Одним из важных и распространённых случаев вычислительных систем реального времени (ВСРВ) является ВСРВ с периодически поступающими на обработку входными данными. Если при этом возможна приемлемая для данного приложения точность оценки времени на обработку входной информации, возможно разбить процесс работы ВСРВ на предварительную и основную стадии. На предварительной стадии и не в режиме реального времени первоначально описывается (средствами построенного для этих целей формального языка) состав и периодичность поступающей информации, соответствующие модули-обработчики, последовательность обработки данных, директивные сроки обработки тех или иных данных и допустимость прерывания этой обработки и т.д. Затем (с учётом необходимого согласования процессов обработки, предотвращения программных тупиков и учёта иных особенностей программируемых ВСРВ) автоматически рассчитывается допустимое расписание работы ВСРВ, автоматизированно составляется программа выполнения режима реального времени, обеспечивается создание необходимого для её выполнения программного окружения, а на основном этапе производится собственно запуск полученной ВСРВ.

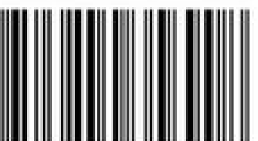
Планирование вычислений в САПР СРВ

Дмитрий Русланович Гончар

Методы планирования вычислений в САПР систем реального времени

Методы планирования вычислений в САПР СРВ

Дмитрий Русланович Гончар



Дмитрий Русланович Гончар

Окончил Московский авиационный институт по специальности "АСУ производством" и Московский физико-технический институт по специальности "Автоматизация экспериментальных исследований". С 1990 г. работаю в Вычислительном центре Российской академии наук.

978-3-8443-5690-8

Содержание

ВВЕДЕНИЕ **7**

ГЛАВА 1. СИСТЕМА АВТОМАТИЗАЦИИ ПРОГРАММИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ. ОБЩАЯ СХЕМА, ПОТОКИ ИНФОРМАЦИИ, СТРУКТУРА СИСТЕМЫ **17**

1.1. Задачи, назначение и общая схема САПР систем реального времени «СРВ-Конструктор» **17**

1.2. Язык реального времени **21**

1.3. Основные блоки транслятора **24**

1.4. Управляющий монитор **25**

ГЛАВА 2. ВХОДНОЙ ЯЗЫК САПР ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ **27**

2.1. Синтаксис **29**

2.2. Типы данных **29**
2.2.1. Целые константы 29
2.2.3. Длинные целые константы 30
2.2.3. Константы с плавающей точкой 30
2.2.4. Константы с плавающей точкой двойной точности 31
2.2.5. Символьные константы 31
2.2.6. Строковые константы 31
2.2.7. Булевские константы 31

2.3. Описания **32**
2.3.1. Константные величины 32
2.3.2. Переменные 32
2.3.3. Источники поступления информации в ВСРВ 42
2.3.4. Кадр входных данных 43
2.3.5. Прикладные модули 44
2.3.6. Таблица переключений 46

2.4. Исполняемые конструкции **47**
2.4.1. РВ-циклы 47
2.4.2. Предварительная и заключительная части простых заданий 52
2.4.3. Фоновые работы 54
2.4.4. Модуль реакции 56

2.5. Структура РВ-программы **57**
2.5.1. Условное задание 58
2.5.2. Простое задание 59

ГЛАВА 3. СИСТЕМА АВТОМАТИЗИРОВАННОГО СИНТЕЗА МОДЕЛИ ВСРВ. ГЕНЕРАТОР СЕТЕВОЙ МОДЕЛИ И РАСПИСАНИЙ **61**

3.1. Основные функции генератора сетевых моделей и расписаний **62**

3.2. Структурная схема и последовательность выполнения основных блоков **63**
3.2.1. Основные определения и обозначения 63

3.3. Основные алгоритмы **65**
3.3.1. Построение сети М-модулей 65
3.3.2. Вычисление директивных интервалов 66
3.3.3. Построение допустимого расписания 67
3.3.4. Построение таблицы соответствия *T*-, *E*- и *B*-модулей 67
3.3.5. Вычисление размеров буферов для входных параметров 68
3.3.6. Назначение стеков Т-модулям 69

ГЛАВА 4. УПРАВЛЯЮЩАЯ ПРОГРАММА САПР «СРВ-КОНСТРУКТОР» **71**

4.1. Выбор операционной среды **71**

4.2. Основные функции управляющей программы **72**

4.3. Структура управляющей программы **72**
4.3.1. Интерпретатор команд 72
4.3.2. Диспетчер 73
4.3.3. Монитор данных 75
4.3.4. Драйверы внешних устройств 75

ГЛАВА 5. ПРОГРАММНЫЙ КОМПЛЕКС «СРВ-КОНСТРУКТОР» **77**

5.1. Сборка и запуск программного комплекса «СРВ-Конструктор» **77**

5.2. Генератор кодов. **78**

ГЛАВА 6. ПЛАНИРОВАНИЕ РАСПИСАНИЙ ДЛЯ МНОГОПРОЦЕССОРНОГО ВАРИАНТА СИСТЕМЫ «СРВ-КОНСТРУКТОР». ЗАДАЧА РАСПРЕДЕЛЕНИЯ *M* ЗАДАНИЙ НА *N* ПРОЦЕССОРОВ **81**

6.1. Постановка задачи для случая процессоров одинаковой производительности **81**

6.2. Постановка задачи для случая процессоров различной производительности **82**

6.3. Существующие методы решения **83**
6.3.1. Алгоритмы случайного поиска 83
6.3.2. Алгоритмы детерминированной коррекции расписаний 84
6.3.3. Алгоритмы имитации отжига 88
6.3.4. Генетические и эволюционные алгоритмы 89
6.3.5. Алгоритмы агрегирования 90

6.4. Выводы **91**

ГЛАВА 7. ЭВРИСТИЧЕСКИЕ АЛГОРИТМЫ РАСПРЕДЕЛЕНИЯ ЗАДАНИЙ ПО ПРОЦЕССОРАМ В САПР СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ 93

7.1. Решение задачи 6.1 (для случая одинаковых процессоров). 94
7.1.1. Эвристический алгоритм 1 94
7.1.2. Контрольный алгоритм 1 95
7.1.3. Контрольный алгоритм 2 95
7.1.4. Сравнительные итоги расчётов по алгоритму 1 с разными процентами калибровки 95

7.2. Решение задачи 6.2 (для случая различных процессоров) 97
7.2.1. Описание жадного алгоритма 97
7.2.2. Описание эвристического алгоритма 2 98
7.2.3. Вычислительные эксперименты и рекомендации к применению 99

ЗАКЛЮЧЕНИЕ 103

ЛИТЕРАТУРА 105

ПРИЛОЖЕНИЕ 1. ПРИМЕР ПРИМЕНЕНИЯ САПР «СРВ-КОНСТРУКТОР». РЕШЕНИЕ В РЕАЛЬНОМ ВРЕМЕНИ ЗАДАЧИ МНОГОМЕРНОЙ ЛИНЕЙНОЙ РЕГРЕССИИ 113

ПРИЛОЖЕНИЕ 2. ТЕКСТ ПРОГРАММ, РЕАЛИЗУЮЩИХ ЭВРИСТИЧЕСКИЙ АЛГОРИТМ РАСПРЕДЕЛЕНИЯ ЗАДАНИЙ ПО ПРОЦЕССОРАМ ДЛЯ МНОГОПРОЦЕССОРНОГО ВАРИАНТА САПР «СРВ-КОНСТРУКТОР» 125

БЛАГОДАРНОСТИ

Введение, главы 1, 5, 6, 7 и Приложение 2 написаны Гончаром Д.Р. При написании глав 2, 3, 4 и Приложения 1 были использованы материалы и результаты, полученные коллективом сотрудников сектора проектирования систем реального времени ВЦ РАН под руководством Сушкова Б.Г.: Сушковым Б.Г., Фуругяном М.Г., Кондратьевым О.Н. и Мирошником С.Н.

Считаю своим долгом выразить благодарность Б.Г.Сушкову, Ю.А.Флёрову, М.Г.Фуругяну, О.Л.Кондратьеву, А.В.Сухих, О.С.Федько и С.Н.Мирошнику за помощь в работе и обсуждение полученных результатов.

Введение

Вычислительные системы реального времени (ВСРВ) предназначены для контроля и управления при жёстких временных ограничениях процессами в автоматизированных производствах, в энергетике (особенно ядерной), химической промышленности, газо- и нефтедобыче, авиационном, железнодорожном и морском транспорте, в системах управления лётными испытаниями, при управлении в чрезвычайных ситуациях. Сегодня ВСРВ всё шире применяются в управлении дорожным движением автотранспорта, экологическом и медицинском мониторинге, разнообразных системах безопасности.

Изменился и диапазон масштабов разрабатываемых ВСРВ, с одной стороны (с учётом новых возможностей и большей доступности сетевых технологий) расширившись с масштаба цеха или предприятия до масштабов отрасли, государства и международных систем (например, спасания на море), с другой – став доступными на уровне бортового компьютера для серийного современного автомобиля.

Одновременно с увеличением масштабов и разнообразия применений ВСРВ становятся более высокими требования к их эффективности, надёжности и скорости разработки. Удовлетворить этим отчасти противоречивым требованиям представляется возможным, прежде всего, путём выхода на более высокий уровень автоматизации построения систем реального времени, созданием соответствующих эффективных, надёжных и удобных инструментов их программирования.
Одним из практически важных и весьма распространённых случаев ВСРВ является ВСРВ с периодически поступающей на обработку входной информацией (см. рис. 1.1). Если при этом возможна приемлемая для данного приложения точность оценки времени на обработку входной информации, возникает возможность разбиения процесса работы ВСРВ на две стадии – предварительную и основную.

При этом на предварительной стадии и не в режиме реального времени первоначально описывается (например, средствами специально построенного для этих целей формального языка) состав и периодичность поступающей информации, соответствующие модули-обработчики, последовательность обработки данных, директивные сроки обработки тех или иных данных и допустимость прерывания этой обработки и т.д.

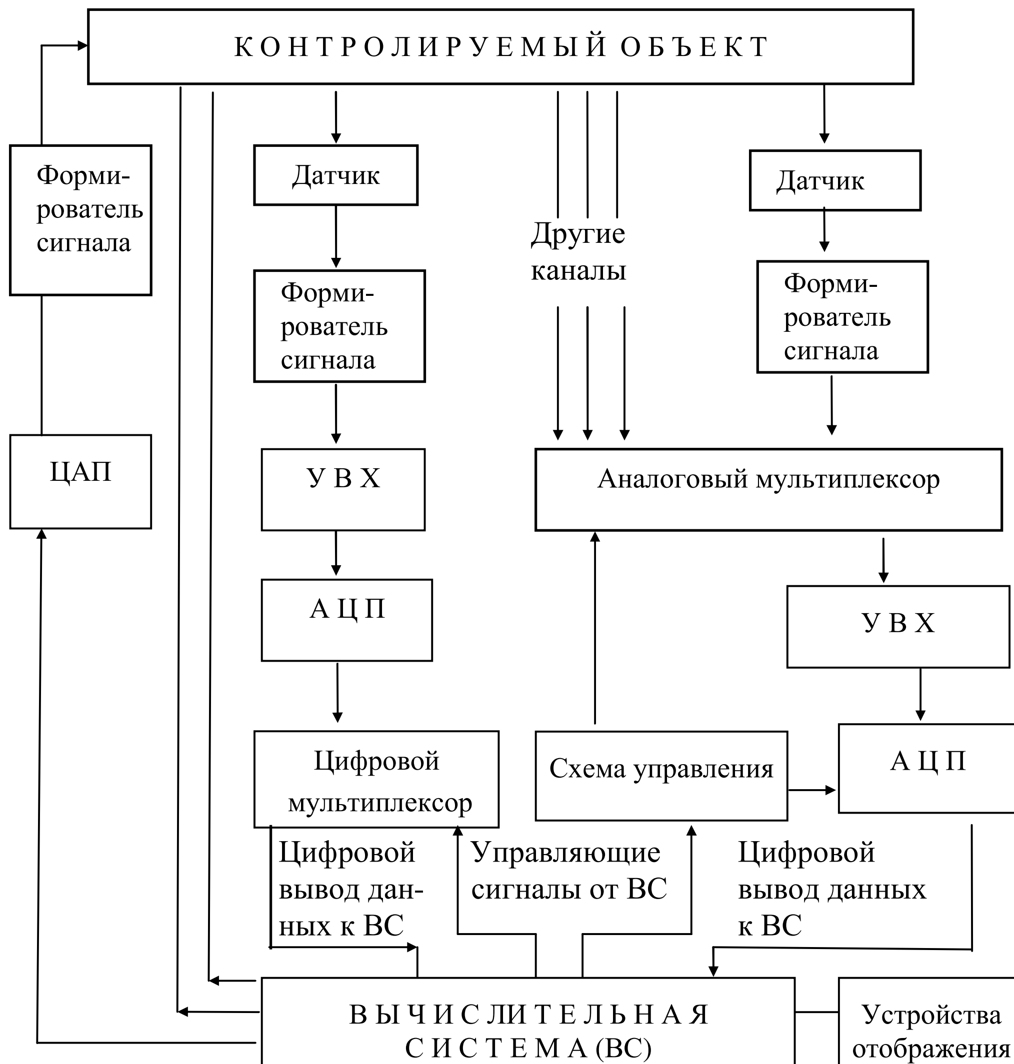


Рис. 1.1. **Схема использования ЭВМ для управления физическими процессами**. *Потоки информации и аппаратные средства типичной многоканальной системы сбора данных и управления представляются на различных устройствах отображения (дисплеях и т.п.) и, кроме того, если необходимо, вырабатываются управляющие воздействия, которые после соответствующих преобразований поступают к объекту, управляя его функционированием. АЦП – аналогово-цифровой преобразователь. ЦАП – цифро-аналоговый преобразователь. УВХ – устройства выборки-хранения аналогового сигнала.*

Затем (с учётом необходимой синхронизации процессов обработки, предотвращения программных тупиков и учёта иных особенностей программируемых ВСРВ) автоматически рассчитывается допустимое расписание работы ВСРВ (либо делается вывод о невозможности существования такого расписания), составля-

ется с использованием средств автоматизации программирования программа выполнения режима реального времени, обеспечивается создание нужного количества копий программных модулей обработчиков входных и промежуточных данных ВСРВ (с учётом возможности одновременной обработки нескольких поколений входных данных), создание средств связи между этими модулями и операционной системой (ОС) реального времени и т.п., а на основном этапе производится собственно выполнение полученной ВСРВ в режиме реального времени.

Именно такой подход, позволяющий не только качественно повысить эффективность, скорость и надёжность разработки (настройки, усовершенствования) ВСРВ, но и существенно расширить возможный диапазон и сложность решаемых задач (особенно при управлении быстропротекающими процессами) был предложен в ВЦ АН СССР лауреатом Премии СМ СССР, к.ф.-м.н. Борисом Григорьевичем Сушковым (1941-1997) для ЕС ЭВМ.

Уже в 90-е годы была понята актуальность и важность разработки подобной системы для персональных ЭВМ. Такая инструментальная система автоматизации программирования (САПР) «СРВ-Конструктор» была создана при непосредственном существенном участии автора диссертации и подробно представлена в данной работе.

Важной составляющей инструментальной САПР «СРВ-Конструктор» является блок, определяющий существование допустимого расписания и (если оно существует) выполняющий его построение. Понятно, что для многопроцессорных вычислительных комплексов важность и, в то же время, сложность этого блока только возрастает. Поэтому большое внимание в диссертации уделено разработке алгоритмов построения расписания для ряда важных частных случаев, часто встречающихся в работе подобных многопроцессорных систем.

В наиболее общей формулировке задачи составления расписаний состоят в следующем. С помощью допустимого набора ресурсов или обслуживающих устройств должна быть выполнена система заданий. Цель заключается в том, чтобы при известных свойствах заданий и ресурсов и наложенных на них ограничениях найти эффективный алгоритм упорядочения заданий, оптимизирующий желаемую меру эффективности. В качестве меры эффективности обычно рассматривается длина расписания или среднее время пребывания заданий в системе. Модели этих задач являются детерминированными в том смысле, что вся информация, на

основе которой принимаются решения об упорядочении, известна заранее. В частности, задания и все данные о них предполагаются известными в начальный момент времени.

Системы жёсткого реального времени – это такие системы, в которых некоторым заданиям сопоставляются директивные сроки (наиболее ранний момент начала выполнения задания и наиболее поздний момент окончания выполнения), не подлежащие нарушению. Для обеспечения работоспособности и надёжности таких систем необходимо иметь заранее рассчитанное расписание работы всей системы. В данной работе рассматриваются, в частности, эффективные алгоритмы составления расписаний для многопроцессорных систем жёсткого реального времени.

Применение таких систем позволяет уменьшить число отказов сложных технических систем. Во многих случаях управления производством или сложными техническими системами целесообразно применение различного рода систем мониторинга в реальном масштабе времени.

В качестве иллюстрации важности временн*ы*х ограничений на работы в системах жёсткого реального времени можно привести следующие примеры:

1. Системы управления ядерными энергетическими (АЭС) или силовыми установками.
2. Системы управления технологическими цепочками в целом ряде производств химической промышленности.
3. Системы раннего обнаружения и отслеживания (мониторинга) опасных природных явлений (землетрясений, цунами, наводнений и т.п.)
4. Системы мониторинга существенных параметров экологической и экономической обстановки.
5. Разнообразные бортовые системы управления и испытаний транспортных систем, особенно в авиации и космонавтике.
6. Весьма распространённой ныне является задача рационального подбора конфигурации самих ЭВМ и локальных вычислительных сетей (ЛВС) на их основе, в том числе в реальном времени (без остановки вычислительного процесса), которая в свою очередь распадается на целый ряд ещё более частных задач, например, таких, как определение оптимальной/ рациональной

последовательности запуска вычислительных заданий на данной конфигурации вычислительных средств.

7. При составлении учебного расписания ВУЗа необходимо учесть разнообразие специальностей учебных групп, разнообразные ограничения с точки зрения служебных обязанностей, человеческих возможностей и интересов студентов, преподавателей, иного обслуживающего персонала, наличных помещений и оборудования. При этом должны учитываться директивные сроки проведения занятий и завершения освоения запланированных курсов.

Вышеперечисленные обстоятельства обусловливают актуальность исследований в области дальнейшего совершенствования методов оперативного планирования, в том числе расчёта и составления рациональных или (если возможно и оправданно) оптимальных расписаний в указанных областях. В том числе с использованием таких инструментальных средств как ВСРВ «СРВ-Конструктор».

Многие варианты задач составления многопроцессорного расписания являются NP-трудными. Принимая во внимание трудность задачи, любая практическая реализация составления многопроцессорного расписания – это всегда своеобразный компромисс между результатом и вычислительной сложностью. Поэтому вопрос составления более эффективных эвристических алгоритмов, в том числе для конкретных видов задач составления расписаний, является в настоящее время достаточно актуальным для рассматриваемой предметной области.

Создание и использование ВСРВ стало актуальной задачей при появлении достаточно надёжной и мощной вычислительной базы, что, как известно, произошло к 70-х годам XX в. С этого времени изучению математических вопросов теории расписаний и её приложений исследователи уделяют повышенное внимание.

В России, а до того в СССР, задачей построения расписаний в системах реального времени занимались Танаев В.С.[1], Барский А.Б.[2], Головкин Б.А.[3], Сушков Б.Г.[4, 5, 6 и др.], Шкурба В.В.[7], Гордон В.С.[1], Костенко В.А.[8], Лазарев А.А.[9], Мищенко А.В. [10], Португал В.М.[11], Сигал И.Х.[12, 13, 14], Шафранский В.С. [1] и др.

Среди зарубежных учёных следует отметить Бернса (Burns)[15], Брукера (Bruker)[16], Гонзалеса (Gonzales) [17], Греневельта (Groenevelt) [18], Гэри М. (Garey)[19], Дертузоса (Dertouzos)[20], Джонсона Д. (Johnson)[19], Конвея Р.В. (Con-

way)[21, 22], Кормена Т. (Cormen) [23], Коффмана (Koffman)[24, 25], Лью (Liu)[26], Лейланда (Layland)[26], Лейзерсона Ч.(Leiserson) [23], Максвелла В.Л.(Maxwell) [21, 22], Мартеля (Martel)[27], Миллера Л.В.(Miller)[21, 22], Мока (Mok)[28], Одсли (Audsley) [29], Рамамритама (Ramamritham)[30], Ривеста Т. (Rivest) [23], Ричардсона (Richardson)[29], Сахни (Sahni)[17], Станковича (Stankovic)[30, 31], Ульмана Дж.(Ullman) [32], Федергрюна (Federgruen) [18] и др.

Считаю своим долгом выразить благодарность Б.Г.Сушкову, Ю.А.Флёрову, М.Г.Фуругяну, О.Л.Кондратьеву, А.В.Сухих, О.С.Федько и С.Н.Мирошнику за помощь в работе и обсуждение полученных результатов.

Цели и задачи исследования.

Основной целью диссертационной работы является разработка программного комплекса инструментальной САПР «СРВ-Конструктор» для персональных электронно-вычислительных машин, а также новых методов составления расписаний, предназначенных для функционирования на многопроцессорной ВС. Эти методы можно включить в состав программных средств, предназначенных для осуществления планирования вычислений на вычислительных системах, в том числе системах жёсткого реального времени.

Для достижения поставленной цели:

– создан программный комплекс, реализующий инструментальную САПР «СРВ-Конструктор»;

– с целью дальнейшего совершенствования указанной САПР для многопроцессорных вычислительных комплексов разработаны и реализованы новые эвристические алгоритмы решения задачи построения оптимального по быстродействию расписания без прерываний.

СРВ-Конструктор состоит из:

Блока синтаксического и семантического анализа, осуществляющего синтаксический и семантический анализ конструкций программы реального времени (РВ-программы), описывающей на формальном языке необходимый порядок выполнения прикладных программ пользователя, генерирующего таблицы данных для работы последующих блоков и вычисляющего размеры буферов обмена данными между программными модулями.

Блока генерации сетевой модели и расписаний, строящего математическую модель вычислений, выполняемых в реальном времени, в виде графа, в котором вершины соответствуют прикладным модулям пользователя, а дуги определяют частичный порядок их выполнения, определяющего директивные интервалы выполнения прикладных модулей, возможность построения допустимого расписания выполнения прикладных модулей и само расписание, если оно существует, а также выполняющего некоторые другие вспомогательные функции построения инструмен-тальной САПР ВСРВ.

Блока генерации кода, формирующего на языке С и записывающего в текущий каталог исходные тексты получившейся программы, а также создающего ряд вспомогательных файлов для последующих определённых действий пользователя, компиляции и редактирования связей.

Управляющего монитора на основе многозадачной оболочки реального времени CTask-RT, обеспечивающего работу в реальном времени прикладной программы пользователя, сгенерированной на этапе предварительной обработки посредством САПР ВСРВ.

Предмет исследования.

В соответствии с поставленной целью предметом исследования является методология построения инструментальной САПР систем реального времени. Это включает в себя разработку формального языка для описания системы реального времени, построение алгоритмов нахождения допустимого расписания выполнения прикладных модулей для однопроцессорных и многопроцессорных вычислительных комплексов, генерацию кода соответствующей инструментальной САПР системы реального времени и управления в реальном времени.

Методологическую и теоретическую основу исследования составили методы разработки систем реального времени, теории расписаний, комбинаторной оптимизации и теории графов.

Научная новизна.

Научная новизна диссертационной работы заключается в построении инструментального программного комплекса, дающего новые качественные возможности при автоматизации построения систем реального времени с периодическим

поступлением входной информации, в том числе разработку алгоритмов построения расписаний работ для многопроцессорной вычислительной системы.

В процессе исследования автором было выполнено следующее:

– разработана и программно реализована инструментальная система САПР «СРВ-Конструктор», включающая язык реального времени, блоки синтаксического и семантического анализа, блок генерации сетевой модели и расписаний, блок генерации кода и управляющий монитор;

– разработаны и программно реализованы алгоритмы решения минимаксной задачи составления расписаний без прерываний с использованием различных правил предпочтения при выборе определения допустимого расписания;

– на основе многочисленных вычислительных экспериментов получены апостериорные оценки точности разработанных алгоритмов;

Практическая значимость, апробация и внедрение результатов работы.

Основные результаты исследования относятся к созданию нового инструментального программного комплекса САПР «СРВ-Конструктор» для однопроцессорных ПЭВМ и разработке ряда эффективных эвристических алгоритмов оптимизации планирования вычислений в системах реального времени для многопроцессорных вычислительных комплексов для дальнейшего развития системы САПР «СРВ-Конструктор».

Результаты данного исследования обсуждались и докладывались на:

– III научной школе "Автоматизация создания математического обеспечения и архитектуры систем реального времени" (Саратовский ф-л Института машиноведения РАН, Саратов, 1992);

– межд. конф. "Проблемы управления в чрезвычайных ситуациях" (М., ИПУ РАН, 1992);

– межд. конф. "Проблемы регионального и муниципального управления" (М.: РГГУ, 1999);

– IX и X межд. конф. "Проблемы управления безопасностью сложных систем" (М., ИПУ РАН, 2001 и 2002);

– III межд. конф. по исследованию операций (М., ВЦ РАН, 2001);

– науч. конф. "Математические модели сложных систем и междисциплинарные исследования" (М., ВЦ РАН, 2002);

– XLV и XLIX науч. конф. Московского физико-технического института (ГУ), (М.-Долгопрудный, 2002, 2006);

– II Всеросс. науч. конф. «Методы и средства обработки информации» (М.: МГУ, 2005);

– научных семинарах сектора проектирования систем реального времени Вычислительного центра им. А.А. Дородницына Российской Академии Наук;

– научных семинарах кафедры математических основ управления Московского физико-технического института (ГУ).

По итогам исследования опубликовано 18 работ, в том числе одна в издании, входящем в список ВАК. Список работ приведён в конце диссертации [68-86].

Данная работа может быть применена для повышения качества и эффективности проектирования и работы широкого класса систем, работающих в жёстком реальном времени и использующих периодически поступающую входную информацию.

Состав и структура работы.

Диссертация состоит из введения, семи глав, заключения, списка использованной литературы и двух приложений.

Во введении говориться о важности и актуальности построения вычислительных систем реального времени в целом и рассмотрена общая схема включения ЭВМ в контур контроля процессов, происходящих в некотором реальном физическом объекте, основные потоки информации и состав аппаратных средств типичной многоканальной системы сбора данных и управления процессами, общая концепция разработки ВСРВ для систем с периодическим поступлением входной информации.

Первая глава диссертации посвящается общему описанию САПР систем реального времени «СРВ-Конструктор», как основного проектируемого инструментального средства. Описывается состав и структура САПР ВСРВ в целом.

Со второй по пятую главах более подробно описаны наиболее важные блоки САПР «СРВ-КОНСТРУКТОР»: язык реального времени, используемый в блоке транслятора САПР ВСРВ, управляющий монитор, один из центральных блоков САПР ВСРВ – Генератор сетевых моделей и расписаний (ГСМР) и математиче-

ские алгоритмы, лежащие в его основе. Приводится также описание общей структуры программного комплекса «СРВ-Конструктор», порядка его сборки и генерации кодов.

Шестая глава посвящена постановке важной задачи планирования вычислений M заданий на N процессорах, связанной с дальнейшим развитием САПР «СРВ-Конструктор» для многопроцессорных вычислительных комплексов. Рассматриваются две постановки этой задачи: в первой процессоры идентичные, во второй могут отличаться производительностью. Приводится обзор существующих методов решения этой задачи.

В седьмой главе предлагается ряд эвристических алгоритмов распределения заданий по процессорам, которые планируется использовать в дальнейших версиях системы «СРВ-Конструктор», даётся оценка точности их вычислений. Приводятся результаты серии расчётов по данным алгоритмам и сравнительный анализ последних.

В Приложении 1 приведён пример применения САПР ВСРВ для решения в реальном времени задачи многомерной линейной регрессии.

В Приложении 2 приведены тексты программной реализации предлагаемых эвристических алгоритмов для многопроцессорного варианта САПР «СРВ-Конструктор».

В заключении сформулированы основные теоретические и практические результаты диссертации.

Глава 1. Система автоматизации программирования вычислительных систем реального времени. Общая схема, потоки информации, структура системы

1.1. Задачи, назначение и общая схема САПР систем реального времени «СРВ-Конструктор»

Вычислительные системы реального времени предназначаются для решения задач, которые с высокой степенью надёжности должны быть выполнены на заданном отрезке времени. Это требование вытекает из области их применения для контроля и управления в реальном времени процессами в химической промышленности, транспортных системах, системах производства энергии, в частности, атомных электростанциях, в нефте- и газодобывающих производствах, в автоматизированных производствах, мониторинге экономической и экологической обстановки и многих других областях человеческой деятельности.

В секторе проектирования систем реального времени ВЦ РАН с существенным участием автора разработана система автоматизации программирования вычислительных систем реального времени (САПР ВСРВ) для IBM PC. Система предназначена для автоматизации проектирования и генерации систем реального времени, осуществляющих обработку циклически поступающей информации в темпе поступления при жёстких временных ограничениях. Она позволяет быстро составлять необходимую пользователю ВСРВ из готовых прикладных модулей. Эта система является принципиально новой. От известных систем в данной области её отличает то, что, во-первых, она позволяет в короткие сроки в автоматическом режиме проектировать конкретную ВСРВ, описанную пользователем с помощью программы реального времени (РВ-программы), и, во-вторых, заранее, т.е. до обработки информации в реальном времени, проводить оптимизацию вычислений. При разработке этой системы был получен ряд вспомогательных результатов, которые также являются новыми и представляют отдельный теоретический и практический интерес. Например, были разработаны и реализованы наглядные языковые средства для описания циклической обработки информации, разработаны оригинальные алгоритмы построения допустимых расписаний в ВСРВ, динамического распределения многоуровневой памяти, некоторые алгоритмы решения

задачи синтеза многопроцессорных ВСРВ.

Существует два основных подхода к созданию САПР ВСРВ. Первый связан с расширением существующих языков новыми операторами, позволяющими пользователю описывать параллельные вычисления и их синхронизацию. При этом вся работа по разделению времени для однопроцессорных систем ложится на пользователя.

Нами был выбран другой, на наш взгляд более перспективный подход, при котором вся работа по разделению времени и оптимизации вычислений выполняется транслятором. Для реализации такого подхода необходимо предварительно построить математическую модель ВСРВ. С помощью этой модели могут быть решены такие задачи, как разбиение программ на параллельные процессы, составление допустимого расписания выполнения процессов, синхронизация вычислений, динамическое распределение памяти.

Для генерации прикладной ВСРВ пользователю необходимо иметь прикладные модули, написанные на языках программирования, например *Си, Фортран, Паскаль* или *Ассемблер*, и написать на специальном языке реального времени задание на обработку информации в реальном времени – РВ-программу. В этой программе пользователь задаёт порядок обработки прикладными модулями входных данных и отображения результатов счёта по отношению к периоду поступления кадров данных в систему. Если данный порядок обработки может быть соблюдён, система обеспечит реализацию заказанной обработки. В противном случае выдаётся сообщение о невозможности вести указанную обработку. В отличие от систем потоковой обработки данных предусмотрена возможность работы прикладных модулей с несколькими поколениями данных. Система автоматически обеспечивает хранение этих данных в специальных буферах нужное время. Предусмотрена также возможность быстрой реакции на поступление апериодической информации. Это может быть использовано, например, при возникновении внештатной ситуации с управляемым объектом, либо для изменения порядка работы программы реального времени оператором. Также предусмотрено выполнение прикладных модулей в фоновом режиме. Вся остальная работа по генерации прикладной ВСРВ будет выполнена автоматически с помощью разработанной САПР ВСРВ. При этом будут решены следующие проблемы:

1) проблема синхронизации параллельных процессов в ВСРВ, под которыми

понимаются как процессы функционирования периферийных устройств системы, являющиеся внешними по отношению к используемой вычислительной системе, так и вычислительные процессы, реализуемые выполнением РВ-программы;

2) проблема тупиков при распределении ресурсов вычислительной системы (процессоров, каналов, памяти, периферийных устройств, а также информационных ресурсов, под которыми понимаются используемые в режиме разделения вспомогательные программы и информационные массивы);

3) проблема завершения тех или иных вычислений к определённым моментам времени или к моментам определённых событий в системе, указанных пользователем в РВ-программе (соблюдение директивных сроков);

4) проблема сохранения и обновления необходимых наборов поколений данных, как поступающих в ВСРВ извне, так и являющихся результатами вычислений с помощью РВ-программы, для последующего использования в вычислениях;

5) проблема обнаружения в режиме реального времени ошибок и сбоев в работе датчиков, каналов передачи данных к ВС и внутри самой вычислительной системы, а также организации соответствующих реакций на обнаруженные ошибки и сбои.

Решение этих задач обеспечивает надёжность программного обеспечения ВСРВ. Отметим, что при использовании ЭВМ не в режиме реального времени эти проблемы либо вообще не возникают, либо решаются просто, например повторным запуском программ. Реализация разработанной САПР ВСРВ является результатом применения простого принципа: максимальная подготовка программы реального времени к выполнению до режима реального времени. Разработанная система наиболее эффективна при составлении программ для детерминированных ВСРВ с циклическим поступлением информации от контролируемого объекта. Детерминированность означает здесь следующее:

1) входная информация поступает на входные регистры системы с известным периодом T, т.е. в моменты времени 0 ,T, $2T$, ... ;

2) входная информация может быть объединена в кадры, имеющие постоянную часть (т.е. поступающую в каждый дискретный момент 0, T, $2T$,...) и несколько переменных частей, каждая из которых поступает в систему реже, с большим периодом, кратным T. Например, одна переменная часть кадра может

поступать в систему с периодом $2T$, т.е. в моменты 0, $2T$, $4T$, ..., другая – с периодом $5T$, т.е. в моменты времени 0, $5T$, $10T$,... Количество переменных частей кадра произвольно;

3) контролируемый процесс состоит из конечного числа детерминированных участков. Для каждого такого участка заранее известно, какие данные необходимо получать от объекта и передавать к нему, определены форматы входных кадров, а также известно, какую обработку необходимо производить, т.е. задан список программных модулей, осуществляющих нужные вычисления. Иными словами, для каждого участка контролируемого процесса известен *режим обработки*. Участки могут следовать друг за другом в произвольном порядке и сменять друг друга в моменты времени, определяемые как результатами контроля, так и выполненных вычислений;

4) для каждого программного модуля, выполняющего вычисления или некоторый обмен данными (например, ввод и/или вывод данных на заданное устройство), известна оценка времени исполнения этого модуля, а также потребности в других ресурсах системы, например, требуемый объём оперативной памяти, объём дисковой памяти, потребность в других ресурсах системы;

5) в процессе выполнения каждого режима обработки может быть вычислен момент смены текущего режима и однозначно определён следующий режим обработки;

6) задан начальный (стартовый) режим обработки (например, режим ожидания первого кадра информации от контролируемого объекта).

При разработке САПР ВСРВ для IBM PC были реализованы (1) язык реального времени; (2) транслятор с этого языка, включающий блоки синтаксического и семантического анализа, построения сетевой модели вычислений и нахождения допустимого расписания выполнения вычислений, автоматической генерации объектного кода программы реального времени; (3) управляющий монитор, контролирующий вычисления в режиме реального времени.

Остановимся кратко на описании языка реального времени и основных блоков САПР ВСРВ.

1.2. Язык реального времени

Синтаксис языка реального времени, опуская некоторые детали, может быть описан следующим образом.

Основным элементарным объектом языка является модуль – обычный для многих языков программирования оператор процедуры. Все процедуры, участвующие в образовании модулей, составляются заранее и помещаются в системную библиотеку процедур вместе с необходимыми спецификациями формальных параметров. Имеется один специальный тип модуля, введённый для облегчения составления РВ программ – это РВ модуль. Он по описанию и смыслу ничем не отличается от обычного модуля. Но кроме описания он ещё имеет тело, внутри которого указаны вызовы других, в том числе и РВ модулей. Для РВ модулей имеется ограничение – они не должны содержать рекурсивных вызовов. Фактические параметры любого модуля могут быть РВ-параметрами, константами, простыми переменными, идентификаторами с индексами и т.д. РВ-параметры имеют следующий вид:

<РВ-параметр> ⇒ (<имя РВ-параметра>; <поставщик>;
<поколение>)

Компонента <поставщик> однозначно указывает программный модуль, в котором вычисляется запрашиваемое модулем-потребителем значение параметра с данным именем. Как будет видно из дальнейшего описания языка, вызовы модулей могут повторяться в программе, а сами модули могут входить в более сложные объекты языка. Поэтому эта компонента, в свою очередь, состоит из нескольких полей. Если имя поставщика отсутствует, то считается, что этот РВ-параметр присутствует во входных данных.

Компонента <поколение> определяет момент времени, в который необходимо взять требуемое значение параметра. В этом разделе может быть указано время, кратное периодичности поступления информации от указанного поставщика данных. Системой будет передано потребителю последнее имеющееся на заданный момент времени значение этого параметра.

Другие виды фактических параметров здесь не обсуждаются, поскольку они аналогичны параметрам процедур известных языков программирования.

Из модулей может быть составлен следующий по иерархии сложности объект языка – цикл реального времени:

<цикл РВ> ⇒ <имя цикла РВ>, <период>, <фаза>,
<тело цикла РВ>

Компонента <период> задаёт интервал времени, через который будет повторяться выполнение модулей, указанных в теле цикла РВ. Задание периода аналогично заданию времени для РВ-параметра. Если в программе реального времени указано несколько источников данных или в качестве базового периода берётся период другого цикла РВ, то требуется явное указание его имени перед значением периода. Кроме этого период может быть задан в абсолютных величинах времени: секундах, миллисекундах. Параметр <фаза> указывает сдвиг начала работы РВ-цикла, относительно начала работы программы. Правила задания параметра <фаза> такие же, как и для <периода>. Фаза не может быть задана больше периода, указанного для данного РВ-цикла. Если фаза явно не указана, то она равна нулю. <Тело цикла РВ> – это список модулей, РВ-параметры которых могут поставляться из блоков входной информации, из модулей других циклов РВ и из модулей данного цикла.

Поименованная совокупность РВ циклов образует безусловное задание (простое задание).

<безусловное задание> ⇒ **smplpgm** <имя задания>;
head <список модулей> **end**;
tail <список модулей> **end**;
inherit <список безусловных заданий>;
<список циклов РВ> **end**

Безусловное задание может быть снабжено конструкциями предварительной и заключительной части. Они служат для проведения набора специфических действий перед началом работы и соответственно после конца работы простого задания. Здесь может проходить инициализация переменных, каналов связи, переключение режима работы датчиков и т.д.

Для облегчения программирования на языке реального времени, в новое простое задание допускается вставить несколько других простых заданий (конструкция **inherit**). В этом случае не придётся дублировать исходный текст ранее оп-

ределённого простого задания. Порядок выполнения предварительных частей простых заданий будет следующий: сначала выполняются все предварительные части включаемых простых заданий в том порядке, который определён в списке после ключевого слова **inherit**, после этого выполняется предварительная часть, указанная в новом простом задании. Порядок выполнения заключительных частей – обратный по отношению к порядку исполнения предварительных частей. РВ-циклы, указанные в телах включаемых простых заданий, добавляются к РВ-циклам, определённым в теле нового простого задания.

Основную часть простого задания составляют РВ-циклы. Все циклы реального времени, входящие в одно простое задание, должны рассматриваться как выполняющиеся параллельно во времени. Если модули некоторого цикла требуют на свой вход данные, поставляемые другим циклом, то необходимая задержка организуется системой.

В ходе обработки эксперимента может потребоваться изменить режим обработки. Такая возможность предусматривается в языке реального времени введением условного задания.

<условное задание> → **condpgm** <имя задания>;
cvector <вектор условий>;
switchtable <таблица переключений>
end;
<флаги и очереди>
<список простых заданий>;
end

В разделе <вектор условий> определяется список булевых переменных, от значения которых будет зависеть порядок исполнения программы реального времени. Здесь же указываются начальные значения компонент вектора. Компоненты вектора условий могут быть использованы в качестве входных и выходных параметров РВ-модулей. В разделе <таблица переключений> задаются правила переключения между простыми и условными заданиями, в зависимости от конкретных значений вектора условий. Таблица переключений представляется в виде списка значений вектора условий и имени простого или условного задания, на которое

требуется переключиться. Для упрощения описания таблицы переключения допускается использовать кроме конкретных значений компонент вектора условий ключевого слова **ignore**. Для таблицы переключений имеется два ограничения:

1) В таблице переключений обязана присутствовать строка, соответствующая начальному значению вектора условий.

2) Таблица переключений должна однозначно определять на какое задание должно происходить переключение. Если для текущего значения вектора условий нет соответствующей записи в таблице переключений, то перехода на другое задание не происходит и продолжается работа программы в старом режиме.

Наконец, РВ-программа представляет собой совокупность условных заданий, все таблицы условий которой не содержат ссылок на неописанные задания.

<программа РВ> ⇒ **rtpgm** <имя программы>;
<описание констант, модулей, переменных, входных данных>;
<условные задания> **end**;

Более подробно язык реального времени описан в главе 2.

1.3. Основные блоки транслятора

Блок синтаксического и семантического анализа выполняет следующие функции:

1. Осуществляет синтаксический и семантический анализ конструкций РВ-программы.
2. Выдаёт сообщения о наличии ошибок в РВ-программе.
3. Генерирует таблицы данных для работы последующих блоков.
4. Вычисляет размеры буферов обмена данными между программными модулями.

Блок генерации сетевой модели и расписаний выполняет следующие функции:

1. Строит математическую модель вычислений, выполняемых в реальном времени, в виде графа, в котором вершины соответствуют прикладным модулям пользователя, а дуги определяют частичный порядок их выполнения.

2. Определяет директивные интервалы выполнения прикладных модулей.

3. Определяет, существует ли допустимое расписание выполнения прикладных модулей, и строит его, если оно существует.

4. Определяет необходимое количество копий для каждого прикладного модуля.

5. Вычисляет размеры буферов для входных параметров прикладных модулей.

6. Назначает стеки прикладным модулям для работы с данными.

Блок генерации кода выполняет следующие функции:

1. Формирует на языке С и записывает в текущий каталог исходные тексты получившейся программы.

2. Создаёт ряд вспомогательных файлов для последующих определённых действий пользователя, компиляции и редактирования связей.

1.4. Управляющий монитор

Управляющий монитор обеспечивает работу в реальном времени прикладной программы пользователя, сгенерированной на этапе предварительной обработки посредством САПР ВСРВ.

Управляющий монитор реализован в виде многозадачной надстройки над операционной системой MS-DOS версии 3.30 и выше на основе многозадачной оболочки реального времени CTask-RT, прототипом к которой послужил пакет Т.Вагнера [33]. CTask-RT позволяет создавать программную среду, использующую многозадачные заготовки BIOS IBM PC/AT, обходящую барьеры нереентерабельности MS-DOS, и даёт возможность совместно работать с резидентными (TSR) программами и под управлением MS-Windows.

Управляющий монитор является составной частью исполняемого EXE-модуля РВ-программы. Его функциями являются:

– приём, хранение и обработка поступающих извне больших порций информации (кадров данных);

– реакция на внешнее апериодическое сообщение;

– исполнение команд, вводимых с клавиатуры консоли;

– переключение между условными и простыми заданиями в соответствии со значениями системного вектора условий;

– запуск процессов согласно директивным срокам и приоритетам;

– обмен данными между процессами;

– действия по окончании работы процесса.

Глава 2. Входной язык САПР вычислительных систем реального времени

Прежде чем приступить к описанию входного языка САПР "СРВ-Конструктор", ещё раз сформулируем те цели, которым она служит. Это поможет лучше понять особенности языка, назначение тех или иных его конструкций.

САПР "СРВ-Конструктор" разработана как инструментальное средство для создания систем реального времени, рассчитанных преимущественно на обработку циклически поступающей извне информации в темпе поступления при соблюдении временных ограничений. "СРВ-Конструктор" даёт пользователю возможность быстро скомпоновать необходимую ему ВСРВ из готовых «кирпичиков» – заранее написанных и отлаженных прикладных модулей. Входной язык служит для описания смысловой части разрабатываемой ВСРВ – пользователь должен указать с какой частотой необходимо активизировать тот или иной прикладной модуль, как модули обмениваются между собой данными, из чего состоят приходящие извне кадры и т.п. САПР же берёт на себя всю часть работы, связанную с технической реализацией замысла разработчика, т.е. всю системную часть.

Прикладные модули пишутся на универсальных языках программирования – C, PASCAL, FORTRAN и оформляются в виде функций (подпрограмм). Каждый прикладной модуль – это в полном смысле обычная последовательная подпрограмма, однократно выполняющая определённое, в некотором смысле элементарное действие.

Кратко остановимся теперь на основных понятиях входного языка «СРВ-Конструктора». Следует отметить, что язык позаимствовал много черт у входного языка САПР ВСРВ, реализованной на ЕС ЭВМ в ВЦ АН СССР в секторе проектирования систем реального времени [5].

Язык программирования, служащий для описания идущих в реальном масштабе времени вычислительных процессов, должен предоставлять возможность синхронизировать производимые вычисления с моментами поступления информации извне – приходами кадров данных. Этой цели во входном языке служит конструкция, названная РВ-циклом (loop). РВ-цикл задаёт период и фазу активизации входящих в него прикладных модулей. Период *T* задаётся в интервалах, равных частоте прихода кадров. Выполнение всех прикладных модулей должно

быть завершено к началу их следующей активизации – за время, равное периоду РВ-цикла. Таким образом, конечным директивным сроком для каждого входящего в РВ-цикл модуля является приход *T*-го с начала текущей активизации РВ-цикла кадра входных данных.

Входящие в РВ-циклы прикладные модули называются основными (foreground). Основные модули выполняются псевдопараллельно (в режиме разделения времени), но с учётом графа зависимости (по данным и пр.).

Совокупность РВ-циклов, определяющих некоторый заданный режим обработки входной информации, составляет тело простого задания (smplpgm) – ПЗ.

РВ-циклам ПЗ может предшествовать однократное выполнение при переключении на данное ПЗ предварительной части (head). По окончании ПЗ может однократно быть выполнена заключительная часть (tail).

Предварительная и заключительная части представляют собой последовательную активизацию некоторого числа прикладных модулей.

Одно или несколько ПЗ, соответствующих взаимозависимым режимам обработки входных данных, компонуются в условные задания (condpgm) – УЗ. Другой составляющей частью УЗ являются фоновые работы (background) – общие для всех ПЗ данного УЗ прикладные модули, выполняемые в фоновом режиме. У фоновых работ отсутствуют директивные сроки. Фоновый модуль может начаться в одном ПЗ и завершиться в другом ПЗ данного УЗ. Конструкцию фоновых работ удобно использовать, например, для организации «плохо» регламентируемого в отношении временных затрат вывода информации на внешние носители.

Переключения между УЗ и между ПЗ внутри УЗ осуществляется в соответствии с текущим значением вектора условий (ВУ) – cvector и таблицей переключений (switchtable). Помимо периодически поступающих кадров входных данных в ВСРВ, в некоторые заранее не известные моменты времени могут поступать апериодические сообщения. Предполагается, что эти сообщения имеют экстренный характер, т.е. требуют немедленной реакции системы. Внутри каждого УЗ пользователь может описать модуль реакции (handler) на данное событие, который будет обрабатывать сообщения непосредственно по их приходу. Модули реакции не должны требовать существенных временных издержек, чтобы их выполнение не привело к нарушению директивных сроков основных модулей.

2.1. Синтаксис

2.1.1. Разделителями в тексте РВ-программы являются пробелы, символы табуляции и перехода на новую строку. Разрешается использование как одного такого символа, так и любого их количества.

2.1.2. Комментарии заключаются в скобки (* и *). Комментарии разрешены всюду, где допустимы пробелы.

2.1.3. Псевдокомментарии служат для задания режима компиляции РВ-программы. Они заключаются в скобки (# и #). Псевдокомментарии должны размещаться на одной строке. Указания внутри псевдокомментария разделяются знаком "/".

(#LIST#) – включить режим распечатки полного текста исходной программы.

(#ERRORS#) – включить режим распечатки только строк с ошибками (задаётся по умолчанию).

2.1.4. Идентификаторы используются как имена константных величин, переменных, программных модулей, условных и простых заданий, РВ-программы. При написании идентификаторов допустимы латинские большие и маленькие буквы, цифры "0"-"9" и подчерк "_". Первый символ в идентификаторе должен быть буквой. Значащими в идентификаторе являются первые 12 символов.

2.1.5. Следующие слова являются зарезервированными во входном языке системы "СРВ-Конструктор": FALSE, TRUE, after, boolean, background, char, calc, const, condpgm, dbl, cvector, entry, end, file, extra, frame, flag, glob, from, head, handler, inout, in, loop, integer, others, modules, period, out, port, phase, queues, prty, rtpgm, real, smplpgm, size, tail, switchtable, type, timeslice.

2.2. Типы данных

Входной язык системы различает следующие типы данных: целые числа (integer, dbl integer), символы (char), булевские величины (boolean), числа с плавающей точкой (real, dbl real) и файлы (file).

2.2.1. Целые константы

При записи целых констант используются знаки "+", "-" и знак экспоненты

"@", цифры "0"-"9". Первой цифрой не должен быть "0". В реализации системы на IBM PC целая константа занимает 2 байта. Система счисления – десятичная.

Примеры: 10 –25
+1000 3@2 (* 300 *).

Примечание: соответствует типу int в языке C, INTEGER*2 в языке FORTRAN, integer в PASCAL.

2.2.3. Длинные целые константы

Длинная целая константа определяется ключевым словом dbl, стоящим после константы. На IBM PC длинная целая константа занимает 4 байта.

Примеры: 10 dbl –25 dbl +75@3 dbl (* 75000 *).

Примечание: соответствует типу long в языке C, INTEGER*4 в FORTRAN, longint в языке PASCAL.

2.2.3. Константы с плавающей точкой

Константа с плавающей точкой записывается при помощи знаков "+", "–", десятичной точки ".", знака экспоненты "@", цифр "0"–"9".

Отличительным признаком константы с плавающей точкой служит либо наличие десятичной точки, либо отрицательной экспоненты.

В реализации системы на IBM PC константа с плавающей точкой занимает 4 байта.

Примеры: 12.5 –135. 0.5@2 (* 50.*)
100@–1 (* 10.*).

Примечание: соответствует типу float языка C, REAL языка FORTRAN, real языка PASCAL.

2.2.4. Константы с плавающей точкой двойной точности

Константа с плавающей точкой двойной точности определяется ключевым словом dbl, стоящим после константы. В реализации системы на IBM PC занимает 8 байтов.

Примеры: 12.5 dbl   1.@-50 (* 1e-50*).

Примечание: соответствует типу double языка C, REAL*8 языка FORTRAN, double языка PASCAL.

2.2.5. Символьные константы

Символьная константа состоит из одного символа кода ASCII, заключённого в апострофы. На IBM PC занимает 1 байт.

Примеры: 'a' 'B' '+' NULL.

Примечание: соответствует типу char языка PASCAL.

2.2.6. Строковые константы

Строковая константа состоит из более чем одного кода ASCII, заключённого в апострофы. Занимает столько байт, столько символов в константе, плюс один байт – автоматически добавляемый конец строки (NULL). Обратный слэш "\" является признаком входящего в константу спецсимвола языка C.

Примеры: 'abcd' '8A8' 'New line \n'.

Примечание: соответствует массиву символов char языка C, packed char языка PASCAL.

2.2.7. Булевские константы

К булевским относятся две константы: TRUE и FALSE. На IBM PC занимают 1 байт.

Примечание: соответствуют типу char в C, LOGICAL*1 в FORTRAN, boolean в PASCAL.

2.3. Описания

Описания используются для объявления переменных и константных величин различных типов, портов ввода-вывода, а также прикладных модулей.

2.3.1. Константные величины

В начале РВ-программы пользователь может описать константные величины, которыми он может пользоваться во всей РВ-программе.

Описание константной величины выглядит следующим образом:
ид_константной_величины = константа

Тип константной величины определяется типом стоящей в правой части константы.

Примеры:
```
NMAX = 50; FIVE = 5; TWO = 2; (* целые *)
PI = 3.14;                     (* с плавающей точкой *)
TETA2=0.25@-3 dbl;  (* с плавающей точкой, двойная точность *)
c = 'c';              (* символьная *)
MESSAGE = 'NOT READY'; (* строковая*)
LOG1 = TRUE ;          (* булевская*)
```

Константная величина может быть использована в качестве фактического параметра прикладного модуля, если только соответствующий формальный параметр описан как имеющий ВВ-тип in (см. ниже).

2.3.2. Переменные

При описании переменных первое, что должен объявить программист – это скалярный тип переменной и размерность.

2.3.2.1. Скалярные типы

Допустимыми скалярными шкалами языка являются:

integer – целое

(int в C, INTEGER*2 в FORTRAN, integer в PASCAL), 2 байта на IBM PC;

real – с плавающей точкой (float в C, REAL*4 в FORTRAN, real в PASCAL), 4 байта на IBM PC;

dbl integer – целое двойной точности (long в C, INTEGER*4 в FORTRAN, longint в PASCAL), 4 байта на IBM PC;

dbl real – с плавающей точкой двойной точности (double в C, REAL*8 в FORTRAN, double в PASCAL), 8 байт на IBM PC;

char – символ (char в C, char в PASCAL), 1 байт на IBM PC;

boolean – булевская (unsigned char в C, LOGICAL*1 в FORTRAN, boolean в PASCAL).

2.3.2.2. Массивы

Если переменная является массивом, то необходимо объявить её размер. Размер указывается в квадратных скобках.

Примеры:

integer a[5];   (* одномерный массив целых чисел *)

real b[2][NMAX]; (* двумерный массив 2*NMAX чисел с плавающей точкой – в FORTRAN (NMAX,2) *)

Остальные характеристики переменных зависят от того, в каком разделе они объявлены.

2.3.2.3. ВВ-тип

Описанные в РВ-программе переменные могут использоваться в качестве фактических параметров при вызове прикладных модулей. Соответствующий формальный параметр имеет одну важную характеристику, именуемую ВВ-типом (ввода-вывода).

ВВ-тип может иметь одно из трёх значений: in, out и inout. При задании типа in модуль использует значение переменной, но не возвращает его обратно, даже если он изменит его. При задании типа out модуль вырабатывает значение переменной, не используя его на входе. При задании типа inout переменная исполь-

зуется модулем, а на выходе получает новое значение.

При описании переменных мы будем указывать, какие значения ВВ-типа допустимы для того или иного типа переменных.

Различаются следующие типы переменных: кадровые, глобальные и расчётные переменные, элементы вектора условий, флаги и очереди.

Кадровые и глобальные переменные описываются на уровне всей РВ-программы.

2.3.2.4. Кадровые переменные

Кадровыми переменными именуются данные, содержащиеся в периодически приходящих в ВСРВ от объекта наблюдения (управления) кадрах информации. Предполагается, что различные кадровые переменные могут приходить с различными частотами, т.е. размер и структура кадра могут периодически варьироваться.

Кадровые переменные перечисляются в разделе описаний frame, который следует непосредственно после раздела константных выражений. Те переменные, которые содержатся в каждом кадре, составляют его постоянную часть. Постоянная часть кадра всегда должна находиться в начале кадра. Если же переменная содержится в каждом T-ом кадре, то она относится к «мигающей» части кадра. Для каждой такой переменной пользователь должен указать период T и фазу F: $0 \le F < T$. Данное описание означает, что переменная будет содержаться в кадрах с номерами $F+k*T+1$ (k = 0, 1, 2 ...). Мигающая часть отделяется от постоянной спецсимволом "$". Эта часть является необязательной.

Порядок описания кадровых переменных играет определяющую роль. Именно он задаёт местоположение той или иной переменной в получаемом системой пуле данных – кадре. Разделителем в описаниях является запятая ",". Описания заключаются в квадратные скобки.

Приведём несколько примеров описания кадровых переменных.

Пример 1.

```
[real x, real y $ real z period = 3 phase = 1]
```

Кадры с номерами 1,3,4,6,7,9,... – содержат по 8 байт – переменные с пла-

вающей точкой *x* и *y*. Кадры с номерами 2,5,8,... – содержат по 12 байт, последние 4 байта в кадре – это переменная *z*.

Пример 2.

[dbl real A[5][TWO] $ real D period = 2 phase = 0, integer C[3] period=3 phase=0]

Кадры с номерами 1,7,13,... содержат по 90 байт: первые 80 занимает массив из 10 чисел с плавающей точкой двойной точности *A*, следующие 4 байта – переменная с плавающей точкой *B*, а последние 6 байт – массив из 3 целых чисел *C*.

Кадры с номерами 2,6,8,12,... содержат по 80 байт – массив *A*.

И, наконец, кадры с номерами 4,10,... – содержат по 86 байт – переменные *A* и *C*.

Пример 3.

[char GAMMA[12], boolean BETTA[8]]

В данном примере в кадре присутствует только постоянная часть.

Пример 4.

[real X $ real Y period=2 phase=0, real Z period=1 phase=0]

Приведённый пример демонстрирует, как нужно описывать кадр данных, если постоянно присутствующая переменная идёт в кадре вслед за мигающей, когда та присутствует в кадре. Здесь все чётные кадры имеют длину 8 (переменные *X* и *Z*). Во всех нечётных кадрах присутствуют все три переменные: первые 4 байта занимает кадровая переменных *X*, вторые 4 – *Y*, третьи 4 байта – переменная *Z*. Длина кадра в этом случае равна 12.

Постоянно присутствующая в кадре переменная *Z* формально отнесена к переменной части кадра.

Очень часто у пользователя бывает необходимость использовать одновременно несколько «поколений» кадровых переменных, то есть значений переменной, относящихся к разным кадрам. Язык предоставляет ему такую возможность.

Транслятор автоматически вычисляет размеры буферов для кадровых переменных, обеспечивающих хранение необходимого числа поколений одной и той же кадровой переменной. При использовании кадровой переменной в качестве фактического параметра необходимо указать, из какого поколения должно быть взято значение. Кадровые переменные можно использовать только в модулях РВ-циклов (основных модулях).

В начале работы ВСРВ может возникнуть такая ситуация, когда в системе ещё отсутствует запрашиваемый прикладным модулем экземпляр кадровой переменной (например, в начальный момент по приходу первого кадра в системе будет отсутствовать переменная, у которой $F > 1$). У пользователя есть возможность указать, какое значение монитор данных должен поставить прикладному модулю в этом случае. Это так называемое «начальное» значение – оно задаётся сразу после описания кадровой переменной. В случае, если переменная является массивом, начальные значения заключаются в фигурные скобки. Начальные значения могут быть указаны только для нескольких первых элементов массива.

Начальное значение ни в коем случае нельзя понимать, как те значения, которыми инициализируется кадровая переменная. Это то значение, которое поставляется при отсутствии необходимого прикладному модулю экземпляра переменной.

Пример 5.

[real X=5.1,integer j $ real Y[3] period=2 phase=1 = {2.2,1@-2}, real Y1 period=3 phase=0 = 3.0]

В настоящем примере переменная X имеет начальное значение 5.1, для переменной j начальное значение не определено. В массиве Y начальные значения определены только для первых двух элементов.

Прикладной модуль не может изменить значение кадровой переменной – он всегда работает с копией переменной, создаваемой монитором данных. По этой причине формальный параметр модуля, на место которого ставится идентификатор кадровой переменной, должен иметь ВВ-тип in.

2.3.2.5. Глобальные переменные

Глобальными называются переменные, доступные из любой исполняемой

конструкции входного языка «СРВ-Конструктора». Глобальные переменные присутствуют в единственном экземпляре, т.е. это глобальные переменные в обычном понимании. Описание каждой глобальной переменной может быть снабжено указанием её начального значения. Отведённый переменной участок памяти будет проинициализирован этим значением в начале работы РВ-программы. При отсутствии начального значения переменная инициализируется нулём.

Описание глобальных переменных должны быть приведены в разделе описаний glob, следующих за разделом frame.

Пример 1.

integer i,j=1, ind[10]={3,4,5};

Здесь описаны целые переменные *i, j* и целый массив из 10 элементов *ind*. Переменной *j* присвоено начальное значение 1. Первые три элемента массива *ind* проинициализированы значениями 3, 4, 5, остальные нулями.

Пример 2.

dbl real alfa[FIVE][TWO]={-3.5,2,1.5@-2,PI,6.515 dbl,1.5};

Массив переменных с плавающей точкой двойной точности *alfa* имеет размерность, например, 5*2. Элементу массива с индексами [0][0] присвоено значение –3.5, элементу с индексами [0][1] – значение 2.0, элементу с индексами [1][0] – значение 0.015, элементу с индексами [1][1] – значение 3.14 (при соответствующем значении переменной PI), элементу с индексами [2][0] – значение 6.515, а элементу с индексами [2][1] – значение 1.5. Остальные элементы массива инициализируются нулями.

Пример 3.

boolean l[]={TRUE, TRUE, FALSE};

Булевский массив *l* состоит из трёх элементов – если размер массива опущен, то он определяется по числу начальных значений.

Пример 4.

char C[80] = {'abc', 'd', 'e', NULL};

Символьный массив *C* из 80 элементов инициализирован строкой abcde (элемент *c*[5] – признак конца строки "\0").

Наряду с переменными обычных скалярных типов в разделе glob также могут быть описаны и файлы. При описании в файле в кавычках указывается внеш-

нее имя файла (и, возможно, путь доступа), и его спецификации. Открытие файла в начале работы РВ-программы и закрытие его в конце будут производиться автоматически. Разрешается задавать любые спецификации файлов, допускаемые в языке С: w, r, t, +, -.

Пример:

file fout = 'C:\\out.dat':'wt';

В настоящем примере описаны выходной тестовой файл out.dat, который будет заведён в корневой директории на диске *C*:.

Использующие файлы прикладные модули должны быть написаны на языке С.

В языке существуют несколько предописанных системных глобальных переменных. Значения этих переменных изменяет монитор данных сгенерированной ВСРВ.

Перечислим эти переменные:

char MSGAsPort[80];

Переменная MSGAsPort заполняется по приходе экстренного апериодического сообщения, в неё заносится текст пришедшего сообщения.

integer LnAsPort;

С приходом внешнего апериодического сообщения в переменную LnAsPort записывается числовой идентификатор источника сообщения.

dbl integer NMBFRM;

Значение переменной NMBFRM равно числу пришедших кадров с начала режима РВ.

Так как прикладные модули работают с глобальными переменными напрямую, без создания промежуточных копий, то для глобальных переменных допустим лишь ВВ-тип inout.

Далее речь пойдёт о переменных, определяемых на уровне условных или простых заданий. Таковыми являются элементы вектора условий, флаги и очереди.

2.3.2.6. Элементы вектора условий

Вектор условий (ВУ) описывается в каждом условном задании. Описание

должно идти сразу вслед за заголовком условного задания (УЗ). ВУ состоит из элементов вектора условий (ЭВУ), число которых не должно превосходить разрядности двойного машинного слова (32 на IBM PC). ВУ – это индикатор состояния системы реального времени: значение ВУ определяет текущий режим обработки кадров данных, т.е. текущее УЗ и текущее ПЗ внутри этого УЗ. Ограничение на количество ЭВУ диктуется необходимостью частой проверки значения ВУ, поскольку его изменение одним из прикладных модулей может привести к смене режима обработки. Описание ВУ начинается ключевым словом cvector, вслед за которым в квадратных скобках идут описания ЭВУ: идентификаторы ЭВУ и значения ЭВУ (при входе в данное УЗ). В качестве разделителя в описаниях ЭВУ используется запятая.

Пример:

cvector [l1=TRUE, l2=TRUE, elem=FALSE];

Вектор условий состоит здесь из трёх ЭВУ: l1, l2 и elem. При инициализации УЗ ЭВУ принимают, соответственно, значения TRUE, TRUE, FALSE.

Значение ВУ хранится в упакованном виде (двойное слово), однако каждый ЭВУ считается переменной типа boolean: монитор данных запроса ЭВУ автоматически производит необходимые распаковку/ упаковку.

В целях повышения надёжности программирования в каждом ПЗ требуется определить подвектор ВУ простого задания. Только ЭВУ, перечисленные в подвекторе, могут использоваться в конструкциях данного ПЗ.

Пример:

cvector elem,l1;

В простом задании разрешается использование элементов вектора условий l1 и elem. Для ЭВУ допустимы все возможные ВВ-типы: in, out и inout. При ВВ-типе in значение ЭВУ будет считано из ВУ, однако никакие изменения считанного значения не приведут к изменению ВУ.

При ВВ-типе inout значение ЭВУ будет считываться из ВУ перед началом работы прикладного модуля, а после его окончания в ВУ будет занесено новое значение ЭВУ, то, которое получилось в результате работы модуля.

При ВВ-типе out прикладной модуль вырабатывает значение ЭВУ, которое записывается в ВУ.

2.3.2.7. Флаги

Флаги являются вспомогательными булевскими переменными, предназначенными для организации фоновых работ. Флаги описываются на уровне УЗ в разделе flags.

Описание каждого флага состоит из идентификатора флага и начального значения флага (значения флага при инициализации УЗ).

Пример: fl1 = TRUE; qqf = FALSE;

Выше описаны флаги fl1 и qqf, принимающие в начале работы УЗ значения TRUE и FALSE соответственно. Допустимые для флагов ВВ-типы: in, out, inout.

2.3.2.8. Очереди

Очереди описываются на уровне УЗ в разделе описаний queues и служат преимущественно для организации интерфейса между основными модулями (выполняемыми в соответствии с расписанием) и фоновыми работами. При описании очереди необходимо задать целый ряд параметров: скалярный тип и размерность, тип очереди (параметр tp, принимающий значения LIFO и FIFO), максимальный размер (параметр size, задающий максимальное число хранимых экземпляров переменной – очереди), идентификатор флага очереди (параметр flag). Значение флага очереди равно TRUE, если очередь непуста, и FALSE, если очередь пуста. Разным очередям поэтому должны быть приписаны разные флаги. Хотя в очереди может одновременно содержаться несколько экземпляров данных, доступным в каждый момент является только один из них, либо самый «старый» (для очереди FIFO), либо самый «свежий» (для LIFO).

При переполнении очереди затирается самый старый элемент.

Пример:

integer qq size = 5, tp = FIFO, flag = fl1 ;

real tele[2][FIVE] size = NMAX, tp = LIFO, flag = qqf;

Очередь qq организована по принципу FIFO – «первый вошёл – первый вышел». В очереди может содержаться не более 5 экземпляров данных типа integer.

Очередь tele содержит массивы чисел с плавающей точкой размерности 2 * FIVE. Очередь организована по принципу LIFO – «последний вошёл – первый вышел» и может содержать не более NMAX экземпляров массива одновременно.

Допустимые ВВ-типы: in, out, inout. При ВВ-типе in из очереди считывается один элемент (если очередь непуста), при ВВ-типе out в очередь записывается один элемент. При ВВ-типе inout из очереди считывается один элемент перед началом работы модуля, а после окончания в очередь записывается новый элемент.

2.3.2.9. Расчётные переменные

Расчётные переменные определяются на уровне ПЗ и служат для организации интерфейса между работающими в соответствии с расписанием основными модулями. Описываются расчётные переменные стандартным образом: разделителем служит точка с запятой.

Пример:
real SUM;

Описана переменная с плавающей точкой SUM.

Аналогично случаю с кадровыми переменными, разработчику ВСРВ предоставляется возможность работать с несколькими поколениями одной и той же расчётной переменной, т.е. одно единственное описание в большинстве случаев порождает существование в системе одновременно нескольких экземпляров переменной. Количество подлежащих хранению экземпляров рассчитывается транслятором автоматически, исходя из использования переменной в качестве фактического параметра основных модулей.

Так же, как и в случае с кадровыми переменными, в начале работы ПЗ требуемый экземпляр расчётной переменной (с глубиной, большей нуля) может быть ещё не выработан – простое задание только началось, и предыдущей активизации прикладного модуля-поставщика расчётной переменной ещё просто не было. На данный случай можно предусмотреть задание «начального» значения расчётной переменной – это значение поставляется при отсутствии в системе запрашиваемого экземпляра переменной.

Пример:
real SUM1 = 0.0;

Так же, как и кадровые переменные, расчётные могут быть использованы в качестве фактического параметра основных модулей РВ-циклов.

Допустимые ВВ-типы: in, out, inout. При значении ВВ-типа in из буфера

считывается заказанный экземпляр (или начальное значение). При ВВ-типе out вычисленное значение заносится в буфер переменной на место самого старого содержащегося там экземпляра. При ВВ-типе inout перед началом работы модуля из буфера считывается один экземпляр, а после окончания в буфер записывается новый.

2.3.3. Источники поступления информации в ВСРВ

В САПР "СРВ-Конструктор" заложено предположение, что поступающие от объекта наблюдения (управления) данные могут иметь двоякую природу. Во-первых, это периодически приходящие кадры данных, содержащие, например, показания периодически опрашиваемых датчиков. Во-вторых, это может быть информация экстренного характера, появляющаяся в заранее не регламентируемые сроки и требующая немедленной реакции системы.

Порт, через который в прикладную ВСРВ периодически поступают кадры данных, задаётся в начале раздела описаний frame при помощи ключевого слова port.

Пример:

port = COM1;

Источники поступления экстренных апериодических сообщений задаются в разделе описаний extra, следующем за разделом glob. В качестве подобного источника может быть указана и клавиатура консоли. Числовой идентификатор источника заключается в круглые скобки и следует за символическим идентификатором. Числовой идентификатор может быть использован модулем реакции для определения источника поступившего сообщения.

Пример:

extra COM2(1), KBD(2);

В данном примере источниками экстренных сообщений являются последовательный порт COM2 и клавиатура KBD. COM2 имеет числовой идентификатор 1, а клавиатура – 2.

Следует отметить, что экстренные сообщения не должны появляться слишком часто, чтобы не нарушить расписание.

В качестве источника экстренных сообщений не возбраняется задание и

порта поступления кадров – в этом случае необходимо предусмотреть возможность системе различать эти два типа информации по управляющим символам.

2.3.4. Кадр входных данных

Мы уже разбирали выше описания кадровых переменных (3.2.3) и порта поступления кадров (3.3). Определим, как должно выглядеть описание кадра входных данных в целом.

Описание начинается ключевым словом frame и состоит из трёх частей. Первая часть – задание порта поступления кадров, вторая – задание периода поступления кадров (ключевое слово timeslice), третья часть – описание структуры кадра (кадровых переменных).

Пример 1.

```
frame
  port = COM1;
  timeslice = 50 ms;
  [real h[FIVE], real Z1]
end;
```

В примере 1 кадры данных поступают через порт COM1 каждые 50 миллисекунд. Каждый кадр состоит из массива h из FIVE чисел с плавающей точкой и скалярной переменной с плавающей точкой Z1.

Пример 2.

```
frame
  port = COM1;
  timeslice = 25 000 mcs;
  [dbl real rd=1.0 $ real r period=5 phase=2]
end;
```

Здесь кадры приходят через COM1 каждые 25 мс. В каждом кадре содержится переменная с плавающей точкой двойной точности *rd*. В кадрах с номерами 3+5*i, i = 0, 1, 2,..., содержится также переменная с плавающей точкой *r*. "Начальное" значение переменной *rd* равно 1.0.

Подчеркнём, что именно период поступления кадров задаёт абсолютное значение темпа обработки входных данных. Период поступления кадров timeslice

– это как раз тот самый выраженный в абсолютных единицах квант времени, относительно которого будут задаваться периоды выполнения модулей в РВ-циклах.

2.3.5. Прикладные модули

Каждый используемый в РВ-программе прикладной модуль должен быть снабжён следующими описаниями: язык программирования, на котором написан данный модуль (C, FORTRAN, PASCAL, ASSEMBLER), верхняя оценка времени выполнения модуля, описание всех его формальных параметров.

Описание выглядит следующим образом:

ид_прикладного_модуля ((описание_параметра,)) : время, язык;

Язык программирования задаётся при помощи одного из следующих обозначений: C, PAS, FOR, ASM. Указание языка программирования необходимо для правильного оформления передачи параметров модулю.

Верхняя оценка времени выполнения задаётся при помощи целой константы и идентификатора единицы измерений: ms – миллисекунды, mcs – микросекунды.

Прикладные модули описываются на уровне РВ-программы в разделе modules.

Пример:

F():3 ms, FOR; (* модуль без параметров *)

Модуль F написан на языке FORTRAN и выполняется не более 3[x] миллисекунд.

2.3.5.1. Описание формальных параметров

Описание каждого параметра прикладного модуля должно содержать следующую информацию: скалярный тип, размер, ВВ-тип. Все этих характеристики мы уже обсуждали выше, когда речь шла о переменных. Отметим лишь, что размер формального параметра по последнему индексу можно не фиксировать, в этом случае последние квадратные скобки в описании нужно оставлять пустыми.

Если модуль написан на языке C или на языке PASCAL, то описание скалярных формальных параметров может снабжаться звёздочкой "*". Как известно, в языке FORTRAN передача параметров в подпрограмму осуществляется по ссылке, в то время как в C и PASCALе передача может осуществляться как по ссылке,

так и по значению. Добавление звёздочки означает передачу по ссылке, отсутствие звёздочки – передачу по значению (речь идёт только о языках С и PASCAL!). Если параметр передаётся по значению, то ВВ-тип соответствующего формального параметра должен иметь значение in.

Приведём примеры «шапок» модулей, написанных на различных языках программирования, и соответствующие им описание на входном языке "СРВ-Конструктора".

Язык «С»: mod1(int, long[5], char[], int*, float[FIVE],FILE *);

"СРВ-Конструктор": mod1 (in integer, inout dbl integer[5], inout char[], inout integer *, inout real[FIVE], out file*): 10 ms, C;

Везде вместо inout можно было бы указать ВВ-типы in или out, в зависимости от того, какие действия производит над переменными модуль mod 1. Для файла ВВ-тип определяется тем, с какими параметрами он открывается.

Язык FORTRAN:

```
SUBROUTINE mod2(A,B,C,D,E)
INTEGER*2 A
INTEGER*2 B(10,5)
REAL*4 C
LOGICAL*1 D
REAL*8 E(4,3)
```

"СРВ-Конструктор":

mod2(inout integer, inout integer[5][10], inout real, inout boolean, inout dbl real[3][4]): 4 ms, FOR;

Язык PASCAL:

procedure mod3(A:integer; var B:real; var C:boolean; D:array [1..5,3..4] of integer);

"СРВ-Конструктор":

mod3(in integer, inout real, inout boolean, inout integer[5][2]);

Напомним в заключение следующие факты: в языке FORTRAN по сравнению с языками С и PASCAL иная индексация *n*-мерных массивов (n>1). Нумерация элементов массива в FORTRANе начинается обычно с 1, в С – с 0, в PASCALе

– с указанной в описании нижней границы.

2.3.6. Таблица переключений

Так как в РВ-программе пользователь обычно описывает несколько режимов обработки входных данных, то соответственно он должен определить условия смены одного режима другим. Смена режима происходит при изменении значения вектора условий. В таблице переключений, присутствующей в каждом УЗ, перечисляются значения ВУ, вызывающие переключение на другое ПЗ в рамках данного УЗ или на другое УЗ. Если текущее значение ВУ не совпадает ни с одним из перечисленных в таблице значений, режим обработки не меняется – продолжается выполнение текущего ПЗ. Напомним, что значение ВУ может измениться только после окончания выполнения прикладного модуля, использующего хотя бы один ЭВУ в качестве фактического параметра, и при этом соответствующее значение ВВ-типа есть либо out, либо inout. Если модуль выдаёт на выходе новые значения сразу нескольких ЭВУ, то проверка на предмет переключения происходит только после всех изменений ВУ.

В таблице переключений обязательно должна присутствовать строка, соответствующая начальному значению ВУ УЗ. Эта строка задаёт, какой режим обработки (простое задание) будет выбран при переключении на данное УЗ.

Описание таблицы переключений начинается ключевым словом switchtable. Сама таблица может состоять из одной или нескольких строк. В каждой строке указывается значение ВУ и соответствующий этому значению идентификатор УЗ или ПЗ. Строка заключается в квадратные скобки.

При задании значения ВУ разрешается использование имён ЭВУ, а также ключевого слова others, при помощи которого указываются значения недостающих ЭВУ. Идентификатор УЗ (ПЗ) отделяется комбинацией символов "->". Одному и тому же имени УЗ (ПЗ) могут соответствовать несколько значений ВУ.

Пример:

Предположим, что ВУ УЗ описан как

cvector [l1=FALSE, l2=FALSE, l3=FALSE;]

Тогда таблица переключений

```
switchtable
  [l1==FALSE,l2==FALSE,l3==FALSE->A1]
```

```
[l1==TRUE,l2==TRUE,l3==TRUE->A2]
[l2==TRUE,others==FALSE->A2]
end;
```

задаёт следующие переключения: на A1, если все ЭВУ равны FALSE (A1 обязательно должен быть идентификатором ПЗ данного УЗ, так как он соответствует начальному значению ВУ); на A2, если значение ВУ равно TRUE, TRUE, TRUE или TRUE, FALSE, FALSE. A2 может быть как именем ПЗ данного УЗ, так и именем другого УЗ.

Иные значения ВУ, кроме перечисленных, не вызывают переключения на другой режим.

2.4. Исполняемые конструкции

Входной язык "СРВ-Конструктора" предназначен для сборки прикладной ВСРВ из заранее запрограммированных и оттранслированных модулей. По этой причине все исполняемые конструкции языка имеют в своей основе оператор вызова прикладного модуля. Синтаксис этого оператора обычен:

оператор_вызова_модуля ::=_ид_модуля ((факт_параметр/,))

Смысл оператора варьируется в зависимости от конструкции, в которой он употреблён.

"СРВ-Конструктор" располагает пятью конструкциями для организации работ, ведущихся в реальном масштабе времени. Это РВ-циклы для работ с жёсткими временными ограничениями, фоновые работы и модули реакции на экстренные апериодические сообщения, а также последовательно исполняемые начальные и заключительные части ПЗ.

2.4.1. РВ-циклы

Конструкция РВ-цикла является базовой для входного языка "СРВ-Конструктор" как языка описания систем жёсткого реального времени, рассчитанных на обработку периодически поступающих данных.

Каждый режим обработки входных данных в реальном масштабе времени – ПЗ – характеризуется периодически повторяемым комплексом завязанных по данным обязательных вычислительных работ. Каждая работа выполняется по-

средством вызова некоего прикладного модуля с определённым набором фактических параметров (основного модуля). Периоды выполнения могут быть не одинаковыми для всех работ, но все они кратны периоду прихода кадров. Работа может начаться не ранее прихода определённого кадра и не ранее окончания других работ, с которыми она связана отношениями предшествования (например, зависит от них по данным), если такие работы есть. Работа должна завершиться к тому моменту, когда возникнет потребность выполнить её ещё раз.

Конструкции РВ-циклов позволяют достаточно компактно задавать весь комплекс работ, подлежащих жёсткому планированию.

```
РВ-цикл ::=
loop   period = период   phase = фаза
   (оператор/;)
end;
период ::= целое | ид_константной_величины
фаза ::= целое | ид_константной_величины
```

РВ-цикл задаёт период и фазу выполнения входящих в него операторов (фаза < период), а также директивные сроки их выполнения.

Так, операторы, входящие в РВ-цикл

```
loop period = 3  phase = 2
   ...
end;
```

активизируются (т.е. переводятся в состояние, когда они могут начать выполняться, как только завершатся предшественники) по приходу входных кадров с номерами INF+2+3*i, i=0,1,2,... и должны завершиться соответственно к приходу кадров INF+2+3*(i+1), где INF – номер первого кадра с начала ПЗ.

Все операторы, входящие в РВ-циклы одного ПЗ, считаются выполняемыми параллельно (псевдопараллельно при одном процессоре), если между ними нет отношения предшествования.

```
оператор ::= метка : оператор_вызова_модуля
   [after (предшественник/,) ]
метка ::= ид
предшественник ::= метка . ид_модуля
```

В качестве предшественников можно задавать любые модули, встречаю-

щиеся в данном РВ-цикле или в других РВ-циклах данного ПЗ. Метки применяются обычно в тех случаях, когда один и тот же прикладной модуль встречается в ПЗ несколько раз с разными фактическими параметрами.

```
факт_параметр ::= [ модификатор_ВВ-типа ]
 константа |
 ид_константной_величины | ид_глобальной_переменной |
 ид_ЭВУ | ид_очереди | ид_флага |
 ид_кадровой_переменной .* [ -глубина ] |
 ид_расчётной переменной .* [ -глубина ]
[ from предшественник ]
модификатор_ВВ-типа ::= ( ВВ-тип )
ВВ-тип ::= in | out | inout
глубина ::= целое
```

При помощи модификатора ВВ-типа можно изменить значение ВВ-типа формального параметра модуля при данном обращении к этому модулю – формальный параметр считается имеющим указанный в модификаторе ВВ-тип.

При использовании кадровых переменных и расчётных переменных при ВВ-типе параметра *in* или *inout* программист должен указать, какое «поколение» данных его интересует. Идентификатор поколения отделяется от идентификатора переменной звёздочкой '*'. Указание *X*.* означает, что кадровая переменная *X* берётся из кадра, на котором активизируется основной модуль в соответствии с конструкцией РВ-цикла.

Так, если модуль GETX описан как

GETX(in real):1 ms,C;

и вызывается в РВ-цикле

```
loop period = 5 phase = 0
 GETX (X.*)
end;
```

а переменная *X* относится к постоянной части кадра, то значение *X* будет браться из кадров с номерами INF + 5*i, i = 0, 1, 2, ..., вне зависимости от того, когда реально в соответствии с расписанием будет выполняться GETX.

Предположим теперь, что обращение к GETX выглядит следующим обра-

зом:

GETX(X.*-2)

Данная запись будет означать, что модуль берёт значения *X* из кадров с номерами INF + 5*i - 2. Если РВ-цикл находится в ПЗ, с которого РВ-программа начинает работу, то при первой активизации GETX запрашивается кадр с номером -1 (INF=1, i=0), поэтому модулю будет поставлено «начальное» значение *X*.

Предположим теперь, что переменная *Y1* относится к переменной части кадра и вырабатывается с периодом 3 и фазой 0.

Пусть *Y1* запрашивается в РВ-цикле

```
loop period = 2 phase = 0
  GETX (Y1.*-1)
end;
```

Модуль GETX запрашивает переменную из кадров с номерами INF+2*i-1, однако она присутствует не во всех кадрах, так как *Y1* вырабатывается с периодом 3. Монитор данных будет действовать следующим образом: он ищет кадр с наибольшим номером, не превосходящим номер запрашиваемого кадра, и содержащий переменную *Y1*, и поставляет модулю значение из этого кадра.

Предположим, что в нашем примере *INF* =1. Тогда запросы будут адресоваться к кадрам с номерами 0,2,4,6,8,... , а переменная *Y1* будет присутствовать в кадрах с номерами 1, 4, 7,... Соответственно монитор данных при первых пяти активизациях GETX поставит такие значения переменной *Y1*: 1) "начальное" значение Y1, 2) Y1 из кадра 1, 3) *Y1* из кадра 4, 4) *Y1* из кадра 4, 5) *Y1* из кадра 7.

Отметим, что значение из кадра 3 в нашем примере будет поставлено два раза подряд – это связано с тем, что запросы переменных идут чаще, чем вырабатываются значения переменной.

При использовании расчётных переменных задание «поколения» данных требуется только в том случае, если соответствующий формальный параметр имеет ВВ-тип *in* или *inout*. Однако сначала надо определить, как монитор данных идентифицирует поколение расчётной переменной.

Для идентификации поколения расчётной переменной важны две характеристики – уникальное имя поставщика (модуля, который выработал значение переменной – одну и ту же переменную могут вырабатывать несколько модулей) и временная привязка. Уникальное имя поставщика состоит из идентификатора

прикладного модуля и, возможно, его метки. Вызовы одного и того же модуля более одного раза в рамках одного ПЗ по этой причине следует снабжать метками. Вопрос с временной привязкой решается следующим образом: вырабатываемый экземпляр переменной приписывается к тому кадру данных (как временной метке), на котором активизируется модуль-поставщик. Поясним сказанное на примере.

Пример 1.

Предположим, что в РВ-программе описаны целая константа *K* и кадровая переменная *C* – массив из *K* чисел с плавающей точкой.

Предположим также, что модуль *ASUM*, суммирующий массив с некими весовыми коэффициентами, описан как

```
 ASUM(in integer, in real[K] ,out real*):1 ms, C;
```

а в одном из ПЗ встречается РВ-цикл

```
 loop period =2 phase = 0
    ASUM(C.* , SUM1);
 end;
```

Вырабатываемые модулем *ASUM* на каждой активизации экземпляры переменной *SUM1* – суммы значений элементов текущего экземпляра массива *C* – помечаются как созданные в моменты INF+2*i, i=0,1,2,... вне зависимости от того, когда именно на протяжении двух временных единиц отрабатывает *ASUM*.

Введя таким образом идентификацию поколений значений расчётной переменной, можно легко указать, какое поколение интересует нас в том или ином случае. Модуль-поставщик задаётся при помощи ключевого слова from, а временная характеристика определяется точно так же, как для кадровых переменных.

Пример 2.

Допустим, что кроме модуля *ASUM* мы располагаем также модулем *ADD*, добавляющим к общей сумме (второй параметр) новую составляющую (первый параметр).

```
ADD(in real, inout real*):1 ms, C;
```

Расчётные переменные SUM и SUM1 описаны как

```
 calc
```

```
  real SUM1;
  real SUM=0.0;
end;
```

Пусть у нас стоит задача вести непрерывное суммирование кадровой переменной *C*. Выбранный нами способ решения этой задачи будет, конечно, весьма экстравагантен, но зато он позволяет прокомментировать использование расчётных переменных

```
loop period = 2 phase = 1
  L1 : ASUM(C.*-1,SUM1);
  L2 : ASUM(C.*,SUM1);
end;
loop period = 4 phase = 3
  M1 : ADD(SUM1.* from L1.ASUM,  SUM.*-1 from M4.ADD);
  M2 : ADD(SUM1.* from L2.ASUM,  SUM.* from M1.ADD);
  M3 : ADD(SUM1.*-2 from L1.ASUM,SUM.* from M2.ADD);
  M4 : ADD(SUM1.*-2 from L2.ASUM,SUM.* from M3.ADD);
end;
```

В первом РВ-цикле осуществляется суммирование массива *C* в переменную *SUM1*, первый раз сумма вычисляется для экземпляра, взятого из предыдущего кадра, второй раз – из текущего. Выполнение модулей L1.ASUM и L2.ASUM может вестись параллельно.

В результате мы получим по два(!) значения расчётной переменной *SUM*, относящихся к кадрам с номерами INF+2*i+1, i = 0,1,2,...

Во втором РВ-цикле, работающем вдвое реже первого, мы добавляем полученные частичные суммы *SUM1* к общей сумме *SUM*. Из значений *SUM1* берутся те, которые относятся к кадрам INF+4*i+3 и INF+4*i+3-2. Все четыре вызова *ADD* связаны отношением предшествования по переменной *SUM*, поэтому они будут выполняться последовательно. При первой активизации M1.ADD на вход подаётся «начальное» значение SUM – 0.0.

2.4.2. Предварительная и заключительная части простых заданий

Режим обработки периодической информации может потребовать одно-

кратного выполнения некоторых действий – при инициализации режима и по его завершению. Например, при выдаче результатов счёта в окно требуется высветить пустое окно в начале режима и затереть его в конце.

Каждое ПЗ РВ-программы может быть снабжено конструкциями предварительной и заключительной частей. Как предварительная, так и заключительная части – это обычные последовательные части программы, не требующие больших вычислительных затрат; суммарное время выполнения всех входящих в такую часть модулей вообще говоря не должно превышать периода прихода кадров – в противном случае при смене режимов будут пропущены один или несколько кадров входной информации.

Предварительная часть ПЗ описывается в разделе head

```
head
  (оператор_вызова_модуля/,)
end;
```

а заключительная – в разделе tail

```
tail
  (оператор_вызова_модуля/,)
end;
```

В качестве фактических параметров операторов вызова разрешается использование констант, константных величин, глобальных переменных, очередей, флагов, ЭВУ, глобальных переменных.

```
факт_параметр ::=
  константа | ид_константной_величины |
  ид_глобальной_переменной |
ид_ЭВУ | ид_очереди | ид_флага
```

Пример:

```
  tail
  close_wnd(1);
  cls_scr();
end;
```

Здесь описана заключительная часть ПЗ, содержащая вызовы двух модулей:

close_wnd, имеющего один параметр и модуля без параметров

cls_scr.

2.4.3. Фоновые работы

Помимо жёстко планируемых по директивным срокам работ в прикладной ВСРВ может иметься потребность в выполнении некоторых операций в фоновом режиме, т.е. в промежутках времени, когда нет готовых к выполнению основных модулей РВ-циклов. Предполагается, что это могут быть плохо регламентируемые операции ввода/вывода информации на внешние носители, а также расчёты, требующие сравнительно больших затрат машинного времени. В связи с этим фоновые работы определяются на уровне УЗ, что позволяет фоновым модулям продолжать выполняться даже при смене ПЗ внутри УЗ. Фоновый модуль готов к исполнению, если выставляется в единицу флаг фонового модуля.

Предполагается, что фоновый модуль может быть привязан к некоторой очереди (очереди, напомним, также описываются на уровне УЗ). Для этого у них должен быть один и тот же флаг, а сам модуль должен потреблять данные из очереди – соответствующий формальный параметр должен иметь ВВ-тип *in*. В этом случае готовность модуля к выполнению будет зависеть от того, пуста или непуста очередь. Очередь здесь может играть роль своеобразного мостика между основными модулями РВ-циклов и фоновым модулем; основные модули могут сбрасывать в очередь данные, которые будут затем востребованы фоновым в некоторый, заранее не известный момент времени.

Раздел фоновых работ начинается с ключевого слова background.

```
фоновые_работы ::=
 background
 (оператор/;)
 end;
оператор ::=
  оператор_вызова_модуля, prty = приоритет, flag = ид_флага
приоритет ::= целое
```

Приоритеты фоновых модулей задаются в диапазоне от 1 до 253. Более высокоприоритетный модуль имеет предпочтение при конкуренции за процессор,–

при равных приоритетах организуется пулинг.

В качестве фактических параметров операторов вызова можно указывать константы, константные величины, глобальные переменные, ЭВУ, флаги и очереди.

Использование кадровых переменных не разрешается из-за нерегламентируемости времени выполнения фонового модуля, так как иначе возникают сложности с идентификацией поколений кадровых переменных.

```
факт_параметр ::= модификатор_ВВ-типа
константа | ид_константной_величины |
ид_глобальной_переменной | ид_ЭВУ | ид_очереди | ид_флага
```

Во избежание возможных излишних переусложнений логики работы программы рекомендуется позаботиться о том, чтобы фоновые модули не осуществляли запись в очереди, по крайней мере в те, из которых читают данные другие фоновые модули. Для этого нужно, чтобы соответствующие формальные параметры прикладных модулей имели ВВ-тип *in*, либо переопределить ВВ-тип при помощи модификатора ВВ-типа (*in*).

Пример:

Пусть в РВ-программе описан прикладной модуль WRFILE

```
WRFILE (out file*, inout integer*) : 5 ms, C;
```

записывающий в файл целое число. Пусть также в разделе glob описан файл

```
file fout='out.txt':'wt';
```

а в одном из УЗ очередь

```
integer qq, size = 10, tp = FIFO, flag = fl;
```

Задав конструкцию

```
background
  WRFILE(fout, (in) qq), prty = 7, flag = fl;
end;
```

мы будем выводить в файл элементы очереди *qq*. Флаг фонового модуля *fl* тот же, что и у очереди *qq*. Модификатор ВВ-типа перед именем очереди нужен по той причине, что в описании параметра типа integer в прикладном модуле WRFILE был задан тип inout. Отсутствие модификатора (*in*) привело бы к своеобразному

зацикливанию WRFILE – очередь *qq* никогда не была бы пуста – каждый раз из очереди считывался бы один элемент, но каждый раз записывался бы и новый. Таких проблем не было бы, если модуль *wrfile* был описан правильно, исходя из его назначения:

WRFILE(out file*, in integer*) : 5 ms, C;

2.4.4. Модуль реакции

Данные двух типов определяют функционирование ВСРВ, проектируемой при помощи "СРВ-Конструктора": периодические кадры данных и экстренные апериодически сообщения, сигнализирующие о событиях, требующих немедленной реакции системы.

В каждом УЗ РВ-программы есть возможность определить свой модуль реакции на апериодические экстренные сообщения. Модуль получает управление сразу по приходу сообщения, т.е. он может прерывать выполнение как фоновых, так и основных модулей. Модуль реакции должен быть достаточно коротким по времени выполнения для того, чтобы не нарушить расписание. Для определения источника сообщения модуль реакции может использовать глобальную системную переменную *LnAsPort*, которая в момент вызова модуля будет содержать числовой идентификатор источника сообщения. Само сообщение содержится в другой глобальной переменной *MSGAsPort*.

Модуль реакции задаётся ключевым словом handler.

handler оператор_вызова_модуля;

В качестве фактического параметра разрешается использование констант, константных величин, глобальных переменных, флагов, очередей и ЭВУ.

факт_параметр ::= модификатор_ВВ-типа
константа | ид_константной_величины |
ид_глобальной_переменной | ид_ЭВУ | ид_очереди| ид_флага

Пример:

Пусть модуль REACT описан как

REACT(in integer, in char[80], out boolean*):1 ms,C;

Тогда в одном из УЗ может встретиться следующий модуль реакции

```
handler REACT(LnAsPort,MSGAsPort,elem);
```

который может изменить значение переменной elem.

2.5. Структура РВ-программы

Теперь, наконец, мы может рассмотреть структуру РВ-программы. Любая РВ-программа начинается с конструкции

```
rtpgm ид_РВ-программы
```

и заканчивается конструкцией

```
end ид_РВ-программы.
```

и состоит из следующих структурных единиц: раздела описания константных величин (const), раздела описания кадров входных данных (frame), раздела описания источников экстренной апериодической информации (extra), раздела описания глобальных переменных (glob), раздела описания прикладных модулей (modules) и одного или нескольких условных заданий – УЗ. Все разделы кроме frame и modules являются необязательными.

```
rtpgm ид_РВ-программы;
const
  описания_константных_величин
end;
frame
  описание_кадров_данных
end;
extra
   источники_апериодических_сообщений
glob
   описание_глобальных_переменных
end;
modules
   описание_прикладных_модулей
end;
УСЛОВНЫЕ_ЗАДАНИЯ
end ид_РВ-программы.
```

Окончание работы РВ-программы осуществляется по команде с консоли оператора.

2.5.1. Условное задание

Условное задание (УЗ) начинается с конструкции

condpgm ид_УЗ;

и заканчивается конструкцией

end ид_УЗ.

Условное задание состоит из раздела описания вектора условий (cvector), таблицы переключений (switchtable), раздела описания флагов (flags), раздела описания очередей (queues), раздела описания модуля реакции (handler), раздела описания фоновых работ (background), и одного или нескольких простых заданий – ПЗ. Обязательными являются разделы cvector и switchtable.

```
condpgm ид_УЗ;
cvector
  ЭВУ
switchtable
  таблица_переключений
end;
flags
  описание_флагов
end;
queues
  описание_очередей
end;
ПРОСТЫЕ_ЗАДАНИЯ
end ид_УЗ;
```

Обычно условное задание объединяет несколько связанных между собой режимов обработки входных данных (ПЗ). Смена ПЗ в рамках одного УЗ никак не отражается на фоновых работах, их выполнение продолжается на протяжении всего периода работы УЗ.

2.5.2. Простое задание

Простое задание (ПЗ) начинается с конструкции

```
smplpgm ид_ПЗ;
```

и оканчивается конструкцией

```
end ид_ПЗ.
```

Простое задание состоит из раздела описания используемого в ПЗ подвектора ВУ (cvector), раздела описания расчётных переменных (calc), предварительной (head) и заключительной (tail) частей ПЗ, а также одного или нескольких РВ-циклов. Разделы calc, head и tail являются необязательными.

```
smplpgm ид_ПЗ;
cvector
  описание_подвектора_ВУ;
head
  предварительная_часть_ПЗ
end;
tail
  заключительная_часть_ПЗ
nd;
calc
  описание_расчётных_переменных
end;
РВ-ЦИКЛЫ
end ид_ПЗ;
```

Простое задание задаёт некий режим обработки периодически поступающих кадров данных. Выполнение РВ-циклов привязано к поступлению кадров и осуществляется параллельно (псевдопараллельно на однопроцессорном IBM PC), с учётом зависимостей по данным.

ПЗ, точнее сказать, РВ-циклы ПЗ, выполняются до тех пор, пока не возникнет необходимость переключения на другое УЗ или ПЗ в связи с изменением значения ВУ.

Предварительная и заключительная части ПЗ выполняются однократно соответственно в начале и в конце ПЗ. РВ-циклы активизируются по приходу перво-

го после окончания всех операций по инициализации ПЗ кадра (включая выполнение предварительной части ПЗ).

Компилятор входного языка САПР СРВ-КОНСТРУКТОР реализован на IBM PC с использованием системы автоматизированного построения трансляторов СУПЕР, созданной в ВЦ РАН под руководством проф. В.А. Серебрякова [34].

Глава 3. Система автоматизированного синтеза модели ВСРВ. Генератор сетевой модели и расписаний

Реализация алгоритмов композиции подмоделей в комплексную модель оформлена в виде системы автоматизированного синтеза модели выполнения программ в реальном времени, которая состоит из следующих блоков: генератор рабочих таблиц (ГРТ), генератор сетевых моделей и расписаний (ГСМР) и генератор кодов (ГК).

На вход блока Генератор рабочих таблиц подаётся исходный текст РВ-программы, которая написана на языке сборки композиции моделей и в которой пользователь описывает, как следует объединить имеющиеся подмодели в комплексную модель. Каждая подмодель должна быть оформлена в виде прикладного модуля.

Генератор рабочих таблиц выполняет следующие функции:

1. Осуществляет синтаксический анализ конструкций РВ-программы.
2. Выдаёт сообщения о наличии ошибок в РВ-программе.
3. Генерирует таблицы данных для работы последующих блоков.
4. Вычисляет размеры буферов обмена данными программных модулей.

Генератор сетевых моделей и расписаний выполняет следующие функции:

1. Строит математическую модель вычислений в виде графа, в котором вершины соответствуют прикладным модулям, а дуги определяют частичный порядок их активизации.
2. Определяет директивные интервалы выполнения прикладных модулей.
3. Определяет, существует ли допустимое расписание выполнения прикладных модулей, и строит его, если оно существует.
4. Определяет необходимое количество копий для каждого прикладного модуля.
5. Вычисляет размеры буферов для входных параметров прикладных модулей.
6. Назначает стеки прикладным модулям для работы с данными.

Функции блока генерации кодов подробно описаны в главе 5 данной работы.

Блоки предварительной обработки написаны на языке Си и отлажены на многочисленных примерах, в частности, на примере решения задачи регрессионного анализа.

3.1. Основные функции генератора сетевых моделей и расписаний

Рассмотрим работу основного блока автоматизированного синтеза модели выполнения программ в реальном времени – ГСМР. Основное требование, предъявляемое к системам объединения подмоделей, зависящих от времени, заключается в том, что каждый этап вычислений должен выполняться в строго определённом временном интервале, называемом директивным интервалом. Иными словами, каждый этап вычислений должен начинаться не раньше некоторого начального момента времени и завершаться не позднее некоторого конечного момента времени. В предлагаемой системе объединения подмоделей этапом вычислений является прикладной программный модуль пользователя, реализующий подмодель, а начальным и конечным директивными сроками – соответственно момент прихода соответствующего кадра данных и момент следующей активизации того же самого модуля, т.е. момент поступления того последующего кадра данных, для которого снова должен запускаться этот модуль. Таким образом, при проектировании системы объединения подмоделей следует так спланировать вычисления, чтобы, во-первых, очередная активизация каждого прикладного модуля пользователя осуществлялась не раньше прихода соответствующего кадра данных, и, во-вторых, момент завершения его выполнения не превосходил момента следующей активизации того же самого модуля. Кроме того, при этом не должно нарушаться отношение частичного порядка выполнения модулей, определяемое их входными и выходными параметрами. Эту задачу решает ГСМР. ГСМР строит математическую модель вычислений, выполняемых в комплексной модели. С помощью этой модели ГСМР определяет, существует ли допустимое расписание выполнения прикладных модулей пользователя, и если оно существует, ГСМР строит его. Кроме того, ГСМР вычисляет некоторые характеристики комплексной модели, информация о которых необходима для её генерации и функционирования. Аналогом подмоделей являются *T*-модули, а аналогом копий подмоделей – *M*-модули. ГСМР строит сетевую модель для каждого случая (режима), определяемого вы-

полнением (или невыполнением) определённых условий (условное задание). Каждое условное задание, в свою очередь, может состоять из более мелких заданий – простых заданий.

3.2. Структурная схема и последовательность выполнения основных блоков

Структурная схема ГСМР изображена на рис. 3.1. Блоки, указанные на структурной схеме, выполняются в каждом простом задании в последовательности, указанной на рис. 3.2 (определения B-, T-, M- и E-модулей даны ниже).

3.2.1. Основные определения и обозначения

Копию прикладного программного модуля пользователя будем называть B-модулем, а B-модуль в совокупности с фактическими параметрами – T-модулем. Пусть T_1, T_2, ... , T_n – T-модули одного простого задания. Для каждого T_i $(i = 1, ..., n)$ известны следующие характеристики: p_i, p_{0i} – соответственно период и начальная фаза его активизации ($p_{0i} \le p_i$), выраженные в кадрах (т.е. k-я активизация (k = 0, 1, 2,...) T-модуля T_i соответствует поступлению $p_{0i} + (k - 1)p_i$ кадра данных); t_i – длительность выполнения T_i.

Кроме того, для каждого T-модуля T_i известны соответствующий B-модуль B_i и входные и выходные параметры. Расчётные входные параметры определяются в виде троек (a, T_j, g), где a – имя параметра, T_j – имя T-модуля, вычисляющего этот параметр, g – глубина (т.е. если некоторая активизация T-модуля T_i соответствует поступлению t-го кадра данных, то параметр a вычисляется T-модулем T_j на той его активизации, которая соответствует поступлению t'-го кадра данных, где $t' = \max\,(k\colon p_{0j} + (k - 1)p_j \le t - g)$.

Будем также использовать следующие обозначения: P – период поступления кадров данных, выраженный в микросекундах; m – максимальное число расчётных входных параметров T-модулей в простом задании; $q = \min\limits_{i=1,\dots,n} p_i$; $Q = \max\limits_{i=1,\dots,n} p_i$.

Рис.3.1. Структурная схема ГСМР.

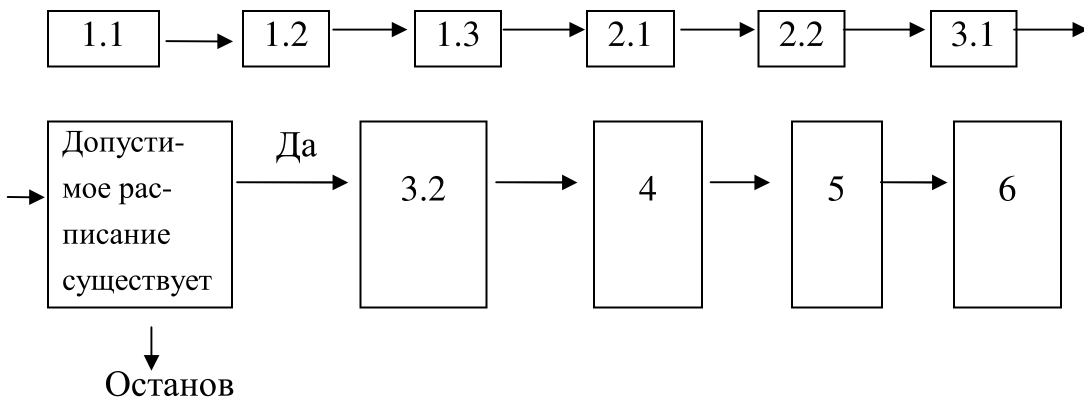


Рис. 3.2. Последовательность выполнения основных блоков ГСМР в каждом простом задании.

3.3. Основные алгоритмы

Перейдём к описанию алгоритмов работы перечисленных выше блоков, составляющих ГСМР.

3.3.1. Построение сети М-модулей

Сеть *M*-модулей задаёт частичный порядок на множестве активизаций *T*-модулей в каждом простом задании. Пусть *L* – наименьшее общее кратное периодов $p_1, p_2, \dots, p_n$. Величина *L* может быть вычислена с помощью алгоритма Евклида. Будем моделировать процесс активизации *T*-модулей на отрезке [*0*, *L*]. Этот процесс будет аналогичным при выполнении одного и того же простого задания для отрезков [*L, 2L*], [*2L, 3L*] и т.д. Определим массив A_{ij} ($i = 1, \dots, L; j = 1, \dots, n$), где *i* – номер кадра, а *j* указывает *T*-модуль T_j, следующим образом: $A_{ij} = 1$ при $i = p_{0j} + (k-1)p_j$, $k = 1, \dots, L/p_j$, $j = 1,\dots, n$; $A_{ij} = 0$ при всех остальных значениях *i, j*. Таким образом, каждому значению $A_{ij} = 1$ соответствует активизация *T*-модуля T_j на *i*-м кадре. Для каждого $A_{ij} = 1$ определим *M*-модуль M_{ij}, соответствующий активизации (или выполнению) *T*-модуля T_j после прихода *i*-го кадра данных. Тогда число *M*-модулей *N* в каждом простом задании равно числу элементов $A_{ij} = 1$, т.е. $N = \sum_{i=1}^{n} L/p_i$.

Определим сеть S, в которой узлы – это M-модули, а дуги указывают обмены параметрами между T-модулями. Так, если (a, T_l, g) – входной параметр M-модуля M_{ij}, то в сеть S вводится дуга $M_{tl} \rightarrow M_{ij}$, где $t = \max(k{:}\ p_{ol} + (k-1)\, p_l \leq i - g)$.

Сеть S используется для построения допустимого расписания прикладных модулей пользователя. Вычислительная сложность алгоритма построения сети M-модулей равна $O(Ln + Nm)$.

3.3.2. Вычисление директивных интервалов

Для нахождения допустимого расписания выполнения M-модулей требуется вычислить их директивные интервалы. Метод вычисления директивных интервалов основан на том, что, во-первых, каждая активизация T-модуля не может быть выполнена раньше прихода соответствующего кадра данных, и, во-вторых, выполнение T-модуля должно быть завершено не позднее следующей его активизации. Таким образом, выполнение M-модуля M_{ij} может быть начато не раньше момента времени $B_{ij} = P_i$ и должно быть закончено не позднее момента времени $F_{ij} = P_i + p_j$, а $[B_{ij}, F_{ij}]$ – директивный интервал M-модуля M_{ij}.

После того, как директивные интервалы M-модулей вычислены, их следует скорректировать следующим образом. Пусть M_{kl} – один из предшественников M-модуля M_{ij} в сети S. Поскольку выполнение M-модуля M_{ij} не может быть начато раньше, чем выполнение M_{kl}, то положим $B_{ij} = \max(B_{ij}, B_{kl})$. Кроме того, если выполнение M-модуля M_{kl} завершится позднее момента времени $F_{ij} - t_j$, то это приведёт к нарушению конечного директивного срока F_{ij} M-модуля M_{ij}. Поэтому положим $F_{kl} = \min(F_{kl}, F_{ij} - t_j)$.

Алгоритм коррекции директивных интервалов выглядит следующим образом. Для каждого M-модуля M_{ij} и каждого его предшественника M_{kl} в сети S вычислить новые величины B_{ij} и F_{kl}. Если при этом хотя бы одна из величин B_{ij} или F_{kl} изменилась, то данную процедуру следует повторить. Если все B_{ij} и F_{kl} остались без изменения, то алгоритм коррекции директивных интервалов на этом завершает работу. Заметим, что алгоритм коррекции позволяет достаточно просто находить допустимые расписания выполнения M-модулей для однопроцессорных систем. Вычислительная сложность алгоритма построения директивных интервалов равна $O(NmQ)$.

3.3.3. Построение допустимого расписания

После выполнения алгоритма коррекции для поиска допустимого расписания выполнения *M*-модулей на однопроцессорных вычислительных системах может быть применён *RU*-алгоритм [25]. Согласно этому алгоритму в каждый момент времени t следует выполнять активный *M*-модуль M_{ij}, у которого величина F_{ij} минимальная среди конечных директивных сроков всех активных *M*-модулей. Если таких *M*-модулей окажется несколько, то выполняется любой из них. *M*-модуль M_{ij} называется активным в момент времени t, если, во-первых, $B_{ij} \le t$, и во-вторых, его выполнение ещё не завершилось к моменту времени t. Выполнение *M*-модуля M_{ij} может быть прервано в некоторый момент времени $t = B_{kl}$, если конечный директивный срок *M*-модуля M_{kl} такой, что $F_{kl} < F_{ij}$. В более поздний момент времени, когда величина F_{ij} вновь окажется минимальной среди конечных директивных сроков всех активных *M*-модулей, выполнение *M*-модуля M_{ij} возобновится. Для проверки существования допустимого расписания моделируется описанный выше процесс выполнения *M*-модулей на однопроцессорной вычислительной системе. Если при этом окажется, что каждый *M*-модуль M_{ij} выполняется во временном интервале $[B_{ij}, F_{ij}]$ (где величины B_{ij} и F_{ij} получены после выполнения алгоритма коррекции), то допустимое расписание выполнения *M*-модулей существует. В противном случае (т.е. когда хотя бы один *M*-модуль M_{ij} некоторый период времени выполняется вне своего директивного интервала $[B_{ij}, F_{ij}]$) допустимого расписания не существует. Вычислительная сложность алгоритма проверки существования допустимого расписания выполнения *M*-модулей равна $O(LN)$.

Поскольку каждая величина B_{ij} при некотором целом k $(k > 0)$ равна P_k, то *M*-модули могут стать активными только в моменты прихода кадров. Это позволяет строить допустимое расписание в виде списков $(k, \{T_i\}, \{F_i\})$, где k – номер кадра $(1 \le k \le L)$, $\{T_i\}$ – список *T*-модулей, таких, что соответствующие им *M*-модули M_{ij} стали активными в момент времени P_k, а F_i – директивные сроки этих *M*-модулей. В таком формате допустимое расписание передаётся управляющей программе.

3.3.4. Построение таблицы соответствия *T*-, *E*- и *B*-модулей

В тех случаях, когда в одном простом задании нескольким *T*-модулям соответствует один и тот же *B*-модуль, а директивные интервалы соответствующих им

M-модулей пересекаются, следует хранить несколько копий этого B-модуля. (Этого можно избежать, если B-модули реентерабельны, т.е. допускают правильное выполнение при новых запусках после завершения или прерывания.) Эти копии B-модуля называются E-модулями.

Алгоритм построения необходимого числа E-модулей выглядит следующим образом. Если паре T-модулей T_i, T_j соответствует один и тот же B-модуль B и, кроме того, директивные интервалы $[B_{ki}, F_{ki}]$ и $[B_{lj}, F_{lj}]$ некоторых M-модулей M_{ki} и M_{lj}, соответствующих T-модулям T_i и T_j, пересекаются, то определить различные E-модули E_i и E_j для T-модулей T_i и T_j. Каждой паре E-модулей E_i и E_j соответствует один и тот же B-модуль B. Эту процедуру выполнить для всех пар T-модулей T_i, T_j $(i, j = 1, ..., n)$. В результате работы описанного выше алгоритма будет построена таблица $(T_i, E(T_i), B(T_i))$, $i = 1, ..., n$, где T_i – имя T-модуля, а $E(T_i)$ и $B(T_i)$ – имена соответствующих ему E- и B-модулей.

Вычислительная сложность алгоритма построения таблицы соответствия равна $O(n^2 (L/q)^2)$.

3.3.5. Вычисление размеров буферов для входных параметров

Буфера строятся для каждого расчётного входного параметра, т.е. параметра, который является входным в некотором T-модуле и вычисляется другим или тем же самым T-модулем. Пусть расчётный входной параметр T-модуля T_i определяется тройкой (a, T_i, g) , где T_j – T-модуль, вычисляющий параметр a, а g – глубина. Пусть M_{ki}, M_{kj} $(k \in K)$ – M-модули, соответствующие T-модулям T_i и T_j, причём параметр a вычисляется M-модулем M_{kj} для M-модуля M_{ki}. Тогда размер $d(a)$ буфера для параметра a определяется как максимально возможное число копий этого параметра, которое следует одновременно сохранять в памяти ЭВМ, и вычисляется по формуле

$$d(a) = \max \max_{k \in K} [(\mathrm{F_{ki}} - \mathrm{B_{kj}} + \mathrm{Pg})/(\mathrm{p_j P})],$$

где первый максимум берётся по всем T-модулям, имеющим параметр a среди расчётных входных параметров.

Вычислительная сложность алгоритма определения размеров буферов для входных параметров равна $O(n^2 mL/q)$.

3.3.6. Назначение стеков Т-модулям

Для работы прикладных модулей с данными используются стеки. В тех случаях, когда директивные интервалы *М*-модулей, соответствующих нескольким *Т*-модулям, пересекаются, для каждого из этих *Т*-модулей следует определить свой стек данных. Алгоритм определения стеков данных выглядит следующим образом. Рассмотрим пару *Т*-модулей T_i, T_j. Если директивные интервалы $[B_{ki}, F_{ki}]$ и $[B_{lj}, F_{lj}]$ некоторых *М*-модулей M_{ki} и M_{lj}, соответствующих *Т*-модулям T_i и T_j, пересекаются, то для этих *Т*-модулей следует определить различные стеки. В противном случае для них следует определить один стек. Данную процедуру выполнить для всех пар *Т*-модулей T_i, T_j $(i, j = 1, ..., n)$. Отметим, что указанную процедуру назначения стеков можно выполнять при построении таблицы соответствия *В*-, *Т*- и *Е*-модулей.

Глава 4. Управляющая программа САПР «СРВ-Конструктор»

4.1. Выбор операционной среды

САПР «СРВ-Конструктор» существенно облегчает работу пользователя по созданию программных продуктов *реального времени* на персональных компьютерах типа IBM PC. Благодаря существованию вышеописанного компилятора языка РВ, системы построения математической модели процесса управления и описываемому в следующей главе работы генератора кодов прикладных и библиотечных модулей задача предварительного этапа – сборки РВ-программы может быть успешно решена.

Работу в реальном времени прикладной программы пользователя, сгенерированной на этапе предварительной обработки, должна обеспечить *управляющая программа* (УП). Основные функции УП описаны ниже.

Хотя операционная система MS-DOS в чистом виде не позволяет организовать многозадачность, но, в отличие от известных систем, предназначенных для этих целей (Unix, OS/2 и т.п.), она обладает такими несомненными достоинствами, как надёжная файловая система и простота использования. Существует множество компиляторов для языков высокого уровня, рассчитанных на работу в среде MS-DOS. В силу этих соображений, выбор был остановлен на MS-DOS и сосредоточены усилия на создании приемлемой многозадачной оболочки реального времени.

Управляющую программу удалось реализовать в виде многозадачной надстройки над операционной системой MS-DOS версии 3.30 и выше на основе многозадачной оболочки реального времени Ctask-RT, прототипом которой послужил распространяемый бесплатно пакет Т.Вагнера [33], относящийся к классу свободнораспространяемого программного обеспечения. Ctask-RT позволяет создать программную среду, использующую многозадачные элементы BIOS IBM PC/AT, обойти барьеры нереентерабельности MS-DOS и даёт возможность работать совместно с резидентными (TSR) программами и под управлением оболочки MS-Windows.

Следует заметить, что написанная Управляющая программа достаточно мобильна, т.е. может быть перенесена на другие операционные системы и машины, поскольку почти 90% кода программы написано на языке С.

4.2. Основные функции управляющей программы

Управляющая программа является составной частью исполняемого EXE-модуля РВ-программы. Её функциями являются:

- приём, хранение и обработка поступающей извне информации (кадров данных);
- реакция на внешние апериодические сигналы;
- исполнение команд, вводимых с клавиатуры консоли;
- переключение между *условными* и *простыми* заданиями в соответствии со значениями системного вектора условий;
- запуск процессора согласно директивным срокам и приоритетам;
- обмен данными между процессами;
- действия по завершению работы процесса.

4.3. Структура управляющей программы

Логически управляющую программу можно условно разделить на следующие части:

- *интерпретатор команд;*
- *диспетчер (ядро);*
- *монитор данных;*
- *драйверы внешних устройств.*

4.3.1. Интерпретатор команд

Интерпретатор команд обеспечивает интерфейс между оператором и системой РВ. Он представляет собой систему меню, работа с которой возможна как с помощью клавиатуры, так и посредством «мыши».

Примерная схема сеанса работы интерпретатора управляющей программы может быть следующей. После запуска EXE-модуля РВ-программы начинает работать интерпретатор команд. На экране появляется главное меню. Выбор из меню осуществляется либо нажатием «горячей» клавиши подсвеченной буквы, либо мышью. Сначала доступны для выбора поля **StartUp** и **Quit**, что соответствует запуску и выходу из системы. После выбора **StartUp** появляется подменю с полями **Overview, Help, Info, Install & Run**, посредством которых можно получить

дополнительную информацию по системе или начать работу.

В системе реального времени может быть важен точный момент начала управления и не всегда оператор в состоянии запустить систему вовремя. В таком случае запуск системы может производить специальное стартовое УЗ.

Сразу после начала работы в реальном времени управляющая программа реагирует на возникновение в системе того или иного события (поступление кадра или асинхронного сообщения, окончание выполнения процесса). При необходимости смены ПЗ или УЗ сначала выполняются операции по инициализации, после чего управляющая программа снова переходит в режим реагирования на события.

После запуска системы становятся активными поля **SendMsg, Stop**, с помощью которых выбором из подменю можно послать сообщение или остановить работу. Завершение выполнения РВ-программы происходит выбором команды **Cancel**. Выбором **Stop** посредством подменю, можно остановить либо все **All**, либо основные **Foreground**, либо фоновые **Background** модули, дав досчитаться остальным. Предусматривается окончание РВ-программы как с печатью протокола работы РВ-программы **LOG**, так и без него.

Для удобства на экране отображаются: номер кадра, пришедшего последним, текущее время, имя выполняющейся программы, имя текущего *условного задания* после **CJob**, имя текущего *простого задания* после **SJob**.

4.3.2. Диспетчер

Функция *диспетчера* – активизация и запуск на счёт процессов в соответствии с приоритетами, которые определяются блоком ГСМР [35] при составлении расписания.

Множество процессов РВ-программы состоит из процессов реакции на внешнее апериодическое сообщение (по одному на каждое УЗ), основных процессов ПЗ и фоновых процессов УЗ.

Каждый процесс состоит из блока управления процесса (шапки) и тела процесса (Е-модуля). Е-модули создаются блоком генератора кодов согласно математической модели. В блоке управления содержится вся информация о параметрах процесса: типы параметров вызова, их имена и, возможно, параметры глубины по времени выборки данных.

При запуске процесса блок управления получает управление, генерирует запрос *монитору данных*, получает требуемые параметры, после чего запускает процесс. После окончания выполнения процесса управление передаётся в блок управления, который вновь генерирует запрос монитору данных на передачу данных от процесса к потребителю. После передачи данных процесс завершает работу.

В режиме реального времени система работает циклично. Периодом планирования всех активизаций основных процессов (работающих по расписанию) для каждого ПЗ является промежуток времени, равный наименьшему общему кратному периодов РВ-циклов, входящих в данное ПЗ.

Длина кадра внешних данных может периодически меняться от одного кадра к другому. Различаются постоянная и переменная части кадра, одна группа данных составляет постоянную часть кадра, другая – переменную.

При поступлении кадра данных управление сначала получает *монитор данных*, который записывает постоянную и переменную части кадра на место самых «старых» экземпляров данных из имеющихся в буфере. Монитор также увеличивает на единицу системную переменную – счётчик абсолютных номеров кадров. *Диспетчер* активизирует все основные процессы, для которых начальные директивные сроки очередных активизаций приурочены к поступлению данного кадра.

Во время периода планирования каждый основной процесс может быть активизирован один или большее число раз. Для каждой активизации процесса устанавливаются начальный (в соответствии с приходом кадра) и конечный директивные сроки. Директивные интервалы разных активизаций одного и того же процесса не перекрываются. Также не могут перекрываться интервалы активизаций процессов, использующих один и тот же В-модуль.

Всякий основной процесс в любой из моментов времени находится в одном из двух состояний: активном или пассивном. Процесс находится в активном состоянии, если точка текущего момента попадает внутрь директивного интервала одной из незавершённых активизаций процесса. В каждый момент времени, когда процессор выполняет основной процесс, этот процесс должен иметь самый ранний конечный директивный срок среди всех активных процессов.

Если в какой-либо момент окажется, что в системе нет активных основных процессов, управляющая программа (*диспетчер*) пытается запускать фоновые

процессы. Множество фоновых процессов данного УЗ просматривается в порядке убывания приоритетов. Если флаг готовности/ ожидания какого-либо процесса выставлен в единицу, то этот процесс, имеющий наибольший приоритет, назначается на исполнение, а просмотр прекращается.

Если готовых к исполнению процессов нет и среди фоновых, управляющая программа переходит в состояние ожидания следующего события в системе.

При появлении внешнего апериодического сообщения управляющая программа помещает в соответствующие системные переменные числовой идентификатор источника сообщения и текст сообщения, после чего диспетчер передаёт управление процессу реакции на сообщение.

При вводе команды с консоли управление передаётся *интерпретатору команд*, который выполняет введённую команду.

Реагирование на возникновение любого из событий является неделимой операцией, оно не может быть прервано. Если в это время произойдёт ещё какое-либо событие (например, поступит кадр данных), то управляющая программа должна отреагировать на него только после того, как «обслужит» первое, более раннее по времени событие.

Если одно из событий – поступление кадра, приход апериодического сообщения или ввод команды с консоли – произошло в момент выполнения какого-либо процесса, то перед тем, как управляющая программа начнёт реагировать на событие, она запоминает контекст текущего процесса (содержимое регистров, состояние стека и т.п.).

4.3.3. Монитор данных

Монитор данных обеспечивает приём и хранение кадров данных, поступающих в систему извне с помощью драйверов внешних устройств, поставку этих параметров процессам, обмен данными между процессами, изменение системного вектора условий и флагов готовности/ ожидания фоновых процессов.

4.3.4. Драйверы внешних устройств

Драйверы внешних устройств позволяют принимать по прерываниям или/ и посредством последовательного опроса приходящие извне кадры данных и поставляют их монитору. В настоящее время реализован следующий стандартный

набор драйверов внешних устройств: драйверы клавиатуры, принтера и последовательного порта (через который по умолчанию поступают внешние данные). Могут обслуживаться несколько печатающих устройств и коммуникационных портов. Для подключения нестандартного оборудования нужно лишь добавить соответствующий драйвер, что обеспечивает достаточную гибкость системы.

Глава 5. Программный комплекс «СРВ-Конструктор»

5.1. Сборка и запуск программного комплекса «СРВ-Конструктор»

В первой главе настоящей работы уже упоминалось, что создание системы «СРВ-Конструктор» явилось итогом последовательного применения и развития простого принципа: максимальная подготовка программы реального времени (РВ-программы) к выполнению до запуска собственно режима реального времени.

Соответственно, различают подготовительный этап использования «СРВ-Конструктора», по окончании которого генерируется готовая к выполнению прикладная РВ-программы, и этап работы в самом режиме реального времени полученной программы.

Для запуска подготовительного этапа от пользователя требуются:

– прикладные модули, написанные на языках программирования C, FORTRAN, PASCAL, ASSEMBLER (в формате компиляторов фирмы Microsoft v. 6.0 (см. [36]));

– задание на обработку информации в реальном времени, написанное на входном языке СРВ-Конструктора.

В связи с различием описания данных в разных языках программирования, использующие дисковые файлы прикладные модули должны быть написаны на языке C.

Общая схема предварительного этапа показана на рис. 5.1.

Предварительный этап работы системы «СРВ-Конструктор» включает последовательный вызов следующих обрабатывающих программных комплексов:

– компилятора для трансляции исходного текста РВ-программы; формат обращения: lang <имя_РВ-программы.rts>;

– программы вывода диагностики и сообщений об ошибках при компиляции, помещаемых компилятором в файл <имя_РВ-программы.lst> (предусмотрен анализ более 70 типов ошибок); формат обращения: spl;

– программы генерации сетевых моделей и расписаний; формат обращения: _gsm_gr;

– программы генерации кодов; формат обращения: _cd_gnr.

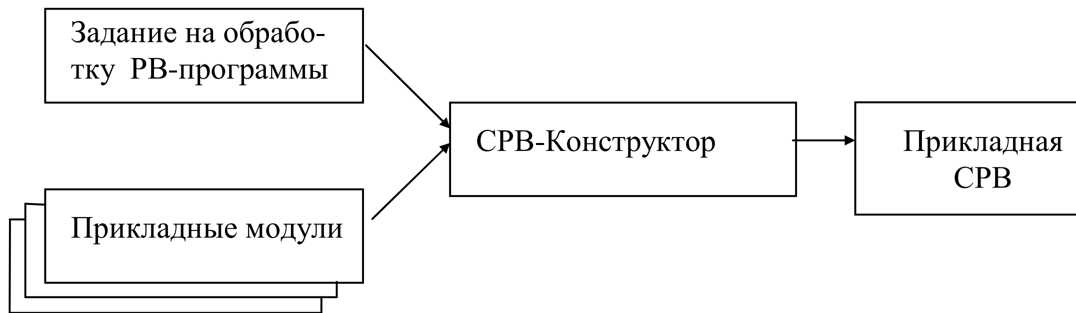


Рис. 5.1. Последовательность проектирования ВСРВ с помощью СРВ-Конструктора.

Далее осуществляется трансляция полученных модулей с помощью компилятора фирмы Microsoft.C (6.0)[36] и сборка готового модуля прикладной программы реального времени с помощью стандартного компоновщика связей фирмы Microsoft link. Файл с расширением <имя_РВ-программы.lrf>, используемый при работе данного компоновщика (включает перечень предназначенных для сборки файлов), создаётся автоматически.

Для выполнения расписания может понадобиться создание копий некоторых прикладных модулей пользователя. Список пар имён таких модулей (имеющееся имя и то, которое необходимо создать) выдаётся в файл «_cg_user.lst». Пользователю необходимо создать файл с указанным новым именем при помощи копирования имеющегося файла и замены в исходном тексте полученного файла имени программы на указанное.

Для успешной работы компилятора необходим также ряд вспомогательных файлов. Часть из них устанавливается с дистрибутивной дискеты системы, другая вырабатывается перечисленными выше программными комплексами СРВ-Конструктора.

Ниже подробно рассмотрены назначение и функции программы генератора кодов (ГК).

5.2. Генератор кодов.

Генератор кодов предназначен для автоматизации одной из заключительных операций проектирования конкретной программы реального времени:

1) формирования на языке C (в формате MICROSOFT) и запись в текущий каталог исходных текстов процессов;

2) создания ряда вспомогательных файлов для последующих некоторых действий пользователя, компиляции и редактирования связей.

Главной и самой объёмной из функций программы является формирование исходных текстов прикладных программ РВ-программы. Каждый основной и фоновый процесс РВ-программы состоит из экземпляра (*E*-модуля) какого-либо прикладного модуля пользователя (*B*-модуля) и Т-модуля, содержащего:

– описание типа *E*-модуля в соответствии с его языком программирования;

– описание параметров *E*-модуля в соответствии с их типами;

– обращение к монитору данных (МД) с помощью макрокоманд для чтения исходных и записи выходных параметров;

– обращение к данному *E*-модулю;

– при необходимости – запрос на наличие элементов в очереди (для фонового процесса).

Последовательность работы процесса, опуская некоторые детали (простейшие циклы по вызову *E*-модуля внутри процесса, обращение к монитору данных для проверки очередей для указанных фоновых процессов и т.п.), иллюстрируется на рис. 5.2.

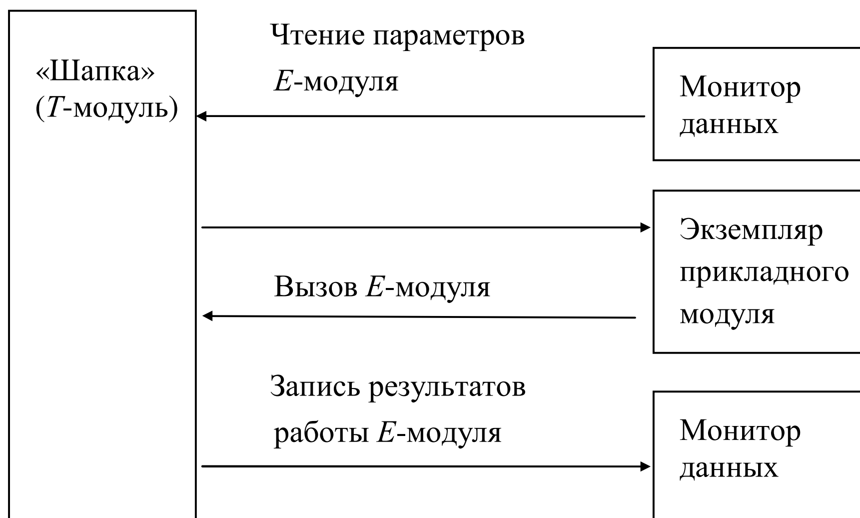


Рис. 5.2. Схема работы модуля – «шапки».

«Шапка» процесса инициируется при запуске процесса. При наличии у *E*-модуля входных параметров он генерирует с помощью макрокоманд запросы к монитору данных, получает требуемые параметры и запускает прикладной модуль. После окончания работы *E*-модуля управление опять передаётся в «шапку», которая снова генерирует запросы монитору данных на передачу данных (если такие запросы есть), и процесс завершается.

На рис. 5.3 представлена «Твёрдая копия» экрана консоли при демонстрационном запуске САПР СРВ-Конструктор для выполнения трёх пользовательских модулей Module1, Module2 и Module3 на языке Си с длительностью работы 150, 250 и 300 мс соответственно. При этом первый и третий модули выполняются в три раза чаще, чем второй. Прерывания разрешены. С периодом времени длинной 1500 мс расписание будет полностью повторяться.

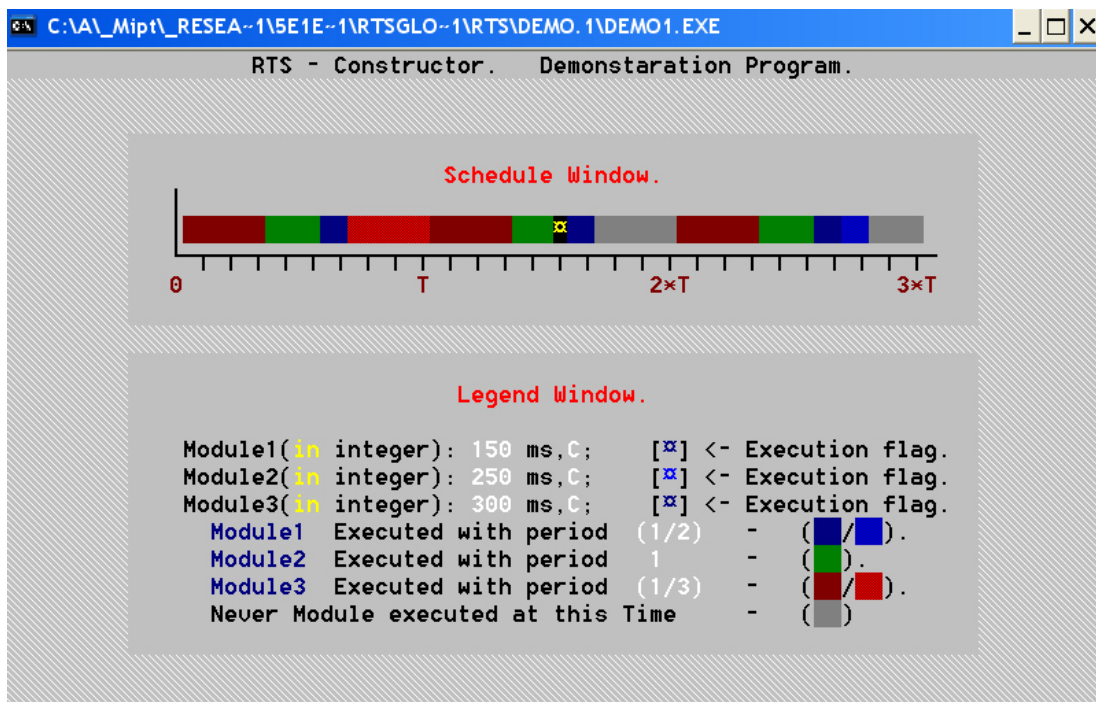


Рис. 5.3. «Твёрдая копия» экрана консоли при демонстрационном запуске САПР СРВ-Конструктор.

Глава 6. Планирование расписаний для многопроцессорного варианта системы «СРВ-Конструктор». Задача распределения *M* заданий на *N* процессоров

В данной главе приводятся основные понятия, принятые допущения и формальная постановка задачи построения рационального расписания выполнения M независимых заданий на N процессорах в системе реального времени. При этом понятия задание и вычислительная работа считаются синонимами, как и понятия выполнение и обработка (по отношению к заданиям). Рассматривается случай выполнения заданий без прерываний. Производительность каждого процессора считается в одном случае одинаковой, затем рассматривается более общий случай с различной производительностью процессоров.

6.1. Постановка задачи для случая процессоров одинаковой производительности

Имеется M независимых заданий и N идентичных процессоров, на каждом из которых может выполняться одновременно не более одного задания. В фиксированный момент времени каждое задание выполняется не более чем одним процессором. Задания выполняются без прерываний и переключений с одного процессора на другой. Для каждого задания i известна длительность t_i его выполнения. Необходимо определить такое распределение заданий по процессорам, чтобы время выполнения всей совокупности заданий (от начала первого до завершения последнего) было минимальным.

Эту задачу можно описать следующим образом:

$$\overline{T} \to \min, \qquad \begin{cases} \sum\limits_{i=1}^{M} t_i x_{ij} \le \overline{T}, & j = \overline{1,N}, \\ \sum\limits_{j=1}^{N} x_{ij} = 1, & i = \overline{1,M}, \\ x_{ij} = \{0,1\}, \quad i = \overline{1,M}, \quad j = \overline{1,N}, \end{cases} \qquad (1)$$

где $\overline{T}$ – время выполнения всей совокупности заданий; $x_{ij} = 1,$ если i-е задание распределено на j-й процессор, $x_{ij} = 0$ в противном случае.

6.2. Постановка задачи для случая процессоров различной производительности

Имеется M независимых заданий и N процессоров, которые могут отличаться производительностью. На каждом процессоре одновременно может выполняться не более одного задания. В фиксированный момент времени каждое задание выполняется не более чем одним процессором. Задания выполняются без прерываний и переключений с одного процессора на другой. Объём работы по выполнению задания i равен Q_i. Производительность процессора j равна S_j. Таким образом, если i-е задание выполняется процессором j в течение интервала t_{ij}, то

$Q_i = S_j \cdot t_{ij}$.

Будем считать, что задания упорядочены по не возрастанию объёмов их работ: $Q_1 \geq Q_2 \geq \ldots \geq Q_M$. А процессоры упорядочены по не возрастанию их производительностей: $S_1 \geq S_2 \geq \ldots \geq S_N$.

Требуется определить такое распределение заданий по процессорам, чтобы время выполнения всей совокупности заданий (от начала первого до завершения последнего) было минимальным.

Эту задачу можно записать следующим образом:

$$\overline{T} \to \min, \qquad \begin{cases} \sum_{i=1}^{M} \left(\frac{Q_i}{S_j} \right) x_{ij} \leq \overline{T}, & j = \overline{1, N}, \\ \sum_{j=1}^{N} x_{ij} = 1, & i = \overline{1, M}, \\ x_{ij} = \{0,1\}, \quad i = \overline{1, M}, & j = \overline{1, N}, \end{cases} \qquad (6.2),$$

где $\overline{T}$ – время выполнения всей совокупности заданий; $x_{ij} = 1$, если i-е задание распределено на j-й процессор, $x_{ij} = 0$ в противном случае.

Задачи, сформулированные в разд. 6.1 и 6.2, являются, как показано в [19, 23], *NP-трудными* в сильном смысле.

6.3. Существующие методы решения

Основные известные алгоритмы для решения поставленной задачи:

- алгоритмы случайного поиска (ненаправленного, направленного, с самообучением);
- алгоритмы детерминированной коррекции расписаний;
- алгоритмы имитации отжига;
- генетические алгоритмы;
- алгоритмы агрегирования.

Выбор данных классов алгоритмов обусловлен следующими причинами:

1. *NP*-трудность задачи исключает возможность использования точных методов при решении задач большой размерности, например метода динамического программирования [37, 38] или полного перебора.
2. Дискретность значений функций и аргументов задач 6.1 и 6.2 делает невозможным использование методов непрерывной оптимизации.

6.3.1. Алгоритмы случайного поиска

Основой методов случайного поиска служит итерационный процесс:

$$X^{k+1} = X^k + \alpha_k \cdot \frac{\xi}{\|\xi\|}, k = 0,1,\ldots,$$

где α_k – величина шага, $\xi = (\xi_1,\ldots,\xi_n)$ – некоторая реализация n-мерного случайного вектора ξ.

Ненаправленный случайный поиск (метод Монте-Карло) заключается в многократном случайном выборе допустимых вариантов решений и запоминании наилучшего из них:

$$X^{k+1} = \xi, \quad \xi \in S, \quad g_i(\xi) \le 0, \quad k = 0,1,\ldots, \quad i = 1,\ldots,m.$$

Все алгоритмы направленного случайного поиска без самообучения делают шаг от текущего значения X^k оптимизируемых параметров. Известны следующие алгоритмы направленного случайного поиска без самообучения [40]: алгоритм с парной пробой, алгоритм с возвратом при неудачном шаге, алгоритм с пересчётом при неудачном шаге, алгоритм с линейной экстраполяцией, алгоритм наилучшей

пробы, алгоритм статистического градиента.

Алгоритмы случайного направленного поиска с самообучением заключаются в перестройке вероятностных характеристик поиска, т.е. в определённом целенаправленном воздействии на случайный вектор ξ. Он уже перестаёт быть равновероятным и в результате самообучения приобретает определённое преимущество в направлениях наилучших шагов. Это достигается введением вектора памяти $P^k = (p_1^k, p_2^k, \ldots, p_n^k)$, где p_j^k – вероятность выбора положительного направления по j-ой координате на k-ом шаге. Алгоритм рекуррентно корректирует значение компонентов этого вектора на каждой итерации в зависимости от того, насколько удачным/неудачным (изменилось значение целевой функции) был сделанный шаг. Описание и анализ различных способов коррекции вектора P^k приведены в [40].

Для алгоритмов случайного поиска можно выделить следующие особенности:

- слабая адаптация или отсутствие её к поведению оптимизируемой функции для алгоритмов ненаправленного и направленного случайного поиска без самообучения;
- низкая скорость сходимости, уменьшающаяся с возрастанием количества оптимизируемых параметров.

Сравнительное исследование известных на сегодня методов поиска, а также обзор ряда новых алгоритмов приведён в трудах [41, 42].

6.3.2. Алгоритмы детерминированной коррекции расписаний

Метод ветвей и границ был предложен в 1960 г. Ленд и Дойг [43] для решения общей задачи целочисленного линейного программирования. Однако действительный интерес к этому методу проявился лишь спустя несколько лет в связи с его применением к задаче коммивояжёра в работе Литтла, Мурти, Суини и Кэрел [44]. Начиная с этого момента, появилось большое число работ, посвящённых методу ветвей и границ и его разновидностям. Это связано с тем, что авторы первыми обратили внимание на широту возможностей метода, отметили важность использования особенностей задачи и сами воспользовались спецификой задачи коммивояжёра.

В основе метода ветвей и границ лежит идея последовательного разбиения

множества допустимых решений на подмножества (стратегия “разделяй и властвуй”). На каждом шаге метода элементы разбиения подвергаются проверке для выяснения, содержит данное подмножество оптимальное решение или нет. Проверка осуществляется посредством вычисления оценки снизу для целевой функции на данном подмножестве. Если оценка снизу не меньше *рекорда* — наилучшего из найденных решений, то подмножество может быть отброшено. Если значение целевой функции на найденном решении меньше рекорда, то происходит смена рекорда. По окончанию работы алгоритма рекорд является результатом его работы.

Если удаётся отбросить все элементы разбиения, то рекорд — оптимальное решение задачи. В противном случае, из неотброшенных подмножеств выбирается наиболее перспективное (например, с наименьшим значением нижней оценки), и оно подвергается разбиению. Новые подмножества вновь подвергаются проверке и т.д.

Опишем работу метода ветвей и границ, приведённого в [45], для случая задачи упорядочения работ, длительности выполнения которых заданы произвольной матрицей $\tau_{ij}, i = 1,...,M, j = 1,...,N.$ Правило оценки границ и способ формирования дерева вариантов для рассматриваемой задачи основано на следующих предположениях.

Пусть $z < M$ работ уже распределены по процессорам. Частично сформированные при этом подмножества обозначим $S_j^z (j = 1, .., N)$. Тогда загрузка системы составит $B_j^z = \sum_{i \in M_j} \tau_{ij}$, $j = 1, .., N$. Предположим, что каждая из оставшихся $M - z$ работ будет обслужена с максимальной для неё производительностью, т.е. в течение времени

$$t_i = \min_{1 \le j \le N} \tau_{ij}, \; i = z + 1, z + 2, \ldots, M,$$

и, кроме этого, эти работы распределяются между наименее загруженными процессорами так, чтобы их загрузка была равномерной. Очевидно, номера этих процессоров будут определяться первыми

$$r = \begin{cases} N, & \text{если } z \le M - N, \\ M - z, & \text{если } z > M - N \end{cases}$$

членами последовательности $\pi = (j_1, j_2, ..., j_s, j_{s+1}, ..., j_N)$, отвечающей неравенству $B^z_{j_1} \le B^z_{j_2} \le ... \le B^z_{j_s} \le B^z_{j_{s+1}} \le ... \le B^z_{j_N}$. При сделанных допущениях условное время занятости каждого из процессоров составит $B^z = \frac{1}{r}\left(\sum_{s=1}^{r} B^z_{j_s} + \sum_{i=z+1}^{M} t_i \right)$.

Общее время обслуживания в системе с такой идеализированной дисциплиной будет определяться наибольшей из величин B^z и $B^z_{j_N}$. Для любого реального распределения справедливо неравенство $\max\limits_{S^z_j \subset S_j} B_j \ge \max\{B^z; B^z_{j_N}\}$. Кроме того, в силу целочисленности задачи

$$\max_{S^z_j \subset S_j} B_j \ge \max_{z+1 \le i \le M} t_i \ . \qquad (6.3.1)$$

Таким образом, полученные выражения могут использоваться для оценки нижней границы решения в зависимости от распределения первых z работ. Запишем оценку в следующем виде:

$$B_{zj} = \max\{B^z; B^z_{j_N}; \max_{1 \le i \le M} t_i\} \qquad (6.3.2)$$

где B_{zj} – нижняя граница варианта распределения, при котором z-я работа назначена на j-й процессор.

Выражение (6.3.2) позволяет установить начальную нижнюю границу B_0 всего множества допустимых планов. Для этого достаточно положить $z = 0$. Получим

$$B_0 = \max\left\{ \frac{1}{N} \sum_{i=1}^{M} t_i; \max_{1 \le i \le M} t_i \right\} \qquad (6.3.3)$$

Процесс оптимального решения состоит в направленном движении по вершинам дерева вариантов распределения работ. Стратегия ветвления заключается в следующем. На z-м уровне дерева формируется N вариантов распределения z-й работы, для чего последовательно принимается $z \in N^z_j$ (j = 1,…, N). Полученные варианты оцениваются с помощью выражения (6.3.2). Вершина, соответствующая варианту с наименьшей оценкой

$$B_{zp} = \min_{1 \le i \le M} B_{zj}, \qquad (6.3.4)$$

выбирается в качестве активной для дальнейшего ветвления. Остальные вершины

данного уровня – концевые. При наличии нескольких вершин, отвечающих условию (6.3.4), выбирается любая из них.

Процесс выполняется до тех пор, пока дальнейшее ветвление становится невозможным. Решение оптимально, если дерево вариантов не имеет концевых вершин с оценками

$$B_{ij} < B^* \text{ при } i \in S_j, \ j = 1, \ldots, N, \qquad (6.3.5)$$

где $B^* = \max\limits_{1 \le i \le N} B_j$ – значение целевой функции полученного решения. В противном случае производится проверка и уточнение путём ветвления из вершин, отвечающих неравенству (6.3.5). Проверку целесообразно начинать с нижних уровней, так как при этом может быть достаточно быстро достигнуто улучшение решения, что, в свою очередь, сократит число вариантов верхних уровней, подлежащих проверке. Ветвление из проверяемой вершины прекращается, если на каком-либо уровне оценка нижней границы достигнет или превысит величину B^*. В случае получения нового решения для проверки используется соответствующее значение целевой функции.

Рассмотрим основные шаги вычислительного алгоритма.

1. По формуле (6.3.3) определяем начальную нижнюю границу B_0, полагая $z = 0$.
2. Увеличиваем значение z на единицу. Проверяем условие $z \le M$. Если условие выполняется, переходим к п. 3, в противном случае – к п. 5.
3. Создаём вершины z-го уровня и по формуле (6.3.2) оцениваем границы вариантов решения (первоначально полагаем $B^* = \infty$). Выявляем вершину с наименьшей оценкой, проверяем для неё условие (6.3.5). Если оно выполнено, переходим к п. 4, если нет – к п. 8.
4. Включаем выбранную вершину в решение и возвращаемся к п. 2.
5. Запоминаем решение $\{S_j\}$ ($j = 1, \ldots, N$) и соответствующее значение B^* целевой функции.
6. Проверяем условие $B^* = B_0$. Если оно выполняется, переходим к п. 10, в противном случае – к п. 7.
7. Исключаем из решения вершину M-го уровня ветвления.
8. Уменьшаем значение z на единицу. Проверяем условие $z = 0$. Если условие выполнено, переходим к п. 10, в противном случае к п. 9.

9. Исключаем из решения вершину z-го уровня. Полагаем соответствующий ей элемент матрицы равным ∞. Восстанавливаем элементы (z + 1)-й строки матрицы $\| \tau_{ij} \|$ и возвращаемся к п. 3.
10. Выводим результаты и завершаем процесс.

Различные аспекты метода ветвей и границ изложены в [46, 47].

Быстрые эвристические алгоритмы. В основу быстрых эвристических алгоритмов легли наборы практических методов – *правил*, применяя которые на каждом этапе работы алгоритма можно получить расписание. Большое исследование, посвящённое различным быстрым эвристикам и их применению для широкого круга задач дискретной оптимизации, проведено в [3]. В книге [47] классифицируются более 100 различных правил распределения работ и проведена попытка проанализировать основную идею, лежащую в основе различных правил. Наиболее известными из таких правил являются: *первой назначается на выполнение работа с наибольшей длительностью*, *первой назначается работа с наименьшей длительностью*. Ряд быстрых эвристических алгоритмов с учётом ограничений по оперативной памяти процессоров и разрешёнными прерываниями на выполнение заданий описан в [48].

Существуют и более сложные современные эвристические алгоритмы, такие как, например, *муравьиные алгоритмы* [49].

6.3.3. Алгоритмы имитации отжига

Представим общую схему работы алгоритмов имитации отжига:

1. Выбрать начальное приближение x и установить начальную температуру t.

2. Сгенерировать новое решение y из x.

3. Вычислить значение $f(y)$, где $f(y)$ – функция оценки качества решения.

$x = y$, если $f(y) - f(x) \le 0$, либо $f(y) - f(x) > 0$ и $e^{\frac{-(f(y)-f(x))}{t}} > random[0,1)$, где $random[0,1)$ – случайная величина, равномерно распределённая в интервале $[0,1)$.

4. Повторить шаги 2-3 заданное число раз.

5. Понизить t.

6. Если не достигнут критерий останова – перейти к 2.

В литературе можно встретить достаточно большое количество упоминаний

использования данного метода. Так, например, в [50] данный алгоритм был применён к задаче составления расписаний, и были получены хорошие результаты, однако время работы алгоритма было очень велико (это подтверждается и результатами, приведёнными в данной работе). В [51] был использовано совместное применение алгоритма имитации отжига и эволюционного алгоритма. Подход заключён в представлении стохастического поиска алгоритма имитации отжига в виде эволюционного процесса. Авторами утверждается, что результаты их подхода лучше, чем остальные представленные в литературе.

6.3.4. Генетические и эволюционные алгоритмы

Большая работа по исследованию генетических алгоритмов (ГА) и эволюционных алгоритмов (ЭА) проведена в [8] и [52]. Общая схема этих алгоритмов может быть представлена следующим образом [53, 54, 55]:

1. Сгенерировать случайным образом популяцию размера P.
2. Вычислить целевую функцию для каждой строки популяции.
3. Выполнить операцию селекции.
4. Выполнить генетические операции (скрещивание и мутацию).
5. Если критерий останова не достигнут, перейти к п. 2, иначе завершить работу.

Популяция – это множество строк, состоящих из символов конечного алфавита. Каждая строка представляет в закодированном виде одно из возможных решений задачи. По строке может быть вычислена функция выживаемости, которая характеризует качество решения. В качестве начальной популяции может быть использован произвольный набор строк. Основные операции алгоритма: селекция, скрещивание и мутация выполняются над элементами популяции. Результатом их выполнения является очередная популяция. Данный процесс продолжается итерационно до тех пор, пока не будет достигнут критерий останова.

ГА отличаются от ЭА в основном способом кодирования и универсальностью основных операций. В ГА допустимо лишь бинарное кодирование и операции могут быть универсальными. Среди основных проблем использования ГА и ЭА можно выделить следующие:

1. Выбор способа кодирования решений.
2. Определение операций скрещивания и мутации для работы с используемым

представлением решения.

3. Определение параметров алгоритма (размера популяции, вероятностей скрещивания и мутации).
4. Задание целевой функции и критерия останова.

Данные алгоритмы обладают следующими преимуществами:

1. при правильной настройке параметров алгоритмы обладают высокой степенью адаптации к характеристикам оптимизируемой функции;
2. высокая скорость сходимости, не зависящая от сложности оптимизируемой функции;
3. возможность настройки параметров алгоритма на задачах небольших размерностей и перенос их в дальнейшем на задачи большей размерности;
4. изменение детализации представления архитектуры вычислительных систем без существенного изменения алгоритма.

В [56] рассматривается пример использования ГА для проектирования контроллеров физических роботов. В работе приводиться большое количество примеров применения ГА для создания как программной, так и аппаратной составляющей контроллеров. В качестве программной составляющей в большинстве случаев рассматриваются нейронные сети, при этом ГА используется для настройки весов нейронной сети.

Возможность обучения нейронных сетей с помощью ГА также широко обсуждается в [57]. В этой работе рассматривается возможность не только обучения нейронных сетей при помощи ГА, но также и вопрос выбора архитектуры нейронной сети, т.е. количества нейронов и связей между ними, с использованием ГА.

Также встречается большое число примеров использования ГА для решения проблемы составления расписаний [58, 59, 60]. В [61] рассматривается применение ГА для решения задачи составления расписания экзаменов в университетах.

6.3.5. Алгоритмы агрегирования

Этот подход содержит следующие основные элементы: последовательное разбиение задачи на подзадачи упорядочения существенно меньшей размерности, формирование дерева подзадач, решение подзадач, формирование решения зада-

чи из решения подзадач в соответствии с построенным деревом подзадач [62]. Одни из последних результатов в разработке методов агрегирования для решения задач теории расписаний можно найти, например, в работах Д.В.Красовского [63-65], где приводится описание алгоритмов решения задачи составления расписания на идентичных процессорах, основанных на общей методике агрегирования.

6.4. Выводы

Из приведённого обзора методов решения задач комбинаторной оптимизации с ограничениями и анализа их применимости для решения задачи синтеза структуры ВС следует, что на выбор метода большое значение оказывает конкретные особенности задачи и целей, поставленных перед разработчиками.

Например, генетические алгоритмы обладают следующими преимуществами по сравнению с другими методами:

1. ГА, в отличие от жадных алгоритмов, позволяют производить синтез структуры ВС одновременно с построением расписаний. При этом ГА позволяют изменять число оптимизируемых параметров без модификации самого алгоритма, так как вычисление функции выживаемости отделено от операции преобразования решений.
2. Скорость сходимости ГА, в отличие от алгоритмов имитации отжига и алгоритмов случайного поиска, не зависит от сложности ландшафта целевой функции и размерности пространства решений.

В тоже время большая вычислительная сложность применения ГА по сравнению с жадными алгоритмами в огромном ряду практических случаев делает предпочтительными применение последних.

Из вышесказанного следует, что целесообразно с учётом особенностей задач совершенствовать методики построения расписаний по всем указанным направлениям. В силу научных интересов автор сосредоточил внимание на новых эвристических алгоритмах. Предлагаемые эвристические алгоритмы подробно изложены в следующей главе.

Глава 7. Эвристические алгоритмы распределения заданий по процессорам в САПР систем реального времени

В данной главе описываются эвристические алгоритмы решения задач, сформулированных в разд. 6.1 для случая одинаковых процессоров и в разд. 6.2 для случая процессоров с различной производительностью. Эти алгоритмы предполагается использовать в дальнейших версиях системы «СРВ-Конструктор», даётся оценка точности их вычислений, приводятся результаты серии расчётов по данным алгоритмам и сравнительный анализ последних;

В данной главе рассмотрены два новых эвристических алгоритма, которые основаны на разных правилах предпочтения. Ниже приводятся их подробные описания.

Для проведения сравнительных испытаний предложенных алгоритмов использовался классический «жадный» алгоритм [23] распределения заданий (предпочтительное распределение самого трудоёмкого из необработанных заданий на наименее загруженный процессор), а для небольших размерностей были проведены контрольные расчёты и с алгоритмом, осуществляющим поиск точного решения задачи. Для определения качества полученных решений также использовались идеальные оценки времени выполнения при заданном числе и других характеристиках процессоров и общем объёме подлежащих обработке заданий.

Для генерации входных данных (число процессоров, число заданий, объём трудоёмкости каждого задания) применялась стандартная программа «генератор случайных чисел». При генерации данных учитывалось то обстоятельство, что, если какое-либо задание по трудоёмкости превосходит идеальную оценку времени выполнения всей совокупности работ (особенно это справедливо для случая одинаковых процессоров), то общее время выполнения будет определяться длительностью этого задания. Понятно, что такой случай для испытания новых эвристических алгоритмов не интересен, и потому соответствующие входные данные не генерировались.

Все программы выполнены на языке С++. Испытания проводились на компьютерах класса Pentium-II – Pentium-IV.

7.1. Решение задачи 6.1 (для случая одинаковых процессоров).

При работе эвристического алгоритма вычисляется идеальная оценка $t^* = (\sum_{i=1}^{M} t_i)/N$ времени завершения всей совокупности заданий, а также её калиброванное значение $\bar{t}$. Под калибровкой понимается увеличение величины t^* на некоторое число процентов, причём алгоритм запускается с разными значениями калибровки и выбирается лучшее достигнутое для рассматриваемого набора входных данных решение.

7.1.1. Эвристический алгоритм 1

а) Отсортировать задания по не возрастанию длительностей, т.е. будет выполняться соотношение $t_i \ge t_{i+1},\ i = 1,..., M - 1$.

б) Вычислить величины t^* и $\bar{t}$.

в) Распределять на каждый процессор *j*, начиная с первого, нераспределённые ранее задания до тех пор, пока суммарная длительность заданий на каждом процессоре не будет превышать величины $\bar{t}$. При этом сначала максимально нагружается текущий выбранный процессор, а лишь затем происходит переход к загрузке заданий следующего. Если в какой-то момент при попытке распределить очередное задание на процессор происходит превышение величины $\bar{t}$, то сначала производится попытка распределения на тот же процессор оставшихся нераспределённых заданий вплоть до наименее трудоёмкого (также с не превышением величины $\bar{t}$) и только потом – переход на загрузку следующего процессора (для каждого $j = 1, ..., N$ определять $x_{ij} = 1$ до тех пор, пока $\sum_{i=1}^{M} x_{ij} t_i < \bar{t}$ для данного *j*).

г) Определить номер k^* процессора, для которого суммарная длительность заданий, назначенных на него, минимальна, т.е.

$$k^* = \arg(\min_{k=1,N} \sum_{i=1}^{M} x_{ij} t_i) \qquad (2)$$

д) Распределить на этот процессор самое длинное из ранее не распределённых заданий.

е) Повторять выполнение пунктов (г) и (д) до тех пор, пока не все задания распределены.

7.1.2. Контрольный алгоритм 1

Для сравнительного контроля качества получаемых предложенными алгоритмами решений используется известный «жадный» алгоритм распределения заданий. Как уже упоминалось, он состоит в том, что распределение осуществляется, начиная с самого трудоёмкого из необработанных заданий, на наименее загруженный на момент распределения процессор.

7.1.3. Контрольный алгоритм 2

Для оценки качества получаемых решений при малых размерностях задачи использовался и метод полного перебора вариантов распределения заданий по процессорам.

Этот алгоритм хорошо известен и не требуется его описание в подробностях. Оценка его трудоёмкости составляет величину порядка $O(N^M)$.

Таким образом, оценки трудоёмкостей обсуждаемых эвристических алгоритмов определяются трудоёмкостью предварительной сортировки и, в зависимости от выбранного метода, составляют либо $O(M^2)$, либо $O(M \log M)$, соответственно. Оба вида сортировки используются в связи с тем, что при малых M ($M \leq 14$), трудоёмкость сортировки методом пузырька оказывается ниже, чем сортировки разделением-слиянием [23].

7.1.4. Сравнительные итоги расчётов по алгоритму 1 с разными процентами калибровки

При испытаниях предложенного алгоритма входные данные для программы формировались следующим образом: количество процессоров и заданий задавалось автором, а длительность заданий формировалось с использованием стандартного генератора случайных чисел (функция C++ *random* ()).

Опыты показали, что при приближении соотношения общего числа заданий и числа процессоров к 8 эффективность применения мультиоценочного алгоритма снижалась. При существенном превышении соотношения числа заданий к числу процессоров указанной величины применение мультиоценочного алгоритма ста-

новилось не целесообразным. Более подробные данные представлены в следующих *рис.* 7.1 и *таблице* 7.1.

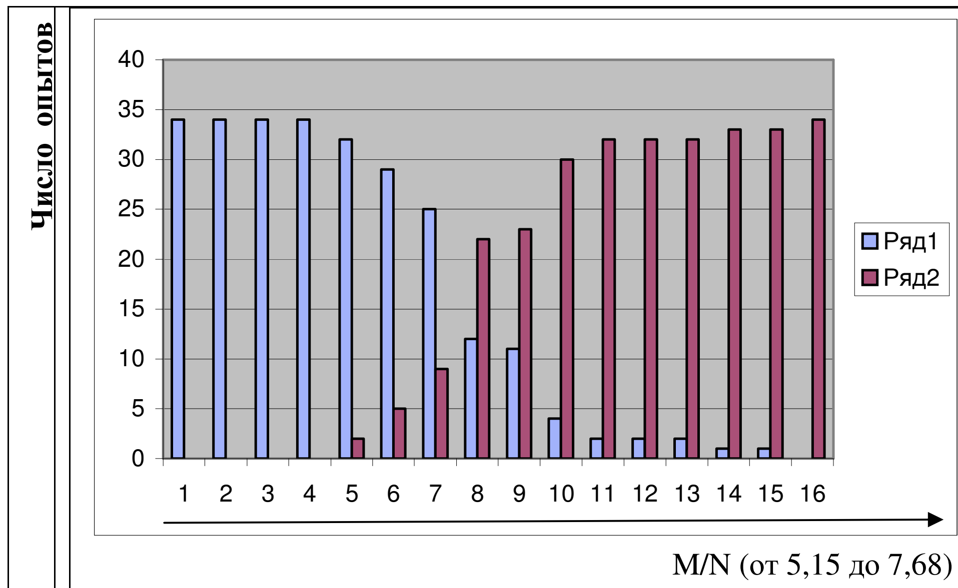


Рис. 7.1. *Результаты ряда экспериментов с эвристическим алгоритмом 1 в графическом представлении.*

Ряд 1 (■) – эвристический алгоритм 1 лучше, чем «жадный».

Ряд 2 (■) – результаты совпадают.

Таблица 7.1. *Сравнение результатов работы мультиоценочного и жадного алгоритмов для процессоров одинаковой производительности.*

Входные данные			*Число запусков, в которых мультиоценивание по сравнению с жадным алгоритмом:*	
Число процессоров	*Число заданий*	*Общее число запусков*	*Лучше*	*Хуже*
33	88	340	334	0
33	108	340	340	0
33	158	340	322	18
33	208	340	179	161
33	308	340	12	328

33	350	340	2	338
33	400	340	0	340
33	512	340	0	340
33	3120	34	0	34
330	700	340	266	11
330	1000	340	340	0
330	1700	34	34	0
330	2200	34	34	0
330	2300	34	34	0
330	2350	34	34	0
330	2400	34	32	2
330	2450	34	29	5
330	2470	34	25	9
330	2480	34	12	22
330	2490	34	11	23
330	2500	34	4	30
330	2510	34	2	32
330	2520	34	2	32
330	2530	34	2	32
330	2533	34	1	33
330	2534	34	1	33
330	2535	34	0	34
330	2540	34	0	34
330	2600	34	0	34
330	3300	34	0	34

7.2. Решение задачи 6.2 (для случая различных процессоров)

7.2.1. Описание жадного алгоритма

Пусть L_j – длина временного интервала загруженности *j*-го процессора.

0. Положить $L_j = 0 \;\; \forall\, j = \overline{1, N}$.
1. Назначить первое задание на первый процессор и $L_1 := Q_1 / S_1$.

2. Для каждого $i=\overline{2,M}$ выполнять шаги 3-5.
3. Положить $R_j = \max(L_1, L_2, \ldots, L_{j-1}, (L_j + Q_i / S_j), \quad L_{j+1}, \ldots, L_N)$ для $\forall\ j=\overline{1,N}$
4. Вычислить $R_{j_0} = \min\limits_{j=\overline{1,N}} R_j$.
5. Назначить задание *i* на процессор j_0: $L_{j_0} := L_{j_0} + Q_i / S_{j_0}$.

7.2.2. Описание эвристического алгоритма 2

При работе эвристического алгоритма вычисляется нижняя оценка $t^* = (\sum\limits_{i=1}^{M} Q_i) / \sum\limits_{j=1}^{N} S_j$ времени завершения всей совокупности заданий, а также её калиброванное значение $\bar{t}$. Под калибровкой понимается увеличение величины t^* на некоторое значение, причём алгоритм запускается с разными значениями калибровки и выбирается лучшее достигнутое для данного набора входных данных решение. В качестве значения верхней оценки T_G длины расписания используется время выполнения всей совокупности заданий, полученное «жадным» алгоритмом для соответствующих данных.

Алгоритм распределения заданий

1) Вычислить величины $t^*,\ T_G$. Пусть *K* – заданное число (значение параметра). Алгоритм запускается для следующих значений $\bar{t}$:

$$\bar{t} = t^* + \left(\frac{t_G - t^*}{K}\right) \cdot h,\ h = \overline{0,K}.$$

2) Распределять на каждый процессор *j*, начиная с первого, нераспределённые ранее задания до тех пор, пока суммарная длительность заданий на каждом процессоре не будет превышать величины $\bar{t}$. При этом сначала максимально загружается текущий выбранный процессор, а лишь затем происходит переход к загрузке заданий следующего. Если в какой-то момент при попытке распределить очередное задание на процессор происходит превышение величины $\bar{t}$, то сначала производится попытка распределения на тот же процессор оставшихся нераспределённых заданий вплоть до наименее трудоёмкого (также с не превышением величины $\bar{t}$) и только потом – переход на загрузку следующего процессора (для ка-

ждого $j = 1, ..., N$ определять $x_{ij} = 1$ до тех пор, пока $\sum_{i=1}^{M} x_{ij} \frac{Q_i}{S_j} < \bar{t}$ для данного j).

Для каждого j вычислить величину L_j.

3) Назначить оставшиеся ранее нераспределённые задания согласно пп. 3-5 «жадного» алгоритма.

7.2.3. Вычислительные эксперименты и рекомендации к применению

При испытаниях предложенного алгоритма входные данные для программы формировались следующим образом. Количество процессоров в каждом эксперименте было равно 64. Соотношения производительностей самого быстрого к самому медленному процессору изменялось в разных опытах от 2 до 64. Число заданий изменялось от 64 до 640. Для каждого набора входных данных проводилось 340 экспериментов со значениями объёмов работ, полученных с помощью программного генератора случайных чисел, позволяющего получать псевдослучайные числа с равномерным распределением на отрезке [1, 2341]. Величина K, определяющая величину шага калибровки, бралась равной 5, 10 и 15.

Опыты показали (см. табл. 7.2 и рис. 7.2), что при приближении соотношения общего числа заданий и числа процессоров к 7 эффективность применения мультиоценочного алгоритма снижалась. Более подробные данные представлены в следующей таблице, в которой использованы следующие обозначения:

n – номер серии экспериментов; M – количество заданий;

$D(S) = \frac{\max_{j=\overline{1,N}} S_j}{\min_{j=\overline{1,N}} S_j}$ – максимальное соотношение производительностей самого быстрого и самого медленного процессора в данной серии экспериментов;

$\tau_{средн.}$– средний процент улучшения длины расписания, полученного мультиоценочным алгоритмом, по сравнению с «жадным» алгоритмом;

$\tau_{\max}$ – максимальный процент улучшения длины расписания, полученного мультиоценочным алгоритмом, по сравнению с «жадным» алгоритмом;

P – процент числа экспериментов, в которых расписание, полученное мультиоценочным алгоритмом, оказалось лучше расписания, полученного «жадным» алгоритмом.

Таблица. 7.2. *Сравнение результатов работы мультиоценочного и «жадного» алгоритмов для случая процессоров разной производительности (N=64).*

n	*M*	*D(S)*	$\tau_{средн.}$, %	τ_{max}, %	*P* %
1	64	2	34.30	78.28	100,00%
		4	17.895	51.132	77,35%
		8	3.918	17.209	76,76%
		16	2.130	10.582	69,41%
		32	1.219	4.776	69,71%
		64	0.587	2.633	59,71%
2	128	2	27.735	53.096	100,00%
		4	3.375	17.199	86,47%
		8	5.114	10.192	95,00%
		16	4.174	10.765	90,59%
		32	3.638	9.401	81,18%
		64	3.540	8.771	81,47%
3	192	2	13.511	38.149	100,00%
		4	12.049	27.597	100,00%
		8	5.115	12.271	100,00%
		16	2.113	9.986	97,94%
		32	1.815	4.900	96,18%
		64	1.587	4.367	94,12%
4	256	2	17.341	41.521	99,71%
		4	6.877	12.789	96,76%
		8	3.563	7.004	95,00%
		16	1.700	3.332	84,41%
		32	0.887	3.303	82,35%
		64	0.745	2.447	77,94%
5	320	2	13.421	28.163	89,41%
		4	6.832	17.671	79,12%
		8	4.939	9.420	82,35%
		16	1.853	3.140	47,65%
		32	0.883	1.560	30,59%
		64	0.460	1.255	30,59%

6	384	2	12.278	24.084	37,94%
		4	8.029	11.980	74,41%
		8	4.054	7.441	37,65%
		16	1.832	2.324	7,94%
		32	0.980	1.164	3,24%
		64	0.467	0.507	3,53%
7	448	2	13.563	20.795	32,06%
		4	7.109	9.346	4,71%
		8	3.829	4.845	2,06%
		16	2.140	2.140	0,29%
		32	0.557	0.557	0,29%
		64	0	0	0,00%
8	512	2	14.058	18.455	7,35%
		4	9.193	13.391	9,41%
		8	5.435	5.491	0,59%
		16	3.550	3.973	0,88%
		32, 64	0	0	0,00%
9	576	2	15.189	16.489	0,88%
		4	8.036	8.101	0,59%
		8	5.952	7.235	0,88%
		16	3.033	3.336	2,35%
		32, 64	0	0	0,00%

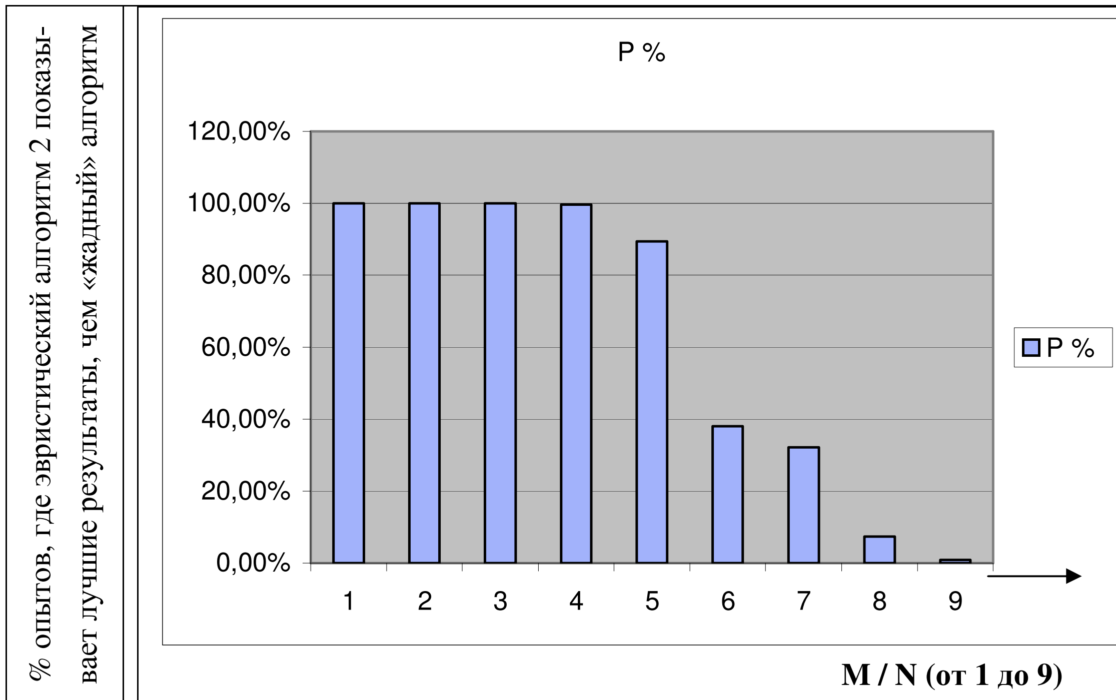


Рис. 7.2. *Графическое представление результатов ряда экспериментов с эвристическим алгоритмом 2.* $(\max \frac{P_{j_1}}{P_{j_2}} = 2 \quad \forall\ j_1, j_2 \le N)$.

Подводя итоги сравнения результатов работы мультиоценочного и «жадного» алгоритмов для случая процессоров разной производительности отметим, что:

1. С используемыми механизмами калибровки мультиоценочный алгоритм целесообразно применять при соотношении числа заданий к числу процессоров, не превосходящем 7.

2. В дополнение к мультиоценочному алгоритму следует использовать также и «жадный» алгоритм, поскольку оба алгоритма работают достаточно быстро.

Заключение

Основные результаты диссертации заключаются в следующем:

1. **Для однопроцессорных персональных ЭВМ** типа IBM PC создан инструментальный программный комплекс автоматизации проектирования систем реального времени «СРВ-Конструктор» для обработки периодически поступающей информации.

Для реализации этого комплекса разработаны:

- **Язык реального времени**, служащий для описания идущих в реальном времени процессов и предоставляющий возможность синхронизировать производимые вычисления с моментами поступления информации извне – приходами кадров данных.
- ***Блок синтаксического и семантического анализа,*** осуществляющий синтаксический и семантический анализ конструкций программы реального времени, описывающей на формальном языке необходимый порядок выполнения прикладных программ пользователя, генерирующего таблицы данных для работы последующих блоков и вычисляющего размеры буферов обмена данными между программными модулями.
- ***Блок генерации сетевой модели и расписаний*****,** строящий математическую модель вычислений, выполняемых в реальном времени, в виде графа, в котором вершины соответствуют прикладным модулям пользователя, а дуги определяют частичный порядок их выполнения, определяющего директивные интервалы выполнения прикладных модулей, возможность построения допустимого расписания выполнения прикладных модулей и само расписание, если оно существует, а также выполняющего некоторые другие вспомогательные функции построения инструментальной САПР ВСРВ.
- ***Блок генерации кода***, формирующий на языке С и записывающий в текущий каталог исходные тексты получившейся программы, а также создающий ряд вспомогательных файлов для последующих определённых действий пользователя, компиляции и редактирования связей.
- ***Управляющий монитор*** на основе многозадачной оболочки реального времени CTask-RT, обеспечивающего работу в реальном времени приклад-

ной программы пользователя, сгенерированной на этапе предварительной обработки посредством САПР ВСРВ.

2. **Для многопроцессорного варианта** САПР «СРВ-Конструктор» разработаны два новых эвристических алгоритма распределения M заданий на N процессоров, основанных на мультиоценке результата. Первый алгоритм предназначен для процессоров с одинаковой производительностью. Второй – для случая, когда производительность процессоров может различаться. Проведены многочисленные машинные эксперименты, показавшие, что предложенные алгоритмы лучше известного жадного алгоритма в определённой области входных данных по точности получаемого решения. Даны рекомендации к эффективному применению данных алгоритмов.

Литература

1. *Танаев В.С., Гордон В.С., Шафранский Я.М.* Теория расписаний. Одностадийные системы. – М.: Наука, 1984.
2. *Барский А.Б.* Параллельные процессы в вычислительных системах. Планирование и организация. – М.: Радио и связь, 1990.
3. *Головкин Б.А.* Расчёт характеристик и планирование параллельных вычислительных процессов. – М.: Радио и Связь, 1983.
4. *Логинова И.В., Сушков Б.Г.* Динамическое распределение памяти в системах реального времени при имеющемся расписании центрального процессора // В сборнике «Теория и реализация систем реального времени», ВЦ АН СССР, стр. 49-69, 1984.
5. *Луганский И.Л., Сушков Б.Г.* Язык программирования детерминированных систем реального времени. М.: ВЦ АН СССР, 1986.
6. *Сушков Б.Г.* ЭВМ управляет экспериментом. (Вычислительные системы реального времени.) // Новое в жизни, науке, технике. Сер. «Математика, кибернетика». – М.: Знание, 1987, N 9.
7. *Шкурба В.В., Селивончик В.М.* Расписания, имитационное моделирование и оптимизация. – Кибернетика, 1981, № 1, с. 91-96.
8. *Костенко В.А., Смелянский Р.Л., Трекин А.Г.* Синтез структур вычислительных систем реального времени с использованием генетических алгоритмов// Программирование, 2000, №5.
9. *Лазарев А.А., Гафаров Е.Р.* Теория расписаний. Минимизация суммарного запаздывания для одного прибора. М.:ВЦ РАН, 2006
10. *Мищенко А.В.* Устойчивость решений в задаче оптимального распределения ресурсов. Автореферат дисс. на соиск. уч. ст. канд.физ.-мат. наук. М.:ВЦ АН СССР, 1987.
11. *Подчасова Т.П., Португал В.М. и др.* Эвристические методы календарного планирования. – Киев: Техника, 1980.
12. *Сигал И.Х.* Задача коммивояжёра большой размерности // ВЦ АН СССР, 1986.
13. *Сигал И.Х.* Приближённые методы и алгоритмы в дискретной оптимизации // ВЦ РАН М.: Изд-во ВЦ РАН, 2000.

14. *Сигал И.Х., Иванова А.П.* Введение в прикладное дискретное программирование // ФИЗМАТЛИТ. 2002.
15. *Burns A.* Scheduling Hard Real-Time Systems: A Review, Software Engineering Journal, 6(3): 116-128, May 1991.
16. *Bruker P.* Scheduling Algorithms. Springer, 2001, 365 p.
17. *Gonzales T., Sahni S.* "Preemptive Scheduling of Uniform Processor Systems". Journal of the Association for Computing Machinery, Vol. 25, No. 1, January 1978.
18. *Federgruen A., Groenevelt H.* "Preemptive Scheduling of Uniform Machines by Ordinary Network Flow Technique". Management Science Vol. 32, No. 3, March 1986.
19. *Гэри М., Джонсон Д.* Вычислительные машины и трудно решаемые задачи // М.: Мир, 1982.
20. *Dertouzos M.L.* Control Robotics: The Procedural Control of Physical Processes, Information Processing 74, North-Holland Publishing Company, 1974.
21. *Conway R., Maxwell W., Miller L.* Theory of Scheduling. // Addison Wesley Publishing Company, 1967.
22. *Конвей Р.В., Максвелл В.Л., Миллер Л.В.*«Теория расписаний», М.: Наука, 1975.
23. *Кормен Т., Лейзерсон Ч., Ривест Р.* «Алгоритмы: построение и анализ» М. МЦНМО, 1999.
24. *Коффман Е. Дж., Грэхем Р.Л.* Теория операционных систем // М.:Наука, 1984.
25. *Коффман Э.Г.* Введение в детерминированную теорию расписаний. В кн.: Теория расписаний и вычислительные машины / Под ред. Коффмана Э.Г. – М. Наука, 1984, с. 9-64.
26. *Liu C.L. and Layland J.W.*, Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment, Journal of the ACM, 20(1): 46-61, Jan 1973.
27. *Martel C.* Preemptive scheduling with release times, deadlines and due times // Journal of the ACM. 1982. V. 29, №3. P. 812-829.
28. *Mok A.K.* Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment, Ph.D Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massa-

chusetts, 1983.

29. *Audsley N., Burns A., Richardson M. and Wellings A.* Hard Real-Time Scheduling: The Deadline Monotonic Approach, IEEE Workshop on Real-Time Operating Systems, 1992.
30. *Ramamritham K. and Stankovic J.A.*, Scheduling Algorithms and Operating Systems Support for Real-Time Systems, Proceedings of the IEEE, 82(1): 55-67, Jan 1994.
31. *Stankovic J.A.,* et. al., Implications of Classical Scheduling Results for Real-Time Systems, IEEE Computer Society Press, 1995.
32. *Ульман Дж.* Полиномиально полные задачи составления расписаний // Operating Systems Review, 1973.
33. *Wagner T.* CTask. A Multitasking Kernel for C. V1.1, V2.0, Programer's manual, Berlin: 1988, 1989.
34. *Бездушный А.Н., Лютый В.Г., Серебряков В.А.* Разработка компиляторов в системе СУПЕР. М.: ВЦ АН СССР, 1991.
35. *Сушков Б.Г., Фуругян М.Г., Гончар Д.Р. и др.* Система автоматизации программирования вычислительных систем реального времени. - В сб. "Теория и реализация вычислительных систем реального времени". М.: ВЦ РАН, 1999, 8 с.
36. Microsoft C. Advanced programming techniques, 1990.
37. *Held M., Karp R.* A dynamic programming approach to sequencing problem. // SIAM, 1962.
38. *Корбут А.А., Финкельштейн Ю.Ю.* Дискретное программирование // М. Наука. Гл. ред. физ.-мат. лит. 1969.
39. *Растригин Л.А.* Статистические методы поиска // М.: Наука, 1968.
40. *Гончаров Е.Н., Кочетов Ю.А.* Вероятностный поиск с запретами для дискретных задач безусловной оптимизации // Дискрет. анализ и исслед. операций. Сер. 2, 2002. Т. 9, №2. С. 13-30.
41. *Кочетов Ю., Младенович Н., Хансен П.* Локальный поиск с чередующимися окрестностями // Дискрет. анализ и исслед. операций Сер. 2, 2003. Т. 10, № 1, С. 11-44.
42. *Land A.H., and Doig A.G.* An automatic method of solving discrete programming problems. // Econometrica. v. 28 (1960), pp. 497-520.

43. *Little J.D.C., Murty K.G., Sweeney D.W., and Karel C.* An algorithm for the traveling salesman problem. // Operations Research. v11 (1963), pp 972-989.
44. *Алексеев О.Г.* Комплексное применение методов дискретной оптимизации // Наука, 1986.
45. *Киселёв В.Д., Карелин Д.В.* Метод ветвей и границ для решения задач целочисленного квадратичного программирования с булевыми переменными // Известия Тульского Государственного Университета, 1995, Том 1, выпуск 3, Информатика.
46. *Малков У.Х.* О реализации методов Вульфа и ветвей и границ для решения частично-целочисленных задач квадратичного программирования // Тезисы доклада на научной конференции "Математическое программирование и приложения", Екатеринбург: ИММ Уральского научного центра РАН, 1996.
47. *Panwalkar S., Iskander W.* A survey of scheduling rules. // Operations Research. 25(1):45-61, 1977.
48. *Гуз Д.С.* Разработка точных и приближённых алгоритмов составления расписаний и синтеза систем жёсткого реального времени // Диссертация на соискание учёной степени канд. физ.-мат. наук, - М., 2005.
49. *Штовба С. Д.* Муравьиные алгоритмы // ExponentaPro. Математика в приложениях, № 4(4), 2003.
50. *Laarhoven P., Aarts E., Lenstra J.* Job Shop Scheduling by Simulated Annealing // Operations Research, 40(1):113-125, 1992.
51. *Shen C., Pao Y., Yip P.* Scheduling multiple job problems with guided evolutionary simulated annealing approach // Proceedings of the First IEEE Conference on Evolutionary Computations, pages 702-706, 1994.
52. *Трекин А.Г.* Структурный синтез вычислительных систем с помощью генетических алгоритмов // Диссертация … канд. физ.-мат. наук, - М., 2002. – 111 с.
53. *Holland J.N.* Adaptation in Natural and Artificial Systems // Ann Arbor, Michigan: Univ. of Michigan Press, 1975.
54. *Michalewicz Z.* Genetic Algorithms + Data Structures = Evolution Programs // Third, Revised and Extended Edition, Springer, 1999.
55. *Goldberg D.E.* Genetic Algorithms in Search Optimization & Machine Learning // Addison Wesley, Reading, 1989.

56. *Mataric M., Cliff D.* Challenges in Evolving Controllers for Physical Robots // Robotics and autonomous systems, 19(1), p. 67-83, 1996.
57. *Periaux J. and Winter G. editors.* Genetic Algorithms in Engineering and Computer Science // John Wiley & Sons Ltd., 1995.
58. *Corne D., Fang H.-L., Mellish C.* Solving the Modular Exam Scheduling Problem with Genetic Algorithms // DAI Research Paper №622.
59. *Bierwirth C., Kopfer H., Mattfeld D.C., Rixen I.* Genetic Algorithm based Scheduling in a Dynamic Manufacturing Environment // Proceedings of the IEEE Conf. on Evolutionary Computation, Perth, IEEE Press, 1995, p.439-443.
60. *Fang H.-L., Ross P., Corne D.* A Promising genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems //Proceedings of the Fifth International Conf. on Genetic Algorithms, S. Forrest (ed.), San Mateo: Morgan Kaufmann, 1993, p.375-382.
61. *Daaldr J., Eklund P.W., Ohmori K.* High-Level Synthesis Optimization with Genetic Algorithms // Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence, Cairns (Australia), 26-30 August 1996, p.276-287.
62. *Tsurkov V.I.,* Large-Scale Optimization – Problems and Methods // Dordrecht, Boston, London: istent Kluwer Acad. Publ., 2001.
63. *Красовский Д.В., Фуругян М.Г.* Агрегирование в задаче составления оптимального расписания для многопроцессорных АСУ // Автоматика и телемеханика. – 2006. – №12 – С. 205-212.
64. *Красовский Д.В., Фуругян М.Г.* Псевдополиномиальные алгоритмы упорядочения работ без прерываний по произвольным процессорам // Вестник Московского университета, сер. 15, Вычислительная математика и кибернетика, 2006, № 4, – С. 25-29.
65. *Красовский Д.В.* алгоритмы решения задачи составления оптимального расписания без прерываний // Диссертация на соискание учёной степени канд. физ.-мат. наук, - М., 2007.
66. *Альберт А.* Регрессия, псевдоинверсия и рекуррентное оценивание. М., Наука: 1977.
67. *Rust B., Burrus W.Q., Schneeborder C.* A simple algorithm for computing the generalized inverse of matrix. ACM. 1966, V.9.
68. *Сушков Б.Г., Фуругян М.Г., Гончар Д.Р., Кондратьев О.Л.* Методология и

программные средства мониторинга чрезвычайных ситуаций. //Тез. докл. конф. "Проблемы управления в чрезвычайных ситуациях", Москва, ИПУ РАН, 1992. С. 62-63.

69. *Сушков Б.Г., Фуругян М.Г., Гончар Д.Р. и др.* САПР систем реального времени на базе IBM PC "СРВ-Конструктор" //Тез.докл. III научной школы "Автоматизация создания мат. обеспечения и архитектуры систем реального времени". Саратовский филиал Ин-та машиноведения РАН, Саратов,1992. С. 3-4.
70. *Гончар Д.Р.* САПР систем реального времени для IBM PC (сборник трудов). М., ВЦ РАН, 1993. С. 83-87.
71. *Сушков Б.Г., Фуругян М.Г., Гончар Д.Р. и др.* САПР вычислительных систем реального времени для IBM PC. // Тез. межд. конф. "Проблемы регионального и муниципального управления". Москва: РГГУ, 1999, 1 с.
72. *Сушков Б.Г., Фуругян М.Г., Гончар Д.Р. и др.* Система автоматизации программирования вычислительных систем реального времени. - В сб. "Теория и реализация вычислительных систем реального времени". М.: ВЦ РАН, 1999, 8 с.
73. *Сушков Б.Г., Белый Д.В., Фуругян М.Г., Гончар Д.Р., Кондратьев О.Л. и др.* Автоматизация программирования вычислительных систем реального времени. //В сб.: Моделирование обработки информации и процессов управления. М.: МФТИ, 2000, 10 с.
74. *Gonchar D., Fourougyan M.* The decision of one problem of distribution of resources at presence of uncertain factors. // Proceedings of III Moscow International Conference on Operations Research / ВЦ РАН. – М., 2001, 1 с.
75. *Гончар Д.Р., Фуругян М.Г.* Автоматизация проектирования систем реального времени для испытаний и мониторинга сложных технических объектов. - Материалы 9-й межд. конф. "Проблемы управления безопасностью сложных систем", М., ИПУ РАН, 2001, 4 с.
76. *Гончар Д.Р., Фуругян М.Г.* Алгоритмы анализа и синтеза многопроцессорных систем реального времени. Тез. конф. "Математические модели сложных систем и междисциплинарные исследования". М:ВЦ РАН, 2002. 1 с.
77. *Гончар Д.Р.* Эвристические алгоритмы распределения заданий в многопроцессорной системе реального времени. Труды XLV науч. конф. МФТИ (ГУ),

2002 г., часть VII, М.: МФТИ, 2002. 1 с.

78. *Гончар Д.Р., Гуз Д.С., Красовский Д.В., Фуругян М.Г.* Эффективные алгоритмы планирования вычислений в многопроцессорных системах. // Проблемы управления безопасностью сложных систем: Труды 10-й международной конференции. Книга 2. /ИПУ РАН. – М., 2002 – С. 207-211.
79. *Gonchar D.* Multibounds Heuristic Algorithm for Design of Scheduling M Tasks on N Equal Processors. М.:Макс Пресс (ВМиК МГУ), 2004 г. С. 97-98.
80. *Гончар Д.Р.* Мультиоценочный эвристический алгоритм распределения *M* заданий на *N* процессоров// Моделирование процессов управления. М.:МФТИ, 2004 – С. 170-173.
81. *Гончар Д.Р.* Мультиоценочный эвристический алгоритм распределения M заданий на N одинаковых процессорах // В сб. Процессы и методы обработки информации. М.: МФТИ, 2005. 3 с.
82. *Гончар Д.Р.* Мультиоценочный эвристический алгоритм распределения M заданий на N процессоров. // II Всероссийская научная конференция «Методы и средства обработки информации». М.: МГУ, 2005. С. 537-539.
83. *Гончар Д.Р.* Мультиоценочный эвристический алгоритм распределения M заданий на N процессоров. // Сообщения по прикладной математике. М.: ВЦ РАН, 2005, 16 с.
84. *Гончар Д.Р.* Мультиоценочный алгоритм решения минимаксной задачи составления расписания // Системы управления и информационные технологии № 1.3 (27), 2007. Москва-Воронеж: Научная книга, 2007. С. 324-328.
85. *Фуругян М.Г., Гончар Д.Р.* Мультиоценочный алгоритм решения минимаксной задачи составления расписания // Проблемы управления безопасностью сложных систем: Труды XV-й международной конференции. Часть 2. /ИПУ РАН. – М., 2007 – С. 149-153.
86. *Фуругян М.Г., Гончар Д.Р.* Мультиоценочный алгоритм решения минимаксной задачи составления расписания. /Сборник научных трудов – Моделирование процессов обработки информации. М.: МФТИ, 2007. С. 202 – 209.

Приложение 1. Пример применения САПР «СРВ-Конструктор». Решение в реальном времени задачи многомерной линейной регрессии

В качестве демонстрационного примера применения САПР «СРВ-Конструктор» рассматривается задача оценки метода наименьших квадратов, а именно тот её вариант, когда поток поступающих данных образует временную последовательность [66]. В настоящем варианте метода [67] используется вся накопленная к данному моменту информация и оценка метода обновляется в момент поступления новой информации.

Для определённости рассматривается задача линейного регрессионного анализа.

Пусть вектор-столбец $Z_n = (z_1, \ldots, z_n)$ – последовательность скалярных наблюдений вида

$$z = H^T \cdot X + v$$

в точках $H \in R_k$, $V_n = \{v_1, \ldots, v_n\}$ – последовательность некоррелированных случайных величин с нулевым математическим ожиданием и одинаковыми дисперсиями при всех наблюдениях и $X \in R_k$ – неизвестный вектор.

Оценка для X задаётся формулой

$$X_n = A_n^+ \cdot Z_n,$$

где A_n^+ – матрица, псевдообратная к A_n:

$$A_n = \begin{Vmatrix} H_1 \\ H_2 \\ \ldots \\ H_n \end{Vmatrix}$$

При поступлении $(n+1)$-го наблюдения z_{n+1} точке H_{n+1} оценка для вектора X_{n+1} задаётся аналогичной формулой:

$$X_{n+1} = A_{n+1}^+ \cdot Z_{n+1}.$$

Значение X_{n+1} связано с X_n рекуррентной формулой

$$X_{n+1} = X_n + YK_{n+1} \cdot (z_{n+1} - H_{n+1}^T \cdot X_n).$$

Здесь YK_{n+1} является вектором сглаживания и не зависит от наблюдений Z_{n+1}, но существенно зависит от свойств матрицы A_n и новой точки наблюдения H_{n+1}. В случае если ранг матрицы A_n меньше k, вектор H_{n+1} может оказаться линейно зависимым или независимым относительно всех предыдущих векторов H_1, ..., H_n.

В зависимости от указанных свойств матрицы A_n и H_{n+1} формулы для вычисления YK_{n+1} различны.

1. Пусть ранг матрицы A_n меньше k. Обозначим $AC_n = (I - A_n + A_n)$, и тогда $AC_n = 0$.

Если H_{n+1} является независимым ($AC_n \cdot H_{n+1} = 0$), то вычисление YK_{n+1} проводится по формуле

$$YK_{n+1} = (AC_n \cdot H_{n+1}) / (H_{n+1}^T \cdot AC_n \cdot H_{n+1}).$$

Существует рекуррентное соотношение, связывающее AC_{n+1} и C_n:

$$AC_{n+1} = AC_n - (AC_n \cdot H_{n+1}) \cdot (AC_n \cdot H_{n+1})^T / (H_{n+1}^T \cdot AC_n \cdot H_{n+1}).$$

Если H_{n+1} является зависимым, то, естественно, ранг матрицы A_{n+1} остаётся меньше k, но теперь $AC_n \cdot H_{n+1} = 0$. Тогда вычисления для YK_{n+1} следует производить по формуле

$$YK_{n+1} = BC_n \cdot H_{n+1} / (1 + H_{n+1}^T \cdot BC_n \cdot H_{n+1}).$$

Здесь $BC_n = (A_n^T \cdot A_n)^+$, и существует рекуррентное соотношение, связывающее BC_{n+1} и BC_n:

$$BC_{n+1} = BC_n - [(BC_n \cdot H_{n+1}) \cdot (AC_n \cdot H_{n+1})^T + (AC_n \cdot H_{n+1}) \cdot (BC_n \cdot H_{n+1})] \; /$$

$$(H_{n+1}^T \cdot AC_n \cdot H_{n+1}) + [(1 + H_{n+1}^T \cdot BC_n \cdot H_{n+1}) \cdot (AC_n \cdot H_{n+1}) \cdot (AC_n \cdot H_{n+1})^T \,] \; /$$

$$(H_{n+1}^T \cdot AC_n \cdot H_{n+1})^2.$$

2. Пусть ранг матрицы A_n равен k. Естественно, что H_{n+1} всегда является зависимым, и вычисления YK_{n+1} проводятся по соответствующей приведённой выше формуле. Однако в этом случае рекуррентное соотношение, связывающее BC_n и BC_n, упрощается:

$$BC_{n+1} = BC_n - (BC_n \cdot H_{n+1}) \cdot (BC_n \cdot H_{n+1})^T \ / \ (1 + H_{n+1}^T \cdot BC_n \cdot H_{n+1}).$$

Начальное значение BC_k вычисляется, если используется свойство для псевдообратных матриц: $BC_k = (A_k^T \cdot A_k)^+ = A_k^T \cdot (A_k^+)^T$.

В приведённой схеме предполагается, что осуществляется первоначальное накопление k измерений и затем требуется однократное вычисление матриц A_k^+.

Здесь вычисление A_k^+ проводилось усовершенствованным методом Грамма-Шмидта, описанным в [67]. К сожалению, программа в [67] реализует лишь частный случай метода, поэтому нам пришлось переработать её на общий случай.

Следует отметить, что в приведённых рекуррентных формулах не используются явно матрицы A_n и A_n^+, и поэтому они не нужны. Таким образом, упоминание о ранге матрицы A_n является условным.

В демонстрационном примере использовались следующие основные вычислительные модули:

1) APZ – образование исходной квадратной матрицы A_k;
2) GIN – вычисление A_k^+, AC_k, BC_k, X_k;
3) KAF – (rk(A) < k) – вычисление X_n, AC_n, BC_n;
4) KBF – (rk(A) ≤ k) – вычисление X_n, BC_n;
5) HGA – вычисление свойства вектор H_{n+1} и соответствующего ему логического предиката.

Прочие модули NG, NPI, NPR имеют вспомогательное функциональное назначение.

Настоящая PB-программа поименована как REGRES_2. Программа состоит из трёх условных заданий (УЗ), причём первое УЗ (COMPUTE_RANK) содержит два простых задания (ПЗ) (FILL_MATR, INVERT_MATR), два других УЗ (N_FULL_RANK, FULL_RANK) – по одному ПЗ (INCR_RANK, QUALITY_COEF). В начале программы проводится описание констант, описание кадра, описание глобальных переменных. Отметим основные параметры примера: число независимых переменных k = 5, число поколений при графическом отображении коэффициента регрессии NGEN = 8. При описании кадра периодически с периодом 5000 мс кадры данных поступают через порт COM2 и состоят из массива H и скаляра Z_1. В примере предусмотрен ввод сообщений с клавиатуры (мо-

дуль KBD с числовым идентификатором 2). В описании глобальных переменных использованы те же самые имена, что и в описании содержательной части примера. Отметим только вспомогательные переменные IND и ARNN, используемые для графического отображения.

Далее приводится список модулей с описанием их формальных параметров, указанием на верхние оценки времени выполнения и язык программирования, на котором эти модули написаны.

Содержательно эти модули выполняют следующие вычисления:

1) APZ – на основании поступающих кадров формируется квадратная матрица A;

2) NR – вспомогательный модуль, вырабатывающий логическое условие формирования A;

3) GIN – вычисление псевдообратной для квадратной матрицы A и первой оценки коэффициентов регрессии X. Кроме того, для реализации рекуррентных вычислений в других модулях здесь вычисляются рабочие переменные AC и BC;

4) HGA – вычисление свойств очередного массива H; в соответствии со значением $AC \cdot H$ определяется линейная зависимость H от всех предыдущих поколений H и вырабатывается соответствующее логическое условие;

5) KAF – вычисление массива коэффициентов регрессии X и вспомогательных массивов AC и BC, если ранг условной (см. замечание выше) матрицы A_n меньше K;

6) KBF – вычисление массивов X и BC, если ранг условной матрицы A_n равен K;

7) NPI – счётчик числа поступивших кадров.

Следующие модули реализуют отображение трассировки задачи значений параметров и их графическое представление на экране монитора:

8), 9) INIT_SCR1, INIT_SCR2 – отображение на экране монитора пустых окон с заголовками;

10) SCR_SCR1 – отображение значений кадровых параметров в окне;

11) SCR_SCR2 – отображение последних пяти поколений коэффициентов регрессии X в окне;

12) INIT_GRF – отображение окна для графика поведения уточняющихся значений коэффициентов X из восьми поколений значений (переменная NGEN);

13) GRF – отображение бегущего графика значений одного коэффициента, номер которого задаётся с клавиатуры;

14) NAKOP – подготовка данных для графика коэффициентов (NGEN поколений).

15) KBD_MOD – ввод сообщений с клавиатуры и преобразование его в нужный формат (целое число);

16) PRF – распечатка очереди *q* в фоновом режиме;

17) EQ_BYTE – запись значений логического параметра *L1* в очередь *q*.

Далее идёт описание условных и простых заданий.

Условное задание COMPUTE_RANK начинается с описания вектора условий, состоящего из трёх компонент и таблицы переключений и двух ПЗ. В первом ПЗ (FILL_MATR) до начала PB-цикла выполняется модуль, подготавливающий пустое окно для выдачи кадровых переменных. Затем по мере прихода кадров проводится заполнение матрицы *A* (модуль APZ) кадровыми переменными и отображение кадра в подготовленное окно (SCR_SCR1). Модуль NR следит за количеством записанных в матрицу *A* кад*р*ов. Выполнение модулей SCR_SCR1 и APZ проводится псевдопараллельно, модуль NR выполняется после них. После того как в матрице *A* остаётся незаполненной одна строка, модуль NR вырабатывает условие переключения на второе ПЗ (L1=TRUE). Во втором ПЗ (IVERT_MATR) в матрицу *A* добавляется недостающая строка (кадр), после этого проводится вычисление ранга *A* и однократное инвертирование этой матрицы (модуль GIN). Если матрица *A* полного ранга, то происходит переключение на третье УЗ FULL_RANK (L2=FALSE), в противном случае – на второе УЗ FULL_RANK (L2=TRUE).

Во втором УЗ N_FULL_RANK описан флаг *flg* и очередь *q* из десяти элементов типа FIFO. Эта очередь предназначена для передачи трассировочных данных фоновому модулю prf. Единственное ПЗ осуществляет рекуррентный пересчёт коэффициентов регрессии и вспомогательных переменных *AC, BC* и *YK* (модули HGA, KAF, KBF). Модуль EQ_BYTE заносит в очередь *q* текущее значение расчётной переменной *L1*, вырабатываемой модулем HGA.

Третье УЗ FULL_RANK состоит из одного ПЗ QUALITY_COEF. Это задание осуществляет рекуррентный пересчёт набора коэффициентов регрессии и других расчётных переменных для условной матрицы A_n полного ранга (модуль

KBF). Отметим, что здесь модуль KBF потребляет логическую переменную *L4*, вырабатываемую этим же модулем на предыдущем кадре. Модуль GRF отображает график коэффициента регрессии вдвое реже (период равен 2), чем другие PB-циклы. Номер отображаемого коэффициента вводится с клавиатуры (модуль KBD_MOD).

```
rtpgm REGRES_2;
const
        NGEN = 8;
        K = 5;
        K1 = 6;
        TtL1 = '              F_R_A_M_E   D_A_T_A';
        TtL2 = 'C_O_E_F_F_I_C_I_E_N_T_S;
        ColTxt1 = 1;
        ColBck1 = 13;
        ColTxt2 = 1;
        ColBck2 = 10;
end;
frame
        port = COM2;
        timeslice = 5000 ms;
        [real h[K], real z1];
end;
extra KBD(1);
glob
        integer N = 0;
        real X [K];
        real A [K][K];
        real YK [K];
        real Z [K];
        real T [K];
        real W [K];
        real BC [K][K], AC [K][K], B [K][K];
        real D [K][K], AI [K][K];
```

```
            integer IND [K];
            real ARRN [NGEN][K];
            integer n_c = 4;
            file outfile = ‘r.reg’:’wt’;
      end;
      modules
      apz (in real [K], in real, inout integer, in integer, inout real [K] [K], inout real[K]): 1 ms, FOR;
      nr (inout integer, inout integer, inout boolean): 1 ms, FOR;
      gin (inout real [K][K], inout real [K], in integer, out integer, inout boolean, inout boolean, inout real [K][K], inout real [K][K], inout real [K][K], inout real [K][K], inout real [K], inout real [K], inout real [K][K], inout integer [K]): 10 ms, FOR;
      hga (in real [K] out boolean, inout integer, inout real [K][K], inout real [K]): 3 ms, FOR;
      kaf (in real [K], in real, inout boolean, inout boolean, inout integer, inout real [K], inout real [K][K], inout real[K][K], inout real [K], inout real [K], inout real [K][K], inout real [K]): 5 ms, FOR;
      kbf (in real [K], in real, inout boolean, inout integer, inout real [K], inout real [K][K], inout real[K], inout real [K]): 15 ms, FOR;
      np1 (inout integer): 1000 mcs, FOR;
      init_scr1 (in integer, in integer, in integer, in integer, in integer, in integer, in integer, in integer, in integer, in char []): 5 ms, C;
      init_scr2 (in integer, in integer, in integer, in integer, in integer, in integer, in integer, in integer, in integer, in char []): 5 ms, C;
      scr_scr1 (in integer, in real [], in real*): 25 ms, C;
      scr_scr2 (in integer, inout real []): 25 ms, C;
      init_grf (in integer): 1 ms, C;
      grf (in integer, in char [], in integer, in integer, in integer, in integer, in integer, in integer, in integer, in integer, inout real [NGEN][K]): 100 ms, C;
      nakop (in real [K], in integer, in integer, inout real [NGEN][K]): 20 ms, C;
      kbd_mod (inout char [], inout integer*, in integer): 10 mcs, C;
      my_sleep (in integer): 15 ms, C;
      PRF (in boolean, out file*): 5 mcs, C;
```

```
eq_byte (in boolean, out boolean*): 1 mcs, C;
end;
entry COMPUTE_RANK;
condpgm COMPUTE_RANK;
        cvector [L1=FALSE, L2=FALSE, L3=FALSE];
        switchtable
        [L1 == FALSE, L2 == FALSE, L3 == FALSE -> FILL_MATR]
        [L1 == TRUE, L2 == FALSE, L3 == FALSE -> INVERT_MATR]
        [L1 == TRUE, L2 == TRUE, L3 == FALSE -> N_FULL_RANK]
        [L3 == TRUE, others == FALSE -> FULL_RANK]
        end;
smplpgm FILL_MATR;
head
        init_scr1 (20, 48, 3, 6, ColTxt1, ColBck1, 15, 5, 1, TtL1);
end;
cvector [L1];
loop period = 1 phase = 0
        scr_scr1 (K1, h.*, z1.*);
end;
loop period = 1 phase = 0
        apz (h.*, z1.*, N, K, A, Z);
        nr (N, K, L1) after apz, scr_scr1;
end;
end FILL_MATR;
smplpgm INVERT_MATR;
        cvector [L2, L3];
        calc
                integer RANK;
        end;
loop period = 1 phase = 0
        scr_scr1 (K1, h.*, z1.*);
end;
loop period = 1 phase = 0
```

```
        apz (h.*, z1.*, N, K, A, Z);
        gin (A, Z, K, RANK, L2, L3, AC, BC, D, AI, YK, X, B, IND)
                        after apz, scr_scr1;
end;
end INVERT_MATR;
end COMPUTE_RANK;
condpgm N_FULL_COMPUTE_RANK;
cvector [l = FALSE];
switchtable
        [l == FALSE -> INCR_RANK]
        [l == TRUE -> FULL_RANK]
end;
flag
        flg = FALSE;
end;
queues
        boolean q, size = 10, type = FIFO, flag = flg;
end;
background
        PRF (q, outfile), flag – flg, prty = 15;
end;
smplgpm INCR_RANK;
        cvector [l];
calc
        boolean L1;
end;
loop period = 1 phase = 0
        scr_scr1 (K1, h.*, z1.*);
end;
loop period = 1 phase = 0
        np1 (N);
        hga (h.*, L1, K, AC, T);
        kaf (h*, z1.*, L1.* from hga, l, K, X, AC, BC, T, YK, D, W)
```

```
                    after scr_scr1;
        kbf (h*, z1.*, L1.* from hga, K, X, BC, YK, T);
        eq_byte (L1.* from hga, q);
end;
end INCR_RANK;
end FULL_RANK;
condpgm FULL_RANK;
        cvector [l = FALSE, lextr = FALSE];
        switchtable
                [l == FALSE, lextr = FALSE -> QUALITY_COEF]
        end;
handler kbd_mod (MSGAsPort, n_c, K);
smplpgm QUALITY_COEF;
  head
        init_scr2 (60, 130, 5, 5, ColTxt2, ColBck2, 14, 2, 1, TtL2);
        init_grf (1);
  end;
  tail
        cls_scr1 ();
        cls_scr2 ();
  end;
  cvector [L];
  calc
        boolean L4 = TRUE;
  end;
  loop period = 1 phase = 0
        kbf (h*, z1.*, L4.*-1 from kbf, K, X, BC, YK, T);
        nakop ((inout) X, K, NGEN, ARRN) after kbf;
end;
loop period = 1 phase = 0
        scr_scr1 (K1, h.*, z1.*);
        scr_scr2 (K, X) after kbf;
end;
```

```
loop period = 2 phase = 1
        grf ((inout) n_c, 'Coef.', K, NGEN, 42, 500, 215, 295, 10, 8, ARRN)
                after nakop;
end;
end QUALITY_COEF;
end FULL_RANK;
end REGRES_2.
```

Приложение 2. Текст программ, реализующих эвристический алгоритм распределения заданий по процессорам для многопроцессорного варианта САПР «СРВ-Конструктор»

```
//=======================================================
//  РЕШЕНИЕ ЗАДАЧИ
//  РАСПРЕДЕЛЕНИЯ ЗАДАНИЙ ПО ПРОЦЕССОРАМ
//  с использованием мультиоценки
//  Гончар Д.Р., г. Москва
//========================================================
#include <alloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
    /* Определение используемых программой наборов данных */
#define TIME_INI     "time.ini"        // Входной н/д
#define TIME_RSL     "time_e.res"      // Выходной н/д
#define ERROR   "error.out"    // Набор данных для сообщений
     /* Максимальные размерности массивов */
#define PRC_MAX   110  /* Максимальное количество процессоров */
#define WRK_MAX   210 /* Максимальное количество заданий */
    /******************* КОНСТАНТЫ *************************/
#define EVR_COD   1    // код оптимизации:
                       //   1 – трудоёмкие приоритетны

/**************** Указатели файлов *****************************/
FILE  *fp,    // Входного н/д
      *fpe,  // Н/д сообщений об ошибках
      *fpo;   // Выходного н/д
/**************** ОПИСАНИЯ ПЕРЕМЕННЫХ *****************/
int     n_prc,   /* число процессоров (указанное во входных данных) */
```

```
        n_wrk,   /* число заданий */
        r [WRK_MAX], /* Вектор признаков распределения заданий r[i] */
        fwi [WRK_MAX], /* Признак распределения i-го задания */
        fpj [PRC_MAX], /* Призн. предв. распред. на j-й процессор */
      x [WRK_MAX] [PRC_MAX], /*признаки распределения i-го задания */
                                        /* на j-й процессор                    */
        i, j, l;    /* переменные циклов */
double  max_time, min_time,
        sum [PRC_MAX],
        t[PRC_MAX],  /* Сумма времён загруж. j-го процессора */
        sum_min, sum_sum, ocenka,
        time [WRK_MAX] [PRC_MAX]; /* = { 336.54 }*/
int    fcode, /* Код завершения программы */
        r_found, first, mx_i, mn_i, sm_i, iii, mx_j, mn_j, sm_j,
        k =80, n, I_char;
char s[81];  /* буфер для ввода комментариев к входным данным */

/*************** ГЛАВНАЯ ПРОГРАММА ****************/
void main()
{             /*<<<<<<  Ввод файла TIME_INI  >>>>>>*/
 system ("cls");
 if((fp=fopen(TIME_INI,"rt"))==NULL) {
 printf ("Не возможно открыть входной н/д < %s >\n", TIME_INI);
 fprintf (fpo,  "!!! Не возможно открыть входной н/д < < < %s > > >\n", TIME_INI);
fcode = 1; }
 else {   if((fpo=fopen(TIME_RSL,"wt"))==NULL) {
   printf ("Не возможно открыть выходной н/д < %s >\n", TIME_RSL);
   fcode = 4; }
 else {   k=80;  fscanf(fp,"%d\n",&n_wrk);
   if (n_wrk > WRK_MAX) {     fcode = 2; printf
 ("!!! <Число ЗАДАНИЙ> (%d) > MAX доп-го значения (%i)\n",
                        n_wrk,WRK_MAX);
 fprintf (fpo, "!!! <Число ЗАДАНИЙ> (%d) > MAX доп-го значения (%i)\n",
```

```
 n_wrk,WRK_MAX);       }
  else     fprintf (fpo,"Число заданий = %d \n", n_wrk);
 fscanf(fp,"%d\n",&n_prc);
 if (n_prc > PRC_MAX) {  fcode = 3; printf
("!!! <Число ПРОЦЕССОРОВ> (%d) > MAX доп-го значения (%i)\n",
                              n_prc,PRC_MAX);
 fprintf (fpo,
"!!! <Число ПРОЦЕССОРОВ> (%d) > MAX доп-го значения (%i)\n",
                              n_prc,PRC_MAX);    }
  else {     fprintf (fpo,"Число процессоров = %d\n\n", n_prc);
  fprintf (fpo,  "Исходные времена выполнения заданий на процессорах");}
 fprintf (fpo, "\nПроцессоры:    ");
 for (j=0; j<n_prc; j++)    fprintf (fpo, " %i  ", j+1);
 for (i=0; i< n_wrk; i++)    {
        fprintf (fpo, "\n*Задание %i*: ", i+1);
        for (j=0; j < n_prc; j++)              {
                fscanf (fp,"%lf",&time[i][j]);
                if (time[i][j] < 0) {
                  fprintf (fpo,
                  "!!! <Одно из времён выполнения < 0>\n");
                  fcode = 5;}
                fprintf (fpo, "%3lg ",time[i][j]); }        }
 fclose(fp); }  }
 if (fcode) {
   fprintf (fpo,   "\n\n * * * Не соответствующие исходные данные * * * \n");
  fprintf (fpo,   "\n * * * Выполнение программы прервано * * * \n");
  return; }
 for (i=0; i<n_wrk; i++)  {        /* Ни на один процессор ещё */
        for (j=0; j<n_prc; j++)     /* ничего не распределено */
                x[i][j] = 0;       /* Конкретное распределение */
        fwi[i] = 0; }
for (l=0; l<n_wrk; l++) {  /* Распределение l-го задания */
  first = 1;
```

```
  for (j=0; j<n_prc; j++) {
      t[j] = 0.;
      fpj[j] = 0;    }
     r_found = 0;
     for (i=0; ((i<n_wrk)&&(r_found != 1)); i++) {
       if (fwi[i] == 0)  {
           sum_min = sum [0]+ time [i][0];
           mn_i = i;      mn_j = 0;
           r_found = 1;    }                    }
     iii = mn_i;
     for (i=iii; i<n_wrk; i++)        /* Поиск min  time [i][j]  */
       if (fwi[i] == 0)            /* среди ещё не распр. заданий */
           for (j=1; j<n_prc; j++)
             if (sum[j]+time [i][j] < sum_min) {
                   sum_min = sum[j] + time [i][j];
                   mn_i = i;  mn_j = j;              }
/**** Нашли кандидата на распределение и распределяем */
 x[mn_i] [mn_j] = 1;
 fwi [mn_i] = 1;
 sum [mn_j] = sum_min; /* Сумма загруженности sm_j процессора */ }
/* ----------- Вывод в выходной набор данных результатов ---------------*/
fprintf (fpo, "\n\n Распределение заданий по процессорам. ");
fprintf (fpo, "Эвристика 1, предпочтение MIN.MIN \n");
for (j=0; j<n_prc; j++)     {  fprintf (fpo, "Процессор # %i : ", j+1);
     for (i=0; i<n_wrk; i++)
       if (x[i][j])
            fprintf (fpo,"(%i) ", i+1);
     fprintf (fpo, "\n");     }
fprintf (fpo, "\n Загруженность по процессорам \n");
sum_min = sum[0]; iii = 0;
sum_sum = 0;
  for (j=0; j<n_prc; j++)
  {  fprintf (fpo, "Процессор # %i : %lg\n", j+1, sum [j]);
```

```
    sum_sum += sum [j];
    if (sum_min < sum [j])
    {  sum_min = sum[j];
        iii = j;   }  }
  ocenka = sum_sum / n_prc;
  fprintf (fpo, "\n Полная длительность выполнения работ: %lg\n", sum[iii]);
  fprintf (fpo,
  "\n Теоретически допустимая оценка времени выполнения работ: %lg\n", ocenka);
fclose (fpo); };
```