ПОСТРОЕНИЕ СОКРАЩЕННОГО ДЕРЕВА ДОСТИЖИМОСТИ ДЛЯ МОДЕЛЕЙ ПРОГРАММ В ТЕРМИНАХ СЕТЕЙ ПЕТРИ*

***Д. В. Леонтьев*[1], *Д. И. Харитонов*[2]**

Аннотация: Рассматривается задача построения пространства состояний для анализа поведения императивных программ. При автоматическом построении моделей программ эффект взрыва числа анализируемых состояний составляет основную проблему для поиска ошибок в исходных текстах программ, причем этот взрыв индуцируется за счет состояния множества переменных программы. Предлагается подход к уменьшению числа состояний дерева достижимости моделей программ через отделение модели потока управления программой от моделей переменных и последующего добавления только переменных, влияющих на поток управления, и сокращения состояний этих переменных. Рассмотренный в статье пример показывает, как на практике может быть применен такой подход.

Ключевые слова: сети Петри; дерево достижимости; проверка корректности программ; моделирование поведения программ

DOI: 10.14357/08696527220203

1 Введение

Зависимость современного образа жизни людей от информационных технологий постоянно растет. Растет объем и разнообразие используемого программного обеспечения, которое должно удовлетворять потребностям общества. При этом постоянное появление нового программного обеспечения происходит в условиях, когда процесс качественной проверки отстает от процесса разработки. В результате надежность программного обеспечения, от которого часто зависит жизнь людей, остается под вопросом. Например, из-за ошибки в программном обеспечении аппарата лучевой терапии Therac-25 неправильно рассчитывалась доза облучения и в результате некоторые пациенты получили дозы не в несколько сотен рад, а в 20 000 рад [1] (доза 1000 рад для человека считается несовместимой с жизнью). Другим примером проблем в программном обеспечении могут

*Статья публикуется по представлению программного комитета VI Международной научно-практической конференции «Информационные технологии и высокопроизводительные вычисления» (ITHPC-2021). Работа выполнена при финансовой поддержке темы госбюджетного задания № 121021700006-0.

[1]Институт автоматики и процессов управления Дальневосточного отделения Российской академии наук, devozh@dvo.ru

[2]Институт автоматики и процессов управления Дальневосточного отделения Российской академии наук, demiurg@dvo.ru

послужить две авиакатастрофы Boing 737 MAX, которые произошли по вине новой системы увеличения маневренных характеристик (MCAS — maneuvering characteristics augmentation system) и повлекли за собой гибель всех пассажиров, находившихся на борту [2]. Из-за ошибок системы автопилот самолета получал неверные данные об угле атаки, вследствие чего считал, что самолет находится на грани сваливания, и опускал нос самолета. Есть еще немало примеров катастрофических ошибок в программном обеспечении, которые могли бы быть обнаружены при надлежащей проверке, что позволило бы избежать трагедий.

В настоящей статье для моделирования программ используются сети Петри, хорошо зарекомендовавшие себя при моделировании параллельных и распределенных систем. В большинстве методов анализа сетей Петри применяется построение графа достижимости, описывающего пространство состояний программы. В сетях Петри, как и в других методах проверки на модели, основной проблемой может стать комбинаторный взрыв числа состояний, так как при построении полного пространства состояний программы необходимо учитывать значение каждой его переменной в каждый момент времени. В настоящей работе предлагается подход к уменьшению числа состояний дерева достижимости моделей программ за счет отделения модели потока управления программой от моделей переменных и последующего добавления только тех переменных, которые влияют на поток управления.

2 Пример программы, построение модели

Для построения дерева достижимости в качестве примера рассмотрим программу, изображенную на рис. 1. В этой программе главная функция main создает поток Produce и вызывает функцию Consume. Поток Produce вычисляет члены некоторого математического ряда и посылает их в функцию Consume, которая вычисляет сумму ряда и определяет момент, когда оба потока должны закончить свое выполнение. Основная функция main дожидается завершения обоих потоков и после этого прекращает свое выполнение.

Тестовый пример составлен таким образом, что если поменять местами восьмую и девятую строки в функции Consume, то произойдет дедлок в синхронизации потоков — поток Consume закончит свое исполнение, в то время как поток Produce будет продолжаться. Если рассчитывать на автоматизацию процесса проверки поведения программы, то, несмотря на небольшой код и небольшое число переменных, в зависимости от скорости сходимости выбранного математического ряда число анализируемых состояний программы может достигать произвольно больших чисел.

Построение модели программы автоматизированным способом осуществляется на основе абстрактного семантического графа (ASG — abstract semantic graph), получаемого после синтаксического разбора исходных текстов [3]. При этом формирование M-сети, включающей все значения переменных в исполняемом процессе, нецелесообразно, так как эффект *взрыва числа состояний* сделает

```
 1: int main ( )
 2: {
 3:     pthread_t ThreadId;
 4:     double X=0.7;
 5:     int errcode = pthread_create (&ThreadId, NULL, Produce, &X);
 6:     if (errcode != 0)
 7:         return 1;
 8:     Consume ( );
 9:     void *res;
10:     pthread_join (ThreadId, &res);
11:     return 0;
12: }
```

```
 1: double Consume ( )
 2: {
 3:       bool flagContinue=true;
 4:       double value,sum=0.0,lastsum=–1;
 5:       while (flagContinue) {
 6:            Recv(value);
 7:            sum+=value;
 8:            flagContinue=(lastsum!=sum);
 9:            Send (flagContinue);
10:            lastsum=sum;
11:       }
12:       return sum;
13: }
```

```
 1: #define MakeValue(x,n) x*pow(–1,n)/(2*n–1)
 2: void *Produce (void* param)
 3: {
 4:     bool flagContinue=true;
 5:     double value,X=*(double *)param;
 6:     int n=1;
 7:     while(flagContinue)
 8:     {
 9:          value=MakeValue(X,n);
10:          n=n+1;
11:          Send(value);
12:          Recv(flagContinue);
13:     }
14:     return NULL;
15: }
```

Рис. 1 Фрагмент программы на языке C

построение графа достижимости программы фактически невозможным. Поэтому существуют два варианта. Во-первых, можно обесцветить модель программы, полностью удалив из нее данные, но сохранив граф потока управления. Во-вторых, можно дополнительно отделить данные от потока управления программой. Для этого необходимо вынести переменные в отдельные подсети, которые группируются по типам используемых переменными данных. Каждая такая сеть описывает свой тип данных как абстрактный, т. е. выделяя методы и операции, которые можно применить к типу данных, доступ к которым осуществляется через точки доступа.

Исходя из текстов программы, представленных на рис. 1, полная модель программы в композициональном виде должна содержать сети Петри, описывающие функции main, Consume, Produce, а также сети, описывающие среду исполнения. К среде исполнения относится загрузчик программы, сетевое окружение, описывающее функции для пересылки сообщений Send и Recv, а также библиотека работы с потоками, представленная в программе функциями

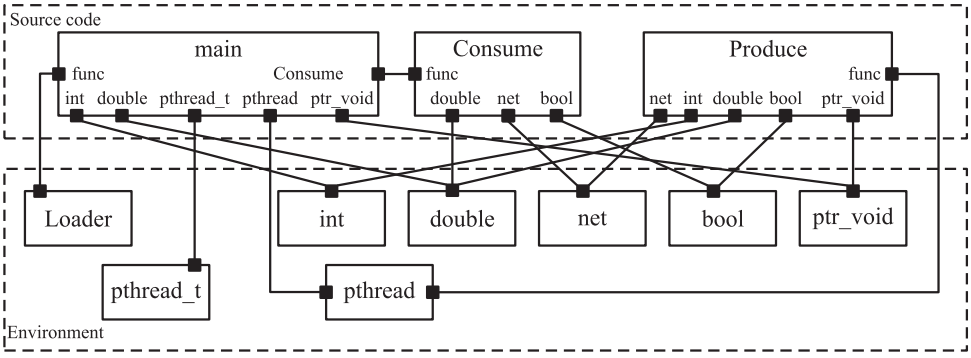

Рис. 2 Композициональное представление модели программы с выделением данных в отдельные сети

pthread_begin и pthread_join. На рис. 2 представлено композициональное представление модели программы с разделением на модели данных и модели функций, включающее все вышеобозначенные сети.

3 Построение дерева достижимости

Для построения дерева достижимости рассмотрим упрощенную модель программы, составленную из функций Consume, Produce и среды передачи данных, изображенную на рис. 3. В эту модель добавлены стартовые и конечные места для функций, а также обозначена начальная маркировка, позволяющая провести моделирование поведения программы. Переходы модели размечены точками доступа для синхронизации с сетями, моделирующими данные: bool, int, double, ptr_void.

Схематично анализ поведения программы можно представить в виде двух шагов: во-первых, необходимо построить пространство состояний, в которых может побывать программа, а во-вторых, проверить на этом пространстве наличие тупиковых состояний, возможность появления необходимых для нормального поведения программы событий (живость переходов), а также отсутствие появления вредных событий, если таковые были заранее обозначены. Представленный ниже классический алгоритм строит итерационно *дерево достижимости*, начиная от начальной маркировки и продолжая добавлять к уже построенным достижимым состояниям сети Петри новые, только если состояния ранее не встречались. Дерево достижимости отличается от графа тем, что в нем не объединяются вершины с одинаковой маркировкой, что позволяет легко построить путь от каждой вершины до стартовой, т. е. начальной маркировки сети.

Классический алгоритм построения дерева достижимости может быть применен к анализируемой модели при условии ее обесцвечивания. С точки зрения

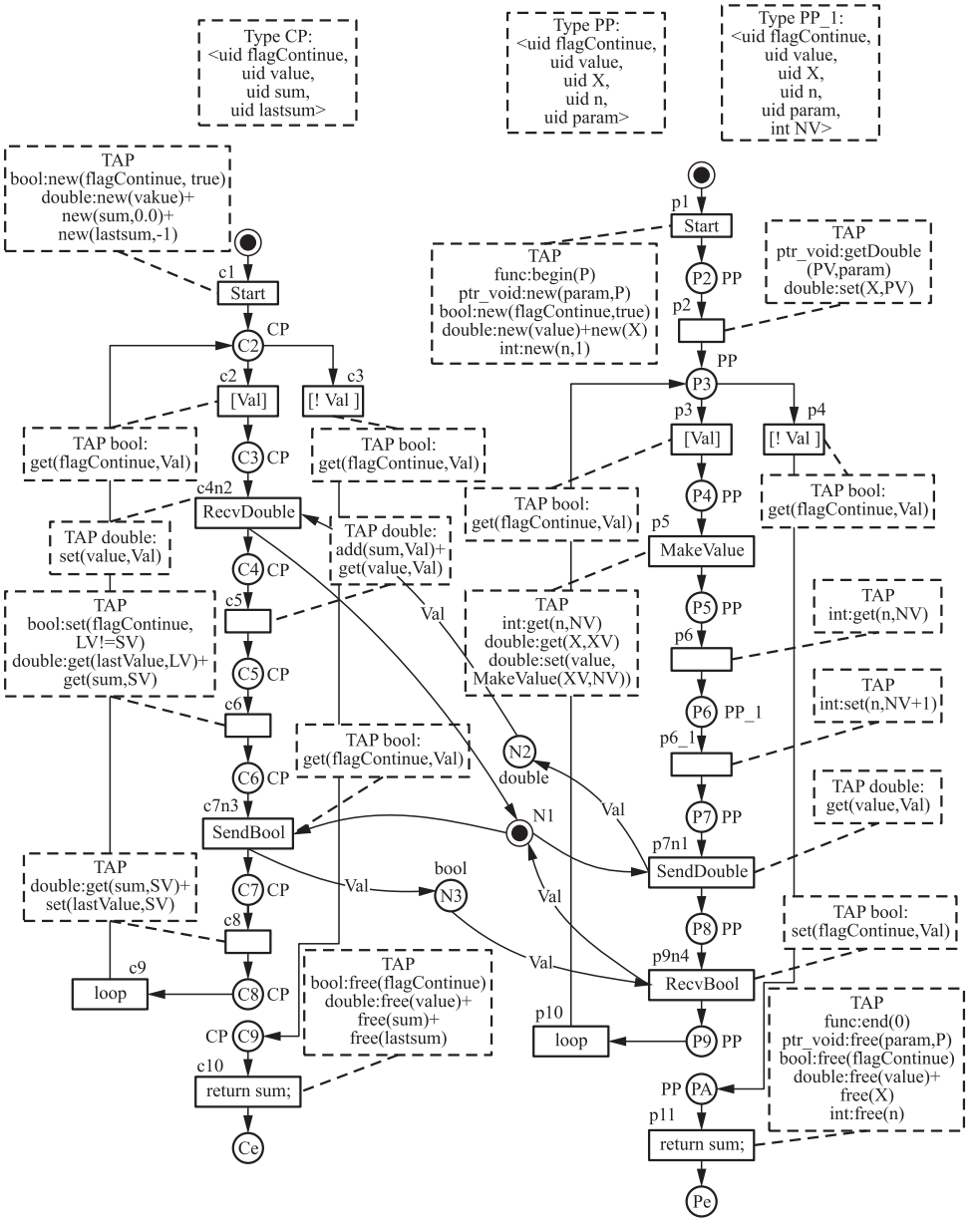

Рис. 3 Анализируемая модель

Классический алгоритм построения дерева достижимости

```
1: Waiting:={M0};
2: Tree.BeginTree(M0);
3: repeat
4:  remove M from Waiting;
5:  for all (t,M')  in  Next(M) do
6:      begin
7:          if !Tree.Has(M')    Waiting.Add(M');
8:          Tree.AddArc(M,t,M');
9:      end;
10: until !Waiting.IsEmpty()
```

`Next(M)` определяется исходя из правил срабатывания переходов следующим образом: `Next(M)` $= \{(t, M') \mid t \in T \wedge M[t\rangle M'\}$, где запись $M[t\rangle M'$ означает, что маркировка M' получается из маркировки M после срабатывания перехода t.

анализа это означает, что рассматриваются все ветви исполнения программы, даже такие, которые не могут реально произойти из-за отсутствия необходимого набора значений переменных в пространстве достижимых состояний программы. На рис. 4 представлено дерево достижимости, построенное для обесцвеченной модели. Узел с маркировкой на более темном фоне означает состояние программы, из которого достижимы конечные состояния программы, которыми служат маркировки Ce + Pe + N1, Ce + P8 + N2 и C3 + Pe + N1. Достижение программой первой маркировки отражает факт корректного исполнения программы, достижение последних двух — факт тупиковых состояний программы или, по-другому, факт ее зависания. Анализ обесцвеченной сети позволяет сказать, что, возможно, в коде программы присутствуют варианты некорректного поведения, а также указать путь от начальной маркировки до тупиковых состояний, который далее программист может проверить *вручную*.

Ранее представленный классический алгоритм построения дерева достижимости для сетей Петри может быть естественным образом расширен для M-сетей, однако, как было упомянуто в разд. 2, программа построена таким образом, что при медленной сходимости ряда (в тексте приведен ряд Лейбница) пространство состояний программы будет включать миллиарды значений переменных. При этом с точки зрения поведения программы необходимо всего лишь ответить на вопрос, достижимы ли маркировки Ce + Pe + N1, Ce + P8 + N2 и C3 + Pe + N1.

Основой для построения сокращенного дерева достижимости служит подход к раздельной верификации императивных программ с применением сетей Петри, описанный одним из авторов в 2009 г. [4]. Согласно этому подходу можно проверять работоспособность классов (модулей) в программе, заменяя их модели на протоколы, причем делать это можно поочередно с каждым классом или каждой переменной в программе. Чтобы определить совместимость программы и исполь-

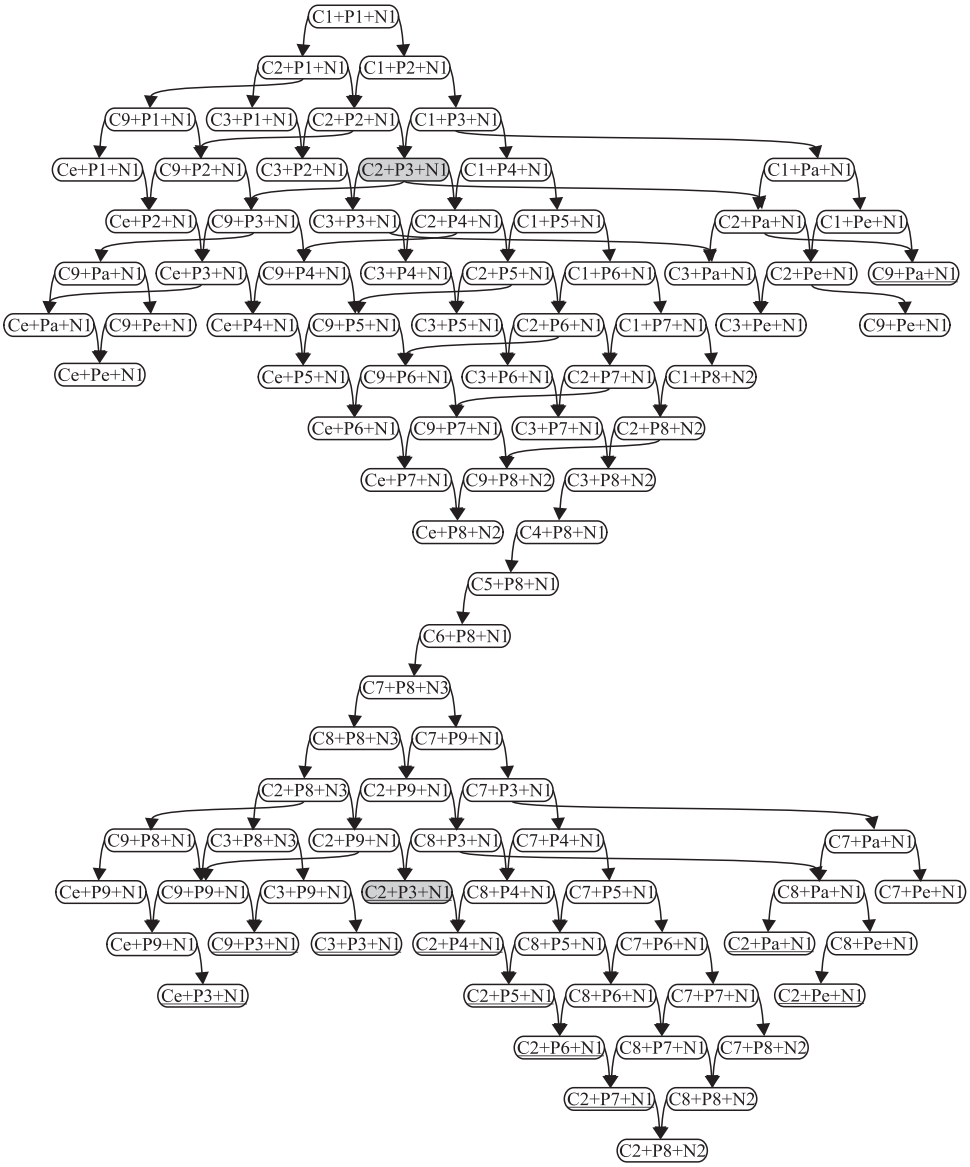

Рис. 4 Дерево достижимости обесцвеченной анализируемой модели

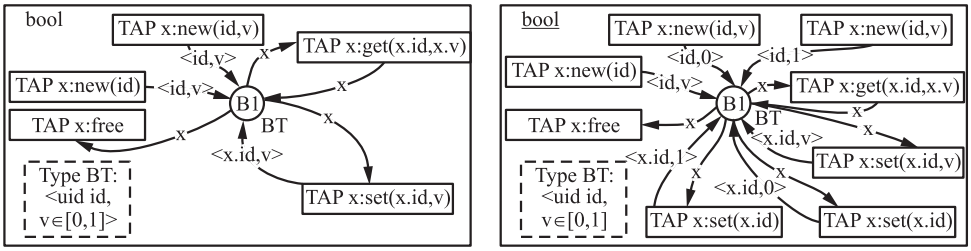

Рис. 5 Два варианта протокола типа данных: bool (*а*) и менее детерминированный bool (*б*)

зуемого ею модуля, необходимо выполнение двух условий. Во-первых, анализ поведения программы, синхронизированной с протоколом модуля, не должен содержать ошибок, а во-вторых, протокол модуля должен описывать поведение, допускающее в качестве одного из вариантов поведение модели реализации модуля. Фактически понятие протокола должно быть менее детерминированным, менее детализированным, т. е. предлагать больше вариантов исполнения, чем реальный модуль. Таким образом получается, что *если иметь набор протоколов разной степени детализации для каждого из основных типов данных, используемых в программе, то можно проверить, насколько программа зависит от конкретной переменной или конкретного типа данных*.

В случае с анализируемой программой можно заменить детерминированный протокол типа данных bool (в котором каждая операция присвоения должна содержать присваиваемое переменной значение) на менее детерминированный протокол, в котором имеются операции присвоения с указанием и без указания присваиваемого переменной значения. На рис. 5 изображены оба варианта модели типа данных bool. На рис. 5, *б* приведен менее детерминированный протокол, в котором операция присвоения set(x.id) случайным образом присваивает переменной значение 0 или 1.

На рис. 6 изображено дерево достижимости анализируемой модели, синхронизированной с менее детерминированным протоколом типа данных bool. К маркировкам состояний дерева достижимости 4 добавлены токены в месте B1 протокола типа данных, для краткости обозначенные как Bc1, Bc0 для переменной в части сети, относящейся к потоку управления из функции Consume, и Bp1, Bp0 — из функции Produce. Видно, что состояние Ce + Pe + N1 из обесцвеченной сети достигается при маркировке Ce + Pe + N1 + Bc0 + Bp0 в более полной модели. Маркировки Ce + P8 + N2 и C3 + Pe + N1 недостижимы в более полной модели, и это означает, что цель построения пространства состояний достигнута при анализе поведения программы с использованием протокола двух переменных одного типа данных. Таким образом, дерево достижимости на рис. 6 описывает полностью поведение анализируемой программы и при этом содержит минимальный набор данных, необходимый для такого описания.

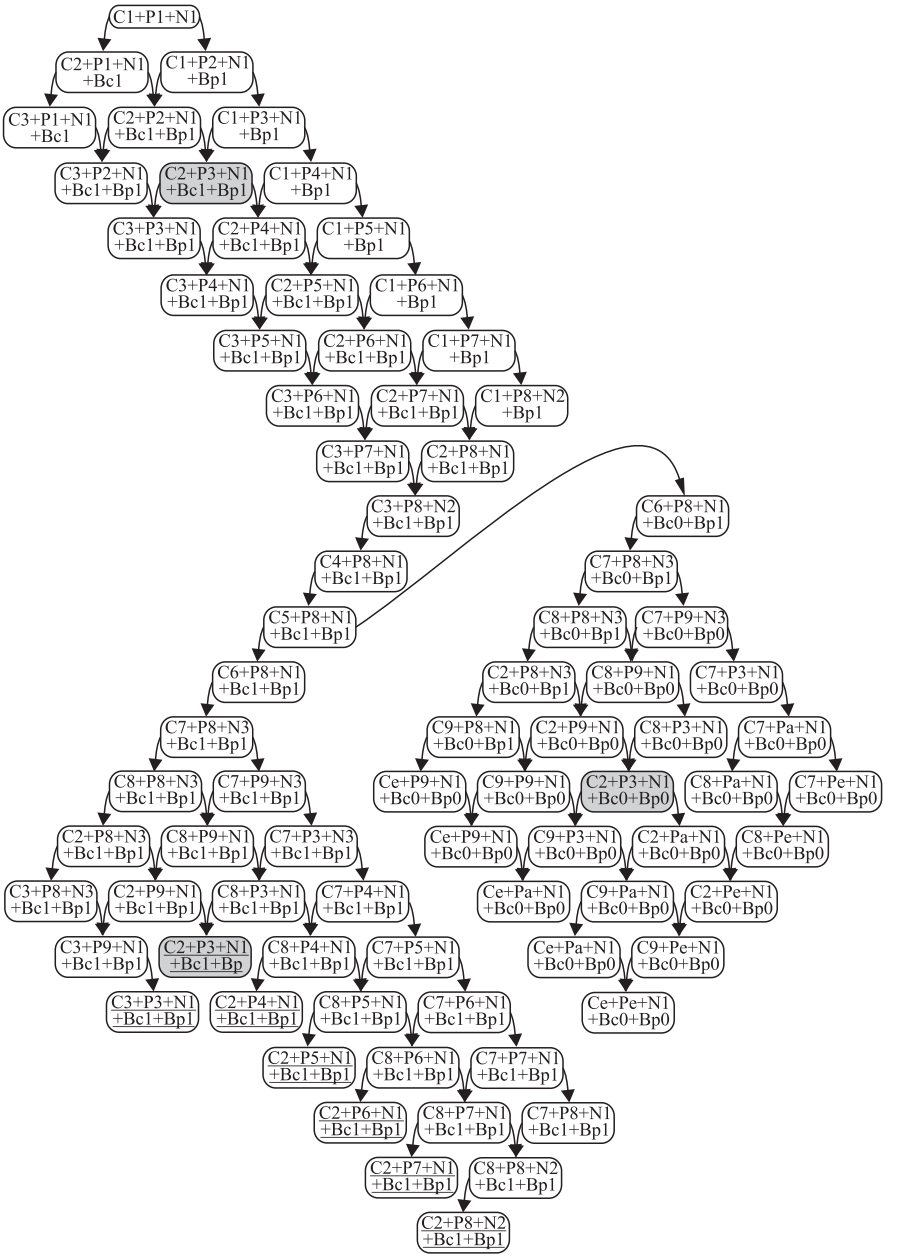

Рис. 6 Дерево достижимости анализируемой модели, синхронизированной с протоколом типа данных bool

4 Заключение

Сети Петри как средство моделирования распределенных систем не так часто применяется для построения моделей программ [5]. В большинстве случаев их применение ограничено этапом дизайна программных систем и либо не доходит до стадии проверки корректности программ, либо ограничивается довольно примитивными моделями. Между тем, как продемонстрировано в работах авторов, сети Петри могут быть использованы для автоматизированного построения моделей программ [3], где размеры моделей могут достигать сотен тысяч мест и переходов. Технически проверка таких моделей программ может выполняться с применением темпоральных логик и символьной верификации [6], однако построение пространства состояний программы выглядит более перспективным, так как поддается полной автоматизации.

К сожалению, при полном моделировании работы моделей программ неизбежно возникает проблема *взрыва числа состояний* (state explosion problem), так как в общем случае размерность пространства состояний сильно зависит от размера модели, числа моделируемых процессов, числа используемых переменных, а также от количества значений, которые могут принимать переменные. В результате прямой подход к построению пространства состояний перестает работать уже для самых простых программ. Поэтому важным направлением развития методов верификации не только программ, но и распределенных систем в целом становится разработка методов сокращения числа состояний. С конца XX в. среди таких методов выделяют: метод упрямых множеств (stubborn set method), использование двоичных разрешающих диаграмм (symbolic binary decision diagrams), методы, основанные на частичном порядке (partial order methods), а также использование симметрии и эквивалентности на рассматриваемых моделях [7].

В контексте использования сетей Петри при верификации распределенных систем для сокращения числа состояний могут быть применены методы, основанные на частичном порядке, а более конкретно — развертки моделей, введенные МакМилланом в 1993 г. [8], и понятие ветвящихся процессов [9]. В отличие от графа или дерева достижимости, где поведение системы описано в виде всех возможных последовательностей срабатывания переходов, развертки и ветвящиеся процессы описывают поведение системы в рамках истинного параллелизма, разворачивая по-отдельности поведение всех независимо существующих процессов. Таким образом, развертки сетей представляют сокращенное представление дерева достижимости за счет уменьшения числа дублирующихся состояний, описывающих независимо исполняющиеся процессы. Развертки могут применяться и к раскрашенным сетям Петри, более подходящим для описания и анализа программ [10]. Однако, например, развертка рассмотренного в настоящей статье примера по-прежнему будет содержать миллиарды состояний, так как развертки сетей не могут сократить число состояний используемых в программе переменных. Предложенный же в настоящей статье подход направлен на сокращение

числа состояний, связанных с использованными переменными. В дальнейшем авторы надеются адаптировать его и для построения более кратких разверток моделей программ.

В предложенном подходе в описании моделей императивных программ в терминах сетей Петри переменные выделяются в отдельные сети, моделирующие типы данных. При таком выделении появляется возможность построения *обесцвеченного* дерева достижимости для потока управления программой без учета данных, на котором могут быть обнаружены корректные и потенциально опасные пути исполнения программ. Построение протоколов типов данных с различной детализацией позволяет проверить достижимость потенциально опасных состояний, не учитывая значения всех переменных программы. Дерево достижимости, дополненное значениями переменных, позволяющих достоверно определить достижимость всех его состояний, представляет собой сокращенное дерево достижимости программы.

Литература

1. *Leveson N. G., Turner C. S.* An investigation of the Therac-25 accidents // Computer, 1993. Vol. 26. No. 7. P. 18–41.
2. *Mako S., Pilat M., Svab P., Kozuba J., Cicvakova M.* Evalution of MCAS system // Acta Avionica J., 2019. Vol. 21. No. 1. P. 21–28.
3. *Kharitonov D., Tarasov G.* Modeling function calls in program control flow in terms of Petri nets // ACSIJ, 2014. Vol. 3. No. 6. P. 82–91.
4. *Харитонов Д. И.* Раздельная верификация объектно-ориентированных программ с построением протокола C++ класса в терминах сетей Петри // Моделирование и анализ информационных систем, 2009. Т. 16. № 1. С. 92–112.
5. *Denaro G., Pezze M.* Petri nets and software engineering // DAIMI Report Series, 2003. Vol. 29. No. 1. P. 439–466.
6. *Latvala T., Makela M.* LTL model checking for modular Petri nets // Applications and theory of Petri nets / Eds. J. Cortadella, W. Reisig. — Lecture notes in computer science ser. — Springer, 2004. Vol. 3099. P. 298–311.
7. *Valmari A.* The state explosion problem // Lectures on Petri nets I: Basic models / Eds. W. Reisig, G. Rozenberg. — Lecture notes in computer science ser. — Springer, 1998. Vol. 1491. P. 429–528.
8. *McMillan K. L.* Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits // Computer aided verification / Eds. G. von Bochmann, D. K. Probst. — Berlin, Heidelberg: Springer, 1993. P. 164–177.
9. *Engelfriet J.* Branching processes of Petri nets // Acta Inform., 1991. Vol. 28. No. 6. P. 575–591.
10. *Козюра В. Е.* Реализация системы проверки моделей раскрашенных сетей Петри с использованием разверток. — Новосибирск: ИСИ им. А. П. Ершова СО РАН, 2002. 44 р.

Поступила в редакцию 30.11.21

CONSTRUCTING OF THE BRIEF REACHABILITY TREE FOR PROGRAM MODELS IN TERMS OF PETRI NETS

D. V. Leontyev and D. I. Kharitonov

Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation

Abstract: The article deals with the problem of building a state space for analyzing the imperative programs behavior. The state explosion problem of analyzed states number in the automatic program models construction is the main problem for finding errors in the programs source code. This explosion is induced mainly due to the composition of the program variables states. The article proposes an approach to reducing the number of states of the reachability tree of program models by separating the program control flow model from the variable models and then adding only variables that affect the control flow and reducing the states of these variables. The example considered in the article shows how such an approach can be applied in practice.

Keywords: Petri nets; reachability tree; checking programs correctness; modeling program behavior

DOI: 10.14357/08696527220203

Acknowledgments

The paper is published on the proposal of the Program Committee of the VI International Scientific and Practical Conference "Information Technologies and High Performance Computing" (ITHPC-2021). The research was supported by state funded program No. 121021700006-0.

References

1. Leveson, N. G., and C. S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26(7):18–41.
2. Mako, S., M. Pilat, P. Svab, J. Kozuba, and M. Cicvakova. 2019. Evalution of MCAS system. *Acta Avionica J.* 21(1):21–28.
3. Kharitonov, D., and G. Tarasov. 2014. Modeling function calls in program control flow in terms of Petri nets. *ACSIJ* 3(6):82–91.
4. Kharitonov, D. 2009. Razdel'naya verifikatsiya ob"ektno-orientirovannykh programm s postroeniem protokola C++ klassa v terminakh setey Petri [Separable object-oriented program verification with C++ class protocol definition in terms of Petri nets]. *Modelirovanie i analiz informatsionnykh system* [Modeling and Analysis of Information Systems] 16(1):92–112.
5. Denaro, G., and M. Pezze. 2003. Petri nets and software engineering. *DAIMI Report Series* 29(1):439–466.

6. Latvala, T., and M. Makela. 2004. LTL model checking for modular Petri nets. *Applications and theory of Petri nets.* Eds. J. Cortadella and W. Reisig. Lecture notes in computer science ser. Springer. 3099:298–311.
7. Valmari, A. 1998. The state explosion problem. *Lectures on Petri nets I: Basic models*. Eds. W. Reisig and G. Rozenberg. Lecture notes in computer science ser. Springer. 1491:429–528.
8. McMillan, K. L. 1993. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. *Computer aided verification*. Eds. G. von Bochmann and D. K. Probst. Berlin, Heidelberg: Springer. 164–177.
9. Engelfriet, J. 1991. Branching processes of Petri nets. *Acta Inform.* 28(6):575–591.
10. Kozyura, V. E. 2002. *Realizatsiya sistemy proverki modeley raskrashennykh setey Petri s ispol'zovaniem razvertok* [Implementation of a system for checking models of colored Petri nets using developments]. Novosibirsk: A. P. Ershov IIS SB RAS. 44 p.

Received November 30, 2021

Contributors

Leontyev Denis V. (b. 1992) — scientist, Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation; devozh@dvo.ru

Kharitonov Dmitriy I. (b. 1973) — Candidate of Science (PhD) in technology, senior scientist, Institute of Automation and Control Processes, Far-Eastern Branch of the Russian Academy of Sciences, 5 Radio Str., Vladivostok 690041, Russian Federation; demiurg@dvo.ru