Схемная эмуляция: новая пропорция старого дуализма, или – паяем без паяльника

Сергей Ковалев (г. Киев, Украина)

Исходными данными предлагаемой автором системы компьютерного проектирования служит графический образ функциональной схемы прикладного алгоритма, выходными – его функционирование с соблюдением всей динамики происходящих процессов. Такой вещи, как компиляция, в этой системе просто нет места.

Предисловие

Цель статьи – показать, что с помощью «обычной» программы схемной эмуляции можно решить множество проблем программирования. «Обычность» такой программы для меня, как для автора, состоит в том, что существует она с 1985 года, необычность – в том, что до сих пор не существует других программ, которые можно было бы отнести к классу программ схемной эмуляции.

Известны такие программы-симуляторы, как Micro-CAP от Spectrum SoftWare, PSpice от MicroSim Corp. и др., которые в советскую эпоху у нас принято было называть программами моделирования электронных устройств. Они работают в условной временной шкале, а не в режиме реального времени, и не предназначены для взаимодействия с внешними реальными электронными устройствами, да к тому же требуют ресурсов PC.

Что касается проблем, присутствующих в индустрии программирования, то можно с уверенностью сказать, что тема эта – бесконечна. Ей посвящено несчётное число статей в IT-литературе и во Всемирной паутине.

Именно растущее число проблем со стороны современного инструментария программирования и самих программистов способствовали росту популярности альтернативных путей написания программ – в частности, развитию идей программирования, основанных на теории автоматов.

Статья ставит своей целью показать:

- почему эмуляция(!), да ещё и схемная(!), чем она отличается от обычных схемных симуляторов и чем лучше привычной компиляции или интерпретации;
- что применение схемной эмуляции для эмуляции графических проектов является прекрасным дополнением к уже развиваемой идее «автоматного» программирования, в частности, даёт возможность использования её как визуализатора и позволяет обойтись без заключительного этапа – программирования;
- что авторская программа схемной эмуляции по своей сути есть Система Графического Программирования;
- что слово «схемный» несёт в себе гораздо большую смысловую нагрузку, чем понятие принципиальной схемы электронного устройства. Теперь это – графический рисунок проекта: схемы функциональной, схемы алгоритма, графа переходов, графа состояний и другого, что может ещё быть придумано человеком и нарисовано им на листе бумаги или дисплее;
- что эмуляция – это имитация аппаратной реализации алгоритма.

Возможность реализации алгоритмов аппаратным путем или программным способом на ПК была названа Э.Э. Клингманом дуализмом.

Предлагаемая к рассмотрению технология эмуляции позволяет вообще отказаться от процедуры программирования для ряда задач: автоматизации технологических процессов (АСУ ТП), экспертных систем, нейронных сетей, баз знаний и др.

Достаточно нарисовать в графическом редакторе графический рисунок проекта – и эмулируй его себе на здоровье, имея, кстати, «под рукой» готовые временные диаграммы для целей отладки или поиска неисправности в периферийном оборудовании (например, при реализации задач АСУ ТП).

Э. Дейкстра писал: «Я знал, что программы могут очаровывать глубиной своего логического изящества, но мне постоянно приходилось убеждаться, что большинство из них появляются в виде, рассчитанном на механическое выполнение, и что они совершенно непригодны для человеческого восприятия...» [Дейкстра Э. Дисциплина программирования. М: Мир, 1979.]

Поэтому всё, что написано ниже, – стоит прочитать.

Думаю, что пример внедрения идеи в область АСУ ТП поможет полнее понять её суть.

Сразу уточню: алгоритм эмуляции реализован в программе, которая по разным причинам пока не получила коммерческого развития, поэтому использовался мною только в личных внедрениях. Видимо, отечественным внедренческим фирмам пока легче жить по принципу: там купил – здесь быстро перепродал. Не важно, какого качества, зато навар в кармане сразу.

Кризис в программировании

Профи от программирования вообще-то не любят говорить о каком-то кризисе в своей отрасли, предпочитая ограничиваться обменом мнениями на форумах различных сайтов на темы вроде: о нестрогой проверке типов, частом использовании указателей и возможных последствий этого, доступности компиляторов для различных платформ...

По их глубокому убеждению, всё это – временные проблемы, которые, конечно же, будут устранены, к примеру, с появлением новой версии соответствующего компилятора. Са-

мые «продвинутые» позволяют себе поднимать вопросы более глобального масштаба, высказываясь о недостатке самой парадигмы объектно-ориентированного программирования (ООП), предлагая переход к парадигме процессов. Другие критикуют процессы, называя всё это лишь подменой терминов. Кто-то верит в появление чуда – какой-нибудь новой CASE-системы, которая поднимет качество программирования и сроки разработки программного обеспечения на доселе невиданный уровень, позволяя при этом создавать сложнейшее программное обеспечение человеку, просто владеющему мышкой.

Мы пережили уже несколько революций в программировании, и каждый раз это преподносилось чуть ли не как панацея. Однако, как это было до сих пор, каждый новый инструментарий приносил и новые недостатки, и новые дыры. Я выделю основные проблемы, которыми страдает программирование как отрасль.

Первая – это лавинообразный рост программного кода в программных продуктах, предлагаемых пользователю, связанный с ростом функциональной сложности как прикладных программ, так и самого инструментария. Всё больше усилий требуется от разработчика на написание и отладку кода, всё меньше сил и времени остаётся на проработку самой задачи, поиск ошибок и оптимизацию.

Известна история про то, как при испытаниях некоей полётной системы на море периодически происходил её сброс. После долгих разбирательств выяснилось, что при переходе уровня моря (уровня Мирового океана) происходила ошибка «деление на ноль». В другом случае перестала работать некая географическая система, когда её решили использовать на другой территории. Как выяснилось, ошибка была связана с переходом через широту 90°.

На заре компьютерной техники программы решали сравнительно несложные задачи, поэтому и были сравнительно невелики. Естественно, и в таких объёмах появлялись ошибки, но для их поиска не надо было перелопачивать мегатонны кода.

Согласно данным отчёта Национального института по стандартам и технологии, объём экономических потерь из-за ошибок в ПО в США достигает нескольких миллиардов долларов в год, что составляет около 1% национального валового внутреннего продукта (Research Triangle Institute, NIST Planning Report 02-3, May 2002).

Вторая – любая программа способна реагировать только на те события и их комбинацию, которые предусмотрел программист.

Вне зависимости от методов разработки у любой программы есть состояния, в каждый момент времени определяемые значениями всех её переменных. Тогда изменение значения одной из управляющих переменных будет означать изменение состояния программы, а число состояний программы будет определяться максимально возможным количеством комбинаций значений управляющих переменных, возникающим при её работе. Если предположить, что в программе используются только двоичные управляющие переменные (флаги), то в этом случае количество состояний будет стремиться к 2^n. Поэтому нет никакой гарантии, что в процессе работы программы не возникнет неожиданное сочетание входных воздействий, вследствие чего программа перейдёт в непредусмотренное состояние. Такие состояния Фредерик Брукс назвал «невизуализированными». «Сложность служит причиной перечисления, а тем более понимания всех возможных состояний программы, а отсюда возникает её ненадёжность...» [Брукс Ф. Мифический человеко-месяц, или как создаются программные системы. СПб.: Символ, 2000.]

В частности, изучение современных компьютерных игр показывает, что при высоком качестве графического и звукового оформления досадные «проколы» чаще всего встречаются там, где на помощь могло бы прийти явное выделение на этапе проектирования всех требуемых состояний программы.

Работа же с графическим отображением проекта, например АСУ ТП, одинаково понятным заказчику и технологу, позволяет достичь необходимой строгости за счёт использования единой и наглядной спецификации.

Третья – отсутствие единой спецификации проекта в цепочке «заказчик–технолог–программист». Результатом совместной работы заказчика с исполнителем (пусть это будет технолог) является некий алгоритм, представленный в виде графического рисунка или словесного описания – что выступает основой документа, называемого техническим заданием, скрепляемого печатями организаций.

Программисту необходимо проделать неформальную операцию перевода этого документа в язык машинных кодов – программного продукта, предоставляемого заказчику.

В большинстве случаев переход от алгоритмизации к программированию представляет определённую проблему. Это объясняется тем, что обычно процесс алгоритмизации почти никогда не завершается тем, чем должно завершаться создание алгоритма в математическом смысле. По определению он должен однозначно выполняться любым вычислителем, а оканчивается лишь некоторой «картинкой», называемой алгоритмом. Именно эту картинку приходится додумывать программисту, чтобы представить ее вычислителю на понятном ему языке – языке программных операторов.

Не существует пока формальных методов такого «перевода», каждый программист проделывает эту работу по-своему, в зависимости от интеллекта и знания языка программирования. Почему до сих пор и принято считать программирование искусством.

Четвертая – участие программиста в совместных проектах. В течение долгого времени программирование обращается за примерами организации производства к другим инженерным дисциплинам и, прежде всего, к проектированию электроники. Тем не менее, программирование, несмотря на свою массовость, остаётся искусством. На сегодняшний день все попытки полностью формализовать или автоматизировать процесс написания программ оказались несостоятельными. Поэтому затраты времени и средств, выделяемых на проект, находятся в прямой зависимости от способностей и амбиций программиста. Зачастую он остаётся единственным человеком, способным разобраться в своем творении, от которого в конечном счёте зависит судьба всего проекта.

Пятая – уровень развития современных инструментальных средств программирования. С сегодняшни-

ми, в общем-то, не такими уж и трудными задачами (управление базами данных, интернет-технологиями и проч.) программирование, как отрасль знаний, ещё справляется. Но вызывает абсолютное сомнение факт, что «интеллектуальный уровень» ныне существующего инструментария позволит решить задачи уже недалекого будущего. А именно, создание экспертных систем нового поколения, баз знаний, искусственного интеллекта и т.п.

Всё вышеперечисленное невольно приводит к мысли о неминуемом приближении кризиса программирования как отрасли.

Возрождение автоматов

Заказчики и разработчики программного обеспечения всегда мечтали о том, чтобы программы работали так же надёжно, как электронные устройства. Отсюда и постоянное стремление заимствовать организацию разработок и проектирования, устоявшуюся в индустрии электроники, а также внедрение формальных методов в технологию программирования.

Поэтому вполне естественным выглядит стремление положить в основу технологии программирования алгоритм работы разрабатываемой системы. Задача эта актуальна и потому, что её решение позволит представителям различных профессиональных групп – Заказчик, Технолог, Программист – однозначно понимать друг друга.

Но что-то здесь не клеится. Тексты программ по-прежнему мало похожи на алгоритмы. В конечном счёте появляются два алгоритма: один на бумаге, для отчётности и документирования проектных решений, а второй – в листинге программы. По существу получается, что программа отображает то, как программист смог понять исходный алгоритм. В итоге он оказывается единственным держателем важнейшей логической информации. Но дело ещё и в том, что разные программисты пишут программы по-своему, в зависимости от интеллекта и знания языка программирования. Нечего уже и говорить про случай, когда человек может просто уволиться в разгар проекта.

Здесь нелишним будет вспомнить, что ещё лет двадцать назад на какой-то стародавней конференции по проблемам формализации спецификаций один из участников в пылу дискуссии заявил, что при наличии полной и строгой спецификации программировать уже не потребуется совсем – специальный транслятор сам преобразует спецификацию в готовую программу. Что он имел в виду? Современные CASE-системы? Скорее всего, те, которые ещё появятся. Только тогда они называться будут по-другому и построены будут, по моему глубокому убеждению, на других принципах, вроде экспертных систем. Такие системы должны быть умнее человека и даже группы людей, их создавших. А этого можно достичь, вероятно, только применяя методы обучения и накопления знаний предметной области. И только тогда с такой системой действительно сможет работать человек, просто владеющий мышкой.

А вот что касается сегодняшнего продвижения в направлении внедрения формальных методов проектирования для задач логического управления, то значительный вклад в решение обозначенных выше проблем уже проделан путем переноса достижений теории конечных автоматов на область программирования.

В начале 70-х Дуглас Росс, автор известной методологии IDEF0, утверждал, что 80 и даже 90% информатики будет в будущем основываться на этой теории. И хотя этого пока не наблюдается, всё же можно сказать, что конечные автоматы играют заметную роль в развитии компьютерных технологий. Они активно используются в реализации сетевых протоколов, системах сжатия информации, криптосистемах, компиляторах, операционных системах (UNIX). Иными словами, там, где требуется большая надёжность и где логика поведения чересчур сложна, чтобы программист смог реализовать её на одном лишь уровне здравого смысла.

Параллельно с развитием в Европе синхронного программирования в России развивается подход к разработке программного обеспечения, названный «автоматным программированием», которое можно рассматривать в качестве разновидности синхронного программирования. Предлагаемая концепция проектирования позволяет максимально подробно описывать процессы, которые необходимо автоматизировать, используя при этом понятный всем формальный язык конечных автоматов. Более того, предлагается чёткий и универсальный метод описания, единый для всех стадий проектирования.

В качестве языка спецификаций в этой технологии предлагается применять язык графов переходов дискретных устройств, соответствующий выбранному типу автомата (Мура, Мили). Такое «аппаратное» программирование называется SWITCH-технологией. Решение этой проблемы имеет особую важность в связи с большой ответственностью при управлении такими объектами, как ядерные или химические реакторы, системы управления летательными аппаратами, и зависит от наличия развитого математического аппарата теории автоматов.

Почему эмуляция?

Для того чтобы программировать, используя SWITCH-технологию, надо владеть теорией конечных автоматов. Лично мне в своих внедрениях, построенных на идее схемной эмуляции (к примеру, 64-канальном «релейном» регуляторе температуры для террариумов по выращиванию змей и прочих гадов или 3-канальном «ПИД-регуляторе» плавильных печей для ювелиров) удобно было пользоваться функциональной схемой выдуманного мною устройства.

Зато всё это при полном отсутствии программиста! И никакого программирования!!! Вроде как – паяем без паяльника!

Я знаю людей, «цифровиков» по духу, которым проще всего было реализовать проект в спроектированном ими цифровом устройстве, спаянном из дорогих сердцу вентилей, триггеров, счётчиков, мультиплексоров и т.д. Доходило до того, что из импортного упаковочного автомата (second-hand) изымался микропроцессор и на его место устанавливалось такое цифровое произведение.

Надо поменять алгоритм работы – пожалуйста: плата на ходу модернизировалась. Но что самое невероятное – всё это делалось быстрее, чем сделал бы человек, разбирающийся в микропроцессорах. Но ещё невероятнее – всё работало как часики и ни разу не дало повода хозяину частного заводика по производству продуктов питания даже задуматься: а не

пригласить ли мне другого специалиста.

Пусть каждый пользуется тем методом, которым владеет. Единственное, что остается неизменным в случае использования технологии эмуляции – наличие самой программы схемной эмуляции и графического рисунка проекта: схемы принципиальной, функциональной, алгоритма функционирования, графа переходов... Но насколько облегчилась бы жизнь этого любителя «цифр», окажись в его распоряжении технология эмуляции. Даже паять не надо было бы и бегать в поисках микросхем.

А теперь о том, что мне не понравилось во всякого рода автоматном программировании.

Во-первых, область применения ограничена программированием задач логического управления, т.е. ситуациями, где есть «1» или «0» («да» или «нет»).

Это – область, традиционно занимаемая обычными PLC-контроллерами.

В большинстве же задач автоматизации – всякого рода регуляторах – присутствуют аналоговые величины. Это могут быть уровни сигналов от датчиков, уровни управляющих напряжений, подаваемых на исполнительные устройства, географические координаты в геодезической системе, и т.д., и т.п.

Во-вторых, не подлежит сомнению прогрессивность идеи внедрения в практику программирования формальных (автоматных) методов, но моя душа восстаёт против того, что, составив алгоритм управления и обливаясь потом, нарисовав по нему граф переходов, я должен... взять клавиатуру в руки и ещё набрать по нему текст SWITCH-программы на Си или Паскале. Потому что, насколько мне удалось понять мировую «автоматную» мысль, программ, автоматически генерирующих программный код по графам переходов, пока нет.

Но даже если бы и были! Опять возвращение к пресловутым программным кодам???

Вы спросите меня, почему я так не люблю программные коды? Пожалуйста, напомню:

- программирование – самый длительный и сложный этап выполнения любого проекта, а сам факт участия программиста в проекте следует рассматривать как вредный и нежелательный;
- только графический рисунок может являться наиболее полной единой спецификацией проекта, понятной всем в цепочке: заказчик – технолог – проектант системы (раньше был программист);
- принципы организации алгоритма управления (рисунка проекта) и его программной модели не совпадают, другими словами, программа, написанная по графу, и сам граф неизоморфны. Получается, что нам предлагают использовать ещё один этап в общей цепочке проектирования, да ещё и такой, который может привносить ошибки. А ведь всюду в литературе, посвящённой вопросам «автоматного» программирования, подчеркивается актуальность его применения в особо ответственных областях, таких как ядерные реакторы, автопилоты и проч.;
- к достоинствам графических схем при их использовании в качестве языка алгоритмизации относится однозначность описания процессов, в том числе и параллельных, что теряется при переходе к текстам программ. Эта проблема вытекает из того, что функционально-потоковая параллельная программа является графом, множество вершин операторов этой программы связаны между собой информационными связями. Разнообразие этих связей и их неструктурированность в потоковых моделях трудно поддаётся линейному представлению в виде текста. О параллельности здесь я вспомнил не зря, поскольку авторский алгоритм эмуляции по своей природе является системой, органично реализующей идею параллельных потоков и поэтому наилучшим образом приспособленной для эмуляции параллельных систем. А наиболее естественный способ отображения таких систем – графический;
- любая программа способна реагировать только на те события и их комбинацию, которые предусмотрел программист, на всё остальное программа будет реагировать неадекватно или не будет реагировать вообще. Этим, кстати, можно объяснить широкое развитие на Западе общественных движений, выступающих против применения компьютеров в опасных технологических процессах, например, в атомной энергетике, ракетных системах стратегического назначения;
- давайте вспомним, что после того, как мы получили исходные коды, уже, возможно, содержащие неточности «перевода» (графическая схема – исходные коды программы), нам их ещё надо и откомпилировать, что также может стать источником ошибок в проекте. Компилятор сам по себе – штука сложная. Граф переходов или схема алгоритма функционирования проекта могут оказаться даже сложнее (пусть и меньшими по объёму). Транслятору же при обработке данных сложной структуры приходится для начала их правильно распознать, а затем вставить в машинный код множество дополнительных проверок, сформировать и обработать в памяти компьютера сложные объекты, что на уровне жёстко заданных машинных инструкций осуществить довольно трудно, а для сложных задач практически невозможно.

Но даже если мы прекрасно справились с задачей «перевода» графического рисунка проекта в исходные коды и нашли прекрасный компилятор, остаётся ещё один вопрос: а насколько вообще программирование способно точно отразить сложную динамику объектных реалий, нас окружающих?

Первые программы были абсолютно линейны, начинались со слова begin и заканчивались словом end. Это – подход «сверху-вниз»: программа начинается с первого оператора и заканчивается последним. Появление процедурно-ориентированного программирования изменило ситуацию настолько, что одной функции позволили временно прерывать другую, чтобы в это время точно так же линейно исполниться. Теперь программа начинается с первого оператора главной функции и заканчивается последним – той же главной функции.

Эти базовые принципы кардинально не изменились и с появлением объектно-ориентированного программирования. Ну, свалили в одну упаковку данные и процедуры, а потом начали обёртывать этой «бумажкой» экземпляры коробок и использовать для доступа к содержимому

название коробки. Ведь почти то же самое можно сделать и с процедурой – запихнуть в неё другие процедуры и данные.

Разбиение программы на объекты и процессы с заменой многоступенчатого ветвления средствами обработки сообщений (событий) заменяет одну проблему на другую: вложенность уменьшается, зато количество взаимодействующих компонентов заметно возрастает, а логика «размывается». Сегодня уже заговорили о недостатках объектов и достоинствах процессов – вместо ООП получилось ПОП. Такое положение вещей, как и подмена терминов, всегда сопутствовали программированию. Методологические надстройки – интересная штука – это такое надуманное творение, которое можно приспособить и к другим идеям. Достаточно заменить «объект» на «процесс», чтобы парадигму объектно-ориентированного программирования заменить на парадигму процессов...

К чему я всё это? Да к тому, что каким бы языком программирования ни пользовался программист, мы имеем на «выходе» один результат – исполняемый код управляющей программы со всем перечнем вложенных в неё статических состояний, которому под силу отреагировать только на достаточно статические внешние процессы. Электронные же устройства «не страдают» такими недостатками.

К сожалению, эти фундаментальные различия между «софтом» и «хардом» не поддаются какому-то вразумительному пониманию со стороны профессиональных программистов по одной простой причине: они не есть профессиональные электронщики.

В понятия «статический» и «динамический» они способны вложить что угодно: типы данных, указатели и прочее, но никак не динамические процессы, протекающие, например, в цифровом устройстве. Тут под динамичностью понимается не только сложная временная форма сигнала в отдельной цепи, но и временные соотношения сигналов в разных цепях, а также во всём устройстве.

Особую «пикантность» любому электронному устройству придают так называемые обратные связи, использование которых способно невероятно усложнить алгоритм его работы. В распоряжении же программиста, переводящего алгоритм задачи на язык программных операторов (каким бы языком он ни пользовался), имеется утомительно однообразный инструментарий структурных операторов: арифметических, логических, конструкций условий и циклов... Однако любой компилятор будет «повергнут в шок», если функции начнут вызывать «за хвост» одна другую. А вот самый обычный RS-триггер, собранный на двух микросхемах «И-НЕ» с обратными связями, преспокойно себе работает не один десяток лет, даже не подозревая о трудностях, его поджидающих на программистской ниве.

Достаточно один раз попросить любого программиста составить программную модель даже несложной электронной схемы, чтобы раз и навсегда отбить у него охоту заниматься этим впредь. А вот любой самый посредственный электронщик, владея каким-нибудь схемным симулятором, легко выполнит это задание.

Программные симуляторы – то, что у нас в советскую эпоху называлось моделированием, – это особый класс программных продуктов, которые реализуют специальные алгоритмы построения программной модели реального устройства в памяти компьютера.

Моделированию предшествует этап прорисовки (составления) схемы из готовых «кубиков» – компонентов схемы. Непосредственно рисунок схемы, а также описание входных воздействий являются необходимыми исходными данными. Тем не менее, современные схемные симуляторы работают в условной временной шкале, а не в режиме реального времени, и требуют ресурсов современных компьютеров.

Естественно, что при этом нет даже смысла говорить о возможностях какой-то стыковки с внешним оборудованием, что ограничивает круг использования такого рода программ этапом анализа при схемотехническом проектировании. Поэтому моделирование можно рассматривать как начальный уровень эмуляции. Настоящая эмуляция – это возможность реализации принципа «черного ящика», когда, наблюдая систему снаружи, невозможно сказать: спрятано там внутри устройство управления, реализованное аппаратно, или там находится PC, на котором имитируется его работа программным образом.

Алгоритм моделирования, изобретённый мною, совершенно оригинальный. Он абсолютно не похож на ранее и ныне известные и во всей полноте позволяет реализовать принцип «черного ящика». Реализован в программном продукте – «Схемный эмулятор “Пульс”» (авторское свидетельство ПА № 214 от 12.08.96 в Государственном Агентстве Украины по авторским и смежным правам). Во всей красе реализует принцип потока данных, обеспечивающий максимальный параллелизм. Рассчитан на эмуляцию аналого-цифровых устройств.

Конечно, каким совершенным ни был бы алгоритм, но попадая в среду фон-Неймана с её регистрами и одиозным счётчиком команд, он обречён на жалкую имитацию параллельных потоков. Тем не менее, даже в этих условиях программа работает в режиме реального времени в таких, например, приложениях, как эмуляция устройств автоматики в системах АСУ ТП. И не важно, что именно мы эмулируем: функциональную схему устройства, схему алгоритма работы или схему принципиальную.

Объём модуля эмуляции настолько мал (<100 кб), что вместе со скомпонованными библиотечными компонентами может разместиться в микропроцессоре для встраиваемых систем, но наиболее полно возможности программы могут раскрыться только при «аппаратной» реализации алгоритма на микросхемах программируемой логики.

Процесс разработки проектов в среде «Пульса» заключается в «сборке» в среде графического редактора графического рисунка из готовых «кубиков» – функциональных блоков. Каждому такому блоку могут соответствовать реальные узлы аналоговых и дискретных приборов: датчиков температуры, оборотов, давления, аналого-цифровых (цифро-аналоговых) преобразователей, релейных входов и выходов, а также функциональных блоков: компараторов, нормализаторов, ПИД-регуляторов и т.д.

Время разработки проекта определяется временем «сборки» рисунка проекта и соединения реальных «кубиков» в единый интерфейс.

Девяностые годы прошлого столетия ознаменовали первую реальную

попытку превратить разработку программного обеспечения в инженерную дисциплину с помощью концепции CBSE (component-based software engineering – «компонентная разработка программного обеспечения») и COTS (commercial off-the-shelf – «готовые коммерчески доступные компоненты»).

Идея состояла в создании высококачественной системы объединения таких модулей, однако по сей день окончательно не решена. Проблема заключается в том, что объединенные вместе высококачественные модули не обязательно превращаются в высококачественную систему.

Комбинированная система может оказаться никуда не годной из-за некорректного способа объединения. Парадигма «разделяй и властвуй», которая оправдывает себя в случае аппаратных систем, может оказаться губительной для систем логических.

Идея схемной эмуляции выступает прекрасной демонстрацией единства математических основ аппаратуры и программного обеспечения.

Во Всемирной сети мне как-то попалась рецензия Дмитрия Гориловского (Компьютерра. 2002. № 14. С. 59) на книгу Шалыто А.А. «Логическое управление. Методы аппаратной и программной реализации алгоритмов», из которой я приведу один абзац:

«Проведённая в книге аналогия между аппаратными и программными реализациями алгоритмов позволяет надеяться, что в недалёком будущем появятся программы, работающие столь же надёжно, как и современное “железо”.»

Я думаю, что использование идеи схемной эмуляции как завершающего этапа автоматного проектирования можно рассматривать как весомый вклад в достижение этой цели. Теперь мы можем рассматривать любой проект так, будто он полностью реализован аппаратными средствами.

АСУ ТП

В 2000 году я встретился с сотрудником института им. Е.О. Патона Дорошенко В.Д., ставшим впоследствии фанатом идеи эмуляции и автором всевозможного «железа». Он, как человек, активно занимающийся всякого рода внедрениями, к тому времени пришёл к пониманию того, что в борьбе за клиента победит тот, кто первым научится реализовывать проекты быстро и с высочайшим качеством. А для этого надо иметь в своём арсенале готовые и тщательно отлаженные «кубики». И не дай Бог, чтобы в проекте участвовал программист – самый для него «ненавистный» человек, поскольку только от программиста можно было услышать что-либо вроде: «Я вчера не успел закончить программировать микропроцессор, потому что у жены был день рождения». Поэтому идея эмуляции «кубиков» была воспринята как естественное решение проблемы.

К слову сказать, тема проектирования систем автоматики и вообще АСУ ТП есть самая подходящая для идеи «игры в кубики». И понятно почему: относительно небыстрые процессы, которые можно отследить, расположив программу в компьютере или микропроцессоре для встраиваемых систем, разнообразие оборудования, распределённость в пространстве, актуальность сроков завершения и т.д.

Через несколько месяцев совместных поисков и трудов на свет появилось первое творение: в пластмассовый пыленепроницаемый промыш-

ленный корпус была установлена материнская плата с Intel 386. Там же прекрасно разместился матричный жидкокристаллический индикатор Bolymin, заменив дисплей, и плёночный пульт управления. Жёсткий диск с программой и библиотекой компонентов был заменён флэш-диском. В другой слот была установлена плата стыковки ISA-I²C.

Интерфейс I²C (100 МГц) был выбран не в результате предвзятых мнений о преимуществах одного интерфейса над другими. Просто этот вариант оказался наиболее подходящим по соотношению цена–доступность–технологичность, а также из-за поддержки этого интерфейса ведущими мировыми производителями микросхем, такими, к примеру, как Atmel и Philips Semiconductors.

Кстати, как индикатор, так и пульт тоже имели встроенный интерфейс и наравне с периферийным оборудованием составляли библиотеку компонентов. Только соединялись в общую сеть «коротким» (двухпроводным) интерфейсом, в то время как все остальные внешние приборы были соединены по четырёхпроводной линии, позволявшей сделать её длиннее почти на два порядка.

На этапе отладки в ISA-слот можно было вставить видеокарту и наблюдать уровни сигналов во всех частях системы.

Разработка такого устройства преследовала цель – отработать идею «универсальной эмулирующей платформы» для АСУ ТП, чтобы затем перенести наработки на уровень микропроцессоров для встраиваемых систем и микросхем программируемой логики.

Это может быть символично, но идея составлять программы при помощи графического рисунка, вероятно, впервые была реализована именно в области автоматизации. Ныне на рынке предлагается огромный выбор подобных средств, я же остановлюсь на примере UltraLogik.

Ниже я попытаюсь коротко описать программу и порядок работы при составлении проекта. Это позволит человеку, внимательно читающему статью с самого начала, увидеть роль и место идеи эмуляции в контексте сравнения возможностей с существующим инструментарием.

Итак, программа состоит из двух компонентов: системы программирования, работающей на PC-совместимом компьютере, и системы исполнения, работающей на целевом контроллере. Система программирования функционирует в операционной среде Windows. Программировать, конечно, можно просто на Си, Паскале или Ассемблере. Но можно программировать и в среде графического интерфейса, опираясь на возможности функциональных блоковых диаграмм – FBD. Интерфейс позволяет описывать функции между входными и выходными переменными. Программы на языке FBD напоминают электрические принципиальные схемы логических устройств и формально соблюдают алгоритм их работы.

Однако на этом сходство заканчивается, поскольку такие схемы обязательно содержат метки, операторы условного и безусловного переходов. Система исполнения работает на открытом PC-совместимом контроллере серии ADAM или MicroPC (Octagon Systems) под управлением ROM-DOS. Это позволило применять PC-совместимые машины в области решения задач «жёсткого» реального времени, традиционно занимаемой «классическими» PLC.

Используя глобальные переменные, UltraLogik осуществляет сетевое взаимодействие между контроллерами и системой визуализации данных, которая устанавливается на обычном ПК (верхнего уровня). В роли системы верхнего уровня можно использовать любой SCADA-пакет, например, Genesis или Genie, используя для стыковки специальные DLL-модули. Система верхнего уровня служит для отображения процессов в объекте регулирования и не выполняет задач алгоритмического управления. Программы управления расположены непосредственно на контроллерах нижнего уровня, которые работают в непрерывном режиме.

Основные этапы создания программы в среде UltraLogik:

- заполнение таблиц глобальных переменных;
- конфигурирование контроллера;
- привязка переменных ко входам и выходам контроллера;
- разработка алгоритмов программ;
- компиляция;
- загрузка исполняемого кода в контроллер;
- отладка программы (?!).

Но UltraLogik – это ещё «цветочки»! Практически все современные пакеты графического программирова-ния – это SCADA-программы, требующие ресурсов мощных промышленных ПК, они не предназначены для работы на контроллерах нижнего уровня.

А теперь проведём сравнение возможностей программ визуального программирования на примере Ultra-Logik с возможностями схемной эмуляции.

Несмотря на декларативные заверения разработчиков средств визуального программирования о простоте и интуитивно понятном интерфейсе, такие системы по-прежнему достаточно сложны и требуют «ручной» работы: конфигурирования, привязки переменных и многого другого, что упускается в рекламных статьях.

В «Пульсе» «программирование» начинается и заканчивается прорисовкой графического рисунка проекта (сборка из готовых «кубиков»).

В современных системах графического программирования могут использоваться: языки функциональных блоковых диаграмм (FBD) и последовательных функциональных блоков (SFC), язык релейных диаграмм (LD), язык структурированного текста (ST), язык инструкций (IL). Всё это достаточно «древние» языки, которые придумывались на заре программируемых логических контроллеров (PLC с закрытой архитектурой) для решения достаточно простых задач автоматизации логического управления. Сейчас прослеживается попытка приспособить их под новые платформы (промышленные компьютеры с открытой архитектурой) и новые, более сложные, задачи управления. Поэтому уверен, что к серьезным проектам, прежде всего, будет привлечён профессионал, владеющий Си.

«Пульс» позволяет работать с разнообразными схемами: принципиальными, функциональными, алгоритмов, графов. Нет необходимости пользоваться такими рудиментами, как метки и переменные. Данные передаются непосредственно по линиям связи, соединяющим «кубики». Это касается и организации мультиконтроллерных систем: если одним сложным агрегатом, входящим в технологическую линию, необходимо управлять с помощью нескольких контроллеров, то достаточно в каждом контроллере разместить соответ-

ствующий фрагмент общей функциональной схемы (рисунка проекта) для данного агрегата.

Программа эмуляции может быть размещена не только на PLC открытой архитектуры (PC-подобной машине), но и на обычном микропроцессоре для встраиваемых систем, например серии AVR, не говоря о более мощных современных кристаллах, что позволит использовать её в условиях более жёсткого режима реального времени и повышенных помех.

Наилучшим решением стала бы разработка универсального контроллера для встраиваемых систем на базе микросхемы программируемой логики. Это будет эквивалентно «аппаратной» реализации алгоритма эмуляции, позволяющей достичь наивысшей скорости работы (в особенности для параллельных алгоритмов) и защиты от электрических помех.

Таким образом, здесь поднимается вопрос о возможности создания универсальных эмулирующих платформ трёх видов: PLC-PC, на встраиваемых микропроцессорах и на микросхемах программируемой логики.

За пользователем остаётся выбор оптимального варианта по своему усмотрению.

Первоначально общий алгоритм управления следует эмулировать на обычном ПК, используя монитор для отладки самого алгоритма, а затем перенести задачу на одну или несколько встраиваемых платформ.

Идею распределённого управления на встраиваемых эмулируемых платформах следует рассматривать как альтернативу централизованному управлению, реализованному на дорогостоящих промышленных компьютерах и серверах.

И никаких отладок программы, никакой компиляции и поиска ошибок!

Новости мира News of the World Новости мира

130 нанометров отправляются на пенсию

Компания Intel намерена постепенно свернуть производство процессоров, изготовленных с применением технологической нормы 130 нм. Официальных причин такого шага не называется, однако ясно, что процессорный гигант планирует освободить производственные мощности, для того чтобы увеличить количество процессоров, изготавливаемых с применением технологической нормы 90 нм. Сокращение затронет как модели, предназначенные для настольных систем, так и мобильные процессоры для ноутбуков. Первым делом пойдут на пенсию процессоры Pentium 4 2,80 ГГц, 3,06 ГГц и Pentium 4 3,20 ГГц с шиной 533 МГц. За ними последуют мобильные процессоры Mobile Pentium 4-M 2,20 ГГц, 2,40 ГГц, 2,50 ГГц и 2,60 ГГц. Та же судьба ждёт и бюджетные Mobile Celeron, с производства будут сняты модели с тактовыми частотами 2,20 ГГц, 2,40 ГГц и 2,50 ГГц. Наконец, компания намерена отказаться от дальнейшего изготовления процессоров Celeron M 1,20 ГГц и Pentium M 1,0 ГГц ULV. Таким образом, в общей сложности приговорены 13 процессоров, некогда считавшихся самыми передовыми.

www.anandtech.com
www.computery.ru

Компания KDDI создала прототип топливного элемента

Японская компания KDDI, объединившись с Hitachi и Toshiba, создала прототип топливного элемента питания для мобильных устройств и настойчиво продвигается к созданию в 2007 г. его коммерческого варианта. Заряда литий-йонных батарей современных мобильных телефонов в режиме просмотра цифрового телевидения (уже доступного в Японии) хватает на 2 часа. Топливный элемент, работающий на метаноле, удвоит это время.

www.the-inquirer.com

Прозрачный гибкий проводник на основе нанотрубок

Гибкое и прозрачное проводящее покрытие на основе углеродных нанотрубок не только ускорит создание гибких дисплеев и электронной бумаги, но и позволит заменить оксид оловяно-индиевого сплава. Учитывая потребность в прозрачных проводниках при производстве дисплеев, исследовательская компания Eikos разработала технологию создания прозрачных проводящих пленок с использованием углеродных нанотрубок. Прочное, гибкое, оптически нейтральное проводящее покрытие, названное Invisicon, позволит создавать гибкие дисплеи и фотогальванические фотоэлементы.

Технология использует преимущество в прочности, гибкости, цветовой нейтральности нанотрубок перед хрупкой и имеющей цветовую окраску окисью индия. Нанотрубки смешиваются со связующим полимером, в результате чего получаются прозрачные проводящие чернила, которые затем наносятся на акриловую или стеклянную подложку.

Материал выдерживает растяжение до 18% и 25 тыс. циклов изгиба и скручивания с минимальными потерями проводимости. Сейчас технология позволяет создавать покрытия с удельным сопротивлением 200 Ом/квадрат, в 2005 г. предполагается достичь значения 50 Ом/квадрат.

www.eikos.com

Полупроводниковые лазеры с изменяемой длиной волны

Японская фирма NTT разработала технологию полупроводниковых лазеров с изменяемой длиной волны, что ранее было возможно только для газовых лазеров. Технология использует оптически нелинейный кристалл ниобата лития для сложения двух лазерных лучей различных частот, в результате чего образуется луч с суммарной или разностной частотой. Технология позволяет генерировать лазерный луч с определённой фиксированной длиной волны в диапазоне от 500 нм до 5 мкм подбором толщины кристалла и длины волны лазеров.

Лазеры, созданные по этой технологии, имеют выходную мощность 10...30 мВт и могут работать непрерывно при комнатной температуре.

www.ntt.com

Уважаемые читатели!

БЕСПЛАТНУЮ ПОДПИСКУ НА ЖУРНАЛ вы можете оформить,
заполнив анкету на этой странице (ксерокопии принимаются)
или на сайте журнала www.soel.ru.

Фамилия, имя, отчество . . . . . .
Фирма . . . . . .
Должность . . . . . .
Телефон (. . .) . . . . . .
Факс (. . .) . . . . . .
E-mail . . . . . .
Сайт . . . . . .

Адрес предприятия
Индекс . . . . . .
Город, район, область . . . . . .
Адрес . . . . . .

Почтовый адрес для доставки журнала, если он отличается от адреса предприятия
Индекс . . . . . .
Город, район, область . . . . . .
Адрес . . . . . .

Размер фирмы
- ❑ До 10 чел.
- ❑ 10...50 чел.
- ❑ 50...100 чел.
- ❑ Более 100 чел.
- ❑ Более 1000 чел.

Ваши должностные обязанности
- ❑ Разработка
- ❑ Комплектация производства
- ❑ Организация производства
- ❑ Менеджмент
- ❑ Реклама, PR
- ❑ Руководство
- ❑ Другое . . . . . .

Сфера деятельности вашей фирмы
- ❑ Силовая электроника, импульсные источники питания
- ❑ Отображение информации
- ❑ Вычислительная техника
- ❑ Системная интеграция
- ❑ Промышленная автоматизация
- ❑ Связь, телекоммуникации
- ❑ Медицинская техника
- ❑ Охранные системы
- ❑ Бытовая электроника
- ❑ Производство печатных плат
- ❑ Контрактное производство
- ❑ Другое . . . . . .

Компоненты каких фирм вы используете в работе . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

Уважаемые рекламодатели!

Журнал «Современная электроника» имеет беспрецедентный для специализированного журнала тираж 15 000 экземпляров. Схема распространения журнала — по подписке, в розницу, через региональных распространителей, на семинарах и выставках, прямая рассылка ведущим компаниям стран СНГ — позволит вашей рекламе попасть в руки людей, принимающих решения о приобретении компонентов, приборов и оборудования, а также об организации производства изделий электроники.