УДК 004(4'22+054)

С.И. Чайников[1], А.С. Солодовников[2]

[1] *Харьковский национальный университет радиоэлектроники, Харьков*
[2] *Харьковский национальный медицинский университет, Харьков*

ОРГАНИЗАЦИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ ДЛЯ ЗАДАННОЙ ПРЕДМЕТНОЙ ОБЛАСТИ С ИСПОЛЬЗОВАНИЕМ МОДЕЛИ КОНЕЧНЫХ АВТОМАТОВ

Рассматривается метод построения граф-модели предметной области для проектирования архитектуры программных комплексов и организации вычислительных процессов с помощью модели конечных автоматов. При описании метода делается упор на применении модели для воссоздания структуры будущего программного комплекса, который формируется из отдельных программных компонентов, что позволяет оценить структуру программного продукта до начала разработки и обеспечить корректную связь между программными модулями. Также показаны принципы построения автоматной модели, необходимой для определения основных свойств и поведения подсистем программного комплекса.

***Ключевые слова**: граф-модель, предметная область, конечный автомат, программный комплекс.*

Введение

Для разработки конкурентоспособного программного обеспечения (ПО) важно обеспечить соблюдение основных показателей качества согласно стандарту ISO/IEC 25051:2014, оптимизировать или снизить влияние человеческого фактора на быстродействие программного средства. На основании результатов анализа требований к программному продукту разработчик делает выводы о возможности и необходимости использования специальных технологий ведения истории вычислений, архивов данных, способах их восстановления, последовательного или параллельного исполнения вычислительных процессов (ВП) для решения задач конкретной предметной области (ПрО). Для сложных ПрО разрабатываемых информационных систем (ИС) становится актуальным вопрос оптимизации действий, сокращения времени диалога пользователя с системой. Также важным является уменьшение времени решения задач, требующих повторного выполнения ВП. Примером могут служить задачи оптимизации или уточнения значений параметров математических моделей, задачи проектирования и разработки продукции машиностроения и другие сходные с ними задачи, для которых характерно совместное использование ресурсов памяти (баз данных (БД) или хранилищ), использование информации из различных территориально распределенных источников, синхронизация вычислительных процессов и доступ к общей памяти. Для таких задач можно выделить следующие характеристики:

1) возможность декомпозиции задачи на подзадачи, которые автоматизируются отдельным программным модулем, приложением или компонентой;

2) наличие подмножества функций, решение которых не зависит от выполнения других функций;

3) наличие подмножества функций, выполняемых в диалоге с пользователем и под его контролем;

4) высокая вероятность возникновения ошибок, связанная с человеческим фактором, и появление в этой связи необходимости корректировки исходных данных и пересчета результата;

5) необходимость тестового режима решения подзадачи;

6) потенциальная возможность распараллеливания реализации функций.

Одним из возможных подходов к решению вопроса восстановления или отката ВП является «backtracking». Метод, предложенный в работе [1], заключается в использовании истории взаимодействий для анализа программы. Для хранения историй используется историческая переменная – массив значений, в который заносится информация о передаче для соответствующего канала передачи данных. Массивы обеспечивают процесс восстановления истории работы программы. Также существует подход, основанный на создании контрольных точек (КТ) вычислений (checkpoint), который подразделяется на два подхода: реактивный (reactive) и проактивный (proactive) [2]. Однако, применение механизма КТ связано с накладными расходами при выполнении параллельных программ. Данная операция характерна интенсивным использованием узлов ввода-вывода, поэтому среднее время создания КТ может быть значительным. По классам поддерживаемых программ средства создания контрольных точек (ССКТ) подразделяются на сосредоточенные и распределённые [3 – 5].

Сосредоточенные ССКТ обеспечивают отказоустойчивость выполнения одного или нескольких процессов в рамках вычислительного узла вычисли-

тельной системы (ВС). Распределённые ССКТ обычно строятся на базе сосредоточенных и применимы для распределённых и параллельных программ, что делает их важным инструментом организации функционирования ВС. Такие распределенные контрольные точки могут использоваться для восстановления программы после сбоя.

Однако предложенные подходы не учитывают возможность извлечения только части данных из архивов КТ – то есть той информации, в которой содержатся ошибки вычислений, и которая необходима только для конкретного ВП. Также подходы не учитывают управление восстановлением ВП при взаимодействии нескольких пользователей. Это было бы возможным, например, при использовании конечных автоматов (КА), как средства управления и выделения подграфа ВП, определяющего относящиеся к конкретному ВП функции и информационные единицы.

Такая технология, очевидно, должна быть применима на этапе синтеза программной системы (ПС), на котором необходимо усовершенствовать архитектуру программного продукта с учетом результатов структурного анализа и проектирования, полученных на предшествующих стадиях ЖЦ ИС. Рассмотрим структуру граф-модели, служащей основой для организации вычислений и восстановления данных.

Принципы получения и сохранения информации, необходимой для отката вычислительных процессов

Для описания структуры ПС воспользуемся понятием ориентированного графа:

$$G = (V, X), \qquad (1)$$

где V – множество вершин v, которым сопоставляются программные модули, реализующие соответствующие ВП, а X – множество дуг $x_{ij} = (v_i, v_j)$ графа G, определяющих зависимость по данным между программными модулями и соединяющих вершины v_i и v_j между собой, для которых задано направление ($i, j = \overline{1,n}$, где n – количество вершин графа G). Вычислительные процессы являются порождением заданных программных модулей.

В случае изменения требований, определяющих модификацию архитектуры ПС и расширения функциональности программного продукта для графа G (1) формируется дополнительное подмножество новых программных модулей $V^H = \{v_i^{\,н}\}$, где $v_i^{\,н} \in V$, включаемых в структуру ПС и подмножество модифицируемых модулей $V^M = \{v_i^{\,м}\}$, где $v_i^{\,м} \in V$, чей функционал будет изменен каким-либо образом.

Для получения новой версии структуры ПС необходимо из графа G (1) исключить подмножество вершин $V^З \subseteq V$, которые требуется заместить на подмножество V^M, то есть необходимо убрать программные модули устаревшей версии. Такую же операцию проделать с дугами $X^З \subseteq X$. Это позволяет определить постоянную часть графа G, определяемую как $G^П = \left(V^П, X^П\right), G^П \subseteq G$, где $V^П = V \setminus V^З$, а $X^П = X \setminus X^З$.

Связи между добавляемыми и изменяемыми программными модулями, полученными таким образом, позволяют задать подмножества дуг $X^H = \{x_p^{\,н}\}$ и $X^И = \{x_q^{\,н}\}$ соответственно. Что, в свою очередь, позволяет выделить подграфы

$$G_1^H = \left(V_1^H, X_1^H\right),\ G_2^H = \left(V_2^H, X_2^H\right), \ldots,$$

$$G_n^H = \left(V_n^H, X_n^H\right) \text{ и } G_1^M = \left(V_1^M, X_1^M\right),$$

$$G_2^M = \left(V_2^M, X_2^M\right), \ldots,\ G_m^M = \left(V_m^M, X_m^M\right).$$

Объединяя указанные подграфы, получаем новую версию структуры ПС в виде графа:

$$G^B = \left(\bigcup_{i=1}^{n} G_i^M\right) \cup \left(\bigcup_{j=1}^{m} G_j^H\right) \cup G^П. \qquad (2)$$

Связи между вершинами граф-модели характеризуются информационными потоками, содержащими данные, которыми обмениваются программные модули между собой. Часть этих данных является константами, служащими начальной информацией для начала работы модуля. Другая часть является изменяемыми массивами, структурами и переменными, которые претерпевают свои изменения в ходе работы программы. Поэтому целесообразно для каждой i-й вершины выделить два вектора данных:

$$D_i^{in} = \left\{d_1^{in}, d_2^{in}, \ldots d_j^{in}\right\}, j = \overline{1,k} \qquad (3)$$

– вектор входных данных, неизменяемых в процессе работы и

$$D_i^{out} = \left\{d_1^{out}, d_2^{out}, \ldots d_j^{out}\right\}, j = \overline{1,l} \qquad (4)$$

– вектор выходных данных, которые были модифицированы в процессе работы программного модуля. Будем считать, что первая вершина граф-модели является фиктивной по своей сути. Также будем полагать, что первая вершина имеет только вектор выходных данных (рис. 1).

Между множествами данных, которыми оперируют программные модули, могут быть разного рода зависимости. Их классификацию следует рассмотреть на примере.

Рассмотрим фрагмент графа вида (1), состоящего из пяти вершин (рис. 1, а, б).

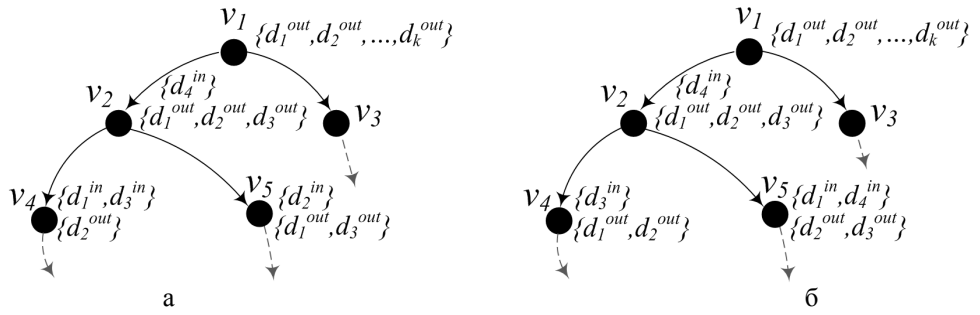

Рис. 1. Пример маркировки вершин граф-модели векторами входных и выходных данных

При этом в соответствии с процедурой формирования ярусно-параллельной формы (ЯПФ) можно установить для первого уровня только первую начальную вершину, для второго – вершины v_2 и v_3, для третьего – вершины v_4 и v_5. Если рассматривать информационные зависимости вершин, то можно получить различные между ними отношения. Так, например, в одном случае (рис. 1, а) вершины v_4 и v_5 используют различные входные наборы переменных, и $D_4^{in} \subseteq D_2^{out}$, $D_4^{out} \subseteq D_2^{out}$, $D_5^{in} \subseteq D_2^{out}$, $D_5^{out} \subseteq D_2^{out}$. А в другом случае (рис. 1, б) также справедливо выражение $D_5^{in} \cup D_2^{in} \neq \varnothing$. То есть для следующей после v_i вершины v_{i+1}, при условии существования ориентированной дуги $x_{i,i+1} = (v_i, v_{i+1})$, существует входной набор данных $D_{i+1}^{in} \subseteq D_i^{in} \cup D_i^{out}$.

Но выходное множество данных четвертой вершины является входным множеством для пятой вершины. Если модуль, соответствующий вершине v_4 закончит свою работу до запуска модуля, соответствующего вершине v_5, то значение на входе пятой вершины будет отличаться от того, какое могло бы быть, если бы на вход пришли данные из вершины v_2 непосредственно до окончания работы вершины v_4 и применения изменений выходных переменных. Такая проблема может существовать в том случае, если при проектировании архитектуры ПС используются ссылки на одни и те же массивы данных, а не их локальные копии. Для вершин, которые находятся на одном ярусе, возможно одно из трех состояний: 1) $D_4^{in} \cup D_5^{out} \neq \varnothing$; 2) $D_4^{out} \cup D_5^{in} \neq \varnothing$ или 3) $D_4^{out} \cup D_5^{int} = \varnothing$ и $D_4^{in} \cup D_5^{out} = \varnothing$. Последнее состояние характерно для вершин, независящих друг от друга по данным. В более общем случае эти выражения будут записаны следующим образом.

Пусть задан ациклический граф G вида (1), приведенный к ЯПФ. Соответственно, для такого графа будет существовать L ярусов. Обозначим подмножество вершин, находящихся на r-м ярусе как $V_r \subset V$, где $r = \overline{1,L}$ – номер текущего яруса ЯПФ графа. Тогда для всех $v_i \in V_r$ и $v_k \in V_r$ справедливо выражение:

$$\forall v_i, v_k \in V_r, x_{ik} = (v_i, v_k) \notin X. \quad (5)$$

И выполняется одно из следующих условий:

$$D_i^{in} \cup D_k^{out} \neq \varnothing, \quad (6)$$

$$D_i^{out} \cup D_k^{in} \neq \varnothing, \quad (7)$$

$$\begin{cases} D_i^{in} \cup D_k^{out} = \varnothing, \\ D_i^{out} \cup D_k^{in} = \varnothing. \end{cases} \quad (8)$$

Отношения между входными и выходными данными вершин определяются согласно с принятой классификацией типов отношений между регионами параллельного или распределенного программного кода [6]. Существуют четыре типа отношений: одновременность, упорядоченность, консервативность и последовательность (табл. 1).

Таблица 1

– Типы отношений между параллельными фрагментами программы

Тип	Описание
Одновременность	Программный модуль v_i и v_k могут выполняться одновременно и обращаться к используемым ячейкам памяти в произвольном порядке
Упорядоченность	Программный модуль v_i должен выбрать все, что ему требуется, прежде, чем модуль v_k запишет свои результаты.
Консервативность	Программный модуль v_i должен записать свои результаты раньше, чем модуль v_k
Последовательность	Программный модуль v_i должен быть завершен до начала работы модуля v_k

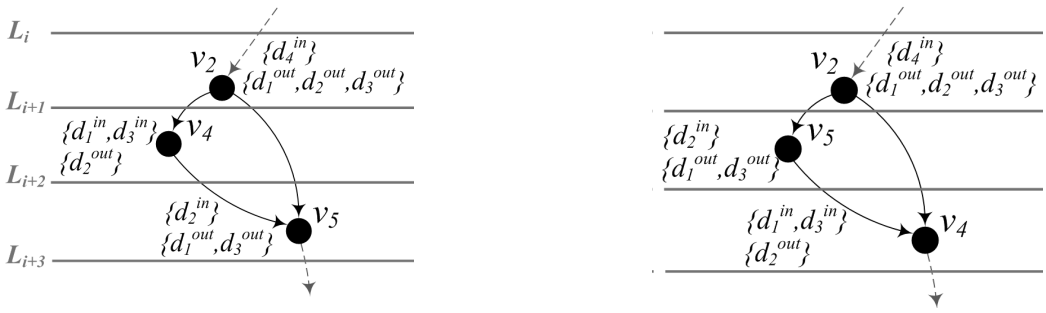

Рис. 2. Пример размещения вершин граф-модели по ярусам ЯПФ граф-модели в зависимости от набора входных и выходных данных

Если при анализе графа обнаруживаются свойства (6) или (7), то это говорит о таком типе отношений как упорядоченность или консервативность, и этот тип уточняется в дальнейшем при рассмотрении условий работы программных модулей проектировщиком системы.

Если же обнаруживается соблюдение свойства (8), то в этом случае вершины граф-модели действительно могут быть размещены на одном ярусе и назначаться диспетчером исполнения ВП к независимому и параллельному запуску.

Для правильного распределения вершин по ярусам с учетом возможности или невозможности параллельного запуска программных модулей для вершин со свойствами (6) и (7) добавляется новый ярус, позволяющий размежевать эти вершины. Например, для графа G (рис. 1) вершины v_4 и v_5 распределяются по двум отдельным ярусам в случае упорядоченного исполнения вершины v_4 по отношению к вершине v_5 (рис. 2, а) или в случае консервативного исполнения (рис. 2, б).

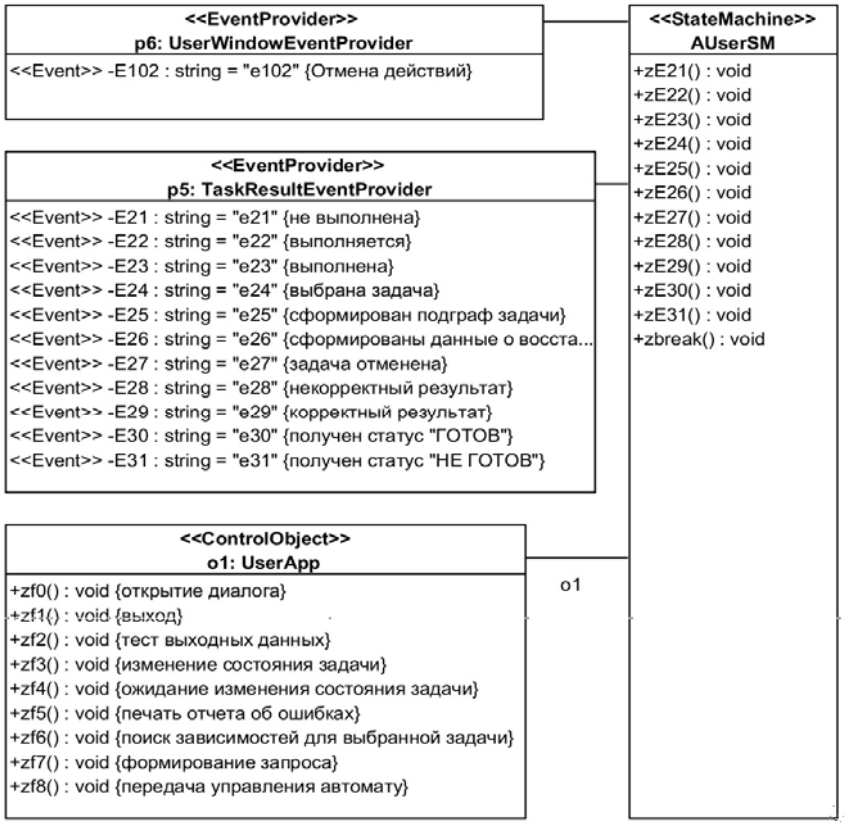

Рис. 3.Обобщенная структура исполнителя в виде управляющего конечного автомата

Такая маркировка вершин граф-модели, а также хранение этой информации позволяет установить информационные зависимости между вершинами при восстановлении истории вычислений и избежать сбоев и блокировок доступа к данным при выполнении процесса backtracking.

Принимая за основу предложенную граф-модель, разработчик использует ее для определения отдельных функций и архитектурных решений, позволяющих эти функции реализовать. Исполнение ВП, имеющих место для заданной таким образом модели ПрО, должно выполняться централизовано с

тем, чтобы обеспечить согласованность действий подсистем ПК и его пользователей, контролировать архивирование результатов и восстановление действий ВП. Для этого управляющий объект (УО), как диспетчер или исполнитель ВП, целесообразно реализовать на основе модели КА. Рассмотрим процесс проектирования модели КА Мура более детально.

Моделирование исполнителя вычислительных процессов

Зададим КА Мура формально:

$$A = \{S, X, Y, s_0, \delta\}, \quad (9)$$

где S – конечное непустое множество состояний; X – конечное непустое множество входных сигналов (входной алфавит); Y – конечное непустое множество выходных сигналов (выходной алфавит); $s_0 \in S$ – начальное состояние; $\delta : S \times X \to S$ – функция переходов.

Учитывая заявленные требования к исполнителю для объектно-ориентированных программ в качестве базового представления схемы связей автоматов выберем диаграмму классов (рис. 3) и определим основные события переходов для автоматной модели ПК.

Основным автоматом (рис. 3) является автомат верхнего уровня AUserSM**.** Данный автомат управляет работой остальных вложенных автоматов. Основным объектом управления для него служит пользовательское приложение UserApp, реализующее диалог с пользователем и ведущее его через все этапы функций, исполнение которых приводит к решению поставленной задачи. Также на данной диаграмме заданы основные события, генерируемые объектом управления в ходе вычислений. Так объекты «EventProvider» описывают возможные события, по которым срабатывают возможные переходы автомата Мура (9), в частности, объект «UserWindowEventProvider» характеризует события, поступающие в следствие действий пользователя, а объект «TaskResultEvenProvider» – определяет события, возникающие при смене состояний выполняемого ВП. В свою очередь «ControlObject.UserApp» характеризует ОУ и описывает основные функции, которыми он управляет. На основании диаграммы классов основного управляющего автомата зададим диаграмму переходов (рис. 4).

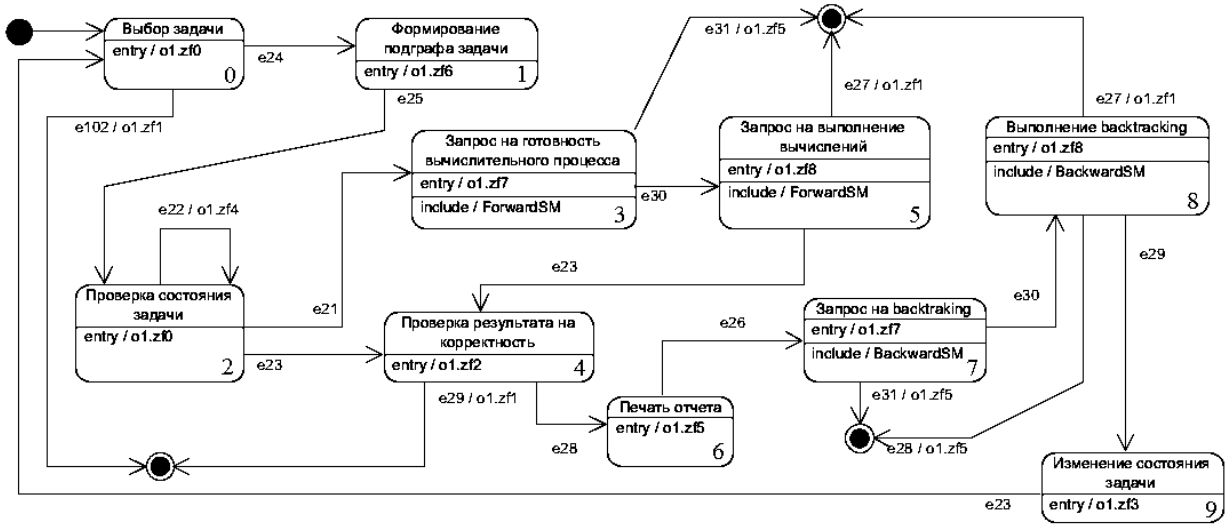

Рис. 4. Диаграмма переходов для управляющего конечного автомата

На этой диаграмме видно, что состояние «Запрос на готовность ВП» и состояние «Запрос на выполнение вычислений» являются составными и описываются с помощью вложенного автомата AForwardSM. Также составные состояния «Запрос на backtracking» и «Выполнение backtracking» реализуются при помощи вложенного автомата ABackwardSM. Два вложенных автомата не действуют параллельно, а выполняются только тогда, когда один из них неактивен (рис. 4). Для того, чтобы КА работал, необходимо хранить в памяти состояния вершин граф-модели (1) и отображать на ней все происходящие изменения. Для этого необходимо сопоставить с каждой вершиной $v_i \in V$ графа G (1) атрибут m_i, который определяет дополнительные подмножества для текущей вершины:

$$m_i = \{ST, AR\}, \quad (10)$$

где $ST = \{0,1,2\}$ – это подмножество состояний ВП, соответствующего i-й вершине, а $AR = \{0,1,2\}$ – подмножество характеристик архивных данных. Другими словами, если задать переменную e^{st}, которая принимает значения из подмножества ST и переменную e^{ar}, принимающую значения из подмножества AR, то

$$e^{st} = \begin{cases} 0, \text{если ВП не запущен;} \\ 1, \text{если ВП запущен;} \\ 2, \text{если ВП выполнен;} \end{cases}$$

$$e^{ar} = \begin{cases} 0, \text{если архив отсутствует;} \\ 1, \text{если запущен процесс архивации;} \\ 2, \text{если архив существует.} \end{cases}$$

Множество функций, сопоставляемых с вершинами, определяется таким образом:

$$D = \left\{ p_1^D, p_2^D, ..., p_q^D \right\}, \quad (11)$$

где $$p_i^D = \left\{ \begin{matrix} UID, PName, PMName, \\ PUPath, PDes, SPath \end{matrix} \right\} - \quad (12)$$

подмножество элементов, необходимых для описания программных компонент или модулей, которые эти функции выполняют.

При этом UID – уникальный идентификатор ВП; PName – имя процесса, используемое компоновщиком, при определении логической связи между вершиной граф-модели и программным модулем; PMName – реальное имя программного модуля (или библиотеки); PUPath – физический путь к файлу программного модуля; SPath – физический путь к файлу спецификации, определяющей основные характеристики модуля, и является ссылкой на спецификацию модуля; PDes – дополнительные сведения об элементе. Полученная модель позволяет скорректировать обнаруженные ошибки функциональности ПК.

Выводы

Рассмотрен метод построения граф-модели ПрО для проектирования архитектуры ПК и организации ВП с помощью модели КА, который может быть использован как дополнение к существующим передовым информационным технологиям разработки с целью усовершенствования процесса проектирования и комплексирования ПК.

Граф-модель ПрО позволяет формализовать отношения между элементами системы и описать их свойства спецификациями программных модулей.

Также показаны принципы построения автоматной модели, определяющей поведение подсистем ПК, на основании которой проводится верификация программы.

Такой подход позволяет сформулировать основные свойства ПС, скорректировать ошибки в архитектуре и функциональности на стадии проектирования и обеспечить управление ВП для сложных ПрО.

Список литературы

1. Малышкин В.Э. Параллельное программирование мультикомпьютеров / В.Э. Малышкин, В.Д. Корнеев – Новосибирск: Новосибирский государственный технический университет, 2006. – 452 с.

2. Proactive fault tolerance for HPC with Xen virtualization / A.B. Nagarajan, F. Mueller, C. Engelmann, S.L. Scott // ICS 2007: Proceedings of the 21st Annual International Conference on Supercomputing. – ACM, New York, 2007. – P. 23-32.

3. Ansel J. DMTCP: Transparent Checkpoint for Cluster Computations and the Desktop / J. Ansel, K. Arya, G. Cooperman // Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS`09). IEEE Press. –2009. – P. 1-12. [Электронный ресурс]. – Режим доступа к ресурсу: http://dmtcp.sourceforge.net/papers/ dmtcp.pdf.

4. The design and implementation of checkpoint/restart process fault tolerance for Open MPI / J. Hursey, J.M. Squyres, T.I. Mattox, A. Lumsdaine // In Proceedings of the 21st IEEE International Parallel and stributed Processing Symposium (IPDPS). – IEEE Computer Society. – 2007. – Vol. 3. – №26. – P. 1-8.

5. Application-transparent checkpoint/restart for MPI programs over InfiniBand / Q. Gao, W. Yu, W. Huang, D.K. Panda // Parallel Processing, Jan 2006. – 2006. – P. 1-8. [Электронный ресурс]. – Режим доступа к ресурсу: http://mvapich.cse.ohio-state.edu:8080/static/media/ publications/abstract/gaoq-icpp06.pdf.

6. Бакулев А.В. Модели и алгоритмы организации мобильных параллельных вычислений в среде многоядерных процессоров: автореф. дис. ... канд. техн. наук: 05.13.11 / Бакулев А.В. – Рязань: Рязанский государственный радиотехнический университет,– 2011. – 20 с.

Поступила в редколлегию 22.12.2015

Рецензент: д-р техн. наук, проф. В.А. Филатов, Харьковский национальный университет радиоэлектроники, Харьков.

ОРГАНІЗАЦІЯ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ ДЛЯ ЗАДАНОЇ ПРЕДМЕТНОЇ ОБЛАСТІ З ВИКОРИСТАННЯМ МОДЕЛІ КІНЦЕВИХ АВТОМАТІВ

С.І. Чайніков, А.С. Солодовніков

Розглядається метод побудови граф-моделі предметної області для проектування архітектури програмних комплексів та організації обчислювальних процесів за допомогою моделі кінцевих автоматів. При описі методу робиться наголос на застосуванні моделі для відтворення структури майбутнього програмного комплексу, який формується з окремих програмних компонентів, що дозволяє оцінити структуру програмного продукту до початку розробки і забезпечити коректний зв'язок між програмними модулями. Також показані принципи побудови автоматної моделі, яка необхідна для визначення основних властивостей і поведінки підсистем програмного комплексу.

Ключові слова: *граф-модель, предметна область, кінцевий автомат, програмний комплекс.*

ORGANIZATION OF COMPUTATIONAL PROCESSES FOR SPECIFIED PROBLEM DOMAIN WITH FINITE STATE MACHINE MODEL

S.I. Chaynikov, A.S. Solodovnikov

The authors propose the method of graph model development for specified problem domain used in design of software complex architecture and in organization of computational processes by finite state machine model. Authors emphasize using model in order to reconstruct the structure of bundled software formed from separate software components that gives ability to judge architecture and correct modules linking by formalized characteristics before starting programming. Also authors describe main construction principles for finite state machine model needed to define basic behavior properties of software subsystems.

Keywords: *graph model, problem domain, finite state machine, bundled software.*