ТЕСТИРОВАНИЕ ИМИТАЦИОННЫХ МОДЕЛЕЙ

А.А. Малыханов, В.Е. Черненко (Ульяновск)

Введение

Впервые термин «модульное тестирование» («unit testing») появляется в материалах Wintersim в 1988 году [1], однако термин применяется к тестированию среды моделирования, а не собственно модели. Конференция по тестированию программного обеспечения существует с 1988 года [2]. Так, уже в 1988 году на этой конференции обсуждалась проблема организации параллельной разработки имитационной модели и «функциональных тестов» («functional tests») к ней [3].

К 1997 году Balchi в своей обзорной статье [4] уже насчитывает 38 видов тестирования имитационных моделей. Детальный обзор эволюции подходов к тестированию имитационных моделей (ИМ) приводит тот же Balchi в 2017 году в [5].

В научном сообществе тестирование моделей признается необходимым и считается частью верификации и валидации. Однако в публикациях и монографиях обсуждаются в основном методологические аспекты тестирования (например, [26, 11, 12]) или делается больший акцент на валидацию и доказательство адекватности моделей. Все виды проверок достоверности моделей, включая тестирование, хорошо систематизированы в [4]. Там же тестирование имитационных моделей признается сложной задачей, заслуживающей внимания исследователей и практиков ИМ. Особенно критичным становится тестирование моделей, на основании которых принимаются решения, касающиеся многих людей или крупных сумм денежных средств. Так, например, далеко за пределы научного сообщества вышла новость об ошибке в модели распространения COVID-19 в Великобритании [13], приведшей к выбору ошибочной стратегии борьбы с заболеванием. В последнее десятилетие осознание необходимости тестирования и проверки моделей привело к появлению понятия «ответственное моделирование» («responsible modeling») [12, 14]. В тех же источниках приводятся детальные рекомендации по верификации, валидации и тестированию моделей. Вместе с тем, известным фактом является то, что на настоящий момент модульное тестирование – стандарт в области разработки любого коммерческого ПО [15].

Терминология: верификация, валидация, модульное и функциональное тестирование

В [6] все виды проверок имитационных моделей разделяются на верификацию, валидацию и тестирование.

Валидация модели (model validation) – это процесс подтверждения того, что построенная модель достаточно хорошо представляет объект исследования. Иными словами, валидация модели отвечает на вопрос – правильная ли модель используется в исследовании.

Верификация модели (model verification) – это процесс подтверждения того, что модель реализована в среде моделирования именно так, как спроектирована, без искажений. Верификация отвечает на вопрос – правильно ли разработана модель.

Тестирование модели (model testing) – это процесс нахождения неточностей и ошибок модели. При тестировании на вход модели подаются заранее подготовленные наборы входных данных, а на выходе проверяется соответствие выходных данных ожидаемым результатам.

Модульное тестирование (unit testing) – термин, широко использующийся в разработке программного обеспечения, относится к проверке корректности отдельных программных модулей, обычно небольших – классов и функций [20]. Термин

«функциональное тестирование» (functional testing) используется реже и имеет более широкое значение. С одной стороны, во многих контекстах функциональное тестирование эквивалентно модульному тестированию. Однако функциональное тестирование может относиться к тестированию некоторого аспекта функциональности программного обеспечения (ПО), проявляющегося во взаимодействии нескольких модулей (классов или функций). На взгляд авторов, в имитационном моделировании целесообразно использовать именно термин «функциональное тестирование», так как чаще всего тестируется именно корректность взаимодействия нескольких логических модулей.

Во многих инструментах моделирования нет встроенных средств тестирования моделей

В материалах Wintersim также встречаются обсуждения практических способов реализации модульного тестирования в проектах по ИМ (например, [9], [10], [11]). Все рассмотренные публикации, в которых для моделирования используются коммерческие средства ИМ общего назначения, признают отсутствие в них встроенных инструментов модульного тестирования. Большинство таких публикаций посвящено техническим способам реализации модульного тестирования моделей на таких платформах как AnyLogic [17], Simio [18], Arena [19] и других. В публикациях обсуждаются, по сути, пути обхода стандартных средств запуска моделей для целей реализации функциональных тестов разной степени автоматизации. Это косвенно свидетельствует об отсутствии встроенных средств автоматизированного тестирования в указанных инструментах.

Анализ собственного опыта авторов и обзор материалов Wintersim позволяют сформулировать следующие основные причины слабого распространения функционального тестирования в имитационном моделировании:

· развитие инструментов моделирования, направленное на уход от подхода к моделированию как к разработке ПО, и, соответственно, пренебрежение лучшими практиками разработки, к которым относится тестирование;

· наличие в ИМ принципиальной концепции модельного времени усложняет понятие состояния объекта тестирования, увеличивая количество трудозатрат;

· сложно выделить отдельные компоненты моделей (функциональные модули) для тестирования, особенно в случаях, когда модели создаются в режиме визуального редактирования блок-схем и диаграмм;

· при тестировании взаимодействия нескольких компонентов быстро растет количество возможных состояний объединяющей их системы, и, соответственно, объем трудозатрат на тестирование;

· наличие случайных величин в логике моделей требует специальных усилий и подходов, которые выходят за рамки устоявшихся стандартов тестирования ПО;

· специфика разработки ИМ заключается в том, что многие аспекты поведения моделируемого объекта становятся известны только в ходе работы над проектом, таким образом, ранее написанные тесты быстро теряют свою актуальность.

Потребность в тестировании возрастает с ростом сложности и срока использования модели

В небольших проектах с применением ИМ, когда модель используется для однократного ответа на конкретные вопросы, применение тестирования нецелесообразно по следующим причинам:

· часто небольшие модели состоят из стандартных блоков, соединенных в простые композиции без сложной логики их взаимодействия; такие конструкции не нуждаются в тестировании;

· небольшая сложность моделей позволяет быстро замечать ошибки без специальных усилий по тестированию;

· тестирование выполняется вручную единоразово в конце разработки модели, такого тестирования оказывается достаточно.

Однако с возрастанием сложности модели, а также длительности ее активного использования и доработки, сложность поддержания корректности модели возрастает. Так, например, при добавлении в модель нового аспекта поведения агента часто требуется проверить корректность взаимодействия добавляемого аспекта с уже реализованными. По результатам такой проверки часто выясняется, что логика уже реализованных аспектов поведения также нуждается в корректировке. Корректировка уже реализованной логики может привести к внесению в нее ошибок. Такие ошибки выглядят особенно нелепо с точки зрения конечного пользователя модели – после добавления новых аспектов поведения старые сценарии, в которых вроде бы нет таких новых аспектов, начинают почему-то исполняться с ошибками.

Усугубляет ситуацию то, что ошибки могут быть внесены не в логику поведения отдельных агентов, а в порядок взаимодействия агентов в определенной ситуации. В таком случае ошибка может обнаружиться много позже того момента, когда приводящая к ней логика была реализована, и затраты времени на ее исправление значительно возрастают.

Хорошим практическим решением описанной проблемы является создание набора тестов, подтверждающих, что работа уже реализованной логики модели соответствует ожиданиям. В тестировании ПО этот подход называется регрессионным тестированием [20]. На практике становится ясно, что тесты логично добавлять непосредственно после реализации соответствующей функциональности. В этом случае разработчик модели наиболее п**о**лно погружен в контекст разрабатываемой функциональности, и для создания тестов требуется минимум затрат времени. Кроме того, успешное выполнение теста служит своеобразным критерием приемки разработанной или доработанной функциональности.

Наличие тестов позволяет более уверенно добавлять новую функциональность, снижая риск внесения ошибок. Тестирование становится одним из ключевых способов управления сложностью, в особенности при командной разработке и создании моделей в несколько этапов.

Потребность в тестировании сложных моделей также косвенно подтверждается наличием в профессиональном сообществе следующих практик, заменяющих тестирование или служащих схожим с тестированием целям:

· создание «эталонных» сценариев, дающих ожидаемые результаты;

· анализ поведения модели при граничных значениях входных параметров;

· подсчет материального баланса и других контрольных значений.

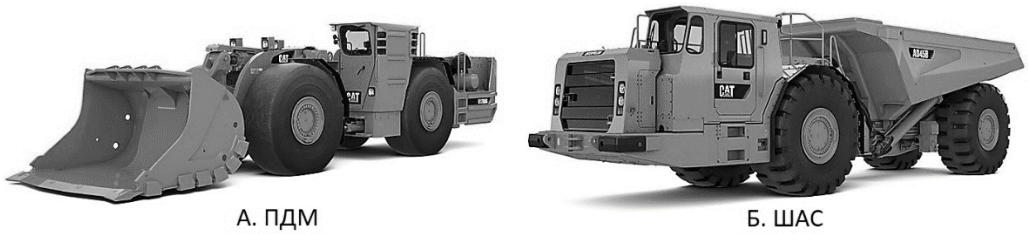
Таким образом, потребность в тестировании заметно возрастает с ростом сложности и срока использования имитационной модели.

Пример организации тестирования одного аспекта модели подземных горных работ

В данной главе приводится пример организации тестирования небольшого подмножества функциональности имитационной модели подземных горных работ. Рассматриваемая модель является частью инструмента планирования и моделирования горных работ MineTwin [21]. Модель реализована на платформе Amalgama Simulation Platform [25], язык разработки – Java, среда разработки – Eclipse [22].

Подземная добыча металлических руд буровзрывным способом состоит из десятков взаимосвязанных процессов, многие из которых воспроизводятся в

рассматриваемой модели [21]. Для целей данной статьи рассмотрение модели ограничивается лишь одним моделируемым процессом – вывозом отбитой руды из выработок в рудоспуски.



А. ПДМ Б. ШАС

Рис. 1. ПДМ и ШАС – моделируемая горная техника

В горных выработках после взрыва образуется отбитая руда, которая вывозится с помощью погрузочно-доставочных машин (ПДМ, рис. 1А) и шахтных автосамосвалов (ШАС, рис. 1Б) по сети подземных дорог к рудоспускам – полостям, из которых руда перемещается далее по транспортной системе рудника. ПДМ, набрав руду в ковш, может сама отвезти руду к рудоспуску и разгрузиться в него или переместить руду к началу выработки и погрузить ее в ШАС. Также ПДМ может выгрузить руду у начала выработки в промежуточное хранилище, откуда руда будет погружена в ШАС той же ПДМ позже. ШАС, в свою очередь, не может загружаться рудой без помощи ПДМ. Обычно в один ШАС помещается 3-4 полных ковша ПДМ с рудой. У операторов ПДМ и ШАС могут случаться перерывы по расписанию, в течение которых работа приостанавливается. В одной выработке может работать одна ПДМ и несколько ШАС. Схема моделируемого процесса показана на рис. 2.

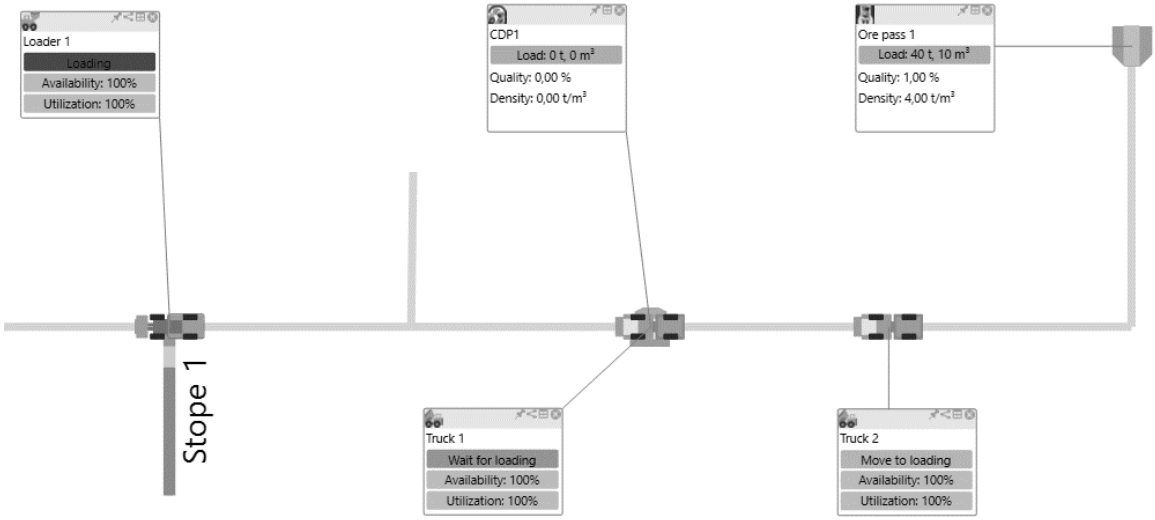


Рис. 2. Схема моделируемого процесса

В имитационной модели ПДМ и ШАС являются подвижными агентами, сеть дорог является средой обитания агентов, выработки, места перегруза и рудоспуски – неподвижными агентами. В рассматриваемой части модели можно выделить следующие уровни функциональности:

1. библиотечный уровень – перемещение агентов по транспортной сети, организация очередей ожидания агентов, выполнение команд по изменению скорости движения и остановке агента; тестирование данной функциональности необходимо, но в настоящей статье не рассматривается;

2. уровень поведения отдельных подвижных типов агентов – погрузка ПДМ в выработке, разгрузка ПДМ и ШАС в рудоспуске, обработка наступления перерыва по

расписанию и события выхода из строя, выполнение ПДМ полного цикла транспортировки из выработки к рудоспуску;

3. уровень взаимодействия агентов – совместная работа ПДМ с одним или несколькими ШАС;

4. уровень формирования заданий на смену по вывозу руды комплексами ПДМ и ШАС.

В таблице 1 приведены примеры тестовых проверок функциональности для уровней 2-4.

Таблица 1. Пример списка тестовых проверок (фрагмент)

Уровень	Тестируемая функциональность	Проверка
2	Работа ПДМ	ПДМ приезжает к выработке и выполняет один рейс в рудоспуск
2	Работа ПДМ	ПДМ приезжает к выработке и выполняет один рейс в точку перегруза
2	Работа ПДМ	ПДМ прерывается на перерыв по расписанию между двумя рейсами в точку перегруза
2	Работа ПДМ	ПДМ прерывается на период выхода из строя после окончания рейса в точку перегруза
…		
2	Работа ШАС	ШАС выполняет один рейс из точки перегруза в рудоспуск
2	Работа ШАС	ШАС прерывается на перерыв по расписанию между двумя рейсами в рудоспуск
	Работа ШАС	ШАС прерывается на период выхода из строя после окончания рейса в рудоспуск
…		
3	Взаимодействие ПДМ и ШАС	ПДМ и 1 ШАС вывозят X тонн руды, работая вместе и используя точку перегруза
3	Взаимодействие ПДМ и ШАС	ПДМ и 2 ШАС вывозят X тонн руды, работая вместе и используя точку перегруза
…		
4	Уровень формирования заданий	Если в выработке есть руда и есть доступные ПДМ и ШАС, то генерируется задание на вывоз руды к ближайшему рудоспуску с использованием доступных ПДМ и ШАС
…		

Тестирование с помощью библиотеки JUnit

Тестирование организуется с помощью специализированной библиотеки JUnit [24]. Эта библиотека широко используется в тестировании ПО на языке Java и имеет развитую поддержку в средах разработки, в частности, в Eclipse [22]. Применение библиотеки JUnit для тестирования имитационных моделей не является новым подходом и обсуждается, например, в [8].

Библиотека JUnit компилирует и исполняет код Java-классов тестов. Методы этих классов, помеченные специальным образом, являются тестовыми методами. Каждый тестовый метод может содержать одно или несколько контрольных

выражений. Тест считается пройденным, если все контрольные выражения соответствующего тестового метода оказались истинными по результатам вычисления. В противном случае тест считается не пройденным, и пользователю показывается соответствующее сообщение в среде разработки.

Каждая тестовая проверка соответствует одному тестовому методу JUnit. Каждый тестовый метод содержит несколько контрольных выражений, пример тестового метода приведен на листинге 1.

Листинг 1.

```
// Добавить еще один тест
addTest(    // Использовать этот файл сценария для теста
            testsFolder + "Cross-docking\\2. Cross Docking 1L 2T.xlsx",
            // Название теста
            "Cross-docking 1L 2T: Tonnes unloaded in orepass",
            // Название группы тестов
            groupName,
            // По окончанию моделирования в рудоспуске №0 должно оказаться 100 тонн руды
            m -> Compare.equalTo(m.getOrePasses().get(0).getBunker().getStock(), 100));
```

В среде разработки Eclipse разработчик может запускать прогон функциональных тестов и отображение результатов в отдельном окне (рис. 3). Такая функциональность дает возможность в процессе разработки оперативно убеждаться в корректности вносимых в логику модели изменений.

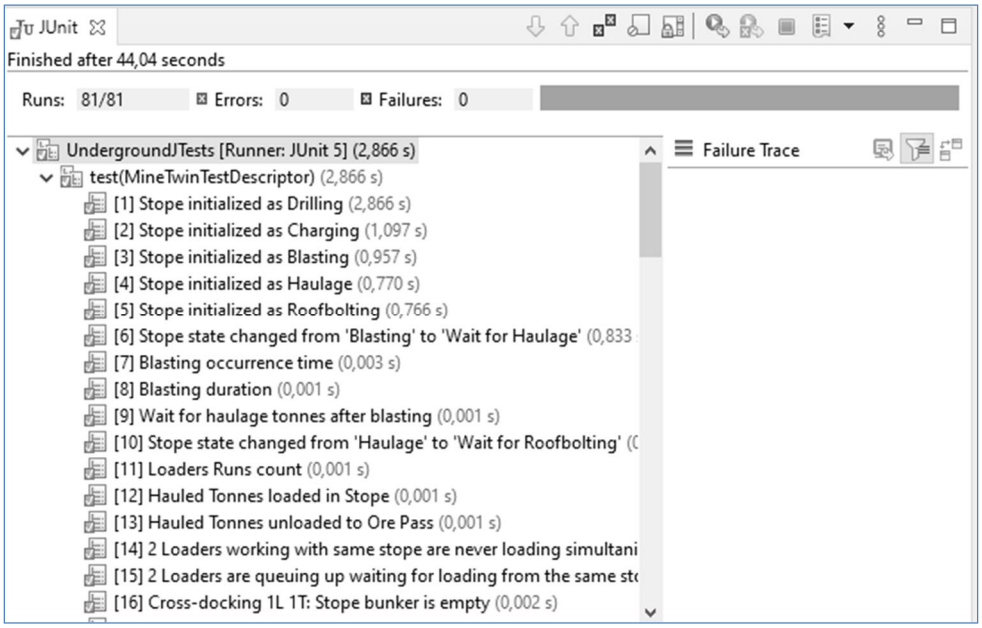


Рис. 3. Окно результатов функционального тестирования в среде Eclipse

Тестовое приложение с анимацией

Практика показывает, что запуск тестов в среде разработки значительно ускоряет процесс создания моделей за счет быстрого обнаружения ошибок, вносимых в ходе разработки в уже реализованную функциональность моделей. В некоторых случаях сообщения о некорректном тесте достаточно, чтобы быстро локализовать и исправить ошибку. Однако некоторые ошибки сложно обнаружить только по сообщению о некорректном тесте. В этом случае самый эффективный метод – запустить тестовую модель с анимацией и визуально проследить за ее некорректным поведением. Затем может потребоваться запуск модели в режиме отладчика и ручная трассировка изменения состояния переменных модели.

Такую функциональность невозможно реализовать средствами JUnit, так как он не предоставляет возможность запускать приложения с графическим пользовательским интерфейсом. Таким образом, для каждого теста появляется необходимость реализовать возможность его запуска с анимацией и в любом масштабе модельного времени. В рассматриваемом проекте было реализовано отдельное приложение, содержащее список всех функциональных тестов и позволяющее запускать любой тест с анимацией. Внешний вид тестового приложения показан на рис. 4.

Приложение при запуске автоматически просматривает все реализованные в модели функциональные JUnit-тесты и отображает их список в одном из своих окон. Разработчик выбирает тест для запуска и может управлять масштабом модельного времени так же, как и в основной имитационной модели.

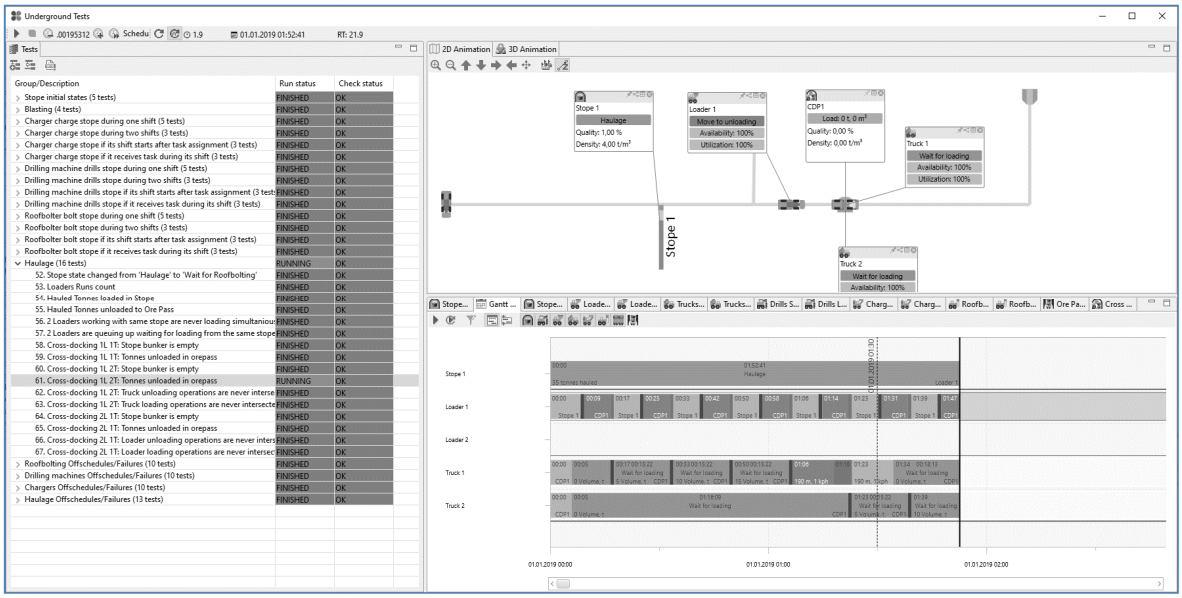


Рис. 4. Внешний вид тестового приложения

Обсуждение опыта тестирования сложных имитационных моделей

Имитационная модель подземных рудников крупного российского предприятия разрабатывается и развивается авторами более 5 лет. В настоящее время модель воспроизводит работу 16 видов горной техники. Регулярно возникает необходимость добавления новых видов горной техники или новых аспектов ее функционирования. Первые 3 года модель не имела функциональных тестов, которые начали добавляться в последние 2 года, и в настоящее время модель содержит 560 функциональных тестов. Такая история развития модели позволяет сравнить скорость разработки (таблица 2) похожих аспектов функциональности до и после появления тестов.

Таблица 2. Сравнение скорости разработки аспектов функциональности модели

Аспект функциональности	Использовалось ли тестирование	Длительность разработки, недель	Количество итераций верификации с заказчиком
Моделирование зарядных машин	Нет	6	5
Моделирование кровлеоборщиков	Да	2	2

Очевидно, что на рост скорости разработки влияют не только факторы, связанные с внедрением тестирования. Например, вклад в ускорение могло внести

лучшее понимание разработчиками предметной области, достигнутое по ходу работы над моделью. Однако наличие и использование тестов стало, на взгляд авторов, решающим фактором, повлиявшим на повышение скорости разработки. Такой вывод сделан на основании следующих фактов:

· в первом случае, при разработке функциональности моделирования зарядных машин, большое количество итераций верификации с заказчиком было обусловлено тем, что при добавлении требуемой функциональности вносились ошибки в уже реализованную функциональность, что делало невозможным завершение верификации; в то же время все ошибки, внесенные в логику модели во втором случае, оперативно обнаруживались с помощью тестов и исправлялись еще до обсуждения с заказчиком;

· функциональность моделирования кровлеоборщиков выполнялась одновременно с созданием тестов, при этом многие тесты были типовыми и разрабатывались по образцу тестов для оборудования других типов; это позволило воспроизвести большинство возможных состояний кровлеоборщиков, не дожидаясь, пока такие состояния встретятся в реальных моделях или при верификации с заказчиком.

Практика применения тестирования при создании нескольких больших имитационных моделей (десятки типов агентов, десятки состояний каждого агента) показывает, что наиболее сложным аспектом тестирования является не разработка самих тестов, а проектирование модели таким образом, чтобы ключевые алгоритмы модели можно было выделить в отдельные, независимые от других алгоритмов, модули. Также значительные усилия тратятся на то, чтобы обеспечить возможность запускать модель с минимально необходимым набором входных данных для воспроизведения каждого тестируемого алгоритма. Такая возможность должна быть заложена в архитектуре системы моделирования. Публикации [12] и [14] уделяют значительное внимание именно вопросам декомпозиции моделей, ориентированной на тестирование.

Обобщение опыта тестирования имитационных моделей позволяет сформулировать требования к средствам их тестирования:

· возможность запуска отдельных компонентов моделей в отрыве от более широкого контекста модели;

· интеграция среды тестирования со средой разработки моделей;

· возможность пакетного запуска тестов в автоматическом режиме.

Приведенные выше требования являются универсальными и могут относиться к тестированию любого ПО, однако специфика имитационного моделирования диктует необходимость выполнения дополнительных, специфичных для ИМ, требований:

· проверка условий тестирования в различные моменты модельного времени;

· учет стохастических экспериментов;

· возможность запуска тестов с визуализацией.

Наличие механизмов функционального тестирования дает разработчикам ИМ следующие преимущества:

· уверенность в сохранении корректности уже реализованной функциональности при изменении и расширении модели;

· ускорение процесса верификации моделей за счет возможности быстро отличать ошибки реализации от концептуальных ошибок декомпозиции предметной области или постановки задачи;

· возможность легко объяснить то, как работает модель, показывая отдельные аспекты ее поведения на примере функциональных тестов;

· постоянный стимул находить и выносить в отдельные тестируемые модули похожие алгоритмы, повышая «сопровождаемость» модели.

Современные средства автоматизации тестирования, используемые в средах разработки ПО общего назначения (например, Eclipse), предоставляют функциональность, позволяющую разработчикам получить дополнительные преимущества от внедрения функциональных тестов:

· определение, какие части кода тестируемой модели покрыты функциональными тестами.

· мутационное тестирование [23] – оценка качества тестов с помощью внесения случайных изменений в тестируемый код; тест считается качественным, если при внесении случайных изменений в тестируемый код тест реагирует на них и начинает выдавать отрицательный результат;

· автоматизированное тестирование – автоматический прогон всех тестов перед формированием версии модели, передаваемой заказчику; в случае отрицательного результата хотя бы одного теста модель считается непригодной для использования; такой прием применяется как дополнительный этап контроля качества модели.

Очевидно, внедрение тестирования требует дополнительных трудозатрат на создание имитационных моделей. Кроме того, необходимо отметить, что внедрение тестирования также сопряжено с менее очевидными трудностями, например:

· потеря актуальности тестов из-за несвоевременного обновления вслед за обновляемой функциональностью;

· ошибки в самих тестах, требующие дополнительного времени на исправление;

· повышенные требования к качеству декомпозиции предметной области и модульности логики моделей, и соответственно, к квалификации разработчиков ИМ.

Заключение

Потребность в тестировании имитационных моделей возрастает с ростом их сложности и срока использования, а встроенных средств функционального тестирования моделей в популярных инструментах ИМ нет. При создании моделей не в средах ИМ, а в инструментах программирования общего назначения имеется возможность использовать подходы к тестированию, являющиеся стандартными в индустрии разработки ПО. Однако тестирование имитационных моделей имеет свои особенности, отличающие его от тестирования ПО в целом. Эти особенности хорошо теоретически освещены в научной литературе, однако требуют осмысления и обобщения для создания практических рекомендаций по тестированию моделей.

Практика показывает, что даже при наличии технических средств тестирования наиболее сложным аспектом тестирования ИМ является не разработка самих тестов, а проектирование моделей таким образом, чтобы ключевые алгоритмы модели можно было выделить в отдельные, независимые от других алгоритмов, модули.

Тем не менее, применение функционального тестирования позволяет разработчикам контролировать сложность даже больших ИМ и избегать снижения скорости разработки по мере роста сложности моделей. Наличие тестирования открывает дополнительные возможности по анализу логики моделей, такие как расчет степени покрытия тестами функциональности модели и оценка качества тестов с помощью мутационного тестирования.

Представляется целесообразным внедрение тестирования в практику работы разработчиков ИМ, в особенности в проекты по разработке больших и сложных имитационных моделей. Это, в свою очередь, требует наличия встроенных или добавления поддержки внешних средств тестирования в популярные инструменты и среды моделирования.

Литература

1. **Darrel A. Quick, Mark A. Roth.** A development methodology for adding map-based graphics to the theater war exercise // Proceedings of the 1988 Winter Simulation Conference, P. 723-730.
2. https://ieeexplore.ieee.org/xpl/conhome/213/proceeding.
3. https://ieeexplore.ieee.org/document/207756.
4. **Osman Balci.** Verification, validation and accreditation of simulation models // Proceedings of the 1997 Winter Simulation Conference, P. 135-141.
5. **Robert G. Sargent, Osman Balci.** History of verification and validation of simulation models // Proceedings of the 2017 Winter Simulation Conference, P. 292-307.
6. **Osman Balchi.** Principles and techniques of simulation validation, verification, and testing // Proceedings of the 1995 Winter Simulation Conference, P. 147-154.
7. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5007758/.
8. **Nicholson Collier, Jonathan Ozik.** Test-driven agent-based simulation development // Proceedings of the 2013 Winter Simulation Conference, P. 1551-1559.
9. **David W. Mutschler.** Language based simulation, flexibility, and development speed in the joint integrated mission model // Proceedings of the 2005 Winter Simulation Conference, P. 1190-1197.
10. **Andreas Klemmt, Jens Kutschke, Christian Schubert.** From dispatching to scheduling: challenges in integrating a generic optimization platform into semiconductor shop floor execution // Proceedings of the 2017 Winter Simulation Conference, P. 3691-3702.
11. **James T. Sawyer, David M. Brann.** How to test your models more effectively: applying agile and automated techniques to simulation testing // Proceedings of the 2009 Winter Simulation Conference, P. 968-978.
12. https://www.sciencedirect.com/science/article/pii/S1755436520300451.
13. https://www.nature.com/articles/d41586-020-01003-6.
14. **James, William.** Rules for responsible modeling, 4th ed. - ISBN 0-9683681-5-8.
15. **Vinay Kulkarni, Tony Clark.** Toward adaptive enterprises using digital twins // Proceedings of the 2019 Winter Simulation Conference, P. 60-74.
16. **Robert G. Sargent.** Verification and validation of simulation models: an advanced tutorial // Proceedings of the 2020 Winter Simulation Conference, P. 16-29.
17. https://www.anylogic.ru/.
18. https://www.simio.com/.
19. https://www.rockwellautomation.com/ru-ru/products/software/arena-simulation.html.
20. **Olan, Michael.** (2003). Unit testing: Test early, test often. Journal of Computing Sciences in Colleges - JCSC. 19.
21. http://mine-twin.ru/.
22. https://www.eclipse.org/ide/.
23. **Falah, Bouchaib & Hamimoune, Soukaina.** (2016). Mutation Testing Techniques: A Comparative Study. 10.1109/ICEMIS.2016.7745368.
24. https://junit.org/junit5/.
25. https://www.amalgamasimulation.ru/.
26. **James T. Sawyer, David M. Brann.** How to build better models: applying agile techniques to simulation // Proceedings of the 2008 Winter Simulation Conference, P. 655-662.