УДК 621.396.6

О.В. Гончаровский, Д.О. Гончаровский

Пермский национальный исследовательский
политехнический университет

ТЕСТИРОВАНИЕ ВСТРОЕННЫХ СИСТЕМ

Рассмотрено решение задачи построения тестов для встроенных систем.

Предметом исследования является тестовое диагностирование дискретных устройств. Объект исследования – решение обратной задачи диагноза (построение тестов) для тестового диагностирования встроенных микропроцессорных систем.

Встроенная система – программно-аппаратная система (обычно микропроцессорная система), выполняющая специализированную прикладную функцию. Большинство встроенных систем относится к системам реального времени. Корректность работы таких систем зависит не только от логики вычисления отклика, но и от времени, затраченного на это вычисление (оговаривается время исполнения или крайний срок).

Под тестированием понимают процесс определения технического состояния изготовленного по проекту устройства. Тестирование выполняется подачей тестовых последовательностей на проверяемое устройство.

Выделяется функциональное и структурное тестирование.

Функциональное тестирование выполняет следующие задачи:

– проверка существования и активности всех подсистем;

– проверка спецификации системы (например, количество доступной в системе памяти);

– проверка критических функций системы на базе тестов верификации.

Функциональное тестирование предполагает подачу полного множества значений входных переменных для проверки функций.

Структурное тестирование выполняет задачу определения исправного или неисправного технического состояния. Структурный

тест базируется на рассмотрении минимизированного множества константных неисправностей на линиях схемы. Под неисправностью понимается модель дефектов, т.е. представление дефекта на абстрактном уровне. Модели неисправностей тесно связаны с уровневой моделью устройства. В иерархии проектирования уровень относится к степени абстракции.

Поведенческий или функциональный уровень (часто называемый высшим) имеет минимальные сведения о деталях реализации. На этом уровне устройство описывается либо на языке программирования (например, *C*), либо на языке описания аппаратуры (*HDL*), либо как совокупность функций, потока данных, внешних объектов и памяти данных. Уровень регистровых передач (*RTL*) или логический состоит из таблицы соединений вентилей.

Транзисторный и другие нижние уровни относятся к уровням компонент (*component levels*) и используются при аналоговом тестировании.

Функциональная модель неисправностей, очевидно, не имеет корреляции с производственными дефектами. К ней относится неправильное выполнение языковых конструкций: неисправности присвоения, ветвления и операций.

Для микропроцессоров была введена функциональная модель неисправностей команд. Выбранная команда выполняется некорректно: получается неправильный результат или выполняется другая команда.

Функциональная модель неисправностей играет большую роль в верификации проекта на базе моделирования, чем при тестировании. Исключение составляет функциональная модель неисправностей полупроводниковой памяти из-за простоты функции памяти, что делает возможным на практике исчерпывающий функциональный тест.

Функциональная модель неисправностей может быть использована и для решения задачи определения технического состояния: правильное/неправильное функционирование.

Пусть из-за дефекта некоторая функция устройства выполняется неправильно. Для пользователя есть только два выбора: продолжать работать с устройством, зная, что неисправная функция не используется в данном приложении, или заменить устройство.

Функциональная модель неисправностей имитирует дефект, приводящий к изменению некоторой функции (например, синхронный

счетчик устанавливается в максимальное значение при поступлении тактового импульса). Однако функциональная модель неисправностей может устроить изготовителей встроенных систем (не все функции задействуются в конкретном приложении), но не производителей компонент или *OEM* модулей, для которых важны корректировка техпроцесса при обнаружении дефектов и выполнение всех функций.

Для встроенных систем диагностическая модель может базироваться на описании поведения программной и аппаратной части в одинаковой манере. Среди множества поведенческих форматов выделяются граф управления потоком данных (*Contol Data Flow Graph – CDFG*) и конечный автомат (*FSM*).

Простая текстуальная модель неисправностей – мера покрытия операторов, введенная для тестирования программ. Эта модель сопоставляет потенциальную неисправность с каждым оператором программы. Требуется, чтобы каждый оператор программы был выполнен в процессе тестирования.

Модель неисправностей на основе графа управления потока данных базируется на прохождение путей в *CDFG*, представляющем поведение системы.

Мера покрытия ветвления сопоставляет потенциальные неисправности с каждым из направлений каждого из условий в *CDFG*.

Другая мера – мера покрытия пути более строгая, чем мера покрытия ветвления, так как покрытие путей отражает большое количество путей потока управления. Предполагается, что ошибка ассоциируется с некоторым путем в *CDFG*, и все пути управления должны быть пройдены, чтобы гарантировать обнаружение дефекта.

Модели неисправностей на базе *FSM* – это мера покрытия состояния – все состояния должны быть достигнуты, и мера покрытия перехода – все переходы должны быть выполнены.

Модели неисправностей, ориентированные на специфику приложения, учитывают поведенческие особенности приложения. Для оправдания расходов на разработку и оценку модели неисправностей, ориентированной на специфику приложения, рынок приложения должен быть очень широк, а модель понятна.

Рассмотрим построение тестов для программно-аппаратного совместного тестирования встроенных систем.

Выделяют два класса поисковых алгоритмов: ориентированные на неисправности и ориентированные на покрытие.

Алгоритмы, ориентированные на неисправности: выбирается очередная неисправность, для нее строится тестовая последовательность, которая затем объединяется с уже найденными последовательностями и оценивается на полноту. С целью построения тестов, соответствующих выбранному пути в *CDFG*, сопоставляется множество ограничений, которые необходимо удовлетворить для его прохождения. Логическое программирование в ограничениях (*Constraint Logic Programming, CLP*) способно обрабатывать большой набор ограничений, включающих нелинейные ограничения на булевых и арифметических *переменных*. Процесс удовлетворения ограничениям представляет собой поиск таких комбинаций значений переменных, которые соответствуют ограничениям. Сама проблема удовлетворения ограничениям формулируется следующим образом.

Дано:

1) множество переменных;

2) области определения, из которых могут выбираться значения переменных;

3) ограничения, которым должны удовлетворять переменные.

Требуется найти такие *значения*, присваиваемые переменным, которые удовлетворяют всем заданным ограничениям. Такие задачи могут быть описаны в терминах известной задачи составления расписаний.

При использовании модели *FSM* выбирают циклы переходов – пути, которые проходят через каждый переход автомата хотя бы 1 раз. Циклы переходов генерируются итеративным улучшением уже существующего частичного цикла добавлением кратчайших путей через непокрытые переходы (ограничение подхода – значительный перебор).

Алгоритмы, ориентированные на покрытие: эти алгоритмы направлены на улучшение покрытия без ориентации на специфику неисправностей. Алгоритмы эвристически модифицируют существующий тест для увеличения покрытия, а затем оценивают покрытие неисправностей модифицированными тестами. Если покрытие улучшается, то модификация принимается, в противном случае используется другая эвристика. Обычно метод модификации случайный либо направленный случайный.

Тестирование систем на кристалле (*SOC*) на базе встроенных ядер: в *SOC* на базе ядер системный интегратор разрабатывает пользовательскую логику и объединяет ее с предварительно спроектированными ядрами производителей. Ядро – это обычно написанный на

HDL проект стандартной микросхемы, т.е. *DSP*-, *RISC*-процессор или *DRAM*. Встраиваемые ядра представляют *IP* для защиты, которых производители не раскрывают для системного интегратора детальной информации о структуре. Вместо этого предоставляются тесты, гарантирующие покрытие специфических неисправностей.

Тест *SOC* является простой композицией тестов ядер, теста пользовательской логики и теста *связей*. Часто требуются определенные операционные ограничения (безопасный режим, режим пониженного энергопотребления, режим транзитной передачи), что неизбежно влечет введение режимов доступа и изоляции компонент. Стандарт *IEEE* 1500 был введен для решения проблем тестирования *SOC*. В нем разработчикам *SOC* предлагается механизм доступа, обеспечивающий подключение входов-выходов ядра к источнику и приемнику тестовых данных.

Основанное на модели тестирование программ: тестирование на основе модели (*Model-Based Testing*) – это тестирование программного обеспечения, в котором варианты тестирования (*test cases*) частично или целиком получаются из модели, описывающей некоторые аспекты (чаще функциональные) тестируемой системы (*system under test*).

Модель эталонного поведения проверяемой реализации (*IUT-implementation under test*) является основой для построения теста и анализа результата тестирования.

Модель эталонного поведения для компонент специфицирует поведения интерфейса поставщика услуги, интерфейса запроса услуги и взаимную зависимость действий обоих интерфейсов.

Аспекты тестирования системы [1].

Тест соответствия (*conformance testing*) – поведение системы, соответствующее функциональной спецификации.

Тест производительности (*performance testing*) – скорость выполнения задачи.

Тест ошибкоустойчивости (*robustness testing*) – реагирование системы, если окружение работает не так, как положено.

Тестирование надежности (*reliability testing*) – определение того, как долго система функционирует правильно.

Тестовая гипотеза – предположение о формальной модели реализации.

Отношение реализации (*implementation relation*) – отношение между формальной моделью и спецификацией.

Проверка соответствия состоит в выполнении эксперимента для решения, соотносится ли неизвестная модель реализации со спецификацией в соответствии с отношением реализации. Проверка должна выносить отрицательное решение, только если реализация некорректна, и если реализация некорректна, то должна быть высокая вероятность вынесения решения о некорректности.

Дано – спецификация, алгоритм генерации должен построить контрольную задачу для проверки соответствия на базе рассмотренных ниже поведенческих моделей.

Маркированная система (*labelled transition system*) переходов является разновидностью реактивной системы (*reactive system*) – системы, которая изменяет свои действия, выходы и режим/состояние в ответ на воздействия изнутри или снаружи. В трансформационных же системах (*transformational system*) – отклик по готовности всех входов.

Маркированная система переходов – структура, состоящая из состояний и переходов, отмеченных действиями между ними. Этот формализм используется для моделирования поведения процессов, таких как спецификации, реализации и тестирование, а также как семантическая модель для различных формальных языков (*LOTOS, SDL*).

Маркированная система переходов – это четверка

$$<S,L, \rightarrow,s0>,$$

где S – счетное непустое множество состояний; L – счетное множество меток; $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ – отношение перехода; $s0 \in S$ – начальное состояние.

Метки (маркеры) из L представляют наблюдаемые действия системы, $\tau / \in L$ обозначает ненаблюдаемое внутреннее действие. Переход $(s, \mu, s') \in \rightarrow$ обозначается как $s -\mu \rightarrow s'$.

Входовыходная система переходов (*input-output transition system IOTS*) – маркированная система переходов, для которой множество действий может быть разбито на выходные действия, инициируемые системой, и входные действия, инициируемые окружением $LI \cup LU = L$, $LI \cap LU = \emptyset$, и для которой все входные действия всегда разрешены в любом достижимом состоянии, т.е. когда $s =\sigma \Rightarrow s'$, то $\forall a \in LI$: $s=a\Rightarrow$. $IOTS(LI, LU)$ может представлять как реализацию, так и спецификацию, а $LTS(LI \cup LU)$ – толь-

ко спецификацию: $IOTS(LI,LU) \subseteq LTS(LI \cup LU)$. Входные действия помечают – ?, а выходные – !

Отличие *FSM* от *IOTS*: в *FSM* – строгое разрешение входа $\forall a \in LI$: s'- $a \rightarrow$, а в *IOTS* – слабое разрешение входа, $\forall a \in LI$: $s' = a \Rightarrow$, т.е. допускают разрешение входа через внутреннее действие.

Контрольный пример (*test case*, совокупность тестовых данных программы) является специальной *LTS*, которая выполняется проверяемой программой *SUT*. Эта *LTS* имеет древовидную структуру с висячими вершинами типа *pass* и *fail*. Для формального различения наблюдаемого и введенного состояния покоя δ для контрольного примера используется θ (наблюдаемое состояние покоя).

Контрольный пример есть *LTS* $t = <S, LI \cup LU \cup \{\theta\}, \rightarrow, s0>$, удовлетворяющая свойствам:

1) является детерминированной с конечным поведением;

2) {pass, fail} $\subseteq$ S и init (pass) = init (fail) = ∅;

3) для любого s∈S\{pass, fail} или *init* (*s*) ={ μ}, для некоторого входа μ∈*LI* или *init* (*s*) = $LU \cup \{\theta\}$.

Контрольный пример выполняется одновременно с реализацией. Входы и выходы выполняются синхронно, состояние покоя синхронизируется с помощью действия θ, а внутренние действия реализации выполняются автономно.

Говорят, что *P* проходит набор испытательных программ *T*, если и только если для всех контрольных примеров проверка не ведет к заключению *fail*.

Контрольный пример есть спецификация поведения тестера во время эксперимента над тестируемой реализацией. Поведение такого тестера моделируется как специальный вид IOTS с естественной с заменой входов и выходов. Поэтому разрешаемость входов контрольного примера означает, что все действия из *LU* (множество выходов реализации и входы для контрольного примера) разрешены.

Символическая система переходов (*Symbolic Transition System – STS*) является расширением *LTS* [2]. Расширение связано с введением точного обозначения данных и управления от потока данных (такого, как защищенные переходы), основанного на логике первого порядка. Модель *STS* точно отражает модель *LTS*.

Символическая система переходов – это семерка

$$<L,l0,V,i,I,G, \rightarrow>:$$

где L счетное множество позиций, а $l0 \in L$ – начальная позиция; V – счетное множество переменных позиций (внутренние переменные); i – инициализация позиционных переменных; I – счетное множество переменных взаимодействия (внешние переменные, не пересекются с V); G – конечное множество переключателей (*gate*). Ненаблюдаемый переключатель обозначается как τ, $G\tau=G \cup \{\tau\}$. $G=GI \cup GU$, где GI – множество входных переключателей, а GU – множество выходных переключателей. Арность переключателя $g \in G\tau$, обозначаемая *arity*(g), есть натуральное число. Тип переключателя $g \in G\tau$, обозначаемый *type*(g), есть кортеж длины *arity*(g) различных переменных взаимодействия. *arity*(τ)=0 означает, что ненаблюдаемый переключатель не имеет переменных взаимодействия.

$\rightarrow \subseteq L \times G\tau \times F(V \cup I) \times T(V \cup I) \times L$ – это отношение переключения, где $F(X)$ – множество формул первого порядка (условий перехода), $T(X)$ – множество термов (преобразований). Семантика *STS* определяется привязкой к *LTS*.

Входовыходная символическая система переходов (Input Output – Symbolic Transition Systems IOSTS). *IOSTS* представляет *IOLTS* в сжатой и более выразительной манере с использованием переменных. *IOSTS* образуется из двух частей графа и данных. Часть данных задается разрешимой теорией первого порядка T языка первого порядка L, вместе со структурой M является моделью этой теории. Переменные относятся к переменным V языка L.

Библиографический список

1. Frantzen L., Tretmans J., Willemse Tim A.C. Test Generation Based on Symbolic Specications / Nijmegen Institute for Computing and Information Sciences (NIII) Radboud University Nijmegen. – The Netherlands, 2007.

2. Thomas A. Henzinger, Rupak Majumdar. A Classication of Symbolic Transition Systems // Report No. UCB/CSD-99-1086. Computer Science Division (EECS) / University of California Berkeley. – California 94720, 2007.

Получено 05.09.2011