Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Бульонков М. А., Кочетов Д. В.

ВИЗУАЛИЗАЦИЯ СВОЙСТВ ПРОГРАММ

Препринт
51

Новосибирск 1998

В работе предлагаются модель свойств программ, не зависящая от языка программирования и предметной области, и основанный на этой модели визуализатор свойств программ HyperCode, разработанный для интеграции в системы автоматизированного сопровождения программного обеспечения.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Michael A. Bulyonkov, Dmitry V. Kochetov

VISUALIZATION OF PROGRAM PROPERTIES

Preprint
51

Novosibirsk 1998

This paper presents a model of program properties. This model does not depend on programming languages and application areas. Also properties visualizer HyperCode is described in the paper. This tool is implemented for integration into different programming and software reengineering systems.

ВВЕДЕНИЕ

Жизнь любого достаточно серьезного программного продукта состоит из двух сильно различающихся по протяженности этапов — этапа первоначальной разработки, заканчивающегося выпуском продукта в свет, и этапа сопровождения. Первый этап компактен и замкнут, характеризуется единством команды разработчиков, технологий, проектных и функциональных спецификаций и т.д. Второй этап, который может длиться десятилетиями, — период столкновений программы с реальным миром, что, независимо от степени первоначальной проработанности проекта, ведет к внесению многочисленных и отнюдь не столь стройных, как исходный проект, модификаций; это период болезни роста, когда введение новых компонентов в программу скрывает из вида ее основу; это период текучки кадров, а значит, утраты знаний об устройстве отдельных компонентов и взаимосвязи между ними. В этот период сопровождаемая программа приобретает качества, существенно снижающие возможность последующей ее модификации, а именно:

- становится эклектичной, поскольку последовательно модифицируется разными программистами, со своими стилями, привычками, правилами идентификации и т.д., и этого не избежать даже при разнообразной административной регламентации труда программистов;
- все больше напоминает кунсткамеру — коллекцию временных решений, загадочных операторов, черных ящиков-функций; положение усугубляется хроническим нежеланием или неумением программистов документировать свои программы;
- утрачивает строгую структурированность, и потоки данных в ней становятся чрезвычайно запутанными;
- резко увеличивается в размерах, выходя за границы возможностей “ручного” понимания.

Учитывая анализ практики длительного сопровождения программного обеспечения и накопленную массу этого обеспечения, становится понятно, что программы все больше читаются и все меньше пишутся, подавляющая часть программистской деятельности относится к пониманию уже созданных компонентов, а не их модификации или разработке новых. Следовательно, возникает острая потребность в инструментах, способствующих этой деятельности, — интегрированных средах, объединяющих в себе анализ и визуализацию свойств программ и позволяющих, используя самые разнообразные представления программ, изучать информацию, накопленную анализом.

В системном программировании первое звено технологии “анализ-визуализация” программ изучено наиболее полно. Во-первых, практически любой транслятор проводит много видов анализа, необходимых для генерации объектного кода и оптимизации, по крайней мере, идентификацию, типовый анализ, анализ информационных зависимостей, анализ синонимов. Даже обладание такой “обыденной” информацией позволяет значительно ускорить понимание программы. Во-вторых, самые разнообразные и специализированные анализаторы программ интенсивно разрабатываются и как самостоятельные инструменты. Но использованию этого богатого инструментария в целях *понимания* свойств программ препятствует ряд обстоятельств:

1) такие инструменты жестко связаны с некоторым языком программирования, в то время как сложное программное обеспечение чаще всего является многоязыковым;

2) информация, накапливаемая трансляторами, не рассматривается как отдельный результат и не доступна пользователю; в лучшем случае, пользователю доступны результаты использования результатов анализа, которые могут иметь самые разнообразные формы — от оптимизированного кода до предупреждений транслятора — но восстановить по ним первичные результаты проведенного анализа программы крайне затруднительно;

3) анализаторы программ, разрабатываемые как отдельные инструменты, чаще всего носят статус академических разработок, в которых очень мало внимания уделяется представлению полученных результатов, — чаще всего это текстовые отчеты. Пользователь вынужден сам связывать эти результаты с элементами исследуемой программы, при этом он, зачастую, должен работать с информацией, по размерам заметно превосходящей исходный программный текст. Как следствие производительность программиста оказывается низкой.

Таким образом, можно утверждать, что в обсуждаемой предметной области, в отличие, к примеру, от трансляции программ, качество и глубина анализа малого стоят, если интерпретация его результатов и привязка этих результатов к тексту исследуемой программы требуют дополнительной трудоемкой деятельности. Нахождение свойств и их изучение пользователем — две составляющие единого процесса. Пользователю нужен *визуализатор свойств*, который бы привязал их к исследуемой программе, чтобы, отталкиваясь от текста и структуры программы, пользователь мог в интерактивном режиме изучить свойства интересующих его фрагментов программы, формулировать запросы о них, использовать наглядные графические представления свойств и в процессе изучения этих свойств уже “зря-

че", направленно принимать содержательные решения о развитии программы.

1. ПОСТАНОВКА ЗАДАЧИ

Проблема понимания особенно остро стоит в контексте сопровождения и перепроектирования сверхбольших программных комплексов и систем, разработанных десятилетия назад на ныне устаревших платформах и языках программирования. Такие системы в англоязычной терминологии называются legacy systems, а процесс их перепроектирования — reengineering technology.

В последнее время, в связи с проблемой 2000 года [6,12], этому направлению стало уделяться особое внимание. Первой причиной, по которой авторы обратились к проблеме визуализации свойств разветвленных многокомпонентных программ, стало участие в проекте Rescueware®, разрабатываемом фирмой Relativity Technologies Inc™, целью которого, помимо других, была автоматизация решения проблемы 2000 года для приложений на языке Кобол. Это было, в некотором роде, движение от промышленного программирования. С этой же проблемой авторы столкнулись при разработке экспериментальной среды смешанных вычислений M2Mix для языка Модула-2 [4,7], когда понадобился инструментарий для автоматизированного повышения специализируемости (приспособленности к смешанным вычислениям) программ, а его использование потребовало от пользователя изучения результатов, специфичных для специализации видов анализа — анализа периода связывания и анализа конфигураций, — с которыми обычный пользователь не знаком. Стремление объединить эти исследования привело к следующей постановке задачи.

1. Необходима *модель свойств программ*, в которой должны выражаться результаты анализа и запросы пользователя к визуализатору и на которой должно основываться представление программ в визуализаторе. Требования, предъявляемые к такой модели:

- достаточная выразительная мощность для представления разнородных видов анализа программ и определения преобразований программ;
- минимум преобразований в анализаторах для генерации результатов в терминах этой модели;

- концептуальная простота, предельно упрощающая интерактивное взаимодействие пользователя с визуализатором и восприятие результатов анализа;
- независимость от языка программирования и конкретного набора изучаемых свойств (видов анализа, результаты которых изучаются).

2. Необходим *визуализатор свойств*, сопряженный с этой моделью, который должен носить форму "открытого компонента", допускающую интеграцию визуализатора в различные системы поддержки и сопровождения программного обеспечения.

Решение этой задачи, выразившееся в создании визуализатора свойств программ HyperCode, разработанного в лаборатории смешанных вычислений Института систем информатики СО РАН, является предметом дальнейшего обсуждения.

2. МОДЕЛЬ = СТРУКТУРА + СУЩНОСТИ + СВЯЗИ

Применимость, естественность и универсальность модели свойств программ определяется тем, насколько она приближена к процессу понимания программы человеком. По мнению авторов, ключевой элемент в этом процессе — структура программы, ее синтаксическое дерево. Со структуры начинается восприятие большой программы, по структуре определяется граф управления, структура задает информационные потоки в программе, и вообще, все вопросы, возникающие в процессе понимания, либо обращены к структуре программы, либо формулируются как запросы о семантических свойствах узлов дерева. То есть программист определяет *вид фрагмента*, который его интересует, *контекст* этого фрагмента, а уже затем пользователь представляет, какими свойствами он может обладать, что о нем можно узнать, на что влияет его исполнение. И, наконец, что очень важно для нашей цели интерпретации свойств в терминах элементов программы, — только структура программы жестко *привязана к тексту* и служит мостом между текстом и результатами анализа. Итак, понятие структуры для нас определяется перечнем видов синтаксических конструкций, вхождения которых — *фрагменты* — образуют исследуемую программу, отношением вложенности, заданном на фрагментах, и отображением, сопоставляющем каждому фрагменту его текстовое представление.

Теперь попытаемся определить систему формальных понятий, в которой могли бы быть выражены свойства синтаксических конструкций. При

этом мы не ставим себе цель охватить искомой моделью любые мыслимые свойства, а руководствуемся следующими критериями:

- моделируемые классы свойств должны быть широко распространены в практике сопровождения программного обеспечения;
- изучение моделируемых классов свойств в значительной степени состоит из рутинной, технической деятельности, допускающей автоматизацию;
- искомые формализмы должны быть максимально абстрактными и примитивными, а их совокупность должна образовывать минимальную полную систему для моделирования свойств.

В поисках таких свойств и формализмов обратимся к классическим видам анализа: идентификации и анализу информационных зависимостей. Идентификация устанавливает соответствие между синтаксическими конструкциями “переменная” и “описание”, сопоставляя каждой переменной ее описание, а каждому описанию — множество использующих вхождений. Таким образом, результаты идентификации — это отношение над множеством синтаксических конструкций вида “переменная” и множеством синтаксических конструкций вида “описание”. Это отношение, в зависимости от интересующего нас направления, или *конца отношения*, является либо отношением типа “один-ко-многим” и, для примера, называется “использующие вхождения”, либо отношением типа “много-к-одному” и называется “определяющее вхождение”. Теперь, мысля в этих же терминах, рассмотрим анализ информационных зависимостей. Обнаруживаем, что результаты этого анализа есть не что иное, как отношение между операторами и вхождениями переменных, устанавливающее связь между результатами присваивания и их использованиями, имеющее тип “много-ко-многим” как со стороны оператора, так и со стороны использующих вхождений переменных. Список таких видов анализа, результаты которых являются отношениями над синтаксическими конструкциями, можно продолжить — построение графа вызовов и графа управления, анализ синонимов, анализ “зацепленности” переменных. Фактически, моделируемые отношениями свойства — это всевозможные взаимосвязи между элементами программы произвольных уровней — от вхождений переменных и операторов до модулей и языковых подсистем. Этот класс свойств, во-первых, наиболее распространен, во-вторых, прекрасно формализуется абстрактным аппаратом реляционной алгебры и, в-третьих, наиболее сложен для “ручного” изучения.

Таким образом, первый класс свойств определен. Это *отношение*, для определения которого в модели необходимо задать

- *концы отношения* — виды синтаксических конструкций, участвующих в отношении;
- *способ вхождения* конца в отношение — один или много;
- *названия отношения*, рассматриваемого с каждого из концов.

Следующая большая группа свойств, к которой мы обратились, — это результаты различных видов разметки (аннотации) программ. За этими свойствами закрепилось название “атрибуты”. Они являются отображениями синтаксических конструкций в абстрактные области значений, *домены*, или множества с такими доменами в качестве базовых типов. Можно перечислить множество видов анализа, вычисляющих подобные свойства, — анализ периода связывания, отображающий вхождения переменных во множество значений (доступно, задержано); целая группа видов анализа, выполняемых для оптимизации в трансляторах и находящих булевскую разметку; анализ достижимости операторов; анализ константности выражений; анализ используемости вычислений. Общим для всех этих видов анализа является то, что используемые в них домены — это конечные множества атомарных значений, т.е. перечислимые типы, для определения которых достаточно перечислить все константы. Ограниченные таким способом разметки программ также образуют класс свойств, удовлетворяющий сформулированным выше свойствам.

Итак, определен второй класс свойств, включаемый в нашу модель, — *атрибут* синтаксической конструкции с областью значений, являющейся перечислимым типом. Для определения атрибута в модели необходимо задать

- *тип синтаксической конструкции*, к которой привязан атрибут;
- *область значений атрибута* — домен или конечное множество атомарных значений.

Предположим, что, ограничившись отношениями и атрибутами, никакие другие свойства в модель мы включать не будем. Насколько универсальной и мощной окажется такая модель? В ее основе лежат реляционная алгебра и механизм абстрактной аннотации. Эти системы не предъявляют никаких требований к предметной области. Они же предоставляют набор методов для конструирования сложных свойств из примитивных элементов. Наконец (и это уже эмпирическое утверждение), практически все известные авторам свойства моделируются этими двумя классами свойств, по крайней мере результаты всех видов анализа, использовавшихся в проектах Rescueware и M2Mix, были определены в предлагаемой модели.

Теперь мы готовы окончательно определить модель свойств программ. Программа есть совокупность вхождений синтаксических конструкций из

заданного множества видов синтаксических конструкций. На вхождениях синтаксических конструкций определены отношение вложенности и отображение в текстовое представление. Для каждого вида синтаксической конструкции определен набор атрибутов и множество отношений, в которых этот вид участвует. Для каждого атрибута определена область значений — конечное множество атомарных значений. Для каждого отношения определен способ участия его концов.

3. СРАВНЕНИЕ ПОДХОДОВ

После окончательного определения модели возникает естественный вопрос: как предлагаемая модель связана с другими подходами к изучению многокомпонентных формальных систем?

Незначительно изменим терминологию: будем называть виды синтаксических конструкций сущностями, а узлы синтаксического дерева — вхождениями сущностей. Тогда оказывается, что в основе предлагаемой модели лежит хорошо известная в базах данных модель “диаграмма сущностей-связей” или “Entity-Relationship Diagram” (ERD). Эта модель прекрасно себя зарекомендовала как эффективное средство проектирования, изучения и визуализации баз данных [9]. Естественно ожидать, что и в наших целях применение этой модели должно себя оправдать. Но модель, предлагаемая авторами, развивает ERD-модель следующим образом:

- вводится выделенное, реализующее структуру программы отношение вложенности, которое стоит над всеми другими отношениями, равноправными между собой;
- к формальным объектам, образующим модель, добавлено текстовое представление изучаемых объектов

Далее будем использовать термин “сущность” наряду с терминами “синтаксическая конструкция”, “вхождение синтаксической конструкции” и “фрагмент”.

Предлагаемую авторами модель можно рассматривать и с иной точки зрения — как наложение на текст программы структуры гипертекста, еще одной универсальной модели визуализации сложных взаимосвязей. Примеры использования гипертекста при разработке программного обеспечения описываются, в частности, в [5,8,14]. Но мы доопределили общий вариант гипертекста:

- носителями ссылок, или связей, могут служить только сущности программы;

- связи типизованы;
- для каждой сущности определена ссылка на предка в синтаксическом дереве;
- с одой сущностью может быть связано несколько ссылок, т.е. ссылкам приписаны аннотации;
- в носитель ссылок (как текстовый фрагмент документа) могут быть вложены другие носители ссылок.

Фактически, мы объединили достоинства традиционного иерархического представления программ с представлением реляционным, или графовым, избавившись от ограниченности их обоих — жесткой заданности у древесной интерпретации и излишней аморфности и отсутствия базиса у гипертекстовой. Тем самым мы выразили дуализм программы, представляющей собой структуру с внутренними взаимосвязями.

Наконец, необходимо отметить, что, совмещая гипертекст и ERD-модель в единой технологии, мы получаем возможность менять направленность процесса изучения свойств программ — мы изучаем либо свойства конкретных текстовых фрагментов, неявно манипулируя с абстрактными объектами (сущностями), либо формальные понятия, обладающие текстовыми представлениями. Выбор конкретного варианта зависит от контекста использования модели и уровня квалификации конечного пользователя.

4. МОДЕЛЬ КАК АРГУМЕНТ ВИЗУАЛИЗАТОРА

При разработке визуализатора, основанного на описанной выше модели, необходимо сделать выбор между двумя формами его реализации — жестким включением в визуализатор конкретной модели с ее набором сущностей, атрибутов и отношений и передачей заданной модели в качестве аргумента. Очевидно, что первый вариант в принципе не приемлем для решения поставленной нами задачи — разработки визуализатора, не зависящего от предметной области и предназначенного для изучения свойств многоязыкового программного обеспечения. Но даже без требования универсальности преимущества модели, “зашитой” в визуализатор, т.е. специализации визуализатора под конкретный язык программирования и/или набор свойств, на поверку оказываются мнимыми. Модель как отдельный аргумент позволяет, во-первых, управлять степенью детализации изучаемой программы за счет изменения набора типов сущностей; во-вторых, позволяет, за счет выбора необходимых атрибутов и отношений, задавать интересующую часть семантики программ, отсекая избыточную информацию;

в-третьих, в случае включения в рассмотрение новых видов анализа, нет необходимости изменять визуализатор, достаточно изменить модель в соответствии с новым набором подготовленных данных.

Строго говоря, в каждом конкретном случае встраивания визуализатора в системы программирования модель, разрабатываемая исходя из потребностей пользователя и реализованных видов анализа, оказывается фиксированной. Но в предлагаемой авторами реализации визуализатора эта фиксация носит внешний характер по отношению к самому инструменту. Весь контекст использования от визуализатора скрыт. По существу, визуализатор превращается в интерпретатор с задаваемой переданной ему моделью семантикой, свободный от каких бы то ни было знаний о языках программирования и видах анализа программ. В частности, HyperCode без изменений использовался как

1) составная часть среды смешанных вычислений для языка Модула-2;
2) составная часть Rescueware — промышленной системы сопровождения и переноса программного обеспечения на языке Кобол;
3) инструмент отладки кодогенерирующей части компилятора языка Модула-2 фирмы XDS™[15], сопоставляющий фрагментам исходной программы полученные по ним фрагменты на ассемблере и накопленную отладочную информацию.

5. ПОДГОТОВКА ДАННЫХ ДЛЯ ВИЗУАЛИЗАТОРА

Выше уже описана подготовка модели. В Приложениях 1 и 2 описывается реальная модель, использовавшаяся в проекте Rescueware.

Второй частью данных является так называемый *проект* визуализатора. Его содержание делится на две составляющие. Первая — это совокупность модулей (файлов), составляющих программу, — их имена и нумерация, используемая в определении структуры программы. Вторая — это наполнение модели — структура программы с привязкой к тексту и накопленные внешними анализаторами свойства изучаемой программы, представленные в виде отношений и атрибутов, определенных заданной моделью.

Конкретная форма представления — предмет отдельного изучения. Требования к ней очевидны: адекватное отражение ERD-модели и структуры программы, с одной стороны, и простота отображения данных в эту форму для внешних анализаторов — с другой. Сначала авторами было выбрано представление в виде текстовых файлов фиксированного формата. Но хотя

такое представление и отвечало заданной спецификации, его поддержка потребовала включения в визуализатор компонентов, в каком-то смысле посторонних, вынуждавших отвлечься от содержательных видов деятельности. После дополнительного анализа было принято решение представлять данные в виде баз данных. Во-первых, базы данных семантически очень близки ERD-модели. Во-вторых, выбор баз данных дает возможность использовать в визуализаторе для работы с отношениями и конструирования запросов мощные средства манипулирования данными, предоставляемые современными СУБД. В-третьих, и это следует особо отметить, выбранный путь приводит к созданию единого базиса для всей технологии сопровождения программ "анализ — визуализация — преобразование", а именно глобальной базы данных проекта. Эта база хранит структуру и все найденные свойства программы, она же используется для изучения этих свойств в визуализаторе, она же предоставляет механизмы для формирования и выполнения заданий на преобразование программ с последующим анализом и замыканием всего цикла. Эта же база данных используется для хранения истории проекта, его автоматизированного документирования. В частности, в системе Rescueware реализована именно такая технология.

Структура базы данных определяется в большой степени субъективными факторами. В качестве примера опишем базу данных, использованную в HyperCode (нижеприведенная диаграмма). База данных основана на двух видах нумераций: нумерации всех модулей проекта и нумерации всех сущностей внутри одного модуля. Соответственно каждой сущности сопоставляется уникальный ключ, состоящий из двух номеров — номера модуля и номера внутри этого модуля. База данных состоит из трех таблиц: **Структура**, **Отношения**, **Атрибуты**.

Таблица со структурой состоит из полей:

FileId, **EntityId** — ключ сущности, состоящий из двух полей,

FromRow, **FromCol**, **ToRow, ToCol** — поля, задающие привязку сущности к тексту в терминах прямоугольных и линейных координат в тексте модуля,

EntityType — тип сущности,

Caption — текст сущности (или начало текста, если он длинный),

NestLevel, **Parent, SeqNo** — поля, задающие местоположение узла в дереве.

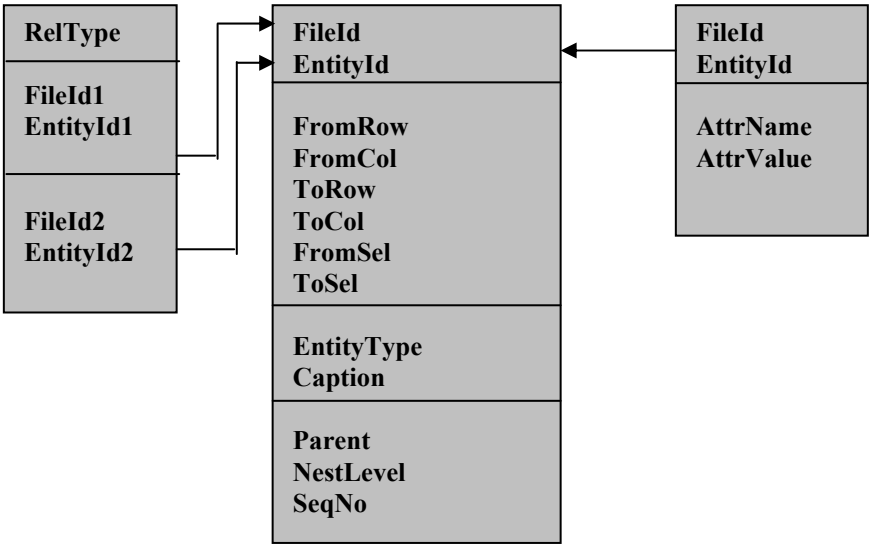


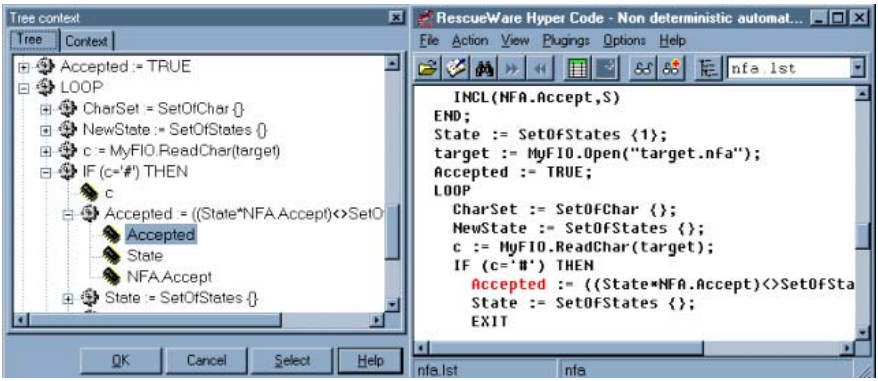
Таблица с атрибутами состоит из полей:
FileId,**EntityId** — ключ сущности,
AttrName, AttrValue— имя и значение атрибута.

Таблица с отношениями состоит из полей:
RelType — имя отношения,
FileId1,**EntityId1** — ключ левой сущности,
FileId2,**EntityId2** — ключ правой сущности.

6. ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ ВИЗУАЛИЗАТОРА

Перемещение по программе

Во время работы в HyperCode в каждый момент времени определены *текущий модуль* и *текущая конструкция*. Текущий модуль параллельно представляется в виде исходного текста и синтаксического дерева. Пользователь может перемещаться по модулю и выбирать текущую конструкцию как структурно (раскрывая и сворачивая узлы дерева, от отца к сыну, от узла к брату и т.д.), так и в текстовом режиме, когда текущей оказывается синтаксическая конструкция, охватывающая позицию курсора. В дереве каждый узел изображается вместе с пиктограммой, соответствующей типу узла, и начальным текстом узла, что позволяет быстро ориентироваться при поиске необходимых сущностей в дереве. Все перемещения происходят либо параллельно в обоих представлениях, либо независимо, когда совмещение представлений выполняется отдельной командой. Кроме того, для текущей конструкции предоставляется структурное окружение — ее контекст в синтаксическом дереве. Типичный внешний вид сессии HyperCode изображен на следующем рисунке:



Свойства сущности

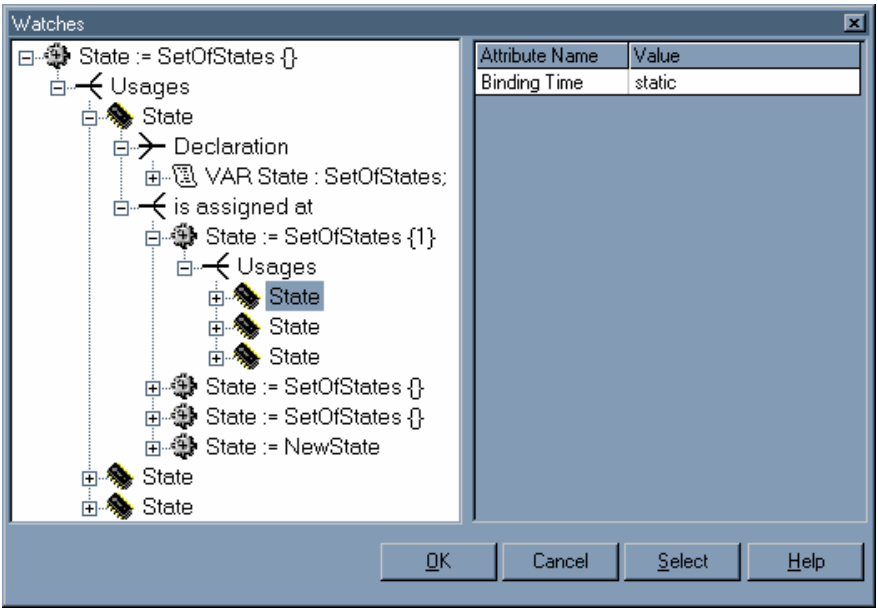
После выбора текущего узла пользователь может интерактивно через совокупность динамических меню выполнить следующие действия:

1) узнать значение интересующего атрибута, выбрав его из списка атрибутов, определенных для вида выбранного узла дерева;

2) перейти к конструкции (сделать ее текущей), состоящей в интересующем нас отношении (отношение точно так же выбирается из меню, содержащего все концы отношений, определенные для данного типа сущностей), либо получить список таких узлов, если их много, и выбрать затем уже из него;

3) получить суммарную информацию о значениях всех атрибутов данной сущности и обо всех конструкциях, входящих с данной конструкцией в какое-либо отношение.

Особенно интересна последняя возможность. Она представляет изучаемый узел в виде корня дерева, сыновьями которого являются отношения, а “внуками” — узлы, входящие в соответствующие отношения с корнем и являющиеся, в свою очередь, корнями поддеревьев, устроенных по такому же принципу. Во-первых, это позволяет изучать взаимосвязи не на атомарном уровне, а в более крупных единицах — подграфах. Во-вторых, это интересный пример визуализации графовых структур в виде деревьев (теоретически бесконечных). В-третьих, таким представлением мы реализуем иерархию, определяемую не структурой программы, а выбранными связями выбранного узла или множества узлов. Внешний вид этого представления (в HyperCode он получил название “Watch”-режима) показан на следующем рисунке:



Представленный перечень возможностей кажется, на первый взгляд, примитивным. Но, учитывая все многообразие возможных отношений и атрибутов, эти действия уже позволяют сделать многое, например: от использования переменной перейти к ее описанию, по оператору присваивания получить весь список использования его результата, найти все вызовы заданной процедуры. Другой пример: поскольку возможно образовывать отношения с сущностями из разных файлов, то мы, во-первых, получаем инструмент для изучения традиционно трудных для исследования результатов межмодульного анализа, а во-вторых, получаем возможность изучать взаимосвязь разных программ (в частности написанных на разных языках), например программ на ассемблере и языке высокого уровня.

Глобальное выделение информации

Очень часто пользователю необходимо выделить какую-то информацию глобально во всем тексте программы, а не в какой-то конкретной конструкции. В HyperCode возможно глобальное графическое (цветом или шрифтом) выделение следующих видов:

- сущностей заданного типа;
- сущностей заданного типа в соответствии со значением заданного атрибута, т.е. когда каждому значению из соответствующего данному атрибуту домена приписывается свое выделение, а текст конструкции данного типа выделяется по значению этого атрибута;
- сущностей заданного типа, у которых не определены заданные атрибуты;
- сущностей заданного типа, которые не входят в заданные отношения;
- комбинации из предыдущих четырех видов.

Файл с заданными выделениями может быть записан в графических форматах, например в реализации для ОС Windows это rich text format. Таким способом HyperCode реализует простейшую форму отчуждаемых отчетов о программе, являющихся полезной формой автоматически генерируемой документации по программе. Кроме того, что эта информация полезна сама по себе, она может использоваться для сравнения различных версий программного обеспечения в терминах заданных свойств.

Интерфейс, определяемый пользователем

Из предыдущего описания можно выделить часть интерфейса, которая, с одной стороны, сопряжена с моделью визуализации, поскольку задается определенными в модели объектами, с другой — относится скорее к внешнему представлению изучаемой семантики, чем к ней самой. Во-первых, это пиктограммы, сопоставляемые типам сущностей, во-вторых — это цвета и шрифты для глобальных выделений информации. Для повышения дружественности визуализатора пользователю предоставлена возможность динамически настраивать эту часть интерфейса через систему диалогов. Пользовательские настройки сохраняются в файле модели в виде специальных уравнений в секциях, соответствующих сущностям.

Режимы поиска

Одним из наиболее традиционных, а зачастую и единственным используемым методом изучения свойств программ является обычный текстовый поиск. Действительно, несмотря на уменьшение информативного веса текста программы в предлагаемом нами подходе, за счет отношений и структуры, последние не могут в полной мере заместить изучение самого текста программы. Но HyperCode, используя модель свойств, позволяет сущест-

венно усложнить запросы к текстовому поиску. Вообще, для систем автоматизированного сопровождения программ характерна тенденция использовать усложненный абстрактными представлениями текстовый поиск как один из основных методов.

В первую очередь, текстовый поиск в HyperCode усложняется средствами определения тех частей программы, в которых он проводится. Изучая практику использования поиска при понимании программных фрагментов, можно сделать следующие выводы. Во-первых, пользователю нужен поиск не во всем тексте, а в узлах дерева заданного вида, например: в операторах присваивания, если проводится поиск всех точек изменения переменной; в строковых константах, передаваемых параметрами в процедуры, если ищутся все выводы формата данных; в описаниях, если проводится поиск всех переменных массивов. Во-вторых, пользователь зачастую обладает дополнительной информацией о тех фрагментах программы, которыми следует ограничить поиск. Такая информация может быть выражена как условия, наложенные на заданные атрибуты или отношения, например поиск в операторах, результаты которых используются, или поиск в выражениях, достижимых по управлению. В соответствии с этими выводами реализован поиск в визуализаторе — это поиск либо во всем тексте, либо в тексте сущностей заданных типов. Во втором варианте возможно ограничить поиск текстом тех сущностей, заданные атрибуты которых имеют конкретные значения, включая граничные случаи определенности или неопределенности этого атрибута. Также возможно включить в ограничения условия включения в заданные отношения.

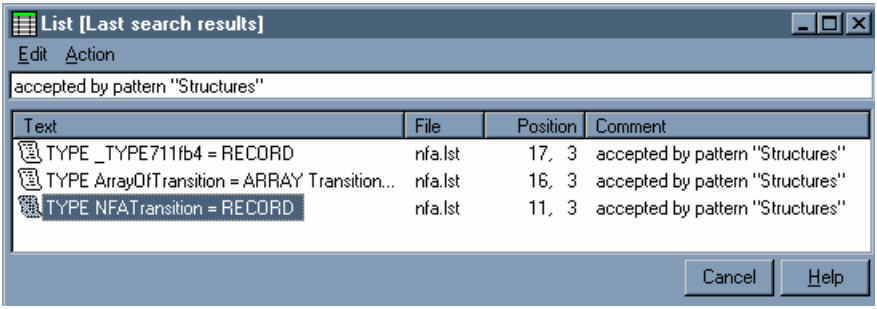
В визуализаторе существует еще один вариант ограничения того текста, в котором необходимо проводить поиск. Выше, при описании структуры базы данных изучаемой программы, упоминалось поле **Caption,** в котором хранится начало текстового фрагмента, соответствующего сущности. Это поле используется не только при построении дерева, но и при специальном виде поиска, который проводится не в самом тексте, а в этом столбце базы данных и выполняется средствами SQL — с помощью предиката *like*, реализующего поиск одной из разновидностей регулярных выражений. Этот вариант был включен в визуализатор в связи с обеспечиваемой средствами СУБД высокой эффективностью такого поиска. Данный вид поиска может быть ограничен такими же способами, как и обычный поиск, — типами сущностей, значениями атрибутов и вхождениями в отношения. Область применения такого поиска в базе данных — поиск в часто встречающихся конструкциях малого текстуального объема, таких как переменные, константы, арифметические выражения. На конструкциях с большими тексто-

выми представлениями (в текущей реализации — больше 80 символов) поиск становится некорректным. В таких случаях необходимо воспользоваться обычным поиском регулярных выражений, также реализованным в HyperCode.

Списки узлов

При изучении программ часто возникает необходимость группирования конструкций программы во множества для их совместного изучения или применения массовых операций. Например, типичная схема при решении проблемы 2000 года, встречающаяся во многих инструментах, выглядит следующим образом: по какому-либо набору критериев выделяется список "подозрительных" конструкций, который затем применением конкретизирующих действий расщепляется на множества "хороших" узлов, не связанных с проблемой 2000 года, и "плохих", требующих преобразований. Еще один распространенный случай — формирование списка конструкций, требующих модификации в связи с изменением внутреннего представления в трансляторе, например определений типов данных и описаний переменных, связанных с этим представлением. Поскольку списки, по сути, являются множествами, то к ним естественным образом применимы теоретико-множественные операции: создание, удаление, копирование, включение и удаление элементов, объединение, пересечение и т.п. Очень полезной кажется возможность при включении элемента в список задать комментарий, объясняющий причины включения элемента в данное множество. Выше описывались действия, которые можно рассматривать как "спискообразующие" операции: перемещение по отношению типа "один-ко-многим" и текстовый поиск, результатом которого является список синтаксических конструкций, в тексте которых был найден заданный образец. Эти операции автоматически генерируют комментарии для элементов списков-результатов. Наличие множеств позволяет доопределять функции, определенные для атомарных областей, и факторизовать их применение. Примером такого использования служат текстовый поиск, ограниченный конструкциями из заданных списков, и генерация текстовых отчетов для заданных списков. Наконец, широкие возможности для логической организации работы со списками и внешнего представления множеств предоставляет *типизация списков в модели*. Тип списка определяет виды сущностей, которые могут включаться в списки этого типа (что позволяет фильтровать добавляемые элементы при обширных модификациях списков). Также тип списка определяет ту информацию о элементах списка, которая предостав-

ляется пользователю при визуализации списка — имена модулей, текстовые позиции, атрибуты. Так, список, являющийся результатом поиска, представляется следующим образом:



НАСЛЕДОВАНИЕ В ERD-МОДЕЛИ

При разработке сложных моделей для языка Модула-2, полностью отражающих его синтаксис, мы столкнулись с проблемой абстрактных типов синтаксических конструкций, т.е. таких, которые в программе присутствуют не явно, а в виде конкретизаций — узлов других типов. Типичные примеры родовых типов — описания и операторы. Проблема состоит в том, что с точки зрения проектирования модели именно абстрактные конструкции являются носителями свойств, общих для всех конкретизаций, но задать атрибуты и отношения можно только для конкретизаций, поскольку только они обладают местом в структуре программы и текстовым представлением. Лежащее на поверхности решение — дублировать общие свойства в определении каждого из конкретизирующих типов сущностей — оказывается совершенно непригодным. При таком подходе скрывается общность природы изучаемых свойств, разрушается единство в методах исследования программы, по сути, скрывается ее семантическая структура. Необходимость ориентироваться в перегруженных формальных системах и всеохватывающее дублирование всех понятий — от компонентов модели до запросов к визуализатору — вытесняет из деятельности пользователя ее основное содержание. Ошибочен и выбор другой крайности — отказ от конкретизаций и определение модели только в терминах абстрактных конструкций. Такой подход значительно снижает глубину изучения программы и детализацию ее представления.

Реализованное авторами решение состоит в воспроизведении обсуждаемой иерархии сущностей в модели точно в таком же виде, как это принято в объектно-ориентированных языках программирования при разработке иерархий классов: есть абстрактные классы (сущности) и их методы (атрибуты и отношения) и есть расширения этих классов, методы которых дополняют базовый набор. Пользователь, указывая в общении с визуализатором интересующие его типы сущностей, сам определяет интересующий его уровень иерархии с соответствующим набором свойств, тем самым поднимая на этот уровень все конкретизации. Степень качественного изменения ERD-модели при введении в нее иерархии сущностей особенно выразительно демонстрируется с помощью графического представления этой модели: если исходный вариант представляется плоской диаграммой, то в ERD-модели с иерархией появляется новое измерение — модель становится пространственной.

В описании модели, используемой в HyperCode, иерархия сущностей задается специальным атрибутом **INHERITS,** определенным для каждого типа сущности. Значением этого атрибута является тип сущности, свойства которого наследуются.

ОТ ВИЗУАЛИЗАЦИИ К АНАЛИЗУ

Самые сильные стороны реализованного в HyperCode подхода — универсальность и независимость от набора внешних инструментов — оказываются одновременно и самым уязвимым местом, поскольку лишают визуализатор самостоятельности. Строго говоря, подобная подчиненность визуализатора предполагалась исходно — визуализатор и разрабатывался для сопряжения с внешними анализаторами. Но в основе сопряжения лежит перевод свойств программы из одной формальной системы в другую, из изучаемой семантики в семантику ERD-модели. И этот перевод позволяет применить в визуализаторе новые методы анализа программ, неизвестные и недоступные в исходном контексте, в котором были получены данные для визуализатора — структура, атрибуты и отношения. Эти данные являются основой для конструирования новых свойств визуализатором, который, таким образом, становится универсальным анализатором программ.

Операции над отношениями

Первая группа таких методов — хорошо известные из теории множеств и реляционной алгебры операции над отношениями: объединение, пересе-

чение, суперпозиция и проекция отношений. Использование таких операций в визуализаторе связано с тем, что, во-первых, эти операции просты для понимания, их необходимые комбинации может определить и малоквалифицированный пользователь, что важно при применении визуализатора в промышленном программировании; во-вторых, наличие отношения вложенности существенно увеличивает область применения методов этой группы. Например, имея отношение *R1* между выражениями и их аргументами-переменными и отношение *R2* между переменными и их описаниями, пользователь может, сконструировав суперпозицию отношения вложенности и суперпозиции *R1* и *R2* и взяв затем проекцию результата на множество описаний процедур, получить отношение, сопоставляющее каждой процедуре описания переменных, определяющих передачу управления в теле этой процедуры.

Срезы программ

Перечисленные выше операции над отношениями уже предоставляют мощный базис для анализа программ, но в HyperCode они в явном виде не реализованы. Вместо этого разработан общий механизм, включивший в себя операции над отношениями как частный случай. Этот механизм называется “конструктор срезов программы”. Его описание начнем с обсуждения трех часто решаемых программистами задач:

1) найти все фрагменты программы, использующую данную переменную (данное множество переменных);

2) найти все фрагменты программы, меняющую данную переменную (данное множество переменных);

3) найти граф вызовов для данной процедуры.

Список подобных задач весьма представителен. Их объединяет необходимость введения некоторого фильтра для структуры и текста программы, позволяющего выделить ее интересующую часть. Общим для всех подобных задач является следующее. Их решение — это некоторый **срез программы**. В процессе изучения любой достаточно сложной программы последняя редко воспринимается как единое целое — она рассыпается на совокупность срезов — частей программы, объединенных какой-либо внутренней логикой и предназначенных для решения какой-либо подзадачи. Набор таких срезов в зависимости от целей динамически меняется и конкретизируется в процессе изучения программы. По сути, вычленение некоторого среза и его последующее преобразование — основной вид деятельности при сопровождении программного обеспечения. Построение среза

определяется некоторой сущностью, а сам срез является некоторым множеством сущностей, т.е. срез — это отношение. Соответствующее срезу отношение обычно не является базовым, а строится на основе более примитивных (структуры программы, информационных зависимостей и т.п.).

В силу вышесказанного можно утверждать, что построение срезов — это динамически задаваемый, определяемый в терминах ERD-модели анализ программ, который может быть реализован не внешними инструментами, а только компонентами визуализатора. В поисках способов построения срезов мы обратились к широко распространенному способу анализа программ — потоковому анализу [3,11,13]. Потоковый анализ можно описать, как распространение информации по дереву программы, начиная с некоторого начального множества сущностей и применяя недетерминированно к уже достигнутым вершинам набор выражающих специфику конкретного вида потокового анализа правил, задающих переходы к новым вершинам. Сами правила, в терминологии ERD-модели, определяются как переходы от одного конца отношения к другому (включая и частный, но особенно важный случай перехода по отношению вложенности). При выполнении потокового анализа выделяются два множества сущностей — множество активных (достигнутых) сущностей **Active** и множество накопленных результатов **Result** (обычно это некоторое подмножество **Active).** Таким образом, выполнение потокового анализа — это вычисление транзитивного замыкания отношения, заданного в виде какой-либо комбинации базовых отношений.

Для вычисления срезов в HyperCode мы использовали модификацию описанного алгоритма, в которой множества **Active** и **Result** отождествляются. Такая модификация определяется спецификой понятия среза — для его вычисления важен, зачастую, не столько сам результат потокового анализа, сколько история его получения, т.е. все конструкции, участвовавшие в конструировании транзитивного замыкания.

Для определения конкретного варианта потокового анализа в нашей модели необходимо задать набор правил, имеющих вид уравнений, где правая часть — определение множества конструкций, включаемых в результат при добавлении в него конструкции вида, указанного в левой части. Определение может быть одним из следующих:

1) непосредственный предок в дереве,
2) ближайший предок заданного типа,
3) сыновья в дереве,
4) сыновья заданного типа,
5) конструкции, участвующие с данной в заданном отношении.

В качестве примера, в котором используется модель из Приложения 1, приведем набор уравнений с очевидным синтаксисом для нахождения среза программы, меняющего и использующего заданную переменную (так называемый Business Rule Extraction):

```
STAT      = Uses
VAR       = .PARENT.STAT
VAR       = .CHILD.VAR
STAT      = .PARENT.
STAT      = .CHILD.VAR
PERFORM   = .CHILD
PERFORM   = Uses
PERFORM   = .PARENT
PAR       = Calls
VAR       = Declaration
DECL      = .PARENT
```

Построенный визуализатором срез представляется в двух формах. Первая — это обычный список узлов. Вторая — текстовое представление среза, сконструированное из текстовых представлений включенных в срез конструкций, из которых удалены фрагменты, соответствующие тем вложенным конструкциям, которые в срез включены не были. Эта форма в явном виде реализует фильтры над текстом программы, выделяя к тому же интересующую часть в виде новой программы, возможно синтаксически и неправильной, но, тем не менее, строго специфицирующей интересующую часть исходной программы.

Модель потокового анализа применяется в визуализаторе не только для построения срезов, но и для конструирования собственно отношений, для чего используется исходный алгоритм потокового анализа, в котором множества **Active** и **Result** различаются. Соответственно, набор правил для построения отношений отличается от набора правил для конструирования срезов — для каждого правила определяется то множество (**Active** или **Result),** в которое должны включаться узлы, вычисляемые этим правилом. Кроме набора правил для построения отношений необходимо определить вид сущностей, для которых это отношение строится. Сам процесс построения выглядит как сопоставление каждой конструкции заданного типа множества **Result,** вычисленного для стартового множества, состоящего из этой конструкции.

ПРАКТИКА ИСПОЛЬЗОВАНИЯ

Технология, обсуждаемая в данной работе, разрабатывалась в течение последних полутора лет. Накоплен значительный опыт по ее реализации в различных программных процессорах и системах программирования — как непосредственно в виде визуализатора HyperCode, так и в виде других инструментов, основанных на той же модели свойств, что и HyperCode. Полученные результаты можно разделить на несколько групп:

- реализация специализированных процессоров на основе визуализатора;
- способы интеграции визуализатора в комплексные системы программирования;
- развитие функциональности визуализатора.

Специализированные процессоры

Самый короткий и простой путь к процессору на основе обсуждаемой в данной работе технологии — это использование HyperCode как отдельного инструмента для изучения результатов фиксированного набора уже реализованных анализаторов. При таком подходе определяются две изолированные друг от друга фазы — анализ программы (и генерация базы данных) и сессия в HyperCode. Единственное, что нужно для такой реализации, — включение в анализаторы компонентов, формирующих базу данных. Такой способ использовался для создания инструментов конечного пользователя из анализаторов, разработанных в лаборатории смешанных вычислений Института систем информатики СО РАН, например анализатора отношений равенства программных термов в императивных программах и анализатора синонимов [1,2,4,10].

Эксперименты с анализаторами показали главное достоинство такого стиля использования визуализатора — минимальность затрачиваемых усилий, поскольку необходимые преобразования анализаторов требуют несколько человеко-дней, как и проектирование конкретной модели, ввиду локальности предметной области. Главный же недостаток — отсутствие обратной связи, невозможность передачи накопленной информации от визуализатора к другим программным процессорам, поскольку единственной формой отчуждаемых результатов оказываются генерируемые визуализатором отчеты о программе. Несколько смягчает этот недостаток расширение визуализатора текстовым редактированием, позволяющее частично использовать накопленную информацию для преобразований внутри самого визуализатора, что было сделано, например, в операционной среде сме-

шанных вычислений, где HyperCode использовался, в частности, для улучшения специализируемости программ.

Менее принципиальный недостаток исходной модели, выявленный при экспериментах с анализаторами, состоит в обязательном определении областей значений для атрибутов сущностей. Например, при анализе периода связывания специализатор в качестве атрибутов сущностей накапливает комментарии, объясняющие результаты связывания. Эти комментарии — произвольные строки. Соответственно, невозможно определить домен для такого атрибута. Поэтому мы сняли это требование, разрешив не определять в модели области значения атрибутов и сделав невозможными некоторые действия, связанные с атрибутами, в частности раскраску программы по значениям атрибутов сущностей (для которой набор значений должен быть зафиксирован). Но такие ограничения отражают природу этих атрибутов — потенциальную неограниченность их областей значений.

Интегрированные системы

Максимально полно использовать информацию, накопленную с помощью нашего визуализатора возможно только при его интеграции в комплексную систему, все компоненты которой используют и модифицируют единое многослойное представление программы. В такой системе визуализатор переходит с уровня конечного инструмента, завершающего технологическую цепочку, на уровень компонента, качественно повышающего степень автоматизации и точность преобразований при разработке программного обеспечения. Фактически, визуализатор становится связующим звеном между анализаторами и преобразователями программ, преобразующим через интерактивное взаимодействие с пользователем данные, полученные от анализаторов, в аргументы для преобразователей. Пример такой системы —проект Rescueware. Основой этой системы являются так называемый *репозиторий проекта* — база данных, содержащая совокупность внутренних представлений изучаемого программного обеспечения, и понятие *маршрута* (roadmap) — одного из типичных для сопровождения программ набора шагов, т.е. последовательного создания нужных совокупностей внутренних представлений компонентами Rescueware. Визуализатор HyperCode включается в эти маршруты в тех точках, где требуется визуализация представлений или вмешательство пользователя для принятия решения о дальнейшем пути продолжения сценария, которое принимается на основе информации, собранной в процессе визуализации. Суммируя опыт участия в этом проекте, особенно важно отметить следующее.

1) В этой системе визуализатор через общую базу данных передает новую информацию не только для использования в преобразователях, но и для конкретизации и настройки анализа — локализации анализируемой части, уточнения контекстных условий, передачи дополнительных знаний, зачастую повышающих точность анализа.

2) Интеграция визуализатора стала мотивом для реализации дополнительных методов передачи информации во внешний контекст, выраженных в терминах исходной модели (т.е. сохраняющих исходную универсальность и открытость технологии), — хранение накопленных списков сущностей в виде дополнительных таблиц базы данных визуализатора, возможность модификации пользователем значений атрибутов сущностей и определения новых атрибутов, реализация специального языка для передачи аргументов от визуализатора и в терминах базы данных визуализатора внешним инструментам, работающим с этой же базой данных.

3) Использование визуализатора в системах, ориентированных на промышленное программирование, логически разбивается на два этапа, на каждом из которых предъявляется свой набор требований к пользователям. Первый этап — разработка модели для визуализатора, включая не только совокупность базовых понятий (виды сущностей, атрибуты, отношения), но и множество необходимых срезов программ, образцы поиска, даже конкретный набор списков сущностей, которые должны существовать в визуализаторе. Такая задача может быть решена только высококвалифицированным программистом, досконально знающим целевую предметную область, семантику изучаемых свойств, наконец, используемый язык программирования. Решение этой задачи — разовый шаг, который может быть выполнен даже до отчуждения системы программирования. Второй этап — собственно изучение свойств конкретной программы в зафиксированной модели и специализированном на нее визуализаторе, т.е. чисто техническая деятельность. На этом этапе может быть привлечена большая команда с не столь высокой, как на первом этапе, квалификацией пользователей, задания для которых определяются в примитивной, предельно детализированной форме. Сам факт существования модели свойств может быть скрыт от этих пользователей — модель растворена в интерфейсе визуализатора, который, строго говоря, определяется динамически по модели—аргументу, но для таких пользователей он статичен. Суммируя все вышесказанное, можно утверждать, что визуализатор свойств оказывается процессором второго порядка, шаблоном для массовой генерации конкретных инструментов за счет его специализации относительно модели свойств. Так, довольно неожиданно, авторы вернулись к области смешанных вычислений, экспери-

менты в которой послужили одной из причин, побудивших заняться визуализацией свойств программ.

Завершая обсуждение опыта использования предлагаемых в данной работе модели анализа и визуализации свойств программ, нужно отметить, что эта технология воплотилась не только в визуализаторе HyperCode. Во-первых, она была использована еще в нескольких, также разработанных совместно с фирмой Relativity Technologies Inc. процессорах, специализированных на конкретные предметные области, например: проблема 2000 года для программ на Коболе и Си и изучение межмодульных потоков данных в приложениях на Коболе. Во-вторых, в последнее время в рамках системы программирования XDS [15] начата разработка базирующейся на визуализации свойств инкрементальной среды программирования, объединяющей разработку программ с трансляцией, анализом и визуализацией свойств программ.

ЗАКЛЮЧЕНИЕ

Предложенная модель визуализации свойств программ и основанный на ней визуализатор HyperCode представляют собой мощную технологию изучения свойств сложных многокомпонентных и многоязыковых программных комплексов. Перспективность этого направления обеспечивается использованием опыта из различных предметных областей, универсальностью и открытостью подхода и подтверждается успешной реализацией экспериментальных и промышленных программных продуктов, включавших в себя либо описанную модель, либо непосредственно визуализатор HyperCode.

СПИСОК ЛИТЕРАТУРЫ

1. **Бульонков М.А., Кочетов Д.В.** *Грамматический подход к анализу синонимов* // Программирование. — 1996. — No.3. — С.36—46.
2. **Емельянов П.Г.** *Методы и средства статического анализа семантических свойств программ*: Автореф. дис. на соиск. учен. степ. канд. физ.-мат. наук. — Новосибирск, 1997. — 18 с.

3. **Котов В. Е., Сабельфельд В.К.** *Теория схем программ*. — М.: Наука. Гл. ред. физ.-мат. лит., 1991. — 248 с.
4. **Кочетов Д.В**. *Алгоритмы анализа для эффективной специализации алголоподобных программ* // Средства и инструменты окружения программирования. — Новосибирск: Институт систем информатики, 1995. — С. 85—100.
5. **Покровский С.Б., Степанов Г.Г.** *Среда разработки программного обеспечения, основанная на гипертексте* // Там же. — С. 101—110.
6. **Bergnel H.** *The Year-2000 Problem and the New Riddle of Induction* // Communs. ACM. — 1998. — Vol. 41, No.3. — P.13—17.
7. **Bulyonkov M.A., Kochetov D.V**. *Practical Aspects of Specialization of Algol-Like Programs* // Lect. Notes Comput. Sci. — 1996. —Vol.1110. — P.17—32.
8. **Cybulski Jacob L., Reed K.** *A Hypertext Based Software-Engineering Environment* // IEEE Software. — 1992. — Vol.9, No.2. — P.62—68.
9. **Date C.J.** *An Introduction to Database Systems*. Vol. 1. — Reading: Addison-Wesley, 1986.
10. **Emelianov P.G., Baburin D.E**. *Semantic Analyzer of Modula-programs* // Proc. of the Fourth International Static Analysis Symposium. — Berlin a.o.: Springer Verlag, 1997. — P. 361—363. — (Lect. Notes Comput. Sci.; Vol.1302).
11. **Kam J. B., Ullman J.D.** *Monotone Data Flow Analysis Frameworks* // Acta Informatica. — 1977. — V.7, No.3. — P.305-318.
12. **Kappelman L.A., Fent D., Keeling K. B. Prybutok V.** *Calculating the Cost of Year-2000 Compliance* // Communs. ACM. — 1998. — Vol. 41, No.2. - P.30-39.
13. **Kildall G. A.** *A Unified Approach to Global Program Optimization* // Conf. Records of ACM Symp. On Principles of Programming Languages. — Boston:MA, 1973. — P.194—206.
14. **Stotts P. D., Furuta R., Cabarrus C.R.** *Hyperdocuments as Automata: Verification of trace-Based browsing properties by Model Checking* // ACM Transact. on Information Systems. — 1998. — Vol.16, No.1. — P.1—30.
15. XDS Family of Products. *Native XDS-x86 for Microsoft Windows NT and Windows95* .Version 2.30. User's Guide - XDS Ltd., 1997.

Модель свойств программ (текстовое представление)

```
[DOMAINS]
# секция, перечисляющая домены
Y2KStat=Y2KStatus

[Y2KStat]
bad=
good=
suspicious=

[ENTITIES]
# секция, перечисляющая сущности в форме уравнений вида
# <короткое имя> = <длинное имя>, где
# <короткое имя> - имя сущности, используемое в данных
# <длинное имя> - имя, используемое в интерфейсе
VAR=Variable
DECL=Declaration
PAR=Paragraph
STAT=Statement
PERFORM=Paragraph perform
PORT=Port

[PORT]
# секция сущности PORT, перечисляющая
# ее атрибуты в форме уравнений вида
# <имя атрибута> = <имя домена>,
# имя домена может быть опущено
DIRECT=
PORTNAME=
PORTTYPE=
[VAR]

[DECL]
Cases=
DataType=
Fraction=
Length=
Name=
Occurs=
Picture=
Usage=
Y2K Status = Y2KStat
Y2K Method=
Y2K Pivot Date=
```

```
[STAT]

[PERFORM]
.INHERITS=STAT

[PAR]

[RELATIONS]
# секция, перечисляющая отношения
Vardecls = Declaration of variable
Calls = Declaration of procedure
Depends = DefUse chains
PrtMatch = Port matching
Ports = Ports and Declarations
Renfrom = Renames from
Rento = Renames to
Redef=Redefines

[Vardecls]
LeftName=Declaration # имя левого конца отношения
RightName=Instances  # имя правого конца отношения
LeftType=M  # способ вхождения левого конца в отношение
RightType=1 # способ вхождения правого конца в отношение
LeftEntity=VAR   # тип левого конца отношения
RightEntity=DECL     # тип правого конца отношения

[Calls]
LeftName=Declaration
RightName=Calls
LeftType=M
RightType=1
LeftEntity=PERFORM
RightEntity=PAR

[Depends]
LeftName=Uses
RightName=Define
LeftType=M
RightType=M
LeftEntity=STAT
RightEntity=STAT

[PrtMatch]
LeftName=Output Ports...
RightName=Input Ports...
LeftType=M
```

```
RightType=M
LeftEntity=PORT
RightEntity=PORT
```

[Ports]

```
LeftName=Variable Declarations...
RightName=Ports...
LeftType=M
RightType=M
LeftEntity=PORT
RightEntity=DECL
```

[Renfrom]

```
LeftName=Renames from
RightName=Renamed by
LeftType=M
RightType=1
LeftEntity=DECL
RightEntity=DECL
```

[Rento]

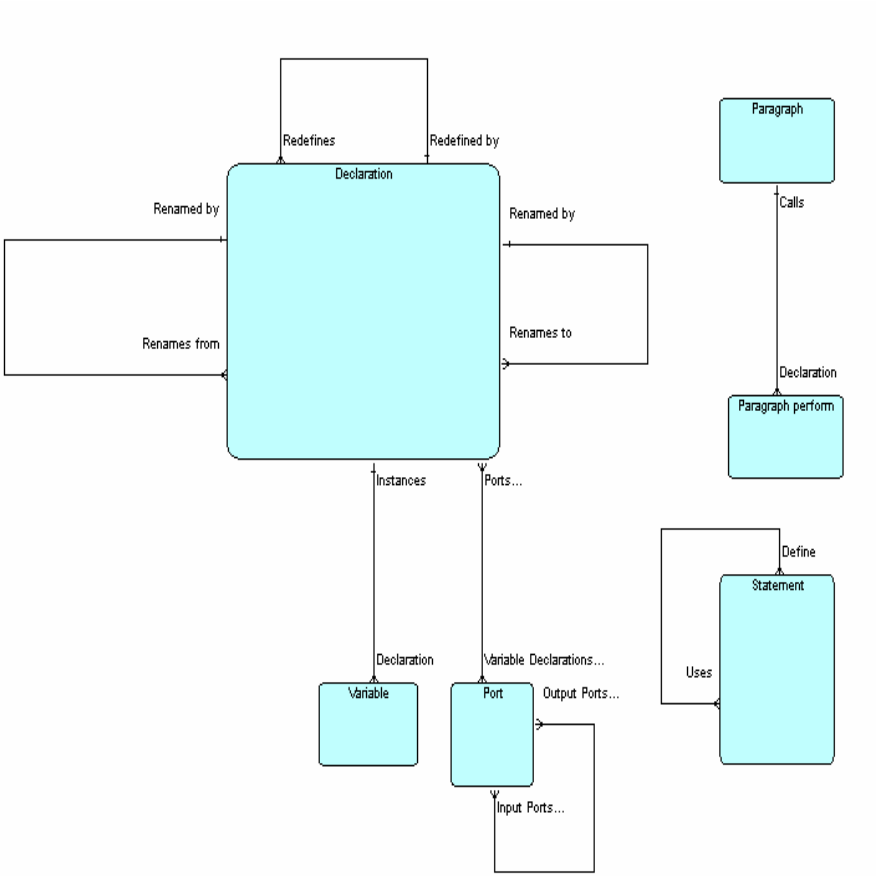
```
LeftName=Renames to
RightName=Renamed by
LeftType=M
RightType=1
LeftEntity=DECL
RightEntity=DECL
```

[Redef]

```
LeftName=Redefines
RightName=Redefined by
LeftType=M
RightType=1
LeftEntity=DECL
RightEntity=DECL
```

Приложение 2

Модель свойств программ (ERD-представление)



М. А. Бульонков , Д. В. Кочетов

ВИЗУАЛИЗАЦИЯ СВОЙСТВ ПРОГРАММ

Препринт
51

Рукопись поступила в редакцию 19.05.98
Рецензент А. Н. Терехов
Редактор Л. А. Карева

Подписано в печать 10.09.98
Формат бумаги 60 × 84 1/16 Объем 2.1 уч.-изд.л., 2.3 п.л.
Тираж 75 экз.

Отпечатано на ризографе “AL Group” 630090, г. Новосибирск: пр. Акад. Лаврентьева, 3