решения задач.

Стратегиям как объектам естественно сопоставить типы. Например, в случае "б" это будут типы, определенные функциональным конструктором, с одним выходным и, как минимум,одним входным параметрами (для указания объектов, тела которых есть исходный текст и результат трансляции соответственно); мы можем оформить и параметрические стратегии, увеличивая число входных параметров объекта. В случаях "в" и "г" стратегии,ориентированные на исполнение,могут не иметь выходного параметра, а неявный выходной параметр типа boolean может, например, указывать на успешность завершения исполнения.

Являясь объектами "специального" назначения,стратегии требуют существования в системе отношений для выделения их роли. Ниже мы кратко, скорее в качестве примеров, опишем три отношения в СИГМА-системе.

Пользователь, создав объект, может сообщить системе, во-первых, какой формальный язык спецификаций использован в теле объекта, и во-вторых, считает ли он спецификацию конструктивной, т.е. исполнимой, и какую стратегию он предпочтет для ее исполнения. Отношение вида (или допустимости) связывает объект со стратегией, производящей синтаксический анализ текстов на языке тела объекта ("сканер"). Отношение вида, указанное в заголовке объекта, собственно, и фиксирует формально этот язык. Это отношение можно представлять себе как расширение идентификатора объекта (в традиционном понимании операционной системы ЭВМ), указывающее на назначение объекта в смысле практического использования. Упоминание о виде (допустимости) естественно считать обязательным для любого текста, претендующего на формальность. Запишем отношение вида так:

ADMITTED_BY имя стратегии-сканера.

Для конструктивных спецификаций важно отношение исполнимости, которое связывает модуль со стратегией,способной интерпретировать или транслировать его тело. В тексте заголовка модуля это отношение можно указать так:

EXECUTED_BY список имен стратегий.

Важно заметить, что допустимость и исполнимость, с определенной точки зрения, суть одно и то же отношение. Действительно, любой транслятор подразумевает предварительный синтаксический анализ, а с другой стороны, сканер можно считать интерпретатором. Результат исполнения в этом случае есть выдача сообщения о синтаксических ошибках. В то же время это различные отношения: они различаются как на методологическом уровне (это различие содержится в приведенных определениях), так и на

фактическом - отношение вида связывает модуль с ровно одной стратегией.

В случае развитого СИГМА-языка с большим количеством различных стратегий удобно группировать стратегии,работающие со спецификациями на похожих языках. Так, скажем, можно ввести следующее бинарное отношение. Будем говорить, что стратегии **S** и **T** находятся в отношении "**T** шире **S** ", если любой язык,исполнимый посредством стратегии **S** , исполняется и стратегией **T** . Другими словами, язык стратегии **T** включает в себя язык **S** . Например, компилятор с языка **C** , оформленный в виде стратегии, допускает все модули, тела которых написаны на **C** .Если мы располагаем и стратегиями (компиляторами) для C+ и C++, то они, естественно,находятся в отношении "шире" с компилятором для C.

Оказывается, что через это отношение можно просто описать основные виды стратегий. Оно позволяет также легко ориентиро - ваться во множестве стратегий, что позволяет развить систему умолчаний при описании объектов. Ясно, что отношение "шире" задает частичный порядок на множестве всех стратегий. Рассмотрев подмножество стратегий, исполняющих спецификации на одном языке, легко увидеть, что оно имеет наименьший элемент, который есть не что иное,как допускающая язык стратегия. Часто бывает полезным иметь в системе так называемые стандартные стратегии, которые используются,если отношение исполнимости не указано в заголовке объекта. Естественно, что если в упоминавшемся подмножестве существует ровно один элемент, непосредственно выше сканера (наименьшего элемента), то это и есть стандартная стратегия: отношение "шире" позволяет однозначно определить ее. Интересна и другая возможность вводить умолчания: отношение "шире" фактически распространяет отношение исполнимости. Дей - ствительно, если объявлено, что некоторый объект исполняется стратегией **S** , то,очевидно, этот объект исполняется и любой

стратегией T, которая "шире" S, и, следовательно, в списке стратегий в описании отношений все такие T можно опустить. Более того, так как наименьшая (в смысле отношения "шире") стратегия - сканер обязательно указана в отношении вида, то все (!) описание отношения исполнимости может опускаться. Однако есть и повод оставить его в синтаксисе модуля: указывая не только вид объекта, но и некоторые исполняющие стратегии, пользователь фактически просит систему проверить, правильно ли он осведомлен об отношении "шире".

§4. Язык обоснований

На данном этапе разработки языка СИГМА в роли языка обос - нований используется тот математический и программистский жар - гон, которым пользуются математики и программисты при написании и оформлении своих результатов. Однако в перспективе, по мере развития СИГМА-метода [4], речь должна идти о специальном фор - мальном языке обоснований, который позволял бы, во-первых, автоматическую верификацию доказательных построений, а,во-вторых, в отдельных случаях и автоматическое построение самих обоснований. Следовательно, кроме языка,речь должна идти и о логике обоснований. В отдельной перспективе должен быть поставлен вопрос о логике решения задач.

З а к л ю ч е н и е

Настоящая статья представляет собой первую попытку доста - точно полно и подробно описать идеи и проектные решения,составляющие концептуальную основу СИГМА-языка. Насколько удачна эта попытка - судить читателю. Единственное, что хотелось бы в связи с этим отметить, так это то, что проект СИГМА, в том числе и СИГМА-язык как его составная часть, открыт для критических замечаний и конструктивных предложений.

В заключение мы приведем вариант синтаксиса заголовка модуля, проиллюстрировав его примером. Заголовок модуля выглядит так:

TITLE
имя IS расширенный тип модуля;
PROPERTIES спецификация свойств;
RELATIONS список отношений;
END_OF_TITLE.

Комментарии:

а) Так как тип модуля есть функциональная или предикатная конструкция, то начало заголовка будет выглядеть так:

имя IS module...

Это облегчит чтение заголовка.

б) Понятие "расширенный тип" трактуется так: в синтаксис определения типа дописываются внутренние имена параметров (сразу после типа соответствующего параметра через пробел, см.примеры).

ПРИМЕР 9.

```
TITLE
lists IS module[IN.type:atom;
                OUT.type:list;
                OUT.[IN.list, OUT.atom]:head;
                OUT.[IN.list, OUT.list]:tail;
                OUT [OUT.list]:nil;
                OUT <UN.list, UN.list>: sublist;
                OUT [IN.list, OUT.integer]:lenght];
PROPERTIES.
```

Этот модуль реализует АТД "линейные списки". Тип полиморфен: зависит от входного параметра atom, определяющего тип элементов списков. Объект одновременно специфицирует тип как множество (list), операции (head, tail, nil, length) и бинарное отношение (sublist).

```
RELATIONS
     IS_IMPORTING integer, zero, succ FROM
          integers (integer, zero,_,succ,_,...);
     ADMITTED_BY c_compiler;
     IS_REFINING OF lists_project;
END_OF_TITLE
```

Раздел свойств модуля в данном случае использован для комментариев. Однако если, например, стратегии системы умеют особенно хорошо работать с частичными порядками, то полезно было бы указать свойство

...sublist IS_A_PARTIAL_ORDER_ON list;... .

(пример формальной записи).

Заметим, что для реализации функции lenght импортируется тип целых чисел, константа ноль и функция прибавления единицы из модуля integers. Предполагается, также, что текст модуля написан на C и стратегия компиляции указана в соответствующем разделе RELATIONS. Проектировочное отношение IS_REFINING показывает, что данный модуль получен уточнением модуля lists_project.

Литература

1. ГОНЧАРОВ С.С., СВИРИДЕНКО Д.И. Σ-программы и их семантики //Логические методы в программировании.-Новосибирск,1987. -Вып. 120: Вычислительные системы. -С. 24-51.

2. ГОНЧАРОВ С.С., ЕРШОВ Ю.Л., СВИРИДЕНКО Д.И. Методологические аспекты семантического программирования //Научное знание: логика, понятия, структура. - Новосибирск, Наука.- 1987. - С. 154-184.

3. ЕРШОВ Ю.Л. Динамическая логика над допустимыми множествами //Докл. АН СССР. - Т. 273. - 1983. -С. 1045-1048.

4. СВИРИДЕНКО Д.И. Проект СИГМА. Цели и задачи // Настоящий сб. - С.68-94.

5. Его же. Теория семантического программирования: Автореф. дисс ... докт. физ.-мат. наук: 05:13.11. - Новосибирск, 1989. - 17 с.

Синтаксис СИГМА-языка

Ниже предлагается предварительный вариант синтаксиса СИГМА-языка, единственным языком спецификаций которого является язык Σ-определений [1]. В перспекиве синтаксис языка будет расширяться добавлением новых спецификационных конструкций.

Используется следующий вариант бэкусовой нормальной формы:

{...} - необязательная конструкция,

$\{...\}^{+}$ - повторение не менее одного раза,

$\{...\}^{*}$ - повторение,

| - альтернатива,

нормальный шрифт - нетерминальные конструкции,

подчеркнутый текст - терминальные конструкции.

1. Типы объектов

Типы в системе делятся на простейшие и производные. Первые состоят из типов классов и типа object. Вторые делятся на специфицированные и конструируемые. В свою очередь, специфици - рованные типы подразделяются на базовые и небазовые:

тип_объекта::= простейший_тип | производный тип,

простейший_тип::= тип_класса | object,

производный_тип::= специфицируемый_тип | конструируемый_тип,

специфицированный_тип::= базовый_специфицированный_тип |
небазовый_специфицированный_тип,

тип_класса::= module | data | strategy | environment |
| relation.

Простейшие типы описаны в статье.

1.1. Специфицированные типы. Специфицированный тип задается одним одноместным предикатом или, может быть, предикатом с двумя одноместными процедурами. Предикат задает множество объ-

ектов данного типа в носителе модели, процедуры связывают внутреннее представление объекта типа с представлением в исходном тексте программы. Идентификатором типа является соответствую - щий предикатный символ, например t, процедурные символы для типа t будут input_t и output_t.

1.1.1. Базовые специфицированные типы. Атомарным объектом, или константой, является либо натуральное число (включая ноль), идентификатор этого типа есть cardinal либо булева константа, идентификатор типа - boolean. Основная структура есть список - тип list, его элементами могут быть любые константы, в том числе и списки. Синтаксис констант типов:

булева_константа::= true | false,

число::= {цифра}$^{+}$,

список::= <{константа>{ ,константа}*}> ,

константа::= число | список | булева_константа,

базовый_тип::= cardinal | list | boolean.

Например, < >, <1,2>, <1,< >, <2,3>> - списочные константы.

1.1.2. Небазовые специфицированные типы. Язык допускает существование других специфицированных типов и соответствен - но их констант и определяющих эти типы предикатов и процедур. Эти типы специфицируются объектами типа <un object> (сокращенно type), смысл конструкции изложен в статье.

1.2. Конструируемые типы. Предикаты, функции и процедуры также обладают типом, которые называются конструируемыми:

конструируемый_тип::=

{тип_класса}[{тип_с_префиксом{,тип_с_префиксом}*]} |

{тип_класса}<{тип_с_префиксом{,тип_с_префиксом}*}>,

тип_с_префиксом::= in тип | out тип | in тип.

Длина набора типов с префиксами в скобках [...] или <...> есть арность конструкции.

Например, [in cardinal] - тип процедуры output_cardinal (процедура вывода натуральных чисел).

1.3. Переменные и описания. Всякая переменная описывается идентификатором с приписанным ему типом,где идентификатор есть некоторое слово фиксированного алфавита:

переменная::= идентификатор,

набор идентификаторов::= идентификатор{,идентификатор}*,

описание переменной::= переменная : тип.

Тип переменной x обозначим $\tau(x)$, а ее арность (для предикатных и функциональных переменных) обозначим $\alpha(x)$.

Например,x:cardinal, queue:list, input_list: [out list] - описания переменных:

описание_набора_однотипных_переменных::=

{переменная,}* описание_переменной,

набор_описаний_переменных::=

{описание_набора_однотипных_переменных}

{,описание_набора_однотипных_переменных}*.

1.4. Размеченные типы. Данная конструкция используется при описании объектов языка, она фиксирует локальные имена параметров объекта:

размеченный_тип::= тип : идентификатор |

{тип_класса}[{ размеченный_тип_с_префиксом}

{,размеченный_тип_с_префиксом}*]|

{тип_класса}[{размеченный_тип_с_префиксом}

{,размеченный_тип_с_префиксом}*>,

размеченный_тип_с_префиксом::=

in размеченный_тип | out размеченный_тип |

un размеченный_тип.

Идентификатор после двоеточия в определении размеченного типа будем называть разметкой.

2. Объект

Объект есть самостоятельная единица системы,представляемая в языке заголовком и телом с одинаковым идентификатором - именем объекта:

заголовок::=

title идентификатор is размеченный_тип;

{properties текст;}

{relations список_отношений;}

end_title;

тело::=

body идентификатор;

begin схема;

end_body.

Текст есть конечное слово в фиксированном алфавите. Общее описание компоненты список_отношений дано в статье.

2.1. Схема. Это есть основная спецификационная конструкция, представляющая Σ-схему [1]. Схема определяет сигнатуру схемы, состоящую из предикатных и функциональных символов, из левых частей определения (смотри ниже):

схема::= {определение;}*.

Синтаксическое описание класса "определение" достаточно сложно и требует введения дополнительных конструкций.

2.1.1. Синтаксис терма. Терм есть обладающая типом и множеством переменных (FV) синтаксическая структура одного из следующих видов:

а) константа типа τ есть терм типа τ;

б) переменная x есть терм типа $\tau(x)$, $FV(x)=\{x\}$;

в) $t \equiv f(t_1,\ldots,t_n)$ есть терм типа τ, если $t_1,\ldots,t_n$ - термы и f есть функциональный символ или переменная, такая что $\alpha(f)=n$ и $\tau(f)=[\underline{in}\,\tau(t_1),\ldots,\underline{in}\,\tau(t_n),\ \underline{out}\,\tau]$, причем $FV(t)=\cup FV(t_i)\cup\{f\}$, если f - переменная, иначе $FV(t)=\cup FV(t_i)$;

г) $t \equiv x\ \underline{in}\ l.P$ есть терм типа list, если l - терм, P - формула и $x \in FV(P)\setminus FV(l)$, $FV(t)=FV(P)\cup FV(l)\setminus\{x\}$;

функ.символ::= идентификатор,

терм::=

константа |

переменная |

функ.символ (терм {,терм}*)|

переменная in терм. формула.

2.1.2. Синтаксис формулы. Формула есть обладающее множеством свободных переменных (FV) выражение одного из следующих видов:

а) $F \equiv p(t_1,\ldots,t_n)$ есть формула,если p есть предикатный символ или переменная, такой что $\alpha(p) = n$ и $\tau(p) = [\underline{un}\tau(t_1),\ldots,\underline{un}\tau(t_n)]$, причем $FV(F) = \cup FV(t_i) \cup \{p\}$, если p - переменная, иначе $FV(t) = \cup FV(t_i)$;

б) $F \equiv t_1 = t_2$ есть формула,если t_1 и $t_2 p$ есть термы одного и того же типа: $FV(F) = FV(t_1) \cup FV(t_2)$;

в) F ≡ not A есть формула, если A - формула; FV(F) = = FV(A);

г) F ≡ A or B есть формула, если A и B - формулы; FV(F)= = FV(A) ∪ FV(B);

д) F ≡ A and B аналогично;

е) F ≡ A imply B аналогично;

ж) F ≡ for_every x in l.A есть формула, если A - формула, l - терм типа list, x - переменная некоторого типа данных; FV(F) = FV(A) ∪ FV(l) \ {x};

з) F ≡ exist x in l.A аналогично;

и) F ≡ for_every_sublist x of l.A есть формула, если A - формула, l - терм типа list, x - переменная типа list; FV(V) = FV(A) ∪ FV(l) \ {x};

к) F ≡ exist_sublist x of l.A аналогично;

л) F ≡ for_every x.A есть формула, если A - формула, x - переменная некоторого типа данных; FV(F) = FV(A) \ {x};

м) F ≡ exist x.A аналогично;

н) F ≡ (A) есть формула, если A - формула; FV(V) = FV(A).

Таким образом,

пред.символ::= идентификатор,

формула::=

пред.символ (терм{,терм}*)|

терм = терм|

not формула|

формула and формула|

формула or формула|

формула imply формула|

exist переменная {in терм}. формула|

for_every переменная {in терм}.формула|

for_every_sublist переменная of терм. формула|

exist_sublist переменная of терм. формула|

(формула).

В перспективе возможно использовать инфиксную запись би - нарных и префиксную или постфиксную запись унарных функций,предикатов и процедур.

2.1.3. Синтаксис определения.

Определение::=

пред.символ (набор_описания_переменных) define формула

{where набор_описаний_переменных}|

функ.символ (набор_описания_переменных) =

= описание_переменной define формула

{where набор_описаний_переменных}.

ЗАМЕЧАНИЕ. Все идентификаторы, встречающиеся в "формула", но не встречающиеся в "набор_описаний_переменных", кроме сиг - натуры схемы "пред.символ" и "функ.символ",входят в сигнатуру схемы, содержащей данное определение.

2.2. Пример.

```
title sorting is module [in <un object, un object>: order;
                         out <un list, un list> :quicksort];
```

properties модуль определяет по бинарному отношению order (предполагается линейным порядком) на произвольных объектах би-

нарное отношение на списках, вторая компонента элементов которого есть отсортированный (быстрая сортировка) вариант первой компоненты; импортируются все объекты из модуля lists,исполь - зуются

```
member, conc, cons, head, tail, list;
     relations;
          import from lists;
          import cardinal from cardinals;
end_title.
body sorting;
cut(L:list,A:cardinal,L1,L2:list) define
     L1 = X in L. order(X,A)) and
     L2 = X in L. order(X,A))
where X:cardinal;
quicksort(L,M:list) define
     ((L = < >) and (M = < >)) or
     (cut(tail(L), head(L),L1,L2) and
     quicksort(L1,M1) and
     quicksort(L2,M2) and
     M = conc(M1,cons(head(L),M2)))
where L1,L2,M1,M2:list;
end_body.
```

Поступила в ред.-изд.отд.
4 мая 1990 года