Диссертация на соискание ученой степени доктора физико-математических наук

А.Н. Терехов

Технология программирования встроенных систем реального времени

Актуальность темы. По всем классификациям программного обеспечения (ПО) встроенные системы реального времени относятся к наиболее трудным и дорогостоящим задачам. Срывы и многолетние отставания в сроках большинства крупных проектов в этой области как у нас в стране, так и за рубежом подтверждают актуальность задачи создания соответствующей технологии.

Цель работы. Создание технологии программирования встроенных систем, базирующейся на "сквозном" использовании современных алгоритмических языков высокого уровня (АЯВУ) на всех этапах жизненного цикла ПО, и реализация необходимых инструментальных программных и аппаратных средств.

Научная новизна. Основной отличительной чертой предлагаемой технологии является полнота, т.е. поддержка определённой методологией и инструментальными средствами всех этапов создания и поддержки ПО - от постановки задачи, системного анализа, графического проектирования и до генерации итоговых программ в кодах целевой ЭВМ. Новыми являются некоторые элементы техники трансляции АЯВУ типа Алгол 68. Оригинальна архитектура ЭВМ "Самсон".

Практическая ценность. Результаты этой работы внедрены в нескольких крупных организациях разных министерств и ведомств. Разработанная под руководством автора техника трансляции может быть использована в реализации статических АЯВУ типа Паскаль, Модула 2, Алгол 68, Ада. ЭВМ "Самсон" может быть использована как управляющий вычислительный комплекс и как ЭВМ, гибко настраивающаяся на новые классы задач и применений.

Апробация работы. Основные полученные автором результаты докладывались на многих Всесоюзных и республиканских конференциях (Ленинград, Новосибирск, Москва, Киев, Кишинев, Фрунзе), на заседаниях рабочих групп по Алголу 68 и по реализации языков программирования. По теме диссертации автором были прочитаны доклады во время научных командировок в Болгарию, Венгрию, Чехословакию, Индию, ФРГ, Бразилию.

Объём работы. Диссертация выполнена в форме научного доклада и состоит из введения, 5 параграфов и заключения.

Библиография - 36 наименований.

Введение

К встроенным системам принято относить различные приборы и установки, в которых ЭВМ непосредственно управляет технологическим процессом. Типичными примерами являются роботы, станки с числовым программным управлением, электронные автоматические телефонные станции, различные военные системы. Трудности реализации этих систем связаны с реальным временем, невоспроизводимостью ситуаций, часто оборудование разрабатывается вместе с ПО. Специализированные ЭВМ, используемые во встроенных системах, обычно обладают малым энергопотреблением, повышенной надежностью, достаточной скоростью, но очень редко при проектировании спецЭВМ учитывается такое "второстепенное"

требование, как удобство программирования. При работе непосредственно на спецЭВМ программист лишен таких ставших уже привычными средств, как многозадачная многопользовательская ОС, развитые средства отладки, базы данных, диалоговые режимы. Все это вынужденно приводит к использованию кроме целевой спецЭВМ мощной инструментальной ЭВМ и набора кросс-трансляторов. В свою очередь, работа на двух ЭВМ затрудняет комплексную отладку.

На наш взгляд, одной из главных трудностей встроенного ПО является организация взаимодействия двух совершенно различных категорий исполнителей - специалистов по конкретному оборудованию (мы называем их "функционалистами") и программистов. Различная специализация, образование, языки предметной области (то, что называется "профессиональный жаргон"), даже способ мышления, как оказалось, довольно сильно различаются. Таким образом, на первый план выходит задача формирования языка (в самом широком смысле этого слова) общения функционалистов и программистов.

Заметим сразу, что речь идет не только о языке программирования. Например, оказалось, что задание алгоритмов в любой текстовой форме плохо воспринимается функционалистами, привыкшими к графическим изображениям. С другой стороны, если ограничиться только графическими формами алгоритмов в любой, даже самой выразительной форме, эти алгоритмы трудно семантически проверить на ЭВМ, для этого необходима формализация до уровня АЯВУ допускающего трансляцию в коды ЭВМ. Эта проблема весьма актуальна и сегодня: если определить язык спецификации очень детально, то он становится сложным для функционалистов, а структура алгоритмов теряется за массой технических деталей, если же язык спецификации определить неформально или формально, но в очень крупных математических единицах, то теряется возможность непосредственной эффективной трансляции.

Понятен и путь решения (по крайней мере, один из возможных вариантов): нужно иметь последовательность языков, позволяющих постепенно конкретизировать постановку задачи от самых общих графических форм до объектных модулей целевой спецЭВМ. Трудность здесь в комплексности задачи, потеря или слабая технологическая поддержка хотя бы одного звена разбивает надежду на успешное решение исходной задачи. Например, в некоторых организациях используют графические языки спецификации в "ручном" исполнении без машинной поддержки. Но любая документация, не представленная на машинном носителе, мертва, т.е. быстро устаревает, практически не поддается исправлениям, не позволяет использовать сегодняшние достижения информатики. Практически невозможно проверить соответствие такой спецификации и разработанного на ее основе машинного алгоритма.

Удивительно, но и профессиональные программисты с большим трудом переходят на программирование на АЯВУ с программирования в кодах спецЭВМ. На наш взгляд, тому есть следующие причины.

1. Ложно понимаемая свобода творчества, когда программисту доступны все аппаратные возможности, любые схемы распределения памяти, любые соглашения о связях и т.п., но в больших коллективах это затрудняет разработку и сопровождение. Вполне реальна ситуация, когда программу одного автора вынужден сопровождать другой специалист, в этом случае фиксация соглашений просто необходима. Лозунг художников "форма освобождает" вполне подходит и в нашем случае.
2. Ложно понимаемые критерии эффективности. Действительно, квалифицированный программист на небольшой программе всегда выиграет у любого транслятора, но, во-первых, современные ЭВМ во многом снимают остроту проблемы эффективности, во-вторых, на больших программах транслятор может провести мощные оптимизирующие преобразования, которые чисто технически трудно выполнить вручную, в-третьих, в больших коллективах разработчиков действительно высокопрофессиональные программисты редки, а транслятор все-таки оптимизирует многие технические решения (например, распределение регистров). Наконец, главным

аргументом является скорость программирования, позволяющая создать несколько вариантов решения, провести анализ эффективности (инструментирование программы). Хорошо отработанный алгоритм без всяких мелких машинных оптимизаций позволит получить более эффективную программу, чем программа, сразу записанная в кодах спецЭВМ.

3. Программиста, привыкшего видеть все действия машины непосредственно, раздражают "интуитивно неожиданные" действия системы, когда ошибка в программе, вызванная неясностью в описании языка или неточностью реализации, не сигнализируется системой. Классическими примерами такого сорта являются несоответствие формальных и фактических параметров процедур, разные описания одной и той же глобальной переменной в разных единицах трансляции, порча констант и т.д. Тут приходится объяснять, что вовсе не любой АЯВУ годится для промышленного использования, особенно, с точки зрения надежности языка, которую мы понимаем как потенциальную возможность транслятора выявлять как можно больший процент ошибок пользователя.

В работе с большими программными комплексами, состоящими из сотен фрагментов, уже недостаточно возможностей традиционных трансляторов. В настоящее время все большую популярность завоевывают интегрированные системы программирования, включающие в себя:

- библиотечные системы с тщательным контролем прав доступа и возможностью автоматической перетрансляции всего дерева программы или отдельных поддеревьев на основе информации о датах и времени трансляции;
- текстовые редакторы, ориентированные на базовый АЯВУ технологии, облегчающие ввод и корректировку программ, а также их структурирование;
- трансляторы и кросс-трансляторы, работающие в пакетном и диалоговом режимах, причем в диалоге обеспечивающие автоматический вызов текстового редактора в случае обнаружения ошибки, чтение нужного файла с указанием ошибочного места; после исправления ошибки пользователь нажатием одной функциональной клавиши имеет возможность записать файл и повторить трансляцию;
- символический отладчик, который вызывается как по команде пользователя, так и автоматически в случае обнаружения ошибок в процессе счета.

По нашему убеждению, АЯВУ и интегрированная система программирования на его основе составляют стержень любой технологии. Технология программирования встроенных систем требует специализированных инструментальных средств, используемых как "до АЯВУ", помогающих спроектировать алгоритм на АЯВУ, так и "после АЯВУ", обеспечивающих перегрузку в коды целевых спецЭВМ, создание информационной базы, комплексную отладку на моделях и реальном оборудовании.

Данный доклад построен по следующей схеме: в параграфе 1 описываются основные концепции нашей технологии, перечисляются инструментальные средства с краткой характеристикой целей и возможностей каждого. Затем в четырех параграфах подробнее описываются ключевые, на взгляд автора, проблемы, связанные с выбором базового языка, трансляцией, проектированием ПО и возможностями аппаратной поддержки в виде новой ЭВМ, ориентированной на данную технологию.

Инициатором работ по технологии программирования в нашем коллективе был В.П.Морозов. Без его убежденности, энергии, организаторских способностей эта работа вообще не была бы возможной. Под руководством В.П.Морозова, совместно с Я.С.Дымарским и О.Е.Климовой, автор участвовал в выработке основных принципов технологии, в работах по стандартизации, определению основных методик.

Все инструментальные средства, упомянутые в данной работе, выполнены под общим руководством автора, во многих случаях автор принимал непосредственное участие в их реализации. Авторами основных инструментальных средств являются:

1. Система сетевого планирования - Э.Ф.Бычков.
2. Архив - С.Н.Комаров, Г.М.Серебрякова и А.И.Будник.
3. Текстовое документирование - Ю.К.Лаврова.

4. АРМ системного аналитика - Л.В.Евтеева.
5. Графическое проектирование - Е.И.Монарх, К.А.Терехова и Ведерников А.А.
6. Проектирование на Алголе 68 и отладка на инструментальной ЭВМ - П.С.Лавров, А.И.Будник и Е.И.Забокрицкий.
7. Трансляторы. Поскольку это ключевой элемент технологии, причем наиболее трудоемкий, остановимся на этом вопросе подробней. В течение длительного времени нашим базовым транслятором был транслятор А68ЛГУ, реализованный под руководством Г.С.Цейтина и А.Н.Терехова в семидесятых годах. В настоящее время основным становится переносимый транслятор WBC, разрабатываемый под непосредственным руководством Б.А.Федотова (им же реализуются административная и интерфейсная системы):
   а) видонезависимый анализ - М.А.Васильев и Б.К.Мартыненко;
   б) видозависимый анализ - Н.Ф.Фоминых;
   в) фаза оптимизации - Е.Е.Федотова;
   г) синтез для ЕС ЭВМ - Г.Х.Терехова, Т.Н.Попова, О.Г.Лапина;
   д) синтез для СМ ЭВМ - Б.А.Федотов, А.М.Перепеч, М.А.Васильев;
   е) синтез для ПК - Ю.К.Лаврова;
   ж) синтез для ЭВМ "Самсон" - Г.А.Швецова;
   з) генерация объектных модулей и поддержка - А.П.Рухлин;
   и) отладчик в терминах исходной программы - П.С.Лавров;
   к) система ввода/вывода - М.В.Евстюнин;
   л) текстовый редактор - А.Б.Васильев;
   м) библиотечная система - А.Е.Москаль.

   До переносимой системы были реализованы, внедрены в производство и использовались в конкретных заказах:
   а) кросс-транслятор с Алгола 68 в коды СМ-4 - группа под руководством Б.А.Федотова;
   б) кросс-транслятор с Алгола 68 в коды УК-1010 - группа под руководством Г.Х.Тереховой.
8. Система информационного обеспечения - группа под руководством В.В.Парфенова.
9. Специализированные средства (редакторы связей, интерпретаторы СЭВМ, генерация машинного носителя) - П.С.Лавров и А.П.Рухлин.
10. Наиболее существенную роль в реализации ЭВМ "Самсон" сыграл Н.Ф.Фоминых, ОС "Самсон" реализовал В.В.Оносовский, аппаратную реализацию осуществила группа инженеров под руководством Ю.Б.Рычагова. В ранних этапах проектирования архитектуры ЭВМ "Самсон" при нимал участие Г.С.Цейтин.

Очень существенную роль в процессе становления работ по технологии сыграли беседы с А.П. Ершовым, его развернутые отзывы. Многие сотрудники лаборатории системного программирования НИИММ ЛГУ и НИИ "Звезда" ЛНПО "Красная Заря", а также студенты и аспиранты, у которых автор был руководителем, внесли свой вклад в эту работу, и всем им, упомянутым или не упомянутым в тексте работы, автор выражает свою глубокую благодарность.

Основные концепции

В 1980 году в ЛГУ обратились сотрудники ЛНПО "Красная Заря" с просьбой помочь в программировании широкого класса задач управления и связи, в частности, создания функционального программного обеспечения (ФПО) телефонных станций, управляемых специализированными ЭВМ (СЭВМ). Несколько лет ушло на изучение предметной области, пробные реализации, решение организационных вопросов. У сотрудников ЛГУ была исходная позиция о важности использования алгоритмических языков высокого уровня (АЯВУ), основанная на опыте предыдущей работы. Однако, начинать пришлось совсем не с этого, а с повышения общей культуры программирования разработчиков ФПО. Дело в том, что в области встроенного ФПО

реального времени традиционно используются СЭВМ с нестандартной архитектурой, ориентированной на заданную предметную область (правда, не очевидно, в чем должна проявляться такая ориентация, например, если ЭВМ хорошо выполняет какие-то специальные операции, но плохо - условные переходы и вызовы процедур, которые встречаются в тысячи раз чаще, то можно ли считать, что ЭВМ соответствует предметной области?). Нестандартность архитектуры и малая тиражность таких СЭВМ приводят к отсутствию достаточно развитых операционных систем, трансляторов, средств отладки и других, ставших уже привычными, инструментов программирования. Поэтому мы столкнулись с работой на перфокартах и непосредственно за пультом СЭВМ "на тумблерах".

Была сделана попытка воспользоваться известными технологиями, однако, оказалось, что, например, Р-технология не имеет никаких средств настройки на СЭВМ, предлагаемые же в ней графический стиль программирования и программа-организатор с ручным вводом мало помогают в решении задач реального времени; технология РУЗА позволяет автоматизированным образом (но с большими доработками) построить кросс-ассемблер нужной ЭВМ и интерпретатор ее системы команд, а также осуществить некоторую регламентацию работы (например, стандартизовать имена объектов). Р-технология была отвергнута практически сразу (за неимением конкретных программных средств на заданных нам СЭВМ), РУЗА в течение полугода была настроена на одну СЭВМ, однако полностью учесть все особенности СЭВМ так и не удалось, кроме того, параметрически настраиваемые кросс-ассемблер и интерпретатор замедляют работу в 5-7 раз.

Мы решили, что из-за таких простых программ, как ассемблер, не стоит "городить огород" и за короткое время реализовали новые кросс-ассемблер и интерпретатор, которые вместе с текстовым документатором и некоторыми сервисными программами составили основу первой внедренной нами в производство (1984 год) технологической системы, интенсивно используемой сотнями разработчиков ФПО. Разумеется, мы отчетливо понимали ограниченность возможностей такой технологии, но в ее популярности были свои объективные причины:

1. Использование вместо СЭВМ широко доступной ЕС ЭВМ с многопользовательской ОС и широкими сервисными возможностями.
2. Работа одновременно многих пользователей за терминалами, а не с перфокартами.
3. Богатые отладочные свойства интерпретатора СЭВМ, недостижимые непосредственно на СЭВМ.
4. Впервые осознанная разработчиками ФПО полезность документации на машинном носителе, легкость ее оформления, исправления и тиражирования.
5. Практически неограниченные возможности развития технологии, удивительно быстро схваченные разработчиками ФПО, в результате, практически ежемесячно появлялись новые идеи и предложения, большинство из которых было быстро реализовано.

На протяжении всей совместной работы сотрудников ЛГУ и "Красной Зари" боролись две точки зрения на предмет технологии программирования: сотрудники ЛГУ, в основном, вкладывали в это понятие широкое использование инструментальных средств, а сотрудники "Красной Зари" настаивали на том, что технология - это, прежде всего, набор методик и регламентирующих средств, позволяющих, в частности, на каждом этапе провести экспертизу, архивацию и измерение объема и качества проделанной работы. Такой подход вызывал постоянное раздражение профессиональных программистов, привыкших считать, что программирование - это искусство. Где-то в глубине души, я (сам профессиональный программист и математик по образованию) так считаю и до сих пор. Разумеется, мы понимали, что работа без четких сроков и ответственности перед соисполнителями возможна только исследова тельская и экспериментальная, да и то в ограниченных объемах, но работа по конкретным заказам в промышленности дала массу новых впечатлений и импульсов для изучения. Например, уход исполнителя в самом разгаре работ, оказалось, что в промышленности это довольно частое явление (отсюда

необходимость в архивации и других формах отчуждения результатов работы от исполнителя); в коллективе из нескольких сот человек всегда есть люди, попросту не желающие работать и, хотя мы понимаем, что оценка работы программиста в байтах весьма сомнительна, это не снимает необходимости учета индивидуально выполненной работы (измерение); даже такое "приземленное" соображение, как разделение ответственности за принятые решения и ошибки, также нельзя сбрасывать со счета (отсюда жесткое документирование).

Особенно много разночтений вызывало требование оформления постановки задачи по стандартным требованиям. Дело в том, что аккуратное оформление требует усилий, причем не только технического порядка, а программисту кажется, что имея в руках столь мощный инструмент, как АЯВУ, легче сразу выразить свое понимание задачи в программе, а не в каких-то таблицах и диаграммах. Но непосредственное программирование лишает программиста обратной связи с функционалистом (постановщиком задачи) и, уж тем более, с заказчиком. Большинство сложных систем невозможно сдать в эксплуатацию из-за огромного количества сравнительно мелких замечаний, вызванных разночтениями и неясностями в постановке задачи.

Мы настаиваем на том, что, занимаясь вопросами документирования, ценообразования, способами регламентирования и контроля за ходом работ, нельзя забывать, что основным результатом применения технологии является программа, действующая в заданной вычислительной среде, хорошо отлаженная и документированная, доступная для понимания и развития в процессе сопровождения ("нам нужны не приборы в принципе, а приборы в корпусе").

Не вдаваясь в детальное определение понятия технологии программирования (сошлемся на доклад А.П.Ершова [1]), отметим все же, что технология программирования должна охватывать весь жизненный цикл программного продукта (ПП), способствовать повышению достоверности и надежности программирования, обеспечивать устойчивость ПП по отношению к смене технических средств и развитие ПП при изменении условий функционирования.

Чтобы избежать разночтений в вопросе корректности использования технологии, нужно иметь строго сформулированные требования к ней, а также к получающимся с ее помощью ПП (например, ОСТ [2]).

Мы по-прежнему считаем, что стержневым вопросом любой технологии является язык программирования [3]. На современном языке программирования (таком, как Алгол 68 или Ада) удобно вести проектирование методом структурной декомпозиции, получая на каждом уровне детализации действующий прототип системы (чего не скажешь про проект, выполненный в виде блок-схемы, графа или текста на естественном языке). Спроектированную на АЯВУ систему можно отладить на инструментальной ЭВМ, а затем с помощью кросс-транслятора перегрузить в коды СЭВМ [4]. Система должна обеспечить удобные средства раздельной трансляции фрагментов программы с полным контролем. Вообще, полнота контроля является важнейшей характеристикой, система не должна провоцировать пользователя на ошибки, допускать "интуитивно неожиданные" действия. Понятно, что контроль практически полностью определяется используемым АЯВУ. Программу на АЯВУ легче отлаживать (меньше ошибок, различные режимы трансляции, трассы, легкость отладочных изменений и т.д.). Даже документирование программ лучше выполнять, зная особенности языка и транслятора. Хорошая модульность в сочетании с полным контролем резко облегчает сопровождение. Таким образом, можно сделать вывод: "Не бывает внеязыковых технологий".

Важность языка для технологии заставила нас серьезно заняться изучением и сравнительной характеристикой различных АЯВУ. С точки зрения надежности программирования был выбран класс статических АЯВУ: Паскаль, Модула 2, Алгол 68 и Ада [5-10] - языков с полным видовым контролем периода трансляции. В Паскале нет средств раздельной компиляции, поэтому его можно использовать только для сравнительно небольших разработок и для обучения. В Модуле 2 нет динамических и подвижных массивов, большие ограничения на использование процедур (передача их параметром, присваивание переменным), нет средств

реального параллелелизма. Алгол 68 и Ада примерно эквивалентны по выразительной силе: к преимуществам Ады можно отнести еще более полный видовой контроль, пакеты, задачи, к преимуществам Алгола 68 - существенно более широкие возможности работы с массивами (можно вырезать строку, столбец и вообще любой прямоугольный подмассив), подвижные массивы, изменяющие свои размеры в процессе счета, возможность работы с процедурами, как с данными (например, список или массив из процедур), условные выражения, более эффективные в реализации, чем операторы Ады.

В любом случае, мы считаем, что использование любого из названных языков предпочтительнее по сравнению с языками типа Фортран, ПЛ 1 или Си. Тот факт, что базовым языком в нашей технологии является Алгол 68, объясняется только историческими причинами - наличием хорошо отработанной реализации, опытом обучения и использования и т.п. В настоящее время у нас имеются свои реализации всех четырех статических АЯВУ для различных ЭВМ.

К настоящему времени накоплен большой опыт применения АЯВУ Алгол 68 в задачах различного назначения. В качестве инструментального средства использовался транслятор А68ЛГУ, который имеет вполне удовлетворительные характеристики по надежности, времени трансляции, качеству объектного кода. Однако, в последние годы авторами Алгола 68 предложены новые интересные расширения языка, связанные с модульностью, раздельной трансляцией, обработкой исключительных ситуаций. С другой стороны, оказалось, что в А68ЛГУ довольно трудно вставлять технологические средства, не предусмотренные с самого начала работ (например, отладчик в терминах исходной программы). В процессе длительной эксплуатации обнаружились и другие, менее существенные, недостатки (например, слишком узкий диапазон целых чисел) и трудно исправимые ошибки (например, ошибки в некоторых вариантах динамического распределения памяти) транслятора А68ЛГУ. Поэтому было решено разработать новую систему программирования, получившую название WBC [11, 36].

Основной отличительной чертой новой системы программирования является ее интегрированность и ориентация, в основном, на диалоговый характер работы.

Имеются специальные средства конфигурационного контроля и поддержки разработки крупных программных комплексов.

Еще одной особенностью системы WBC является ее ориентация сразу на несколько инструментальных ЭВМ (ЕС, СМ, "Самсон", ПК, ПС 1001 и некоторые спецЭВМ). На базе А68ЛГУ было разработано несколько кросс-трансляторов, имелся также опыт и переноса транслятора на другие ЭВМ. Тот факт, что большая часть транслятора написана на Алголе 68, конечно же подталкивает к мысли о его переносимости. Одна ко реальный перенос оказался существенно сложнее. Понадобилось переработать всю структуру транслятора и его динамического окружения, выделить машинно - и операционно - зависимые части, унифицировать обращение к этим частям. Особого внимания требует работа с таблицами транслятора. На мини-ЭВМ с малым объемом прямоадресуемой памяти таблицы лучше хранить в файлах на дисках, а на больших ЭВМ это только замедляет работу транслятора. Поэтому был разработан набор процедур, "скрывающих" от транслятора истинную структуру таблиц, причем даже для одной ЭВМ возможны различные их реализации в зависимости от цели транслятора.

Во всех трансляторах системы WBC одинаковы:

- синтаксис промежуточных языков;
- структура таблиц и набор процедур доступа;
- программы монитора интерфейсной системы, видонезависимого и видозависимого анализа, фрагменты фазы оптимизации, выдачи листинга, текстового редактора и отладчика;
- способы распределения памяти и регистров;
- техника генерации объектного кода;
- набор процедур динамической поддержки (включая процедуры ввода/вывода);
- способ выбора вариантов реализации конструкций языка;

- инструментальный язык Алгол 68.

Такая унификация делает систему открытой к расширению в сторону новых ЭВМ и реальный опыт движения по пути

ЕС --> СМ --> "Самсон" --> ПК --> ...

подтверждает это.

Обязательным элементом технологии программирования мы считаем использование архива с контролируемым доступом. Архив должен следить за историей исправлений текстов и программ, хранить несколько версий важнейших документов, обеспечивать согласованность изменений (например, если изменилось ТЗ, то соответствующий текст программы помечается, а если изменился текст программы, то соответствующий загрузочный модуль удаляется). Разные группы пользователей должны иметь разные возможности доступа к архиву, в частности, должно обеспечиваться реальное отчуждение результатов работы от автора после приемки.

Разработка программного обеспечения ведется на технологической линейке, состоящей из нескольких рабочих мест. Распределение рабочих мест по различным ПК или дисплеям ЕС ЭВМ носит логический характер - несколько рабочих мест могут быть собраны на одном компьютере и использованы последовательно, какое-то рабочее место может целиком занимать все ресурсы одного компьютера - все зависит от наличных ресурсов и объемов производимого ПО.

Главным является рабочее место научного руководителя, включающее в себя систему сетевого планирования и архив хранения отчуждаемых от разработчика полуфабрикатов. Сетевой график позволяет наглядно представить себе состав и последовательность работ, необходимые ресурсы, связи между работами, ресурсами и исполнителями. Основная идея состоит в том, что сетевой график - это не "картинка для начальника", а удобный инструмент для исследования. Соответственно, основной объем программной поддержки этого рабочего места составляет диалоговая система, позволяющая в реляционной форме задавать довольно сложные и разнообразные вопросы и помогающая руководителю оптимизировать планы, отслеживать текущее состояние и предпринимать какие-то действия в случае аварийного несоблюдения планов.

Перед занесением в архив руководитель может выполнить какие-то проверки (трансляция, запуск теста) или просто просмотреть те материалы, которые предъявляются, никакого особого формализма тут не предусматривается. Важно, что исполнитель уже никак не может повлиять на содержимое архива.

Такие технологические средства, как текстовые документаторы, трансляторы, архивы, средства планирования и контроля носят универсальный характер и могут быть использованы в очень широком классе применений ЭВМ. Для встроенных применений ЭВМ нужны дополнительные средства.

Поскольку ЭВМ управляет оборудованием, неизбежно появляется интерфейс со специалистами в этом оборудовании, но далекими от программирования (мы называем их "функционалистами"). После нескольких неудачных попыток внедрить в качестве средства общения функционалистов с программистами какие-то текстовые формы записи (даже на узко специализированных языках), мы решили [11] использовать в качестве основы графический язык SDL, рекомендованный МККТТ (Международный консультативный комитет по телефонии и телеграфии). В этом языке есть средства определения параллельно протекающих процессов и синхронизации с помощью аппарата сообщений. Были реализованы графические средства редактирования SDL-диаграмм, первичного контроля, автоматического преобразования SDL-диаграмм в тексты на Алголе 68, средства конфигурационного контроля.

Для того, чтобы программа могла выдавать конкретные воздействия на управляемое оборудование, она должна иметь полную и точную информацию о размещении различных блоков оборудования, их функцио нальных особенностях, связях между ними с точностью до каждого ТЭЗа и каждого контакта. Во многих реальных случаях это выливается в сотни тысяч объектов учета, трудности

ввода, проверки, коррекции. Некоторые задачи информационного обеспечения имеют принципиально новый и трудный характер, например, замена сложных агрегатов данных на работающем объекте. В наших работах средства создания и поддержки информационного обеспечения часто превосходят по объему и сложности основное ФПО, поэтому мы считаем такие средства важнейшей частью технологии встроенных систем.

Особую трудность составляет отладка встроенных систем, например, трудно обеспечить повторяемость ситуаций, критические режимы, часты расхождения в поведении реального оборудования с паспортными данными. Мы предлагаем последовательно усложняющуюся последовательность отладки на моделях оборудования - сначала на простых имитационных моделях (аргумент - результат), затем на структурных моделях (аргумент - задержка - результат) с вероятностным моделированием отказов, наконец, комплексная отладка на моделях оборудования и окружения с вероятностным моделированием нагрузки. Разработаны специальные инструментальные средства как для моделирования, так и для собственно отладки (различные трассировки и средства анализа трасс, сбор статистики, возобновление с нужного места и т.д.).

Комплексная отладка на объекте, если целевая ЭВМ не совпадает с инструментальной, все же остается очень трудной задачей, даже с использованием инструментальных средств. Нужно понимать, что использование кросс-средств является вынужденной мерой, связанной с низкими технологическими качествами многих СЭВМ. Поэтому актуальной является задача построения ЭВМ, которая могла бы служить как технологической, так и целевой. Оказалось, что если ограничить класс входных языков классом статических АЯВУ, можно построить довольно компактную ЭВМ, ориентированную на определенную технологию программирования на базе Паскаля, Модулы 2, Алгола 68 и Ады [12].

С нашей точки зрения, ЭВМ, для которой одновременно и со взаимным влиянием разрабатываются архитектура, транслятор, операционная система, программное и микропрограммное обеспечение, является неотъемлемой составной частью технологии программирования.

Выбор базового АЯВУ

Все программы для старых ЭВМ (и, к сожалению, многие программы для современных спецЭВМ) писались непосредственно в кодах ЭВМ.

Понятно, что программа в кодах ЭВМ или на языке ее ассемблера тесно привязана к этой ЭВМ и при смене ЭВМ программу нужно будет выкинуть. Еще хуже тот факт, что ассемблер практически не может обнаруживать ошибки пользователей (несоответствия операций их операндам, неправильные адреса и номера регистров и многие другие ошибки), т.е. программирование на языке ассемблера очень ненадежно.

Еще в 1956 году был придуман язык Фортран, в котором впервые алгоритм записывался в обычной алгебраической форме без привязки к конкретной ЭВМ, например:

```
запуск = время + интервал
```

Разумеется, для Фортрана нужен транслятор в коды ЭВМ, который значительно сложнее ассемблера, зато многие классы ошибок просто исчезли (неправильный код операции, неправильное использование регистров и адресации). Постепенное понимание, что легкость программирования и скорость получения результата важнее затрат на создание трансляторов, времени трансляции и некоторых потерь во времени счета, позволило организовать триумфальное шествие тысяч алгоритмических языков высокого уровня (АЯВУ).

Первоначально развитие АЯВУ шло по пути повышения уровня (выразительности). Действительно, проще написать F(A,B) для вызова функции с параметрами A и B, чем

```
LET X = A
LET Y = B
```

```
      GOSUB 100
```
на языке Бейсик (1965 год), причем, поскольку в этом языке нет блочной структуры, нужно аккуратно разделить возможные идентификаторы между авторами программы.

Проще написать присваивание матриц в виде A := B, чем

```
      DO 1 I = 1, N
      DO 2 J = 1, N
      A (I,J) = B (I,J)
   2  CONTINUE
   1  CONTINUE
```

Дело не только в длине записи программы, в варианте A := B не перепутаешь размеры матриц, не ошибешься в использовании вспомогательных переменных I и J, да и в реализации на ЭВМ A := B наверняка будет эффективнее.

Избавив пользователя от одних ошибок, Фортран породил новые классы ошибок, например:

а) X := 1/3

Пользователь ожидает, что вещественной переменной X присвоится значение 0.3333, а вместо этого присваивается 0. Если бы он написал 1./3 или 1/3. , то все было бы в порядке.

б) DO 3 I = 1.4

Пользователь хотел написать заголовок цикла, но по ошибке вместо запятой поставил точку, в результате получился оператор присваивания переменной DO3I. Говорят [13], что это самая дорогая ошибка мира, из-за нее сорвался запуск американского космического корабля на Венеру.

в)
```
      CALL F(1)
      I = 1
      WRITE ( )I
      STOP
      SUBROUTINE F(A)
      A = 0
      RETURN
      END
```

В этом примере интересны только две выделенные строчки, где переменной I присваивается 1, а затем печатается I, равное 0. Все остальные строки только объясняют причину ошибки.

Ошибки такого рода очень трудно находить. Если пользователь по ошибке написал A + B вместо A - B, то, внимательно читая программу, эту ошибку он обнаружит. Если же он не подозревает об эффекте присваивания X = 1/3, то сколько бы он ни читал программу, ошибку он не найдет.

В настоящее время системы программирования принято делить на "дружественные пользователю" и не помогающие ему. Способность системы обнаруживать ошибки пользователя и не провоцировать его на новые становится ключевой для успешного применения ее в качестве технологической. С другой стороны, не все зависит от транслятора, в частности, приведенные выше примеры не противоречат определению языка Фортран. Таким образом, можно говорить о потенциальной надежности языка программирования как такового. Единого определения надежности АЯВУ пока не существует, мы будем исходить из следующих предпосылок.

1. Надежный язык программирования должен обеспечивать возможно более раннее и полное обнаружение ошибок.
2. Должна быть обеспечена "локальность видения" программы. Это связано с психологией работы любого специалиста и особенно программиста. Человек может эффективно изучать только небольшие участки проекта, мало связанные с другими участками; увеличение длины участка или числа возможных

интерфейсов с другими участками катастрофически сказывается на способности восприятия человека.

Известно два основных способа конструирования языков программирования - статический и динамический. В статических языках программирования типы данных фиксируются и контролируются при написании программы и лишь в небольших пределах могут изменяться в процессе счета, в динамических же языках тип данного определяется полностью во время счета, соответственно тогда же должны выполняться и контролирующие функции.

Очевидно, что первому критерию надежности лучше удовлетворяют статические языки, так как при первой же трансляции будут выявлены все ошибки, связанные с типами данных и неправильным их использованием. При работе с динамическим языком ошибка может быть обнаружена только в том случае, если в нужную точку программы мы попадаем с нужной комбинацией операндов. Это может случиться и через несколько лет после начала эксплуатации программы.

Динамические языки, разумеется, дают некоторые дополнительные возможности по разнообразному использованию памяти и преобразованию типов данных, но в реальных программах необходимость в этих средствах очень мала (если уж программист определил целую переменную, тов подавляющем большинстве случаев он и будет ее использовать как целое число, а не набор из двух литер, вещественное число, или еще что-нибудь). Всегда можно ввести динамические средства в статический язык в качестве дополнительной, тщательно охраняемой возможности.

Типичными примерами статических языков можно назвать Паскаль, Модула 2, Алгол 68, Ада, примерами динамических языков являются Лисп и Автокод МВК "Эльбрус".

Перечислим некоторые основные характеристики надежного языка программирования, непосредственно вытекающие из приведенных выше требований. Этим требованиям удовлетворяют Алгол 68 и Ада, а в большинстве случаев и только они из известных языков программирования.

1. Все объекты, использованные в программе, должны быть описаны (в языках Алгол 68 и Ада описываются не только идентификаторы, но и новые типы данных и операции). Так как каждый объект используется по крайней мере дважды (в описании и применении), то случайное искажение записи объекта будет обнаружено транслятором. Определение типа идентификатора по первой букве - одно из самых слабых мест Фортрана и ПЛ1.

По соображениям локальности желательно, чтобы все описания были как можно ближе к применению, для того, чтобы для определения рабочего идентификатора не надо было возвращаться, например, к началу процедуры. Те конструкции, в которых можно помещать описания, принято называть блоками. В Алголе 68 блоками являются не только процедуры или последовательные предложения, но и условные предложения и циклы.

В примере

```
WHILE INT MI = M[I] ; MI /= 0
DO PRINT (MI) ; I := MI OD ;
```

идентификатор MI, описанный в предложении после символа .WHILE, используется в теле цикла. В языке Ада пришлось бы дважды использовать конструкцию M[I] (примерно 10 лишних команд), так как блочная структура в нем значителбно слабее, чем в Алголе 68.

Необходимо, чтобы инициализацию идентификаторов можно было делать прямо в описании, так как ошибки, связанные с использованием идентификаторов без начального присваивания, относятся к числу наиболее трудных: при разных пусках программа ведет себя по-разному.

2. Не должно быть никаких ограничений на длину идентификаторов, в составе идентификаторов должен допускаться пробел или иной разделитель, должна быть обеспечена возможность использования русского алфавита в идентификаторах и сообщениях транслятора. В противоположность известной фразе из учебников по

программированию следует сказать: "Каждый идентификатор имеет внутренний, присущий только ему, смысл".

Ключевые слова должны обязательно выделяться в тексте, это существенно повышает наглядность программ. Должны допускаться русские варианты ключевых слов.

3. Язык должен обеспечивать полный видовой контроль во всех точках программы. Это дает новое мощное контролирующее средство (в частности, именно отсутствие видового контроля приводит к казусам в приведенных выше примерах на Фортране) и, кроме того, существенно облегчает построение эффективного объектного кода, так как не нужны команды динамических проверок и появляется возможность лучше учитывать особенности системы команд для разных типов данных. В таких языках, как Алгол 60, Паскаль, Си и ПЛ1, видовой контроль нарушается при передаче параметров процедуры.

4. Способы использования в языке идентификаторов-констант должны отличаться от способов использования идентификаторов-переменных. В технических текстах вообще принят термин "вводим обозначение", причем это обозначение своего смысла не изменяет. В учебниках по программированию даются советы не использовать численных констант в явном виде, так как, если когда-либо понадобится заменить константу на другую, это будет очень затруднительно сделать, да и наглядность у символических обозначений констант гораздо выше. С другой стороны, такой текст на ПЛ1, как

DCL PI BIN FLOAT INIT (3.1415926)

не является выходом из положения, так как ничто не помешает исполнению присваивания PI = 2.718281828, кроме того, описание идентификатора требует динамически исполняемых команд и закрывает многие возможности для оптимизации. Например, -PI потребует динамического исполнения, а -3.1415926 - нет. В языках Алгол 68 и Ада есть специальные описания констант, лишенные этих недостатков.

5. Современный язык программирования должен содержать средства расширения, т.е. возможности определения специализированных языков программирования, для которых было бы возможным использовать трансляторы и другие элементы системы программирования базового языка. Основу каждого специализированного языка составляют специфические типы данных и операции над ними, а также специальные функции, константы и переменные. Все это можно определить, не выходя за рамки языков Алгол 68 и Ада, причем оба языка предоставляют средства "накопления" контекста, т.е. средства построения специализированных окружений.

6. Сегодня уже очевидно, что большие системы на языках высокого уровня можно строить только при использовании независимой трансляции модулей. При независимой трансляции особую трудность представляет организация связи между отдельно транслированными модулями. Передача информации через параметры процедур слишком обременительна (ведь нужно передавать иногда десятки параметров). Операторы COMMON из Фортрана слишком опасны, так как, если в одном модуле написать COMMON A, B , а в другом COMMON B, A, то никакой сигнализации не будет. Не менее опасны идентификаторы с атрибутом EXTERNAL в ПЛ1, так как для таких идентификаторов не ведется видовой контроль, кроме того, каждое использование такого идентификатора требует лишних команд (загрузка V-константы в регистр).

Наиболее аккуратным решением является отдельная трансляция процедур с глобальными (т.е. не описанными в них) идентификаторами с полным видовым контролем. Такие идентификаторы должны описываться в специальной конструкции - библиотечном вступлении в языке Алгол 68 и пакете в языке Ада. Организация пакетов в языке Ада - одна из самых сильных сторон языка. В дополнение к своей основной функции (определение глобальных идентификаторов для других процедур) пакеты дают возможность ограничивать доступ к глобальным идентификаторам (иногда нужно, чтобы некоторые глобальные идентификаторы были доступны только определенным процедурам, а все другие процедуры могли их использовать только через эти выделенные процедуры), с помощью пакетов можно строить заготовки для

многих однотипных программ, отличающихся, например, только типом обрабатываемых данных.

В Алголе 68 эти возможности представлены слабее, однако, уже опубликованы предложения по расширению языка, перекрывающие возможности языка Ада. В реализации ЛГУ многие эти возможности уже предусмотрены.

Если в АЯВУ предусмотрены возможности иерархического накопления контекстов (набор описаний) и отдельной трансляции модулей, этот АЯВУ можно (и нужно) использовать в качестве языка спецификаций при проектировании больших систем. Спецификация, в которой приводится описания видов данных, констант, переменных, заголовков процедур и операций (их тела описываются и транслируются отдельно), имеет гораздо большую ценность, чем спецификация, записанная на русском языке, даже при соблюдении жестких правил и стандартов: правильность спецификации, записанной на АЯВУ, проверяется транслятором (особенно важна проверка соответствия параметров модулей), а правильность спецификации, записанной на русском языке, формальными методами проверить (по крайней мере, в настоящее время) невозможно.

В двух случаях построение языка Ада не удовлетворяет требованиям надежности:

1. В языке не требуется выделения ключевых слов, формулируется лишь запрет использования их в качестве обычных идентификаторов. Даже в официальном сообщении о языке ключевые слова выделяются полужирным шрифтом "для удобства чтения", программы же пользователя будут лишены такого удобства.

В Алголе 68 ключевые слова, а также обозначения для видов и операций специальным образом выделяются в тексте программы.

2. Ада является языком операторов, а не языком выражений. Никому не придет в голову оператор X := A + B*C разделить на два

```
R := B*C ;
X := A + R ;
```

Формула же (на Алголе 68)

```
IF A > B THEN A ELSE B FI + 1
```

на языке Ада потребует согласованного изменения в четырех различных точках программы:

1) Описание рабочего идентификатора R в начале блока или процедуры,

2) R := A после символа **THEN**,

3) R := B после символа **ELSE**,

4) R + 1 после условного оператора.

Разумеется, это противоречит принципу "локальности видения". Бэкус называет присваивание "бутылочным горлышком", справедливо указывая, что каждое присваивание требует обращения к оперативной памяти, создавая тем самым "узкое место" вычислительной среды. Кроме очевидного вреда, приносимого введением "слепых" рабочих идентификаторов, ухудшается эффективность программы, так как выражения обычно вычисляются на регистрах процессора, а каждое присваивание - это засылка в память.

Первый транслятор с Алгола 68 (для ЕС ЭВМ) появился в 1978 году, в 1982 году началось его промышленное использование. Сейчас есть трансляторы для МВК "Эльбрус", СМ 4 (с операционными системами "Рафос" и ОС РВ), ПЭВМ, совместимых с IBM PC, а также более 10 кросс-трансляторов для различных спецЭВМ. Наш опыт внедрения языка в промышленность убедил нас в сложности этой задачи (пробное прог раммирование, создание специализированных окружений, реализация кросс-трансляторов, обучение пользователей и многое другое) и ничто не дает оснований предполагать, что внедрение языка Ада будет проще.

Таким образом, в настоящее время базовым языком выбран Алгол 68. Учитывается возможность использования в качестве второго базового языка в начале девяностых годов языка Ада.

Трансляция

Основой нашей технологии является сквозное применение АЯВУ на всех этапах жизненного цикла создания ПО, начиная от постановки задачи и проектирования. Понятно, что без соответствующей поддержки транслирующими средствами этот тезис превращается в пустой лозунг.

Долгие годы бытовало мнение, что все вопросы решат СПТ - системы построения трансляторов, большая часть публикаций, выступлений на конференциях по технике трансляции посвящалась именно СПТ. Дело дошло до того, что на обложке сборника работ всесоюзного симпозиума по методам реализации новых алгоритмических языков [14] была нарисована корова: когда художник спросил, что же такое СПТ, ему объяснили, что на вход СПТ подается описание языка, а на выходе получается транслятор, да и "доить" эту тему очень удобно.

Несмотря на обилие СПТ, промышленных трансляторов с широко известных АЯВУ с их помощью разработано очень мало. Более того, даже для родственных по построению языков, необходима специфическая техника трансляции. Автору этих строк довелось открывать всесоюзную конференцию по методам трансляции (Новосибирск, 1981 г.) приглашенным докладом о необходимости различных специализированных инструментов и приемов трансляции в противовес единым СПТ. В данной работе мы сосредоточимся на вопросах трансляции базового АЯВУ технологии - Алгола 68.

Так получилось, что с Алголом 68 наш коллектив связан около 20 лет. Менялся язык (Пересмотренное сообщение опубликовано в 1975 году), менялись ЭВМ и их операционные системы, менялось наше представление о роли языка и трансляторов с него. Если в первых реализациях главным для нас было "влезть" в маленькую оперативную память первых моделей ЕС ЭВМ, из-за этого принимались жесткие количественные ограничения, беспощадно обрезалась структура внутренних таблиц транслятора, даже структура транслятора подбиралась таким образом, чтобы в каждом проходе было не более двух таблиц с заранее неизвестными размерами [15], то в последующих реализациях, по мере накопления опыта практического использования, акценты смещались в сторону технологии программирования, удобства отладки, вопросов переносимости трансляторов на другие ЭВМ.

В 1976 году вышла монография [16]. В ней хорошо отражены представления о реализации Алгола 68 на тот период времени. Однако сегодня ни одна глава этой монографии не соответствует нашим нынешним представлениям: вместо автоматически построенного магазинного анализатора используется рекурсивный спуск, который позволяет существенно улучшить диагностику ошибок; применяется другая структура таблиц и техника их построения, новые алгоритмы работы с видами Алгола 68; разработаны новые способы управления памятью и распределения регистров; применяются другие способы реализации конструкций языка и организации взаимодействия с операционной системой; вместо синтеза объектной программы на основе макроязыка ассемблера используется специальная техника написания синтезирующих процедур на Алголе 68; не нашли практического применения предложенные в [16] средства оптимизации объектной программы и методы отладки.

Синтаксис Алгола 68 не ориентирован на однопроходную трансляцию. С символа "(" может начаться как замкнутое предложение, так и, например, условное предложение или текст процедуры. Не предусмотрен жесткий порядок описаний, поэтому использующее вхождение идентификатора может встретиться до его описания. В Алголе 68 можно описывать не только идентификаторы, но и новые виды, операции. Это, несомненно, усиливает выразительность языка, но и усложняет структуру транслятора [15]. Например,

```
m X;
```

может быть либо описателем переменной X, если **m** описан как индикант вида, либо унарной формулой, если **m** - операция. Таким образом, идентификация должна быть двухступенчатой: сначала идентифицируются индикаторы вида и операций,

при этом выделяются описания идентификаторов, после чего идентифицируются идентификаторы и операции.

Как обычно, работа транслятора начинается с этапа видонезависимого анализа. Для каждой инструкции языка написана процедура анализа, например, по символу if вызывается процедура анализа условного предложения, которая проверяет, что за символом if следует выясняющее предложение, затем символ then и т.д.

Поскольку внутри, например, выясняющего предложения может снова встретиться условное предложение, процедуры анализа должны быть рекурсивными, соответственно и метод называется "методом рекурсивного спуска". Если с какого-то символа может начаться несколько конструкций (например, с левой круглой скобки или идентификатора), составляется процедура анализа "обобщенной" конструкции, в которой анализ сначала идет в расчете сразу на несколько конструкций, когда же встречается какой-то символ, позволяющий уточнить тип конструкции, анализ продолжается только в расчете на эту конструкцию. Таким образом, за один просмотр можно разметить окончания почти всех констукций, после чего довольно просто на обратном просмотре (т.е. двигаясь от конца текста к его началу) разметить начала конструкций.

Минимальной единицей текста для рекурсивного спуска является не отдельная литера, а более крупный элемент - лексема, например, идентификатор, константа, ключевое слово (if, do, ...), составные символы типа :=, +:= и т.п. Лексемы выделяет в тексте процедура сканер (от англ. scan - просматривать). Иногда удобно заглядывать вперед на одну лексему. В этом случае повторного просмотра текста не происходит, запоминаются только результаты сканера. В процессе работы сканер строит различные таблицы, для поиска в них используются хэш-списки. После сравнения многих вариантов организации таблиц традиционным стал метод поиска со следующей функцией: вычисляется сумма кодов литер, составляющих искомый элемент, в качестве значения берется младший байт суммы. Во всех практических случаях разбиение на 256 классов достаточно, чтобы находить любой элемент не более, чем за 2-3 сравнения.

С помощью хэш-таблиц осуществляется и идентификация. При входе в блок просматривается блочная секция таблицы, в каждом элементе есть ссылка на хэш-таблицу, в хэш-таблице помещается ссылка на данный элемент блочной таблицы (старое значение ссылки запоминается либо в стеке, либо в этой же блочной таблице). Для каждого использующего вхождения индикатора без какого-либо поиска сразу указывается соответствующая строка блочной таблицы.

Таким образом, фаза видонезависимого анализа (прямой и обратный просмотр) включает в себя следующие действия. Размечаются конструкции языка в исходном тексте, строятся таблицы исходных представлений индикаторов, описателей, блочная таблица индикаторов, идентифицируются индикаторы вида и приоритета, исходный текст преобразуется в последовательность целых чисел - кодов конструкций и ссылок на таблицы. Синтаксис этого промежуточного представления формализован. Когда фаза видонезависимого анализа уже была написана, Б.К.Мартыненко предложил новую версию системы автоматического построения таблиц анализатора по грамматике исходного языка с существенно улучшенной диагностикой ошибок пользователя. Хотя объем полученных таким образом таблиц оказался довольно большим (порядка 50К байтов только для прямого просмотра), было решено использовать этот метод, поскольку уменьшилось время трансляции, улучшилась диагностика, были сняты практически все ограничения на входной язык, кроме того, в автоматически построенном анализаторе существенно меньше вероятность ошибок в трансляторе.

Второй фазой транслятора является фаза видозависимого анализа. Концептуально эта фаза мало изменилась по сравнению с монографией [16], но все программы неоднократно переписывались и улучшались.

Интересно отметить, что и здесь неоднократно наблюдалось некое спиральное развитие: если в первом варианте сложность установления эквивалентности видов и

упорядочения видов, входящих в состав объединенного вида, оценивалась как exp(n), где n - число этажей в де-
реве вида, то в [17] был предложен алгоритм с оценкой n*ln(n). Однако оказалось, что для нового алгоритма существенно рассматривать всю таблицу видов целиком, а не только ту часть, которая соответствует текущей единице трансляции. Таким образом, в алгоритме заложена довольно большая постоянная составляющая (только в стандартном вступлении используется несколько сот описателей). В настоящее время в трансляторе используются заново написанные программы практически по первоначальным алгоритмам.

Результатом этой фазы является текст на промежуточном языке, в нем все конструкции явно выделены, все приведения вставлены, все идентификационные связи установлены. Для каждой конструкции указаны виды операндов и нужный вид результата.

Традиционно, в наших трансляторах отсутствовала фаза глобальной оптимизации.

Многие неудачные языковые конструкции (типа изменяемых пределов цикла, параметров, вызываемых по наименованию), ради которых стоило бы реализовывать глобальные оптимизаторы, отсутствовали в Алголе 68. С другой стороны, глобальная оптимизация все равно не помогала при отдельной трансляции процедур, а для нас такой режим стал основным. Тщательный выбор представления значений, многовариантный синтез с широким использованием локальной оптимизации позволил добиться вполне приемлемого качества объектного кода. Однако во многих случаях локальной информации существенно не хватает, например, пусть переменная X описана как int X; X многократно используется в программе, поэтому желательно разместить ее в регистре, однако, если эта переменная передается параметром вида ref int в какую -то процедуру, то размещать X в регистре нельзя, так как трудно будет уследить за соответствием значения регистра нескольким переменым. Много примеров такого рода дает управление динамической памятью (именуется каким-то именем массив или нет, плотный или мог быть получен триммерной вырезкой и т.д.).

Собрать необходимую информацию нетрудно (на прямом просмотре обобщается информация о различных использованиях объектов блока, на обратном просмотре суммарная характеристика приписывается каждому использованию). Пока таких заказов было немного, они выполнялись на фазе видозависимого анализа, что, разумеется, ее не упрощало. В конце концов, эти два просмотра (прямой и обратный) было решено выделить в отдельную фазу оптимизации. Оказалось, что для многих оптимизаций нет необходимости строить графы информационных связей и связей по управлению с последующим их изучением (весьма дорогостоящим), а достаточно двух строго линейных просмотров. В более поздних работах [18] приводятся результаты экспериментов по выявлению влияния различных оптимизаций на качество объектного кода. Оказалось, что, в основном, влиятельны оптимизация вырезок в циклах и различные редукции (константные вычисления, понижение степени операций), для которых как раз достаточен описанный механизм, а типичные глобальные оптимизации (нахождение общих подвыражений, чистка циклов, не связанная с простейшей оптимизацией вырезок) влияют на качество объектного кода очень мало.

Другой причиной для выделения отдельной фазы оптимизации была необходимость понижения уровня промежуточного языка, получающегося на фазе видозависимого анализа. Оказалось, что самой сложной и объемной фазой является фаза синтеза, например, в А68ЛГУ фаза синтеза по объему была больше всех остальных фаз вместе. Поскольку текст на промежуточном языке в значительной степени отражал структуру исходной программы, а не объектной, во время синтеза приходилось заново разбираться, была ветка elif или нет, простое присваивание или присваивание массивов (и каких), надо ли копировать элементы массива фактического параметра и т.д. Пока был один транслятор для одной ЭВМ, было в значительной степени все равно, кто и когда проведет этот разбор, однако для переносимой реализации оказалось весьма желательно вынести эти

вопросы в отдельную фазу, чтобы не повторять их в фазах синтеза для различных ЭВМ.

Основные проблемы фазы синтеза связаны с необходимостью учета вариативности системы команд, например, существенно различаются, загрузки 0, короткой или длинной константы; переменной с коротким или длинным смещением; вызовы подпрограмм, статические вызовы процедур без глобальных объектов или требующих установки статической цепочки, нестатические вызовы.

Интересную и, на мой взгляд, закономерную эволюцию претерпела архитектура МВК "Эльбрус". Если "Эльбрус 1" и "Эльбрус 2" [19] были рассчитаны на практически прямую трансляцию, в надежде что все варианты аппаратура вытянет сама, в результате практически не было прямоадресуемых регистров (кроме аппаратной верхушки стека), почти все команды были представлены в одном варианте (и тут были исключения, например, выгрузки с сохранением или без), то в "Эльбрусе 3" и в микропроцессорном исполнении [20] уже есть отдельные команды, например, для сложения целых, вещественных и чисел статически неизвестного вида, в каждой процедуре есть 64 прямоадресуемых регистра, введены элементы статического анализа, то есть существенную роль в оптимизации времени исполнения должен сыграть транслятор. На 1 конференции по МВК "Эльбрус", состоявшейся в Новосибирске в 1981 году, я предсказывал такое развитие событий и даже внес конкретные предложения, близкие к тем вариантам, которые реализованы сейчас.

На наш взгляд, вариативность системы команд носит объективный характер, и транслятор наравне с аппаратурой должен отвечать за эффективность (в том числе и за то, чтобы не использовать сложных команд там, где можно обойтись простыми).

Метод синтеза в наших трансляторах основан на предложенной Г.С.Цейтиным (который, в свою очередь, опирался на работы Бранкара и Леви [21]) системе запросов и ответов. Во многих трансляторах автору этих строк пришлось осуществлять практическое руководство реализацией фазы синтеза и разрабатывать технику синтеза [22-26].

На вход поступает текст на промежуточном языке (ПЯ) - линейная развертка дерева программы. Обход этого дерева осуществляется с помощью рекурсивной процедуры inner:

```
proc inner = void:
case ЧИТАТЬ КОД in
# 1 # конструкция с кодом 1,
...
# N # конструкция с кодом N
esac
```

Каждая конструкция читает из ПЯ свои параметры, затем, если она имеет подконструкции, снова вызывает inner, например, присваивание вызывает inner два раза. Перед вызовом inner конструкция выдает запрос на значение подконструкции:

V - не нужно значения (выдается последовательным предложением на все основы, кроме последней);

R - результат поместить в указанном в запросе регистре, для стековых машин используется разновидность этого запроса - S, т.е. поместить на стек. Выдается вызовом на параметры, ветвящимися предложениями на значения;

A0 - голову значения разместить в указанном в запросе определенном месте памяти, выдается записью структуры и теми же ветвящимися предложениями, если их значения не размещаются на регистрах;

AB - запрос на значение вида bool; в ответ на него подконструкция должна породить команду передачи управления по значению false на указанную в запросе метку;

F - свободный запрос; получив такой запрос, подконструкция размещает свое значение любым доступным образом (не затрачивая ни одной лишней команды) и сообщает о выбранном размещении через стандартную глобальную переменную "ответ"; этот запрос выдает большинство подконструкций;

FD - запрос на разыменование - аналог запроса F, но с разыменованием результата.

Смысл запросов очевиден - подсказать подконструкции, что будет делаться с ее значением дальше, например:

```
struct ( int A, B ) C = if УСЛОВИЕ
then ( X, Y )
else ( U, T )
fi
```

Здесь на правую часть описания тождества выдается запрос A0, сопровождающийся адресом C. Благодаря этому записи структур (X, Y) и (U, T) будут формировать свои значения сразу в C, а не в рабочей памяти с последующей пересылкой.

```
int X, Y; X := Y := 0;
```

Здесь присваивание Y := 0 получит запрос FD и ответит номером регистра, через который будет пересылка 0, а не адресом памяти, соответствующим Y.

Обычно структура ответа описывается примерно следующм образом:

```
mode ответ = struct ( int rep, displ, base, ... );
ответ ответ;
```

Перечислим несколько типовых вариантов представления (rep):

C - константа, в displ - ее значение, в base - ссылка на таблицу исходных представлений;

L - значение в памяти, displ, base описывают размещение;

A - разыменованное значение статической переменной, т.е. описанной в описании переменной (**int** X;). Разыменованное значение используется значительно чаще, поэтому лучше хранить информацию о нем, а имя (т.е. адрес) получать по мере необходимости командой загрузки адреса.

R - значение в регистре, номер которого записан в base.

P - значением является текст процедуры, в displ - метка входа, в base - уровень процедуры, если первичное вызова выдало такое представление, вызов может породить более эффективные команды статического вызова.

V - отдельно транслированный текст процедуры; в ответ на него также удается породить статический вызов, но с указанием необходимой информации для редактора связей.

В реальных трансляторах структура ответа шире, а вариантов значительно больше. Таким образом, для каждой языковой конструкции нужно написать процедуру генерации, которая выдает нужные ей запросы на подконструкции, вызывает inner, а затем, на основании ответов от подконструкций и запроса от объемлющей конструкции, порождает наиболее подходящие для каждого варианта команды.

Все эти годы шла настоящая борьба с большим объемом фазы синтеза. По крупицам, из транслятора в транслятор собирались технические приемы, позволяющие без ухудшения качества объектного кода собрать несколько конструкций в одну параметризованную процедуру, выделялись удобные рабочие процедуры.

Ранее уже рассказывалось, как удалось часть действий, связанных с разбором структуры программы и анализом вида обрабатываемых значений, вынести "назад" - в фазу оптимизации. Аналогично другую часть действий, связанных с выбором конкретной команды из набора однотипных, можно вынести "вперед" - в следующую фазу генерации объектного модуля. В любом случае, фаза синтеза не может быть однопроходной (если не вводить обременительных количественных ограничений) из-за программирования переходов вперед. Но если раньше фаза синтеза выдавала практически готовый объектный модуль, в котором следующий просмотр должен был только подправить передачи управления, то в последних трансляторах фаза синтеза выдает весьма обобщенные заготовки команд, а фаза генерации сама выбирает подходящие команды и формирует их операнды, генерирует заголовки процедур и литеральные пулы.

У нас сложилась традиция, что транслятор выдает не только объектный модуль, но и текст на правильном языке ассемблера, в котором в качестве комментариев расставлены строки исходной программы. Во-первых, это заметно

облегчает отладку трансляторов, во-вторых, некоторые пользователи используют транслятор таким способом: транслируют программу, затем в тексте на ассемблере вручную вставляют действия, связанные с работой с нестандартным оборудованием, или вручную исправляют наиболее критичные по времени участки. В кросс-системах это особенно важно, т.к. часто на объектах нет инструментальной ЭВМ, а на целевой - кроме ассемблера ничего нет. Поэтому в процессе комплексной отладки изменения вносятся сначала на ассемблере, после накопления определённой порции исправлений они вносятся в тексты на исходном языке, затем повторяется цикл выпуска "единого машинного носителя", т.е. трансляция, редакция связей, объединение с массивами данных, формирование носителя в специальном формате, пригодном для развертки на целевой ЭВМ.

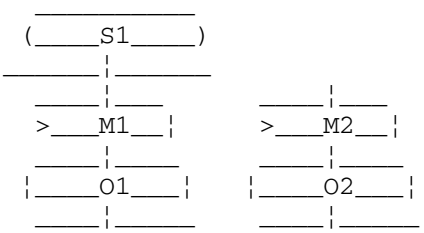
Выдача текста на ассемблере и различной дополнительной информации для последующей сборки и выпуска также входит в задачу фазы генерации объектного модуля.

Проектирование

Известно, что при разработке алгоритмического и программного обеспечения для повышения качества результирующего продукта нужно большую часть средств вкладывать в технологию разработки и только сравнительно небольшую часть - в заключительную проверку (по некоторым данным, соотношение примерно 80% и 20%). Применительно к алгоритмическому обеспечению (АО) высказанный тезис означает важность разработки инструментальных средств, позволяющих уйти от "разговорной" манеры описания и предоставляющих возможности документирования, модификации и содержательной проверки алгоритмов [27,28].

Если следовать генеральному сценарию использования SDL [29,30], предложенному МККТТ, проектирование встроенных систем осуществляется следующим образом (мы будем вести изложение на примерах и с использованием терминологии коммутационных систем, которые можно считать типичными):

1. 1. Каждый узел связи рассматривается как "черный ящик" с перечнем интерфейсов.
2. Для каждого интерфейса задается перечень и вербальное описание входных и выходных сигналов.
3. 3. Работа узла связи описывается в виде множества процессов, взаимодействующих друг с другом посылкой сообщений. В традициях проектирования "сверху вниз" строится дерево декомпозиции с постепенным уточнением состава и реализации процессов.
4. Работа каждого процесса описывается конечным автоматом, представленным в виде графа переходов, в узлах которого состояния, а ребра помечаются входными сигналами и некоторыми условиями.
5. В графе переходов задаются условия для перехода из одного состояния в другое, но вначале не даются алгоритмы этих переходов (способы определения условий, выдача сигналов и т.д.). Поэтому следующим шагом детализации является декомпозиция графа переходов: для каждого состояния (S1) строится отдельная SDL-диаграмма (рис. 1), состоящая из операторов (О1, О2) и проверок условий (как обычная блок-схема), а также из операторов посылки (М3) и приема (М1, М2) сигналов, причем каждая ветка должна кончаться переходом в какое-то состояние (S2, S3):



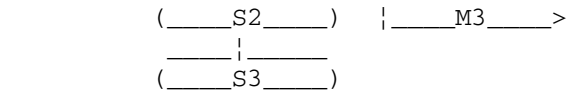


Рис. 1

У нас разработаны программные инструментальные средства, позволяющие строить SDL-диаграммы на экранах дисплеев, заносить диаграммы в архивы, корректировать их и печатать. Машинное ведение документации (как графической, так и текстовой) позволяет обеспечить неустаревающую документацию, легко модифицируемую и размножаемую, повысить взаимопонимание между различными разработчиками, выполнить простейшие формальные проверки (например, что все сигналы и условия входят в заранее заданный словарь, что каждому выходу есть соответствующий вход и т.д.), однако, разумеется, все это не заменяет настоящей семантической проверки.

На практике попытки формального отношения к генеральному сценарию SDL обычно приводят к неудачам, причем в этом никто не признается, просто функционалисты рисуют одни картинки, а программисты работают по другим. Попробуем разобраться в этой ситуации подробней.

В самой формулировке сценария заложено противоречие. Последовательность сигналов определяется не только их источником, но и зависит от работы получателя (различные формы подтверждения или "квитирования", наличие свободных и исправных устройств), например, абонент, подняв трубку и услышав длинный гудок, начнет набирать номер, но может возникнуть ситуация, когда неисправно соответствующее подключающее оборудование, и абонент, подняв трубку, услышит короткие гудки (или ничего не услышит) и тогда не начнет набирать номер.

Генеральный сценарий SDL есть только формальная оболочка, а ее конкретное наполнение является технической тайной фирм, которые не заинтересованы в предоставлении излишней информации конкурентам, поэтому в каждой организации разрабатываются свои методики и инструментальные средства проектирования АО.

Чтобы определить разумную реакцию на входной сигнал, нужно знать, какие приборы обрабатывают этот сигнал, как они между собой связаны, что они должны подключить, получив такой сигнал и т.д. Таким образом, чтобы определить структуру объекта, нужно знать входные сигналы и их последовательность, а для того, чтобы определить эту последовательность, нужно знать структуру объекта. В частности, задача проверки алгоритмической корректности внешних интерфейсов, в каком-то смысле, эквивалентна задаче построения всего ФПО.

Противоречие это снимается типичным для системного анализа способом - одновременной и постепенной детализацией станции и интерфейсов (в системном анализе говорят "системы и среды"). Практически это означает, что описание оборудования нельзя откладывать на самый последний момент, как это обычно делается.

У нас разработана система информационного обеспечения, позволяющая разработчикам аппаратуры представить узел связи в виде набора последовательно усложняющихся моделей, причем одна модель служит средством и дает терминологическую базу для моделей более высокого уровня. Иерархию этих моделей можно описать следующим образом.

На специальном формальном языке (построенном на базе Алгола 68) описываются отдельные приборы, для каждого прибора дается перечень конструктивных, монтажных и функциональных характеристик. Все описания приборов накапливаются в базе постоянных данных.

Затем описываются секции (стойки) и связи между ними, и формируется база станционных данных. В каком-то смысле, секция также относится к постоянным данным, поскольку серийно на заводе выпускаются не отдельные приборы, а целые секции, но нужно быть готовым к возможным изменениям в

конструкции. Заполнение базы станционных данных осуществляется в диалоговом режиме, причем формы таблиц и программы ввода генерируются автоматически на основе информации из базы постоянных данных.

Наконец, описанием абонентов и внешних интерфейсов заканчивается описание станции. Эта часть работы также выполняется в диалоговом режиме, опять-таки, программы ввода строятся автоматически на основе двух предыдущих баз данных.

Доступ ко всем базам данных возможен только через специальные процедуры доступа, перечень которых, а также виды параметров и результатов получаются автоматически, что обеспечивает надежную привязку программного обеспечения к аппаратуре.

Таким образом, система информационного обеспечения помогает в решении нескольких важных задач:

1. Обеспечивает формализацию представления о составе и структуре станции и ее интерфейсов.
2. Конкретизирует возможность описания алгоритмов, поставляя процедуры доступа к уже существующим базам данных, что обеспечивает возможность отладки алгоритмов на инструментальной ЭВМ (в совокупности с моделями оборудования).
3. Дает принципиальную возможность автоматизировать формирование станции по внешним спецификациям (указание необходимого числа типовых стоек, перечисление кабельных соединений и их порядок, тестирование межстоечных соединений и т.д.).

Граф переходов, даже с уточнениями на SDL для каждого состояния, дает слишком грубую, "поведенческую" модель. Попытки дальнейшего его уточнения методом декомпозиции отдельных операторов, учета состояний аппаратуры, введения так называемых "обратных веток", конкретизации сигналов в виде реальных команд коммутационного оборудования и т.д. приводят к громоздким схемам, для которых трудно построить адекватную программную реализацию.

По-видимому, дело в том, что исходное допущение об адекватности коммутационной системы конечноавтоматной модели истинно только на самых верхних (наименее детальных) описаниях системы. При более глубокой декомпозиции нужно привлекать дополнительную информацию, касающуюся не только состояний отдельных приборов, но и более глобальных взаимоотношений и связей. Например, некоторые условия могут зависеть от состояний многих приборов, т.е. от информации, собранной в различных таблицах, причем этими таблицами пользуются одновременно многие процессы. Особенно много таких ситуаций в процессе техобслуживания.

В более поздних рекомендациях МККТТ предлагается концепция расширенного автомата с памятью, но с математической точки зрения - это уже совершенно иная модель, требующая привлечения других понятий и реализационных механизмов, касающихся концепции локальных и глобальных данных, механизмов "видимости" объектов, понятия монопольного доступа к данным в критических интервалах.

Нами предлагается концепция построения АО и ПО на базе параллельно протекающих процессов. Вкратце эту концепцию можно изложить следующим образом. Процесс - это программа плюс некоторый сегмент данных. Часто используются разные процессы с одной и той же программой, но с разными сегментами данных, такие процессы называются экземплярами одного и того же "родового" процесса. Имеются специальные примитивы создания и уничтожения процессов (т.е. их сегментов данных). Главное отличие процесса от обычной процедуры состоит в том, что любой выход из процедуры приводит к уничтожению ее локальных данных, затем эту процедуру можно вызвать только заново. Процесс можно приостановить, затем продолжить, снова приостановить и т.д. Таким образом, в системе существует много процессов, большинство из которых находится в приостановленном состоянии, а один или несколько (по числу процессоров в системе) - активны.

Имеется несколько способов организации взаимодействия процессов, практически эквивалентных по выразительной силе, но отличающихся в деталях реализации и, следовательно, в эффективности, причем относительная эффективность сильно зависит от типа ЭВМ и ОС. Следуя концепции МККТТ, мы остановились на модели сообщений с примитивами "послать сообщение" и "принять сообщение с ограниченным временем ожидания".

Аварийные ситуации и ошибки в ПО не должны приводить к отказу всей системы. Для локализации таких ситуаций предусмотрены специальные средства, например, задание в процессе обработчиков исключительных ситуаций, причем операционная система обеспечивает вызов динамически ближайшего обработчика. Для того, чтобы избежать "зависших" процессов, мы не используем ожидания сообщения без ограничения времени, поэтому какое-то сообщение процесс всегда получит.

Очень тщательно ведется учет используемых ресурсов, причем ресурсом может быть не только память, время, конкретный прибор, но и абстрактный логический ресурс, что позволяет унифицировать работу с самыми разнообразными ресурсами. В частности, на понятии ресурса основан механизм обеспечения монопольного доступа в критических интервалах.

Принципы разбиения системы на процессы могут быть самыми разнообразными, например, один процесс на весь процесс обслуживания вызова или процесс только на часть тракта с тем, чтобы из таких процессов формировать разнообразные типы соединений, или же, наконец, отдельный процесс на каждое устройство, чтобы локализовать в этом процессе все данные об этом устройстве и обеспечить максимальную независимость ПО от конкретных устройств.

Аппаратная поддержка технологии программирования

В настоящее время используются сотни различных типов микро-ЭВМ с разнообразной архитектурой, причем многие из них имеют высокое быстродействие, но малый объем прямоадресуемой оперативной памяти. Создание для всех микро-ЭВМ трансляторов или кросс-трансляторов с АЯВУ, файловых систем, текстовых редакторов и других инструментальных средств - очень трудоемкая задача. В этих условиях мы с успехом использовали следующий технический прием: была создана одна унифицированная архитектура некоторой виртуальной (выдуманной) машины [31,32] и набор ориентированных на нее инструментальных средств, а на всех используемых микро-ЭВМ были реализованы ее интерпретаторы. При этом, кроме унификации программного обеспечения, обычно удавалось добиться экономии оперативной памяти в 3-5 раз ценой замедления счета в 2-4 раза.

В дальнейшем было решено разработать новую ЭВМ, получившую название "Самсон", удовлетворяющую требованиям встроенных применений, но одновременно достаточно удобную для использования в качестве инструментальной. Машины, ориентированные на АЯВУ, уже разрабатывались как у нас в стране, так и за рубежом [19,20,33], однако обычно получались очень сложные и дорогие ЭВМ, причем, на наш взгляд, сложность обуславливалась не столько ориентацией на АЯВУ, сколько универсальностью применений. Поскольку наша предметная область - управление и связь, была предпринята попытка ограничить класс решаемых задач так называемыми "задачами коммутационного типа". Действительно, для таких задач нужны в основном целая арифметика и логика. Однако для управления сетью связи необходимы диалог с оператором, файлы, обработка строк, для информационного обеспечения - эффективная работа с большими базами данных, для реконфигурации сети связи - сложные расчеты с плавающей точкой и т.д. Таким образом, и в этой предметной области класс решаемых задач очень широк.

На данный момент мы убеждены, что разрабатывать ЭВМ под конкретный узкий класс задач практически не имеет смысла, но для встроенных применений при современном состоянии нашей микроэлектроники какие-то ограничения нужно было искать.

Решение пришло довольно неожиданно. Решили ограничить класс входных языков. Опыт использования статических АЯВУ убедил нас, что транслятор с

такого языка способен взять на себя многие функции, обычно выполняемые аппаратурой. Действительно, если представить путь от задачи к результату в виде:

задача ---> транслятор ---> ЭВМ ---> результат,

причем никаких обходных путей здесь нет, понятно, что при одновременном проектировании транслятора и ЭВМ можно свободно перераспределять между ними обязанности. Очевидно, что стоимость трансляции для встроенных применений исчезающе мала по сравнению со стоимостью счета, поэтому транслятор выгодно нагружать, упрощая аппаратуру. С другой стороны, ортогональность архитектуры [34], отсутствие традиционных досадных ограничений и различных трюков резко упрощает синтез кода, так что в результате транслятор все равно получается существенно проще, чем для традиционных ЭВМ.

Перечислим некоторые особенности ЭВМ "Самсон".

1. В "Самсоне" не реализуется аппаратно какой-то один АЯВУ, но его система команд поддерживает основные операторы современных АЯВУ (вызов процедуры, вырезка элемента массива, циклы и т.д.). Сегодня уже можно утверждать, что зафиксировался "золотой фонд" основных конструкций АЯВУ, возможно различающихся по способу записи, но практически с одной и той же семантикой.

2. Система команд - безадресная, типа обратной польской записи. Например, (a + b) * (c + d) записывается как ab + cd + *. Эта запись придумана более полувека назад, она очень компактна и легко интерпретируется, но в ней очень легко ошибиться, причем ошибка будет обнаружена далеко не сразу, поэтому лишь немногие системы программирования (например, Форт) доверяют такую запись пользователю. Транслятор же переведет программу в обратную польскую запись без ошибок.

Основная часть команд кодируется одним байтом, более редкие команды кодируются 12 или 16 битами. Четыре наиболее частые команды кодируются 4 битами, например, загрузка одной из первых 16 переменных текущей процедуры в стек кодируется однобайтовой командой.

3. Есть три регистровых стека:

- стек целых - 16 позиций по 16 разрядов;
- стек вещественных - 8 позиций по 32 разрядов;
- стек адресов - 16 позиций по 40 разрядов.

Транслятор со статического АЯВУ всегда определяет, когда каким стеком нужно пользоваться; нет никаких аппаратных средств проверки переполнения или исчезновения стеков - за все отвечает транслятор. Таким образом "Самсон" имеет довольно большую внутреннюю регистровую память (40 регистров), причем имеется прямой доступ ко всем регистрам, а не только к верхушке стека, что позволяет хранить многие объекты (например, переменные циклов) не в памяти, а на регистрах.

Эффективное распределение регистров является сложной задачей, однако современные трансляторы с ней успешно справляются. Еще одним аргументом против использования регистров в архитектуре ЭВМ является замедление вызова процедуры, связанное с необходимостью сохранения регистров в момент ее вызова. "Самсон" обеспечивает для каждой процедуры независимую нумерацию регистров (от верхушки ко дну стека), что позволяет выгружать при вызове не все регистры, нужно только обеспечить вычисленный при трансляции "аппетит" вызываемой процедуры. При возврате из процедуры нет необходимости сразу же восстанавливать выгруженные регистры, возможно, до использования их придется снова выгружать (такую ситуацию можно назвать "толкотней" регистров). Таким образом, эту стратегию можно сформулировать следующим образом: выгружать регистры, когда стек близок к переполнению, восстанавливать - когда стек близок к исчерпанию.

4. Чтобы не тратить время на выборку команд из памяти, в ЭВМ "Самсон" предусмотрена специальная схема "водопровод", осуществляющая подкачку команд на фоне работы процессора. Ширина буфера водопровода - 8 байт, ширина шины доступа к памяти - 4 байта. Скорость водопровода такова, что

временем выборки команд можно пренебречь во всех случаях, кроме команд передачи управления.

5. "Самсон" - микропрограммная ЭВМ, собранная на секционных микропроцессорных наборах (в основном, серии 1804). Для реализации довольно сложной системы команд "Самсона" была разработана система автоматизации микропрограммирования на базе Алгола 68 [35], которая позволила обойтись вообще без микроассемблера (на конец 1987 года нам не была известна ни одна промышленная система микропрограммирования на АЯВУ). Микропрограммирование на Алголе 68 оказалось таким удобным, что было решено поставлять эту систему вместе с "Самсоном". Микропамять "Самсона" составляет 4К слов по 64 разряда, причем это ОЗУ, куда можно в процессе счета загружать микропрограммы для различных предметных областей, что обеспечивает выигрыш по времени счета в 4-5 раз.

6. Известно, как облегчает жизнь разработчиков программного обеспечения виртуальная память, однако встречается она в специализированных машинах крайне редко из-за сложностей реализации. В "Самсоне" эта задача решается также с помощью трансляторов.

Память делится на сегменты переменной длины, размер каждого сегмента определяется транслятором. В памяти могут храниться только математические адреса, состоящие из номера сегмента и смещения; непосредственно матадресом пользоваться нельзя, нужно предварительно загрузить его в адресный стек. В процессе загрузки обычным образом (через таблицу сегментов) матадрес преобразуется в физический адрес, стоит это довольно дорого (около 10 тактов процессора), зато загруженным адресом можно пользоваться много раз без каких-либо накладных расходов. ЭВМ, ее операционная система и трансляторы обеспечивают неперемещаемость сегмента по памяти, пока на него есть ссылки из адресного стека. Трансляторы оптимизируют использование команд загрузки адресного стека, в частности, выносят эти команды из циклов. Проверку на невыход адреса за границы сегмента в большинстве случаев берут на себя тоже трансляторы.

Заключение

Основными результатами данной работы являются:

1. Исследованы принципы создания технологии программирования на базе АЯВУ для сложного класса программно-аппаратных систем.
2. Реализован набор технологических инструментальных средств.
3. Разработана методология и произведено сравнение языков программирования с целью определения их пригодности в заданной области. Обосновано использование в качестве базовых языков технологии статических языков типа Алгол 68 и Ада.
4. Разработана техника трансляции класса статических АЯВУ, реализовано более 15 трансляторов и кросс-трансляторов.
5. Создана оригинальная ЭВМ "Самсон", ориентированная на данную технологию программирования.

Предложенные методы и инструментальные средства с успехом использованы в нескольких крупных системных заказах.

Литература

1. **Ершов А.П.** *Отношения методологии и технологии программирования.* Труды II Всесоюзной конференции "Технология программирования" Киев, 1986 г., стр. 10-13.

2. *ОСТ 4.091.283-87. Создание программной продукции. Общие требования к модульно-векторной промышленной технологии.*
3. **Морозов В.П., Терехов А.Н.** *Внедрение в производство языков высокого уровня*. В сб. Трансляция и преобразование программ. Новосибирск, ВЦ СОАН СССР, 1984 г., стр. 20-29.
4. **Терехов А.Н.** *Промышленное программирование на базе языков высокого уровня*. В сб. Методы трансляции и конструирования программ. Новосибирск, ВЦ СОАН СССР, 1986 г., стр. 131-136.
5. **Йенсен К., Вирт Н.** *Паскаль*. М., ФиС, 1986 г.
6. **Вирт Н.** *Программирование на языке Модула 2*. М., Мир, 1987 г
7. *Пересмотренное сообщение об Алголе 68*. М., Мир, 1979 г.
8. **Дейкало Г.Ф., Новиков Б.А., Рухлин А.П., Терехов А.Н.** *Новые средства программирования для ЕС ЭВМ. Транслятор с языка Алгол 68 и диалоговая система JEC*. М., ФиС, 1984 г.
9. *ГОСТ 27974-88, ГОСТ 27975-88. Язык программирования Алгол 68 и Алгол 68 расширенный*. М., Госкомитет СССР по стандартам, 1989 г
10. *ГОСТ 27831-88. Язык программирования Ада. М., Госкомитет СССР по стандартам 1989 г.*
11. **Терехов А.Н.** *Развитие системы программирования на базе языка Алгол 68*. В сб. Информатика и программирование. Новосибирск, ВЦ СОАН СССР, 1989 г., стр. 107-109.
12. **Терехов А.Н.** *"Самсон" как средство для рабочего места программиста*. В сб. АРМ программиста. Новосибирск, ВЦ СОАН СССР, 1988 г., стр. 26-32.
13. **Майерс Г.** *Надежность программного обеспечения*. М., Мир, 1980 г.
14. **Терехов А.Н., Цейтин Г.С.** *Язык синтеза объектной программы с учетом последующего контекста*. В сб. Труды всесоюзного симпозиума по методам реализации новых алгоритмических языков. Новосибирск, ВЦ СОАН СССР, 1975 г., стр. 227-236.
15. **Терехов А.Н.** *Процессы идентификации и структура компилятора с языка Алгол 68*, Программирование, №2, 1975 г.
16. *Алгол 68. Методы реализации*. Под ред. Г.С. Цейтина. Изд. ЛГУ, Ленинград 1976 г.
17. **Фоминых Н.Ф.** *Выделение классов эквивалентности и упорядочивание видов в трансляторе А68ЛГУ*. В сб. Всесоюзная конференция по методам трансляции. Новосибирск, ВЦ СОАН СССР, 1981 г., стр. 180-183.
18. **Cocke K., Markstein P.W.** *Measurement of program improvment algorithms* // Proc IFIP Congress-80. Amsterdam, 1980, p. 221-228
19. **Бабаян Б.А., Сахин Ю.Х.** *Система "Эльбрус"*. Программирование №6, 1980 г., стр. 72-86.
20. **Пентковский В.М., Зайцев А.И., Коваленко С.С., Фельдман В.М.** *Структура высокопроизводительного микропроцессора с аппаратной поддержкой языков высокого уровня*. Программирование, №3, 1990 г., стр. 24-35.
21. **Branquart P., Cardinael J.P., Lewi J.** *An optimized translation process and its applications to ALGOL 68. Part 1. General Principles.* MBLE Lab, de Rech, 1972, Report R204.
22. **Терехов А.Н., Цейтин Г.С.** *Средства эффективного синтеза объектной программы*. Программирование №6, 1975, стр. 38-48.
23. **Терехов А.Н.** *Распределение регистров в рабочей программе*. Программирование №1, 1977 г. стр. 37-41
24. **Терехов А.Н.** *Реализация Алгола 68 на ЕС ЭВМ*. В сб. Всесоюзный симпозиум по перспективам развития в системном и теоретическом программировании. Новосибирск, 1978 г., стр. 109-113.
25. **Терехов А.Н.** *Библиотечные вступления и отдельная трансляция процедур в трансляторе с Алгола 68*. Тезисы докладов и сообщений к всесоюзной конференции. Часть 2, Паланга-Вильнюс, 1980 г., стр. 253-255.

26. **Терехов А.Н.** *Реализация кросс-систем, транслирующих программы на языке Алгол 68 в коды спецЭВМ*. IV Всесоюзный симпозиум по системному и теоретическому программированию. Кишинев, 1983 г., стр. 354-356.
27. **Терехов А.Н.** *Повышение качества разработки алгоритмического обеспечения встроенных систем*. В сб. Адаптируемые средства программирования. Методы оценки трансляторов. Кишинев, 1989 г., стр. 122-127.
28. **Терехов А.Н.** *Технология программирования встроенных систем*. В сб. Информатика и программирование. Новосибирск, ВЦ СОАН СССР, 1989 г., стр. 5-16.
29. *Язык программирования для телефонных станций с программным управлением. Оранжевая книга*. МККТТ. М.,"Связь", 1979 г.
30. **Барздинь Я.М., Калниньш А.А., Стродс Ю.Р., Сыцко В.А.** *Язык спецификаций SDL/plus и его приложения*. Рига, 1982 г.
31. **Матиясевич Ю.В., Терехов А.Н.** *16-разрядная виртуальная ЭВМ, ориентированная на АЯВУ*. Программирование микропроцессорной техники. Таллин, 1984 г., стр. 68-72
32. **Матиясевич Ю.В., Терехов А.Н., Федотов Б.А.** *Унификация программного обеспечения микроЭВМ на базе виртуальной машины*. Автоматика и телемеханика №5, М., 1990 г.
33. **Майерс Г.** *Архитектура современных ЭВМ*. М., Мир, 1985 г.
34. **Фоминых Н.Ф.** *Разработка архитектуры микроЭВМ, ориентированной на языки программирования высокого уровня*. Программирование микропроцессорной техники. Таллин, 1984, стр. 59-67.
35. **Фоминых Н.Ф.** *Использование стандартного расширяемого алгоритмического языка в микропрограммировании*. В сб. Диалоговые микрокомпьютерные системы. Изд. МГУ, 1986 г., стр.108-115
36. **Терехов А.Н.** *Особенности технологиии разработки ПО для персональных компьютеров*. В сб. Программное оснащение персональных компьютеров. Изд. МГУ, 1990 г., стр. 68-71