МЕТОДЫ АНАЛИЗА ДАННЫХ
(Вычислительные системы)
1985 год Выпуск III

УДК 519.685

ПРОЕКТИРОВАНИЕ Σ-ПРОГРАММ. ПОСТАНОВКА ПРОБЛЕМЫ

Д.И.Свириденко

Настоящая статья продолжает описание методологии Σ-программирования [6] - нового направления в программировании, ориентированного на решение логических задач - важного класса задач обработки символьной информации. Главным объектом исследования, как следует из названия работы, является процесс проектирования Σ-программ, т.е. процесс написания исходных спецификаций задач и превращение их в Σ-программы. Как было отмечено в [6], разработка какого-либо нового подхода в области программирования существенно зависит от того, какой видится исследователю природа программирования (и как деятельности и как науки, обслуживающей эту деятельность). В настоящее время сложились три точки зрения на природу программирования - ремесленническая, инженерная и логико-математическая [17]. Из описания концепции Σ-программирования видно, что мы отдаем предпочтение последней точке зрения (см., также [13,14]). В частности, в основу предлагаемого ниже подхода к проектированию Σ-программ положен хорошо известный в математической логике тезис - "извлечение алгоритмов из доказательств". Данная работа существенно опирается на идеи, изложенные в работе [9], и развивает их применительно к проблеме создания Σ-программ.

I. Σ-проектирование

Использование идей и концепций математической логики в программировании идет, в основном, по четырем направлениям.

I. Использование формальных языков, разработанных в математической логике, в качестве языков описания (спецификаций) задач и/или представления их алгоритмов-реализаций. Предполагается, что

внедрение и использование подобных языков придает оформлению программ математико-подобный вид, позволяет уточнить их интенсионально-динамические представления, которые в традиционных языках программирования страдают неточностью, двусмысленностью и, как следствие, ошибками в окончательных текстах. Следствием подобного заимствования является появление новых стилей программирования. Примерами могут служить функциональное программирование [3,4,22], абстрактные типы данных [I], языки спецификаций и проектирования [2].

2. Обогащение традиционных языков и методов программирования логико-математическими средствами решения задачи верификации программ. Решение данной задачи осуществляется в формальной системе, т.е. в некотором логическом исчислении, язык которого позволяет выражать условие корректности программ. Так, например, часто используется формализм (логика Хоара), в котором возможны записи вида $\varphi\{P\}\psi$, интерпретируемые следующим образом: если входные данные программы P (написанной в традиционном языке программирования) удовлетворяют условию φ, то ее выходные данные, если они получены, будут удовлетворять условию ψ. Подобное обогащение традиционных программистских формализмов также приводит к появлению новых методов и стиля программирования (см., например, обзор [I5]). Заметим, что данным методам уже свойственно наличие формальных критериев правильности программ и средств ее проверки.

3. Третье направление представляет собой практическую реализацию одного из весьма популярных тезисов в математической логике - "выводимость = вычислимость" (см., например, [I6,2I]). Для данного подхода характерно то, что в качестве программы выступает ее спецификация, представляющая собой специального вида формулу языка некоторого прикладного логического исчисления. Исполнение же такой программы-спецификации (т.е. вычисление) заключается в построении доказательства. Таким образом, от программиста при данном подходе требуется прежде всего умение формулировать (специфицировать) свою задачу в соответствующем формальном логическом языке, а не создавать алгоритм решения этой задачи, который как бы "упрятывается" в спецификацию. Наиболее известной практической системой программирования, реализующей тезис "выводимость = вычислимость" является PROLOG [I0]. Как и любая система программирования, PROLOG обладает и достоинствами, и определенными недостатками. Если первые вытекают из предоставляемой PROLOG'ом возмож-

ности писать декларативные нотации вместо традиционной императивной записи (т.е. "что" нужно решать, вместо того "как" решать задачу), то слабые места связаны в основном со структурными ограничениями на вид PROLOG-программ (так называемые хорновы дизъюнкты) и стратегией построения доказательств (SL-резолюция). Как следствие - значительный класс задач остается либо вне рамок данной системы, либо их спецификация в виде PROLOG-программ представляет собой серьезную проблему. Пытаются расширить возможности данной системы, встраивая в нее новые механизмы (см., например, [II]). Другой путь - это поиск новых систем, реализующих тезис "выводимость = вычислимость" и ориентированных на решение других классов задач. Подобные работы весьма важны и желательны, поскольку их результаты имеют самое непосредственное отношение к созданию новых ЭВМ. Так, например, в некоторых проектах система PROLOG рассматривается как один из кандидатов на концептуальную вычислительную основу ЭВМ 5-го поколения [23]. Но в любом случае остается проблема спецификации сложных и масштабных задач. Другими словами, с каждой подобной системой программирования должна быть связана определенная дисциплина построения логических программ-спецификаций или, как будем говорить далее, методика проектирования программ-спецификаций, концептуально расширяющая возможности такой системы. Однако при этом возникает проблема "правильности" построенных программ-спецификаций.

4. Четвертое направление связано с реализацией основного тезиса конструктивной математики, согласно которому каждое математическое доказательство теоремы существования, осуществленное в конструктивной логике, содержит в себе необходимую информацию для эффективного построения той конструкции, существование которой утверждается в теореме. Другими словами, речь идет о концепции "извлечение программ из доказательств". Возможны различные уточнения данной концепции, но для всех них свойственно то, что корректность программ гарантируется правильностью доказательных построений. При данном подходе, как и в предыдущем случае, деятельности программистов придаются черты математического творчества, поскольку здесь явно постулируется первичность рассуждений и вторичность алгоритмической записи. Концепция "извлечение программ из доказательств", как и концепция "выводимость = вычислимость", дает новые возможности автоматизации процессов построения программ. Так, например, обе эти концепции позволяют избавить прикладного программиста от роли посредника между пользователем и ЭВМ. От пользователя будет тре-

боваться только умение решать свою задачу в терминах соответствующего проблемно-ориентированного логического языка, а в отдельных случаях - уметь только формулировать задачу. Все, что будет требоваться при подходе "извлечение программ из доказательств" от пользователя - это уметь строить конструктивные доказательства, включая умение специфицировать задачи, предоставляя ЭВМ не только возможность извлечь программу и ее исполнить, но, в отдельных случаях, и построить нужное доказательство. Наиболее привлекательным выглядит последняя возможность, что и объясняет тот огромный интерес, который проявляется сейчас к системам, умеющим конструктивные доказательства строить автоматически, автоматически извлекать из них программы и исполнять их (системы с автоматическим синтезом программ). Имеются успешные разработки подобных систем, например, ПРИЗ [I2,20]. Общая черта этой и других практически ориентированных систем в том, что автоматический синтез программ в них осуществляется как комбинация уже разработанных подпрограмм. Кроме того, данные системы (в частности, ПРИЗ) являются, как правило, узкоспециализированными, ориентированными на решение четко ограниченного класса задач. В определенном смысле это их достоинство. Подобная узость существующих систем является естественной платой за эффективность - все универсальные алгоритмы автоматического поиска доказательств крайне неэффективны. Входной язык таких систем позволяет удовлетворительно специфицировать только те задачи, на которые они ориентированы. Таким образом, вне подобных систем, как и при подходе "вычислимость = выводимость", остается довольно-таки значимая совокупность задач. Чтобы охватить эти задачи, опять-таки имеется несколько возможностей. Можно, например, расширять существующие системы автоматического синтеза программ. Так и поступают зачастую. Например, система ПРИЗ сейчас модифицируется: вместо входного языка УТОПИСТ создается новый входной язык НУТ [I9]. Можно создавать подобные системы для новых классов задач (см., например, [5]). Но нужно отметить еще одну отличительную черту существующих систем автоматического синтеза программ - небольшие размеры и простоту решаемых задач. Поэтому, на наш взгляд, вполне разумным выглядит следующий путь - попытаться расширить область применения концепции "извлечение программ из доказательств", пожертвовав, скажем, идеей автоматического синтеза программ по спецификациям, основу которой составляют методы автоматического поиска доказательства. Более точно, речь идет о превращении кон -

цепции "извлечение программ из доказательств" в методику проектирования программ, основу которой уже будут составлять методы человеко-машинного построения доказательств. При этом за человеком остается, естественно, творческая часть такого построения, а за машиной - все рутинные операции по проверке правильности доказательств, заполнению несущественных пробелов в рассуждениях, конкретизации схем, набросков доказательств и т.п. И, конечно, извлечение программ из доказательств и их исполнение также должно остаться прерогативой ЭВМ. При этом в роли программ могут выступать, скажем, Σ-программы [6].

Эти соображения и явились отправной точкой описываемого в настоящей работе подхода к проектированию Σ-программ, называемого в дальнейшем Σ-проектированием.

В соответствии с идеями, высказанными в [I3,I4], разработка подобного подхода должна осуществляться в четыре этапа: прагматический → методологический → теоретический → технологический. Естественно, что одним из главных объектов изучения (помимо понятия "Σ-программа") при разработке Σ-проектирования на первых трех этапах должен быть "процесс Σ-проектирования". На прагматическом и методологическом этапах должно быть создано представление о желаемых свойствах и структуре понятия "процесс Σ-проектирования", а на теоретическом - построена его логико-математическая модель (логика Σ-проектирования). При дальнейшем изучении данной модели важны не только теоретические результаты, но и практические выводы и рекомендации - в конечном счете, нас интересует технологическое воплощение разрабатываемой концепции, т.е. структура и содержание технологических решений в зависимости от рассматриваемого класса задач, решение вопросов планирования, организации и управления проектами, инструментальные средства поддержки Σ-проектирования. Однако прежде чем концепция Σ-проектирования получит свое окончательное технологическое воплощение, она должна пройти экспериментальную проверку для различных классов задач*). Для каждого такого класса логика Σ-проектирования должна быть конкретизирована, наполнена определенным содержанием. Соответствующие рекомендации по конкретизации логики Σ-проектирования также должны составлять часть технологии Σ-проектирования. Главное при таком эк-

*) Наиболее адекватными для этих целей средствами моделирования идей Σ-программирования и, в частности, идей Σ-проектирования, в настоящее время представляются системы логического программирования типа ПРОЛОГ и системы типа ЛАДА. Описание последней представлено в настоящем сборнике.

сперименталном апробировании Σ-проектирования - глубже развить методологическое, теоретическое и практическое представление о процессе построения определенного вида конструктивных доказательств, предписываемых концепций Σ-программирования, т.е. сформировать дисциплину проектирования. Подобные представления должны иметь не только вид технологических рекомендаций, но и, что весьма существенно, быть воплощенными в инструментальные средства, ориентированные на поддержку процесса логического проектирования. Здесь нужно понимать, что инструментальные средства способны вести за собой программистов-спецификаторов, во многом предопределяя успех применения концепции Σ-программирования на практике.

Итак, программа исследований по созданию Σ-проектирования в общих чертах выглядит как:

I) поиск и разработка методологических принципов Σ-проектирования, включая и представление о проектируемых Σ-программах, могущих составить основу логико-математической модели этого процесса; переход к п.2;

2) создание и теоретическое исследование модели процесса Σ-проектирования (логики Σ-проектирования); при осуществлении этой задачи возможны возвраты в п.I для уточнения и модификации методологических принципов; переход к п.3;

3) экспериментальная апробация и анализ концепции Σ-проектирования, включая разработку отдельных инструментальных средств; при осуществлении этого пункта возможны возвраты в п.I и/или в п.2; переход к п.4;

4) разработке методики Σ-проектирования применительно к конкретному классу задач, включая и создание соответствующих инструментальных средств поддержки; переход к п.5;

5) эксплуатация методики Σ-проектирования в реальных условиях; при этом возможны возвраты к п.I и/или к п.2.

В последующих параграфах представлены некоторые результаты исследований по п.I, посвященные отдельным вопросам прагматики и методологии Σ-проектирования.

2. Спецификация задач и проектирование программ

Рассмотрим процесс Σ-проектирования в свете концепции "извлечение программ из доказательства" для некоторого класса к проблемных задач.

Создаваемая программа должна быть адекватна решаемой задаче (иначе она нам просто не нужна). Следовательно, нужно позабо - титься о существовании системы аргументов, у б е ж д а ю щ и х н а с в упомянутой выше адекватности. Подобная система аргумен - тов естественно распадается на две подсистемы: первая отвечает за убедительность обоснования адекватности исходной формулировки ре- шаемой задачи - ее природе и целям, а вторая - за убедительность правильности соответствия программы - исходной формулировке зада- чи (включая и цели проекта). Заметим, что подобное разделение хо- тя и естественно, но носит, как мы убедимся позже, чисто техниче- ский характер - эти подсистемы тесно между собой связаны, так как сделанный нами в §I выбор характера второй системы аргументации ("извлечение программ из доказательств") влечет за собой опреде - ленные требования, которым должна удовлетворять и первая система.

Подчеркнем, что уход от обсуждения проблем, связанных с по - строением системы убедительных аргументов, касающихся адекватно - сти исходных спецификаций, равносилен отказу от ответа на чрезвы- чайно важный и принципиальный для программирования вопрос: от - куда берутся и какова природа исходных спецификаций задач? Ис- ключительность этого вопроса в том, что неумение (нежелание?) от- вечать на него оборачивается в конечном счете невозможностью правильно оценивать практическую реализуемость исходных специфи - каций и преобразовывать их к нужному виду. Следовательно, вопросы построения системы аргументов, обосновывающих адекватность исход- ных спецификаций задач, также необходимо включать в круг интере - сов теории и практики программирования. Естественно, на эти воп - росы должна уметь отвечать и концепция Σ-программирования.

Исходная спецификация задачи и описание целей проектирования является заодно и своеобразным описанием наших представлений о бу- дущей, еще не существующей программе. На процесс проектирования можно смотреть как на демонстрацию того, что такая программа су- ществует. Д р у г и м и с л о в а м и, исходную спецификацию можно рассматривать как своеобразную формулировку следующей задачи существования: "существует ли конструкция (объект) f, обладаю- щая свойством ϕ ?" (или, короче, "$\exists f \phi$?"). В нашем случае под конст- рукцией f понимается Σ-программа. (Задача существования может ока- заться вырожденной, например, когда исходная спецификация задачи уже является Σ-программой.) Итак, процесс проектирования начи- нается с формулировки задачи существования. Мы не оговариваем спе-

циально условие, чтобы полученная в процессе решения задачи "$\exists f \varphi$?" Σ-программа f_0 обязательно была бы воспринимаемой и исполняемой реальной вычислительной машиной. Прежде всего, эта конструкция, как и сама задача, должна быть понимаемой "постановщиком" задачи.

В чем же заключается своеобразие и специфика задач существования в программировании? Поиск ответа на данный вопрос позволит прояснить еще одну важную для нас проблему, касающуюся природы логики проектирования.

К решению проблемы существования, как подсказывает опыт математики, можно подходить с разных позиций. Так, например, в классической логике считается, что объект существует, если предположение о его несуществовании ведет к противоречию. Можно усилить эту точку зрения, считая, что объект существует, если невозможно доказать (в рамках, естественно, некоторой системы знаний или действий) его несуществование. Нетрудно понять, что подобные точки зрения неприемлемы для нас в качестве универсальных конструктивных руководств, поскольку при решении задач придется не только постулировать или доказывать (неявное) существование объектов, но и непосредственно их конструировать. Это не означает, что подобные подходы к проблеме существования вообще неприменимы для нас; задач, в том числе и задач существования, могущих быть удовлетворительно решенными в соответствии с вышесформулированными принципами, в программировании имеется в избытке.

Более привлекательной для нас (и вообще для программирования в целом) представляется точка зрения, согласно которой объект существует тогда и только тогда, когда могут быть указаны инструкции (методика), согласно которым объект может быть построен (в частности, это может быть и тривиальная инструкция, представляющая тождественное действие; в этом случае речь идет о явном указании на объект, например, когда исходная спецификация задачи есть Σ-программа). Однако такой конструктивный подход приводит к необходимости решения другой проблемы, которую можно сформулировать в виде вопроса: почему мы должны верить, что объект, построенный согласно предъявленной методике, будет искомым, нужным нам объектом? Другими словами, речь вновь идет об обосновании, т.е. о системе аргументации, убеждающей нас в том, что построенный объект будет обладать требуемыми свойствами. Ориентация на тезис "извлечение программ из доказательств" дает недвусмысленный намек на то, что

подобной системой аргументации будет служить доказательство исходной спецификации "$\exists f \varphi$?" в некоторой подходящей конструктивной системе. (Для исходной спецификации, уже являющейся Σ-программой, требуемой аргументацией будет ее самоочевидность.)

Прежде чем воспользоваться некоторой методикой построения объектов, писать и воспринимать обоснование корректности процесса построения, мы должны быть убеждены в том, что понимаем смысл утверждения: "объект (конструкция) обладает требуемыми свойствами", и чтобы двигаться дальше, должны предварительно решить проблему осмысленности как исходных спецификаций, так и проектируемых программ.

Говоря о существовании объекта, формулируя и решая соответствующую задачу существования, мы предварительно должны уяснить для себя, как будут при этом использоваться привлекаемые понятия, факты, конструкции (т.е. релевантные задаче знания), в том числе и сам объект. Другими словами, проблема существования в программировании - это проблема использования понятий. Хорошей иллюстрацией этого тезиса может служить проблема существования короля в шахматах - мы распознаем некоторую фигуру, как короля, если осознаем (понимаем), что эта фигура фактически есть символ (знак), играющий определенную роль в шахматной игре, подчиняющийся определенным знакам функционирования*). Смысл символов, знаков (это в полной мере относится и к символам, используемым в программировании и в математике) зачастую можно свести к правилам их использования (в рамках, естественно, соответствующей системы знаний и/или действий). Именно поэтому, приступая к формулировке и последующему решению задачи "$\exists f \varphi$?", мы должны прежде всего позаботиться о ее осмысленности, т.е. задаться вопросом: знаем ли мы зачем нам нужна Σ-программа f, когда, как и где мы собираемся ее использовать? Нетрудно понять, что если мы по исходной спецификации смогли найти ответы на эти вопросы и полученные ответы совпадают с первоначальными целями, то тем самым мы можем считать, что исходные спецификации адекватны поставленным целям. Следовательно, удовлетворительное решение проблемы адекватности исходных спецификаций, как и проблемы существования, предполагает удовлетворительное решение проблемы осмысленности. С этой целью мы будем придер-

*) Этим обстоятельством зачастую пользуются шахматисты при неполном наборе шахматных фигур.

держиваться следующего методологического принципа (критерий осмысленности) [9], согласно которому:

> некто имеет право считать исходную спецификацию осмысленной, а следовательно, и адекватной, если он убежден, что располагает реальной процедурой, которая способна отличать решение задачи "$\exists f \varphi$?" от нерешения, коль скоро оно предъявлено.

Проанализируем данное положение. Если спецификация задачи такова, что одновременно является и явным описанием искомой конструкции f, то принимаемый тезис может рассматриваться как методологическое определение самоочевидности. Несколько слов об убежденности, о которой идет речь в критерии осмысленности. Нетрудно понять, что эта убежденность подкрепляется, главным образом, привлекаемыми для постановки задачи "$\exists f \varphi$" - релевантными знаниями - контекстом, в котором (и с помощью которого) происходит понимание смысла формулировки данной задачи. Предположим, что мы имеем в своем распоряжении некую совокупность знаний и собираемся ее использовать с целью сформулировать или понять какую-то конкретную задачу "$\exists f \varphi$?". Согласно критерию осмысленности, данная совокупность знаний должна также быть в состоянии объяснить нам те действия (или, во всяком случае, результаты действий), которые будут предприниматься в связи с решением этой задачи. Отсюда следует, что данная совокупность релевантных задаче знаний должна быть непротиворечивой, ибо в противном случае с точки зрения этих знаний было бы допустимым (осмысленным) любое действие. Ясно также, что релевантные знания должны производить впечатления убедительно обоснованных. Это требование к системе релевантных знаний является необходимым, так как сомнение в истинности или ложности привлекаемых знаний равносильно сомнению в осмысленности формулировки самой задачи (эту осмысленность обеспечивают именно релевантные знания). Если же некто сомневается в осмысленности исходных спецификаций, то это означает, что он просто недопонимает саму исходную задачу и поставленные цели. Обычно убедительная обоснованность знаний достигается на пути выделения некоторых принципов, положений или фактов, которые производят впечатление непосредственно очевидных (самоочевидных), и последующего оформления всей совокупности знаний как единой и обозримой схемы, логически вытекающей из этих первоначальных положений. Другими словами, речь идет

о соответствующим образом обоснованной структуризации релевантных знаний. Помимо всего вышесказанного, релевантные знания должны быть удобными в использовании и полезными, что, опять-таки, находит свое отражение, главным образом, в структуре знаний: полезные и удобные релевантные знания должны своим видом, т.е. структурой, подсказывать, направлять, предопределять тот путь, по которому нужно вести поиск решения задачи. Очень часто имеет важное значение и удачное представление формулировки задачи, которое обеспечивается, в свою очередь, опять-таки релевантными знаниями.

Убедительная обоснованность и, следовательно, хорошая структуризация знаний обычно эксплицируется в виде формальной теории (модели) вместе с обоснованной содержательной интерпретацией этой теории (модели). Далее будем называть теорию, релевантную задаче, r-теорией. В отличие от традиционного взгляда на формализацию знаний, относительно r-теорий не оговаривается такого свойства, как минимальность, т.е. независимость ее аксиом. Наоборот, избыточность r-теории (но не излишняя детализация) здесь только полезна.

Когда говорят о формализации, то подразумевают прежде всего математику. Однако экспликация релевантных знаний в виде формальных теорий при Σ-проектировании программ является также вполне естественной и даже, более того, неизбежной. Вообще говоря, формализация играет чрезвычайно важную роль в концепции Σ-проектирования. Прежде всего, поскольку проектируемая в рамках Σ-программирования Σ-программа является формальной конструкцией, то и доказательство, из которого будет автоматически извлекаться искомая конструкция, также должна быть формальной структурой. Реальная особенность процесса Σ-проектирования - сложность и масштабность решаемых задач, а следовательно, и формальных конструкций, возникающих при Σ-проектировании. Это обстоятельство может вынуждать проектировщиков-спецификаторов прибегать к коллективной деятельности, что, естественно, привнесет в процесс Σ-проектирования дополнительные организационные трудности.

Кроме масштабности и сложности проектируемых Σ-программ, необходимо учитывать то обстоятельство, что эти формальные структуры, хотя и имеют дескриптивный вид, в то же время и интенсиональны: в конечном итоге, они должны будут "исполняться" Σ-машиной. Обычно считается, что языки математической логики, являясь языками для формального представления суждений (описывающих некоторый

"мир") и их доказательств, предназначаются в основном для представления статичных аспектов исследуемых явлений, в то время как программы - это, дескать, конструкции, представляющие динамические сущности, преобразующие некий "мир" ("мир" состояний памяти ЭВМ, например). Подобное мнение о существующем "принципиальном" различии между динамикой программистских конструкций и статикой логических структур представляется неверным. Главное, чтобы формализм, в терминах которого мы собираемся специфицировать (и решать) задачи, должен позволять удобно и полно отражать как динамические, так и статические аспекты проблемной области. Отличительной особенностью программистской деятельности и конструируемых при этом объектов является их "неизбежная реальность" - проект должен быть закончен в отведенные ему реальные сроки, созданные конструкции должны вызывать в реально приемлемое время нужные изменения состояний памяти реальных вычислительных машин, "быть уложенными" в реальную память ЭВМ. Другими словами, роль ресурсных ограничений при создании программ несравнима по своему значению с их ролью при доказательстве теорем в математике. (Заметим, что когда речь шла о реальности процедуры в критерии осмысленности спецификаций, то подразумевались именно ресурсные ограничения.)

Подводя итоги обсуждению проблемы адекватности спецификаций и связанных с ней проблем существования и осмысленности, мы уже можем сделать вывод, что для того чтобы концепция "извлечение программы из доказательства" действительно могла бы стать научно-обоснованной методикой Σ-проектирования, необходимы серьезные усилия и исследования во многих направлениях, в том числе и по созданию соответствующих поставленным целям логико-математических формализмов. Ниже мы обсудим желаемые характеристики формальных средств, в рамках которых будет осуществляться спецификация и решения задач (из некоторого класса $\mathbb{K}$). Ради краткости обозначим совокупность этих средств через $Syst(\mathbb{K})$.

Все вышесказанное, естественно, приводит к выводу, что $Syst(\mathbb{K})$ нужно мыслить как теорию, представляющую релевантные знания, но уже не отдельной задаче, а всему классу $\mathbb{K}$. Релевантность $Syst(\mathbb{K})$ классу $\mathbb{K}$ заключается в ее полноте. Однако речь идет не о традиционной логической полноте формальных систем, а о несколько другом свойстве. Прежде всего мы потребуем, чтобы $Syst(\mathbb{K})$ была спецификационно полна относительно класса $\mathbb{K}$, что означает возможность адекватно специфицировать любую задачу из класса $\mathbb{K}$. Таким обра-

зом, Syst(K) можно рассматривать как формальное представление (модель) той проблемной области, к которой относится класс исходных задач, ассоциированный с рассматриваемым классом K задач вида "∃fφ?". Ранее, обсуждая проблему осмысленности, мы поняли, что существенной компонентой задачи является ее r-теория. Но если от r-теории потребовалась непротиворечивость (и это было существенно), то от Syst(K) мы этого свойства не требуем. С нашей точки зрения, выполнение этого свойства для Syst(K) является несущественным, т.е. допускаются к рассмотрению и использованию в качестве систем спецификаций и противоречивые системы. Главное здесь, чтобы извлекаемая из Syst(K) часть знаний, относящихся к задаче, в виде r-теории была непротиворечивой.

Выше выдвигалось также требование, чтобы полученная в процессе решения задачи "∃fφ?" Σ-программа была в принципе понятной пользователю (этого требует критерий осмысленности исходных спецификаций). Но это возможно, если эта конструкция записана в понятных для него терминах соответствующей предметной области. Поэтому мы потребуем от Syst(K) так называемой реализационной полноты относительно класса K: решение каждой задачи из класса K, если оно существует, должно быть также выразимо в языке Syst(K). Таким образом, частью Syst(K) будет система программирования, в терминах которой пишутся (извлекаются) программы, являющиеся решением задач, т.е. Σ-язык.

Говоря о решениях задач, мы не должны забывать и об аргументации (а как мы знаем, речь идет о доказательствах) - она также должна быть представима в Syst(K). Прежде всего предстоит ответить на очень важный для Σ-проектирования вопрос: какие доказательства считать конструктивными; а также решить проблему извлечения программы из доказательств в виде программ-предикатов, а не программ-термов, как обычно делается, когда речь идет о функциональном (а не Σ-) программировании.

В силу определенных причин (см., например, [I3, I4]), мы останавливаем свой выбор на системах натурального вывода. Но, естественно, что для наших целей эта система может быть соответствующим образом модифицирована. Поскольку любое конструктивное доказательство содержит гораздо большую информацию, чем необходимо для выполнения извлекаемой программы, то целесообразно все преобразования над решениями задач (модификацию, адаптацию, оптимизацию и т.п.) осуществлять не на уровне извлекаемых конструкций, а на уров-

ще соответствующих им доказательств. Следовательно, Syst(K) должна обладать развитым аппаратом преобразований доказательств.

Говоря о свойствах Syst(K) , мы ничего до сих пор не говорим о самом процессе проектирования, а ведь именно он представляет для нас главный интерес. Поскольку проектирование начинается со спецификации задачи, то на процесс спецификации необходимо смотреть как на часть проектировочного процесса: постановка задачи "Зf? " столь же важна, как и ее решение. Нетрудно понять, что в действительности только немногие задачи допускают вначале точную их постановку, а затем решение. Как правило, эти процессы (спецификация + решение = проектирование) тесно переплетены, и невозможно точно указать ту границу, где кончается специфицирование и начинается собственно проектирование. Если Syst(K) представляет средство спецификации задач, то естественно потребовать, чтобы она поддерживала как процесс спецификаций, так и весь процесс проектирования. Данное требование мы сформулируем как свойство проектировочной полноты, обобщающее ранее сформулированные свойства спецификационной и реализационной полноты: каждая задача из класса K может быть адекватно специфицирована и решена в системе Syst(K).

На этом мы закончим первичный анализ проблемы спецификации задач и проектирования Σ-программ. Как мы убедились, в формальной системе, предназначенной для этих целей, понятие доказательства играет одну из центральных ролей. Но при этом, в отличие от традиционных логических систем, данная система должна позволять не только создавать отдельные доказательства, но и выполнять преобразования над ними. Учитывая данное обстоятельство, а также ранее выдвинутые принципы и требования, которым должна удовлетворять такая система Σ-проектирования, можно сделать вывод, что формализация подобной системы будет существенно отличаться от существующих формальных систем.

Что же касается логики Σ-проектирования, т.е. логико-математической модели систем проектирования типа Syst(K) , то она должна позволять не только говорить о спецификациях, доказательствах и операциях над ними, но, что очень важно (для этого фактически и нужна данная модель), рассуждать об этих операциях. И, самое главное - логика Σ-проектирования должна позволять рассуждать о процессах Σ-проектирования, поскольку это ее главный объект исследований. Данное обстоятельство также делает ее непохожей на сущест-

вующие формальные системы и ставит проблему поиска соответствующего данной проблеме стиля формализации - языка и теории, в которой бы могла быть сформулирована и исследована логика проектирования.

3. Понятие Р-процесса*)

Мы уже выяснили, что процесс проектирования в концепции Σ-проектирования начинается с построения исходных спецификаций и заканчивается созданием доказательства - прообраза будущей Σ-программы. Предполагается при этом, что вся проектировочная деятельность разворачивается в формальной среде, определяемой некоторой системой проектирования вида Syst(K).

Представляется весьма естественным описывать процесс Σ-проектирования в терминах основных этапов решения задач вида "$\exists f\varphi$?", рассматривая этот процесс в виде последовательности (серии) уточняющихся семантических представлений данной задачи (и, возможно, некоторой дополнительной информации, описывающей переходы от одного представления к другому) в том порядке, в котором они (представления) возникают из наших начальных представлений о задаче, о плане ее возможного решения до окончательного результата в виде доказательства. Другими словами, процесс Σ-проектирования - это история процесса решения задачи, начиная с ее постановки, или, как говорят в программировании, жизненный цикл (задачи). При таком понимании процесса проектирования само Σ-проектирование выступает как совокупность знаний, методов и соответствующих им инструментов, поддерживающих развитие процессов Σ-проектирования, их преобразования (о них будет идти речь ниже) и рассуждения о них, а методика (технология) Σ-проектирования - это искусство и наука реализации концепции Σ-проектирования применительно к конкретной ситуации.

Общая направленность всего проектировочного процесса от абстрактного к конкретному ориентируется при этом на эффективность конструкций при сохранении их понятности. Реализация этого процесса всегда связана с поиском компромиссов, тупиками и ошибками. Поэтому представляется целесообразным знать генезис, происхожде -

*) Это понятие во многом похоже на понятие "программа", введенное Ю.Л.Ершовым в его лекции, прочитанной на советско-болгарском семинаре по математической логике (Новосибирск, 1982). Кроме того, понятие Р-процесса близко по своему характеру процессу доказательного программирования, рассмотренному в [18].

ние конкретных воплощений абстракций, т.е. нужно каким-то образом хранить информацию об абстрактном и процессах его конкретизации. Эту функцию и будет осуществлять понятие "процесс Σ-проектирования", понимаемое нами как последовательность описаний промежуточно-устойчивых семантических представлений задач (в роли которых будут выступать фрагменты доказательств) вместе с описанием тех операций (проектировочных решений), которые преобразуют предыдущие представления в последующие. Данные семантические представления задачи далее будем называть состояниями процесса Σ-проектирования, а проектировочные решения - операторами перехода. При такой идеализации процесс проектирования можно мыслить себе как формальную конструкцию, состоящую из состояний и операторов перехода. Будем также различать две сущности - идеальный процесс проектирования (iP-процесс) и реальный (rP-процесс) - прообраз идеального и потребуем от системы проектирования Syst(K) наличия инструментальных средств фиксации iP-процессов и манипулирования ими*).

Нетрудно понять, что введение понятия iP-процесса позволит существенно повысить уровень аргументированности адекватности исходных спецификаций задач и их решений, так как в данном понятии представлен процесс возникновения соответствующих конструкций-спецификаций и доказательств. Нам представляется, что имеется и другое практическое достоинство понятия iP-процесса, которое связано с возможностью в рамках концепции Σ-проектирования удовлетворительно подойти к решению проблемы адаптации и модификации Σ-программ. В терминах iP-процессов эта проблема решается следующим образом: модификация старой Σ-программы в новую осуществляется тогда, когда в iP-процессах для новой и старой программ имеется нечто общее - например, имеются общие состояния или фрагменты состояний. Желательно, чтобы в системах Σ-проектирования имелись инструментальные средства, поддерживающие деятельность по внесению изменений в состояния iP-процессов и по возможности автоматического их распространения на последующие состояния - в этом видится решение проблемы автоматизации деятельности по адаптации и модификации Σ-программ. Нужно сразу заметить, что при таком взгляде на проблему модификации и адаптации желательна локализация вносимых

*) Более того, напрашивается мысль о встраивании в Syst(K) средств, автоматически "очищающих", "проверяющих", "дополняющих" и "утоняющих" rP-процессы и превращения их в iP-процессы.

изменений - изменения, касающиеся некоторого фрагмента состояния должны быть по возможности локализованными и в последующих состояниях при их распространении, т.е. касаться по возможности только образов данного фрагмента. Требование локализации изменений приводит к очень интересной и сложной проблеме модуляризации спецификаций и доказательств (т.е. состояний iP-процессов) и структурной взаимозависимости состояний iP-процессов*).

Выясненная нами роль понятия iP-процесса в модификации и адаптации программ позволяет представить еще одну очень важную и интересную проблему - проблему накопления и использования опыта решения задач. Подобный опыт можно было бы фиксировать в виде системы схем iP-процессов. Не уточняя в данной статье содержания этого понятия, объясним его смысл через его роль в концепции Σ-проектирования - решение конкретных задач могло бы заключаться в отдельных случаях в конкретизации имеющихся схем iP-процессов. Как известно, стратегия декомпозиции задач и сведение их к уже решенным, поиск аналогий в постановках задач является одной из фундаментальнейших стратегий решения задач. Главными механизмами при этом будут являться операторы абстракции и конкретизации.

Помимо этих и ранее упомянутых операторов, важное значение в логическом проектировании в связи с ориентацией на систему натуральных выводов будет иметь оператор введения допущений. Применительно к тем состояниям iP-процессов, которые являются фрагментами доказательств, этот оператор определяет шаги дальнейшего развития доказательства, состоящего в поиске подходящих правил вывода: "допустим, что мы умеем решать задачи $\mathcal{A}_1, \ldots, \mathcal{A}_n$, тогда решение задачи $\mathcal{A}$ в терминах решений задач $\mathcal{A}_1, \ldots, \mathcal{A}_n$ заключается в следующем....". Что же касается состояний iP-процесса, являющихся спецификациями, то оператор введения допущений есть не что иное, как оператор введения понятий (определений): "пусть S_1 обозначает ..., S_2 есть ..., ..., S_m определяется как ...; тогда в этих терминах задача (понятие) $\mathcal{A}$ формулируется следующим образом...". Нужно отметить, что оператор введения допущений имеет самое непосредственное отношение к проблеме модуляризаций состояний iP-процессов.

Итак, мы выявили три типа концептуальных операторов, определяющих как шаги переходов от одного состояния iP-процесса к дру -

*) Можно требовать, скажем, гомоморфности структур состояний iP-процесса и, в частности, окончательного доказательства и извлекаемой из него Σ-программы [I4].

гому, так и манипуляции над самими iP-процессами в целом: абстракции, конкретизации и введения допущений. Естественно, что необходимы дальнейшие уточнения и исследования этих операторов, а также поиск новых. Кроме концептуальных операторов в Σ-проектировании, большая роль будет отводиться и чисто формальным операциям типа правил вывода, упрощению формул, нормализации термов и доказательств, а также операторам подрезки и чистки доказательств и т.п. Все это и должно составить в будущем арсенал инструментальных средств и методов Σ-проектирования.

4. Заключение

Методологические результаты, приведенные выше, представляют собой только первые шаги на пути к методологии Σ-проектирования. И хотя построение методологии еще далеко не закончено, можно уже выделить некоторые особенности будущих систем логического проектирования, делающих их так непохожими на традиционные формализмы как математической логики, так и программирования. Сформулируем еще раз выявленные в процессе анализа основные структурные моменты, касающиеся систем вида Syst(K):

I. Syst(K) должна содержать теоретические знания, которые можно рассматривать как формально-теоретическое представление предметной области, релевантное классу K интересующих нас задач; в рамках этого представления для каждой задачи из K должна существовать возможность сформулировать ее r-теорию.

2. Syst(K) должна содержать в себе развитые декларативные и императивные средства описания знаний, действий и конструкций.

3. Доказательства образуют в Syst(K) отдельную синтаксическую категорию; в связи с этим в Syst(K) должны быть развитые средства манипулирования ими.

4. Синтаксическую категорию в Syst(K) образуют также iP-процессы; следовательно, как и в случае 3, Syst(K) должна также содержать операторы манипулирования iP-процессами.

5. Syst(K) должна поддерживать реальный процесс проектирования, при осуществлении которого большое место отводится процедурам интерактивного взаимодействия системы и проектировщиков, а также процедуры "чистки" и "дополнения" rP-процессов с целью их превращения в iP-процессы.

6. Syst(K) должна позволять хранить, пополнять, систематизировать и модифицировать программистский опыт с целью его исполь -

зования при решении задач (в частности, при модификации и адаптации Σ-программ); решение этой проблемы мыслится в виде организации базы проектировочного знания, основу которой будет составлять структурированная совокупность абстрактных схем iP-процессов.

Дальнейшие методологические исследования будут посвящены уточнению уже выявленных особенностей систем Σ-проектирования и поиску новых. Но прежде всего необходимо уточнить понятие iP-процесса и связанных с ним операторов, в частности, их структурные аспекты.

В заключение автор выражает благодарность К.Ф.Самохвалову и С.С.Гончарову за плодотворные дискуссии.

Литература

I. АГАФОНОВ В.Н. Типы и абстракция данных в языках программирования. -В кн.: Данные в языках программирования. М., 1982, с. 265-327.

2. АГАФОНОВ В.Н. Языки и средства спецификации программ. -В кн.:Требования и спецификации в разработке программ (сборник статей). М., 1984, с.285-344.

3. BACKUS J. Can programming be liberated from the von Neumann Style? A functional style and its algebra of Programms.-Comm. ACM,1978,v.21,p.613-641.

4. BAUER E.L., WÖSSNER H. Algorithmic Language and program development.- Berlin-Heidelberg-New York: Springer-Verlag,1982. - 497 p.

5. БЕЛЬТЮКОВ А.П. Язык дедуктивного программирования.-В кн.: Теория языков программирования, Ижевск, 1983, с.3-18.

6. ГОНЧАРОВ С.С., СВИРИДЕНКО Д.И. Σ-программирование. В кн.: Логико-математические основы проблемы МОЗ (Вычислительные системы, вып. 107). Новосибирск, 1985, с.3-29.

7. ГРИСС Д. Наука программирования. -М.: Мир, 1984. - 416 с.

8. ДЕЙКСТРА Э. Дисциплина программирования. -М.: Мир, 1978. - 276 с.

9. ЕРШОВ Ю.Л., САМОХВАЛОВ К.Ф. О новом подходе к философии математики. -В кн.: Структурный анализ символьных последователь - ностей (Вычислительные системы, вып. 101). Новосибирск,1984,с.141-148.

10. CLOCKSIN W.F., MELLISH C.S. Programming in Prolog.- Berlin-Heidelberg-New York: Springer Verlag,1981.- 327 p.

11. KOWALSKI R. Logic Programming. - In: Information Processing 83 (IFIP-83). Amsterdam, North-Holland,1983, p.133-145.

12. КЯХРО М. и др. Инструментальная система программирования для ЕС ЭВМ (ПРИЗ). - М.: Финансы и статистика, 1981.

I3. НЕПЕЙВОДА Н.Н., СВИРИДЕНКО Д.И. Логическая точка зрения на программирование. - Новосибирск. Б.и. - 1981. -49 с. (Препринт/ ИМ СО АН СССР: T1).

I4. НЕПЕЙВОДА Н.Н., СВИРИДЕНКО Д.И. Программирование с логической точки зрения. -Новосибирск. Б.и. - 1981. - 51 с. (Препринт/ ИМ СО АН СССР: T2).

I5. НЕПОМНЯЩИЙ В.А. Практические методы верификации программ. -Кибернетика, 1984, №2, с.21-28.

I6. САЗОНОВ В.Ю., СВИРИДЕНКО Д.И. Абстрактная выводимость и теория областей. -Тез. VII Всесоюзной конференции по математической логике. Новосибирск, 1984, с. 158.

I7. СВИРИДЕНКО Д.И. О природе программирования. -В кн.: Математическое обеспечение ВС из микро-ЭВМ (Вычислительные системы, вып. 96). Новосибирск, 1983, с.51-74.

I8. SCHERLIS W.L., SCOTT D.S. First stepts towards inferential programming. - In: Information Processing 83 (IFIP-83). Amsterdam, North-Holland,1983,p.199-212.

I9. TYUGU E.H. NUT - an object oriented language. - In: Artificial Intelligence and Information-Control Systems of Robots, Amsterdam, North-Holland,1984,p.69-76.

20. ТЫУГУ Э.Х. Концептуальное программирование. - М.: Наука, 1984. - 255 с.

21. УСПЕНСКИЙ В.А., СЕМЕНОВ А.Л. Теория алгоритмов: ее основные открытия и приложения. -В кн.: Алгоритмы в современной математике и ее приложениях. Ч.I, Новосибирск, 1982, с. 99-342.

22. ХЕНДЕРСОН П. Функциональное программирование. М.: Мир, 1983. - 346 с.

23. ЭВМ пятого поколения. Концепции, проблемы, перспективы (пер.с японского). - М.: Финансы и статистика, 1984. - 110 с.

Поступила в ред.-изд.отд.
15 января 1985 года