УДК 004.05

Метод масштабируемой верификации PROMELA-моделей протоколов когерентности кэш-памяти

В.С. Буренков[1], А.С. Камкин[2]

[1]АО «МЦСТ», burenkov_v@mcst.ru

[2]Институт системного программирования РАН, kamkin@ispras.ru

***Аннотация* — В статье рассматривается метод масштабируемой верификации моделей протоколов когерентности кэш-памяти, описанных на языке PROMELA. Под масштабируемостью понимается независимость затрат на верификацию от размера модели (по сути, от числа процессоров в проверяемой системе). Метод состоит из трех этапов. Сначала PROMELA-спецификация протокола, выполненная для фиксированной конфигурации системы, обобщается до спецификации, параметризованной числом процессоров (для этого используются некоторые предположения об устройстве протокола и простые правила индукции). Затем выполняется построение абстрактной спецификации, независимой от числа процессоров (это делается путем синтаксических преобразований спецификации). Наконец, построенная абстрактная спецификация верифицируется инструментом SPIN обычным образом. Метод успешно применялся для верификации протоколов семейства MOSI, реализованных в вычислительных комплексах «Эльбрус».**

***Ключевые слова* — многоядерные микропроцессоры, мультипроцессоры с общей памятью, протоколы обеспечения когерентности кэш-памяти, проверка моделей, SPIN, PROMELA.**

I. Введение

Мультипроцессоры с общей памятью (*SMP, Shared memory Multi-Processor*) — базовый класс высокопроизводительных вычислительных систем. В последнее время SMP-системы получили новое развитие в форме *многоядерных микропроцессоров*, объединяющих несколько процессоров (*ядер*) на одном кристалле [1]. Первые многоядерные микропроцессоры появились в 2000-х гг.: POWER4 и POWER5 (IBM, 2001 и 2004 гг.), UltraSPARC IV (Sun, 2004 г.), Pentium D (Intel, 2005 г.), Opteron и Athlon 64 X2 (AMD, 2005 г.). В настоящее время серийно производятся 8- и 16-ядерные микропроцессоры; ведущие компании анонсировали разработку 48-, 80- и даже 100-ядерных чипов. Ожидается, что количество ядер будет удваиваться каждые два года [1]. Разработкой многоядерных микропроцессоров и мультипроцессорных комплексов занимаются и российские компании, включая АО «МЦСТ» и ПАО «ИНЭУМ им И.С. Брука»: МЦСТ-R500S (2 ядра, 2008 г.), Эльбрус-2С+ (2+4 ядра, 2011 г.), МЦСТ-R1000 (4 ядра, 2012 г.), Эльбрус-8С (8 ядер, 2015 г.) [2].

Основной проблемой, возникающей при создании мультипроцессоров с общей памятью (в частности, многоядерных микропроцессоров), является обеспечение *когерентного состояния памяти*. Каждый процессор имеет в своем составе локальную кэш-память, из-за чего в системе могут сосуществовать несколько копий одних и тех же данных: одна копия в основной памяти и несколько копий в кэш-памяти процессоров. При изменении какой-либо копии другие копии должны быть либо удалены, либо изменены согласованным образом. За эту функцию отвечают так называемые *кэш-контроллеры* — устройства подсистемы памяти, объединенные в сеть и взаимодействующие друг с другом по специальному протоколу — *протоколу когерентности* [3].

Разработка механизмов обеспечения когерентности памяти осуществляется в два этапа: проектирование протокола когерентности и реализация протокола в аппаратуре. Ввиду сложности современных протоколов на обоих этапах возможны ошибки. Для их обнаружения применяются соответственно методы верификации протоколов и методы верификации аппаратуры [4]. Ошибки в протоколе когерентности особенно критичны и должны быть выявлены до начала реализации подсистемы памяти. Общепризнанным методом верификации протоколов является *проверка моделей* (*model checking*) [5]. Подход обеспечивает высокую автоматизацию верификации, но имеет принципиальный недостаток — он не масштабируется из-за *комбинаторного взрыва* числа состояний. Верификацию протоколов когерентности для систем из 4 и более процессоров невозможно (по крайней мере, проблематично) осуществить традиционными средствами [6].

В последнее время появляются работы, нацеленные на преодоление комбинаторного взрыва и создание масштабируемой технологии верификации. Основные усилия сосредоточены на *проверке параметризованных моделей* (*parameterized model checking*) [7]. Такие методы основаны на построении абстрактной модели протокола, не зависящей от числа процессоров и поддающейся верификации существующими инструментами. Корректность абстрактной модели гарантирует корректность исходного протокола; обратное, вообще говоря, неверно — при верификации абстрактной модели возможны *ложные сообщения об ошибках*. Подход, предложенный в этой статье, относится к указанному классу методов. Его основные отличия от существующих решений — поддержка языка PROMELA

(PROcess MEta LAnguage), входного языка инструмента SPIN (Simple Promela INterpreter) [8] и, что более важно, конструкций передачи сообщений. Подход был успешно использован при верификации протоколов когерентности, реализованных в вычислительных комплексах «Эльбрус» [2].

Оставшаяся часть статьи устроена следующим образом. В разделе 2 делается обзор существующих методов масштабируемой верификации протоколов когерентности памяти. В разделе 3 предлагается и обосновывается метод построения абстрактной модели протокола когерентности по его спецификации на языке PROMELA. В разделе 4 описывается опыт использования метода для верификации протокола семейства MOSI (Modified, Owned, Shared, Invalid) [3]. В разделе 5 приводится заключение, и указываются направления дальнейших работ.

II. Существующие методы

Как отмечалось во введении, классический метод *проверки моделей* хорошо подходит для верификации протоколов когерентности, оперирующих с небольшим числом кэшей (до 4) [6], однако не позволяет проверять протоколы в общем случае (для произвольного числа процессоров). Альтернативный подход — *дедуктивная верификация* — требует от пользователя предоставления *индуктивных инвариантов*, а в случае ошибки в протоколе не выдает никакой диагностической информации [9]. Большие надежды связываются с методами проверки параметризованных моделей; здесь можно выделить два направления.

Первое направление предполагает сведение задачи верификации параметризованной модели (по сути, семейства моделей) к верификации одной модели (представителя семейства). Соответствующие методы нацелены на нахождение такого числа N, что верификации модели в конфигурации из N компонентов (процессоров, кэш-контроллеров и т.п.) достаточно для доказательства ее корректности в общем случае. В работе [7] представлен метод этого типа и для некоторых протоколов доказана оценка $N = 7$. Данное значение является слишком большим и говорит о невозможности применения метода к протоколам когерентности SMP-систем промышленного уровня сложности [6].

Второе направление использует абстракцию модели (параметризованной модели), чтобы сделать пространство состояний обозримым (независимым от числа компонентов). В работе [10] предложен метод, позволяющий абстрагироваться от точного числа *однотипных компонентов* (например, от числа кэшей, в которых кэш-строка находится в заданном состоянии). Метод существенно сокращает число состояний модели, но привязка к модифицированной системе Murφ затрудняет применение подхода. Схожая идея, названная $(0, 1, \infty)$-*абстракцией счетчиков,* описывается в работах [11], [12] и [13]. Методика достаточно эффективна, однако часто приводит к слишком подробным абстрактным моделям, что делает подход неприменимым к сложным протоколам.

В работе [14] предложен общий метод *композиционной верификации*. Суть подхода состоит в замене некоторого подмножества однотипных компонентов одним абстрактным компонентом — *окружением*. Такая замена обычно приводит к возникновению ложных сообщений об ошибках, для устранения которых требуется большой объем работы. Представляет интерес развитие этого метода, ориентированное на верификацию протоколов когерентности. В работах [15]-[18] предлагается метод, основанный на синтаксических преобразованиях Murφ-модели, а также подход к уточнению абстракции на основе контрпримеров. К недостаткам данных работ можно отнести:

1) язык Murφ не поддерживает примитивов передачи сообщений, что затрудняет описание протоколов;

2) не определяется класс протоколов, допускающий верификацию предложенным методом, и ограничения на Murφ-модели;

3) упомянутые в работах инструменты не находятся в открытом доступе.

III. Предлагаемый метод

Решаемая в работе задача ставится следующим образом. Имеется PROMELA-модель протокола когерентности для некоторой конфигурации системы, то есть для фиксированного числа процессоров $n > 2$. Требуется проверить корректность данного протокола для любой конфигурации системы, то есть для всех $N \geq n$. Отметим, что для того чтобы данная постановка была корректной, необходимы правила, позволяющие переходить от конкретной модели протокола к параметризованной (см. раздел III.A).

Будем считать, что в модели используются только операторы следующих типов: **if** (выбор), **do** (повторение), **goto** (переход), = (присваивание), ! (запись в канал), ? (чтение из канала). Действия и их защита в операторах **if** и **do** всегда объединены в **atomic**-блоки (действия исполняются сразу после проверки условий без прерывания другими процессами); альтернативы вида **else** отсутствуют. В правых частях присваиваний используются только элементарные выражения: переменные и константы; в левых частях — переменные и элементы массивов (индекс массива — элементарное выражение). Элементарные логические формулы имеют вид $x = c$ или $B(ch)$, где x — переменная или элемент массива, c — константа, ch — канал, B — предикат: **empty**, **full** и т.п.

A. *Обобщение конкретной модели*

Концептуально процессы, участвующие в описании протокола, делятся на *основные* и *вспомогательные*: основные процессы однотипны, их число не ограничено; вспомогательные процессы могут быть разных типов, но их фиксированное число. Не ограничивая общности рассуждений, будем считать, что имеется ровно один вспомогательный процесс. Процессы занумерованы от 0 до N (N — параметр модели): 0 — идентификатор вспомогательного процесса; 1, …, N — идентификаторы основных процессов.

Все массивы, используемые в модели (массивы переменных и массивы каналов передачи сообщений), имеют длину N и индексируются идентификаторами основных процессов (заметим, что в PROMELA массивы индексируются, начиная с 0, однако для наглядности мы будем игнорировать этот факт, считая, что индексация начинается с 1). Для перехода от конкретной модели к параметризованной используются следующие правила:

1) всякое условие, если оно задействует массив, является либо конъюнкцией, либо дизъюнкцией однотипных условий на *все* элементы массива:
   a) формула $\varphi\{i/1\} \wedge \ldots \wedge \varphi\{i/n\}$ интерпретируется как $\forall i \in \{1, \ldots, N\}: \varphi$;
   b) формула $\varphi\{i/1\} \vee \ldots \vee \varphi\{i/n\}$ интерпретируется как $\exists i \in \{1, \ldots, N\}: \varphi$;
2) всякая цепочка операторов вида $\alpha\{i/1\}; \ldots ; \alpha\{i/n\}$ интерпретируется как цикл **for** (i: 1 .. N) { α }.

Здесь φ (α) — формула (оператор), содержащая индекс i (идентификатор процесса) в качестве свободной переменной; запись $\varphi\{i/t\}$ ($\alpha\{i/t\}$) означает результат подстановки в φ (α) выражения t вместо всех вхождений переменной i.

B. Уточнение вида модели

В работе рассматриваются протоколы когерентности, в которых исполнение запросов координируется системным коммутатором процессора-владельца данных (процессора, к чьей памяти происходит обращение). Таким образом, в модели определены два типа процессов: *proc* — кэш-контроллер процессора (основной процесс) и *home* — системный коммутатор процессора-владельца данных (вспомогательный процесс). В модели протокола, как это принято в рассматриваемой области, принимается во внимание только одна кэш-строка.

Несколько упрощая, протокол когерентности работает следующим образом. Каждый процесс *proc* может инициировать операцию над кэш-строкой, отправив запрос в процесс *home*. Процесс, инициировавший запрос, называется *запросчиком*. При получении запроса процесс *home* анализирует его и отправляет *снуп-запросы* (от англ. *snoop* — подглядывать) всем процессам *proc*, исключая запросчик. При получении снуп-запроса процесс *proc* отправляет ответ (данные кэш-строки или подтверждение о совершении действия) запросчику. Запросчик, собрав все ответы, отправляет в процесс *home* сообщение о завершении исполнения запроса. После получения такого сообщения процесс *home* может принять следующий запрос.

Заметим, что в каждый момент времени обрабатывается не более чем один запрос к кэш-строке. Таким образом, запросчик всегда определен однозначно. Будем считать, что значения разделяемых переменных (например, идентификатор запросчика, используемый для посылки ответов на снуп-запросы) устанавливаются процессом *home* при получении исходного запроса и не изменяются в процессе его обработки.

Модель устроена таким образом, что сообщения из каждого канала читает только один процесс, однако писать могут несколько: канал, в который пишет только один процесс, будем называть *простым*, а канал, в который пишут несколько процессов, — *мультиплексным*. Обозначим через $C_{S\to r}$ множество каналов, в которые пишут процессы с идентификаторами из множества S, а читает процесс с идентификатором r. Каналы модели делятся на следующие группы (для краткости в записи синглетонов опущены фигурные скобки):

1) $C_* = \cup_{j\geq 0} C_{\{1,\ldots,N\}\to j}$ — мультиплексные каналы емкости N, через которые процессы (*home* или *proc*) получают информацию от основных процессов (например канал, через который процесс *home* получает исходные запросы, и каналы, через которые процессы *proc* получают ответы);
2) $C_{h\to p} = \cup_{j\geq 1} C_{0\to j}$ — простые каналы положительной емкости (зависящей от протокола, но не зависящей от N), через которые основные процессы получают информацию от процесса *home* (например каналы, через которые процесс *home* передает снуп-запросы);
3) $C_{p\to h} = \cup_{i\geq 1} C_{i\to 0}$ — простые каналы емкости 1, через которые процесс *home* получает информацию от основных процессов (например каналы, через которые запросчики передают в процесс *home* подтверждение о завершении операции).

Сообщения, передаваемые по каналам, имеют вид (*opc*, *i*), где *opc* — код операции, а i — идентификатор процесса, отправившего сообщение.

Свойства, проверяемые на модели протокола, имеют вид $\mathbf{G}\{\forall k, l \in \{1, \ldots, N\}: ((k \neq l) \to \varphi\{i/k, j/l\})\}$, где φ — формула, параметризованная двумя индексами (i и j), характеризующая допустимые состояния кэш-строки в соответствующих процессорах. Для протоколов семейства MOSI формула φ имеет вид конъюнкции следующих условий:

1) $\neg(cache[i] = M \wedge cache[j] \neq I)$;
2) $\neg(cache[i] = O \wedge cache[j] = O)$.

Здесь *cache* — массив состояний кэш-строки в процессорах. Напомним, что **G** — *оператор*, требующий истинности свойства-аргумента во всех достижимых состояниях модели [5].

C. Описание предлагаемого метода

Предлагаемый метод заключается в построении абстрактной модели, структура которой не зависит от значения N. На концептуальном уровне метод близок к существующим подходам, но допускает более широкий класс спецификаций, в частности спецификации, содержащие конструкции приема и передачи сообщений. Построенная модель, как бывает при абстракции, допускает поведение, не свойственное исходной модели (что приводит к ложным сообщениям об ошибках), однако ситуации, когда абстрактная модель корректна, а исходная — нет (пропуски ошибок), исключены.

Метод основан на синтаксическом преобразовании (абстракции) PROMELA-модели. Описания процесс-

ных типов видоизменяются, а вместо $N + 1$ процесса порождаются четыре: один измененный процесс *home* ($home_{abs}$), два измененных процесса *proc* ($proc_{abs}$) и один процесс окружения ($proc_{env}$), представляющий все остальные процессы *proc*. Таким образом, процесс инициализации абстрактной спецификации имеет следующий вид (*ABS* — константа, отличная от 0, 1 и 2).

```
init {
  atomic {
    run home_abs(0);
    run proc_abs(1);
    run proc_abs(2);
    run proc_env(ABS);
  }
}
```

Проверяемое свойство принимает вид $\mathbf{G}\{\varphi(1, 2)\}$. Для протоколов семейства MOSI $\varphi(1, 2)$ определяется как конъюнкция следующих условий:

1) $\neg(cache[1] = M \wedge cache[2] \neq I)$;
2) $\neg(cache[1] = O \wedge cache[2] = O)$.

Над кодом спецификации совершаются следующие преобразования. Во-первых, длина всех массивов меняется с N на 2 (напомним, что массивы индексируются идентификаторами процессов *proc*). Во-вторых, каждое обращение к массиву снабжается защитой вида $i \le 2$, где i — индекс элемента массива:

1) при чтении (в условии) элементарная формула, включающая доступ к массиву, заменяется на *undef* (неопределенное значение), если индекс не пропускается защитой.

$$B(x[i], \ldots) \Rightarrow (i \le 2 \rightarrow B(x[i], \ldots) : undef)$$

В PROMELA формула $(B \rightarrow t_1 : t_2)$ соответствует условному выражению **if** B **then** t_1 **else** t_2 **fi**.

2) При записи (в присваивании) доступ к массиву обрамляется условным оператором.

$$x[i] = t \Rightarrow \textbf{if} :: \textbf{atomic}\ \{i \le 2 \rightarrow x[i] = t\} :: \textbf{else fi}$$

Присваивания в глобальные переменные и условия над глобальными переменными не изменяются.

Каналы множества $C_{h \rightarrow p}$ представляются массивом (обозначим его *ch*). Как и другие массивы, он преобразуется в массив длины 2. Каждая элементарная формула над каналом $ch[i]$, где $i > 2$, заменяется на *undef*, а операции над каналом снабжаются защитой.

Каналы множеств C_* и $C_{p \rightarrow h}$ обрабатываются иначе. Эти каналы (за исключением $C_{\{1,\ldots,N\} \rightarrow 0}$) используются для передачи сообщений от запросчика или к запросчику. Поскольку запросчик может быть только один, в модели для простоты используются одиночные каналы вместо массивов.

Итак, операторы записи в канал либо сохраняются без изменений, либо удаляются. Оператор *ch*!*m*, содержащийся в процессном типе *P*, удаляется в следующих случаях (и только в них):

1) $ch \in C_{h \rightarrow e}$ и $P = home_{abs}$ — $home_{abs}$ не шлет снуп-запросы в $proc_{env}$ ($C_{h \rightarrow e} = \cup_{j \ge 3} C_{0 \rightarrow j}$);
2) $ch \in C_*$ и $P = proc_{env}$ — $proc_{env}$ не отправляет исходные запросы и ответы на снуп-запросы.

Операторы чтения из канала могут оставаться без изменения, изменяться или удаляться. Оператор *ch*?*m*, содержащийся в процессном типе *P*, удаляется только в следующем случае:

$ch \in C_{h \rightarrow e}$,

$P = proc_{env}$ — $proc_{env}$ не получает снуп-запросы.

Преобразование осуществляется, если:

$ch \in C_*$ и $P \in \{home_{abs}, proc_{abs}\}$.

Суть преобразования состоит в замене защищенного действия **atomic** $\{B \rightarrow ch?m\}$ (в том числе *ch*?*m*, когда используется неявная защита **nempty**(*ch*)) на следующую конструкцию.

```
if
:: atomic {B' → ch?m};
:: atomic {m.opc = opc_1; m.i = ABS}
...
:: atomic {m.opc = opc_k; m.i = ABS}
fi
```

Здесь B' — результат преобразования защиты B, $opc_1, \ldots, opc_k$ — все возможные коды операций, которые могут быть отправлены по каналу *ch*.

После описанных преобразований логические формулы, содержащие *undef* (по сути, формулы сильной трехзначной логики Клини), преобразуются в формулы классической логики таким образом, что значение истинности *undef* на внешнем уровне интерпретируются как *true*. Это делается с помощью следующего преобразования F:

1) $F(\varphi) = F'(\varphi, true)$;
2) $F'(undef, T) = T$;
3) $F'(B, T) = B$, где B — атом, отличный от *undef*;
4) $F'(\neg\varphi, T) = \neg F'(\varphi, \neg T)$;
5) $F'(\varphi \circ \psi, T) = F'(\varphi, T) \circ F'(\psi, T)$, где $\circ \in \{\wedge, \vee\}$.

При преобразованиях спецификации выполняются следующие оптимизации:

1) распространение и схлопывание констант;
2) исключение мертвого кода.

Примеры простейших оптимизаций:

1) $(i \le 2) \Rightarrow true$ в $home_{abs}$ и $proc_{abs}$;
2) $(true \wedge B) \Rightarrow B$, $(false \wedge B) \Rightarrow false$;
3) **atomic** $\{true \rightarrow \alpha\} \Rightarrow \alpha$, если α не блокируется.

D. Корректность метода

Предложенный метод является корректным в следующем смысле. Пусть φ — произвольная формула, определенная на локальных данных процессов *proc*(1)

и *proc*(2). Если в исходной модели достижимо состояние, в котором истинна (ложна) формула φ, то в абстрактной модели также достижимо состояние, в котором истинна (ложна) формула φ. Соответственно, если условие когерентности (φ) истинно во всех состояниях абстрактной модели, то оно истинно во всех состояниях исходной модели.

Доказательство приведенного утверждения требует формализации языка PROMELA и правил трансформации PROMELA-моделей. Ввиду ограничений на объем статьи, здесь приводится набросок доказательства.

Пусть в исходной модели достижимо состояние s, в котором истинна формула φ. Рассмотрим произвольный путь из начального состояния s_0 в состояние $s = s_n$: $\pi = \{s_j, [i_j: \alpha_j], s_{j+1}\}_{j=0,n-1}$. Пометки на переходах имеют вид $[i_j: \alpha_j]$, где i_j — идентификатор процесса, исполняющего действие, а α_j — само действие (в PROMELA подразумевается асинхронный параллелизм — чередование процессов). Построим в абстрактной модели путь из начального состояния s'_0 в состояние, в котором истинна формула φ: $\pi' = \{s'_j, [i'_j: \alpha'_j], s'_{j+1}\}_{j=0,n-1}$. Предполагается, что у пути π' та же длина, что у π, однако возможны пустые действия — ε (начальные и конечные состояния ε-переходов совпадают).

Путь π' строится таким образом, чтобы для любого $0 \le k \le n$ локальные данные процессов *proc*(1) и *proc*(2) в состоянии s'_k совпадали с аналогичными данными в s_k. Это гарантирует истинность формулы φ в состоянии s'_n. Предположим, что префикс длины $0 \le k < n$ пути π' построен. Покажем, как можно его продлить, оставляя согласованными локальные данные процессов *proc*(1) и *proc*(2). Рассмотрим возможные варианты для пометки $[i_k: \alpha_k]$ очередного перехода.

В случае, когда $i_k \le 2$, полагается, что $i'_k = i_k$, а α'_k — результат преобразования действия α_k (говоря точнее, абстракция действия может привести либо к одному действию, возможно ε, либо к множеству альтернативных вариантов: в последнем случае в качестве α'_k берется исполнимый вариант). Заметим, что если исполнимо действие α_k, то исполнимо α'_k (преобразования могут только ослабить условия исполнимости). Требуется показать, что исполнение α'_k приводит к тем же локальным данным, что и исполнение α_k. Возможны следующие варианты:

1) если α_k есть $x = t$, то α'_k — либо ε (если x — локальная переменная процесса *proc*(i), где $i > 2$), либо $x = t$ (иначе);

2) если α_k есть $ch!m$, то α'_k — либо ε (если $i_k = 0$ и $ch \in C_{h\to e}$), либо $ch!m$ (иначе);

3) если α_k есть $ch?m$, то α'_k — либо **atomic** $\{m.opc = OPC; m.i = ABS\}$ (если $m.i > 2$ и $m.opc = OPC$), либо $ch?m$ (иначе).

Присваивания в локальные данные процессов *proc*(1) и *proc*(2) остаются в абстрактной модели без изменений. Соответственно, эти данные в исходной и абстрактной моделях либо не меняются, либо меняются одинаково. Операторы записи не меняют значений переменных, но могут привести к изменениям в будущем. Так, прием исходного запроса, вообще говоря, изменит состояние, но сделано это будет только при обработке снуп-запросов в *proc*(1) и *proc*(2); состояние при этом будет изменено согласованным образом, поскольку сообщения посылаются в неизменном виде. Сообщения, читаемые в абстрактной модели, идентичны сообщениям, читаемым в исходной модели.

В случае, когда $i_k > 2$, пометка в абстрактной модели имеет вид $[ABS: \alpha'_k]$: все процессы исходной модели с идентификаторами $i > 2$ отображаются в один процесс $proc_{env}(ABS)$ — модель окружения. Чтобы согласовать состояние управления модели окружения с состоянием управления его прообразов $\{proc(i) \mid i > 2\}$, задействуется то ограничение, что в каждый момент времени обрабатывается не более одного исходного запроса. Пусть i_{cc} — идентификатор процесса-запросчика (от англ. *current client* — текущий клиент). Если $i_k \ne i_{cc}$, то $\alpha'_k = \varepsilon$ (модель окружения не обрабатывает снуп-запросы); иначе возможны следующие варианты:

1) если α_k есть $x = t$, то α'_k — либо ε (если x — локальная переменная процесса *proc*(i), где $i > 2$), либо $x = t$ (иначе);

2) если α_k есть $ch!m$, то α'_k — либо ε ($ch \in C_*$), либо $ch!m$ (иначе);

3) если α_k есть $ch?m$, то α'_k — либо ε ($ch \in C_{h\to e}$), либо $ch?m$ (иначе).

Если $i_k \ne i_{cc}$, возможные действия — это прием снуп-запроса, его обработка и посылка ответа на него. Такие действия никак не влияют на локальные данные процессов *proc*(1) и *proc*(2). В противном случае ($i_k = i_{cc}$) в абстрактной модели можно выбрать действие, согласованное с действием исходной модели: присваивания в локальные данные остаются без изменений; удаление записи в канал компенсируется добавлением соответствующего присваивания в читающем процессе; удаляемые операторы чтения соответствуют приему снуп-запросов моделью окружения и ни на что не влияют.

IV. Опыт практического использования

Описанный в статье метод использовался для верификации протоколов семейства MOSI, реализованных в вычислительных комплексах «Эльбрус». Разработанная PROMELA-модель поддерживает обращения к памяти типов Write Back, Write Through и Write Combined. Испытания проводились на машине Intel Core i7-4771, 3.5 ГГц. Проверяемые свойства включали:

1) $\mathbf{G}\{\neg(cache[1] = M \wedge cache[2] = M)\}$;

2) $\mathbf{G}\{\neg(cache[1] = O \wedge cache[2] = O)\}$;

3) $\mathbf{G}\{\neg(cache[1] = M \wedge cache[2] \in \{O, S\})\}$.

В табл. 1 и табл. 2 показаны ресурсы, затрачиваемые на проверку свойства (1) для $n = 3$ в исходной и абстрактной моделях соответственно. Заметим, что

при $n = 3$ число процессов при абстракции не меняется: процессы *home*(0), *proc*(1) и *proc*(2) заменяются на их абстрактные версии, а *proc*(3) — на $proc_{env}(ABS)$.

Таблица 1

Требуемые ресурсы — исходная модель

Оптимизация SPIN	Число состояний модели	Объем памяти	Время проверки
Отсутствует	5.1×10^6	682 Мб	9 с
COLLAPSE	5.1×10^6	328 Мб	15 с

Таблица 2

Требуемые ресурсы — абстрактная модель

Оптимизация SPIN	Число состояний модели	Объем памяти	Время проверки
Отсутствует	2.2×10^6	256 Мб	3.7 с
COLLAPSE	2.2×10^6	108 Мб	6.2 с

Видно, что даже при $n = 3$ (при совпадении числа процессов в исходной и абстрактной моделях) число состояний абстрактной модели ощутимо меньше числа состояний исходной модели. Между тем, абстрактная модель описывает протокол в общем случае (для любого $n \geq 3$). Таким образом, задача верификации протокола для произвольной конфигурации системы сводится к проверке порядка 10^6 состояний, что требует порядка 100 Мб оперативной памяти.

V. Заключение

Для современных компьютеров характерны высокий уровень параллелизма, наличие памяти, разделяемой между процессорами, и многоуровневое кэширование данных. Важная роль в них отводится механизмам, обеспечивающим согласованное состояние кэш-копий данных, — протоколам когерентности. Ошибки проектирования и реализации таких протоколов могут приводить к серьезным последствиям (порче данных, зависанию системы), что говорит об актуальности развития соответствующих методов верификации.

Основной проблемой, возникающей при верификации протоколов когерентности, является комбинаторный взрыв числа состояний: уже для 4 процессоров верификация традиционными средствами становится невозможной. В статье предложен метод масштабируемой верификации протоколов, описанных на языке PROMELA. Подход основан на построении абстрактной модели протокола, не зависящей от числа процессоров и поддающейся верификации инструментом SPIN.

Предложенный метод отличается от других подобных подходов прежде всего тем, что поддерживает язык PROMELA, распространенный в сообществе инженеров-верификаторов. Метод успешно применялся для верификации протоколов когерентности, реализованных в вычислительных комплексах «Эльбрус».

В проведенных экспериментах абстрактная модель строилась вручную, однако уже сейчас разрабатывается инструмент с открытым исходным кодом, способный это делать автоматически. К направлениям дальнейших исследований следует также отнести создание средств автоматического уточнения модели при возникновении ложных сообщений об ошибках.

Литература

[1] Паттерсон Д., Хеннесси Дж. Архитектура компьютера и проектирование компьютерных систем. СПб.: Питер, 2012. 784 с.

[2] Ким А.К., Перекатов В.И., Ермаков С.Г. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». СПб.: Питер, 2013. 272 с.

[3] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.

[4] Камкин А.С., Петроченков М.В. Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов // Вопросы радиоэлектроники. Сер. ЭВТ. 2014. Вып. 3. С. 27-38.

[5] Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. 416 с.

[6] Буренков В.С. Анализ применимости инструмента SPIN к верификации протоколов когерентности памяти // Вопросы радиоэлектроники. Сер. ЭВТ. 2013. Вып. 3. С. 126-134.

[7] Emerson E.A., Kahlon V. Exact and Efficient Verification of Parameterized Cache Coherence Protocols // IFIP WG 10.5 Advanced Research Working Conference, 2003. P. 247-262.

[8] Holzmann, G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional. 2003. 608 p.

[9] Park S., Dill D.L. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions // Annual ACM Symposium on Parallel Algorithms and Architectures, 1996. P. 288-296.

[10] Ip C.N., Dill D.L. Verifying Systems with Replicated Components in Murphi // International Conference on Computer Aided Verification, 1996. P. 147-158.

[11] Pnueli A., Xu J., Zuck L. Liveness with (0, 1, ∞)-Counter Abstraction // 14th International Conference on Computer Aided Verification, 2002. P. 107-122.

[12] Clarke E., Talupur M., Veith H. Environment Abstraction for Parameterized Verification // Verification, Model Checking, and Abstract Interpretation, 2006. LNCS, Vol. 3855. P. 126-141.

[13] Clarke E., Talupur M., Veith H. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems // International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008. P. 33-47.

[14] McMillan K. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking // Conference on Correct Hardware Design and Verification Methods, 2001. P. 179-195.

[15] Chou C.-T., Mannava P.K., Park S. A Simple Method for Parameterized Verification of Cache Coherence Protocols // Formal Methods in Computer-Aided Design, 2004. LNCS, Vol. 3312, P. 382-398.

[16] Krstic S. Parameterized System Verification with Guard Strengthening and Parameter Abstraction // International Workshop on Automated Verification of Infinite-State Systems, 2005.

[17] Talupur M., Tuttle M.R. Going with the Flow: Parameterized Verification Using Message Flows // Formal Methods in Computer-Aided Design, 2008. P. 1-8.

[18] O'Leary J., Talupur M., Tuttle M.R. Protocol Verification Using Flows: An Industrial Experience // Formal Methods in Computer-Aided Design, 2009. P. 172-179.

A Method for Scalable Verification of PROMELA Models of Cache Coherence Protocols

V.S. Burenkov[1], A.S. Kamkin[2]

[1]JSC MCST, burenkov_v@mcst.ru

[2]Institute for System Programming of the Russian Academy of Sciences, kamkin@ispras.ru

***Keywords* — multicore microprocessors, shared memory multiprocessors, cache coherence protocols, model checking, SPIN, PROMELA.**

ABSTRACT

This paper introduces a method for scalable functional verification of cache coherence protocols described in the PROMELA language. Scalability means that verification efforts do not depend on the model size (that is, the number of processors in the system under verification). The method is comprised of three main steps. First, PROMELA specification written for a certain configuration of the system under verification is generalized to the specification parameterized with the number of processors (to do it, some assumptions on the protocol are used as well as simple induction rules). Second, the parameterized specification is abstracted from the number of processors (it is done by syntax transformation of the specification). Finally, the abstract specification is verified with the SPIN model checker in a usual way. The method has been successfully applied to verification of the MOSI protocol implemented in the Elbrus computer systems.

REFERENCES

[1] Patterson D.A., Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 2013. 800 p.

[2] Kim A.K., Perekatov V.I., Ermakov S.G. Mikroprocessory i vychislitel'nye kompleksy semejstva «El'brus» (Microprocessors and computer systems of the Elbrus familty). Saint Petersburg, Piter, 2013. 272 p.

[3] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.

[4] Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikacii realizacij protokolov kogerentnosti s ispol'zovaniem formal'nyh metodov (A system to support formal methods-based verification of coherence protocol implementations) // Voprosy radioehlektroniki. Ser. EVT. 2014. Vyp. 3. P. 27-38.

[5] Clarke E.M., Grumberg O., Peled D.A. Model Checking. MIT Press, 1999. 314 p.

[6] Burenkov V.S. Analiz primenimosti instrumenta SPIN k verifikacii protokolov kogerentnosti pamyati (An analysis of the SPIN model checker applicability to cache coherence protocols verification) // Voprosy radioehlektroniki. Ser. EVT. 2014. Vyp. 3. P. 126-134.

[7] Emerson E.A., Kahlon V. Exact and Efficient Verification of Parameterized Cache Coherence Protocols // Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, 2003. P. 247-262.

[8] Holzmann, G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional. 2003. 608 p.

[9] Park S., Dill D.L. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions // 8th Annual ACM Symposium on Parallel Algorithms and Architectures, 1996. P. 288-296.

[10] Ip C.N., Dill D.L. Verifying Systems with Replicated Components in Murphi // 8th International Conference on Computer Aided Verification, 1996. P. 147-158.

[11] Pnueli A., Xu J., Zuck L. Liveness with (0, 1, ∞)-Counter Abstraction // 14th International Conference on Computer Aided Verification, 2002. P. 107-122.

[12] Clarke E., Talupur M., Veith H. Environment Abstraction for Parameterized Verification // Verification, Model Checking, and Abstract Interpretation, 2006. LNCS, Vol. 3855. P. 126-141.

[13] Clarke E., Talupur M., Veith H. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems // 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008. P. 33-47.

[14] McMillan K. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking // Conference on Correct Hardware Design and Verification Methods, 2001. P. 179-195.

[15] Chou C.-T., Mannava P.K., Park S. A Simple Method for Parameterized Verification of Cache Coherence Protocols // Formal Methods in Computer-Aided Design, 2004. LNCS, Vol. 3312, P. 382-398.

[16] Krstic S. Parameterized System Verification with Guard Strengthening and Parameter Abstraction // 4th International Workshop on Automated Verification of Infinite-State Systems, 2005.

[17] Talupur M., Tuttle M.R. Going with the Flow: Parameterized Verification Using Message Flows // Formal Methods in Computer-Aided Design, 2008. P. 1-8.

[18] O'Leary J., Talupur M., Tuttle M.R. Protocol Verification Using Flows: An Industrial Experience // Formal Methods in Computer-Aided Design, 2009. P. 172-179.