Об информационных технологиях в фундаменте "вавилонской башни" цифровизации – не пора ли изменить архитектуру проекта?

Сердечный Александр Петрович

Аннотация

Охарактеризованы проблемы современных информационных технологий (ИТ) мало пригодных для широкого применения во всех сферах деятельности и не обеспечивающих формирование единого информационного пространства.

Сформулированы необходимые требования к новым принципам организации распределённых баз данных, обеспечивающие их естественную интеграцию. Предложен экономически перспективный способ реализации общедоступных ИТ и средств интеграции данных в рамках существующих технологий.

Изложена авторская концепция потоковой обработки данных реализуемая в мультикомпьютерных комплексах предельно простой архитектуры. Представлен пошаговый план перехода на прогрессивные технологии потоковой обработки и общего информационного пространства.

Разработаны принципы построения языка описания блок-схем модулей для программирования параллельной потоковой обработки данных, применимого как на уровне декомпозиции модулей блок-схем вплоть до машинных команд, так и при оперировании любыми структурами с возможностью подстановки функций как аргументов, подобно как в объектно-ориентированных или функциональных языках.

Ключевые слова: dataflow, поток данных, архитектура ЭВМ, базы данных, параллельные вычисления, язык программирования, потоковая обработка данных, суперкомпьютер

Содержание

1. Неразрешимые проблемы в ИТ

В основе современных информационных технологий (ИТ), как и более полувека назад, лежит стандартная парадигма, по которой, для того чтобы получить результаты вычислений или обработать исходные данных, *необходимо запустить* на электронной вычислительной машине (ЭВМ) компьютерную программу, которая выполнит ввод исходных данных, обсчитает их, и выдаст результаты на указанное в программе выходное устройство.

Получением доступа к файлам и устройствам, предоставлением и освобождением памяти, управлением очерёдностью и временем исполнения программ – то есть *управлением процессами* (*Controlflow*) исполнения задач и распределением затребованных ими ресурсов занимается операционная система (ОС).

Разных моделей ЭВМ (часто секретных) было разработано много, и у каждой свой набор программных кодов. Поэтому озаботились разработкой единого языка высокого уровня. Сначала Фортрана, а потом более продвинутого Алгола. А потом ещё и ещё (сейчас их количество около 30 и увеличивается) – всё более продвинутых, вплоть до языков объектно-ориентированного программирования (ООП), либо и вовсе функциональных.

А в языках ООП важно знать иерархию классов объектов, структуру, свойства и методы объектов – ибо даже одинаковые операторы языка могут работать с объектами по разному. И вот даже зная что надо делать и зная язык – без изучения применяемых в каждой прикладной сфере объектов, вы ничего не сможете.

Тем более, что в разных сферах применения принято использовать разные языки. А для каждой задачи даже в одной отрасли разными разработчиками придумываются разные классы объектов, так что воспользоваться в одной задаче объектами другой, особенно если эти задачи разрабатывались в разных организациях, довольно сложно, ибо требует документированного согласования и может даже совместной разработки средств обмена данными.

Если сравнить программирование с вождением автомобиля, то раньше было везде понятно куда рулить, но в разных автомобилях были разные рычаги и кнопки, хотя на любом можно было ездить в любом городском районе. А теперь в каждом квартале города свои дорожные правила, и в каждом районе надо пересаживаться в другой автомобиль.

Поэтому в области ИТ существует объективная тенденция к монополизации сфер информационной деятельности, где не последнюю роль играют языки, как например 1С, почти безраздельно занявший нишу бухгалтерии. Да и требование установки определённых платформ среды исполнения тоже привязывает к разработчику программного обеспечения (ПО). Если задача губернского или государственного уровня, то применяется и адмресурс, устраняющий независимых разработчиков, хотя сплошь и рядом ПО устанавливаемое "сверху" гораздо хуже.

Как отмечалось, последовательность обработки всех данных задаётся в процессе исполнения главной ведущей программы. При появлении данных нового вида необходимо в ведущей программе обеспечить их создание, условия и способ их использования другими программами.

Даже в масштабах предприятия число классов (таблиц или структур, содержащих данные) бывает до тысяч. Вряд ли там найдутся специалисты, знающие, что, где и

как во всех них хранится. Тем более, что на другом предприятии то же самое может быть организовано в других структурах.

В технологиях ООП часто легче добавить новую надстройку над объектами старых классов под новые особенности применения, чем разбираться в алгоритмах методов и структуре данных. И так неоднократно за время эксплуатации ПО. Со временем в базах данных (БД) накапливается «мусор» из забытых уже старых структур, который может быть подхвачен при выполнении запроса.

Конечно технологии развиваются. Разрабатываются системы модификации БД «на лету», протоколы обмена сообщениями, кроссплатформенные магистрали (шины) обмена данными и пр. И все эти средства ПО обновляются по несколько раз за год. В итоге всё больше сил и средств затрачивается только для поддержания систем в работоспособном состоянии, которые становятся всё более монопольными и всё менее обозримыми. По мере прогресса и расширения цифровых технологий все эти проблемы только усугубляются. Вплоть даже до того, что оригинальные исходные коды оказываются погребены под напластованием модернизаций, из-за чего ревизовать и очистить программы от ошибок уже невозможно, ибо слишком дорого. Вот, в частности и об этом, мнение профессионала https://habr.com/ru/post/423889/ Так недалеко и до времени, когда выигрыш от очередного развития ИТ станет меньше, чем затраты на это самое развитие.

2. Неутешительные результаты развития ИТ, которые никого не огорчают

Не огорчают потому, что знание средств и умение преодолевать существующие проблемы в ИТ является специфической сложностью профессиональной деятельности специалистов ИТ и обеспечивает ей высокий статус. К тому же наличие проблем создаёт широкое поле деятельности по разработке средств их преодоления, которые вливаются в объём необходимой компетенции, заставляя специалистов постоянно повышать свой уровень. Сейчас это наиболее востребованный аспект деятельности фирм предлагающих услуги ИТ.

В кодировании алгоритма программы проблем нет, особенно если он отображён блок-схемой, а вот с объектами, даже не столько с их придумыванием и созданием, сколько с получением информации об их структуре и свойствах в конкретной прикладной деятельности, о формах их представления – очень серьёзные проблемы, и не всегда "help" поможет. Подытожим вышеуказанные проблемы ИТ.

1. Трудоёмкость получения содержательной информации о назначении и атрибутике информационных объектов в предметных отраслях.
2. Сложность выполнения обмена данными между прикладными задачами, невозможного без согласований и взаимных действий разработчиков ПО, либо изменений в главной ведущей программе устанавливающей момент и состав обмена.
3. Трудоёмкость и рискованность модернизации прикладных задач вследствие наличия в тексте программ описаний и операций с объектами, которые *по сути специфичны объектам баз данных* и, в принципе, должны бы оперативно редактироваться по мере развития.
4. Необходимость, по мере расширения роли автоматизации и усложнения прикладных задач, постоянного изменения главных управляющих программ и модернизации баз данных, для чего требуется постоянное сопровождение продуктов сертифицированными ИТ специалистами.

Причём адекватная новым требованиям кардинальная модернизация структур данных практически невозможна[1] вследствие нереальности изменения запросов в текстах программ прикладных задач у всех пользователей.

5. Естественная монополизация услуг разработчиками, ибо сторонние и не имеющие сертификата программисты могут не владеть информацией о форматах данных разработчика.
6. Разнообразие и увеличение количества языков программирования, что также способствует монополизации своих услуг разработчиками ПО пользующих специфический язык.
7. Ненаглядность текстового способа представления алгоритма программы. Хотя в BPM-системах в программировании бизнес процессов сделан существенный прогресс при переходе к графическому отображению блок-схем обработки документов.

3. Принципы построения общего информационного пространства

Решить вышеназванные проблемы можно только пересмотром самой парадигмы ИТ. Подобно тому, как в живой природе существует единая технология оперирования генетическим кодом на основе которого формируются популяции животных (как бы "объектов") различных видов, в информационном пространстве должна существовать единая система структурной организации, позволяющая формировать объекты (документы) самого разнообразного вида.

- Должны быть разработаны единые принципы организации и структурирования данных в базах независимо от специфики прикладных задач.

Подавляющее число деловых данных по сути являются поименованными документами или справками содержащими набор текстовых и цифровых данных, в том числе там могут быть указаны сведения о других необходимых документах.

Любой специалист в своей профессиональной деятельности легко оперирует бумажными документами, а при необходимости заводит новые. Он также может сделать выписки из требуемых сопутствующих документов. Однако в электронном формате для создания новых документов пока необходимо прибегать к услугам программистов или администраторов БД.

Естественно, появились предложения от разработчиков предлагающих пользователям средства лёгкого конструирования БД адекватно своим потребностям и понятиям, что в корне противоречит объективной и насущной общественной необходимости интеграции данных.

Унифицированная БД (УБД) позволит оперировать электронными документами с такой же лёгкостью, как и бумажными, не привлекая ИТ-специалистов. Это возможно потому, что при создании новых документов в УБД не возникает новых сущностных объектов и её структура не изменяется, ибо хранимые в ней данные не распределены по экземплярам документов.

Унификация позволит также на основе локальных УБД создать иерархически организованную глобальную распределённую УБД. Причём для обмена данными между локальными БД не требуется знать их адреса. Достаточно, чтобы каждая

[1] А виноватым будет младший программист, который не смог сочинить SQL-запрос, который бы собрал и сгруппировал затребованные отчётные данные по вовсе отсутствующим у них атрибутам в БД.

локальная БД имела связь только с единственной БД выше неё по иерархии на одну ступеньку. А уже через неё запрос на выборку будет растиражирован в локальных БД, либо отправлен для тиражирования в БД более высокой иерархии и т.д.

В такой унифицированной БД можно было бы хранить произвольного формата любые сведения из энциклопедий и даже актуальные научные статьи, если человечество со временем всё-таки создаст единый научный язык, в котором отношения сведений между собой типа "*кто, кого, как, чем, зачем, когда, где, содержит, зависит от, принадлежит к, было, будет, до, после, сейчас, всегда, от ... до ..., если ... то и т.п.*" были единообразно формализованы в языковых конструкциях или символах отношений, допуская в то же время расширение по мере необходимости спектра учитываемых отношений. Эта УБД могла бы стать актуальной базой знаний, причём она оперативно отражала бы всё богатство взаимных связей и *смысловое содержание* хранящихся понятий, фактов и данных.

Автором разработаны архитектура и принципы работы такой БД. Некоторые её упрощённые варианты несколько лет эксплуатировалась в системе делопроизводства городской мэрии.

- Для каждого типа данного (вида документа) в УБД должны быть приведены его онтологические характеристики, т.е. назначение и сопутствующее текстовое описание, отношения с другими данными и *алгоритм получения нового документа этого типа из других документов и данных*.
  Должна быть описана форма представления типа данного в пользовательском интерфейсе и указан нужный инструментарий.

Эти характеристики и средства, называемые *метаданными*, сами тоже являются обыкновенными данными и, следовательно, должны содержаться в самой базе данных, если не в локальной, то в БД более высокого уровня иерархии. Метаданные сами должны иметь краткое *стандартизованное* описание в виде метаданных же, называемое *классификатор*. На каждом уровне иерархии БД классификатор позволяет определить наличие в подведомственных БД затребованных документов. В живой природе аналогом метаданных является, по-видимому, хромосомный набор отвечающий определённому виду организмов.

Нельзя сказать, что этой проблемой не занимаются. Во-первых стандарты XML позволяют характеризовать данные тегами, хоть и в линейных файлах. Есть и более глобальные подходы к проблеме, например, "язык описания онтологий OWL". Но в унифицированной БД принято более кардинальное решение, так что фактические данные "на лету" группируются в форматах тех или иных документов согласно онтологии в метаданных.

Обработка данных, большей частью состоит в том, чтобы в определённый момент выбрать нужные данные из уже существующих документов и создать новый документ, содержащий результаты обработки. Алгоритм обработки, более чем в 99% ситуаций, заключается в суммировании, вычислении среднего, нахождении максимума или минимума, вычислении статистические отклонений, определении регрессивной функции и т.п. по массиву данных в выборке исходных документов.

Момент обработки наступает, когда созданы и получены все исходные данные, нужные для формирования выходного документа с результатами. В состав исходных данных также могут входить срабатывание сигнального датчика,

исполнение оговоренного события (например включение компьютера на рабочем месте специалиста), календарная дата, истечение заданного интервала времени и пр.

Тип требуемых исходных данных, как и способ (алгоритм) их использования, специфицированы в метаданных нового создаваемого документа. В большинстве случаев получение данных нового документа из исходных реализуемо выполнением SQL-запросов. И так, выполняя эстафету последовательного получения очередных новых документов, могут быть в итоге получены все нужные расчётные таблицы, отчёты и справки на рабочих местах каждого специалиста.

Нечто подобное вышеизложенному уже выполняется в BPM-системах управления бизнес-процессами и эффективностью, позволяя перейти от служебных инструкций к автоматическому исполнению и контролю процессов. Но в каждой из таких систем создаются свои объекты в БД со своими специфическими структурами, что, повторюсь, "*в корне противоречит объективной и насущной общественной необходимости интеграции данных*".
И кроме того, эксплуатация и сопровождение, например, "крутейшей" BPM-системы ELMA, которую автору довелось некоторое время администрировать, требует постоянного участия программиста со знанием языка C#.

4. *"Деловой интернет" с унифицированной БД может быть полезен и выгоден*

Использование унифицированной БД (УБД), в отличие от существующей практики, позволит пользователям самостоятельно конструировать типы выходных документов, создавая требуемые SQL-запросы с помощью относительно простого конструктора использующего метаданные исходных документов и их атрибутов. Формируя метаданные нового документа пользователи тем самым опишут и получение данных для его создания. Ввиду унификации структуры БД все SQL-запросы строятся по готовым шаблонам. Технология работ по сути ничем не будет отличаться от работы с бумажными документами создаваемыми специалистами по мере необходимости, причём точно такого же формата, как если бы они были бумажными.

В процессе эксплуатации сформированный документ будет создаваться уже автоматически, по мере появления исходных данных. Таким образом, логика цели и смысла обработки данных приводит к тому, что *готовность исходных данных должна инициировать и получение результатов*, т.е должно выполняться управление по данным (*Dataflow*).

Благодаря метаданным пользователи смогут обмениваться данными интегрируя их простым формированием документа с SQL-запросами данных из массива документов партнёрской организации, а благодаря встроенным средствам связи по системе иерархий БД, это будет возможно даже в случае их нахождения в различных локальных БД. Понятно, что должна быть разработана система контроля доступа к данным.

Вследствие унификации есть основания предполагать, что учётно-финансовую и управленческую деятельность пользователи смогут автоматизировать по мере потребностей самостоятельно. Пользователи, до нужд которых у крупных ИТ фирм "не доходили руки", или те, кому их услуги были "не по карману", смогут легко автоматизировать свои дела.

Кроме того, каждый может создать личный раздел в унифицированной БД на Web-сервере и, предоставляя доступ другим пользователям по их заявкам к некоторым форматам документов, организовать, например, клуб по интересам с возможностью диалогов в чате. Естественно, все участники должны быть клиентами УБД. Кстати, наличие обмена сообщениями считается полезным и в BPM-системах при проведении конференций.

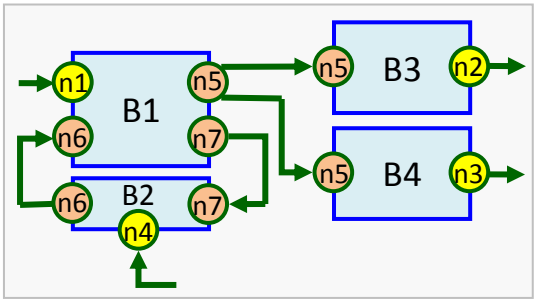
Таким образом в рамках существующих Web-технологий создаётся сегмент интернет-пространства в котором предприниматели смогут вести полноценный учёт ресурсов и продукции, организовать работу с клиентурой и создать систему управления распределённым предприятием без обращения к профессиональным программистам, а все пользователи получают возможность заниматься организацией быта и общением по интересам. Возможно для этого сегмента будет уместным название "Деловой интернет".

Коммерческий интерес разработчика может осуществляться, например, сдачей в аренду пользователям доступа через интернет к унифицированной БД и средствам управления ею, а также к сервисам ПО. А благодаря унифицированному интерфейсу обмена данными и классификаторам метаданных, разрабатываемые в технологии унифицированных БД "свободными" программистами сервисы смогут получить более широкое поле применения, чем это возможно сейчас ввиду узкой специализации и привязки к платформам.

5. Концепция потоковой обработки данных – Dataflow

Итак, в гл.4 сделан вывод, что "*готовность исходных данных должна инициировать и получение результатов*", т.е должно выполняться управление по данным (*Dataflow*). В начале 80-х годов были эксперименты (на Западе) создания Dataflow-машин, в которых поток данных состоял практически из операндов машинных команд. Однако такие архитектуры оказались неэффективными и к тому же использовали дорогую и медленную ассоциативную память для выбора одновременно обрабатываемых операндов. Поэтому перешли к "крупнозернистым" моделям, где каждый модуль обработки данных представлял собой блок операторов.

Излагаемое в данной статье не является описанием мировых трендов в принципах потоковой обработки, а отражает исключительно *авторскую* концепцию. Наглядно "крупнозернистая" модель отображается блок-схемой алгоритма программы. Понятно, что алгоритм, исполняемый в каждом блоке, тоже может быть представлен блок-схемой из модулей, выполняющих части алгоритма блока. И т.д., вплоть до машинных команд.



Каждый блок (или модуль) программы выполняет некоторую группу связанных операций над входными данными получая содержательный результат, который используется на следующих этапах обработки в других блоках.

Исходные данные поступают на *входные порты* (изображены кружочками) блока, а результаты помещаются в *выходные порты*.

Не связанные на приведенном фрагменте блок-схемы, т.е. *свободные*, порты отмечены жёлтым цветом. Данные, передающиеся из порта *n7* блока B1 в порт *n6* того же блока через блок B2, возможно участвуют в некоем цикле или управляемом

контуре обратной связи. Или это может быть исполняемая в блоке B1 функция обработки данных, которая таким образом вынесена вовне и может быть подключаема та или иная по выбору. Её выбор или параметры могут управляться через порт B4. В функциональных языках подобные возможности, когда функция указывается (или программируется) в аргументах, называются функциями высшего порядка.

Если вся программа размещена в одной ЭВМ, то связанные выходной и входной порты могут быть представлены одним и тем же объектом. Назовём исполняемый модулем алгоритм *функцией*, а данное, помещаемое ею в объединённый порт, назовём *возвращаемым значением* функции. В результате получим программу, где функции связаны друг с другом через возвращаемые значения, так что присваивать переменным какие-либо промежуточные значения для временного хранения нет необходимости.

Не берусь безоговорочно утверждать, но полагаю, что вышеописанная модель соединения и работы модулей блок-схемы вполне соответствует концепции функциональных языков программирования. И если бы в *концепции блок-схем* во внутренних программах модулей можно было обойтись без операций присваивания значений переменным, то, наверное, соответствовала бы полностью. Однако в случае блок-схем, ввиду возможности особо наглядного представления алгоритма, вряд ли целесообразно следовать столь жёстким ограничениям.

Поскольку процесс вычисления *управляется передачей данных* от блока к блоку, то совершенно не играет роли порядок, в котором алгоритмы каждого блока будут описаны. Существенным является только описание связей выходных и входных портов блоков – от какого и какому передаются данные – и тоже не важен порядок перечисления.

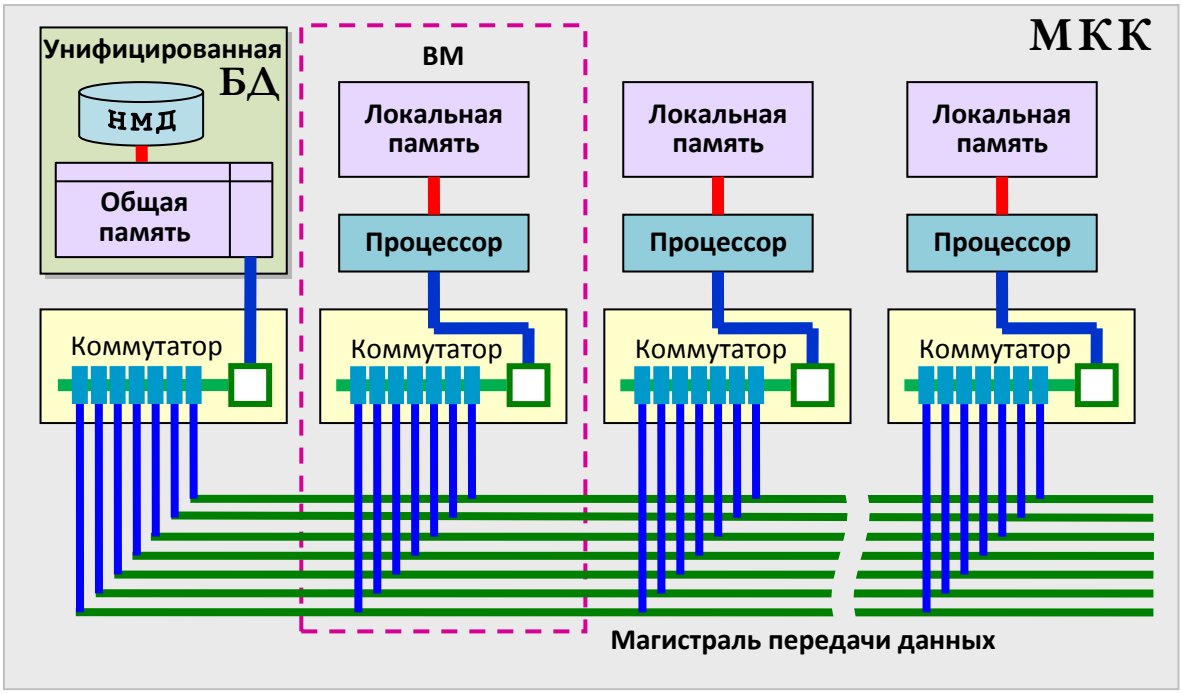
Для того, чтобы получить выходной документ, достаточно, как и в случае "Делового интернета" (см гл.4), просто указать его метаданные в унифицированной БД. В них будет содержаться информация об SQL-запросах исходных данных и программа блок-схемы модулей для формирования требуемого документа. Далее будет вызван *загрузчик*, который в соответствии с программой разместит в вычислительной сети указанные в программе модули. Поскольку структура и типы данных обрабатываемых каждым модулем описаны в метаданных, то сразу будут сформированы входные и выходные порты, установлены их связи и обеспечены средства транспортировки данных между выходными и входными портами.

После получения нового документа и занесения его в БД проводится "уборка мусора" из памяти, т.е. участки памяти зарезервированной для размещения модулей и данных записываются в список свободной памяти, упорядоченный по их адресам. Смежные участки объединяются в один, а список свободной памяти корректируется. Из этого списка по мере надобности снова выбираются участки для размещения данных и программ.

6. Мультикомпьютерный комплекс для Dataflow

Архитектура *мультикомпьютерного* комплекса (МКК) выбиралась по критериям простоты и дешевизны. Блоки программы могут быть размещены в различных вычислительных модулях (на рисунке ниже "ВМ" – обведён пунктиром) комплекса и следовательно работать параллельно. Некоторые из ВМ могут иметь специализированные процессоры. Поскольку данные находятся в основном в

портах, в том числе портах модулей блок-схем внутри блоков, и иногда в очередях, то особо большой локальной памяти процессорам не потребуется. Несколько сотен ячеек для портов и несколько тысяч для команд (только чтение) и собственно самих данных. Некоторые порты могут быть аппаратными устройствами с прерыванием исполнения программы при изменении состояния. Возможно здесь пригодятся последние разработки IBM Research, позволяющие производить вычисления в ячейках памяти. Желателен кэш для размещения переменных и пр.



Основные же массивы данных должны храниться в унифицированных базах данных. Оптимально, если базы данных будут выполнены как специализированные устройства с идентичным интерфейсом управления и доступом к данным – независимо от структуры и связей конкретных данных. И вот только в этих устройствах, для чтения или записи данных, и для хранения выборок из БД, и нужна многогигабайтная оперативная и многотерабайтная постоянная память.

Даже если программа не занимается обработкой данных из БД, она тем не менее является объектом БД, поскольку именно там должны храниться программы её модулей, инструкции и контекстные подсказки. А по поступившему от пользователя требованию загрузчик разместит её в МКК и запустит в работу.

Кстати, при наличии спецустройства с унифицированной БД, в которой (в метаданных) хранятся также и программы модулей обрабатывающих затребованные данные, отпадает необходимость в использовании привычных файловых систем и привычной же необходимости инсталляции программ.

7. Управление исполнением и распараллеливанием программ в МКК

Очевидно, что если данное отсутствует в каком-либо входном порту, или не взято в обработку следующими блоками из какого-либо выходного порта, то программа в данном блоке не может начать или завершить работу и должна ждать изменения ситуации.

Данное, инициирующее обработку, нумерует очередной кадр потока данных. Если на какой-то вход блока поступает данное из кадра с большим номером, чем обрабатываемый сейчас кадр, то оно помещается в зарезервированную область памяти и его местоположение фиксируется в очереди по порядку номеров кадра. В

обработку по ходу алгоритма в блоке забираются данные из очереди только с тем же номером кадра, что и обрабатываемый блоком кадр.

Если в процессе обработки, программе в блоке потребуется обратиться к новому данному, например, обрабатывая документ, понадобиться взять и данные из связанного с ним другого документа, то новое данное связывается с тем же кадром, что и обрабатываемый документ.

Данные между вычислительными модулями передаются по шинам данных. Аппаратные порты используются для обмена данными между ВМ по шинам данных и, возможно иногда, также между программными модулями в одном ВМ. При управлении по данным не важен их источник, так что модуль, устройство или порт, файл или запрос к базе данных, как источники или адресаты данных взаимозаменяемы. Поэтому любая из шин данных может быть "продолжена" любым подходящим устройством связи к другим МКК и иным устройствам.

Перед началом пересылки данных шину следует "захватить", чтобы не было взаимных помех. Обычно захват выполняется методом "взвешивания" адресов устройств, когда право передачи получает тот из претендентов, чей адрес на шине наименьший, но этот процесс занимает как минимум столько тактов, какова разрядность адресов.

Мне довелось заведовать лабораторией системного матобеспечения в филиале Московского специализированного НИИ АН СССР, в котором, до самой "перестройки" приведшей к его ликвидации, разрабатывались алгоритмы параллельных вычислений в децентрализованных мультипроцессорных управляющих системах и сопутствующее "железо". И вот тогда там были разработаны принципы захвата шины за один такт независимо от числа устройств на ней, тем более это можно сделать при современных технологиях. В таком случае, в МКК целесообразно использовать "магистраль" состоящую из множества шин данных и запрос на захват посылать одновременно на все пока как бы свободные в данный момент – какую-нибудь и удастся захватить.

При управлении по данным, работа модулей управляется "*протоколом движения данных*" (ПДД), а не операционной системой, которая совсем не нужна, даже для управления ресурсами. Все её функции могут исполняются запуском в нужный момент или по ситуации сервисных модулей и посылкой им пакета данных с соответствующими требованиями.

Сервис ПДД запускает программу модуля исполнение, если в его входном порту появилось данное. Результат исполнения программы помещается в его выходной порт и одновременно инициируется работа сервиса ПДД, который по шине данных выставляет "заявку" на передачу в связанный с ним (например, с таким же идентификационным номером) входной порт очередного модуля в другом ВМ. Появление "заявки" инициирует в работу уже местный сервис ПДД, который принимает и размещает в памяти пересылаемый массив данных, а его адрес заносит во входной порт модуля, либо в очередь, если порт занят. Затем в этом ВМ возобновляется исполнение прерванной программы.

Если данные из выходного порта должны передаваться нескольким входным, то в нём должен быть счётчик семафор, который будет сигнализировать, что данный выходной порт занят, пока данные из него не будут переданы всем потребителям. Пример такого порта с именем "*n5*" на рисунке в гл.5 приведён в гл.9.

В *мультипроцессорном* комплексе с общей памятью пересылки массива данных не потребовалось бы, но в конечном итоге к каждому его данному было бы выполнено обращение и, быть может, неоднократно. К тому же в схеме с общей памятью одна шина данных, а в МКК их может быть много.

Если оба модуля, отсылающий и принимающий, находятся в одном ВМ, то связанные выходной и входной порты представлены одним объектом, причём передача управления следующему модулю организовывается ещё при их загрузке. Работы сервиса ПДД здесь не требуется.

В принципе, сервис ПДД можно дооснастить сервисом распараллеливания, подгружающим в свободные или малонагруженные ВМ копии программных блоков, во входных портах которых создаются очереди. При оптимальном распараллеливании блоков по ВМ должен получиться конвейер вычислений, когда темп получения кадров результатов на выходе будет успевать за темпом поступления кадров на вход конвейера и очередей не образуется.

Здесь описаны не все технологии распределённых вычислений разработанные в вышеупомянутом НИИ и являющиеся тогда секретными, а только самые "очевидные". Также не описана структура унифицированной БД.

Пусть уж взявшиеся за реформацию, которая только и способна вывести ИТ из трясины устоявшихся стереотипов, сами изобретут заново эти технологии и полностью получат все полагающиеся в этом случае "призы". Обнародование всех "Ноу-хау":

во-первых, может дать возможность легко дискредитировать всю идею придравшись, с неких как бы авторитетных позиций, к некоторым недостаткам предлагаемой конкретной реализации, которые несомненно всегда можно отыскать;

во-вторых, лишить экономических преимуществ и заслуженного профита организацию или фирму, решившихся на разработку "Делового интернета" или мультикомпьютерных комплексов. Опасение такого исхода может снизить их заинтересованность в деле, что не на пользу прогрессу.

8. Что делать и с чего начать

1. Разработать единую структуру унифицированной базы данных (УБД), пригодную для хранения любых данных в их взаимосвязи, включая описания метаданных.
2. Разработать модель унифицированной БД в рамках существующих Web-технологий и реляционных баз данных.
3. Разработать систему иерархической организации БД и технологию транзакций для обеспечения обмена данными и их интеграции.
4. Начать пересохранение своих данных в формат унифицированной БД. Сначала это будет делаться в индивидуальном порядке, что откроет каждому перспективы дальнейшего прогресса. Затем появятся и отработанные стандартные технологии перехода. Госструктуры, можно предположить, останутся в аутсайдерах, зато подрядные ИТ фирмы, не исключено, станут использовать унифицированные БД.
5. Разработать *эффективный* алгоритм протокола движения данных (ПДД).
6. Разработать язык программирования параллельных вычислений управляемых потоками данных и компилятор.
7. Разработать технологии создания и хранения библиотечных программных модулей в хранилищах данных, компиляции и загрузки в вычислительные блоки.

8. Разработать и начать производить электронные устройства хранилищ баз данных с унифицированным интерфейсом согласно п.3.
9. Начать производить мультикопьютерные системы с мультишинными магистралями данных и электронными портами обмена данными, с хранилищем БД заменяющим также файловую систему

Пункты 1, 2, 3 и 4 выполнимы в стандартных Web-технологиях и на данный момент кажутся наиболее эффективным вложением средств в ИТ разработках. Об этом в главе 4 "Деловой интернет". Реализация унифицированной БД возможна в любой СУБД реляционных баз данных, хотя технически это наверное не оптимальное решение.

Ожидается, что успешность применения УБД в задачах пользователей будет стимулом к выполнению п.8 по разработке аппаратных хранилищ УБД, но к этому моменту уже должен быть выполнен п.7, которому должно предшествовать выполнение пп.5 и 6.

На "Западе" разработки языков функционального программирования пригодных для "крупнозернистой" модели *Dataflow* были давно сделаны. Но *широкого* развития направления *Dataflow* не отмечено. А какой бы смысл, если существующий статус-кво пока всех устраивает?

Можно, и достаточно легко, занять лидирующую позицию в ИТ, если опереться на относительно недорогое "железо" и язык программирования достаточно высокого уровня, чтобы без особых проблем *воспроизвести* все современные наработки (ОС более не нужна) и навсегда избавиться от информационной зависимости.

Избавление от ОС и перепрограммирование используемого ПО, необходимость чего давно перезрела, при возможности наглядного представления алгоритмов в виде блок-схем, несомненно улучшит его надёжность, качество и быстродействие.

9. *О языке программирования параллельных вычислений управляемых потоком данных*

Поскольку в мультикомпьютерном комплексе передача данных между вычислительными модулями имеет существенное значение, то целесообразно иметь язык программирования, на котором её удобно описать. Попробую вообразить, как мог бы выглядеть подобный язык программирования, обозначенный далее как DFCP **(**data flow control programming**)**.

Введём некоторые определения и соглашения.

Угловые скобки в примере и в описании языка ниже заключают *описание* того, что в реальном тексте программы должно бы быть на их месте.

Комментарий – это строка любых символов (кроме самих апострофов) между апострофами. Это **'комментарий'**.

Разделители – это, символы "**,**" "**;**" "**.**" и *символ пробела*. В тексте программы *комментарий*, *множество пробелов* подряд, или *знак разделителя* эквивалентны одному *символу пробела*. Это позволяет акцентировать разделение ими данных и их групп, отличающихся по смыслу, принадлежности или просто для наглядности.

Символ *конца строки* в тексте программы игнорируется, что позволяет писать длинные выражения в нескольких строках программы. Знак разделителя "**;**" в конце строки предотвращает *случайное* слияние текстов соседних строк.

Пример: Для начала покажу, как *могла бы быть* представлена программа для блок-схемы, приведенной на рисунке в гл.5.

code **(** *module* **(**ThisBlock**;** 'свободные порты:' n1,n2 ; n3,n4**)** *begin*

'1.Назначение свободным портам n1,n2,n3,n4 данного блока фактических портов внешнего блока осуществляется через объявление формальных параметров. Структуры этих портов описанные внутри и вне блока должны совпадать.'

include (*<подключаемые библиотеки>, ...*) 'Выполняется выборка из базы данных'
'Поиск в БД и выполнение загрузки библиотечных модулей:'
load (*lib_module_1* 'далее могло быть перечисление других библ. модулей');

'2.Компиляция и загрузка программ модулей:'
code **(***module***(**'имя:' *NewMod2***;** 'формальные параметры:' *p1*, *p2* ; *p3*, *p4***),** *begin*
<программа тела модуля … >end **)**;
code **(***module***(***NewMod3***;** *p1*, *p2*; *p3***),** *begin< программа тела модуля … >end* **)**;

'3.Имена модулей B1,B2,B3,B4 связываются с их содержанием:'
block **((***lib_module_1*; 'псевдонимы:' *B3*,*B4***)**, **(***NewMod2*; *B1***)**, **(***NewMod3*; *B2***))**;

'4.Описываются структуры портов с целым числом и числом с пл.точкой:'
struct **((** *port_i***;** *b*, *flag***;** *i*, *data***;** *l*, *next***)**, **(***port_f***;** *b*, *flag***;** *f*, *data***;** *l*, *next***))**;
'Структура порта с указателем адреса и несколькими потребителями данных:'
struct **(** *port_n***;** *b*, *flag***;** *l*, *pnt***;** *l*, *next***;** *b*, *mmb***;** *b*, *cnt***)**;

'5.Описываются порты по их принадлежности к типам'
dec **((***port_i*; *n4*, *n6*, *n7***) (***port_f*; *n1*, *n2*, *n3***) (***port_n*, *n5***))**;

'6.Указываются связи портов с блоками модулей'
#({*n6.next*}'←'B1); #({*n7.next*}'←'B2); #({*n5.next*}'←'B3);

'7.'dec (*f*, [*MM*, 19]) '← здесь объявлен массив MM на 20 чисел с пл.точкой'
'Устанавливается связь с данными и число получателей данных порта n5:'
#({*n5.pnt*}'←'@MM); #({*n5.mmb*}'←'*b*<*i*(2));

'8.Порты связываются с модулями через их формальные параметры:'
assign **((**B1**;** *n1*, *n6* **;** *n5*, *n7***), (**B2**;** *n7*, *n4* **;** *n6***), (**B3; *n5* **;** *n2***), (**B4**;** *n5* **;** *n3***))**;

end **)**.

Вот, примерно, и всё. О некоторых выражениях в программе из этого примера расскажу далее. Правда, в примере не показаны операторы занесения данных в порт и передачи управления следующему модулю, которые должны использоваться в модулях, но в данном языке не будет ничего такого, чего не было бы в языках Си либо Лисп. Разве только он имеет гораздо более ограниченный синтаксис. В частности, в нём допустимы выражения только вида <*Что делать*>**(**<*перечень над чем*>**)**, то есть состоящие из *оператора* и *операндов:*

<инструкция>(<операнд>, …, <операнд>). Так что профи могут сразу переходить к следующей главе.

Оператор – это группа символов с *инструкцией* компилятору, или указанием на *действие* выполняемое программой, а *операнд* – это символьное имя (идентификатор) *данного* или константа представляемые адресом[2] в памяти ячеек содержащих их значения.
Круглые скобки определяют иерархию вложенности операторов и порядок обработки операндов.

Оператор *присвоения значения* данному использовался в примере выше. Так, операция #(*a*, *b*) означает, что в ячейки по адресу данного "*a*" будет занесено количество байт соответствующее типу "*a*" из ячеек по адресу данного "*b*". Если типы данных не совпадают, например, *a-float,* а *b-integer*, тогда надо использовать функцию "***f<i***" для преобразования типов: #(*a*, ***f<i***(*b*)).

Опишу, для примера, ещё несколько операторов.

Сумму целочисленных данных *k*, *l* и *m* получим операцией ***+i***(*k*,*l*,*m*), а сумму чисел с пл.точкой *a*, *b* и *c* операцией ***+f***(*a*,*b*,*c*), так как для данных разного типа и операторы нужны разные.

Оператор массива пусть выражается функцией ***arr(***<*имя массива*>, <*индекс*>***)***. Индекс представлен целым положительным числом начиная с нуля. Но в таком виде операции с массивами было бы трудно находить среди прочего текста программы. Несколько более привычным и заметным был бы вариант написания в тексте [<*имя массива*>, <*индекс*>].

Но если предусмотрим оператор замены текста "*def* (<*подстрока*>,<*подмена*>)" в компиляторе программы, то инструкции *def* **(** "[", "arr(" **)** и *def* **(** "]", ")" **)** обеспечат желаемое.

Имени массива сопоставляется адрес в памяти его *первого* элемента. Например, операция #(LL, <*имя массива*>) содержимое именно этого элемента и загрузит в ячейку LL. Функция [<*имя массива*>, <*индекс*>] выполняет операцию
<указатель> = <адрес первого элемента> + <значение индекса>*<размер элемента>.
И уже полученный <указатель> подставляется вместо адреса первого элемента.

Указателю адреса (pointer) значение присваивается выполнением функции <*адрес*><*данное*>. *Значение данного* можно получить функцией <*значение*><*указатель*>. Используем, например, в качестве имени функции <*адрес*> символ "@", а функции <*значение*> символ "~"

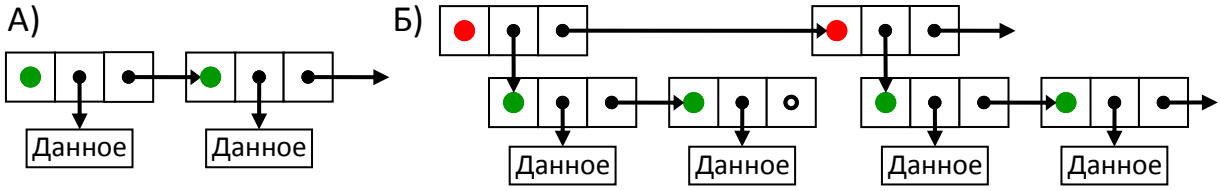
Операция #(Pnt_NN, **@**(NN)) заносит адрес данного NN в ячейку с именем данного Pnt_NN. Операция ~(Pnt_NN) выдаст как операнд содержимое ячейки обозначенной Pnt_NN. Значит значением выражения ~(Pnt_NN) будет данное NN.

Например, присвоить данному LL значение 3-го элемента из массива целых чисел M можно операцией #(LL [M,2]) или операцией #**(**LL,**~(+*l*(@(**M**), ******i***(2,4)))**)**, поскольку 1-й элемент имеет индекс =0, а каждое целое число занимает 4 байта в памяти. Оператор ******i*(…) перемножает целые числа, а оператор **+***l*(…) прибавляет результат к адресу (типа длинное целое – *long*) первого элемента массива.

Описание *структур* выполняется *оператором*:
struct **(**<*назв. структуры*>, <*тип*>, <*название поля*>, … <*тип*>, <*название поля*>**)**.

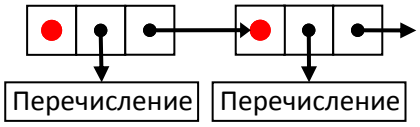
[2] В машинных командах операнд может быть не только адресом, но и числом

Например, в описании элемента узла *списка*: ***struct*** **(*node*, *b flag*, *l Pnt_data*, *l next*)**, поле "*flag*" содержит сведения о данных в списке и их состоянии, поле "*Pnt_data*" указывает адрес данного, а поле "*nex*t" – адрес следующего узла списка.
На рисунке ниже, на схеме **А** показан список именно такого типа. На схеме **Б** показан возможный вариант списка списков.



Операция со структурами ***rec*** **(**<*назв. структуры*>, <*название поля*>**)** возвращает, как операнд, адрес ячеек с указанным данным. Как и в случае массивов, нас бы более устроило написание **{**<*назв. структуры*>**.**<*название поля*>**}**. Для этого надо сделать подмену текста при компиляции программы инструкциями *def* **(** "{", "rec(" **)** и *def* **(** "}", ")" **)**, а точка и так является разделителем.

Объявление данных выполняется *оператором*:
dec (<*тип данного*>, <*имя данного*>,**…**, <*имя данного*>).
Например, в строке объявлений *dec* **((*i*, *a*, *b*, *c*) (*f* , *w*, [*M* 19], *r*, *t*) (*node*, *list*))** объявлены целочисленные данные (*a*, *b*, *c*), данные (*w*, [*M* 19], *r*, *t*) с пл.точкой, включая массив M на 20 чисел, а также начат пустой пока список с именем *list*. На схеме слева показан формат списка списков соответственный объявлению данных.



Вероятно целесообразно установить следующие правила обработки списков аргументов и формирования выходных результатов, которые, полагаю, проще показать, чем объяснить. Например, применяя унарные операторы (функции f) к списку $f(a, b, c)$ получим список значений ($f(a)$, $f(b)$, ($f(c)$), а для бинарных, т.е. вида $f(a, b)$, операторов получим:

$f(a, b, c, d)$	=> $f(f(f(a,b), c), d)$	=> значение
$f(a, (b, c, d))$	=> ($f(a,b)$, $f(a,c)$, $f(a,d)$)	=> список значений
$f(a, (b, (c, d)))$	=> ($f(a, f(b,c))$, $f(a, f(b,d))$)	=> список значений
$f(a, b, (c, d), e)$	=> ($f(f(a,b), c)$, $f(f(a,b), d)$, $f(f(a,b), e)$)	=> список значений
$f((a, b), (c, d))$	=> ($f(a,b)$, $f(c,d)$)	=> список значений

10.Технологичность VS красоты и изящества

Я считаю, что инструменты работы с ЭВМ должны быть адекватны применяемым в ЭВМ технологиям: <инструкция> (<операнд>, …, <операнд>).
Сравним запись выражения $D = \sqrt{b^2 - 4ac}$ на Алголе: D:=sqtr(b↑2–4×a×c)
с записью на предполагаемом языке DFCP: #(*D*, sqrt(+(^(*b* 2),–(*(*4 a c*))))).

В первом случае требуется синтаксический анализ выражения, а во втором программу его вычисления можно в процессе компиляции сразу загружать в память ЭВМ прямо "с листа". Конечно, в последнем выражении многовато скобок. Поэтому в Лиспе скобки и запятые не ставятся, если можно понять, где оператор, а где операнды – тогда здесь, например, стало бы: *# D* sqrt(+ ^ *b* 2 –(* *4 a c*)). Но похоже, теперь придётся задумываться, что к чему относится. Особенно если имена функций

(операторов) тоже буквенные, как и у операндов. А компилятор тогда должен знать сколько операндов может быть у конкретного оператора (функции).

Как бы облегчая труд программирования, изобрели множество языков, которые мало соответствовали машинным технологиям, но были как бы близки к разговорным человеческим. Изобретатели языков всё более отдаляют в область абстракций описание алгоритма от способа его машинной реализации.

Цитата: "*Лямбда-исчисление, разработанное А. Черчем*[3] *для преодоления некоторых проблем в математической логике, фактически служит теоретическим базисом функционального программирования*" /выделено А.С./ (http://www.recyclebin.ru/BMK/FP/Lambda_FP.pdf). Не оспариваю гениальность и изящество воплощения, но по мере перехода ко всё более сложным аспектам применения, простой и удобный поначалу язык превращается в трудно понимаемый набор вложенных одно в другое заклинаний.

Вероятно это приносит эстетическое удовольствие некоторым программистам, но скорее всего потребует больше памяти в ЭВМ, а многие программисты предпочли бы употребить простой рекуррентный цикл вместо рекурсий и вспомогательных функций, нужных чтобы всего лишь избежать применения оператора присваивания. Могу предположить, что невозможность присваивания значения переменным в функциональных языках может быть чревата необходимостью повторного вычисления функций.

Язык DFCP предназначен для программирования модулей обменивающихся потоками данных через порты ввода/вывода. Для портов справедлива концепция "*однократного присваивания*", т.е. в каждом кадре потока (см гл.7) данные один раз заносятся в порт и затем изымаются из него. Но если аналогично поступать и с просто переменными, т.е. не использовать одно и то же переменное для разных по смыслу и назначению данных, то они фактически не будут отличаться от портов. Таким образом рекомендации не использовать переменные в потоковых языках весьма условны.

Кстати, упомянутое упорядочение применения переменных позволит любую императивную программу изобразить в виде блок-схемы операторов и операндов, так что синтаксис программ может быть совершенно одинаков и для машинных команд и для блоков модулей. Но для блоков программа станет декларативной.

Наверное можно попенять, что достаточного описания языка так и не представлено. Но на самом деле всё *уже сказано* тем, что вся семантика языка исчерпывается конструкциями <оператор> (<операнд>, …, <операнд>). А согласно синтаксису, операнд в свою очередь может быть выражением с операндами включая операторы функций. Иерархия вычислений и результатов определяется вложением скобок.

Язык применим от самого низкого уровня, когда его операторы копируют машинные команды или инструкции ассемблера, и до неограниченно высокого, когда операндами являются объекты сколь угодно сложной структуры, нисколько не уступая в этом языкам ООП.

Более того, не так сложно запрограммировать модуль, чтобы во время исполнения программы в некоторые его формальные параметры подставлялись бы

[3] Алонсо Чёрч

адреса тех процедур, которые нужны согласно алгоритму и должны исполняться в теле модуля.

При выставлении данного в порт, если обрабатывающие его модули находятся в одном вычислительном модуле (см гл.6), то вместе с данным требуется передать и управление следующему обрабатывающему его модулю. Поэтому в программной части языка DFCP предполагается использование команд перехода "Goto" (передачи управления). А в таком случае в программе естественно использовать и переход на метки, что как бы противоречит принципам структурного программирования.

У меня есть опыт программирования начиная с кодировки команд на перфокартах в двоичном машинном коде и абсолютных адресах. До НИИ я работал в ОКБ, где проектировались двигатели для "лунной" ракеты Королёва, и там для обработки телеметрии испытаний авиационных и ракетных двигателей были установлены первые полупроводниковые аппаратно-программные комплексы изготовленные пензенским заводом САМ (з-д счётно-аналитических машин). Но матобеспечения не было никакого. Все драйверы для устройств ввода/вывода и регистраторов телеметрии, все библиотечные программы пришлось разрабатывать с нуля, включая переводы чисел из двоичного в 10-й формат и обратно.

Поэтому пришлось сначала разработать простенький язык типа ассемблера, затем отладчик, который (в целях экономии бумаги) во всех циклах распечатывал только два первых пропуская остальные и диагностируя бесконечные, и пр. Потом, и для перфоленточной ЭВМ, хотя уже и следующего поколения, пришлось разработать и совместимую магнитноленточно-дисковую ОС (на случай, если диски откажут в критический момент). Ну это при том, что приходилось заниматься и собственно обработкой данных, как то спектральным анализом телеметрии и даже заново изобретать для этого алгоритм быстрого преобразования Фурье, и пр.

Исходя из собственного опыта программирования, могу утверждать, что программа с метками и командами перехода обычно компактнее и более понятна, чем с продублированными копиями блоков и набором «флажков» ради исключения команд перехода. А уж если кто-то сознательно захочет сделать код непонятным, то в языках ООП возможностей для этого гораздо больше. Аналогичного мнения придерживаются и некоторые профи: https://habr.com/ru/post/114211/

Поскольку операндом может быть любой цифровой объект или параметры состояния и ресурсов вычислительной системы, а оператором любой выполнимый в ней алгоритм, то ясно, что *возможности и универсальность* применения данного языка почти не ограничены. При том, что его синтаксис и структура остаются неизменными и не появляется никаких новых синтаксических определений и правил. Любые "расширения" языка будут иметь всё тот же формат операторов включая, быть может, некие сигнатуры для операндов. В примере гл.9 это служебные слова "*begin*" и "*end*". И совершенно неважно, соблюдались ли какие-либо правила при программировании кода этих операторов.

Представление в виде блок-схем соответствует концепции программирования «сверху вниз», а взаимодействие модулей только через порты гарантирует соблюдение принципов инкапсуляции. Кроме того, естественный интерфейс по границам отсылки или приёма данных оптимально подходит для коллективной разработки программ.

11.Как это будет работать

Важнейшую роль в программировании играет доступность сведений обо всём арсенале средств системных платформ и их использования в аспектах прикладных применений. Сейчас это достигается изучением немалого объёма знаний и приёмов в каждой отрасли деятельности.

Использование унифицированной БД предоставило бы, *благодаря классификаторам*, *метаданным и содержащихся в них описаниям*, возможность компетентного выбора модулей адекватных возникшей потребности.

Наличие технологической поддержки в виде разработанных *CASE*-средств позволило бы визуально сконструировать блок-схему программы для решения задачи и её технологическую карту в виде выборок элементов программы из баз данных, включая средства отображения данных и работы с ними.

Вновь созданная программа пополнила бы базу данных средств программирования. А при запуске программы в работу, как и при использовании любой вообще программы, пользователи временно получают специфические средства отображения её данных и управления процессом исполнения. – так что никаких предварительных инсталляций ПО на все случаи жизни не потребуется.

Реализация всех вычислительных процессов в системе в виде блоков связанных передачей данных через порты, обеспечила бы возможность компиляции и включения новой задачи в состав работающей системы обработки данных без потерь данных и заметной остановки – и даже без перепрограммирования действующей части системы. Естественно при наличии у разработчика должных прав. Внесённые изменения в свою очередь должны быть отражены в описаниях (метаданных) действующей обработки.
Кстати, распараллеливание потоков тоже было бы выполнимо "на ходу", без перезагрузки.