Обзор подходов управления согласованностью артефактов разработки ПО при использовании UML*

В. А. Зверева
vera@tercom.ru

Д. В. Кознов
dim@dk12687.spb.edu

А. С. Бережной
alexander.berezhnoy@gmail.com

В процессе создания ПО создается и используется множество разнообразных артефактов: требования, модели анализа и проектирования, программный код, документы и т.д. Управление согласованностью (consistency management) всех этих артефактов разработки — важная задача, способствующая поддержанию концептуальной целостности разработки ПО. В данной работе представлен обзор методов поддержки согласованности UML-моделей: между собой, с требованиями и с программным кодом. Основной акцент делается на автоматизированных подходах.

Введение

В процессе разработки программного обеспечения создается и используется множество различных артефактов: спецификации требований, визуальные модели, программный код, тесты, документация. Управление согласованностью (consistency management) всех этих артефактов друг с другом на протяжении всего процесса разработки является актуальной и сложной задачей. Этой тематике посвящены многочисленные исследования, различающиеся типом рассматриваемых артефактов, видами деятельности процесса разработки, пониманием характера задачи и методами решения.

В данной статье приводится обзор работ, посвященных задаче управления согласованностью при использовании языка UML. UML [68] является фактическим стандартом визуального моделирования при разработке ПО. С его помощью создаются разнообразные модели ПО, представляющие разрабатываемую систему с различных точек зрения и в разные моменты ее существования, предназначенные для использования разными специалистами: архитекторами, тестерами, менеджерами, инженерами по оборудованию и т. д. Часто требуется, чтобы эти модели не просто единожды выполнили какую-либо задачу и были после этого забыты, но поддерживались в актуальном состоянии (так называемая long-term стратегия использования UML [54]), были связаны как друг с другом, так и с иными представлениями ПО. В связи с этим задача управления согласованностью является, во-первых, достаточно сложной, во-вторых — актуальной.

В [78] делается попытка определить задачу управления согласованностью в общем виде и рассмотреть все имеющиеся подходы к ее решению. Однако рассматриваются главным образом формальные методы, не рассмтариваются разнообразные предметные области программной инженерии, где данная задача возникает на практике. В [26] приводится подробный обзор методов управления согласованностью UML-моделей, однако эта задача не рассматривается для отношения между UML-спецификациями и другими моделями ПО. В [73] дается представление о частном случае задачи управления согласованностью разных артефактов — итеративной разработке (round-trip engineering) и подходах в данной области. Еще один частный случай управления согласованностью — преобразование (transformation) UML-моделей в рамках MDA-подхода[1], обзор и классификация которых дается в [23].

Задача данного обзора — рассмотреть различные подходы к управлению согласованностью UML-спецификаций как друг с другом, так и относительно других артефактов разработки: программ-

*Исследование проводилось при финансовой поддержке МинРосНауки (грант 2006-РИ-19.0/001/248).

[1] Model Driven Architecture — стандарт международного комитета OMG, описывающий методологию разработки ПО на основе визуального моделирования [67].

ного кода и требований. Мы приводим и обсуждаем имеющиеся в литературе определения основных понятий этой области, рассматриваем академические подходы, а также практические контексты, в которых тема согласованности UML-моделей востребована и реализована в коммерческих инструментальных средствах: генерация кода по моделям, циклическая разработка моделей и кода (round-trip engineering), возвратная инженерия, трассировка требований (requirement traceability). При этом мы рассматриваем только те подходы, которые оперируют с формально определенными артефактами и предлагают автоматизированные методы управления согласованностью. Мы не затрагиваем задачу организации взаимодействия участников проекта в целях предотвращения или разрешения несогласованности, которая относится к области CSCW (Computer Supported Cooperative Work) [21].

Работа имеет следующую структуру. В первом разделе мы приводим краткий исторический обзор данной области. Далее, во втором разделе, дается определение управлению согласованности и другим важным понятиям. В третьем разделе приводится обзор подходов к управлению согласованностью при использовании UML: раздел 3.1 посвящен согласованности UML-моделей между собой, в разделе 3.2 дается представление о задаче управления согласованностью UML-моделей и требований, в разделе 3.3 обсуждается согласованность UML-моделей и программного кода. Наконец, в четвертом разделе мы делаем выводы о перспективах промышленного использования методов управления согласованностью.

1. История проблемы

Задача управления согласованностью информации и, в частности, различных артефактов разработки ПО нашла отражение в литературе до появления UML (1997 год) и помимо UML. Известен ряд работ, посвященных устранению несогласованности в базах знаний экспертных систем и системах логического вывода [36, 77]. Несогласованность понималась там как наличие в системе противоречащих друг другу утверждений. Давно известна и активно исследуется задача поддержки эволюции схем баз данных и контроля версий с точки зрения сохранения данных, внесенных в систему, схема которой впоследствии изменилась, а также сохранения работоспособности унаследованных приложений[2]. Ряд работ посвящен проблеме согласованности спецификаций ПО, заданных при помощи различных формальных нотаций (Z, LOTOS) [13, 14, 15, 24]. В [78] представлена попытка обобщить задачу управления согласованностью артефактов разработки ПО на произвольные артефакты.

Данная задача стала активно исследоваться применительно к UML после его появления и широкого практического распространения. По этой теме в 2002–2004 годах, в рамках самой авторитетной конференции по UML — International Conference on the Unified Modelling Language — the Language and its Applications — проводился специальный симпозиум Consistency Problems in UML-based Software Development. Однако широкого практического использования предложенные методы пока не нашли, что заставляет критически рассмотреть данную область и сконцентрироваться на выявлении причин этого явления.

2. Базовые определения

2.1. Согласованность/несогласованность

Рассмотрим, как дается определение несогласованности набора артефактов разработки в [78, 77], где информационные артефакты разработки называются моделями ПО. Модель ПО определяется как множество формул над некоторым словарем констант и предикатных символов. Для модели определяется интерпретация, которая состоит из пары: предметной области и полного морфизма, отображающего каждый константный символ модели в элемент предметной области и каждый предикатный символ степени n — в n-арное отношение элементов предметной области. Далее приводится определение четырех типов пересечений элементов моделей. Оно дается для пары моделей, для каждой из которой неформально определена интерпретация. Два элемента называются *непересекающимися*, если пересечение их интерпретаций пусто. Пересечение элементов называется *полным*, если образы их интерпретаций совпадают. *Включающее* пересечение имеет место, когда образ одного элемента строго включает образ другого, а *частичное* пересечение означает наличие непустого пересечения образов элементов и

[2] http://www.altova.com/products/umodel/uml_tool.html.

непустых разностей этих образов. Далее, в [77] доказывается ряд формальных свойств отношений пересечения, а также приводится алгоритм преобразования отношений пересечения в логические формулы. Две модели ПО называются *несогласованными относительно некоторого правила согласованности*, если из совокупности утверждений, представленных в виде формул отношений пересечения и аксиом равенства, выводимо логическое отрицание правила согласованности. Чуть иначе, как «отсутствие допустимой интерпретации модели», несогласованность определяется в [11, 76].

Такое определение хорошо подходит для методов управления согласованностью, в которых применяются системы логического вывода. Теоретически такие подходы могут применяться в контексте UML при надлежащем преобразовании UML-моделей в спецификации, выраженные в логике предикатов первого порядка. Однако, ввиду ограниченной применимости логических систем для решения практических задач, определения подобного рода оказываются слабо связанными с реальными ситуациями и методами управления согласованностью.

В [12] рассматривается задача согласованности спецификаций на языках Z и LOTOS. Согласованность двух спецификаций проверяется следующим образом. Сначала спецификацию на LOTOS транслируют в язык Z. Потом две спецификации объединяют в третью, которую проверяют на противоречивость. Объединение спецификаций и анализ непротиворечивости результата могут происходить разными способами. Соответственно согласованность бывает разных типов. Все виды согласованности образуют иерархию по отношению вложенности. Такое понимание согласованности оказывается осмысленным в связи с некоторыми особенностями языков Z и LOTOS. Его применение в другом контексте представляется затруднительным.

Сходное понимание несогласованности можно найти и в [31, 35], где также предлагается транслировать две спецификации в третью (под последней понимается общая семантическая область), и там проанализировать отношения между исходными спецификациями. Однако, как отмечается в [73], такие переводы на практике трудноосуществимы.

Итак, формальные определения согласованности/несогласованности оказываются существенно ограниченными в применении. Другой подход — неформальные определения. Например, в [26] несогласованность понимается как «отсутствие гармоничного един-

ства между частями»; в [58] несогласованность определяется в контексте проектирования ПО как «наличие противоречащих друг другу архитектурных решений».

В заключение мы рассмотрим определение согласованности/несогласованности, данное в [73], которое, на наш взгляд, наиболее адекватно для практического использования. Там это понятие рассматривается в контексте разработки артефактов, а не изолированно, само по себе. Предлагается определить, в каких состояниях артефакты согласованы (например, диаграммы и программный код, сгенерированный по ним, согласованы сразу после генерации), а также те действия, которые приводят к потере согласованности (например, изменение в диаграмме расположения элементов не приведет к потере ее согласованности со сгенерированным кодом, а удаление и добавление той или иной конструкции — может). Далее следует определение шагов по преодолению несогласованности.

Будем применять понятие согласованности/несогласованности не только к паре артефактов, но и к одному артефакту: в смысле его внутренней непротиворечивости, корректности, соответствию определенным для него правилам согласованности. Например, формально определенный синтаксис языка программирования задает правила внутренней согласованности для программы, создаваемой на этом языке. Язык OCL определяет правила согласованности UML-спецификации [68]: те из них, которые не удалось выразить в синтаксисе метамодели.

Как в случае пары артефактов, так и в случае одного проблема одна и та же: ПО изменчиво, и вносимые в него изменения должны быть выполнены корректно. Кроме того, мы вынуждены рассматривать вместе оба случая, поскольку часто в них используются одни и те же методы. Это является наследием формальных методов: математической логики, где несогласованность сводится к доказательству противоречивости некоторой логической формулы или трансляции пары спецификаций в третью и сведение задачи их согласованности к внутренней согласованности одной (этой третьей) спецификации. Такая формула может выражать правило согласованности как двух артефактов, так и одного (опять-таки, можно вспомнить UML и OCL).

2.2. Вспомогательные понятия

Теперь мы определим ряд понятий, относящихся к задаче управления согласованностью. При этом мы не ставим своей целью дать им формальные определения, так как такие определения часто оказываются не в состоянии охватить все многообразие реальных ситуаций и могут применяться лишь в достаточно узком контексте. Вместо этого мы приводим для каждого понятия его описание и обсуждение, стремясь наиболее полно раскрыть содержание. Итак, рассмотрим:

- отношение идентификации;
- алгоритм обнаружения несогласованности;
- правило согласованности;
- действие по восстановлению согласованности;
- управление согласованностью/несогласованностью.

Прежде всего, необходимо ввести понятие *отношение идентификации*, т. е. некоторое правило, позволяющее говорить о том, что два артефакта интерпретируют (определяют) одно понятие или по крайней мере два пересекающихся. Это означает, что мы находим способ четко определить, что элемент α первого артефакта значит то же, что элемент β второго. Однако на практике определить отношение идентификации оказывается непросто: часто не удается «красиво» отобразить элементы одного артефакта на элементы другого, образы из одной спецификации оказываются «размазанными» по другой.

В качестве примера рассмотрим диаграмму состояний переходов и соответствующий ей код на языке C. Предположим, что на C конечный автомат, изображенный на диаграмме, реализуется с помощью двух вложенных операторов `switch`: внешнего — по состояниям, внутри каждой его ветки — по событиям, принимаемым и обрабатываемым в состоянии. Тогда, например, конструкции «состояние A» в программном коде будет соответствовать, во-первых, константа в определении перечислимого типа множества всех состояний, во-вторых, метка в операторе `switch`, который реализует конечный автомат в программном коде (`case A:`), в-третьих — все вхождения константы при определении следующего состояния в

переходах-обработчиках, переводящих автомат в состояние (например, `goto A`). Мы видим, что образ конструкции «состояние A» не является одним оператором языка C, а «размазан» и состоит из подконструкций (константа в определении перечислимого типа и метка ветки оператора `switch`), а также множества операторов `goto`.

В случае такой «размазанности» непросто автоматически вычислять отношение идентификации для отдельных элементов, что актуально при изменении артефактов и последующего восстановления согласованности. Возникают также проблемы с обращением этого отношения. Для преодоления этих проблем часто приходится хранить в артефактах дополнительную информацию [73] или привлекать для принятия решения человека. Отношение идентификации может быть обратимым или нет, хорошо устойчивым к изменению артефактов или нет. Когда мы говорим о внутренней согласованности одного артефакта, то отношение идентификации нам не нужно.

Определим *алгоритм обнаружения несогласованности* как вычислимую процедуру, которая по двум артефактам и отношению идентификации решает, являются ли два данных артефакта согласованными. Артефакты называются согласованными или несогласованными в зависимости от результата работы алгоритма (согласованность и несогласованность оказываются, таким образом, взаимоисключающими понятиями). Когда речь идет о внутренней согласованности одного артефакта, то алгоритм обнаружения несогласованности «работает» с одним артефактом. Как обсуждалось выше, алгоритм обнаружения несогласованности может оказаться тривиальным, если следовать подходу, изложенному в [73], т. е., когда мы постоянно следим за состоянием наших артефактов.

Правилом согласованности называется утверждение в некоторой формальной системе, интерпретация которой в соответствии с ее семантикой определяет алгоритм обнаружения несогласованности. Формальной системой для формулирования правил согласованности может служить декларативный язык (например, OCL [34, 41, 51, 57, 58]), язык логических утверждений [18, 61, 71, 78], образец графа [8] и т. д. Важно отметить, что в нашем случае правило согласованности всегда вычислимо, в отличие, например, от логических систем, в которых процедура логического вывода может не закончить свою работу [78]. На практике важно не только определить, что два артефакта оказались несогласованными (например,

диаграмма состояний и переходов не соответствуют конечному автомату на языке C), но и выявить детали этой несогласованности: например, на диаграмме есть состояние , которого нет в программном коде. Эта так называемая *дельта* может быть сложно устроена. В нашем примере это может быть набор конструкций с той и другой стороны, которые не идентифицировались. Выявить эту *дельту* — несложная задача; гораздо труднее разобраться, например, какие из этих конструкций получились путем переименования прежних, а какие добавлены заново, чтобы избежать потери информации и обеспечить более точное соответствие.

Поэтому оказываются необходимыми действия по восстановлению согласованности, которые по описанию обнаруженной несогласованности (элементы артефактов, нарушенное правило и т. п.) приводят несогласованные артефакты в согласованное состояние. Это может быть внесение изменений из одного артефакта в другой, слияние артефактов, перегенерация одного артефакта по другому и т. д. В случае неоднозначности или наличия неустранимого противоречия может запрашиваться дополнительная информация у пользователя.

Предлагаемые в литературе подходы различаются по своему способу обращения с несогласованностью [75]. Одни из них стремятся к поддержанию постоянной согласованности артефактов, не допуская появления несогласованности [46][3]. В таких подходах реализуется управление согласованностью. Другие подходы допускают наличие несогласованности между артефактами (к этой категории относится большинство работ по устранению несогласованности UML-моделей между собой ввиду неоднозначности семантики UML). В подходах к управлению несогласованностью обнаружение несогласованности и ее устранение могут быть разделены во времени. Некоторые несогласованности могут быть оставлены, так как они либо некритичны, либо легко могут быть устранены в любой подходящий момент. В целом управление согласованностью/несогласованностью включает в себя набор типов артефактов, правил согласованности между ними, алгоритмов и политик обнаружения несогласованности и восстановления согласованности.

[3] http://www.objecteering.com/objecteering6.php.

3. Описание подходов к управлению согласованностью

Мы предлагаем классифицировать подходы к управлению согласованностью UML-моделей для следующих типов артефактов следующим образом (рис. 1):

- согласование UML-моделей между собой;
- согласование UML-моделей и требований;
- согласование UML-моделей и программного кода.

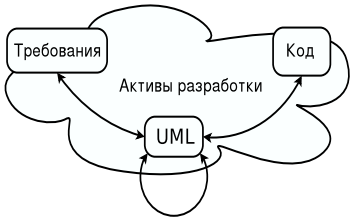

Рис. 1. Согласованность UML с другими артефактами

Мы особо выделили требования и программный код, поскольку для их спецификаций используются хорошо определенные формализмы, отличные от UML: например, для требований используются такие формализмы, как RAISE [70], DOORS [25], FOREST [5]; для программного кода — языки программирования. Несмотря на то, что есть подходы, выражающие требования на UML, а также подходы, предлагающие полную генерацию конечного кода по UML-моделям, на практике UML не оказывается вездесущим: модели требований сохраняют близость к текстам на естественных языках, а программный код невыразим в терминах UML-диаграмм. В подавляющем большинстве случаев UML используются как средство спецификации моделей анализа и проектирования. И требования, и модели анализа и проектирования, и программный код существуют в течение всего жизненного цикла разработки ПО, а несогласованность этих артефактов зачастую приносит большие неприятности.

3.1. Управление согласованностью UML-моделей

Существует несколько различных аспектов согласованности для произвольной пары информационных артефактов, которые можно отслеживать на практике. Для этого предлагаются различные методы, которые, в идеале, следует интегрировать в рамках единой среды разработки. Отметим, что для конкретной пары артефактов не всегда осмысленны все эти аспекты.

3.1.1. Аспекты согласованности

Следуя [26], выделим следующие аспекты, которые потом используем для классификации методов управления согласованностью:

1. *Синтаксический/семантический аспект*: синтаксическая согласованность — это соответствие моделей синтаксису языка моделирования, а семантическая — соответствие семантике.

2. *Статический/динамический аспект*: в первом случае согласованность проверяется путем статического анализа модели, во втором — путем ее трансформации или выполнения на некотором вычислителе.

3. *Горизонтальная/вертикальная согласованность*: в первом случае разные диаграммы или элементы модели представляют одну и ту же версию модели, иными словами, созданы в рамках одной итерации моделирования; во втором случае модели связанны некоторым преобразованием, т. е. одна получается из другой. В последнем случае возможны варианты: либо одна модель является дальнейшим развитием, уточнением или модификацией другой (это могут быть, в том числе, модели разных типов), либо преобразование всего лишь создает другое представление той же самой информации, при этом речь идет о создании другой модели.

4. *Наличие ошибок*: этот аспект позволяет диагностировать наличие несогласованности. То есть некоторая автоматизированная процедура, выполняемая над UML-спецификациями, выдает ошибку; ошибки бывают разных видов: неполнота, противоречие и т. д. Последнее подразумевает наличие логического противоречия между элементами атрефактов. Неполнота означает отсутствие в одном артефакте информации, которая присутствует в другом артефакте или которая должна содержаться в этом артефакте в соответствии с заданными правилами.

5. *Многоуровневость*: представленные выше виды правил согласованности могут использоваться (задаваться) для разных информационных «слоев» программного проекта:

   - *в предметной области* могут появляться правила согласованности, связанные с адекватным отображением в UML-моделях объектов реального мира, на работу с которыми нацелено разрабатываемое ПО;
   - *платформа реализации* может задавать отдельные правила согласованности: ведь, например, некоторые средства языка UML могут не находить отражения в конкретном языке программирования или платформе, или использоваться специфичным образом; в частности, в языке Java отсутствует множественное наследование классов, которое есть в диаграммах классов UML;
   - *архитектурные особенности проекта* могут определять специальный язык с уточненной по отношению к исходному UML семантикой конструкций; особенности процесса разработки ПО данной компании могут также определять стиль использования UML;
   - *парадигматический уровень* самого языка моделирования, основанный на его синтаксисе и семантике, задает определенные правила согласованности различных видов диаграмм и конструкций языка, а также оставляет определенную свободу для уточнений, которые могут появиться в рамках определенного проекта или предметной области.

3.1.2. Подходы

Перейдем теперь к описанию подходов управления согласованностью UML-моделей.

Синтаксическая и семантическая согласованность. Поскольку синтаксис — это наиболее формальная часть языка, то

управление такой согласованностью не является сложной задачей. Синтаксические правила могут определяться структурой метамодели UML и реализовываться на уровне структуры репозитория инструмента моделирования. Однако кое-что из синтаксиса определяется OCL-правилами и может быть реализовано различными способами: на уровне ограничений схемы репозитория (ограничения схемы базы данных), средства контроля информации при вводе (то есть проверки в пользовательском интерфейсе инструмента). Все эти способы могут не допускать создания синтаксически некорректных моделей (как на уровне отдельных конструкций, так и на уровне диаграмм и всего UML-проекта). Однако нужно предоставлять пользователям UML возможность создавать незавершенные спецификации. Незавершенность нельзя рассматривать как ошибку: проектировщик может сегодня еще не решить, с каким состоянием будет связано данное в диаграмме состояний и переходов, попробовать закрыть диаграмму и уйти с работы (наступил конец рабочего дня), а ему не дают возможность сохранить некорректную диаграмму. Но забытая незавершенность уже является ошибкой. Для этих и подобных случаев может использоваться пакетная валидация, интегрированная, например, с техникой конфигурационного управления под названием continuous integration, как это представлено в [7].

Семантика UML-моделей почти никак не формализована (впрочем трудности здесь общие для всех формальных языковых средств, используемых в программировании). Поэтому управление этим видом согласованности труднее осуществить, чем управление синтаксической согласованностью. Здесь могут использоваться средства верификации [40], также симуляция и отладка исполняемых UML-моделей [55].

Статическая и динамическая согласованность. Статическая семантическая согласованность переходит в синтаксическую при наличии соответствующих ограничений на языке OCL [26, 76]. В ряде работ делаются попытки «улучшить» метамодель UML, включив в нее статические правила согласованности, выраженные в текущем стандарте неформально [20, 38, 51, 74]. Для проверки динамической согласованности применяется преобразование одних моделей в другие (например, диаграмм последовательностей в диаграммы конечного автомата и обратно [53, 79, 80, 81]), а также пре-

образование UML-моделей в другие формализмы, обладающие исполняемой семантикой (например, диаграмм конечных автоматов в язык CSP [28, 29, 30, 31, 32, 33, 56]). В последнем случае возникает проблема обратного преобразования полученных результатов на уровень исходной UML-модели для отображения выявленных несоответствий [56, 76]. Важно отметить, что даже преобразовав исходную UML-модель в другой формальный язык и проверив согласованность получившейся спецификации средствами выбранного формализма, мы не можем гарантировать согласованность полученной спецификации и окончательной реализации системы, так как, возможно, формальная спецификация и программный код поразному интерпретируют UML-модель и доказать единство интерпретации крайне затруднительно [76].

Горизонтальная и вертикальная согласованность. Большинство работ посвящено горизонтальной несогласованности между разными диаграммами одного типа в рамках одной UML-модели. Проблема вертикальной согласованности затрагивается в [74], где представлен подход к поддержанию согласованности при уточнении отношений на диаграмме классов, и в [45], где вводится понятие структурного и поведенческого уточнения диаграмм взаимодействия и предлагается метод поддержания их согласованности.

Наличие ошибок. Как указывалось выше, наличие противоречия (другими словами, ошибки) часто принимается за определение несогласованности [11, 36, 37, 58, 76, 77, 78]. Это основной тип несогласованности, с которым считаются все подходы. Различие заключается в способе работы с противоречием. Это может быть сообщение об ошибке, запуск автоматизированного процесса синхронизации, регистрация противоречия и сохранение его в модели до тех пор, пока не будет выбрана одна из альтернатив или противоречие не станет критичным [26, 76, 82]. В [7] предлагается техника конфигурационного управления continuous integration для нахождения ошибок в спецификациях.

Другим типом ошибки, который часто рассматривается отдельно, является неполнота. Неполнота на уровне диаграмм разрешается достаточно просто при наличии репозитория для хранения UML-моделей [76]. Выявление и устранение неполноты, связанной с семантикой моделей, может потребовать достаточно сложных мето-

дов (преобразование моделей, применение других формальных нотаций, имеющих исполняемую семантику), многие из которых рассматриваются в контексте динамической согласованности. Неполнота, связанная с вертикальной согласованностью, может быть частично предотвращена в результате следования определенному процессу моделирования [41, 57] и соблюдения правил уточнения моделей [45, 74]. Относительно неполноты следует отметить, что некоторое ее наличие является естественным, так как разные модели представляют систему с определенной точки зрения, опуская не относящиеся к этой точке зрения детали. Принудительное устранение неполноты может привести к созданию перегруженных артефактов, в которых отсутствует четкий фокус модели.

Уровневая согласованность. В простейшем случае — на уровне синтаксиса языка моделирования — управление согласованностью на парадигмальном уровне не представляет трудностей. Однако другие виды согласованности здесь могут представлять значительные трудности для проверки. Любое достаточно мощное средство моделирования, как правило, умеет определять, соответствует ли UML-модель его метамодели [26, 76]. Что касается других уровней, то тут используется механизм UML-профайлов: в профайле задаются ограничения и уточнения [8, 76]. Поскольку не все ограничения и уточнения удается задать синтаксически (с помощью метамодели), для этих целей можно использовать также язык OCL, как показано в [7]. Существуют подходы, нацеленные на определение правил согласованности UML-моделей на уровне процесса [41, 57]. Эти правила определяются неформально в соответствии с определенным процессом разработки (COMET — Concurrent Object Modeling and Architectural Design Method [106], USDP — Unified Software Development Process [9, 49]) и задают последовательность создания диаграмм, а также те элементы диаграмм, на согласованность которых следует обратить внимание.

3.2. Согласованность требований с UML-моделями

Эта проблема — частный случай задачи трассировки требований (requirement traceability), которая, согласно [42], определяется как наличие возможности описать требования и отследить все их связи в прямом и обратном направлении в процессе разработки: как

к артефактам, реализующим требования (спецификациям проектирования, коду и тестам и пр.), так и от этих артефактов назад, с проверкой того, правильно ли они реализуют требования.

Многочисленные усилия тратятся на то, чтобы сблизить парадигмы требований, проектирования и кода. Хочется, чтобы структурно каждый из этих слоев был изоморфен другому. Для этого нужно, чтобы их разработка происходила в рамках единой парадигмы, которая ограничивала бы свободу (и, возможно, эффективность) на каждом из уровней, но зато давала возможность для трассировки, тем самым гарантируя их согласованность. Именно это соображение подтолкнуло методологов в конце 80-х годов прошлого века, во время распространения объектно-ориентированного программирования, начать разрабатывать объектно-ориентированные методы анализа и дизайна, признав, что господствовавшие до этого методы структурного анализа и проектирования в новой ситуации не эффективны [16]. Однако объектно-ориентированная парадигма, поднявшись до уровня анализа и проектирования, не смогла «захватить» требования.

В дополнение к объектно-ориентированным подходам для решения этой задачи оказывается эффективным применение аспектно-ориентированного программирования [59], intentional programming[4], subject-oriented programming [43] и некоторых других подхоов. При этом для построения моделей проектирования может с успехом использоваться UML. Однако пока эти подходы не нашли широкого практического применения в силу больших ограничений, которые они накладывают на процесс разработки.

Еще одной парадигмой, в рамках которой согласованность моделей требований и проектирования решается положительно, являются формальные методы и языки (например, подход RAISE [69]), используемые как для спецификации требований, так и при проектировании. Несмотря на то, что при этом предлагается использовать различные формальные языки, их принципиальная общность позволяет формулировать проблему согласованности как математическую задачу (см., например, работу [17]). Практическое применение этих подходов затруднено в связи с большой трудоемкостью использования формальных методов в реальных проектах.

На практике согласование UML-моделей и моделей требований осуществляется следующим образом. Требования раз-

[4] www.research.microsoft.com/ip.

рабатывают в специальных распространенных средствах: IBM RequisitePro, Borland CaliberRM, DOORS, а UML-модели — в пакетах типа Together ControlCenter, IBM Rational Rose. При этом для представления некоторых аспектов требований активно используются диаграммы случаев использования UML (use case diagrams). Именно для этих диаграмм существуют «мосты» между средствами UML-моделирования и инструментами разработки требований. Трассировка, как правило, поддерживается в обе стороны и существует в виде возможности генерировать требования по диаграммам случаев использования и наоборот, а также для удобной навигации по требованиям и там и там с быстрым и удобным переключением между моделями. Трассировка в смысле изменения требований средствами одного инструмента с внесением изменений в другую модель (без перегенерации!) реализуется лишь частично из-за того, что модели в обоих средствах значительно отличаются. Более детально с этим подходом можно ознакомиться в [6].

3.3. Управление согласованностью визуальных моделей и программного кода

Можно выделить три основные парадигмы использования визуальных моделей в разработке ПО, которые задают различные способы управления согласованностью между моделями и программным кодом:

- прямая разработка (forward engineering; еще один термин, практически, эквивалентный — модельно-ориентированная разработка (model driven engineering, MDE));
- возвратная инженерия (reverse engineering);
- циклическая разработка (round-trip engineering, далее — RTE).

Рассмотрим подробнее каждую из них.

3.3.1. Прямая разработка (forward engineering)

Согласно [48], разработка ПО является процессом последовательного создания уточняющих друг друга моделей. Программный код является последней, самой детальной моделью, которая может быть исполнена на вычислительном устройстве [60]. Для создания

различных моделей ПО используется UML и другие языки визуального моделирования. Существует потребность максимально формализовать процедуры уточнения моделей для того, чтобы избегать ошибок и неточностей при их преобразованиях, а также с тем, чтобы автоматизировать эти переходы. В том числе, хочется автоматизировать создание программного кода по моделям проектирования. Например, подход MDA [67] нацелен на формализацию преобразования моделей на основе теории трансформаций [23].

Однако в последнем случае есть существенное препятствие — семантический разрыв между моделями проектирования и программным кодом, на что указывалось во многих работах, например в [72]. Поэтому полные генерационные решения в общем случае невозможны, их удается реализовать лишь для отдельных предметных областей:

- баз данных[5] (используются различные варианты диаграмм сущность-связь с последующей автоматической генерацией схемы базы данных на SQL/DDL);
- структур классов объектно-ориентированного приложения[6];
- поведения систем реального времени, представленного с помощью диаграмм состояний и переходов (для телекоммуникационных систем [39][7]);
- при проектировании бизнес-логики аппаратуры на языках типа VHDL[8];
- в разработке бизнес-процессов [66];
- при разработке семейств программных продуктов (product lines) [22][9].

Связи UML-моделей с кодом могут быть следующими:

[5] http://www3.ca.com/solutions/Product.aspx?ID=260.

[6] http://www.borland.com/together/controlcenter/index.html, http://www-306.ibm.com/software/awdtools/developer/rosexde.

[7] http://www.telelogic.com/products/tau/index.cfm, http://www-306.ibm.com/software/awdtools/developer/technical.

[8] http://www.aldec.com, http://www.gmvhdl.com/VHDLStudio.html.

[9] Здесь речь идет только о подходах, активно используемых в индустрии. Область охвата академических разработок по этой теме значительно шире.

- *неформальные*: в этом случае нет автоматических процедур поддержки согласованности;
- *генерация прототипов*: исходная версия кода генерируется по моделям, а дальше дорабатывается «руками»; связь с UML-моделями не поддерживается;
- *полная генерация кода* по моделям с вынесением всей разработки на модельный уровень; при этом код может незначительно меняться помимо моделей, но он вносится в UML-репозиторий в виде новых шаблонов генерации [75][10];
- *циклическая разработка* (об этом — см. ниже).

3.3.2. Возвратная инженерия

Согласно [19], возвратная инженерия является частью процесса сопровождения ПО и позволяет понять систему с тем, чтобы внести в нее соответствующие изменения. То есть речь идет о восстановлении высокоуровневой информации о системе. Например, восстанавливаются требования к системе (хороший обзор методов по этой теме можно найти в [27]), архитектура и т. д. Мы остановимся на случае, когда восстанавливается архитектура, а сама процедура восстановления является в существенной степени автоматизированной (различные когнитивные и коммуникационные методики восстановления информации о ПО, в том числе, и основанные на применении UML, можно найти в [4, 5])

В этой области существует большое количество различных подходов и промышленных средств: программные инструменты восстановления архитектуры[11], в том числе, для legacy-приложений [2], для структуры объектно-ориентированных приложений (поддерживают все основные UML-пакеты, в частности, `IBM Rational Rose`, `Borland Together ControlCenter`), поиск и восстановление различных образцов проектирования [64] и т. д. Хороший обзор методов возвратной инженерии, ориентированных на использование UML, можно найти в [52].

В этих подходах создаваемые автоматически UML-модели не являются самостоятельными артефактами. Как правило, они оказываются диаграммным представлением программного кода, легко автоматически обновляются при его изменении и не допускают внесение какой-либо информации кроме той, что была получена автоматически или относится к графическим свойствам диаграммы. Как правило, диаграммы и методы их построения оказываются средством извлечения знаний, некоторым побочным информационным артефактом (а главный артефакт при этом — понимание, возникающее у того, кто выполняет процедуру возвратной инженерии) или заготовками (прототипом) для последующего редактирования вручную.

[10] http://www.objecteering.com/objecteering6.php.
[11] http://www.laatuk.com/tools/documentation_tools.html.

3.3.3. Циклическая разработка (RTE)

В [10] RTE определяется как система для двух доменов A и B, таких, что существует функционал i, который для любой трансформации f из A в B «умеет» вычислять обратную трансформацию f^{-1}, т. е. $i \circ f = f^{-1}$. Считается, что если нам неудобно работать в домене A, мы преобразуем нашу спецификацию, используя некоторую трансформацию f, в домен B, работаем с ней там, а потом преобразуем ее обратно в домен A. Но если «работа» над спецификацией заключается в ее изменении, то f^{-1} не подходит для обратного преобразования, так как не учитывает этих наших изменений. Более того, во многих случаях нужно, по возможности, максимально сохранить исходный образ спецификации в домене A, так как он содержал информацию, которая не попадает в B. Например, мы сгенерировали код по UML-диаграмме, потом его исправили, и по нему перегенерировали диаграмму. При этом графическое расположение элементов диаграммы, тщательно выверенное проектировщиком (UML-диаграммы должны быть «красивыми», т. е. удобными для понимания), заменилось на стандартную графическую раскладку.

В [10], а также в [44, 60, 75] RTE рассматривается как совокупность методов прямой разработки и возратной инженерии. В [73] это ограничение снимается, и RTE понимается как возможность средств разработки ПО автоматически поддерживать целостность различных изменяемых артефактов разработки. То есть эти артефакты могут изменяться естественным путем, когда это требуется в процессе разработки, и небходимые изменения будут автоматически проведены во все другие, связанные артефакты. Существенно, что по сравнению с [10] вместо доменов здесь фигурируют собствен-

но сами артефакты (спецификации, модели и т. д.) и появляется возможность учитывать их частичные изменения, а не довольствоваться только полной перегенерацией.

На практике RTE оказывается востребованным для двух специальных типов артефактов разработки: визуальных моделей и программного кода. Многие исследователи, минуя излишнюю степень общности, определяют RTE именно для этих артефактов [50, 52, 60, 75]. Самая известная промышленная реализация этого подхода — пакет `Together Borland Control Center`[12], реализующий RTE для диаграмм классов UML и Java-кода. Имеются и другие реализации RTE для визуальных моделей и кода: пакет `Fujaba` [50, 62, 63], университетский пакет `DQ` [60], `RationalRose XDE`[13] и многие другие.

Отличительной чертой RTE является равноценность кода и визуальных моделей [55, 60, 73][14]. Точное определение связей между кодом и визуальными моделями, а также стратегия поддержания согласованности двух этих артефактов сильно варьируется в зависимости от потребностей предметной области: где-то в самом деле хватает совокупности прямой разработки и возвратной инженерии, как, например, в подходе, представленном в [60], где-то принципиально сохранение изменений в обоих артефактах и их синхронизация, как, например, в [55]. При этом могут рассматриваться многочисленные специальные ограничения на изменения кода, моделей, процесс перегенерации кода по моделям и сохранения «ручных» изменений. Один такой подход представлен в работах [1, 3, 47].

Заключение

Нужно честно отметить, что, несмотря на обилие подходов к управлению согласованностью, их практический эффект невелик. С одной стороны, математизированные подходы сложны в использовании и имеют высокую вычислительную сложность при решении реальных задач. С другой стороны, не удается найти элегантных концепций в этой области, которые могли бы послужить хорошей концептуальной основой для проектирования промышленных

[12] http://www.borland.com/together/controlcenter/index.html.

[13] http://www-306.ibm.com/software/awdtools/developer/rosexde.

[14] http://www.borland.com/together/controlcenter/index.html.

инструментов[15]. Существующие коммерческие решения в этой области создаются *ad hoc*.

Нам кажется, что управление согласованностью нужно рассматривать в контексте функций процесса разработки, соблюдая информационно-функциональный баланс. Только на уровне информационных артефактов данная задача не может быть решена удовлетворительно, что осознается, например, авторами работы [73]. Эта задача не существует в «чистом виде»: она смешана с более общими задачами программной инженерии.

Еще одним направлением развития этой темы является повышение формальности процесса разработки ПО в целом. Какая-то из частей конкретного процесса создания ПО (в частности, задача управления согласованностью каких-либо артефактов) не может иметь меру формализованности и автоматизации существенно большую, чем значение этой меры для всего процесса в целом. Одним из подходов, решающим задачу более строгой формализации процесса разработки и существенно повышающим уровень его автоматизации, является организация выпуска семейств программных продуктов [65]. Создание и реализация особых стратегий управления согласованностью UML-моделей в этом контексте представляется перспективной.

Список литературы

[1] *Артамонов Н.* Итеративная разработка в технологии REAL-IT/Web: сохранение изменений пользовательского интерфейса. Дипломная работа. СПбГУ. 2006. 58 с.

[2] *Бабурин Д. Е., Бульонков М. А., Емельянов П. Г., Филаткина Н. Н.* Средства визуализации при перепроектировании программ // Программирование. № 2. 2001. С. 21–34.

[3] *Иванов А. Н.* Механизмы поддержки циклической разработки ИС в рамках модельно-ориентированного подхода // Системное программирование. СПб.: 2004. С. 101–123.

[4] *Кознов Д. В., Перегудов А. Ф., Романовский К. Ю.* и др. Опыт использования UML при создании технической документации // Системное программирование. Вып. 1. СПб.: Изд. СПбГУ, 2005. С. 18–35.

[15] Например, теория синтаксического анализа является хорошим фундаментом для практической разработки промышленных компиляторов.

[5] *Кулямин В. В., Пакулин Н. В., Петренко О. Л.* и др. Формализация требований на практике. Препринт ИСИ РАН. 2006. 50 с.

[6] *Фесенко Т. А.* Реализация системы интеграции средства моделирования Borland Together Designer и средства управления требованиями Borland CaliberRM. Дипломная работа. СПбГУ. 2005. 49 с.

[7] *Ольхович Л. Б., Кознов Д. В.* Метод автоматической валидации UML-спецификаций на основе OCL // Программирование. №6. 2003. С. 44–50.

[8] *Agrawal A., Karsai G., Shi F.* Graph Transformations on Domain-Specific Models // Technical Report ISIS-03-403. Institute for Software Integrated Systems. Vanderbilt University. Nashville, TN 37203. 2003. 43 p.

[9] *Arlow J., Neustadt I.* UML and the unified process: practical object-oriented analysis and design. Addison-Wesley, 2002. 528 p.

[10] *Assmann U.* Automatic Roundtrip Engineering // Workshop on Software Composition, ENTCS. Vol. 82. No. 5. 2003. P. 1–9.

[11] *Astesiano E., Reggio G.* An Algebraic Proposal for Handling UML Consistency // Consistency Problems in UML-based Software Development II. 2003. P. 62–70.

[12] *Boiten E., Bowman H., Derrick J., Steen M.* Viewpoint consistency in Z and Lotos: A case study // Proc. 4th Symposium of Formal Methods Europe, LNCS 1313. 1997. P. 644–664.

[13] *Boiten E., Derrick J., Bowman H., Steen M.* Consistency and refinement for partial specification in Z // FME'96: Industrial Benefit of Formal Methods, 3rd International Symp. of Formal Methods, LNCS 1051. 1996. P. 287–306.

[14] *Bowman H., Boiten E., Derrick J., Steen M.* Viewpoint consistency in ODP, a general interpretation // Proc. First IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems. 1996. P. 189–204.

[15] *Bowman H., Derrick J., Steen M.* Some Results on Cross Viewpoint Consistency Checking // IFIP TC6 International Conference on Open Distributed Processing. 1995. P. 399–412.

[16] *Champeaux D., Constantine M. L., Jacobson I.* e.a. PANEL: Structured Analysis and Object Oriented Analysis // Proc. ECOOP/OOPSLA. 1990. P. 135–139.

[17] *Chechik M., Gannon J.* Automatic Analysis of Consistency Between Requirements and Designs // IEEE Transactions in Software Engineering. Vol. 27. №7. 2001.

[18] *Cheng B. H. C., Wang E. Y., Richter H.* Formalizing and Integrating the Dynamic Model within OMT // Proc. IEEE International Conference on Software Engineering. 1997. P. 45–55.

[19] *Chikofsky E. L., Cross J. H.* Reverse Engineering and Design Recovery: Taxonomy // IEEE Software. January 1990. P. 13–17.

[20] *Clark T., Evans A., Kent S.* e.a. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach // www.puml.org. September 2000.

[21] *Cook C.* Collaborative Software Engineering: An Annotated Bibliography // Technical Report TR-02/04, Software Engineering & Visualization Group, Department of Computer Science and Software Engineering, Christchurch, New Zealand. 2005.

[22] *Czarnecki K., Eisenecker U. W.* Generative programming: Methods, Tools, and Applications. Addison-Wesley, 2000. 832 p.

[23] *Czarnecki K., Helsen S.* Classification of Model Transformation Approaches // Proc. 2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture. 2003.

[24] *Derrick J., Bowman H., Steen M.* Maintaining cross viewpoint consistency using Z // IFIP TC6 International Conference on Open Distributed Processing. 1995. P. 413–424.

[25] DOORS Reference Manual. QSS, Oxford Science Park, Oxford OX4 4GA. 1993–1998.

[26] *Elaasar M., Briand L.* An Overview of UML Consistency Management // Technical Report SCE-04-18. Department of Systems and Computer Engineering. 2004.

[27] *El-Ramly M., Stroulia E., Sorenson P.* Recovering Software Requirements from System-user Interaction Traces // Proc. 14th Int. Conf. on Software Eng. and Knowledge Eng. (SEKE'02). 2002.

[28] *Engels G., Hausmann J., Heckel R., Sauer S.* Testing the consistency of dynamic UML diagrams // Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002). 2002.

[29] *Engels G., Heckel R., Kuster J. M.* Rule-based specification of behavioral consistency based on the UML metamodel // Proc. Int'l Conf. UML 2001. The Unified Modeling Language. Modeling Languages, Concepts, and Tools, LNCS 2185. 2001. P. 272–286.

[30] *Engels G., Heckel R., Kuster J. M., Groenewegen L.* Consistency-preserving model evolution through transformations // Proc. Int'l Conf. UML 2002. The Unified Modeling Language. Model Engineering, Concepts, and Tools, LNCS 2460. 2002. P. 212–227.

[31] *Engels G., Kuster J. M., Groenewegen L., Heckel R.* A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models // Proc. 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9). 2001. P. 186–195.

[32] *Engels G., Kuestler J. M., Groenewegen L.* Consistent Interaction of Software Components // Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002). 2002.

[33] *Engels G., Kuster J. M., Heckel R., Groenewegen L.* Towards Consistency-Preserving Model Evolution // Proc. ICSE Workshop on Model Evolution, LNCS 2300. 2002. P. 257–276.

[34] *Feng T. H., Vangheluwe H.* Case Study: Consistency Problems in a UML Model of a Chat Room // Consistency Problems in UML-based Software Development II. 2003. P. 18–25.

[35] *France R., Evans A., Lano K., Rumpe B.* Developing the UML as a formal modeling notation // Computer Standards and Interfaces: Special Issues on Formal Development Techniques. 1998.

[36] *Gabbay D., Hunter A.* Making inconsistency respectable: Part 1 — A Logical framework for inconsistency in reasoning // Proc. Fundamentals of Artificial Intelligence Research. 1991.

[37] *Gabbay D., Hunter A.* Making Inconsistency Respectable: Part 2 — Metalevel handling of inconsistency // Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU'93), LNCS 747. 1993. P. 129–136.

[38] *Genova G, Llorens J., Fuentes J. M.* The Baseless Links Problem // Consistency Problems in UML-based Software Development II. 2003. P. 58–61.

[39] *Gery E., Harel D., Palachi E.* Rhapsody: A Complete Life-Cycle Model-Based Development System // Proceedings of IFM 2002 — Third International Conference on Integrated Formal Methods, LNCS 2335. 2002. P. 1–10.

[40] *Giese H., Tichy M., Burmester S.* e.a. Towards the compositional verification of real-time UML designs // Proc. European Software Engineering Conference (ESEC). 2003. P. 38–47.

[41] *Gomaa H., Wijesekera D.* Consistency in Multiple-View UML Models: A Case Study //Consistency Problems in UML-based Software Development II. 2003. P. 1–8.

[42] *Gotel O., Finkelstein A.* An Analysis of the Requirements Traceability Problem // Proc. First International Conference on Requirements Engineering. 1994. P. 94–101.

[43] *Harrison W., Ossher H.* Subject-Oriented Programming (a critique of pure objects) // Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA). 1993. P. 411–428.

[44] *Henriksson A., Larsson H.* A Definition of Round-trip Engineering // Technical report, Linkopings Universitet, Department of Computer and Information Science. 2003.

[45] *Hnatkowska B., Huzar Z., Kuzniarz L., Tuzinkiewicz L.* Refinement relationship between collaborations // Consistency Problems in UML-based Software Development II. 2003. P. 51–57.

[46] *Homrighausen A., Six H.-W., Winter M.* Round-Trip Prototyping Based on Integrated Functional and User Interface Requirements Specifications // Requirements Engineering. Vol. 7. N 1. 2002. P. 34–45.

[47] *Ivanov A. N., Koznov D. V.* REAL–IT — Model-Based Interface Development Environment // IEEE ISoLA Workshop. 2005.

[48] *Jacobson I.* Object-Oriented Software Engineering. ACM Press, 1992. 528 p.

[49] *Jacobson I., Booch G., Rumbaugh J.* Unified Software Development Process. Addison-Wesley, 1998.

[50] *Klein T., Nickel U., Niere J., Ündorf A. Z.* From UML to Java And Back Again // Technical Report tr-ri-00-216, University of Paderborn, Germany. 2000.

[51] *Kleppe A., Warmer J.* Unification of Static and Dynamic Semantics of UML. A Study in redefining the Semantics of the UML using the pUML OO Meta Modelling Approach // www.klasse.nl/english/uml/semantics.html. 2001.

[52] *Kollmann R., Selonen P., Stroulia E.* e.a. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering // Proc. 9th Working Conference on Reverse Engineering. 2002.

[53] *Koskimies K., Systä T., Tuomi J., Männistö T.* Automated Support for Modeling OO Software // IEEE Software. January 1998. P. 87–94.

[54] *Koznov D.* Visual Modeling and Software Project Management // Proceedings of 2nd International Workshop «New Models of Business: Managerial Aspects and Enabling Technology». 2002. P. 161–169.

[55] *Koznov D., Kartachev M., Zvereva V.* e.a. Roundtrip engineering of reactive systems // Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA). 2004. P. 343–346.

[56] *Kuster J. M.* Towards Inconsistency Handling of Object-Oriented Behavioral Models // Electronic Notes in Theoretical Computer Science. Vol. 109. 2004. P. 57–69.

[57] *Kuzniarz L., Staron M.* Inconsistencies in Student Designs. Consistency in Multiple-View UML Models: A Case Study // Consistency Problems in UML-based Software Development II. 2003.

[58] *Lange C., Chaudron M. R. V., Muskens J.* e.a. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs // Consistency Problems in UML-based Software Development II. 2003. P. 26–34.

[59] Proc. ICSE'98 Workshop on Aspect-Oriented Programming. 1998.

[60] *Maciaszek L. A.* Roundtrip Architectural Modeling // 2nd Asia-Pacific Conference on Conceptual Modelling (APCCM). 2005 P. 17–23.

[61] *McUmber W. E., Cheng B.* A General Framework for Formalizing UML with Formal languages // Proc. 23rd International Conference on Software Engineering. 2001. P. 433–442.

[62] *Nickel U., Niere J., Wadsack J., Zündorf A.* Roundtrip Engineering with FUJABA // Proc. 2nd Workshop on Software-Reengineering (WSR). 2000.

[63] *Nickel U., Niere J., Zündorf A.* The Fujaba Environment // Proc. 22nd International Conference on Software Engineering. 2000. P. 742–745.

[64] *Niere J., Schafer W., Wadsack J. P., Wendehals L.* Towards Pattern-Based Design Recovery // Proceedings of the 22nd International Conference on Software Engineering (ICSE). Limerick, Ireland: ACM Press, June 2000. P. 241–251.

[65] *Northrop L. M.* A Framework for Software Product Line Practice Version 4.2. // http://www.sei.cmu.edu/productlines/framework.html. 2006.

[66] Business Process Modeling Notation. Final Notation Specification dtc/06-02-01. Object Management Group (OMG). 2006.

[67] MDA Guide Version 1.0.1 Object Management Group (OMG). 2003. // http://www.omg.org/mda.

[68] UML 2.0 Infrastructure Specification Object Management Group (OMG). 2004. // http://www.omg.org.

[69] The RAISE Development Method. The RAISE Method Group. 1995.

[70] *Roddik J. F.* A Survey of Schema Versioning Issues for Database Systems // Information and Software Technology. N 37(7). 1995. P. 383–393.

[71] *Sannella D.* The Common Framework Initiative for algebraic specification and development of software // Proc. 3rd International Conference on Perspectives of System Informatics (PSI'99), LNCS 1755. 2000. P. 1–9.

[72] *Selic B., Gullekson G., Ward P. T.* Real-Time Object-Oriented Modeling. John Wiley & Sons. Inc, 1994. 525 p.

[73] *Sendall S., Kuster J.* Taming Model Round-Trip Engineering // Proc. Workshop on Best Practices for Model-Driven software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, Applications). 2004.

[74] *Shen W., Lu Y., Low W. L.* Extending the UML Metamodel to Support Software Refinement // Consistency Problems in UML-based Software Development II. 2003. P. 35–42.

[75] Guarantee permanent Model/Code consistency: «Model driven Engineering» versus «Roundtrip engineering» // Softeam White paper. 2000.

[76] *Sourrouille J. L., Caplat G.* A Pragmatic View on Consistency Checking of UML Models // Consistency Problems in UML-based Software Development II. 2003. P. 43–50.

[77] *Spanoudakis G., Finkelstein A., Till D.* Overlaps in Requirements Engineering // Automated Software Engineering Journal. Vol. 6. 1999. P. 171–198.

[78] *Spanoudakis G., Zisman A.* Inconsistency Management in Software Engineering: Survey and Open Research Issues // Handbook of Software Eng. and Knowledge Eng. Vol. 1. 2001. P. 24–29.

[79] *Systä T.* Dynamic Modeling in Forward and Reverse Engineering of OO Software Systems // Proc. Doctoral Symposium of 13th IEEE International Conference of Automated Software Engineering (ASE98). 1998. P. 47–50.

[80] *Systä T.* Dynamic reverse engineering of Java software // Proc. ECOOP Workshop on Experiences in Object-Oriented Re-Engineering. 1999.

[81] *Tsiolakis A.* Integrating Model Information in UML Sequence Diagrams // Electronic Notes in Theoretical Computer Science. Vol. 50. N 3. 2004. P. 1–9.

[82] *Wagner R., Giese H., Nickel U. A.* A Plug-In for Flexible and Incremental Consistency Management // Consistency Problems in UML-based Software Development II. 2003. P. 78–85.