Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 94–105

УДК 519.686.4+519.688

Автоматическое обнаружение ошибок конкурентной модификации данных в моделях на языке SystemC

Захаров А.В., Моисеев М.Ю.

Санкт-Петербургский государственный политехнический университет

e-mail: zakharov@kspt.ftk.spbstu.ru, mikhail.moiseev@gmail.com

получена 15 сентября 2011 года

Ключевые слова: статический анализ, ошибки конкурентной модификации данных

Модели систем на языке SystemC, как правило, являются параллельными программами и поэтому могут содержать ошибки синхронизации. Одним из типов ошибок синхронизации являются ошибки конкурентной модификации данных.

В данной статье предлагается подход к обнаружению ошибок конкурентной модификации данных в моделях на языке SystemC на основе статического анализа. Разработаны алгоритмы, обеспечивающие анализ программ на языке SystemC без количественного времени. Эти алгоритмы позволяют обнаружить все ошибки конкурентной модификации данных, имеющиеся в программе. Эффективность предложенного подхода подтверждается экспериментальными исследованиями разработанного средства обнаружения ошибок на наборе тестовых программ.

1. Введение

В последнее время с ростом количества элементов сложность разработки систем на кристалле постоянно возрастает. Поэтому необходимо использовать языки высокоуровневого описания и моделирования. Одним из таких языков является язык SystemC [1].

Основной целью применения языка SystemC является моделирование поведения программно-аппаратных систем для определения характеристик функционирования, сравнения различных вариантов реализации и др. Целевая система представляется с помощью модели на языке SystemC. Моделирование поведения системы осуществляется с помощью симулятора, входящего в состав дистрибутива SystemC.

Модель на языке SystemC является параллельной программой. Одним из распространенных классов ошибок в параллельных программах являются ошибки синхронизации: взаимная блокировка процессов (Deadlock), конкурентная модификация данных (Data race), отсутствие прогресса (Livelock). В данной работе рассматриваются методы автоматического обнаружения ошибок конкурентной модификации данных. Для определения ошибки конкурентной модификации данных введем

понятие блока программы. Блок программы – множество конструкций, выполняющихся в одном процессе без переключения контекста. Параллельные блоки программы – блоки программы, выполняющиеся в разных процессах на одном дельта-цикле. Порядок выполнения параллельных блоков является недетерминированным. В программе на SystemC имеется ошибка конкурентной модификации данных, если несколько операций с некоторым объектом выполняются в параллельных блоках программы и при этом хотя бы одна из этих операций изменяет значение объекта.

Наличие ошибок конкурентной модификации данных приводит к недетерминированному поведению программы и получению неверных результатов при проведении моделирования. В случае попадания таких ошибок в реализицию целевой системы, созданная система может оказаться неработоспособной.

В данной работе предлагается подход к автоматическому обнаружению ошибок конкурентной модификации данных в программах на языке SystemC. В основе предлагаемого подхода лежат разработанные алгоритмы статического анализа, обеспечивающие извлечение информации о возможных операциях с объектами программы в параллельных блоках с учетом поведения планировщика SystemC. Предлагаемый подход обеспечивает обнаружение всех ошибок конкурентной модификации данных, имеющихся в программе, при этом могут иметь место ложные обнаружения.

2. Обзор существующих методов

Обнаружение ошибок конкурентной модификации данных может выполняться с использованием динамических и статических методов. Динамические методы обнаружения ошибок требуют инструментирования исходного кода или объектного кода. Инструментирование может применяться для протоколирования информации об использовании переменных процессами программы или для проверки условий возникновения ошибки. Например, в работе [2] инструментирование объектного кода используется для выполнения Lockset-проверок, позволяющих обнаружить ошибки конкурентной модификации данных. В работе [3] модель системы на языке SystemC уровня TLM инструментируется мониторами, генерируемыми на основе спецификации темпоральных свойств, описанных на языке PSL. Динамические методы не гарантируют анализ всех возможных путей выполнения при всех возможных исходных данных и поэтому не обеспечивают полноту результатов обнаружения.

В работе [4] используется комбинированный анализ программ на языке SystemC, включающий статический анализ и проверку модели. Статический анализ используется для извлечения информации о компонентах системы и их взаимосвязях. С использованием проверки моделей выводится условие коммутативности процессов. Описанный в статье подход предлагается использовать для оптимизации времени выполнения SystemC-программ. Указанный подход реализован в средстве Scoot. Данное средство не обнаруживает ошибки конкурентной модификации в явном виде, а лишь вычисляет инварианты перестановки процессов.

Проверка моделей используется во многих работах, посвященных обнаружению ошибок в программах на языке SystemC. В некоторых работах модель программы представляется на входном языке одного из существующих верификаторов [5],[6], в

то время как в других работах предлагаются собственные формализмы и алгоритмы проверки моделей [7],[8].

Статический анализ широко используется для обнаружения ошибок в параллельных программах на языках программирования C, C++, Java и др. Подход к обнаружению ошибок конкурентной модификации данных в многопоточных программах на языке C предлагается в работе [9]. В данном подходе используется система типов и эффектов, позволяющая определять объекты синхронизации, под защитой которых выполняются операции с разделяемыми объектами программы. Подход к обнаружению ошибок синхронизации в многопоточных программах на языке C рассматривается в работе [10]. В предложенном подходе выполняется построение графа совместно достижимых конструкций программы, который уточняется с использованием редукции частичных порядков и Lockset-анализа. Модификация данного подхода для программ с асинхронным запуском задач и пулом потоков рассматривается в работе [11]. Обнаружение ошибок конкурентной модификации данных на основе извлечения инвариантов рассматривается в работе [12].

Подход для обнаружения ошибок конкурентной модификации данных в программах на языке Java предлагается в работе [13]. В основе данного подхода лежит итеративное применение алгоритмов статического анализа, выполняющих анализ указателей, определение разделяемых объектов синхронизации, анализ возможности параллельного выполнения конструкций в разных потоках.

3. Внутреннее представление SystemC-программы

Предлагаемый в данной статье подход использует внутреннее представление программы в виде расширенного графа потока управления. Представление программы является множеством базовых блоков, соединенных дугами условной и безусловной передачи управления. Базовый блок – это цепочка операторов, выполняющихся строго последовательно.

Представление программы использует ограниченное множество типов операторов, в частности, операции обращения к полю структуры и к элементу массива преобразуются к операциям арифметики указателей. Сложные выражения разбиваются на несколько операторов, каждый из которых имеет не более двух операндов и переменную для сохранения результата. Условные выражения в операторах ветвления заменяются переменными, значения которых вычисляются заранее. Для представления конструкций организации параллельного выполнения и синхронизации используются специальные конструкции, перечень которых приведен в таблице 1.

4. Алгоритмы статического анализа

Предлагаемый подход основан на статическом анализе построенного представления программы. Для каждого процесса программы проводится анализ всех возможных путей выполнения в рамках дельта цикла. Используются алгоритмы интервального анализа и анализа указателей [14], которые определяют возможные значения переменных в конструкциях программы. Учет нотификаций, выполненных в данном

Таблица 1. Специальные конструкции

Конструкция	Описание
`run(this, p, f)`	Запустить в процессе `p` функцию `f` Используется для представления `create_thread_process`
`run_update(` `  this, update)`	Выполнить метод `update()` в фазе обновления Используется для представления метода `request_update`
`start()`	Начать симуляцию. Соответствует методу `sc_start()`
`sense(p, e)`	Добавить в список чувствительности потока `p` событие `e` Используется для представления оператора `<<`
`wait(e, t)`	Ожидать список событий `e`, но не более времени `t` Используется для представления методов `wait(t), wait(n, time_unit), wait(e)`
`notify(e, t)`	Отправить событие `e` через время `t` Используется для представления методов `e.notify(), e.notify(t), e.notify(n, time_unit)`

процессе, осуществляется с помощью алгоритма Lockset-анализа. Взаимные влияния процессов учитываются при переходе к следующему дельта-циклу за счет распространения измененных значений разделяемых объектов в другие процессы. Для этого используется разработанный алгоритм определения значений разделяемых объектов. Все перечисленные алгоритмы работают под управлением алгоритма анализа параллельного выполнения, который определяет границы дельта-циклов для каждого процесса и учитывает синхронизацию между процессами.

4.1. Алгоритм Lockset-анализа

Для анализа конструкций синхронизации используется Lockset-анализ. Алгоритм Lockset-анализа вычисляет количество конструкций `notify` на всех возможных путях выполнения в различных процессах программы.

Пусть B_i^j – блок программы, выполняющийся в j-м процессе в текущем дельта-цикле. Обозначим $l(s_k, e_m)$ число конструкций `notify`(e_m) на всех путях выполнения процесса от первой конструкции B_i^j до конструкции s_k включительно (количество нотификаций для любого события перед первой конструкцией блока равно нулю). Набор конструкций `notify` для блока B_i^j определяется с помощью следующих правил (см. рис. 1):

$$[notify(e_m)]_k : \quad \begin{cases} l(s_k, e_r) = l(s_n, e_r) + 1, & r = m \\ l(s_k, e_r) = l(s_n, e_r), & r \neq m \end{cases}, \quad s_n \in Pred(s_k),$$

$$[phi()]_k : \qquad l(s_k, e_r) = \bigcup_{\forall s_n \in Pred(s_k)} l(s_n, e_r), \quad \forall r,$$

$$[if(...)]_k : \qquad \begin{cases} l(s_k, e_r)_{true} = l(s_n, e_r) \\ l(s_k, e_r)_{false} = l(s_n, e_r) \end{cases}, s_n \in Pred(s_k), \quad \forall r,$$

где $phi()$ – конструкция модели, в которой объединяются ветки программы, $Pred(s_k)$ – множество предыдущих конструкций для конструкции s_k.

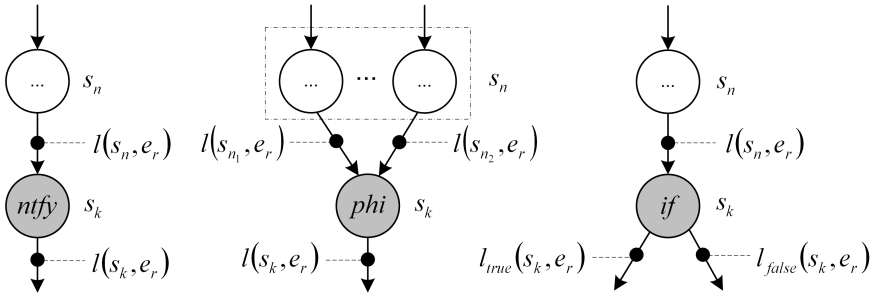

Рис. 1. Правила Lockset-анализа

4.2. Алгоритм анализа параллельного выполнения

Рассмотрим алгоритм анализа параллельного выполнения. Этот алгоритм определяет начало и конец каждой фазы дельта-циклов – фактически определяет блоки программы в текущем дельта цикле. Вычислительная фаза дельта цикла завершается при достижении конструкции `wait` либо конструкции завершения процесса. После этого проверяется наличие немедленных нотификаций для событий, которые ожидаются в процессах, заблокированных на конструкциях `wait`. При наличии таких нотификаций выполняется анализ соответствующих конструкций `wait` в рамках вычислительной фазы текущего дельта-цикла. Этот процесс продолжается до тех пор, пока находятся новые конструкции `wait`, для которых имеются немедленные нотификации.

После того как анализ вычислительной фазы всех процессов завершится, начинается анализ фазы обновления. В фазе обновления выполняется анализ методов `update()`, запущенных с помощью вызовов методов `request_update()` в вычислительной фазе текущего дельта-цикла. После завершения фазы обновления выполняется переход к следующему дельта-циклу: определяются все события, для которых была выполнена нотификация в текущем дельта-цикле, и разблокируются соответствующие процессы.

4.3. Алгоритм определения значений разделяемых объектов

Алгоритмы интервального анализа, анализа указателей и Lockset-анализа выполняются для каждого процесса в отдельности. Учет взаимодействий между потоками

программы осуществляется за счет объединения актуальных состояний программы в отдельных потоках в конце каждого дельта-цикла. Правило вычисления значений произвольного объекта программы o_k выглядит следующим образом:

$$
\begin{aligned}
&o_k \in Def(B_{i_1}^{j_1}) : \\
&V(o_k, B_{i_1}^{j_1})' = \bigcup_{\forall i,j: j \neq j_1, o_k \in Def(B_i^j)} V(o_k, B_i^j) \cup V(o_k, B_{i_1}^{j_1}), \\
&o_k \in Ndef(B_{i_1}^{j_1}) : \\
&V(o_k, B_{i_1}^{j_1})' = \bigcup_{\forall i,j: j \neq j_1, o_k \in Def(B_i^j)} V(o_k, B_i^j) \cup \bigcup_{\forall j \exists i: o_k \in Ndef(B_i^j)} V(o_k, B_{i_1}^{j_1}),
\end{aligned}
$$

где $V(o_k, B_i^j)$ – множество значений объекта o_k в конце блока B_i^j, $V(o_k, B_i^j)'$ – множество значений объекта o_k после завершения дельта-цикла в конце блока B_i^j, $Def(B_i^j)$ – множество объектов программы, которые могли быть модифицированы в блоке B_i^j в данном дельта-цикле, $NDef(B_i^j)$ – множество объектов, которые могли быть не модифицированы в текущем дельта-цикле в блоке B_i^j.

Первое правило учитывает, что новое значение o_k может быть равно значениям, присвоенным в B_i^j в данном процессе, а также значениям, присвоенным в других процессах. Несмотря на то, что изменение значений объекта o_k в одном дельта-цикле из разных процессов считается ошибкой конкурентной модификации данных, необходимо объединять значения из других процессов для сохранения полноты результатов. Второе правило учитывает, что значение объекта o_k может не измениться при условии наличия для каждого процесса хотя бы одной точки завершения, в которой значение o_k могло не модифицироваться. Эти правила применяются для всех блоков дельта-цикла. Правила применяются последовательно ко всем видимым переменным в конце каждого блока.

Работа рассмотренных правил может быть проиллюстрирована следующим рисунком (см. рис. 2). В данном примере определяются значения двух разделяемых объектов: a и b. Переменная a после завершения дельта-цикла в B_1^1 может принимать значения из текущего процесса (значение 1) и из других процессов (значение 3). Значения переменной b не менялись в данном процессе, значения b определяются как все измененные значения из других процессов (значение 1) и текущее неизмененное значение (значение 0).

4.4. Анализ вариантных нотификаций

Одной из проблем при определении параллельных блоков программы являются вариантные нотификации – конструкции `notify`, находящиеся в ветках операторов ветвления. Такие конструкции могут выполняться только на некоторых наборах входных данных программы. Наличие вариантных нотификаций может приводить к различным порядкам выполнения блоков программы из разных процессов.

Для обнаружения некоторых видов ошибок синхронизации, например взаимных блокировок процессов в случае вариантной нотификации, необходимо выполнять

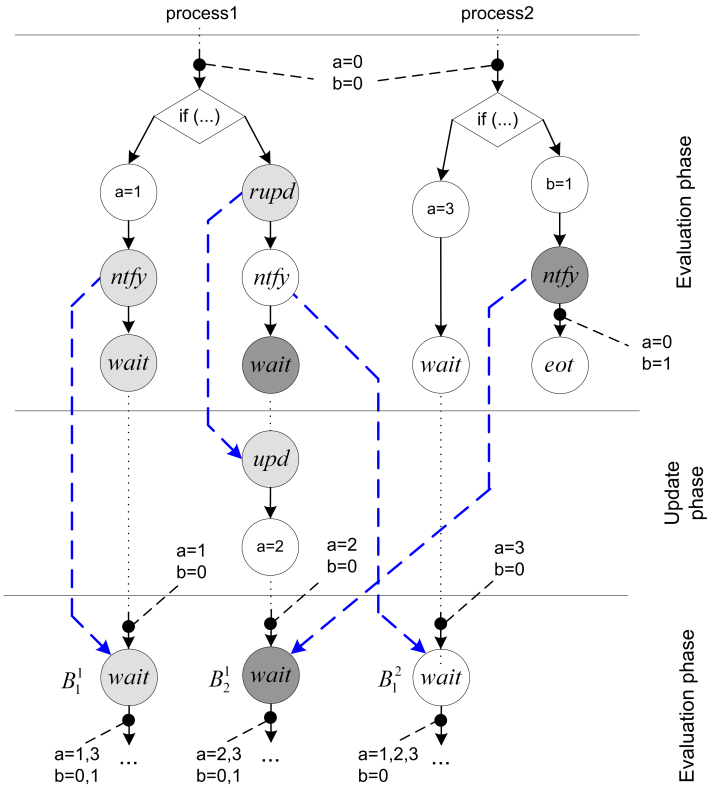

Рис. 2. Пример определения значений разделяемых объектов

раздельный анализ программы при наличии и при отсутствии такой нотификации. Для обнаружения ошибок конкурентной модификации данных вариантные нотификации могут анализироваться совместно. При этом могут возникать ситуации, когда конструкция `wait` может оказаться заблокированной и разблокированной одновременно (см. рис. 3).

Алгоритм совместного анализа вариантов выглядит следующим образом:

- В конце дельта-цикла определяются конструкции `wait`, для которых имеет место неоднозначная нотификация.
- Для каждой такой конструкции `wait` в следующем дельта-цикле создаются две копии: заблокированная и незаблокированная конструкция `wait`, анализ этих и последующих конструкций выполняется независимо, так же как для разных веток процесса программы.
- Выполняется объединение одноименных конструкций `wait` в заблокированном состоянии, задержанных на одинаковое число дельта-циклов.

Предложенный алгоритм выполняет анализ возможных порядков выполнения блоков программы, что обеспечивает учет всех возможных точек синхронизации процессов. При этом выполняется объединение состояний программы в одних и тех же конструкциях `wait`, достигнутых на разных путях в одном дельта-цикле. Последнее действие обеспечивает приемлемую вычислительную сложность алгорит-

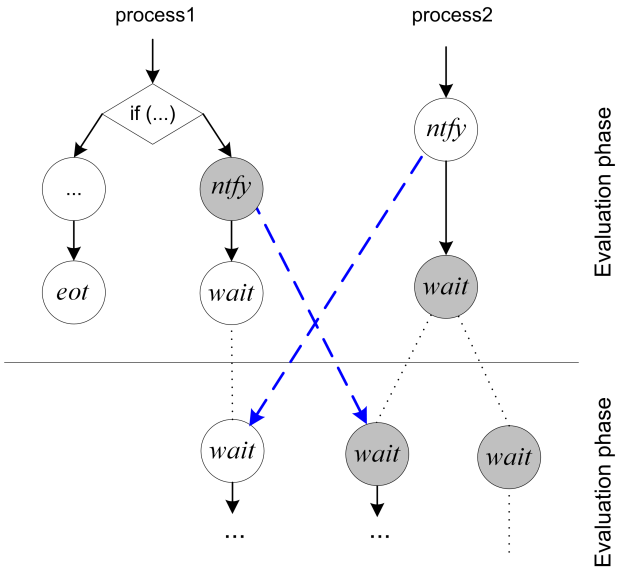

Рис. 3. Пример совместного анализа вариантных нотификаций

ма, которую можно оценить как $O(t^2)$, где t – число проанализированных дельта-циклов. Такая оценка вычислительной сложности следует из того, что для каждой конструкции `wait` на i-м дельта-цикле потребуется проанализировать не более чем i копий этой конструкции (на каждом дельта-цикле создается не более одной копии конструкции `wait`). Таким образом, вычислительная сложность анализа i-го дельта-цикла линейно зависит от номера этого дельта-цикла и может оценивается как $O(i)$, и, следовательно, сложность анализа t дельта-циклов может быть оценена как $O(t^2)$. Вычислительная сложность алгоритма, использующего раздельный анализ вариантов, экспоненциально зависит от числа анализируемых дельта-циклов.

Необходимо отметить, что алгоритм совместного анализа вариантов имеет более низкую точность по сравнению с алгоритмом, использующим раздельный анализ вариантов. Одним из способов повышения точности получаемых результатов является использование зависимостей между значениями объектов программы. В нашем подходе используются линейные зависимости для переменных в отдельных процессах программы. Развитием данного подхода является учет зависимостей между объектами программы из разных потоков.

Другим методом повышения точности является использование зависимостей между объектами программы и наличием нотификации для событий. Такие зависимости позволяют повысить точность анализа конструкций, следующих за конструкцией `wait`.

4.5. Алгоритм обнаружения ошибок

Алгоритм обнаружения ошибок конкурентной модификации данных использует следующее правило:

$$\exists B_{i_1}^{j_1}, B_{i_2}^{j_2} : (B_{i_1}^{j_1} \parallel B_{i_2}^{j_2}) \wedge (Def(B_{i_1}^{j_1}) \cap (Def(B_{i_2}^{j_2}) \cup Use(B_{i_2}^{j_2}))) \neq \emptyset),$$

где $B_{i_1}^{j_1}, B_{i_2}^{j_2}$ – параллельные блоки программы, $Use(B_i^j)$ – множество объектов программы, которые использовались в блоке B_i^j в данном дельта-цикле. Правило позволяет выявить случаи модификации разделяемых объектов, а также случаи модификации и чтения разделяемых объектов в разных процессах.

Пример программы, содержащей ошибку конкурентной модификации данных, приведен на рис. 4. В данном примере блоки $B_{i_2}^2$ и $B_{i_3}^3$ могут выполняться в произ-

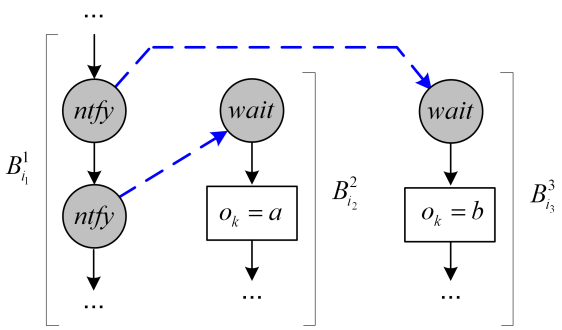

Рис. 4. Пример ошибки конкурентной модификации данных

вольном порядке, при этом в этих блоках выполняется модификация объекта o_k.

5. Оценка эффективности

Показателями эффективности обнаружения ошибок являются точность обнаружения – доля истинных ошибок среди всех обнаруженных ошибок, и полнота обнаружения – отношение числа истинных ошибок к общему числу ошибок в программе.

Для оценки эффективности обнаружения ошибок конкурентной модификации данных были разработаны тестовые примеры программ на языке SystemC. Часть примеров содержит искусственно внесенные ошибки конкурентной модификации данных. В разработанных примерах использованы различные способы синхронизации: на основе ожидания интервала времени, на основе немедленной нотификации событий, на основе дельта-нотификации событий. Некоторые из разработанных тестов включают замаскированные ошибки – ошибки, которые проявляются только при выполнении определенного условия или на некоторых итерациях цикла. В общей сложности было разработано 18 тестов. Размер каждого примера составляет 100-150 строк. Описание групп тестов приведено в таблице 2.

Для анализа тестовых примеров использовался стенд на базе процессора Intel T9300 2.5 ГГц с 2 Гб оперативной памяти. Время анализа каждого примера составило от 5 до 10 секунд. При проведении экспериментальных исследований для каждого тестового примера фиксировалось общее число ошибок, число обнаруженных ошибок, число правильно обнаруженных ошибок. В результате исследований все контрольные ошибки были найдены, при этом ложных ошибок обнаружено не было. Отсутствие ложных обнаружений объясняется относительной простотой используемых тестов.

Таблица 2. Описание тестовых примеров

Наименование группы тестов	Число программ	Число процессов	Число ошибок
с использованием `wait(0, SC_NS)`	4	2	2
с немедленной нотификацией и циклами	6	2-3	4
с дельта-нотификацией	2	2	1
с вызовами request_update	2	3	1
реализации «обедающих философов»	4	2-5	3

Для оценки масштабируемости предложенного подхода проведен ряд экспериментов с реализациями задачи об обедающих философах на языке SystemC. Для оценки зависимости ресурсоемкости от числа анализируемых дельта-циклов были проведены эксперименты для 10 обедающих философов при раздельном и совместном анализе вариантов. Время анализа для различного числа дельта-циклов представлено на левой части рисунка 5. На этом рисунке штрих-пунктирной линией представлены графики для раздельного анализа вариантов, сплошной линией – графики для совместного анализа вариантов.

Для оценки зависимости ресурсоемкости от числа параллельных процессов были проведены эксперименты для различного числа обедающих философов (для каждого философа создается отдельный процесс) при раздельном и совместном анализе вариантов. Время анализа для различного числа процессов представлено на правой части рисунка 5.

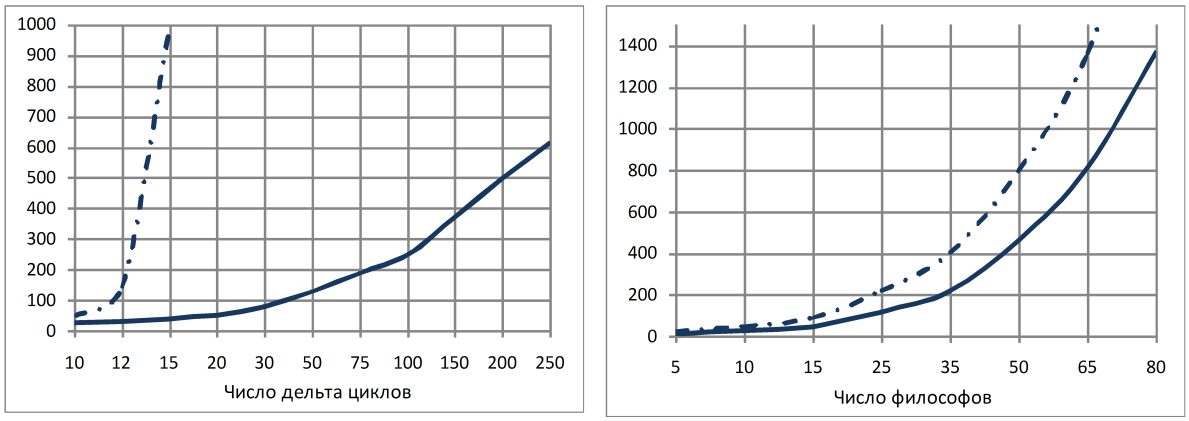

Рис. 5. Зависимость времени анализа от числа дельта-циклов

6. Заключение

В данной статье представлен подход, обеспечивающий автоматическое обнаружение ошибок конкурентной модификации данных в программах на языке SystemC на основе статического анализа. Предлагаемый подход обеспечивает полноту обнаружения ошибок, при этом возможны ложные обнаружения.

Разработанные алгоритмы статического анализа учитывают семантику планировщика SystemC. Поддерживается немедленная нотификация, дельта-нотификация, ожидание следующего дельта-цикла. Для снижения ресурсоемкости предложен алгоритм совместного анализа порядков выполнения блоков программы при вариантных нотификациях. Применение данного алгоритма обеспечивает квадратичную сложность от числа анализируемых дельта-циклов. Рассмотрены пути повышения точности результатов анализа на основе использования зависимостей между объектами программы. Предлагаемый подход и разработанные алгоритмы анализа реализованы в средстве обнаружения ошибок Aegis for SystemC [15].

Для оценки эффективности предложенного подхода были разработаны тестовые примеры, содержащие различные варианты взаимодействия параллельных процессов. Результаты экспериментов позволяют сделать вывод о работоспособности предложенного подхода и возможности его применения для анализа промышленных моделей на языке SystemC.

В настоящее время ведется работа по развитию алгоритмов анализа для поддержки количественного времени. Реализация алгоритмов, учитывающих время, позволит анализировать широкий класс моделей на языке SystemC.

Список литературы

1. 1666-2005 IEEE Standard SystemC Language Reference Manual.

2. Savage S., Burrows M. et al. Eraser: a Dynamic Data Race Detector for Multithreaded Programs // J. ACM Transaction of Computer System. 1997. 15. P. 391–411.

3. Pierre L., Ferro L. Enhancing the Assertion-based Verification of TLM Designs with Reentrancy // 8th International Conference on Formal Methods and Models for Codesign. 2010. P. 103–112.

4. Blanc N., Kroening D. Race Analysis for SystemC using Model Checking // ICCAD. 2008. P. 356–363.

5. Traulsen C., Cornet J., Moy M., Maraninchi F. A SystemC/TLM Semantics in Promela and Its Possible Applications // 14th Workshop on Model Checking Software SPIN. 2007. P. 204–222.

6. Garavel H., Helmstetter C., Ponsini O., Serwe W. Verification of an Industrial SystemC/TLM Model Using LOTOS and CADP // 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design. 2009. P. 46–55.

7. Hojjat H., Mousavi M.R., Sirjani M. Process Algebraic Verification of SystemC Codes // 8th International Conference on Application of Concurrency to System Design. 2008. P. 62–67.

8. Zhang Yu, Vedrine F., Monsuez B. SystemC Waiting-State Automata // 1st International Workshop on Verification and Evaluation of Computer and Communication Systems. 2007.

9. Pratikakis P., Foster J.S., Hicks M. LOCKSMITH: Practical Static Race Detection for C // J. ACM Transactions on Programming Languages and Systems. 2011.

10. Kahlon V., Sankaranarayanan S., Gupta A. Semantic Reduction of Thread Interleavings in Concurrent Programs // 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2009.

11. Kahlon V., Sinha N., Kruus E., Zhang Y. Static Data Race Detection for Concurrent Programs with Asynchronous Calls // 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2009.

12. Terauchi T. Checking Race Freedom via Linear Programming // ACM SIGPLAN conference on Programming language design and implementation. 2008.

13. Naik M., Aiken A. Conditional Must Not Aliasing for Static Race Detection // 34th annual ACM SIGPLAN-SIGACT symposium on POPL. 2007.

14. Itsykson V.M., Moiseev M.Ju., Zakharov A.V. et al. Automatic Defects Detection in Industrial C/C++ Software // 5th Central and Eastern European Software Engineering Conference in Russia. 2009.

15. Aegis Error Detection Tool. `http://www.digiteklabs.ru/aegis/`

Automatic Data Race Error Detection in SystemC Models

Zakharov A.V., Moiseev M.J.

Keywords: SystemC, static analysis, data error race detection

Hardware/software systems simulated by using the SystemC language are usually parallel and, therefore, may contain synchronization errors. One widespread type of synchronization errors is data races. In this paper we propose an approach to data race detection in SystemC programs which is based on the source code static analysis. We have developed some static analysis algorithms that can extract information for data race detection in a SystemC program without quantitative time. These algorithms can detect all the errors that exist in the program. The efficiency of our approach is shown by the evaluation results of the developed tool on a set of test SystemC programs.

Сведения об авторах:
Захаров Алексей Владимирович,
Санкт-Петербургский государственный политехнический университет,
ведущий программист в телекоммуникационном центре ФТК.
Моисеев Михаил Юрьевич,
Санкт-Петербургский государственный политехнический университет,
доцент кафедры компьютерных систем и программных технологий.