УДК: 004.4’22

СЕМАНТИЧЕСКИЕ МОДЕЛИ ПРОГРАММ В ИНТЕЛЛЕКТУАЛЬНЫХ СРЕДАХ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Г. Ф. ДЮБКО, Е. Л. ЛЕЩИНСКАЯ, В. М. ЧЕРЕПАХИН

Методология MDD призвана перевести процесс разработки ПО на качественно новый уровень. Но до сих пор не разработаны необходимые модели и методы, способные удовлетворить все необходимые для MDD процессы. Предложенные формальные семантические модели программ – первый шаг на пути внедрения MDD. Они позволяют организовать разработку ПО как процесс поэтапного уточнения его моделей от описания бизнес-компонент и их взаимодействия до программных компонент. Семантическое аннотирование позволяет задействовать мощную базу знаний прежде разработанных решений и условий их применения.

MDD methodology was introduced to transfer the software development process to a new high-quality level. But still there is no holistic approach to realize it in practice, not enough models and methods are developed to satisfy all MDD processes. The proposed formal models of the software are the first step to put MDD in practice. They allow to arrange the software development as a sequential process of its model refinement starting with the business components and their interaction description up to the software components. The semantic annotation allows to utilize a wide knowledge base of previously made decisions and conditions for their application.

1. ЭВОЛЮЦИОННЫЕ ПЕРСПЕКТИВЫ ТЕХНОЛОГИЧЕСКИХ ПРОЦЕССОВ В ИНДУСТРИИ ПО

Современный этап развития IT-индустрии характеризуется появлением большого количества новых архитектур и технологий, сред программирования, программных платформ. При этом требования к срокам разработки и бюджетам создаваемых проектов жестко ограничиваются потребностями и возможностями заказчиков, а также желанием повышения конкурентоспособности IT-компании разработчика на рынке аналогичных услуг. Для сохранения высокого качества разрабатываемых продуктов и выживания в условиях всевозрастающей конкуренции IT-компаниям необходима модернизация способа проектирования, разработки, сопровождения и реинжиниринга программного обеспечения (ПО). Важными этапами в этом направлении стали: 1) разработка языков визуального проектирования, наиболее популярным из которых является Unified Modeling Language (UML); 2) создание множества различных утилит, формализующих и автоматизирующих процесс проектирования узко специализированного ПО [1] (в основном в сфере разработки баз данных и программирования микроконтроллеров); 3) создание консорциумом OMG методологии разработки, управляемой моделью (Model Driven Development – MDD), и лежащую в её основе архитектуру, управляемую моделью (Model Driven Architecture – MDA) [2], что должно перевести процесс разработки ПО на качественно новый уровень; 4) индустриализация процесса разработки ПО, выразившаяся в появлении исследовательской концепции фабрик программного обеспечения [3] – согласованного набора процессов и средств (настраиваемых компонент, прототипов систем и др.), ускоряющих цикл создания ПО; 5) разработка наборов компонент Windows Workflow Foundation (MS VS.NET 2008), Enterprise Core Objects (BDS 2007), Unify NXJ, WebSphere, позволяющих создавать отдельные типы высокоуровневых моделей программных элементов и автоматически строить на их основании исполняемые модули.

Применение описанных технологий нацелено на формализацию и интеграцию процессов проектирования и реализации ПО, путем введения понятия *формализованной модели программной системы*, попытку выявления большей части ошибок путем её автоматической верификации еще на этапе проектирования будущего продукта, проведение оптимизации на уровне модели, интеллектуализацию сред проектирования, что должно привести к повышению уровня повторного использования однажды предложенных программных решений, ускорению разработки, облегчению процесса документирования и сопровождения.

Очевидно, что достигнутых результатов еще недостаточно для реализации методологии MDD: 1) необходимо провести формализацию моделей ПО, применяемых в рамках MDA, т.к. UML хотя и является технологическим стандартом, но лишь высокоуровневого средства документирования, с нечетко определенной семантикой исполнения; 2) отсутствует определенный формализованный переход от UML-проекта системы к её конкретной реализации; 3) используемый для проектирования, разработки и сопровождения ПО в течение всего его жизненного цикла пакет продуктов, как правило, является интеграцией решений различных производителей, из-за чего неизбежно возникают проблемы совместимости и синхронизации; 4) WWF и ECO охватывают только 2 вида формальных моделей программных компонент.

Для полноценной реализации методологии MDD необходимо разработать: 1) согласованную последовательность моделей для представления цепочки трансформаций

$$CIM \rightarrow PIM \rightarrow PSM \rightarrow code\ ;$$

2) методы автоматизированной генерации уточняющих моделей; 3) методы автоматического накопления и использования готовых архитектурных решений; 4) методы автоматической верификации формальных моделей.

Представляет интерес разработка собственных математических методов для создания ПО на базе моделей.

2. МЕТОДОЛОГИЯ РАЗРАБОТКИ ПРИЛОЖЕНИЙ С АРХИТЕКТУРОЙ, УПРАВЛЯЕМОЙ МОДЕЛЬЮ

Схематически возможный процесс разработки приложения в соответствии с методологией MDD приведен на рис. 1. Под каждой моделью на рис. 1 перечислено неполное множество языков, методов, технологий, на основании которых возможно выражение каждой из моделей, реализация методов её анализа и трансформации. Из них к языкам визуального моделирования относятся UML (диаграммы UseCases, Classes, Activity, StateChart), BPMN (диаграмма BPD). Формальные средства задания семантики моделей [4,5]: Object-Z, TCOZ, Event-B Method, CSP, TimedCSP, OCL. Языки трансформаций: QVT, ATL. Языки временных логик: LTL. Конечные языки реализации: C#, Java, Jass, SQL.

Преимущества использования описанного выше подхода очевидны: 1) он не имеет конкурентов среди существующих технологий создания приложений (NET, J2EE, Sun ONE), т.к. фактически находится на более высоком уровне обобщения процесса разработки; 2) применение его возможно не только при использовании известных на данный момент технологий разработки; интегрируя в себя все будущие технологии, MDD требует лишь создания нового адаптера, переводящего PIM в желаемую PSM; 3) разработанное на базе модели приложение может быть формально протестировано на соответствие требованиям, заложенным в спецификацию модели, например, пре- и пост- условиям выполнения действий [6]; 4) уменьшается стоимость разработки, сопровождения и реинжиниринга проекта.

3. СЕМАНТИЧЕСКИЕ МОДЕЛИ ПРОГРАММ

Принимая во внимание определенную степень свободы, предоставляемую спецификацией MDA разработчикам средств проектирования ПО при интерпретации определения программных моделей, был разработан законченный набор моделей, позволяющий реализовать проектирование ПО в соответствии с принципами описанной выше методологии и последующей их автоматической трансформацией в исходный код.

3.1. CIM модель

По определению OMG, модель первого этапа *не зависит от вычислений* (CIM). Следовательно, CIM модель может описывать предметную область разрабатываемой системы, как понятия двух подмножеств сущностей: 1) относящиеся к системе активные бизнес-сущности и интерфейсы их взаимодействия; 2) пассивные бизнес-сущности, которыми манипулирует система, и отношения между ними.

DAC-диаграммы. Для проектирования понятий первого подмножества введена диаграмма «*Взаимодействия агентов предметной области*» (*D*omain *A*gents *C*ommunication – *DAC*), где под агентами подразумеваются активные сущности моделируемой предметной области, каждый из которых предоставляет набор предопределенных сервисов и/или преследует свои цели.

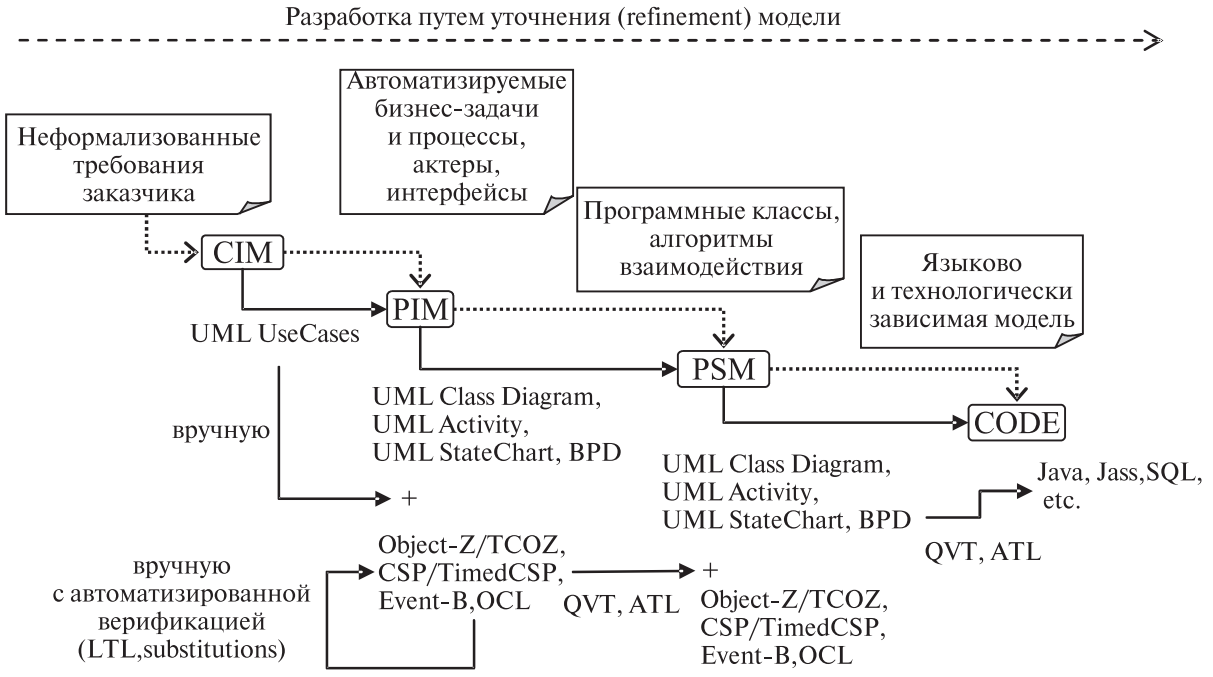


Рис. 1. Процесс разработки приложения в соответствии с архитектурой MDA

Каждый узел *DAC* – это агент, характеризующийся типом *Type*, именем *Name*, набором поддерживаемых интерфейсов взаимодействия с внешней средой *InterfaceList* (*IL*). Набор поддерживаемых интерфейсов зависит от типа агента и определяет методы взаимодействия для вызова сервисов *ServiceList* (*SL*), предоставляемых агентом потребителям.

Каждый сервис – это процесс. *DAC* определяет только имя и естественно языковое описание процесса. Все агенты *DAC* могут быть расклассифицированы как внешние к разрабатываемой ПС сущности или её компоненты. Эта характеристика отражается в *DAC* как свойство каждого агента. Каждый агент *DAC* имеет дополнительно такую характеристику, как множественность *Multiplicity*, которая показывает, сколько экземпляров агентов такого типа может существовать в момент функционирования системы. Множественность задается целым числом, строгим диапазоном из двух чисел или одним из предопределенных шаблонов: ?, *, + (0 или 1, 0 или более, 1 или более). Визуально агент *DAC* представляется предопределенным типом визуального изображения.

Связи в *DAC* – это бинарные отношения одного из двух типов: 1) взаимодействия; 2) наследования.

Отношение *взаимодействия* (Communication Relation, CR) – это некоммутативное (однонаправленное) отношение использования, помеченное именем интерфейса *Intf*, поддерживаемого каждым из участников отношения. Если один участник отношения, например $A1$, – внешняя к ПС сущность, а второй $A2$ – её компонент, то интерфейс выбирается из перечня доступных для $A1$; $A2$ в свою очередь должен реализовать поддержку этого интерфейса. Конечная точка отношения, соответствующая используемому агенту далее именуется «*исполнитель*» и может быть помечена дружественным именем, противоположная ей точка именуется «*потребителем*». Агент-потребитель определяется еще одной дополнительной характеристикой – множеством целей *Goals*. Каждая цель – это процесс, соответствующий одному из множества сервисов исполнителя текущего отношения, который потребитель может инициировать.

Отношение *наследования* (Inheritance Relation, IR) – это отношение расширения. Если агент $A2$ является наследником $A1$, то

а) $IL(A1) \subseteq IL(A2)$;

б) $r1 = CR(A1, A3) \wedge r2 = CR(A2, A3) \wedge$
$Intf(r1) = Intf(r2) \wedge$
$IR(A2, A1) \rightarrow Goals(A1, r1) \subseteq Goals(A2, r2)$.

BOD-диаграммы. Для проектирования понятий второго подмножества сущностей предметной области *CIM* введена «*Диаграмма бизнес-объектов предметной области*» (*B*usiness *O*bjects *D*iagram – *BOD*) типа сущность-связь. Под бизнес-объектом понимается пассивный концепт предметной области, используемый одним или более агентами *DAC* как группирующая единица при структурировании манипулируемой информации. *BOD всегда* находится в контексте некоторого внутреннего агента *DAC*. По своей сути *BOD* является упрощенной диаграммой классов. От полной диаграммы классов её отличает отсутствие ассоциативных классов, а также упрощенная модель класса: каждый класс *BOD* характеризуется набором свойств, агрегативных связей, и связей наследования. Спецификаторы доступа для свойств, ассоциативных связей и методы в классах *BOD* отсутствуют.

Обязательным элементом BOD-диаграммы является класс, представляющий внутреннего агента, в контексте которого составляется диаграмма.

GUI-модели. Одна *GUI-модель* описывает протокол взаимодействия исполнителя с пользователем (потребителем) при инициации им некоторого процесса, связанного с одной из перечисленных для потребителя на DAC-диаграмме целей. *GUI-модель* представляет собой ориентированный граф переходов без циклов с выделенной вершиной-источником. Вершина-источник инцидентна только исходящим из неё ребрам. Каждая вершина *GUI-модели* – это описание окна и его элементов управления. Ребра графа определяют возможные переключения между окнами при взаимодействии пользователя (агента-потребителя) с процессом.

Для описания интерфейса каждого окна должен быть использован платформонезависимый язык разметки наподобие XAML [7], UIML или XUL. Язык разметки интерфейса – это XML-подобный декларативный язык, который с помощью иерархической структуры вложенности задает расположение и свойства всех элементов управления окна. Манипулируя набором общепринятых базовых для визуального интерфейса элементов управления (таких как кнопка, список, выпадающий список выбора, таблица) этот язык становится средством проектирования интерфейсов независимо от результирующей платформы приложения (host-, web-, mobile и т.д.). Настройка внешнего вида элементов управления возможна благодаря принципу «навешивания» стилей. Сложные элементы управления могут быть реализованы как комбинация уже определенных – пользовательские составные элементы управления.

Вспомогательным для языка разметки интерфейса является язык запросов к элементам BOD-диаграммы (Object Query Language – OQL). Его использование позволяет осуществить привязку данных к свойствам элементов управления окна, например, надпись на кнопке, элементы выпадающего списка, высота столбика на диаграмме и т.д. Гибкость такой привязки определяется мощностью языка запросов. В основу языка запросов может быть положена нотация OCL описания ограничений объектов, предложенная OMG [8]. OQL допускает задание контекстного объекта, об-

ращение к его простым и агрегативным свойствам. Поскольку в общем случае агрегативные свойства представляют собой коллекции однотипных объектов, то в OQL определены query-методы работы с коллекциями: фильтрация по условию, обращение к элементу по индексу, подсчет кол-ва элементов, итерация с произвольным действием и т.д.

3.2. PIM-модель

По определению OMG, модель второго этапа *не зависит от платформы* (PIM). Следовательно, PIM модель должна формально определить структуру и поведение компонентов ПС так, чтобы, выбрав параметры платформы, можно было получить её окончательную реализацию. PIM может определять программные компоненты и алгоритмы их взаимодействия на различном уровне детализации. Также формальное описание элементов PIM должно позволять осуществить их автоматический анализ и детализацию. Являясь продолжением CIM, PIM-модель может уточнить BOD-диаграммы агентов, преобразовав их в полноценные диаграммы классов (без деталей, касающихся конкретного языка реализации), а также должна содержать формальное описание алгоритмов реализации всех сервисов internal-агентов DAC и их связи с реализуемыми этими агентами интерфейсами, например, пользовательским.

Диаграммы классов. Диаграмма классов (*C*lass *D*iagram – *CD*) является расширением BOD. Множество классов BOD дополняется внутренними классами ПС, а сами классы BOD дополняются методами, использование которых позволяет реализовать целевые процессы DAC.

К отношениям между ролевыми сущностями кроме стандартных отношений обобщения и агрегации добавлено отношение уточнения (Refine), которое позволяет связывать различные представления одних и тех же элементов на диаграммах разных этапов в процессе уточнения PIM (необходимо для проверки корректности процесса уточнения). Введена также вспомогательная сущность Expression, представляющая собой выражение, прикрепляемое к классам, атрибутам и процессам для их дополнительной характеристики. Например, каждая структурная сущность RoleEntity CD-диаграммы может быть снабжена агрегативным отношением predicate с множеством объектов Expression, задающих инварианты класса. Такие свойства могут быть использованы для контроля корректности реализации методов класса.

Диаграммы процессов. Диаграмма процессов (*P*rocess *D*iagram – PD) позволяет на алгоритмическом уровне описать происходящие в ПС процессы. Начиная с бизнес-процессов, перечисленных в DAC, PD расширяется уточняющими процессами, происходящими внутри ПС прозрачно для пользователя. В связи с наибольшей выразительностью в основу PD-диаграмм были положены диаграммы описания бизнес-процессов.

Ключевым элементом PD-диаграммы является процесс (сущность ProcessEntity). Контекстом для процесса считается тот класс, к которому отнесен метод, содержащий реализацию процесса. С целью контроля корректности реализации процесса для любого процесса могут быть определены логические пред- и постусловия его выполнения. Процесс может начаться только в случае, если его предусловие истинно в данном контексте. Процесс считается корректно завершенным, если его постусловие истинно в данном контексте. Пред- и постусловия могут быть пустыми, тогда они считаются истинными.

При проектировании до реализации плана исполнения процесса (сущность ExecutionPlan) для процесса может быть создано «*упрощенное семантическое описание*» (*shallow semantic description*, свойство desc сущностей ProcessEntity и BehavioralEntity). Такое описание формализует *цели* исполнения процесса и может быть использовано интеллектуальной средой проектирования для их автоматической реализации на основе шаблонных решений.

В модели концептов PD-диаграммы выделены два типа состояний: начальное и конечное. Для обеспечения детерминизма в описании процесса допускается единственное начальное состояние и множество конечных, для каждого из которых может быть определено свое постусловие. Свойство IsSucceed конечного состояния позволяет разделить все конечные состояния процесса на два логических множества: успешные и ошибочные.

Для удобства выражения ветвлений, связанных с обработкой ошибок и прочих нештатных сценариев выполнения процесса, предусмотрена специальная разновидность поведенческого элемента – *компенсационная сущность*. Выделены два типа компенсаций: по условию, по событию. Семантика поведения компенсационных элементов следующая: если к блоку-действию прикреплен один или более компенсационных элементов, то в любой момент ветвления (независимо от степени вложенности) процесса, если условие компенсации выполнено, то выполнение немедленно передается в соответствующий блок компенсации в порядке вложенности этих блоков. После выполнения компенсационных действий управление передается в точку прерывания, если свойство CanContinue компенсационного блока истинно и выполнение компенсационного блока завершено успешно (вывод делается по типу завершающего состояния, в котором остановился процесс компенсации).

Предусмотрено также несколько режимов ветвлений: в зависимости от свойства SplitType блока ветвления возможно запрещение или разрешение одновременного движения сразу по нескольким веткам после ветвления, что позволяет моделировать распараллеливание процессов.

Смысл прочих поведенческих сущностей пакета (SendMessage, Assign, ProcessCall, WhileLoop ForEachLoop и др.) однозначно следует из их названия.

Формализация «упрощенного семантического описания» поведенческих сущностей. Для создания упрощенного семантического описания некоторого процесса, реализуемого поведенческой сущностью, предлагается использовать *последовательность функционалов*.

Определение 1. Функционалом будем называть описательное отношение-шаблон *F*, характеризующееся именем отношения *RelationName*, множеством варьируемых параметров-участников отношения *VarParams,* репрезентативным названием *Title* и множеством доменов *Domains* – предметных областей, которым принадлежит функционал. Коротко будем обозначать функционал *F*(*RelationName, VarParams, Title, Domains*).

Каждый $p \in VarParams$ – это кортеж.

$$< Name : String,$$
$$AllowedTypes : Set(Classifier),$$
$$Multiplicity : Number,$$
$$Direction : d \in \{in, out, in\_out, const\}, Value : Thing >$$

Name(p) – имя параметра;

AllowedTypes(p) – множество допустимых типов для значения параметра;

Multiplicity(p) – допустимое количество экземпляров *p* в отношении *F*;

Direction(p) – место формирования значения параметра. Возможны следующие значения: *in* – параметр получает свое значение вне поведенческого элемента, характеризуемого отношением *F*, и используется *F* как источник информации; out – получает свое значение в поведенческом элементе, характеризуемом отношением *F*, используется как канал передачи результата функционирования поведенческого элемента; in_out – параметр получает свое значение вне поведенческого элемента, характеризуемого отношением *F*, но может быть изменен в процессе его функционирования; const – значение параметра задано как константа и не передается во время вызова (используется в процессе уточнения функционалов).

Value – значение параметра (может быть не определено).

Пример функционала:

F(
CalculateEquationRoots,
{ <equation, {SquareEquation}, 1, in, null>,
<roots,{List<float>}, 1, out, null>
},
"Вычислить корни %roots% квадратного уравнения %equation%",
{Square Equation}
).

Универсумы типов параметров и функционалов не замкнуты (расширяемы), т.е. их множества не предопределены, а дополняются по мере необходимости при появлении функциональных блоков, которые невозможно выразить комбинацией уже определенных элементов.

Функционал является декларативным описанием действия, не определяя метод его реализации.

Отношение «принадлежности действия к предметной области» (свойство *Domains*) транзитивно, т. е.

$$((F \in Domain) \wedge (\exists Domain_1 \bullet Domain \in Domain_1)) \rightarrow F \in Domain_1 .$$

Определение 2. Функционал *F2*(*R2, VP2, T2, D2*) уточняет функционал *F1*(*R1, VP1, T1, D1*), если R2=R1 и можно установить взаимно однозначное соответствие $VP1 \xleftrightarrow{R} VP2$, т.е.

$$R(VP2, VP1) \text{ и } R^{-1}(VP1, VP2):$$

$$|VP1| = |VP2| \wedge$$
$$\exists p1 \in VP1,\ p2 \in VP2 \bullet$$
$$AllowedTypes(p2) \subseteq AllowedTypes(p1) \wedge$$
$$Multiplicity(p2) \subseteq Multiplicity(p1) \wedge$$
$$(Direction(p2) = Direction(p1) \wedge Value(p2) = null \vee$$
$$Direction(p2) = const \wedge Value(p2) \neq null)$$

Отношение уточнения будем обозначать $F1 \triangleright F2$, означает «*F*2 уточняет *F*1».

Пример уточнения $F1 \triangleright F2$:

*F*1:

В <obj, {графе, дереве}, 1, in_out> придать значение свойству <prop_name, {string}, 1, in> так, чтобы выполнялись условия

<cond,{BoolExpression}, +, in>.

*F*2:

ColoredGraph:

$$< vertices : Set(v \in Vertex),$$
$$edges : Set(v1 \in Vertex \times v2 \in Vertex),$$
$$coloring : Set(v \in vertices \rightarrow c \in Color) >$$

Для <obj, {ColoredGraph}, 1, in_out> придать значение свойству <prop_name,{string},1,const, «coloring»> так, чтобы выполнялись условия

<cond,{BoolExpression},+, const,

$$inv1 : \forall v1, v2 \in vertices(obj) \bullet$$
$$(v1, v2) \in edges(obj) \rightarrow coloring(v1) \neq coloring(v2),$$

$$inv2 : \forall c1 : Set(coloring(obj)) \bullet$$
$$\bullet |ran(c1)| \geq |ran(coloring(obj))| .$$

Определение 3. Последовательность функционалов *S2 уточняет* последовательность *S1,* если $|S1| = |S2| \wedge \forall i = 1..|S1| \bullet F1_i \triangleright F2_i$.

Использование отношения уточнения и принадлежности к некоторому домену позволяет структурировать множество функционалов и, соответственно, описываемые ими процессы.

3.3. PSM-модель

Является детализацией PIM, содержащей зависящие от платформы составляющие. К ним относятся: тип целевого приложения, технология

реализации его компонент и их взаимодействия, используемые языки программирования. Информация об этих параметрах может быть получена частично из DAC-диаграмм (интерфейсы взаимодействия агентов), частично от эксперта-проектировщика.

PSM-модель состоит из нагруженных дополнительными атрибутами DAC-диаграмм, а также трансформированных UI-диаграмм CIM, CD и PD-диаграмм PIM.

Модификации состоят в следующем: 1) в DAC-диаграммах каждый internal-агент помечен двумя дополнительными характеристиками – технологией реализации (ASP.NET Application, WinForms, .NET WebService и т.д) и языком реализации (C++, C#, Python и т.д.); 2) в UI-диаграммах каждый независящий от платформы визуальный компонент заменен его конкретным аналогом на основании выбранной платформы реализации агента DAC; 3) CD-диаграмма дополнена утилитными классами из библиотек выбранной платформы разработки для обеспечения реализации указанных в DAC протоколов взаимодействия агентов (например, каналы связи, real и transparent proxy в .NET Remoting); 4) классы CD-диаграммы модифицированы: все унифицированные типы данных заменены на их конкретные аналоги в выбранном для агента DAC языке программной реализации; 5) PD-диаграммы дополнены вызовами методов утилитных классов модифицированной CD-диаграммы для реализации заявленных протоколов взаимодействия (например, созданием канала связи и регистрацией предоставляемого сервиса в .NET Remoting).

Информация о соответствии типов данных PIM и PSM, а также о сущностях, необходимых для реализации того или иного протокола извлекается из базы знаний системы проектирования.

ВЫВОДЫ

Идеи, заложенные в основу методологии MDD, призваны перевести процесс разработки ПО на качественно новый уровень. В работе выделены важные теоретически или практически непроработанные составляющие этой методологии, не позволяющие применять её на практике. Предлагаемые формализованные семантические модели программ являются первым звеном в их цепочке. Они позволяют организовать разработку программной системы как последовательный процесс уточнения её модели от идеи бизнес-компонент и описания их взаимодействия до функционирующих элементов системы. Формальная спецификация семантики элементов модели в перспективе предполагает разработку методов их машинной верификации и автоматизацию процессов трансформации с более высоких уровней описания до программных модулей. Семантическое описание позволяет присоединить к системе проектирования постоянно пополняющуюся базу знаний уже однажды созданных решений и критериев, позволяющих определить возможность их повторного использования.

Литература.

[1] *Маклаков С.В.* BPwin и ERwin. CASE - средства разработки информационных систем. М.: «Диалог - МИФИ», 2000. – 256 с.

[2] *Kleppe A., Warmer J., Bast W.* MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley Professional, 2003. –192 p.

[3] *Greenfield J., Short K.* Software Factories. Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, 2004. – 500 p.

[4] *Snook C., Butler M., Edmunds A., Johnson I.* Rigorous development of reusable, domain-specific components, for complex applications //Proceedings of Third International Workshop on Critical Systems Development with UML. Lisbon, 2004. PP. 115-129.

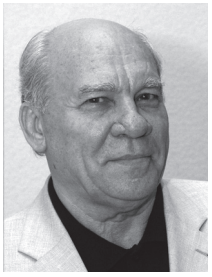
[5] *Hoare C. A. R.* Communicating Sequential Processes. University of Oxford, Prentice Hall International, 2004. – 260 p.

[6] *Khurshid S., Marinov D.* TestEra: Specification-Based Testing of Java Programs Using SAT // Automated Software Engineering. 2004, № 11. PP. 403–434

[7] *Мак-Дональд М.* WPF: Windows Presentation Foundation в .NET 3.0 для профессионалов. «Вильямс», 2008. – 992 с.

[8] UML 2.0 OCL OMG Adopted Specification / ptc/03-10-14, April 30, 2004. – 226 p.

Поступила в редколлегию 12.05.2008



Дюбко Геннадий Федорович, кандидат технических наук, профессор каф. ПО ЭВМ ХНУРЭ. Круг научных интересов: методы формализации естественного языка, формальные подходы в проектировании программных и аппаратных систем, разработка трансляторов.

Лещинская Елена Леонидовна, аспирант, асс. каф. ПО ЭВМ. Круг научных интересов: формальные подходы в проектировании программных систем, разработка трансляторов, поисковые системы.

Черепахин Виталий Михайлович, кандидат физ.-мат. наук, профессор каф. ПО ЭВМ ХНУРЭ. Круг научных интересов: демонстрационные модели для обучающих систем, автоматизированные системы управления, формальные подходы в проектировании программного обеспечения.