УДК 004.4'22:004.8

Г.Ф. Дюбко, Е.Л. Лещинская

Харьковский национальный университет радиоэлектроники

ИНТЕЛЛЕКТУАЛЬНЫЕ МЕТОДЫ ПОДДЕРЖКИ АВТОМАТИЗИРОВАННОЙ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В рамках разработанной на основе методологии MDD общей модели интеллектуальной системы автоматизированного проектирования ПО предложено формальное описание следующих методов, позволяющих получить её окончательную реализацию: метод подбора уточняющих компонент модели ПО на основе их семантических аннотаций; метод проверки выполнимости ограничивающих спецификаций моделей ПО, а также контроля соответствия реализации ПО его спецификации; метод обучения системы проектирования.

Ключевые слова: *MDD, автоматизация программирования, семантическая аннотация, верификация, обучение.*

Введение

Постановка проблемы. Model Driven Development (MDD) – стремительно развивающаяся инновационная методология разработки ПО, способная перевести этот процесс на качественно новый уровень [1]. MDD рассматривает процесс разработки как постепенное уточнение модели будущего продукта от высокоуровневой спецификации до программного кода. Выполненная в работе [2] формализация моделей программы, используемых на различных этапах MDD, и предложенный авторами подход семантического аннотирования программных компонентов позволяют разработать в рамках MDD модель обучаемой среды проектирования ПО и развить для неё целый ряд неприменимых ранее методов автоматизации создания и верификации модели, позволяющих повысить производительность разработчика и надежность создаваемого продукта.

Целью настоящего исследования является разработка методов: 1) подбора уточняющих компонент модели на основе их семантических аннотаций; 2) проверка выполнимости спецификаций ограничений для построенных моделей ПО, а также контроль соответствия реализации ПО его спецификации; 3) обучения системы проектирования.

Анализ последних исследований и публикаций. В основу предлагаемого метода подбора уточняющих компонент модели по их семантическим аннотациям были положены результаты исследований, связанные с концепцией Semantic Web (SW). В рамках SW развита методология создания онтологического хранилища знаний о некоторой предметной области и аннотирования ресурсов (документов) системы с помощью набора RDF-триплетов [3]. Наличие единой онтобазированной БЗ позволяет определять степень близости двух ресурсов на основе сравнения их семантических характеристик [4], что значительно повышает точность сравнения.

Исследования по выполнимости формальных спецификаций основываются на теории моделей Тарского. Вопросы определения выполнимости формальных спецификаций ПО, таких как Object-Z и Event-B, рассматриваются в работах [5,6], основанных на методах автоматического доказательства теорем, что в общем случае приводит к неразрешимости задачи выполнимости. В данной работе развита идея ограниченной верификации модели на основе решения задачи CSP и теории программирования в ограничениях, впервые предложенная в работе [7].

В работе также развит метод верификации реализации на соответствие спецификации, основанный на идее контрактных спецификаций, изложенный в [8].

Среди различных подходов к обучению для интеллектуальной системы проектирования было выбрано обучение на основе подкрепления [9], наиболее отражающее естественные процессы, проходящие в выбранной предметной области (процесс обучения проектировщика).

Метод подбора уточняющих компонент модели на основе их семантических аннотаций

Уточнение – основная операция, выполняемая в процессе проектирования модели программной системы (ПС). «Уточнением» в терминах семантической модели программы будем называть преобразование, состоящее в замене элементарного блока любой из её диаграмм на отличный от него, согласующийся с ним по формальным признакам (зависят от типа уточняемого блока) элементарный или составной блок. Степенью семантической близости компонент С1 и С2 некоторой ПС (на основании

прикрепленных к ним аннотаций A1 и A2 соответственно) назовем число, полученное при анализе взаимной семантической близости понятий, образующих аннотации из терминов, определенных в онтологии O.

Для вычисления *степени семантической близости* аннотаций определим функцию Sim(U,U), вычисляющую степень семантической близости двух понятий, находящихся в единой таксономии, а также функцию $\Xi(A, A)$, вычисляющую *степень семантической близости двух аннотаций* на основании Sim.

Пусть E – множество всех возможных составляющих некоторой аннотации (действие, объект действия, направление действия и т.д.). Введем функцию hasValue(A,E) задающую отображение $A \times E \xrightarrow{\text{hasValue}} U$, определяющее концепт, соответствующий определенному элементу аннотации *A*. Если некоторый элемент $e \in E$ отсутствует в аннотации A, то функция hasValue(A,e) вернет наиболее общее понятие онтологии Thing. Тогда функция $\Xi(A, A)$ может иметь следующее определение:

$$\Xi(A1, A2) = \sum_{\substack{\forall e:E \text{ in } A1 \bullet \\ c1=\text{hasValue}(A1,e), \\ c2=\text{hasValue}(A2,e)}} \text{Sim}(c1, c2).$$

Определение функции Sim(U,U) основывается на том, что в простейшем случае о концептах из онтологии O известно лишь их положение в единой иерархической структуре. Пусть depth – функция, сопоставляющая каждому узлу иерархии целое число, соответствующее его глубине. Тогда, если c_0 – наиболее близкий общий предок c_1 и c_2, то:

$$\text{Sim}(c_1, c_2) = \text{depth}(c_1) + \text{depth}(c_2) - 2 * \text{depth}(c_0).$$

Пусть необходимо подобрать все компоненты, которые могут являться уточнением элемента P с аннотацией A. Подбор начинается поиском компонент $P_i, i \in [1, n]$, для аннотации A_i которых выполнен критерий формальной согласованности и условие:

$$\exists e : E \bullet \text{Sim}(\text{hasValue}(A, e), \text{hasValue}(A_i, e)) \le \text{Threshold}.$$

Threshold – это некоторый порог (целое положительное число), задающий границу области близости понятий, которые принимаются к рассмотрению при поиске семантических аналогов.

Для каждого из найденных компонент вычисляется $\Xi(A, A_i)$, на основании которого они упорядочиваются по возрастанию.

Метод верификации спецификаций ограничений для построенных моделей ПО

Формальное ограничение на компонент модели зависит от вида компонента (структурный, поведенческий) и может содержать: инварианты компонента, предусловия и постусловия функционирования компонента (для поведенческих компонент), допустимую трассу вызовов методов компонента (для структурных компонент). В отличие от стандартного определения постусловия предлагается использовать постусловие, обязательно представленное в виде посылка $\rightarrow$ следствие.

Формальная спецификация ограничений считается корректной, если она выполнима. *Выполнимой* называется такая формальная спецификация ограничений программной системы, для которой возможно инстанциировать хотя бы один экземпляр программной системы, полностью удовлетворяющий всем указанным в спецификации ограничениям. Следовательно, для проверки корректности формальной спецификации необходимо показать возможность создания конечного числа определенных в модели объектов и связей между ними, таким образом, чтобы ни одно ограничение спецификации не было нарушено. Множество контролируемых ограничений формируется на основании явно специфицированных ограничений (инвариант, пред- и постусловий и т.д.) и неявных ограничений, которые извлекаются из BOD, CD или PD-диаграмм.

Пусть $\text{StaticState}_M =< \text{Instances}, \text{Relations} >$ – состояние конкретной инстанциации модели программной системы, представляющее кортеж двух множеств: Instances (множество экземпляров специфицированных классов) и Relations (множество отношений между элементами Instances):

$$\text{Instances} = \bigcup_{\forall C \in \text{Classes}(M)} \text{Instances}_C;$$

$$\text{Instances}_C = \{\text{inst} \mid \text{inst} =< \text{Iid}, \text{aid}_1 \ldots \text{aid}_n >\},$$

где Iid – уникальный в пределах Instances_C идентификатор экземпляра;

$n = C.\text{ownedAttributes} \rightarrow \text{size}()$ – кол-во непосредственных атрибутов класса C;

aid_i – идентификатор экземпляра, содержащего значение соответствующего его позиции атрибута класса.

$$\text{Relations} = \bigcup_{\forall as \in \text{Associations}(M)} \text{Relations}_{as};$$

$$\text{Relations}_{as} = \{\text{re} \mid \text{re} =< \text{Iid}_{\text{role1}}, \text{Iid}_{\text{role2}} >\},$$

где $\text{Iid}_{\text{role}_i}$ – идентификатор экземпляра, соответствующего i-ой роли ассоциации.

Для определения выполнимости формальной спецификации необходима её трансляция из терминов MDD модели в набор ограничений $\text{Constraints}(\text{StaticState}_M)$, накладываемых на элементы StaticState_M. Приведем пример ограничений, формируемых на основе спецификации метода для определения его применимости, т.е. существования

инстанциации модели, при которой метод может быть вызван (его предусловие должно быть истинно).

Пусть $Parameters_p$, где $p \in Processes_M$, – множество параметров процесса p,

$$Parameters_p = \{pr \mid pr =< ParamName, ClassName, Iid > ParamName, ClassName : String, Iid \in Instances_{ClassName}\}.$$

Обозначим множество инвариант, ограничений на основе множественности ассоциативных связей и связей наследования группой статических ограничений $StaticConstraints(StaticState_M)$, а объединение $StaticState_M$ и $Parameters_p$ – Runtime состоянием системы в контексте процесса p, которое будем обозначать

$$RuntimeState_{M/p} = StaticState_M \cup Parameters_p .$$

Пусть также Pre_p – множество предусловий процесса p, тогда ограничение, соответствующее корректному предусловию будет выглядеть так:

$$\forall p \in Processes_M (\exists r \in Runtime_{M/p} = s \in StaticState_M \cup Params \in Parameters_p \bullet StaticConstraints(s) \wedge Pre_p(s, Params)).$$

Множество составленных для диаграммы MDD ограничений – это не что иное, как самостоятельная задача удовлетворения ограничений [10] (Constraint Satisfaction Problem – CSP) – одна из базовых задач искусственного интеллекта.

Фиксация доменов переменных, участвующих в решении задачи, позволяет получить её решение за конечное время.

Метод контроля соответствия реализации ПО его спецификации

Формальная спецификация ограничений не определяет однозначно способ реализации компонента. Поэтому задача полной реализации компонента не может быть решена автоматически. Следовательно, после завершения реализации компонента он должен быть проверен на соответствие прикрепленной к нему формальной спецификации посредством функционального тестирования компонента.

Процесс тестирования заключается в создании экземпляра тестируемого объекта в его начальном состоянии и вызове методов компонента с целью покрытия всех достижимых комбинаций значений элементарных условий, из которых составлена формальная спецификация компонента.

Достижимой назовем такую комбинацию элементарных условий, которая не противоречит инвариантам тестируемого компонента.

Результатом функционального тестирования является модель поведения тестируемого компонента в виде конечного автомата $AM =< A, S, \delta >$, где

A – множество всевозможных примитивных тестовых воздействий (входной алфавит автомата);

S – множество состояний автомата (т.е. множество обобщенных состояний объекта тестирования);

δ – множество функций перехода вида:

$$\exists st1, st2 \in S, act \in A \bullet (st1, act) \rightarrow st2 .$$

Примитивное тестовое воздействие – это воздействие, соответствующее единичному вызову одного из методов тестируемого компонента с определенным набором параметров.

В основе *метода построения автомата состояний проверяемого компонента* в процессе функционального тестирования лежит принцип обхода ориентированного графа в глубину. Нетривиальность обхода связана с невозможностью в общем случае (если в строящемся автомате отсутствует допустимое воздействие $(Sl, act) \rightarrow S2$) выполнить возврат из состояния S1, все переходы из которого уже добавлены в автомат, в некоторое ранее анализируемое состояние S2, в котором еще остались допустимые, но не проанализированные переходы. Для решения этой проблемы предлагается хранить множество не полностью обработанных состояний автомата в виде отображения:

$$SId \xrightarrow{SatesMap} < AvailableActs, ReachSeq > ,$$

где SId – идентификатор состояния автомата;

AvailableActs – множество допустимых в состоянии SId воздействий, еще не добавленных в автомат;

$$ReachSeq - \|a_i\|, i \in Z, \exists a \in A \bullet a_i = a$$

и $(Init, ReachSeq) \Rightarrow SId$,

а также $\forall rs \in \{rs \mid (Init, rs) \Rightarrow SId\} \bullet |ReachSeq| \le |rs|$ – минимальная последовательность воздействий, переводящая автомат из начального состояния в SId.

Стоимостью перехода в состояние автомата s_i будем называть количество воздействий, которые необходимо приложить к начальному состоянию автомата, чтобы перевести его в s_i.

Для оптимизации выбора очередного наиболее дешевого по стоимости перехода еще не до конца обработанного состояния предлагается использовать очередь с приоритетами StatesQ. Каждый узел такой очереди представляет собой кортеж $< SId, Priority >$, где Priority – стоимость перехода в SId, вычисляется как

$$Priority = | ReachSeq(SatesMap(SId)) | .$$

Элементы из StatesQ выбираются в порядке возрастания Priority.

Тогда метод построения автоматной модели тестируемого компонента может быть выражен сле-

дующим образом:

1) создать тестируемый компонент с начальным состоянием Init;

2) построить

$AvailableActs(Init)$, $ReachSeq(Init) = 0$,

$StatesMap \leftarrow (Init, < Available(Init), ReachSeq(Init) >)$

$StatesQ \leftarrow < Init, ReachSeq(Init) >$;

3) выбрать очередной элемент s из StatesQ;

4) выполнить очередное допустимое воздействие act из $AvailableActs(s)$;

5) $AvailableActs(s) = AvailableActs(s) @ pre \setminus \{act\}$;

6) добавить переход $(s, act) \rightarrow s\_new$ в автомат тестируемого компонента;

7) если $AvailableActs(s) = \varnothing$, то $StatesQ = StatesQ @ pre \setminus \{s\}$;

8) пусть новое состояние тестируемого компонента после воздействия s_new, тогда:

8.1) если $StatesMap(s\_new) = \varnothing$, то построить $AvailableActs(s\_new)$, $ReachSeq(s\_new)$ и добавить s_new в StatesMap

$$StatesMap \leftarrow (s\_new, < AvailableActs(s\_new), ReachSeq(s\_new) >);$$

8.2) если $StatesMap(s\_new) \neq \varnothing$, то пусть $ReachSeq_s(s\_new)$ – путь к состоянию s_new через s;

9) введем операцию RefreshBy(st), которая заключается в следующем:

$$\forall s \in StatesMap \wedge \exists ReachSeq_{st}(s)$$

если $ReachSeq_{st}(s) < ReachSeq(StatesMap(s))$, то $ReachSeq(StatesMap(s)) = ReachSeq_{st}(s)$, RefreshBy(s); применить операцию RefreshBy(s_new);

10) если $AvailableActs(s\_new) \neq \varnothing$, то перейти к шагу 4, иначе, если $StatesQ \neq \varnothing$, то создать тестируемый компонент с начальным состоянием Init, перейти к шагу 3, иначе – автоматная модель тестируемого компонента построена.

Если в процессе тестирования очередное состояние компонента нарушает набор ограничений формальной спецификации, то построение автомата немедленно прекращается, что свидетельствует об обнаружении ошибки реализации. В противном случае построенный автомат может быть предоставлен проектировщику как результат положительного тестирования компонента.

Метод обучения системы проектирования

Цель обучения – пополнение базы знаний системы проектирования правилами предпочтения, на основании которых возможно осуществить т.н. Case Based Reasoning (CBR) [11] – логический вывод на основании прецедентов или поиск решения по аналогии. В контексте интеллектуальной системы автоматизации разработки ПО под принятием решений на основе прецедентов будем подразумевать уточнение приоритета при выборе одного из близких по семантической аннотации ранее разработанных компонентов или процессов на основании истории ранее принятых решений в аналогичном контексте, выраженной в виде набора правил предпочтения следующего формата:

$$ContextPattern \rightarrow Recommendation,$$

где ContextPattern – это конъюнкция фактов о свойствах диаграммы, для использования в которой происходил подбор компонента; Recommendation – рекомендация по поводу использования компонента в обозначенном контексте. Множество фактов посылки может быть следующим: License(c, licInfo), Version(c, versInfo), Assembly(c, assemblInfo), ComponentTypeName(c) и пр.

Блок следствия может принимать одно из следующих значений: «рекомендуется использовать», «не рекомендуется использовать», «рекомендуется заменить на».

В добавок к вышеописанному, каждое правило характеризуется неотрицательным рациональным числом – степенью доверия к выраженной в нем рекомендации. При установлении гомоморфизма между ContextPattern и характеристиками диаграммы, для которой осуществляется подбор уточняющего компонента, степень доверия влияет на изменение положения характеризуемого правилом компонента в общем перечне найденных при поиске по принципу близости аннотаций.

Правила предпочтения формируются и модифицируются системой автоматически в ответ на действия пользователя в процессе проектирования. Память, в которой могут храниться *правила,* разделяется на две области: долгосрочная и рабочая. Долгосрочная память – это база знаний системы проектирования. Рабочая память сохраняется только в период между двумя запусками проекта.

Принципы создания *правил предпочтения* в рабочей памяти:

- пусть на запрос проектировщика о поиске некоторого компонента для заполнения неуточненного блока из результирующего списка был выбран и применен компонент c, который до следующего момента запуска не был заменен, тогда для компонента c формируется правило вида: $ContextPattern \rightarrow RecommendToUse$;
- пусть компонент $\mathbf{C}_1$ модели был заменен на компонент $\mathbf{C}_2$, который остался на этом месте до следующего запуска проекта, тогда для компонента $\mathbf{C}_1$ формируется правило вида:

$$ContextPattern \rightarrow RecommendToReplaceWith(\mathbf{C}_2);$$

– пусть была удалена часть диаграммы, содержащая компоненты $\mathbf{C}_i$, находящиеся на своих местах с момента последнего запуска проекта, тогда для компонента $\mathbf{C}_i$ формируется правило вида: $\text{ContextPattern} \rightarrow \text{RecommendNotToUse}$.

Сформированное обозначенным методом множество *правил* в момент запуска генерации исходного кода проекта приводит к модификации постоянной памяти по следующим принципам: для каждого правила r_i рабочей памяти

– если r_i подтверждает присутствующее в постоянной памяти правило r, но не несет дополнительных знаний, то уровень доверия r увеличивается согласно функции изменения

$$\text{DofB}_r^{j+1} = \text{DofB}_r^{j} + f_\Delta(\text{DofB}_r^{j});$$

– если r_i подтверждает присутствующее в постоянной памяти правило r, и несет дополнительные знания, например в постоянной памяти находится правило

$r : \text{ContextPattern} \rightarrow \text{RecommendNotToUse}$,

а в рабочей

$r_i : \text{ContextPattern} \rightarrow \text{RecommendToReplaceWith}$,

то r замещается r_i со степенью доверия r;

– если r_i опровергает присутствующее в постоянной памяти правило r, то уровень доверия к r уменьшается согласно функции изменения

$$\text{DofB}_r^{j+1} = \text{DofB}_r^{j} - f_\Delta(\text{DofB}_r^{j});$$

– если в постоянной памяти отсутствуют правила, как подтверждающие, так и опровергающие r_i, то r_i добавляется в постоянную память с начальным уровнем доверия, равным 1.

Отношения подтверждения и *опровержения* правил определяются следующим образом:

– для двух правил r_1 и r_2 r_1 *опровергает* r_2, если $\text{Context}_1 = \text{Context}_2$ и пара $(\text{Recommendation}_1, \text{Recommendation}_2)$ принимает одно из следующих наборов значений (порядок значений в паре неважен): $(\text{RecommendToUse}, \text{RecommendNotToUse})$, $(\text{RecommendToUse}, \text{RecommendToReplaceWith})$;

– для двух правил r_1 и r_2 r_1 *подтверждает* r_2, если $\text{Context}_1 = \text{Context}_2$ и пара $(\text{Recommendation}_1, \text{Recommendation}_2)$ принимает одно из следующих наборов значений (порядок значений в паре неважен): $(\text{RecommendToUse}, \text{RecommendToUse})$, $(\text{RecommendNotToUse}, \text{RecommendNotToUse})$, $(\text{RecommendNotToUse}, \text{RecommendToReplaceWith})$.

При совпадении степени доверия с 0 правило удаляется из постоянной памяти.

На основании того, что выбор проектировщика является субъективным, а также для учета возможности принятия проектировщиком ошибочного решения f_Δ должно быть пропорционально по модулю текущему DofB_r, меньше его, но сравнимо с ним.

Использование знаний, выраженных в форме *правил предпочтения*, позволяет на основании контекста текущей диаграммы распознать встречавшуюся ранее ситуацию. Поскольку *правило* несет информацию о принятом и одобренном проектировщиком в той ситуации решении, то и в текущем случае логично предложить проектировщику поступить аналогично. Такие рассуждения дают основания при определении, что некоторый компонент релевантен запросу проектировщика, откорректировать его позицию в списке релевантных запросу компонентов на основании тех присоединенных к нему *правил*, которые содержат информацию о принятых в схожей ситуации решениях.

«Схожесть» формализуем в виде следующего *принципа согласования:* пусть к компоненту *c* присоединено множество *R* правил предпочтения. Пусть текущая диаграмма характеризуется набором фактов DContext о присутствующих на ней компонентах и их атрибутах, тогда согласованность правила $r \in R$ с DContext может быть определена как наличие непустого подмножества фактов $\text{SubContext} \subseteq \text{DContext}$, соответствующих посылке правила *r*. Для выполнения этого действия наиболее оптимальным является алгоритм Rete сопоставления с шаблоном, предложенный Ч. Форги [12].

Для компонента с на основании его согласующихся с контекстом правил $R' \in R$ вычисляется *показатель предпочтения* $\rho = \lfloor \rho' + \rho'' \rfloor$, где ρ' – *собственный показатель* предпочтения, ρ'' – *приобретенный показатель* предпочтения, вычисляющиеся по формулам*:*

$$\rho' = \sum_{r \in R'} \begin{cases} \text{DofB}_r, \text{ если } \text{Recommendation}_r = \\ \qquad \text{RecommentToUse}; \\ -\text{DofB}_r, \text{ если } \text{Recommendation}_r = \\ \qquad \text{RecommentNotToUse}; \\ -0,5 \times \text{DofB}_r, \text{ если } \text{Recommendation}_r = \\ \qquad \text{RecommendToReplaceWith}; \end{cases}$$

$$\rho'' = \sum_{r \in \bigcup_{\substack{\forall cp \in \text{SoftwareElement} \bullet \\ \text{Relevant}(cp,Q)}} R_{cp}'} 0,5 \times \text{DofB}_r,$$

если

$$\text{Recommendation}_r = \text{RecommendToReplaceWith}(c).$$

Компонент с смещается на ρ позиций (согласно знаку ρ вверх или вниз) в перечне компонентов,

релевантных семантическому запросу, упорядоченных по убыванию релевантности.

Выводы

В результате проведенного исследования был предложен набор интеллектуальных методов поддержки разработки ПО.

Предложенный авторами метод подбора уточняемых компонент основывается на методах поиска семантических аналогов, развитых в рамках SW, и дополняет их элементами CBR на основе правил предпочтения, учитывающих историю предпочтений использования ранее разработанных компонентов в схожих ситуациях.

Развитые в работе методы статической верификации расширяют результаты аналогичных исследований добавлением поддержки статической верификации поведенческих сущностей. Развитие принципов верификации реализации на соответствие спецификации компонента заключается в добавлении возможности автоматического построения модели состояний тестируемого компонента, цепочек тестовых воздействий.

Общий подход обучения с подкреплением перенесен на рассматриваемую проблемную область и на его основе предложены методы обучения системы проектирования, заключающиеся в автоматическом запоминании и последующем использовании предпочтений проектировщика при выборе одного из подобных, подобранных по семантическому запросу компонент в зависимости от контекста.

Разработанный набор методов делает возможным создание интеллектуальной системы автоматизированного проектирования ПО, которая способна значительно превосходить современные аналоги по показателям производительности использующей её команды разработчиков.

Список литературы

1. Дюбко Г.Ф., Черепахи В.М., Лещинская Е.Л. Современные парадигмы программирования, классификация и анализ // Восточно-Европейский журнал передовых технологий. – 2007. – № 5/2 (29). – С. 26-30.

2. Дюбко Г.Ф., Лещинская Е.Л., Черепахин В.М. Семантические модели программ в интеллектуальных средах автоматизированного проектирования программного обеспечения // Прикладная радиоэлектроника. – 2008. – Т. 7, № 2. – С. 145-150

3. OWL Web Ontology Language // World Wide Web Consortium, - September 2004. www.w3.org/2004/OWL/

4. Шевченко А.Ю. Построение систем интеграции и обработки информации на основе "онтобазируемого" подхода // Восточно-Европейский журнал передовых технологий. – 2004. – № 4. – С. 149-153.

5. Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science, 9(2):152–172, February 2003.

6. Achim D. Brucker and Burkhart Wolff. HOL-OCL: Experiences, consequences and design choices. In Jean-Marc J´ez´equel, Heinrich Hussmann, and Stephen Cook, editors, UML 2002: Model Engineering, Concepts and Tools, number 2460, pages 196–211. Springer-Verlag, Dresden, 2002.

7. Finite satisfiability of UML class diagrams by Constraint Programming / M. Cadoli, D. Calvanese, G. D. Giacomo, T. Mancini. // In Proc. Int.Workshop on Description Logics (DL'2004). – Volume 104 of CEUR Workshop Proc., 2004.

8. Clemens Fischer Combination and Implementation of Processes and Data: from CSP-OZ to Java. Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften des Fachbereichs Informatik der Carl-von-Ossietzky Universit¨at Oldenburg. – Oldenburg, 24. Januar 2000. – 298 p.

9. Рассел С., Норвиг П. Искусственный интеллект: современный подход (AIMA): 2-е издание. – С.-Пб.: Вильямс, 2007. – 1408 с.

10. Handbook of Constraint Programming / Francesca Rossi, Peter van Beek, Toby Walsh. Elsevier. – Oxford, 2006 – 977 p.

11. Aamodt A., Plaza E. Case-Based Reasoning // Foundational Issues, Methodological Variations, and System Approaches. AI Communications. IOS Press. – 1994. – Vol. 7: 1. – P. 39-59.

12. Robert B. Doorenbos Production Matching for Large Learning Systems. – CMU-CS-95-113, Computer Science Department Carnegie Mellon University, Pittsburgh, PA, 1995 – 208 p.

Поступила в редколлегию 8.08.2008

Рецензент: д-р техн. наук, проф. М.И. Сидоренко, Институт радиофизики и электроники НАН Украины, Харьков.

ІНТЕЛЕКТУАЛЬНІ МЕТОДИ ПІДТРИМКИ АВТОМАТИЗОВАНОЇ РОЗРОБКИ ПРОГРАНОГО ЗАБЕЗПЕЧЕННЯ

Г.Ф. Дюбко, О.Л. Лещинська

Розглядаються метод підбору уточнюючих компонент на основі семантичних анотацій та правил переваги, метод статичної верифікації формальних специфікацій, метод верифікації реалізації на відповідність специфікації, метод навчання системи проектування.

Ключові слова: *MDD, автоматизація програмування, семантична анотація, верифікація, навчання.*

INTELLIGENT SUPPORTING METHODS FOR AUTOMATED SOFTWARE ENGINEERING

G.F. Dubko, H.L. Leshchynska

Search method for refined components matching according to semantic annotations, method to determine formal specification satisfiability, to check realization to formal specification conformance, method of intelligent system learning are introduced.

Keywords: *MDD, automation of programming, semantic annotation, verification, teaching.*