УДК 519.686

А.В. Колчин, А.А. Летичевский, С.В. Потиенко, В.С. Песчаненко

ОБЗОР СОВРЕМЕННЫХ СИСТЕМ И МЕТОДОВ ВЕРИФИКАЦИИ ФОРМАЛЬНЫХ МОДЕЛЕЙ

Приведен обзор автоматических методов проверки правильности формальных моделей программных систем. Рассмотрены проверяемые свойства, методы редукции и современные инструментальные средства проверки моделей.

Введение

Потребности современной промышленности провоцируют рост сложности программных систем. Параллельность (возможность одновременных переходов в нескольких подпроцессах) и асинхронность (отсутствие ограничений на относительную длительность осуществления переходов, зависящую от многочисленных неконтролируемых факторов) являются естественными чертами практически всех программных комплексов, при этом многие исследователи считают их самыми сложными и запутанными (в смысле числа различных типов составляющих) созданиями человека [1]. Методы тестирования не обеспечивают исчерпывающего анализа всех возможных вариантов поведения систем, тем самым, не могут гарантировать отсутствие нарушения свойств, которыми должны обладать разрабатываемые системы. Промышленность затрачивает 40–75% трудовых ресурсов на тестирование и валидацию [2, 3]; доля дефектов в требованиях составляет 40–50% обнаруженных на стадии тестирования [4]. При разработке систем со сложной моделью поведения становится невозможным обходиться без автоматизации проверки правильности – верификации.

Верификацией называется проверка свойств формального описания модели. В качестве математических объектов моделирования систем часто используются транзиционные системы, сети Петри, конечные, временные и гибридные автоматы, алгебра процессов и др. В качестве примеров проверяемых свойств можно назвать свойства живости, безопасности, полноты и непротиворечивости. В работе Хоара сформулирована [5] наиболее грандиозная задача верификации программ, известная как «Grand challenge» – создание верифицирующего компилятора. Различают два основных подхода к формальной верификации: логический вывод (logical inference) и проверка моделей (model checking).

Логический вывод базируется на технике доказательства теорем. Для доказательства правильности программ применяются аксиоматические подходы, например методы Хоара [6] и Флойда [7]. Обычно они автоматизированы частично, используются в интерактивном режиме и требуют участия эксперта. Примеры таких систем – HOL (High Order Logic), Isabelle [8], PVS [9]. Существуют так же статические методы проверки правильности модели. Как правило, в основе таких методов лежит поиск типичных ошибок по некоторым шаблонам (например, [10]), а так же принципы синтаксического анализа переходов на предмет выявления потенциальных нарушений. К проблемам статического анализа можно отнести отсутствие проверки достижимости, вследствие чего такие методы обнаруживают либо слишком большое число ложных ошибок, либо могут пропускать фактические. Инструменты автоматической верификации на основе статического анализа применяются достаточно широко, поскольку не требуют специальной подготовки от пользователя и достаточно удобны в эксплуатации. Большинство доказавших свою эффективность на практике таких методов часто включают в состав компиляторов, на их основе усовершенствуют семантические правила языков програм-

мирования, а так же стили и стандарты разработки ПО [11].

Второй подход, проверка модели, предполагает систематическое исчерпывающее исследование математической модели [12]. Метод был впервые предложен в 1981 г. Эдмундом Кларком и его аспирантом Алленом Эмерсоном для верификации параллельных систем с конечным числом состояний. Они рассматривали глобальный граф переходов параллельной системы как конечную структуру Крипке, и предложили эффективный алгоритм для определения того, является ли данная структура моделью заданной формулы темпоральной логики. В последствии Эдмунд М. Кларк, Аллан Эмерсон и Джозеф Сифакис в 2007 году были удостоены премии Тьюринга за «их роль в развитии проверки моделей – высоко эффективную технику верификации программ, широко применяемую при разработке как программного, так и аппаратного обеспечения» [13] (в сфере информационных технологий премия Тьюринга имеет статус, аналогичный Нобелевской премии в академических науках).

Проверяемые свойства обычно задаются в виде формулы темпоральной логики. Темпоральная логика есть классическая логика, расширенная темпоральными модальностями [14]: операторы необходимости □ и возможности ◊. Оператор □ интерпретируется как "всегда" ("always"), оператор ◊ – "со временем" ("eventually"), такие операторы позволяют специфицировать события во времени без введения явного понятия времени. Традиционные логики могут специфицировать свойства, описывающие мгновенные состояния систем, темпоральные же логики применяют для проверки динамических свойств поведения моделей недетерминированных реактивных систем. Различают линейные и ветвящиеся темпоральные логики. В линейной состояние системы в каждый момент времени определяется однозначно, тогда как в ветвящейся состояние в следующий момент времени определяется недетерминированным образом и процесс может иметь несколько продолжений. Динамические свойства систем так же могут быть представлены временными автоматами, которые функционируют в реальном времени, совершая переходы из состояния в состояние. При этом переходы совершаются мгновенно, и, пока автомат находится в каком-либо состоянии, время непрерывно увеличивается. Однако автомат может находиться в состоянии до тех пор, пока не нарушены ограничения этого состояния. Временные автоматы являются обобщением конечных ω-автоматов (автоматов Бюхи и Мюллера) [15]. Автоматы Бюхи используются для моделирования конечных параллельных процессов, а проблема верификации сводится к проблеме языковых включений [16].

Теории и технике верификации, объединенной названием "model checking", посвящена огромная литература, от монографий и вводных обзоров до многочисленных слайдов университетских курсов [12, 17–32], проводятся специализированные конференции [33]. В последнее время метод широко используется для проверки свойств полноты, непротиворечивости и безопасности как программного, так и аппаратного обеспечения промышленных систем.

Важной проблемой формальной проверки моделей является проблема разрешимости. Так, при моделировании времени как непрерывной сущности, модель имеет уже бесконечное число состояний. Например, счетчиковая машина Минского уже всего с двумя счетчиками может моделировать машину Тьюринга, в которой проблема достижимости терминального состояния (останова) из некоторого инициального состояния в общем случае является неразрешимой [34]. Многие работы, посвященные проверке моделей с бесконечным числом состояний накладывают различные ограничения как на описание поведения модели, так и на проверяемые свойства, применяют техники абстракций и символьного моделирования [21–26, 28–32, 36, 37].

Основной проблемой верификации все же есть проблема комбинаторного взрыва числа состояний модели. Как правило, состояние проверяемой модели включает большое количество атрибутов

и процессов. Даже если число процессов конечно и все атрибуты могут принимать лишь конечное число значений, общее число состояний может быть очень большим. Основными источниками взрыва числа состояний являются число компонент (процессов) модели и интерливинг между ними, а так же степень недетерминизма их поведения и используемые типы данных, включающие большое число значений. Реальные системы обычно параллельны, а число состояний моделей параллельной системы растет экспоненциально с числом компонент. Так, опыт верификации промышленных систем показал, что оценочное число состояний, как правило, превышает 2^{1000}. Очевидно, что при таком количестве достижимых состояний проверка модели путем наивного перебора практически не осуществима. Решению проблемы посвящено множество различных технологий – разработаны методы частичного порядка для элиминации избыточного интерливинга, используются методы определения симметрии при проверке эквивалентности состояний, исследуются информационные зависимости для поэтапной верификации составляющих компонент, применяются техники абстракции, факторизации, аппроксимации, символьного моделирования; к процессу верификации активно привлечен пользователь (эксперт предметной области) – его «подсказки» используются для направленного поиска, построения абстракций, накладывания ограничений на пространство поиска; используются симуляторы, интерактивные режимы и пр. Однако, существующие методы имеют ряд недостатков и, к сожалению, на сегодняшний день не существует приемлемого для промышленности универсального решения данной проблемы. Неудачное упорядочение атрибутов в худшем случае экспоненциально увеличивает число OBDD узлов, известно так же, что вне зависимости от способа упорядочения, булева функция, представляющая любое из двух средних выходных значений n-битного умножителя, имеет экспоненциальную сложность OBDD-представления [38]. Верификаторы, не имеющие канонической формы состояний модели, испытывают трудности с проверкой эквивалентности, что порождает дополнительный комбинаторный взрыв либо по размеру используемой памяти, либо по времени, затраченному на сравнения состояний. Методы построения абстракций плохо автоматизируются, часто приводят к ложным результатам (т.н. false negatives или false positives), которые нуждаются в уточнениях и, как следствие, в повторных запусках экспериментов; критерии для завершения процесса уточнений, как правило, основаны на эвристиках или требуют участия эксперта. Статические методы выявления зависимостей преувеличивают фактические информационные связи, что отражается на эффективности абстракций.

Проверяемые свойства формальных моделей

Типичными распространенными ошибками в моделях большинства программных и аппаратных систем, как правило, являются: выход за пределы допустимых значений (переполнение буфера, неправильная индексация), переполнение и потеря значимости в арифметических операциях, обращение к неинициализированным атрибутам, неоднозначная реакция на воздействующие сигналы, недостижимость функциональности, тупики, зацикливание, а так же нарушение условий безопасности и живости поведения системы, сформулированных для проверки. Современные системы верификации предлагают формулировать желаемые свойства моделей в виде формул темпоральной логики [39]. При этом проверяемые свойства могут задаваться как в виде линейных, так и ветвящихся темпоральных логиках, например, таких, как:

– MTL (metric temporal logic) – пропозициональная логика с ограниченными операторами, включает темпоральные операторы *until*, *next*, *since* и *previous;*

– TPTL (timed temporal logic) –

пропозициональная логика полупорядка, которая использует только темпоральные операторы *until* и *next*;

– RTTL (real-time temporal logic) – логика с явными таймерами;

– XCTL (for explicit-clock temporal logic) – пропозициональная логика с явными таймерами, в которой временные ограничения содержат только сравнение и добавление;

– MITL (metric interval temporal logic) оперирует неотрицательными вещественными числами в качестве домена времени, интерпретируется через последовательности временных наблюдений;

– RTCTL (for real-time computation tree logic) – пропозициональная логика разветвленного времени для синхронизированных систем, является расширением CTL с точечным представлением строго монотонного реального времени;

– TCTL (timed computation tree logic) – пропозициональная логика с менее ограниченной семантикой, является расширением CTL с точечным представлением строго монотонного реального времени. Последующие версии этой логики используют интервальную семантику времени и синтаксис полупорядка с примитивами сравнения и добавления констант. Для решения задач верификации распределенных систем на основе временных логик создаются языки, позволяющие описывать временные спецификации системы. При этом подходе возникает проблема разрешимости языка, которая зависит от временного домена и операций над временными переменными [30].

Наиболее используемыми в современных системах верификации логиками ввиду их универсальности являются LTL, CTL, а так же их объединение CTL*.

LTL (Linear Temporal Logic)

LTL построен из набора пропозициональных переменных, классических логических связок $\neg$, $\vee$, $\wedge$, $\Rightarrow$ и темпоральными операторами:

- **X** для выражения следующего состояния (ne**X**t);
- **G** всегда (**G**lobally);
- **F** когда-нибудь в будущем (**F**uture);
- **U** до тех пор, пока (**U**ntil).

Семантика формул LTL определяется на пути поведения системы переходов. Формула истинна на пути, если и только если она истинна в начальном состоянии этого вычисления. Семантика темпоральных операторов приведена далее [40].

Унарные операторы:

$\mathbf{X}\,\varphi$	ne**X**t: φ должно выполнятся в следующем состоянии.
$\mathbf{G}\,\varphi$	**G**lobally: φ должно выполнятся на всем дальнейшем пути.
$\mathbf{F}\,\varphi$	**F**uture: φ должно выполниться когда-нибудь на пути.

Бинарные операторы:

$\psi\ \mathbf{U}\ \varphi$	**U**ntil: ψ должно выполнятся до тех пор, пока не выполнится φ, при этом φ должно когда-нибудь выполнится.
$\psi\ \mathbf{R}\ \varphi$	**R**elease: φ выполняется до первого состояния, в котором выполняется ψ, или всегда, если ψ не выполняется.

Примеры формул LTL для записи некоторых свойств:

$\mathbf{G}(q \Rightarrow \mathbf{XG}\neg q)$ – *q* встретится в будущем не более одного раза,

$\mathbf{F}q \wedge \mathbf{G}(q \Rightarrow \mathbf{XG}\neg q)$ – *q* встретится в будущем точно один раз,

$\mathbf{G}(p \Rightarrow \mathbf{F}q)$ – на *p* всегда будет реакция *q*.

Две темпоральные формулы ϕ и ψ эквивалентны, записывается $\phi \equiv \psi$, когда они выражают одно и то же свойство. Иными словами, $\phi \equiv \psi$ означает, что для любого состояния пути σ утверждение $\sigma|=\phi$ верно тогда и только тогда, когда верно $\sigma|=\psi$. Операторы **U**, **F** и **G** можно определить бесконечной формулой с помощью дополнительного оператора **X**:

$$p\mathbf{U}q \equiv q \vee p \wedge \mathbf{X}q \vee p \wedge \mathbf{X}p \wedge \mathbf{XX}q \vee \ldots$$

$$\mathbf{F}q \equiv q \vee \mathbf{X}q \vee \mathbf{XX}q \vee \ldots$$

$$\mathbf{G}q \equiv q \wedge \mathbf{X}q \wedge \mathbf{XX}q \wedge \ldots$$

А также рекурсивно конечной формулой:

$$p\mathbf{U}q \equiv q \vee p \wedge \mathbf{X}(p\mathbf{U}q) \ ;$$

$\mathbf{G}q \equiv q \wedge \mathbf{XG}q$;

$\mathbf{F}q \equiv q \vee \mathbf{XF}q$.

Пара операторов **X** и **U** образует темпоральный базис LTL. Действительно:

$\mathbf{F}\,\varphi = \mathbf{true}\ \mathbf{U}\,\varphi$;

$\mathbf{G}\,\varphi = \mathbf{false}\ \mathbf{R}\,\varphi = \neg\mathbf{F}\neg\varphi$;

$\psi\ \mathbf{R}\,\varphi = \neg(\neg\psi\ \mathbf{U}\ \neg\varphi)$.

Иногда используются так же нестандартные операторы. В некоторых системах вводится бинарный оператор weak until (записывается **W)**, семантика которого схожа с оператором until, но условие остановки не требуется:

$\psi\ \mathbf{U}\,\varphi = \mathbf{F}\,\varphi \wedge (\psi\ \mathbf{W}\,\varphi)$;

$\psi\ \mathbf{R}\,\varphi = \varphi\ \mathbf{W}\,(\psi \wedge \varphi)$;

$\varphi\ \mathbf{W}\,\psi = \psi\ \mathbf{R}\,(\psi \vee \varphi)$;

$\varphi\ \mathbf{W}\,\psi = (\varphi\ \mathbf{U}\,\psi) \vee \mathbf{G}\,\varphi$.

Существуют также два важных свойства моделей, которые можно выразить с помощью линейной темпоральной логики: свойства безопасности (**safety**), которые утверждают, что что-то плохое никогда не случится (**G**$\neg\varphi$), и свойства живости (**liveness),** которые используются для выражения того, что что-то хорошее будет периодически случаться (**GF**ψ). Для нарушения свойств безопасности можно построить контрпример конечной длины, тогда как для свойств живости префикс каждого контрпримера можно продолжить бесконечным путем.

Для проверки выполнимости LTL формулы на модели современные системы верификации пользуются тем, что любую LTL формулу можно транслировать в автомат Бюхи [16]. Автоматом Бюхи называется пятерка $A = (S, s_0, F, \rho, T)$, где S – множество состояний, $S_0 \subseteq S$ – начальных состояний, а $F \subseteq S$ – допускающих состояний, $\rho : S \times T \to S$ недетерминированная функция переходов и T – алфавит над множеством переходов. Выполнение A над бесконечным словом $w = a_1 a_2 \ldots$ это последовательность $s_0 s_1 \ldots$, где $s_0 \in S_0$ и $s_i \in \rho(s_{i-1}, a_i)$, для всех $i \geq 1$. Выполнение $s_0 s_1 \ldots$ является принимающим, если существует допускающее состояние, которое повторяется бесконечно часто, т. е. для некоторого $s \in F$ существует бесконечно много таких i, что $s_i = s$. Бесконечное слово w принимается автоматом A, если существует допускающее выполнение A над w. Множество бесконечных слов, которые принимаются A, обозначается $\mathcal{L}(A)$.

Варди и Волпер показали [16], что такой автомат принимает только те (бесконечные) последовательности состояний, которые удовлетворяют заданной LTL формуле. Автомат, соответствующий свойству может иметь $2^{O(n)}$ состояний, где n – число подформул (темпоральных операторов) в формуле свойства. Таким образом, размер произведения автоматов, который определяет общую сложность метода, пропорционален $N \cdot 2^{O(n)}$, где N – число достижимых состояний модели.

Линейная темпоральная логика LTL является подмножеством CTL*.

CTL* (Computation Tree Logic)

CTL* [41] формулы включают в себя кванторы путей и темпоральные операторы. Определены два вида кванторов: A («для всех вычислительных путей») и E («для некоторого вычислительного пути»). Такие кванторы, заданные для некоторого состояния, определяют, что для всех путей или для некоторого пути, берущего начало в этом состоянии, выполняется некоторое свойство. Темпоральные операторы описывают свойства пути на всем дереве. Некоторые системы проверки моделей предполагают, что свойства модели специфицированы во фрагменте логики CTL* называемой ACTL*, которая исключает возможность описать существование пути. При этом отрицание ограничено до уровня атомарных формул, а квантификация путей ограничена квантором всеобщности.

Методы и системы проверки формальных моделей

Для сокращения пространства анализируемых состояний разработано множество методов редукции, их можно разделить на два вида – точные, которые гарантируют итоговый результат и эвристические, которые уменьшают время проверки, но при этом не гарантируют точности результатов. К точным можно отнести такие методы, как:

Методы частичного порядка. Проектирование и тестирование параллельных систем усложняется тем, что компоненты могут взаимодействовать множеством зачастую непредусмотренных способов. Интерливинг (перестановка выполнения действий параллельно работающих компонент) является одним из источников комбинаторного взрыва вариантов поведения верифицируемой модели. Точки сообщения (доступ к общим ресурсам) асинхронно работающих сообщающихся между собой и внешней средой процессов могут быть определены статически, основываясь на структурном синтаксическом описании верифицируемой модели; такая информация может быть использована для ограничения числа перестановок путем исключения незначимых с точки зрения проверяемых свойств. Использование метода частичного порядка для элиминации избыточного интерливинга описано в 1994 году в работе Peled [42], подробное описание метода и его адаптацию в различных верификаторах можно найти так же в [43–46]. Необходимо отметить, что такая техника статического определения точек интерливинга не подходит для случая символьного моделирования, так как можно задать состояние (начальное, либо построить динамически в процессе обхода), включающее условие зависимости атрибутов разных процессов, при этом оставляя их синтаксически «локальными».

Использование симметрий при проверке эквивалентности состояний. На практике часто встречаются модели, содержащие множество однотипных процессов, например, телекоммуникационные задачи, в которых участвуют N телефонов. Как правило, в таких случаях стоит вопрос: выполняется ли требуемое свойство для всех N, и каковым должно быть это значение для доказательства (опровержения). Известно, что в общем случае проблема не разрешима [47, 48]. Однако, при некоторых ограничениях, можно воспользоваться перестановкой экземпляров процессов при проверке эквивалентности состояний. Так, система Merφ [48, 49] вводит специальный тип данных (scalarset) и ограничивает операции над ним (сравнения вида «равно» и «не равно», а так же индексацию и присваивания). Такое ограничение влечет неупорядоченность элементов данного типа, что позволяет модифицировать функцию проверки эквивалентности состояний добавлением проверок дополнительных состояний, порождаемых путем перестановок атрибутов модели по принципу их принадлежности к процессам, а так же хранению значения идентификаторов процессов. Таким образом, отношение проверки эквивалентности ослабляется, и достигается сокращение числа анализируемых состояний. Методы использования симметрии описаны так же в [50, 51].

Сжатие состояний. Каждый процесс модели может содержать большое количество атрибутов, в то время как один переход изменяет лишь несколько атрибутов. Так называемый state compression метод [52] экономит память в ходе выполнения проверки свойств, используя указатели на состояния тех процессов, которые не изменялись в последнем переходе на пути вычисления. Таким образом, одинаковые части в различных состояниях модели не дублируются. Сопутствующим методом является возврат к точке выбора ветви поведения. Так, если из некоторого состояния существует допустимый детерминированный переход, не нуждающийся в интерливинге с другими процессами, и таким образом, не ветвящий пространство поведения верифицируемой модели, то такое состояние не является точкой выбора. Алгоритм обхода при одном шаге назад восстанавливает состояние модели, являющееся точкой выбора, при этом достигается эффект «сворачивания» (collapsing) линейных участков поведения модели.

Абстракции на основе исследования зависимостей. Такие методы успешно практикуется в компиляторах, где информационные связи исследуются, как правило, статически. Например, локализированная редукция Куршана [53] представляет собой итеративную технику, которая начинает проверку с небольшим подмножеством релевантных L-процессов, которые топологически близки к спецификациям проверяемых свойств в графе зависимости переменных. Все остальные переменные проверяемой программы абстрагируется путем замены недетерминированными присваиваниями. Если обнаруживается, что порожденный таким образом контрпример ложный, вводятся дополнительные переменные для элиминации такого контрпримера. Так же описание методов редукции, основывающихся на информационных зависимостях можно найти в [54–56].

В работе [57] описан метод «расслоения» (slicing) целью которого является уменьшение исходной формальной модели путем удаления тех переходов (операторов программы), которые не влияют на проверяемые свойства. Для этих целей строятся (статически) графы потока управления (control flow graph) и программы (program flow graph). Основываясь на информационных связях в этих графах, создаются их проекции, сохраняющие такие поведения программы и значения переменных, которых достаточно для верификации проверяемых свойств. Метод оказывается полезным и эффективным при отладке формальных моделей.

В работах [37, 58] описан метод символьного моделирования, в котором состояния представляются формулами многосортного исчисления предикатов первого порядка; для их преобразования используется предикатный трансформер.

В работе [59] описан метод ослабления эквивалентности состояний за счет игнорирования незначимых значений атрибутов. Под «незначимым» на некотором состоянии понимается значение атрибута, которое не используется ни одним переходом, а так же не различается ни одним из проверяемых свойств на всех состояниях, достижимых из этого состояния. Такие абстракции состояний строятся «на лету», таким образом, метод не чувствителен к зависимостям атрибутов, которые обусловлены недостижимыми переходами.

Абстракции предикатов. В последнее время популярной техникой проверки моделей стала техника построения абстракций с последующим уточнением. Уточнения подразумевают повторные запуски эксперимента для устранения ложных поведений. Такой подход часто называется Counter-Example Guided Abstraction Refinement (CEGAR), и составляет основу для многих популярных верификаторов (см., например, [22, 24, 25, 35, 36, 54]). В основе метода абстракции предикатов положена техника удаления данных, при этом отслеживаются только определенные предикаты над данными. В абстрактной модели каждый предикат представляется булевской переменной, в то время как подлинные значения переменных элиминируются. Далее приведен пример (рис. 1), описанный в работе [24] поясняющий метод абстракции предикатов.

init$(x) := 0;$	**init**$(y) := 1;$
next$(x) :=$ **case**	**next**$(y) :=$ **case**
$reset = \text{TRUE} : 0;$	$reset = \text{TRUE} : 0;$
$x < y : x + 1;$	$(x = y) \wedge \neg(y = 2) : y + 1;$
$x = y : 0;$	$(x = y) : 0;$
else $: x;$	**else** $: y;$
esac;	**esac**;

Рис. 1. Программа P

Пусть дана программа P с тремя переменными: x, y с диапазоном допустимых значений {0,1,2}, и *reset* {TRUE, FALSE} (см. рис. 1). Множество атомарных формул Atoms(P) = {(reset=TRUE, ($x = y$), ($x < y$), ($x = 2$)}. Имеются два кластера формул, FC1 = {($x = y$), ($x < y$), ($y = 2$)} и FC2 = {(*reset* = TRUE)}. Соответствующие им кластеры переменных {x, y} и {*reset*}. Значения (0,0) и (1,1) первого кластера в одном классе эквивалентности потому что все атомарные формулы в кластере FC1 выполняются одинаково: (0,0) |= f если и только если (1,1) |=f. Домен {0,1,2}**x**{0,1,2} разделен на пять классов эквивалентности по этому критерию:

$0 = \{(0,0), (1,1)\}$;

$1 = \{(0,1)\}$;

$2 = \{(0,2),(1,2)\}$;

$3 = \{(1,0), (2,0), (2,1)\}$;

$4 = \{(2,2)\}$.

Домен {TRUE, FALSE} имеет два класса эквивалентности – один включает FALSE, другой – TRUE. Так определяются функции абстрагирования $h_1:\{0,1,2\}^2 \rightarrow \{0,1,2,3,4\}$ и $h_2 = \{\text{TRUE, FALSE}\} \rightarrow \{\text{TRUE, FALSE}\}$. Функция h_1 определяется следующим образом: $h_1(0,0) = h_1(1,1) = 0$, $h_1(0,1) = 1$, $h_1(0,2) = h_1(1,2) = 2$, $h_1(1,0) = h_1(2,0) = h_1(2,1) = 3$, $h_1(2,2) = 4$. Вторая функция h_2 тождественна $h_2(reset) = reset$. После определения функций отображения применяются стандартные методы экзистенциальной абстракции [21, 60]. Однако необходимо отметить, что при таком подходе стоит проблема числа возможных абстракций. Например, метод, описанный в работе [60] для диапазона допустимых значений конкретной модели $D_c = \{0, 1, 2\}$ **x** $\{0, 1, 2\}$ и абстрактной $D_a = \{0, 1\}$**x**$\{0, 1\}$ имеет $4^9 = 262144$ функций отображения h из D_c в D_a. В развитии этой работы [25], авторы предложили использовать кластеры переменных, и соответственно функцию отображения для каждого кластера отдельно: $h = (h_1, h_2)$. Число функций отображения из $\{0, 1, 2\}$ в $\{0, 1\}$ равно 2^3, таким образом, имея два кластера, общее число возможных вариантов построения абстракции для данного примера сокращается до 64.

Метод предполагает три (повторяющихся) этапа: 1) построение абстракции, 2) проверка свойств, 3) анализ контрпримера для уточнения абстракции.

Пример. Рассматривается модель управления светофором (рис. 2) и свойство для доказательства f = AGAF(state = red); используемая функция абстракции h(red) = = red и h(green) = h(yellow) = go.

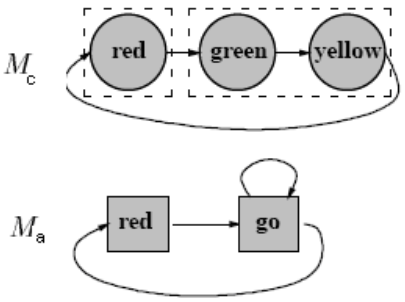

Рис. 2. Модель светофора M_c и его абстракция M_a

Легко видеть, что $M_c \models f$ в то время как $M_a \not\models f$, потому что существует бесконечный путь {red, go, go, …}, который опровергает это свойство. Если абстрактный контрпример не соответствует поведению конкретной модели, его называют ложным. В приведенном примере таковым является путь {red, go, go, …}.

Таким образом, метод может выдавать ложные контрпримеры, истинность которых проверяется на конкретной модели. Если при такой проверке оказывается, что контрпример недопустим, модель уточняется (строится новая, более детальная абстракция), и процедура проверки повторяется. Некоторые методы используют абстракции, определяемые пользователем, как, например [17, 20–22].

Популярные эвристические методы используют такие техники проверки моделей:

Накладывание ограничений на пространство поиска. Чаще всего прибегают к ограничению на длину пути, а максимальное значение фиксируется изначально. Так же используют [61] такие виды ограничителей, как время, число ветвей и др. Многие верификаторы используют заданные пользователем специальные состояния («fairness constraints», «hints», «restricted states») [17, 20] для отсечения ветвей анализируемого поведения.

Аппроксимации. В работе [62] строится бинарный вектор длины, равной количеству различных предикатов в переходах модели. Пройденные состояния характеризуются только этим вектором, что

позволяет говорить о покрытии предикатов. В работе [24] авторы предлагают подход, аналогичный локализированной редукции Куршана, при котором элиминируются переменные, находящиеся в графе зависимости на дистанции, превышающей заданную. В работе [63] описана техника аппроксимации bit hashing, которая позволяет не различать состояния, близкие по их бинарному представлению.

Направленный поиск. В работе [64] предложен метод автоматического направления поиска нарушения свойств модели, в основе которого лежит структурный анализ переходов модели и проверяемых свойств. В работах [65, 66] помимо спецификаций поведения системы, на вход подается сценарий тестирования, называемый обычно целью теста (test purpose), такой сценарий задается пользователем в виде последовательности сообщений, которыми обмениваются компоненты модели (например, в виде MSC [66]) или регулярного выражения [67] и может быть использован в качестве направления при поиске труднодостижимых состояний. Использование эвристических методов направления поиска для обнаружения ошибок так же описаны в [68, 69].

За последние два десятилетия разработано множество различных как коммерческих, так и свободно распространяемых систем верификации для программных и аппаратных систем. Большое число верификаторов разрабатывается из-за отсутствия удовлетворительного для промышленности универсального метода проверки правильности разрабатываемых систем. Различные верификаторы решают узкоспециализированные проблемы, например, проверку свойств, специфических для языков реализации систем. Как правило, верификаторы имеют собственные специальные языки спецификаций и методы проверки моделей. Ниже приведены краткие описания наиболее популярных систем верификации.

– Cadence SMV (Symbolic Model Verifier) [29] – разработка Cadence Berkeley Laboratories для символьной верификации моделей. Проверяет требования корректности, выраженные в CTL*. SMV снабжен двумя языками моделирования – Verilog и расширенным SMV. Для формального задания систем переходов используется *структура Крипке* – модель, которая имеет конечное множество состояний и переходов между ними, причем каждое состояние помечено некоторым множеством истинных в этом состоянии предикатов. Структура Крипке K – это четверка K = (S, I, R, L), где S – конечное непустое множество состояний; I – непустое множество начальных состояний; $R \subseteq S \times S$ – множество переходов; $L: S \to 2^{AP}$ – функция разметки, AP – конечное множество атомных утверждений (атомных предикатов). Путь в структуре Крипке – любая бесконечная цепочка $\pi = s_0 s_1 s_2 s_3 \ldots$, такая, что $s_0 \in I$ и $(s_i, s_{i+1}) \in R$.

За последние десятилетия структуры Крипке признаны адекватной моделью реагирующих дискретных систем управления, параллельных и распределенных алгоритмов [32]. Для сжатия проверяемого множества состояний используются символьные способы представления моделей – двоичные разрешающие диаграммы (BDD, Binary Decision Diagrams) [38]. Алгоритмы верификации моделей с помощью BDD могут быть значительно эффективнее явного перебора состояний в том случае, когда BDD системы переходов модели и промежуточных результатов остаются компактными [29].

– NuSMV (New Symbolic Model checker [23, 70]**)**. Символьный верификатор моделей, разработанный как общий проект Formal Methods Group в ITC- IRST, the Model Checking group в Carnegie Mellon University, the Mechanized Reasoning Group в University of Genova и Mechanized Reasoning Group в University of Trento. NuSMV – это обновленная версия SMV, был разработан в качестве системы с открытой архитектурой для верификации моделей, которая может быть использована для проверки промышленных разработок, как ядро составных верификационных инструментов и как основа для тестирования других технологий формальной верификации.

NuSMV производит верификацию,

комбинируя техники использования BDD и проверку моделей, основанную на SAT (SAT-based model checking). Проверяет требования корректности, выраженные в CTL*. По сравнению с SMV обеспечивает такие возможности как взаимодействие конечных автоматов, анализ инвариантов, реализация методов декомпозиции.

– VCEGAR (Verilog Couter Example Guided Abrstaction Refinement [36, 71]). Предназначен для проверки моделей аппаратного обеспечения; использует язык Verilog для описания поведения модели. Реализует метод абстракции и уточнений предикатов, который разработан для проверки моделей ПО, и реализован в SLAM. Разрабатывается при поддержке General Motors Collaborative Research Lab at CMU, Naval Research Laboratory, Semiconductor Research Corporation и др. Использует Cadence SMV, NuSMV.

– BLAST (Berkeley Lazy Abstraction Software Verification Tool [19, 72]). Верификатор С программ. Использует метод абстракции предикатов. Строит абстракции на лету с требуемой точностью.

– SLAM – развитие BLAST, разработка Microsoft Research [73]. Производит верификацию кода, написанного на языке С. Требования корректности задаются на специализированном языке SLIC. Продукт ориентирован на верификацию драйверов. Так же необходимо отметить разработку той же группы разработчиков Terminator, которая предназначена для проверки зацикливания программ.

– CBMC [27, 74] – верификатор, который обеспечивает возможность ограниченной проверки моделей (Bounded Model Checker) для языков ANSI-C и C++. Он осуществляет такие проверки, как выход за границы массивов (переполнение буфера), безопасность указателей, исключения и утверждения, сформулированные пользователем.

– VeriSoft [45] разработан для проверки свойств параллельно работающих процессов. Его отличительной особенностью является использование так называемого поиска «без состояний» (stateless), запоминаются только переходы текущей трассы. При этом используются техника выборочного поиска (selective search), в основу которого положен метод частичного порядка. Так как состояния не запоминаются, и как следствие, циклы не обнаруживаются, накладываются ограничения на глубину поиска. Проверяются пользовательские свойства (выраженные пропозициональными формулами), а так же наличие тупиков, отклонений (когда процесс не выполняет ни одного наблюдаемого перехода в течении некоторого времени, заданного пользователем) и ливлоков (когда выполнение очередного наблюдаемого перехода некоторого процесса заблокировано на протяжении некоторого числа переходов). Верификатор успешно применялся в тестировании целевого кода ряда промышленных проектов.

– SPIN [18, 52] – инструмент формальной верификации распределенных систем. Использует прямой перебор, проверяя свойства модели в оперативном режиме. Разрабатывался в Bell Labs в группе в Computing Sciences Research Center, начиная с 1980. С 1991 года находится в свободном распространении. SPIN – это популярный инструмент с открытым кодом, используемый во многих организациях мира, в 2001 году был награжден ACM престижной премией System Software Award.

SPIN имеет три основных режима работы: режим симулятора, позволяющий осуществлять быстрое прототипирование со случайным, управляемым или интерактивным моделированием; режим верификатора, способный строго доказать истинность требований корректности, указанных пользователем; режим аппроксимации, который способен подтверждать правильность больших по объему протоколов с максимальным покрытием пространства состояний.

SPIN поддерживает высокоуровневый язык PROMELA (PROcess MEta LAnguage), с помощью которого можно специфицировать описание программных систем. Этот язык поддерживает только дискретные типы данных, а также функции работы с многопоточностью.

SPIN может быть использован для отслеживания ошибок логического дизайна в операционных системах, протоколах обмена данными, системах переключателей, параллельных и распределенных алгоритмах. Для выражения проверяемых свойств SPIN использует линейную темпоральную логику (LTL); обеспечивает прямую поддержку для использования языка C как части спецификации модели. Это делает возможным проведение прямой верификации уровня реализации спецификации программного обеспечения, используя SPIN как драйвер и логический инструмент для верификации высокоуровневых темпоральных свойств. Инструмент поддерживает динамические рост и уменьшение количества процессов, используя технику эластичных векторов состояний (rubber state vector technique); поддерживаются так же синхронизированный и буферизированный обмены сообщениями, а также общение через разделяемую память.

SPIN работает оперативно (on-the-fly), таким образом, избегая необходимости построения глобального графа состояний, или структуры Крипке, в качестве предпосылки для верификации каких-либо свойств системы. Для оптимизации верификации используется техника частичного порядка (partial order reduction), а так же техника слияния состояний (*statement merging*) – которая производит слияние внутренних (ненаблюдаемых) переходов процесса для редукции количества достижимых состояний модели. Для эффективного хранения состояний используется компрессия состояний (разновидность «byte-sharing»), а так же техника аппроксимации bit hashing. Теоретическое обоснование для SPIN, можно найти в [18, 52]. Метод аппроксимации bit hashing детально обсуждается в [63]. На основе SPIN реализовано множество верификаторов для проверки специфик доменов. Создатели утверждают, что он приспособлен для решения крупномасштабных задач.

– Mergφ [48, 49] является инструментом формальной верификации распределенных систем, использует прямой перебор состояний модели, которые представлены значениями всех атрибутов. Отличительной особенностью является метод использования симметрий при проверке эквивалентности состояний. Метод эффективен при наличии повторяющихся компонент (процессов) модели, число которых заранее неизвестно. Ввиду ограничений на размер оперативной памяти, применяется техника записи состояний модели на жесткий диск.

Заключение

Данная работа представляет собой обзор современных методов автоматической проверки свойств формальных моделей. Рассмотрены точные и эвристические методы редукции пространства поиска. Так же приведен обзор современных популярных инструментальных средств верификации.

1. *Brooks F.* No Silver Bullet – Essence and Accidents of Software Engineering // Proc. of the IFIP10–th World Computing Conf. – 1986. – P. 1069–1076.
2. *Nelson M., Clark J., Spurlock M.* Curing the Software Requirements And Cost Estimating Blues // Program manager. – 1999.
3. *Rashinkara P., Paterson P., Singh L.* System-on-a-Chip Verification – Methodology and Techniques // Kluwer Academic Publishers. – 2001. – 392 P.
4. *Boehm B., Basili V.* Software Defect Reduction Top 10 List // IEEE Computer. – 2001. – Vol. 34(1). – P. 135–137.
5. *Hoare T.* The verifying compiler: A grand challenge for computing research // J. ACM. – 2003. – Vol. 50(1). – P. 63–69.
6. *Hoare C.A.R.* An axiomatic basis for computer programming // Communications of the ACM. – 1969. – Vol. 10, N 12. – P. 576–585.
7. *Floyd R.* Assigning meaning to programs // Proc. of Symposium in App. Mathematics // J.T. Schwartz, ed. Mathematical Aspects of Computer Science. – 1967. – Vol. 19. – P. 19–32.
8. *Nipkow T., Paulson L., Wenzel M.* Isabelle/Hol: a proof assistant for higher–order logic // LNCS 2283, Springer–Verlag. – 2002. – 218 p.

9. *Owre S., Shankar N.* Writing PVS Proof Strategies // Design and application of strategies/ Tactics in Higher Order Logics. – 2003. – P. 1–15.
10. *http://www*.klocwork.com
11. *Кулямин В.* Методы верификации программного обеспечения // Всероссийский конкурсный отбор обзорно–аналитических статей по приоритетному направлению "Информационно телекоммуникационные системы". – 2008. – 117 с.
12. *http://en*.wikipedia.org/wiki/Model_checking
13. *http://awards*.acm.org/homepage.cfm?srt=all &awd=140
14. *Emerson E.* Temporal and modal logic // J. van Leeuwen editor: Handbook of Theoretical Computer Science, Elsevier. – 1990. – P. 997–1072.
15. *Wolfgang T.* Automata on infinite objects // Handbook of theoretical computer science. – 1990. – P. 133–191.
16. Sistla A., Vardi M., Wolper P. The complementation problem for Buchi automata with application to temporal logic // Theoretical Computer Science. – 1987. – N 49. – C. 217–237.
17. *Barner S., Glazberg Z., and Rabinovitz I.* Wolf – bug hunter for concurrent software using formal methods // In Computer Aided Verification. – 2005. – P. 153–157.
18. *Ben–Ari. M.* Principles of Spin // Springer Verlag. – 2008. – 216 p.
19. *Beyer D., Henzinger T., Jhala R., Majumdar R.* The software model checker BLAST // Int. J. Software Tools Technol Transfer. – 2007. – N 9. – P. 505–525.
20. *Bloem R., Ravi K., and Somezi F.* Symbolic guided search for CTL model checking // In Design automation conference. – 2004. – P. 29–34.
21. *Burch J., Clarke E., McMillan K., and oth.* Symbolic model checking: 10^20 states and beyond // Information and Computation. – 1992. – Vol. 98, N 2. – P. 142–170.
22. *Chaki S.* A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs // Doctoral Thesis. Carnegie Mellon University. – 2005. – P. 253.
23. *Cimatti A., Clarke E. M., Giunchiglia E., and oth.* NuSMV 2: An Open-Source Tool for Symbolic Model Checking // In Proc. of Int. Conf. on Computer–Aided Verification, Copenhagen, Denmark. – 2002. – P. 359–364.
24. *Clarke E., Grumberg O., Jha S. et al.* Counterexample–guided abstraction refinement // Computer Aided Verification. – 2000. – P. 154–169.
25. *Clarke E., Grumberg O., Jha S., and oth.* Counterexample-guided abstraction refinement for symbolic model checking // Journal of the ACM. –2003. – Vol. 50. – № 5. – P. 752–794.
26. *Clarke E., Jain H., Kroening D.* Verification of SpecC using predicate abstraction // Formal Methods in System Design. – 2007. – Vol. 30. – P. 5–28.
27. *Clarke E., Kroening D., Lerda F.* A tool for checking ANSI–C programs // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by K. Jensen, A. Podelski. LNCS 2988. – 2004. – P. 168–176.
28. *Dams D.* Abstraction in software model checking: principles and practice (Tutorial) // In Proc. of SPIN'02, LNCS 2318. – 2002. – P. 14–21.
29. *McMillan K.* Symbolic Model Checking // Kluwer Academic Publishers. – 1993. – 216 p.
30. *Бурдонов И., Косачев А., Пономаренко В., Шнитман В.* Обзор подходов к верификации распределенных систем. – М.: ИСПРАН. – 2006. – 61 с.
31. *Jhala R., Majumdar R.* Software model checking // ACM Comput. Surv. – 2009. – Vol. 41(4). – 54 p.
32. *Карпов Ю.Г.* Model Checking. Верификация параллельных и распределенных программных систем. – БХВ-Петербург. – 2010. – 552 с.
33. *http://spinroot*.com/spin/workshops/index.html
34. *Минский М.* Вычисления и автоматы. – М.: Мир, 1971. – 368 с.
35. *Henzinger T., Jhala R., Majumdar R., and Sutre G.* Lazy abstraction // In Proc. of the 29th ACM Symposium on Principles of Programming Languages. – 2002. – Vol. 37. – P. 58–70.
36. *Jain H., Kroening D., Sharygina N., and Clarke E.* VCEGAR: Verilog counterexample guided abstraction refinement // In Tools and Algorithms for the Construction and Analysis of Systems '07. – 2007. – 4 p.
37. *Летичевский А.А., Годлевский А.Б., Летичевский А.А. (мл.), Потиенко С.В., Песчаненко В.С.* Свойства предикатного трансформера системы VRS. // Кибернетика и системный анализ. – 2010. – № 4. – С. 3–16.
38. *Bryant R.* Graph–based algorithms for Boolean function manipulation // IEEE Transactions on Computers. – 1986. – Vol. 35. – P. 677–691.
39. *Pnueli A.* The temporal logic of programs // In proc. of the 8th IEEE Symposium on Foundation of Computer Science. – 1977. – P. 46–57.

40. *http://en*.wikipedia.org/wiki/Linear_temporal_logic
41. *http://en*.wikipedia.org/wiki/Computational_tree_logic
42. *Peled D.* Combining partial order reductions with on-the-fly model checking // J. of Formal Methods in System Design. – 1996. – Vol. 8(1). – P. 39–64.
43. *Gerth R., Kuiper R., Peled D., Penczek W.* A partial order approach to branching time logic model checking // Information and Computation. – 1999. – Vol. 150(2). – P. 132–152.
44. *Godefroid P.* Partial-order methods for the verification of concurrent systems – an approach to the state–explosion problem // Lecture notes in computer science, Springer-Verlag. – 1996. – Vol. 1032. –143 p.
45. *Godefroid P.* Software model checking: the VeriSoft approach // Formal methods in system design, Springer science. – 2005. – Vol. 26. – P. 77–101.
46. *Peled D.* Partial order reduction: linear and branching temporal logics and process algebras // POMIV '96: Proc. of the DIMACS workshop on Partial order methods in verification. NY, USA: AMS Press, Inc. – 1997. – P. 233–257.
47. *Apt K., Kozen D.* Limits for automatic verification of finite-state concurrent systems // Information Processing Letters. – 1986. – Vol. 22. – P. 307–309.
48. *Ip C., Dill D.* Better Verification through Symmetry // Formal Methods in System Design. – 1996. – Vol. 9. – P. 41–75.
49. *Ip C., Dill D.* Verifying Systems with Replicated Components in Murphi // Int. Conf. on Computer-Aided Verification. – 1996. – P. 147–158.
50. *Miller A., Donaldson A., and Calder M.* Symmetry in temporal logic model checking // ACM Comput. Surv. – 2006. – Vol. 38. – 37 p.
51. *Nilsson M.* Structural Symmetry and Model Checking // Ph.D. thesis, Uppsala University, Uppsala, Sweden. – 2005. – 157 p.
52. *Holzmann G.* The SPIN Model Checker: Primer and Reference Manual. – Addison-Wesley Professional. – 2003. – 596 p.
53. *Kurshan R.* Computer-Aided Verification of Coordinating Processes // Princeton University Press. – 1994. – 270 p.
54. *Gupta A., Wang C., Kim H.* Hybrid CEGAR: combining variable hiding and predicate abstraction // Proc. IEEE/ACM Int. Conf. on computer-aided design. – 2007. – P. 310–317.
55. *Kesten Y., Pnueli A.* Control and data abstraction: The cornerstones of practical formal verification // Int. J. on Software Tools for Technology Transfer. – 2000. – Vol. 2(4). – P. 328–342.
56. *Lind–Nielsen J., Andersen H., Behrmann G. and oth.* Verification of large state/event systems using compositionality and dependency analysis // Journal of Formal Methods in System Design. – 2001. – Vol. 18(1). – P. 5–23.
57. *Krinke J.* Program Slicing // In Handbook of software engineering and knowledge engineering. – Vol. 3. – 2005. – P. 307–332.
58. *Годлевский А.Б.* Предикатные трансформеры в контексте символьного моделирования транзиционных систем // Кибернетика и системный анализ. – 2010. – № 4. – С. 91–99.
59. *Колчин А.В.* Автоматический метод динамического построения абстракций состояний формальной модели // Кибернетика и системный анализ. – 2010. – № 4. – С. 70–90.
60. *Clarke E. Grumberg O., Long D.* Model checking and abstraction // In ACM Transactions on programming languages and systems. – 1994. – Vol. 16(5). – P. 1512–1542.
61. *Jussila T.* On Bounded Model Checking of Asynchronous Systems // Doctoral Thesis. Helsinki University of Technology, Laboratory for Theoretical Computer Science. Research report A97. – 2005. – 136 p.
62. *Ball T.* A theory of predicate–complete test coverage and generation // In FMCO'2004: Symp. on Formal Methods for Components and Objects. SpringerPress. – 2004. – P. 1–22.
63. *Holzmann G.* An analysis of bitstate hashing // For-mal Methods in Systems Design. – 1998. – P. 301–314.
64. *Peranandam P., Weiss R., Ruf J., Kropf T. and Rosenstiel W.* Dynamic guiding of bounded property checking // In IEEE International High Level Design Validation and Test Workshop. – 2004. – P. 15–18.
65. *Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A.* UniTesK Test Suite Architecture // In Proc. of FME 2002, LNCS 2391, Springer-Verlag. – 2002. – P. 77–88.
66. *Grabowski J., Hogrefe D.* SDL and MSC-Based Specification and Automated Test Case Generation for INAP // Telecommunication Systems J. – 2002. – Vol. 20(3,4). – P. 265–291.
67. *Летичевский А.А., Колчин А.В.* Генерация тестовых сценариев на основе формальной модели // Проблеми програмування. – 2010. – № 2–3. – С. 209–215.

68. *Edelkamp, S., Leue S. and Lafuente A.* Directed explicit-state model checking in the validation of communication protocols // International j. on software tools for technology transfer. – 2003. – N 5. – P. 247–267.
69. *Lafuente A.* Directed Search for the Verification of communication protocols // PhD thesis, University of Freiburg. – 2003. – 157 p.
70. *http:*//nusmv.irst.itc.it
71. *http:*//www.cs.cmu.edu/~modelcheck/vcegar
72. *http:*//mtc.epfl.ch/software–tools/blast
73. *http:*//research.microsoft.com/slam
74. *http:*//www.cs.cmu.edu/~modelcheck/cbmc

Получено 12.09.2011

Об авторах:

Колчин Александр Валентинович,
кандидат физико-математических наук,
научный сотрудник,

Летичевский Александр Александрович,
кандидат физико-математических наук,
старший научный сотрудник,

Потиенко Степан Валериевич,
кандидат физико-математических наук,
научный сотрудник,

Песчаненко Владимир Сергеевич,
кандидат физико-математических наук,
доцент.

Место работы авторов:

Институт кибернетики
имени В.М. Глушкова НАН Украины.
Тел. (044) 200 8423,
e-mail: kolchin_av@yahoo.com,
e-mail lit@iss.org.ua,
e-mail stepan@iss.org.ua,

Херсонский государственный университет,
доцент кафедры информатики
Тел. (095) 324 1557,
e-mail: vladimirius@gmail.com