УДК 004.82, 004.42, 004.021

Язык спецификации дискретных динамических систем, ориентированных на знания, структурированные в онтологиях

Ануреев И.С. (Институт систем информатики СО РАН)

В статье рассматривается язык ABML (Attribute-Based Modeling Language), предназначенный для спецификации и прототипирования дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Язык позволяет формально описывать как онтологические модели систем, так и правила их функционирования, включая динамическое изменение структуры знаний и состояний объектов.

ABML реализован как лексическое расширение диалекта Common Lisp (SBCL) и опирается на минимальный, но выразительный концептуальный базис, включающий объекты, атрибуты и типы объектов. Особое внимание уделяется разделению объектов на изменяемые и константные, а также механизмам типизации, основанным на атрибутах.

В работе подробно описаны средства языка для задания типов, создания и модификации объектов, сопоставления с образцом и вычисления атрибутов. Ключевым элементом ABML является механизм атрибутных замыканий, позволяющий формализовать контекстно-зависимые вычисления атрибутов и моделировать динамику систем в дискретном времени.

Практическая применимость языка демонстрируется на примере моделирования сушилки для рук, для которой построена онтология, а также описаны правила инициализации и функционирования системы. Представленный подход показывает, что ABML может служить удобным инструментом для онтологического моделирования интеллектуальных, информационных и программных систем.

Ключевые слова: онтологии, атрибуты, онтологические модели, графы знаний, атрибутные замыкания, ABML, дискретные динамические системы

1. Введение

Современные информационные и программные системы все чаще проектируются как сложные дискретные динамические системы, функционирование которых опирается на структурированные знания. Такие знания, как правило, представлены в виде онтологий, определяющих понятия предметной области, их свойства и отношения. В этой связи воз-

никает потребность в формальных языках, способных единообразно описывать как структуру знаний, так и динамику изменения состояний системы во времени.

Существующие языки онтологического моделирования, такие как OWL и связанные с ним формализмы, в первую очередь ориентированы на декларативное представление знаний и логический вывод. Однако они ограничены в средствах описания поведения систем и моделирования их функционирования как последовательности дискретных шагов. С другой стороны, традиционные языки программирования обладают мощными вычислительными возможностями, но не всегда обеспечивают адекватную поддержку онтологического уровня моделирования и явной работы со знаниями.

В данной работе предлагается язык ABML, который разрабатывается как средство онтологического моделирования дискретных динамических систем. Основной целью языка является объединение онтологического подхода с возможностями процедурного и функционального программирования, что позволяет описывать как структуру системы, так и правила ее функционирования в рамках единого формализма.

ABML построен как лексическое расширение диалекта Common Lisp – языка SBCL. Выбор Lisp-подобного языка обусловлен его высокой выразительностью, развитой системой макросов и удобством встраивания предметно-ориентированных языков. Это позволяет реализовать ABML с минимальным числом новых конструкций, сохраняя при этом возможность использования всех средств базового языка.

Концептуальный фундамент ABML основан на трех ключевых понятиях: объектах, атрибутах и типах объектов. Объекты используются для представления элементов системы, атрибуты – для задания их свойств и отношений, а типы объектов – для онтологической классификации. Существенной особенностью языка является различие между изменяемыми и константными объектами, что позволяет явно контролировать семантику изменений состояний системы.

Важным элементом ABML являются механизмы сопоставления с образцом и атрибутных замыканий. Сопоставление с образцом обеспечивает удобное средство задания условий и правил поведения системы, а атрибутные замыкания позволяют формализовать вычисление атрибутов в фиксированном контексте, что особенно важно при моделировании динамики и зависимостей между компонентами системы.

Для демонстрации возможностей языка в статье рассматривается пример моделирования сушилки для рук. На этом примере показывается, как с помощью ABML можно

построить онтологическую модель системы, задать ее начальное состояние и формально описать правила функционирования в виде дискретных тактов. Тем самым иллюстрируется применимость языка для моделирования реальных технических и информационных систем.

В оставшейся части статьи последовательно рассматриваются основные компоненты предлагаемого подхода. В разделе 2 вводится базис языка ABML и формулируются его ключевые концепции. Раздел 3 посвящён системе типов языка, включая базовые типы и типы объектов, основанные на атрибутах. В разделе 4 описываются объекты, их создание и принципы работы с изменяемыми и константными экземплярами. Раздел 5 рассматривает механизмы работы с атрибутами и способы доступа и изменения их значений. В разделе 6 вводятся средства сопоставления с образцом, используемые для задания правил функционирования систем. Раздел 7 посвящён атрибутным замыканиям и их роли в моделировании динамики. В разделе 8 описывается онтология сушилки для рук как пример онтологической модели системы. В разделах 9 и 10 рассматриваются запуск и функционирование сушилки в терминах ABML. В разделе 11 проведен анализ родственных работ. В заключительном разделе подводятся итоги работы и обсуждаются направления дальнейших исследований.

2. Базис языка ABML

Язык *ABML (Attribute-Based Modeling Language)* предназначен для прототипирования дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Он позволяет специфицировать как онтологии таких систем, так и правила функционирования этих систем, меняющие как саму онтологию, так и ее содержимое. Язык является лексическим расширением SBCL (Steel Bank Common Lisp) [21] – популярного диалекта Common Lisp. Мы выбираем язык из семейства Common Lisp в качестве основы ABML как благодаря его хорошо развитым возможностям по встраиванию предметно-ориентированных языков, так и для того, чтобы иметь возможность использовать при необходимости напрямую лисповские средства.

ABML вводит в SBCL лишь небольшой набор дополнительных функций. Хотя большинство этих функций являются макросами, мы далее для универсальности будем использовать термин функция.

Мы проектируем ABML как язык для онтологического моделирования дискретных ди-

намических систем (далее – *систем*), в том числе информационных и программных систем.

ABML основан на минимальном концептуальном фундаменте, состоящем из трех базовых понятий – *объекты*, *атрибуты* и *типы объектов*:

- *Объекты* являются базовыми единицами для моделирования элементов и подсистем системы. Существуют два вида объектов: *изменяемые* (mutable) и *константные* (constant). Изменение атрибутов *изменяемого объекта* (добавление, удаление или изменение атрибута) сохраняет идентичность объекта, тогда как любое изменение атрибута *константного объекта* приводит к созданию нового константного объекта.
- *Атрибуты* определяют свойства и отношения объектов. Каждый атрибут имеет *имя*, *значение* и *тип*, который ограничивает множество допустимых значений.
- *Типы объектов* классифицируют группы объектов, обладающих общими характеристиками, и соответствуют понятиям в онтологии.

3. Типы

Типы в ABML делятся на *базовые типы* и основанные на атрибутах *типы объектов*. ABML поддерживает следующие базовые типы:

- `lispt` – множество значений Lisp;
- `symbol` – множество символов Lisp;
- `atom` – множество атомов Lisp;
- `string` – множество строк Lisp;
- `int` – множество целых чисел;
- `nat` – множество натуральных чисел (включая `0`);
- `real` – множество вещественных чисел;
- `(listt` t`)` – списки элементов типа t;
- `(uniont` t_1 ... t_n`)` – объединение типов t_1, ..., t_n;
- `(enumt` v_1 ... v_n`)` – тип, состоящий из значений v_1, ..., v_n;
- `(funt` t_1 ... t_n t`)` – функции из $t_1 \times \ldots \times t_n$ в t;
- `any` – объединение всех базовых типов и типов объектов;
- `bool` – синоним `any`, подчеркивающий, что `nil` интерпретируется как ложь, а любое значение, отличное от `nil`, – как истина.

Типы объектов делятся на *типы изменяемых объектов* и *типы константных объек-*

тов.

Тип изменяемых объектов t'' определяется как `(mot` ad_1 `...` ad_r`)`, где декларации атрибутов ad_j задают ограничения на значения атрибутов изменяемых объектов этого типа.

Пусть a', t', t'_1, t'_2 и v' – значения выражений a, t, t_1, t_2 и v соответственно.

Существует четыре вида деклараций атрибутов:

1. `:av` a v – объявляет атрибут a' со значением v'. Значение этого атрибута для любого значения (называемого экземпляром) типа t'' всегда равно v'.
2. `:at` a t – объявляет атрибут a' с типом t'. Значение этого атрибута для любого экземпляра типа t'' должно принадлежать типу t'. На месте t может быть лямбда-выражение с одним аргументом, выступающее в роли характеристической функции: если на значение атрибута функция возвращает значение, отличное от `nil`, то такое значение атрибута считается допустимым.
3. `:atv` a t v – объявляет атрибут a' с типом t' и значением по умолчанию v'. Помимо ограничения, описанного в пункте (2), это объявление присваивает значение v' атрибуту a' во всех создаваемых экземплярах типа t'', если иное значение не было задано при создании экземпляра.
4. `:amap` t_1 t_2 – объявляет множество значений типа t'_1 в качестве атрибутов, значения которых принадлежат типу t'_2. Значение любого атрибута a в экземпляре типа t'' должно принадлежать t'_2, если a является элементом t'_1.

Тип константных объектов определяется как `(cot` ad_1 `...` ad_r`)`, где объявления атрибутов ad_j аналогичным образом задают ограничения на значения атрибутов константных объектов.

Для краткости обозначения

```
1 mot
2 cot
```

используются как сокращения для стандартных определений типов объектов

```
1 (mot)
2 (cot)
```

В ABML новые типы могут определяться с помощью конструкции `(typedef` n t`)`, которая объявляет тип с именем n как синоним типа t. Для удобства используются сокращенные формы

```
(mot n ad_1 ... ad_r)
(cot n ad_1 ... ad_r)
```

вместо эквивалентных определений

```
(typedef n (mot ad_1 ... ad_r))
(typedef n (cot ad_1 ... ad_r))
```

4. Объекты

Объекты могут появляться в модели системы только через специальные функции, которые порождают экземпляры типов объектов.

Для изменяемых объектов функция генерации экземпляров имеет вид

```
(mo t ad_1 ... ad_r),
```

где допускаются только объявления атрибутов вида `:av` a v. Эта функция создает новый изменяемый объект o типа t и присваивает ему атрибуты и их значения в соответствии с объявлениями атрибутов ad_j. Объект o также наследует все атрибуты со значениями по умолчанию, определенные в типе t.

Генерация экземпляров типов объектов и последующие изменения их атрибутов и значений этих атрибутов подчиняется двум принципам.

Принцип ограниченности гласит, что присваиваемые атрибутам объекта значения должны удовлетворять ограничениям деклараций атрибутов типа, экземпляром которого этот объект является.

Принцип открытости утверждает, что экземпляр любого типа объектов может содержать атрибуты, явно не объявленные в этом типе, причем значения таких необъявленных атрибутов ничем не ограничены.

ABML включает предопределенные функции для работы с изменяемыми объектами и типами изменяемых объектов:

- `(uid` o`)` – возвращает уникальный идентификатор объекта o. Этот уникальный идентификатор является натуральным числом, однозначно идентифицирующем этот объект – никакие два сгенерированных экземпляра типа изменяемых объектов не могут иметь одинакового идентификатора;
- `(imax` t`)` – возвращает количество экземпляров типа t;

- (otype o) – возвращает тип объекта o, т. е. (otype o) = t;
- (is-instance o t) – проверяет, является ли объект o экземпляром типа t;
- (attributes o) – возвращает список непустых атрибутов объекта o. Атрибут считается непустым, если его значение отличается от nil.

Для константных объектов функция генерации экземпляров имеет вид

```
(co t ad_1 ... ad_r).
```

Все, что описано выше для изменяемых объектов, также применимо и к константным объектам, за исключением того, что для них и их типов не определены функции uid и imax, соответственно.

Для удобства используются сокращенные формы

```
(mo ad_1 ... ad_r)
(co ad_1 ... ad_r)
```

вместо эквивалентных определений

```
(mo mot ad_1 ... ad_r)
(co cot ad_1 ... ad_r)
```

5. Атрибуты

Для добавления новых деклараций атрибутов к типам объектов используется функция

```
(att t ad_1 ... ad_r),
```

которая добавляет декларации атрибутов ad_1, ..., ad_r к типу t.

Для получения значения атрибута a объекта o используется функция (aget o a). Для установки значения v атрибута a объекта o применяется функция (aset o a v).

Эти функции также работают со списками (listt), где индексы трактуются как атрибуты. В этом случае индекс не должен превышать длину списка при использовании aset и должен быть строго меньше длины списка при использовании aget (так как индексация списков начинается с 0). В противном случае возвращается ошибка.

Они поддерживают также работу с атрибутами на любом уровне вложенности

```
(aget o a_1 ... a_n)
(aset o a_1 ... a_n v)
```

В этом случае, сначала вычисляется атрибут a_1, затем вычисляется атрибут a_2 на значении v_1 атрибута a_1 и т. д.

Для вложенного вычисления атрибутов также применяются эквивалентные записи

```
(aget o (aseq a1 ... an))
(aset o (aseq a1 ... an) v)
```

с использованием формы (aseq a_1 ... a_n).

Если список атрибутов явно не задан, вместо формы (aseq ...) используется форма (aseql e). В этом случае список атрибутов вычисляется как значение выражения e.

Также имеется сокращенная форма

```
(aset o :av a1 v1 :av ... :av an vn)
```

эквивалентная вложенной форме

```
(aset (... (aset o a1 v1) ...) an vn)
```

представляющей последовательные применения функции aset.

Поведение функции aset зависит от того, применяется ли она к изменяемому или константному объекту. Для изменяемых объектов функция обновляет значение указанного атрибута без изменения самого объекта. В отличие от этого, при применении к константным объектам создается новый константный объект, идентичный исходному, за исключением обновленного значения атрибута. Для списков функция ведет себя так же, как и для константных объектов.

Функция acall является атрибутно-ориентированным вариантом функции aget и трактует атрибут как функцию:

- (acall a o) эквивалентна (aget o a);
- (acall a o v_1 ... v_s) применяет функцию с s аргументами, хранящуюся в атрибуте a объекта o, к аргументам v_1, ..., v_s.

6. Сопоставление с образцом

Язык ABML имеет развитые средства сопоставления с образцом, основанные на сопоставителях (matchers) вида

```
(match c1 ... cr)
(nmatch c1 ... cr)
```

которые состоят из последовательности клозов сопоставления c_j и реализуют чередование сопоставления с образцом и действий, выполняемых при успешном или неуспешном сопоставлении.

Клозы сопоставления делятся на три категории: *клозы атрибутов*, *клозы выражений* и *клозы действий*.

Пусть e', a', v' и t' обозначают значения e, a, v и t, соответственно.

Клозы атрибутов выполняют сопоставление значений атрибутов. ABML поддерживает три вида клозов атрибутов:

1. `:av` e a v – сопоставление успешно, если e' является объектом и его атрибут a' имеет значение v'.
2. `:at` e a t – сопоставление успешно, если e' является объектом и значение его атрибута a' принадлежит типу t'.
3. `:ap` e a p – сопоставление успешно, если e' является объектом. Параметру p, называемую параметром сопоставителя, присваивается значение e'.

Формы `(aseq` a_1 ... a_n`)` и `(aseql` e`)` также могут использоваться вместо одиночных атрибутов.

Клозы выражений выполняют сопоставление значений выражений. ABML поддерживает три вида клозов выражений:

1. `:v` e v – сопоставление успешно, если e' равно v'.
2. `:t` e t – сопоставление успешно, если e' принадлежит типу t'.
3. `:p` e p – сопоставление всегда успешно. Параметру сопоставителяp присваивается значение e'.

Клозы действий задают действия, выполняемые при успешном или неуспешном сопоставлении. ABML поддерживает два вида клозов действий:

1. `:do` e_1 ... e_m – последовательно вычисляет выражения e_j. Эти выражения могут использовать параметры сопоставителя. После вычисления выражений сопоставление продолжается.
2. `:exit` e_1 ... e_m – последовательно вычисляет выражения e_j. Эти выражения могут использовать параметры сопоставителя. После вычисления выражений сопоставитель завершает работу, возвращая последнее вычисленное значение.

Сопоставители помимо возвращения значения, также возвращают признак того, успешно ли прошло сопоставление или нет. Поэтому их можно использовать на месте клозов

атрибутов и выражений, обеспечивая таким образом вложенные сопоставления.

Сопоставитель `match` последовательно вычисляет клозы сопоставления, входящие в него по следующим правилам:

1. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, и успешно сопоставляется, то переходим к вычислению следующего клоза.
2. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, сопоставление терпит неудачу, и оставшаяся последовательность клозов содержит `exit`-клоз, то вычисляем ближайший `exit`-клоз и завершаем работу сопоставителя с признаком успешного сопоставления.
3. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, сопоставление терпит неудачу, и оставшаяся последовательность клозов не содержит `exit`-клозов, то завершаем работу сопоставителя с признаком неудачного сопоставления.
4. Если очередной клоз является `do`-клозом, то вычисляем его и переходим к вычислению следующего клоза.
5. Если очередной клоз является `exit`-клозом, то пропускаем его и переходим к вычислению следующего клоза.
6. Если клозов больше нет, то завершаем работу сопоставителя с признаком неудачного сопоставления.

Сопоставитель `nmatch` также как и `match` последовательно вычисляет клозы сопоставления, но действует противоположным образом:

1. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, сопоставление терпит неудачу, то переходим к вычислению следующего клоза.
2. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, и успешно сопоставляется, и оставшаяся последовательность клозов содержит `exit`-клоз, то вычисляем ближайший `exit`-клоз и завершаем работу сопоставителя с признаком успешного сопоставления.
3. Если очередной клоз является клозом атрибутов, клозом выражений или сопоставителем, успешно сопоставляется, и оставшаяся последовательность клозов не содержит `exit`-клозов, то завершаем работу сопоставителя с признаком неудачного сопоставления.

4. Если очередной клоз является `do`-клозом, то вычисляем его и переходим к вычислению следующего клоза.
5. Если очередной клоз является `exit`-клозом, то пропускаем его и переходим к вычислению следующего клоза.
6. Если клозов больше нет, то завершаем работу сопоставителя с признаком неудачного сопоставления.

7. Атрибутные замыкания

В дополнение к способам вычисления значений атрибутов, описанным выше, ABML предоставляет механизм связывания атрибутов со значениями (экземплярами) любых типов и вычисления этих атрибутов в фиксированном контексте с использованием *атрибутных замыканий*.

Атрибутное замыкание задает:

- вычисляемый атрибут,
- конкретный экземпляр типа, для которого вычисляется этот атрибут,
- а также конечное множество дополнительных параметров вместе с их значениями (называемое контекстом вычисления атрибута), которые влияют на вычисление атрибута.

Замыкания атрибутов представляются в виде константных объектов.

Константный объект ac называется *атрибутным замыканием* относительно атрибута a и типа t, если выполняются следующие условия:

- `(aget` ac `"attribute") =` a
- `(aget` ac `"instance") =` i, где i является экземпляром типа t.

Остальные атрибуты объекта ac образуют контекст вычисления атрибута a.

Способ вычисления атрибутных замыканий задается декларацией атрибутного замыкания одного из следующих видов:

```
(aclosure ac :attribute a :type t :instance i s1 s2 s3 :do e1 ... er)
(aclosure ac :attribute a :type t :instance i s1 s2 s3
  :match c1 ... cr)
(aclosure ac :attribute a :type t :instance i s1 s2 s3
  :nmatch c1 ... cr)
```

где s_1, s_2 и s_3 имеют вид

```
1 :a_1 p_1 ... :a_n p_n
2 :ap w_1 b_1 q_1 ... :ap w_m b_m q_m
3 :p u_1 t_1 ... :p u_k t_k
```

соответственно.

Результат вычисления атрибутного замыкания ac для атрибута a и типа t определяется λ-функцией `(lambda (`ac`) `b`)`, где тело b имеет вид

```
1 (match :ap ac "instance" i :ap ac a_1 p_1 ... :ap ac a_n p_n
2   :ap w_1 b_1 q_1 ... :ap w_m b_m q_m :p u_1 t_1 ... :p u_k t_k :do e_1 ... e_r)
3
4 (match :ap ac "instance" i :ap ac a_1 p_1 ... :ap ac a_n p_n
5   :ap w_1 b_1 q_1 ... :ap w_m b_m q_m :p u_1 t_1 ... :p u_k t_k c_1 ... c_r)
6
7 (match :ap ac "instance" i :ap ac a_1 p_1 ... :ap ac a_n p_n
8   :ap w_1 b_1 q_1 ... :ap w_m b_m q_m :p u_1 t_1 ... :p u_k t_k
9   (nmatch c_1 ... c_r))
```

соответственно. Здесь выражения e_1, ..., e_r, c_1, ..., c_r могут зависеть от параметров i, ac, p_1, ..., p_n, q_1, ..., q_m, t_1, ..., t_k.

Часть `:instance` i s_1 s_2 s_3 декларации атрибутного замыкания называется *префиксом декларации*. Элементы префикса могут как переставляться (при этом соответствующим образом переставляются элементы в λ-функции), так и опускаться. Часть декларации, следующая за префиксом, называется *телом декларации*.

Декларация атрибутного замыкания задает способ вычисления, а само вычисление выполняется функцией `(eval-aclosure` ac`)`. Напомним, что вычисление атрибутного замыкания ac эквивалентно вычислению значения атрибута `(aget` ac `"attribute")`.

Помимо функции `(eval-aclosure` ac`)` над атрибутными замыканиями определены следующие функции:

- `(clear-aclosure` ac`)` – удаляет все атрибуты у атрибутного замыкания ac кроме `"attribute"` и `"instance"`, т. е. контекст вычисления атрибута в ac;
- `(update-eval-aclosure` ac `...)` – сначала выполняет `(aset` ac `...)`, модифицируя значения атрибутов замыкания ac, а затем `(update-eval-aclosure` ac'`)` для моди-

фицированного замыкания ac';

- `(clear-update-eval-aclosure` ac `...)` – сначала выполняет `(clear-aclosure` ac`)`, удаляя контекст вычисления атрибута в замыкании ac, а затем `(update-eval-aclosure` ac' `...)` для модифицированного замыкания ac'.

В следующих разделах будет рассмотрен такой пример дискретной динамической системы как сушилка для рук и для нее на языке ABML будет построена онтология (онтологическая модель) и правила первого запуска этой системы и ее дальнейшего функционирования.

8. Онтология сушилки для рук

Онтология (или онтологическая модель) сушилки для рук определяется тремя типами изменяемых объектов.

Тип `"system"` определяет сушилку как систему, состоящую из сенсора и контроллера:

```
(mot "system"
  :at "controller" "controller"
  :at "sensor" "sensor")
```

Тип `"sensor"` описывает сенсор через его состояние, моделирующее замечены руки или нет:

```
(mot "sensor"
  :at "state" (enumt "detected" "not detected"))
```

Тип `"controller"` моделирует контроллер, определяя такие его компоненты как

- связанный с ним сенсор `"sensor"`;
- состояние `"state"`, в котором находится контроллер (режим его работы);
- константы `"maximum drying time"` и `"cooling time"`, характеризующие максимальное время непрерывной работы сушилки и время охлаждения сушилки, заданные для простоты числом тактов работы контроллера;
- локальные часы `"local clock"`, подсчитывающие количество тактов, которые контроллер непрерывно находился в состоянии сушки или состоянии охлаждения.

Он задается следующим образом:

```
(mot "controller"
```

```
  :at "sensor" "sensor"
  :at "local clock" nat
  :at "state" (enumt "waiting" "drying" "cooling")
  :av "maximum drying time" 100000
  :av "cooling time" 1000)
```

Конкретная сушилка для рук (точнее ее состояние в определенной момент времени) может, например, быть задана (порождена) следующим образом:

```
(match
  :p (mo "sensor" :av "state" "not detected") sen
  :p (mo "controller"
    :av "sensor" sen
    :av "local clock" 0
    :av "state" "waiting") cont
  (mo "system" :av "controller" cont :av "sensor" sen))
```

Это выражение возвращает экземпляр типа "system".

Заметим, что конечные наборы экземпляров типов можно рассматривать как граф знаний, в котором вершины помечены этими экземплярами и значениями базовых типов, а дуги помечены именами атрибутов и ведут от объекта, для которого вычисляется атрибут к значению этого атрибута.

В данном примере, граф знаний, соответствующий состоянию сушилки для рук, определенному выше, имеет вид как на Рис.1.

9. Запуск сушилки

Запуск сушилки моделируется декларацией атрибутного замыкания для атрибута "init" и типа "system":

```
(aclosure ac :attribute "init" :type "system" :instance i
  :match
  :ap i "sensor" sen :ap i "controller" cont
  :do
    (aset sen "state" "not detected")
```

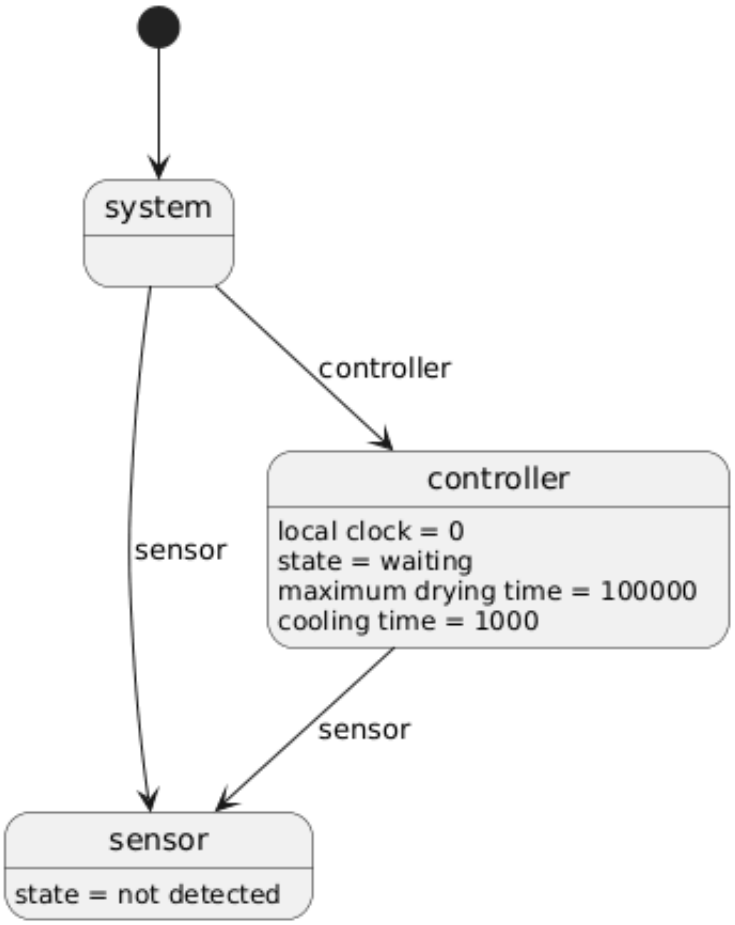

Рис. 1: Граф знаний для состояния сушилки для рук

```
    (aset cont :av "sensor" sen :av "state" "waiting" :av "local
       clock" 0))
```

Эта декларация присваивает начальные значения атрибутам компонент *sen* и *cont* системы *i*.

10. Функционирование сушилки

Функционирование сушилки также моделируется через декларации атрибутных замыканий.

Декларации для атрибута "step" определяет один такт работы системы и ее компонент.

Для системы целиком она определяется как

```
(aclosure ac :attribute "step" :type "system" :instance i :do
  (update-eval-aclosure ac :av "instance" (aget i "sensor"))
  (update-eval-aclosure ac :av "instance" (aget i "controller")))
```

Сначала выполняется такт сенсора с получением данных из окружающей среды, а затем такт контроллера на полученных данных.

Шаг для сенсора состоит в получении случайных данных и моделируется с помощью функции `random-list-element`, выбирающей случайное значение из списка:

```
(aclosure ac :attribute "step" :type "sensor" :instance i
  :do (aset i "state" (random-list-element
    (list "detected" "not detected"))))
```

Шаг контроллера состоит в определении его состояния и запуска соотвествующего режима функционирования (ожидание, сушка, пассивное охлаждение) в данном состоянии:

```
(aclosure ac :attribute "step" :type "controller" :instance i
  :match :ap i "state" s (nmatch
    :v s "waiting"
    :exit (update-eval-aclosure ac :av "attribute" "waiting")
    :v s "drying"
    :exit (update-eval-aclosure ac :av "attribute" "drying")
    :v s "cooling"
    :exit (update-eval-aclosure ac :av "attribute" "cooling")))
```

Режим ожидания задается следующей декларацией:

```
(aclosure ac :attribute "waiting" :type "controller" :instance i
  :match :av i (aseq "sensor" "state") "detected"
    :do (aset i :av "state" "drying" :av "local clock" 0))
```

В этом режиме отслеживается срабатывание датчика и переход контроллера в этом случае в состояние сушки с обнулением локального времени.

В режиме сушки, задаваемом декларацией

```
(aclosure ac :attribute "drying" :type "controller" :instance i
  :nmatch
    :v (< (aget i "local clock")
      (aget i "maximum drying time")) T
    :exit (aset i :av "state" "cooling" :av "local clock" 0)
    :av i (aseq "sensor" "state") "not detected"
```

```
    :exit (aset i "state" "waiting")
    :do (aset i "local clock" (+ (aget i "local clock") 1)))
```

сначала выполняется проверка не превышен ли лимит непрерывной сушки. Если лимит превышен, контроллер переходит в режим пассивного охлаждения с обнулением локального времени. В противном случае, проверяется состояние датчика и если он ничего не обнаруживает, то контроллер переходит в состояние ожидания. Если ни одно из выше проверяемых условий не выполнено, то увеличивается время локальных часов на 1 (один такт). Заметим, что в случае перехода в состояние ожидания время не сбрасывается в ноль, так как для этого состояния время локальных часов не учитывается.

Декларация, моделирующая режим пассивного охлаждения, определяется аналогичным образом:

```
(aclosure ac :attribute "cooling" :type "controller" :instance i
  :nmatch
    :v (< (aget i "local clock")
      (aget i "cooling time")) T
    :exit (aset i "state" "waiting")
    :do (aset i "local clock" (+ (aget i "local clock") 1)))
```

Таким образом, мы построили как модель состояний такой системы как сушилка для рук, так и модель функционирования этой системы в терминах онтологии.

11. Родственные работы

Исследования, посвящённые формальному описанию знаний и динамики систем, ведутся в нескольких взаимосвязанных направлениях, включая онтологическое моделирование, языки спецификации динамических и реактивных систем, предметно-ориентированные языки (DSL), а также подходы, основанные на графах знаний и атрибутных вычислениях [18, 20, 24]. Язык ABML находится на пересечении этих направлений, объединяя элементы онтологий, типизированных объектных моделей и механизмов описания поведения.

Онтологические языки и представление знаний. Наиболее распространённым формализмом для представления онтологий является семейство языков, основанных на дескриптивных логиках, прежде всего OWL (Web Ontology Language) [5, 7, 9]. Эти языки

обеспечивают строгую семантику, поддержку логического вывода и широко применяются в задачах семантического веба и интеграции знаний [8, 18]. Однако OWL и родственные ему формализмы ориентированы преимущественно на статическое описание знаний и обладают ограниченными возможностями для моделирования динамики и изменения состояний объектов во времени [2].

Для расширения онтологического подхода в сторону описания поведения разрабатывались различные онтологические модели процессов и событий, включая OWL-S и SOSA/SSN [18, 20, 23, 24]. Эти модели позволяют описывать действия, события и наблюдения, но, как правило, не предоставляют формального механизма исполнения или пошагового моделирования динамических систем. В отличие от них, ABML изначально ориентирован на моделирование дискретной динамики и допускает явное описание шагов функционирования системы.

Языки спецификации динамических и реактивных систем. Значительный пласт родственных работ связан с языками спецификации динамических, реактивных и киберфизических систем [6]. Классическими примерами являются языки временной логики, такие как LTL и CTL [19], а также формализмы на основе автоматов и систем переходов [4]. Эти подходы широко используются для верификации свойств систем, однако они слабо приспособлены для непосредственного описания сложных структур знаний и онтологий.

Языки спецификации, такие как Event-B и TLA+ [11, 16], предлагают строгие математические средства для описания состояний и переходов, но требуют значительных усилий для моделирования предметной области на уровне объектов и атрибутов. В отличие от перечисленных формализмов, ABML ориентирован на знание-центричный подход, в котором онтологическая структура системы и динамика её функционирования описываются в рамках единого атрибутного формализма.

Объектно-ориентированные и атрибутно-ориентированные модели. Многие идеи ABML перекликаются с объектно-ориентированным моделированием и индустриальными языками моделирования, такими как UML и SysML [17]. В этих языках объекты, атрибуты и состояния играют центральную роль, однако формальная семантика большинства их конструкций либо задаётся неявно, либо выходит за рамки стандартов, а средства исполнения моделей, как правило, носят ограниченный или инструментально-зависимый характер (см, например, [3, 15, 22]).

Атрибутно-ориентированные подходы к моделированию рассматривались, в частности, в контексте систем правил и продукционных систем, где вычисление значений атрибутов определяется набором явно заданных зависимостей и условий [12]. В таких системах вычисление значений атрибутов может зависеть от контекста и состояния других объектов. ABML развивает эти идеи, вводя формализованный механизм атрибутных замыканий, который позволяет явно задавать контекст вычисления и связывать его с конкретным экземпляром типа.

Предметно-ориентированные языки и Lisp-подобные системы. Разработка ABML как расширения Common Lisp тесно связана с традицией создания предметно-ориентированных языков (DSL) [13]. Lisp и его диалекты исторически используются для создания языков моделирования и спецификации благодаря мощной макросистеме и однородному синтаксису [1].

Существуют Lisp-ориентированные системы для представления знаний и онтологий, такие как Loom и OCML, которые предоставляют средства описания понятий и отношений. Однако они, как правило, либо ориентированы на логический вывод, либо не поддерживают явное моделирование дискретной динамики. ABML отличается тем, что сочетает декларативное описание структуры знаний с процедурным описанием поведения.

Графы знаний и вычисления на графах. В последние годы активно развиваются подходы, основанные на графах знаний, где информация представляется в виде вершин и дуг с семантической интерпретацией [10]. Графы знаний используются в интеллектуальных системах, анализе данных и моделировании сложных взаимосвязей. В этом контексте модель ABML может интерпретироваться как граф знаний, в котором объекты и значения образуют вершины, а атрибуты — помеченные рёбра.

Отличительной особенностью ABML является то, что вычисления и изменения состояния системы формализуются как преобразования такого графа знаний во времени. Это сближает ABML с подходами, основанными на трансформациях графов [14], но при этом сохраняет удобство атрибутного и типизированного моделирования.

Итоги сравнения. Таким образом, существующие родственные работы либо фокусируются на статическом представлении знаний, либо на формальной спецификации динамики без явной онтологической структуры. Язык ABML занимает промежуточную позицию,

предлагая унифицированный формализм для онтологического моделирования и описания дискретной динамики систем. Его атрибутно-ориентированный подход и механизм атрибутных замыканий позволяют выразить широкий класс моделей, что отличает его от большинства существующих решений.

12. Заключение

В работе представлен язык ABML, предназначенный для спецификации дискретных динамических систем, ориентированных на знания, структурированные в онтологиях. Язык объединяет онтологическое моделирование и формальное описание поведения систем в рамках единого, компактного и выразительного формализма.

Основным достоинством ABML является минимальный, но универсальный концептуальный базис, включающий объекты, атрибуты и типы объектов. Разделение объектов на изменяемые и константные позволяет явно задавать семантику изменений и облегчает моделирование эволюции состояний системы. Атрибутно-ориентированная типизация обеспечивает гибкий механизм задания ограничений и классификации объектов, соответствующий онтологическому подходу.

Развитые средства работы с атрибутами, включая вложенный доступ, массовое обновление и интерпретацию атрибутов как функций, делают язык удобным для описания сложных структур знаний. Механизмы сопоставления с образцом позволяют компактно и наглядно формулировать правила функционирования систем, а также реализовывать условные переходы между состояниями.

Ключевым элементом языка является механизм атрибутных замыканий, который обеспечивает контекстно-зависимое вычисление атрибутов и служит основой для моделирования дискретной динамики. Использование атрибутных замыканий позволяет рассматривать поведение системы как последовательность вычислений атрибутов, что хорошо согласуется с онтологической интерпретацией модели в виде графа знаний.

Пример моделирования сушилки для рук наглядно демонстрирует практическую применимость ABML. В рамках одного языка удалось задать онтологию системы, ее начальное состояние и правила функционирования, описывающие поведение сенсора и контроллера во времени. Это подтверждает, что ABML может использоваться для прототипирования и анализа поведения реальных технических, информационных и программных систем.

В перспективе язык ABML может быть расширен средствами верификации, анализа свойств моделей и интеграции с внешними онтологическими и логическими инструментами. Такой подход делает ABML перспективным средством для исследования и разработки интеллектуальных систем, основанных на знаниях и онтологиях.

Список литературы

1. Alneami A., Mc Kevitt P. On Lisp: Advanced Techniques for Common Lisp. Paul Graham // Artificial Intelligence Review. — 1999. — Vol. 13, no. 3. — P. 239–241.
2. Baader F., Horrocks I., Sattler U. Description logics as ontology languages for the semantic web // Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday. — Springer, 2005. — P. 228–248.
3. Clark T., Warmer J. Object modeling with the OCL: the rationale behind the Object Constraint Language. — Springer, 2002.
4. Clarke E. M. J., Grumberg, O., Peled, DA: Model Checking. — 1999.
5. (Comet-) atomic 2020: On symbolic and neural commonsense knowledge graphs / Hwang J. D., Bhagavatula C., Le Bras R., Da J., Sakaguchi K., Bosselut A., and Choi Y. // Proceedings of the AAAI conference on artificial intelligence. — 2021. — Vol. 35. — P. 6384–6392.
6. Compositional model checking of consensus protocols via interaction-preserving abstraction / Gu X., Cao W., Zhu Y., Song X., Huang Y., and Ma X. // 2022 41st International Symposium on Reliable Distributed Systems (SRDS) / IEEE. — 2022. — P. 82–93.
7. Consortium W. W. W. et al. OWL 2 web ontology language document overview. — 2012.
8. A derived information framework for a dynamic knowledge graph and its application to smart cities / Bai J., Lee K. F., Hofmeister M., Mosbach S., Akroyd J., and Kraft M. // Future Generation Computer Systems. — 2024. — Vol. 152. — P. 112–126.
9. Design ontology supporting model-based systems engineering formalisms / Lu J., Ma J., Zheng X., Wang G., Li H., and Kiritsis D. // IEEE Systems Journal. — 2021. — Vol. 16, no. 4. — P. 5465–5476.
10. Ehrlinger L., Wöß W. Towards a definition of knowledge graphs. // SEMANTiCS (Posters, Demos, SuCCESS). — 2016. — Vol. 48, no. 1-4. — P. 2.

11. Farrell M., Monahan R., Power J. F. Building specifications in the Event-B institution // Logical Methods in Computer Science. — 2022. — Vol. 18.
12. Forgy C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem // Readings in artificial intelligence and databases. — Elsevier, 1989. — P. 547–559.
13. Fowler M. Domain-specific languages. — Pearson Education, 2010.
14. Fundamentals of algebraic graph transformation / Ehrig H., Ehrig K., Prange U., and Taentzer G. — Springer, 2006.
15. Kleppe A. G., Warmer J. B., Bast W. MDA explained: the model driven architecture: practice and promise. — Addison-Wesley Professional, 2003.
16. Lamport L. Specifying systems. — Addison-Wesley Boston, 2002. — Vol. 388.
17. Merging OMG standards in a general modeling, transformation, and simulation framework. / Schneider V., Yupatova A., Dulz W., and German R. // SimuTools. — 2015. — P. 299–301.
18. Ontologies in digital twins: A systematic literature review / Karabulut E., Pileggi S. F., Groth P., and Degeler V. // Future Generation Computer Systems. — 2024. — Vol. 153. — P. 442–456.
19. Pnueli A. The temporal logic of programs // 18th annual symposium on foundations of computer science (sfcs 1977) / ieee. — 1977. — P. 46–57.
20. Representing Time-Continuous Behavior of Cyber-Physical Systems in Knowledge Graphs / Gill M. S., Jeleniewski T., Gehlhoff F., and Fay A. // arXiv preprint arXiv:2506.13773. — 2025.
21. Rhodes C. SBCL: A sanely-bootstrappable Common Lisp // Workshop on Self-sustaining Systems / Springer. — 2008. — P. 74–86.
22. Rumpe B. Agile modeling with UML: Code generation, testing, refactoring. — Springer, 2017.
23. The SSN ontology of the W3C semantic sensor network incubator group / Compton M., Barnaghi P., Bermudez L., Garcia-Castro R., Corcho O., Cox S., Graybeal J., Hauswirth M., Henson C., Herzog A., et al. // Journal of Web Semantics. — 2012. — Vol. 17. — P. 25–32.
24. Наместников А.М. Применение онтологического подхода в задаче генерации событийных данных с помощью имитационных моделей // Онтология проектирования. — 2023. — Vol. 13, no. 2 (48). — P. 243–253.