MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 33, NO. 2, 2026
journal homepage: www.mais-journal.ru

THEORY OF SOFTWARE

Building Operational Semantics of Programming Languages as Executable Ontological Models of Programs

I. S. Anureev[1], N. O. Garanina[1], D. A. Kondratyev[1], S. P. Gorlatch[2]

DOI: 10.18255/1818-1015-2026-2-122-149

[1]A.P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk, Russia
[2]University of Muenster, Muenster, Germany

MSC2020: 68Q55, 68N15
Research article
Full text in Russian

Received April 28, 2026
Revised May 21, 2026
Accepted May 28, 2026

A critical factor in ensuring the quality of software written in current and next-generation programming languages is the ability to rely on a formal operational semantics of the language. This gives developers a systematic, well-founded way to address reliability, performance, and security in the target software. We introduce a new formal framework where operational semantics can be developed and used not as a traditional non-executable abstract system, but as runnable code. That brings several advantages: easier modification, the ability to run across different execution environments, testing and debugging, version control, and more. We illustrate our approach using a practically relevant subset of the widely used C programming language. The strengths of our semantics framework are demonstrated by comparing it against the most popular current approaches across a set of practically relevant criteria. The paper presents the Attribute-Based Modeling Language (ABML) — a new domain-specific language designed for ontological modeling of programs and for defining their operational semantics in executable form. We describe the proposed method for building executable operational semantics with ABML. A worked example shows how our approach applies to a practically relevant subset of C. We then build an enriched ontology of C by adding semantic attributes, define algorithms for computing those attributes in ABML, and discuss how the enriched ontology influences the development of operational semantics. A detailed review of related work lets us compare our approach to state-of-the-art frameworks in terms of meeting current requirements for operational semantics development.

Keywords: programming languages; programming language ontology; ontological model of a programming language; operational semantics of a programming language

INFORMATION ABOUT THE AUTHORS

Anureev, Igor S. (corresponding author)	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com PhD, Senior Researcher
Garanina, Natalia O.	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su PhD, Senior Researcher
Kondratyev, Dmitry A.	ORCID iD: 0000-0002-9387-6735. E-mail: apple-66@mail.ru PhD, Researcher
Gorlatch, Sergei P.	ORCID iD: 0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de PhD, Professor

For citation: I. S. Anureev, N. O. Garanina, D. A. Kondratyev, and S. P. Gorlatch, “Building operational semantics of programming languages as executable ontological models of programs”, *Modeling and Analysis of Information Systems*, vol. 33, no. 2, pp. 122–149, 2026. DOI: 10.18255/1818-1015-2026-2-122-149.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 33, № 2, 2026
сайт журнала: www.mais-journal.ru

THEORY OF SOFTWARE

Построение операционной семантики языков программирования в виде исполняемых онтологических моделей программ

И. С. Ануреев[1], Н. О. Гаранина[1], Д. А. Кондратьев[1], С. П. Горлач[2]

DOI: 10.18255/1818-1015-2026-2-122-149

[1]Институт систем информатики им. А.П. Ершова СО РАН, Новосибирск, Россия
[2]Мюнстерский университет, Мюнстер, Германия

УДК 004.423.42+004.434
Научная статья
Полный текст на русском языке

Получена 28 апреля 2026 г.
После доработки 21 мая 2026 г.
Принята к публикации 28 мая 2026 г.

Важнейший аспект обеспечения качества программ, написанных на существующих и перспективных языках программирования, — это возможность опираться на формальную операционную семантику языка. Такой подход позволяет разработчикам системно и обоснованно решать вопросы надёжности, производительности и безопасности целевого программного обеспечения. Мы предлагаем новую формальную среду, в которой операционную семантику можно разрабатывать и использовать не как традиционную невыполнимую абстрактную систему, а как исполняемый программный код. Это даёт множество преимуществ: упрощение модификации, возможность выполнения в разных средах исполнения, тестирование и отладку, управление версиями и так далее. Мы иллюстрируем наш подход на примере практически значимого фрагмента широко используемого языка программирования C. Преимущества нашей среды для построения семантики демонстрируются путем сравнения с наиболее популярными современными подходами по набору практически значимых требований. В работе представлен язык моделирования на основе атрибутов (ABML) — новый предметно-ориентированный язык, предназначенный для онтологического моделирования программ и определения их операционной семантики в исполняемой форме. Описан предлагаемый подход к построению выполняемой операционной семантики с использованием ABML. Мы приводим также модельный пример, демонстрирующий применение нашего подхода к практически значимому фрагменту широко используемого языка программирования C. Далее строится обогащённая онтология языка C путём добавления семантических атрибутов, задаются алгоритмы вычисления этих атрибутов в ABML и обсуждается, как обогащённая онтология влияет на разработку операционной семантики. Подробный обзор связанных работ позволяет сравнить предлагаемый подход с современными фреймворками с точки зрения соответствия актуальным требованиям к разработке операционной семантики.

Ключевые слова: языки программирования; онтология языка программирования; онтологическая модель языка программирования; операционная семантика языка программирования

ИНФОРМАЦИЯ ОБ АВТОРАХ

Ануреев, Игорь Сергеевич (автор для корреспонденции)	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com Канд. физ.-мат. наук, старший научный сотрудник
Гаранина, Наталья Олеговна	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Канд. физ.-мат. наук, старший научный сотрудник
Кондратьев, Дмитрий Александрович	ORCID iD: 0000-0002-9387-6735. E-mail: apple-66@mail.ru Канд. физ.-мат. наук, научный сотрудник
Горлач, Сергей Петрович	ORCID iD: 0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de PhD, профессор

Для цитирования: I. S. Anureev, N. O. Garanina, D. A. Kondratyev, and S. P. Gorlatch, “Building operational semantics of programming languages as executable ontological models of programs”, *Modeling and Analysis of Information Systems*, vol. 33, no. 2, pp. 122–149, 2026. DOI: 10.18255/1818-1015-2026-2-122-149.

Введение

Прогресс в области информационных и компьютерных технологий в значительной степени зависит от разработки и внедрения новых языков программирования: как универсальных языков, таких как Python, Rust и других, так и предметно-ориентированных языков для развивающихся прикладных областей, включая искусственный интеллект, высокопроизводительные вычисления и т. д.

Ключевым аспектом обеспечения качества программного обеспечения, использующего существующие и появляющиеся языки программирования, является возможность опираться на хорошо определённую формальную семантику языка. Это позволяет разработчикам систематически решать вопросы надёжности, производительности и безопасности целевого программного обеспечения на строгой формальной основе.

Среди различных типов семантики языков программирования *операционная семантика* особенно широко используется на практике благодаря своему интуитивному базису, который традиционно понимается как абстрактная машина, выполняющая языковые конструкции в виде переходов между её состояниями.

Настоящая статья вносит новый вклад в определение и реализацию операционной семантики языков программирования. Большинство существующих подходов (мы приводим подробный обзор связанных работ в разделе 5) рассматривают операционную семантику как декларативную спецификацию, записанную в формальной системе, которая, однако, не может быть выполнена или отлажена как обычная программа. Наш вклад заключается в новой формальной основе, в рамках которой операционная семантика может разрабатываться и анализироваться как исполняемая программа, что даёт значительные преимущества. В частности, предлагаемый подход обеспечивает более легкую модификацию семантики, её выполнение в различных средах выполнения, тестирование и отладку, управление версиями семантики, специфичными для разных задач, унифицированную нотацию для нескольких языков, бесшовную композицию семантики для многоязычных программ, повторное использование существующей семантики и модульную разработку с изолированной отладкой отдельных компонентов. Мы формализуем эти и другие полезные свойства в виде набора из 19 требований и показываем, что ни один существующий семантический фреймворк не удовлетворяет всем этим требованиям. Цель нашей работы — восполнить этот пробел.

Оставшаяся часть статьи организована следующим образом. В разделе 1 представлен язык моделирования на основе атрибутов (ABML) — новый предметно-ориентированный язык, предназначенный для онтологического моделирования программ и определения их операционной семантики в исполняемой форме. В разделе 2 описан наш подход к построению выполняемой операционной семантики с использованием ABML. Раздел 3 содержит модельный пример, демонстрирующий применение предложенного подхода к практически значимому фрагменту широко используемого языка программирования C. В разделе 4 показано, каким образом строится обогащённая онтология языка C путем добавления семантических атрибутов, задаются алгоритмы вычисления этих атрибутов в ABML и обсуждается, как обогащённая онтология влияет на разработку операционной семантики. Раздел 5 представляет подробный обзор связанных работ и сравнение предлагаемого подхода с современными фреймворками с точки зрения соответствия актуальным требованиям к разработке операционной семантики. Наконец, в заключении суммируются основные результаты работы и описываются направления будущих исследований.

1. Язык моделирования на основе атрибутов (ABML)

Новизна предлагаемого подхода заключается в том, что для определения операционной семантики языка программирования мы концентрируемся не на самом языке, а на его онтологической

модели — структурированном представлении, охватывающем все синтаксические и семантические конструкции языка.

Для поддержки онтологического моделирования языков программирования и формальной спецификации их операционной семантики используется ранее разработанный нами предметно-ориентированный язык *ABML (Attribute-Based Modeling Language)* [1]. Он представляет собой лексическое расширение SBCL (Steel Bank Common Lisp) [2] — популярного диалекта Common Lisp. Выбор языка из семейства Common Lisp в качестве основы обусловлен его развитыми возможностями для встраивания и инкапсуляции вновь создаваемых предметно-ориентированных языков.

ABML является легковесным расширением SBCL, вводящим лишь небольшой набор дополнительных функций. Хотя большинство этих дополнений технически можно рассматривать как макросы, для удобства мы называем их «функциями». Язык ABML изначально разрабатывался как средство моделирования дискретных динамических систем (в дальнейшем именуемых *системами*), включая те, которые описывают операционную семантику. В данной статье представлена новая доработанная версия ABML по сравнению с [1]. В частности, использовавшееся ранее понятие *атрибутного замыкания* заменено на понятие *аспекта*, более точно отражающее назначение языка.

ABML базируется на минимальной концептуальной основе, включающей семь основных понятий: *объекты*, *атрибуты*, *объектные типы*, *аспекты*, *агенты*, *окружения* и *стадии*. Ниже приводится их неформальные определения.

- *Объекты* служат моделями элементов и подсистем дискретной динамической системы. Каждый объект характеризуется набором атрибутов с соответствующими значениями. Выделяются два вида объектов: *мутабельные* и *константные*. Изменение атрибутов мутабельного объекта (добавление, удаление или модификация атрибута) не нарушает его идентичности, тогда как любое изменение атрибута константного объекта приводит к созданию нового константного объекта.
- *Атрибуты* описывают свойства и отношения объектов. Каждый атрибут обладает *именем*, *значением* и *типом*, который ограничивает множество допустимых значений.
- *Объектные типы* выделяют классы объектов с одинаковыми ограничениями на значения атрибутов.
- *Аспекты* определяют способы работы с объектами. *Выполнение аспекта* представляет собой осуществление работы с объектом в соответствии с заданным аспектом. Момент начала выполнения аспекта называется *запуском аспекта*.
- *Агенты* выступают в роли исполнителей, реализующих способы работы с объектами, закреплённые за аспектами. Виды агентов задаются объектными типами. Следовательно, сами агенты являются объектами, а их знания выражаются через наборы атрибутов и их значений применительно к объектам.
- *Окружения* моделируют среду, в которой действуют агенты. Среда хранит глобальные параметры системы и используется как разделяемый ресурс для организации взаимодействия между агентами. Типы окружений, как и типы агентов, представляются объектными типами.
- *Стадии* служат для моделирования отдельных этапов выполнения аспекта.

1.1. Объектные типы

Тип для мутабельных объектов (mutable type, далее — *мутабельный тип*) t'' определяется как (`mot` $ad_1 \ \ldots \ ad_r$), где объявления атрибутов ad_j задают ограничения на значения атрибутов мутабельных объектов, принадлежащих данному типу.

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

В языке ABML выделяются четыре типа объявлений атрибутов:

1. `:av` a v — объявляет атрибут a' со значением v'. Значение данного атрибута для любого экземпляра типа t'' всегда должно быть равно v'.

2. `:at` a t — объявляет атрибут a' с типом t'. Значение этого атрибута для любого экземпляра типа t'' всегда должно принадлежать типу t'.
3. `:atv` a t v — объявляет атрибут a' с типом t' и значением по умолчанию v'. Данное объявление накладывает ограничение, описанное в пункте 2, и, кроме того, присваивает атрибуту a' значение по умолчанию v' во всех порождаемых экземплярах типа t'' (если при генерации не указано иное).
4. `:amap` t_1 t_2 — объявляет множество значений t_1' в качестве атрибутов, значения которых берутся из типа t_2'. При этом значение любого атрибута a в экземпляре типа t'' должно принадлежать типу t_2' при условии, что a является элементом множества t_1'.

Тип для константных объектов (constant object type, далее *константный тип*) определяется как (`cot` ad_1 ... ad_r), где объявления атрибутов ad_j задают ограничения на значения атрибутов константных объектов, принадлежащих этому типу, по аналогии с мутабельными типами.

Для краткости `mot` и `cot` используются как сокращения для определений типов объектов (`mot`) и (`cot`) соответственно, то есть как типы для объектов, на атрибуты которых не налагается никаких ограничений.

В ABML новые типы могут быть определены с помощью конструкции (`typedef` n t), которая объявляет тип с именем n как синоним для типа t. Для удобства используются сокращённые обозначения (`mot` n ad_1 ... ad_r) и (`cot` n ad_1 ... ad_r) вместо эквивалентных определений (`typedef` n (`mot` ad_1 ... ad_r)) и (`typedef` n (`cot` ad_1 ... ad_r)).

1.2. Объекты

Порождение объектов в модели системы осуществляется исключительно посредством специальных функций, создающих экземпляры типов объектов.

Для мутабельных объектов функция порождения экземпляра имеет вид (`mo` t ad_1 ... ad_r), где допускаются только объявления атрибутов вида `:av` a v. Данная функция создает новый мутабельный объект o типа t и присваивает атрибуты со значениями согласно объявлениям ad_j. Объект o наследует также все атрибуты со значениями по умолчанию, определенные в типе t.

Порождение экземпляров типов подчиняется *принципу открытого мира*, согласно которому экземпляры могут содержать атрибуты, явно не объявленные в данном типе; значения таких необъявленных атрибутов ничем не ограничены.

В ABML предусмотрены предопределенные функции для работы с мутабельными объектами и мутабельными типами:

- (`imax` t) — возвращает количество экземпляров типа t;
- (`otype` o) — возвращает тип объекта o;
- (`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 отлично от `nil`.

Для константных объектов функция порождения экземпляра имеет вид (`co` ad_1 ... ad_r). Всё описанное выше для мутабельных объектов применимо и к константным объектам.

Для удобства используются сокращённые обозначения (`mo` ad_1 ... ad_r) и (`co` ad_1 ... ad_r) вместо эквивалентных определений (`mo mot` ad_1 ... ad_r) и (`co cot` ad_1 ... ad_r).

1.3. Атрибуты

Для расширения объектного типа t новыми атрибутами в ABML предусмотрена функция (`att` t ad_1 ... ad_r), которая добавляет ограничения $ad_1, \ldots, ad_r$ к типу t.

Для получения значения атрибута a из объекта o в ABML используется конструкция (`aget` o a). Для присваивания атрибуту a в объекте o значения v применяется (`aset` o a v).

Указанные функции работают также со списками (`listt`), где индексы трактуются как атрибуты.

Обобщения этих функций обеспечивают доступ к атрибутам на произвольном уровне вложенности с использованием синтаксиса: (`aget` o (`aseq` a_1 ... a_n)) (или (`aget` o a_1 ... a_n)) для получения значения атрибута и (`aset` o (`aseq` a_1 ... a_n) v) (или (`aset` o a_1 ... a_n v)) для его присваивания. Если список атрибутов не задан явно, используется форма (`aseql` e) вместо (`aseq` a_1 ... a_n); в этом случае список атрибутов вычисляется как значение выражения e.

Существует также сокращённая запись (`aset` o `:av` a_1 v_1 `:av` ... `:av` a_n v_n), эквивалентная вложенной форме (`aset` (... (`aset` o a_1 v_1) ...) a_n v_n) и представляющая последовательные применения функции `aset`.

Для работы с атрибутами и выражениями в ABML используются также сопоставители вида (`match` c_1 ... c_r) и (`nmatch` c_1 ... c_r), состоящие из последовательности сопоставляющих клозов c_j, которые реализуют чередование сопоставления с образцом и действий, выполняемых при успешном или неуспешном совпадении.

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

Сопоставляющие клозы подразделяются на три категории: *клозы для атрибутов*, *клозы для выражений* и *клозы для действий*, которые описываются ниже.

Клозы для атрибутов служат для сопоставления значений атрибутов. В ABML поддерживаются три вида таких клозов:

1. `:av` e a v сопоставляется (т. е. сопоставление успешно), если e' является объектом и его атрибут a' имеет значение v'.
2. `:at` e a t сопоставляется, если e' является объектом и значение его атрибута a' принадлежит типу t'.
3. `:ap` e a p сопоставляется, если e' является объектом. При успешном сопоставлении переменной p, называемой *сопоставляемой переменной*, присваивается значение атрибута a' этого объекта.

Выражения (`aseq` a_1 ... a_n) и (`aseql` e) также могут использоваться на месте `a`.

Клозы выражений служат для сопоставления значений выражений. В ABML поддерживаются три вида таких клозов:

1. `:v` e v сопоставляется, если e' равно v'.
2. `:t` e t сопоставляется, если e' принадлежит типу t'.
3. `:p` e p сопоставляется всегда; при этом сопоставляемой переменной p присваивается значение выражения e'.

Клозы действий определяют операции, выполняемые при успешном или неуспешном сопоставлении. В ABML поддерживаются два вида таких клозов: `:do` e_1 ... e_m и `:exit` e_1 ... e_m. Каждый из них последовательно вычисляет выражения e_j и возвращает значение последнего из них, то есть ведёт себя аналогично функции `progn` языка Lisp. Различие заключается в том, что в первом случае выполнение сопоставителя продолжается на следующем клозе, а во втором — завершается. Данные клозы могут использовать в своих выражениях ранее связанные сопоставляемые переменные. При выполнении этих клозов сопоставление всегда считается успешным.

Сопоставитель `match` выполняет клозы до тех пор, пока они сопоставляются, пропуская клозы `:exit`. Как только встречается клоз, для которого сопоставление не выполняется, управление передается следующему за ним клозу `:exit` (если таковой имеется); в противном случае сопоставитель завершает работу.

Сопоставитель `nmatch` действует противоположным образом: он выполняет клозы до тех пор, пока они *не* сопоставляются, также пропуская клозы `:exit`. Как только встречается клоз, для которого сопоставление происходит, выполняется следующий клоз `:exit` (при его наличии), после чего работа завершается.

Сопоставители допускают вложенность: в качестве клоза в одном сопоставителе может использоваться другой сопоставитель. Сопоставитель `match` считается неуспешным только в том случае, если был найден несопоставляемый клоз, за которым не следует `:exit`-клоз. Напротив, сопоставитель `nmatch` считается неуспешным только в том случае, если был найден сопоставляемый клоз, за которым не следует `:exit`-клоз.

1.4. Аспекты и аспектные контексты

Контекст, в котором выполняется аспект (далее — *аспектный контекст*), задается константным объектом, содержащим следующие обязательные атрибуты:

- `"aspect"` — имя аспекта;
- `"instance"` — объект, к которому применяется аспект;
- `"agent"` — агент, используемый аспектом для работы с объектом;
- `"env"` — окружение, в котором функционируют агенты.
- `"stage"` — имя стадии выполнения аспекта.

Данный объект может также включать и другие атрибуты.

Поскольку аспектный контекст содержит имя аспекта, далее словосочетания «выполнение аспекта в контексте» и «выполнение аспектного контекста» будут рассматриваться как синонимы; аналогичным образом синонимичными считаются выражения «запуск аспекта в контексте» и «запуск аспектного контекста».

Функция `(clear-acontext` *ac*`)` удаляет все атрибуты аспектного контекста *ac*, кроме обязательных.

Действия, выполняемые аспектным контекстом, задаются *декларацией аспекта* (также используется термин *аспектная декларация*) вида

```
(aspect a :type t :stage st :context ac :instance i :agent ag :env env
   :do e1 ... en),
```

где *ac* — переменная, представляющая контекст; *i*, *ag*, *env* — переменные, представляющие значения соответствующих атрибутов *ac*; *a*, *st* и *t* — имя аспекта (в *ac*), имя стадии (в *ac*) и тип объекта *i* соответственно. Часть декларации до `:do` называется *аспектным префиксом*. Префикс идентифицирует контекст, в котором запускается аспект. Выражения после `:do` образуют *аспектное тело*, которое выполняется при запуске аспектного контекста. Эти выражения могут зависеть от переменных префикса. Если переменные *i*, *ag*, *env* не используются в выражениях аспектного тела, соответствующие части префикса могут опускаться.

В аспектный префикс могут также добавляться сопоставляющие клозы вида `:ap` *e a p* и `:p` *e p*, чтобы сохранять значения промежуточных вычислений в сопоставляемых переменных. Эти переменные могут входить в выражения аспектного тела. Для сопоставляющего клоза вида `:ap` *ac a p* используется сокращение `:`*a p*.

Результат запуска *ac* определяется как результат вычисления `(progn` e_1 ... e_n `(acontext-next` *ac*`))`. Функция `acontext-next` определяет, какой следующий аспектный контекст запускается после вычисления `progn`. Фактически она задает продолжение (continuation) подобно тому, как это делается в функциональных языках программирования. Её реализация оставляется на усмотрение пользователя.

Для моделирования ситуации, когда `next-acontext` ничего не делает, и, таким образом, ABML-программа завершает свою работу, используется значение типа `"stop next acontext"` из атрибута `"value"` агента, связанного с аспектным контекстом *ac*. Этот тип определяется следующим образом:

```
(cot "stop next acontext" :at "type" string :at "acontext" (cot))
```

Атрибут `"type"` хранит тип остановки аспекта (например, `"error"`), а атрибут `"acontext"` хранит аспектный контекст, на котором произошла остановка.

Функция (`eval-acontext` *ac*) запускает аспектный контекст *ac*.

1.5. Агенты и окружения

Агенты и окружения могут быть различных видов, в связи с чем они определяются наборами типов языка ABML. В случае, когда для каждой из этих сущностей существует только один тип, он называется `"agent"` для агентов и `"env"` для окружения соответственно.

Типы, используемые для задания агентов и окружений, могут различаться для разных моделируемых систем, однако обладают общими характеристиками.

Тип для агентов содержит обязательный атрибут `"value"`, который хранит последнее значение, вычисленное агентом. Тип этого значения зависит от конкретной моделируемой системы. Поскольку данный атрибут часто используется в аспектных контекстах, он может быть добавлен в аспектный префикс в форме `:value` *v*; переменная *v* в таком случае может применяться в выражениях аспектного тела.

Тип для окружений включает обязательные атрибуты `"agents"` и `"acontexts"`. Первый атрибут, имеющий тип (`listt "agent"`), хранит список всех активных агентов. Второй атрибут, тип которого определяется как (`cot :amap "agent" (listt cot)`), моделирует для каждого агента стек, содержащий аспектные контексты (в виде константных объектов), выполнение которых отложено.

В языке ABML предусмотрены следующие функции для работы со стеками:

- (`push-acontext` *ac*) — добавляет аспектный контекст *ac* в стек соответствующего агента;
- (`pop-acontext` *ac*) — удаляет аспектный контекст *ac* из стека соответствующего агента, возвращая его в качестве значения функции;
- (`peek-acontext` *ac*) — считывает аспектный контекст *ac* из стека соответствующего агента, возвращая его в качестве значения.

Также используются сокращения для часто встречающихся комбинаций:

- (`update-push-acontext` *ac* ...) как сокращение для (`push-acontext` (`aset` *ac* ...));
- (`clear-update-push-acontext` *ac* ...) как сокращение для (`update-push-acontext` (`clear-acontext` *ac*)).

1.6. Стадии

Стадии аспекта могут обладать параметрами, которые представляются в виде дополнительных атрибутов аспектного контекста.

Первая стадия выполнения аспекта всегда имеет имя `nil`. Для данной стадии указание `:aspect nil` в префиксе аспектной декларации может опускаться.

Все введенные выше обозначения и сокращения распространяются также на функции `aget`, `aset` и их модификации при условии, что они используются в выражениях аспектного тела. Например, запись (`aset` *ac* `:agent` *ag* `:stage` *st* `:"evaluate 1"` e_1) является сокращённой формой для (`aset` *ac* `:av "agent"` *ag* `:av "stage"` *st* `:av "evaluate 1"` e_1).

2. Использование ABML для построения операционной семантики языков программирования

Основная идея предлагаемого подхода состоит в том, что вместо построения операционной семантики самого языка строится операционная семантика его модели с использованием единого языка моделирования, применимого к любому языку программирования. Таким унифицированным языком выступает ABML, определенный в предыдущем разделе.

Предлагаемый подход к построению формальной операционной семантики некоторого языка программирования (обозначим его через PL) состоит из следующих пяти этапов:

1. Построение *онтологии* (или *онтологической модели*) языка PL как набора объявлений типов в ABML. Каждый вид конструкций PL представляется типом, объявленным в этом наборе.

2. Разработка алгоритма трансляции, преобразующего конструкции языка PL (программы, операторы, выражения, декларации и т. п.) в экземпляры (*онтологические модели*) соответствующих типов на языке ABML.
3. Реализация транслятора, выполняющего указанный алгоритм.
4. Определение типов для агентов и окружений. Агенты отвечают за выполнение моделей конструкций языка PL, а окружение задает общую среду выполнения для всех агентов.
5. Определение операционной семантики моделей конструкций PL в виде семейства деклараций контекстов для аспекта `"opsem"`, который реализует операционную семантику моделей конструкций языка PL. Элементы этого семейства индексируются типами конструкций PL из этапа 1 и аспектными стадиями, определяющими разбиение выполнения модели конструкции соответствующего типа на отдельные шаги.

3. Пример: операционная семантика для фрагмента C

В качестве иллюстративного практического примера применим предложенный подход к построению операционной семантики небольшого фрагмента широко используемого языка программирования C. Данный фрагмент (в дальнейшем будем называть его языком ФР) включает только те конструкции, которые необходимы для представления следующей программы на C:

```
{
  int x = 0;
  int y;
  int *p = &x;
  if (*p == 0) y = 2;
}
```

Несмотря на простоту языка ФР, программы на нём, с одной стороны, достаточно часто встречаются на практике, а с другой — обладают необходимой выразительностью для демонстрации общности и мощности нашего подхода, поскольку охватывают многие ключевые элементы языка C.

3.1. Онтология языка ФР

Следуя этапу 1 описанного выше подхода, начнём с построения онтологии языка ФР.

Во-первых, определим типы для основных сущностей языка ФР: имен, переменных, констант и типов данных.

```
(typedef "name" string)
(typedef "variable" "name")
(typedef "constant" int)
(typedef "type" (uniont "int type" "pointer type"))
(typedef "int type" (enumt "int"))
(mot "pointer type1" :at 1 "type")
```

Таким образом, для языка ФР приняты следующие соглашения.

- Целочисленные константы моделируются объектом `"constant"` и являются значениями базового типа `int`.
- Объект `"int type"` содержит единственный экземпляр `"int"`.
- Тип `"pointer type1"` описывает указатели и включает атрибут 1, задающий тип значения, на которое производится ссылка. Начиная с этого момента, будем придерживаться соглашения, согласно которому натуральные числа в именах типов, представляющих программные конструкции, обозначают позиции соответствующих аргументов внутри этих конструкций.
- Имена программных объектов представлены экземплярами `"name"`, которые могут быть только строками.

- Переменные моделируются как экземпляры "variable", задаваемые именами.

Во-вторых, определим типы для выражений языка ФР (онтологию выражений):

```
(mot "&1" :at 1 "variable")
(mot "*1" :at 1 "variable")
(mot "1=2" :at 1 "variable" :at 2 "expression")
(mot "1==2" :at 1 "expression" :at 2 "expression")
(mot "expression" :union ("variable" "constant" "&1" "*1" "1=2"
   "1==2"))
```

Такое определение онтологии гарантирует, что моделируются только допустимые конструкции языка ФР. Например, левая часть присваивания (атрибут 1) ограничена переменными, поскольку в исходной программе на C в этой позиции используются исключительно переменные. Данный пример демонстрирует, что предлагаемый подход естественным образом поддерживает инкрементальное моделирование: операционная семантика может разрабатываться лишь для конструкций, присутствующих в заданном наборе целевых программ, и при необходимости расширяться в дальнейшем.

Затем определим типы для операторов языка ФР (онтологию операторов):

```
(mot "1;" :at "expression" "expression")
(mot "{1}" :at 1 (listt "statement"))
(mot "if12" :at 1 "expression" :at 2 "statement")
(mot "variable declaration" :at "type" "type" :at "name" "name" :at
   "initializer" "expression")
(typedef "statement" (uniont "1;" "{1}" "if12" "variable
   declaration"))
```

Следует отметить, что тип "if12", соответствующий условному оператору `if`, не включает ветвь `else`, поскольку в рассматриваемой программе на C она отсутствует. Это подчёркивает гибкость подхода: операционная семантика может быть точно настроена на конструкции, реально встречающиеся в целевом коде.

Наконец, определим типы для семантических конструкций, используемых в ФР:

```
(mot "location")
(typedef "c value" (uniont "constant" "location"))
```

Тип "location" моделирует ячейки памяти, а тип "c value" — значения, возвращаемые выражениями языка ФР. Последний тип будет использоваться в качестве типа атрибута "value" агентов.

3.2. Трансляция конструкций языка ФР в их модели

На этапе 2 разрабатывается алгоритм трансляции конструкций языка ФР в их онтологические модели. Данный этап не представляет сложности, поскольку конструкции ФР напрямую отображаются в соответствующие модели. Например, онтологическая модель приведенной выше программы на C имеет следующий вид:

```
(mo "{1}" :av 1 (list
  (mo "variable declaration" :av "type" "int" :av "name" "x" :av
     "initializer" 0)
  (mo "variable declaration" :av "type" "int" :av "name" "y")
  (mo "variable declaration" :av "type" (mo "pointer type1" :av 1
     "int")
    :av "name" "p" :av "initializer" (mo "&1" :av 1 "x"))
  (mo "if12" :av 1 (mo "1==2" :av 1 (mo "*1" :av 1 "p") :av 2 0)
    :av 2 (mo "1;" :av 1 (mo "1=2" :av 1 "y" :av 2 2)))))
```

Этот пример демонстрирует взаимно однозначное соответствие между конструкциями языка ФР и их моделями, благодаря чему алгоритм трансляции сводится к простому структурному преобразованию.

На этапе 3 выполняется реализация указанного алгоритма. Для разбора кода языка ФР может использоваться любой стандартный генератор парсеров, например Xtext[1], ANTLR[2] или аналогичный инструмент. Затем осуществляется рекурсивный нисходящий обход полученного абстрактного синтаксического дерева (AST) с целью генерации соответствующих моделей.

3.3. Определение среды и агентов

На этапе 4 определяются типы `"env"` и `"agent"`.

Тип `"env"` задается следующим образом:

```
(mot "env"
  :at "init construct" (uniont "expression" "statement")
  :at "agents" (listt "agent")
  :at "acontexts" (co :amap "agent" (listt cot)))
```

Атрибут `"init construct"` определяет модель конструкции языка ФР, с которой операционная семантика начинает выполнение.

Тип `"agent"` определяется как

```
(mot "agent"
  :at "location" (cot :amap "variable" "location")
  :at "location value" (cot :amap "location" "c value")
  :at "value" "c value")
```

Атрибуты `"location"` и `"location value"` представляют модель памяти языка ФР. В рамках этой модели получение значения переменной x требует выполнения двух шагов:

1. Получить ячейку памяти l, ассоциированную с x, через выражение l `= (aget` a `"location"` x`)`.
2. Затем получить значение, хранящееся в этой ячейке, с помощью `(aget` a `"location value"` l`)`.

3.4. Операционная семантика языка ФР

На заключительном этапе 5 специфицируется операционная семантика выполняемых конструкций языка ФР посредством определения семейства деклараций для аспекта `opsem`.

Рассмотрение ограничивается тремя репрезентативными конструкциями: доступ к переменной, выражение равенства и блок операторов. Данные примеры иллюстрируют распространённые шаблоны проектирования операционной семантики: выполнение аспекта в одну стадию, выполнение аспекта, состоящее из последовательности стадий, и выполнение аспекта с рекурсивным повторением стадии. Определение семантики остальных конструкций языка ФР следует тем же шаблонам.

Операционная семантика доступа к значению переменной задается следующей аспектной декларацией:

```
(aspect "opsem" :context ac :type "variable" :instance i :agent ag :do
  (aget ag "location value" (aget ag "location" i)))
```

Операционная семантика выражения типа `"1==2"` разбивается на несколько стадий:

```
(aspect "opsem" :context ac :type "1==2" :do (update-push-acontext ac
   :av "stage" "evaluate 1"))

(aspect "opsem" :context ac :type "1==2" :stage "evaluate 1"
   :instance i :ap i 1 e1 :do
  (update-push-acontext ac :av "stage" "evaluate 2")
  (update-eval-acontext ac :av "instance" e1))
```

[1] https://eclipse.dev/Xtext/
[2] https://www.antlr.org/

```
(aspect "opsem" :context ac :type "1==2" :stage "evaluate 2" :value
   v1 :instance i :ap i 2 e2 :do
  (update-push-acontext ac :av "stage" "compare" :av "val1" v1)
  (update-eval-acontext ac :av "instance" e2))

(aspect "opsem" :context ac :type "1==2" :stage "compare" :value v2
   :val1 v1 :do (equal v1 v2))
```

Разбиение выполнения аспекта на стадии позволяет разделить нормальные и исключительные ситуации: если на какой-либо стадии возникает исключительная ситуация, переход к следующей стадии не происходит. Следовательно, для каждой стадии специфицируется только нормальное выполнение, поскольку при обработке исключения аспектные контексты, связанные с отложенными стадиями, удаляются из соответствующего стека окружения.

Операционная семантика для блока задается двумя аспектными декларациями:

```
(aspect "opsem" :context ac :type "{1}" :instance i :ap i 1 sts :do
  (nmatch :v sts nil :do
    (update-push-acontext :av :stage "block iteration" :statements
       sts :current 0
      :bound (length sts))))

(aspect "opsem" :context ac :type "{1}" :stage "block iteration"
   :statements sts :current j
:bound n :do
  (match :v (< j n) T :do
    (update-push-acontext ac :av "current" (+ j 1))
    (clear-update-eval-acontext ac :instance (nth j sts))))
```

Здесь `(nth k l)` — это Lisp-операция взятия k-го элемента списка l (нумерация элементов начинается с 0).

Аспект `"opsem"` для блока операторов выполняется в две стадии:

1. На стадии `nil` инициализируются параметры следующей стадии.
2. Стадия `block-iteration` выполняет оператор с индексом `current` в теле блока, после чего рекурсивно вызывает себя с параметром `current`, увеличенным на 1.

4. Обогащённая онтология языка ФР и операционная семантика обогащённой модели языка ФР

В предыдущем разделе была построена онтология языка ФР, представляющая собой абстрактное синтаксическое дерево (AST) программных конструкций. В настоящем разделе показано, каким образом строится обогащённая онтология языка ФР путём добавления семантических атрибутов, как задаются алгоритмы вычисления этих атрибутов в ABML и какое влияние обогащённая онтология оказывает на разработку операционной семантики.

4.1. Обогащение онтологии языка ФР

Адаптация предложенного подхода к обогащённым онтологиям языков программирования включает четыре этапа:

1. Расширение типов онтологии дополнительными атрибутами для упрощения, специализации или обобщения операционной семантики.
2. Разработка алгоритмов заполнения новых атрибутов.
3. Определение окружения и агентов для аспектов, реализующих эти алгоритмы.
4. Реализация алгоритмов посредством аспектных деклараций, где в качестве имён аспектов используются названия соответствующих алгоритмов.

На этапе 1 тип "variable declaration", моделирующий объявления переменных, расширяется новым семантическим атрибутом "nonshared":

```
(mot "variable declaration" :at "type" "type" :at "name" "name"
  :at "initializer" "expression" :at "nonshared" bool)
```

Если атрибут "nonshared" установлен в true в модели объявления переменной, то значение такой переменной может быть доступно только по её имени. В дальнейшем будет показано, что доступ к значениям таких переменных может быть реализован с использованием упрощенной модели памяти.

На этапе 2 определяется алгоритм присваивания значений данному атрибуту:

1. Пусть SH — множество разделяемых переменных. Изначально оно пусто.
2. Выполняется обход модели программы, в ходе которого в SH добавляются все переменные, к которым применяется оператор &.
3. Выполняется повторный обход модели, и для каждого объявления переменной, не входящего в SH, атрибут "nonshared" устанавливается в true.

На этапе 3 строятся типы для окружений и агентов "alg:env":

```
(mot "alg:env" :at "SH" (listt "variable"))

(mot "alg:agent")
```

Атрибут "SH" хранит множество разделяемых переменных. Для простоты обязательные атрибуты в этих типах опущены.

На этапе 4 алгоритм реализуется с использованием аспектов, соответствующих второму и третьему шагам алгоритма: "nonshared analysis 1" и "nonshared analysis 2".

Для типа "&1" аспект "nonshared analysis 1" определяется следующим образом:

```
(aspect "nonshared analysis 1" :context ac :type "&1" :instance i :env
   e :ap i 1 x :ap e "SH" SH
  :do (match :v (not (member x SH)) T :do (aset e SH (cons x SH))))
```

Здесь (member x l) — функция Lisp, проверяющая принадлежность элемента x списку l.

Для типа "1==2" аспект "nonshared analysis 1" задаётся как

```
(aspect "nonshared analysis 1" :context ac :type "1==2" :do
  (update-push-acontext ac "stage" "apply to 1"))

(aspect "nonshared analysis 1" :context ac :type "1==2" :stage "apply
   to 1" :instance i :ap i 1 e1
  :do (update-push-acontext ac :stage "apply to 2")
     (clear-update-eval-acontext ac :instance e1))

(aspect "nonshared analysis 1" :context ac :type "1==2" :stage "apply
   to 2" :instance i :ap i 2 e2
  :do (clear-update-eval-acontext ac :instance e2))
```

Для типа "variable declaration" аспект "nonshared analysis 2" определяется как

```
(aspect "nonshared analysis 2" :context ac :type "variable
   declaration" :instance i :ap "env" e
  :ap i "name" x :ap e "SH" SH :do
    (match :v (not (member x SH)) T :do (aset i "nonshared" T)))
```

Для остальных типов аспектные декларации строятся по тому же образцу.

Обогащённая модель исходной программы на языке C после применения описанной процедуры принимает следующий вид:

```
(mo "{1}" :av "1" (list
  (mo "variable declaration" :av "type" "int" :av "name" "x" :av
     "initializer" 0)
  (mo "variable declaration" :av "type" "int" :av "name" "y" :av
     "nonshared" T)
  (mo "variable declaration" :av "type" (mo "pointer type1" :av 1
     "int")
    :av "name" "p" :av "initializer" (mo "&1" :av 1 "x") :av
       "nonshared" T)
  (mo "if12" :av 1 (iobj "1==2" :av 1 (mo "*1" :av 1 "p") :av 2 0)
    :av 2 (iobj "1;" :av 1 (iobj "1=2" :av 1 "y" :av 2 2)))))
```

4.2. Операционная семантика обогащённой модели языка ФР

Теперь можно построить операционную семантику обогащённой модели.

Сначала уточним типы окружений и агентов. Тип окружения не изменяется, тогда как тип агентов получает следующее определение:

```
(mot "agent"
  :at "location" (mot :amap "variable" "location")
  :at "location value" (mot :amap "location" "c value")
  :at "nonshared" (mot :amap "variable" bool)
  :at "nonshared variable value" (mot :amap "variable" "c value")
  :at "value" "c value")
```

Новые атрибуты `"nonshared"` и `"nonshared variable value"` характеризуют неразделяемые переменные, предоставляя для них упрощенную модель памяти.

Рассмотрим, как изменятся декларации аспекта `"opsem"` с учетом обогащённой модели на примере типа `"variable"`:

```
(aspect "opsem" :context ac :type "variable" :instance i :agent ag :do
  (match :av (aget ag "nonshared" i) T
    :do (aget ag "nonshared variable value" i)
    :exit (aget ag "location value" (aget ag "location" i))))
```

Эта декларация по-прежнему следует шаблону выполнения в одну стадию, однако теперь она сначала определяет, является ли переменная разделяемой, а затем соответствующим образом извлекает её значение, используя состояние памяти агента *ag*, задаваемое его атрибутами.

5. Обзор родственных работ

Для того чтобы перейти от абстрактного определения операционной семантики к практико-ориентированному построению последней в виде обычного исполняемого кода, в настоящей статье сначала формулируются требования к фреймворку, который должен обеспечить такой переход. Затем анализируется, в какой степени существующие подходы к семантике удовлетворяют этим требованиям.

Чтобы быть формально строгим и одновременно практически полезным в контексте современной программной инженерии, фреймворк для построения операционной семантики языков программирования должен отвечать следующим 19 требованиям:

1. Обеспечивать возможность легкой модификации операционной семантики отдельных языковых конструкций.
2. Предоставлять унифицированную нотацию для операционной семантики различных языков.
3. Поддерживать простое определение семантики для комбинаций языков.
4. Допускать выполнение операционной семантики как программного кода.
5. Позволять строить семантику для новых языков, не имеющих устоявшегося синтаксиса.
6. Поддерживать программирование и отладку операционной семантики.

7. Обеспечивать изоляцию и отладку операционной семантики отдельных конструкций в выделенной среде выполнения.
8. Сохранять структурную форму исходной программы в том промежуточном представлении, для которого определяется семантика.
9. Предлагать выразительный язык программирования для описания операционной семантики.
10. Упрощать операционную семантику за счет включения дополнительной информации о программе.
11. Поддерживать адаптацию семантики при добавлении или изменении языковых конструкций.
12. Позволять строить семантику для фрагментов языков или даже для отдельных программ.
13. Обеспечивать возможность построения семейства специализированных для конкретных задач операционных семантик для одного языка программирования, разделяющих общие компоненты.
14. Допускать повторное использование существующих операционных семантик при разработке новой.
15. Позволять настраивать операционную семантику под целевую среду выполнения.
16. Быть легким в освоении.
17. Отражать конкретные особенности отдельных языковых конструкций, сохраняя при этом целостное представление.
18. Обеспечивать быстрое построение операционной семантики.
19. Позволять строить операционную семантику, разбитую на независимые компоненты.

Существует обширное число исследований, посвященных определению и разработке операционной семантики языков программирования. Для обзора и сравнительного анализа были отобраны подходы, удовлетворяющие по крайней мере одному из следующих критериев:

- предлагают оригинальный, отличный от других подход к разработке операционной семантики;
- предоставляют практический фреймворк для построения операционной семантики;
- определяют семантику, охватывающую существенное подмножество популярного или промышленно значимого языка программирования.

В таблицах 1 и 2 представлено обобщённое сравнение предлагаемого подхода с отобранными родственными подходами на основании требований 1–19. При этом используется трёхбалльная шкала:

- 0 — требование не выполняется или выполняется лишь в исключительных случаях;
- 1 — требование выполняется достаточно часто;
- 2 — требование выполняется в большинстве случаев.

В работе Чаппе с соавторами [3] структуры CTrees в среде Coq применяются для построения слоистой монадической операционной семантики минимального конкурентного подмножества LLVM IR, обозначаемого как μthread IR. Данная семантика допускает исполнение за счет извлечения (extraction) в OCaml; авторы также устанавливают такие свойства, как эквациональные рассуждения, бисимуляцию и отношения симуляции между моделями памяти. Несмотря на гибкость и модульность, обеспечиваемые монадической архитектурой, последняя жёстко привязана к LLVM IR, что не позволяет предложить действительно универсальную нотацию, применимую к различным языкам. Как следствие, определение комбинированной семантики для нескольких языков оказывается затруднительным. Кроме того, опора на развитые концепции функционального программирования (например, монады) делает данный подход сложным для освоения теми, кто не имеет фундаментальной математической подготовки.

Штайнгартнер и Новицка [4] предложили структурную операционную семантику (SOS) для двух предметно-ориентированных языков управления роботами. В их подходе задаются правила боль-

Table 1. Implementation of requirements 1–19 in related works (articles 5–27)

Таблица 1. Реализация требований 1–19 в родственных работах (статьи 5–27)

Описание работы	1 2	3 4	5 6	7 8	9 10	11 12	13 14	15 16	17 18	19	Количество оценок 2
Чаппе и др. (2025) [3]	1 1	1 2	1 1	1 2	2 1	2 2	2 2	1 0	2 2	2	10
Штайнгартнер и Новицка (2024) [4]	1 0	0 2	1 1	1 2	2 1	1 2	0 0	1 2	2 1	1	6
Гончаров и др. (2023) [5]	0 2	1 0	1 0	0 2	0 1	1 1	1 2	1 0	2 0	1	4
Махе и др. (2022) [6]	1 1	0 1	1 1	0 2	0 1	1 1	0 0	0 1	2 1	1	2
Джоно (2024) [7]	2 1	1 2	2 1	1 2	2 2	2 2	1 2	2 1	2 1	2	12 ✓
Проенца и Эдикссовен (2023) [8]	2 1	1 2	2 2	2 2	2 1	2 2	1 2	2 1	2 2	2	14 ✓
Чен и др. (2023) [9]	2 1	1 2	1 2	2 2	2 1	2 2	2 2	1 1	2 2	2	13 ✓
Сюй и др. (2025) [10]	1 0	0 0	1 0	0 1	0 1	1 1	1 1	2 1	2 1	1	2
Рехенбергер и Фрухвирт (2025) [11]	2 0	0 2	1 1	1 2	1 2	2 2	1 1	2 1	2 2	2	10
Ли и др. (2023) [12]	1 1	1 0	1 0	0 1	0 1	1 1	2 1	2 0	2 1	1	3
Лемридже и др. (2024) [13]	2 1	0 2	1 1	1 2	1 2	2 2	1 1	1 1	1 1	2	7
Ченчиарелли и др. (1997) [14], Ченчиарелли и др. (1999) [15]	1 0	1 0	1 0	0 2	0 1	1 1	1 1	1 0	2 1	1	2
Дроссопулу и Айзенбах (1998) [16]	2 1	0 0	1 0	0 2	0 1	2 2	1 1	1 0	1 1	2	5
Коция и Реджио (1998) [17]	1 0	0 0	0 0	0 2	0 1	1 1	0 0	0 1	2 1	1	2
Весонга (2012) [18]	2 1	0 2	1 2	1 2	2 1	2 1	0 1	1 1	2 1	2	8
Белблидия и Дебаби (2007) [19]	1 0	0 2	0 1	0 2	1 1	1 0	0 0	1 0	2 0	1	3
Аттали и др. (1995) [20], Аттали и др. (1998) [21]	1 0	0 1	0 1	0 1	1 1	1 0	0 0	1 0	2 1	2	2
Гуревич и Хаггинс (1992) [22]	1 1	0 1	0 0	0 1	0 1	1 0	0 0	0 0	2 0	1	1
Уоллес (1993) [23]	1 0	0 0	0 0	0 1	0 1	1 1	0 0	0 0	2 0	1	1
Камиллери (1989) [24]	1 0	0 1	0 1	1 1	0 0	1 0	0 1	0 1	1 0	1	0
Мэтьюз и Финдлер (2009) [25]	2 2	2 2	1 2	2 1	1 1	2 2	1 2	2 1	2 1	2	12 ✓

Table 2. Implementation of requirements 1–19 in related works (articles 28–41) and our approach

Таблица 2. Реализация требований 1–19 в родственных работах (статьи 28–41) и нашем подходе

Описание работы	1 2	3 4	5 6	7 8	9 10	11 12	13 14	15 16	17 18	19	Количество оценок 2
Вердехо и Марти-Олиэт (2006) [26]	2 2	1 2	1 2	1 1	2 1	2 2	1 2	1 0	2 1	2	10
Штродер и др. (2011) [27]	0 0	0 0	0 0	0 2	0 1	0 1	0 0	1 0	1 1	1	1
Росу и Сербнат (2010) [28]	2 2	2 2	2 2	2 2	2 1	2 2	2 2	1 1	2 1	2	15 ✓
Нипков и Клейн (2014) [29]	1 1	0 1	1 1	1 2	0 1	1 2	1 1	1 0	2 0	1	3
Непомнящий и др. (2002) [30]	1 0	0 0	1 0	0 2	0 1	1 2	1 1	0 0	2 1	1	3
Норриш (1998) [31]	0 0	0 1	0 1	1 2	0 1	1 2	1 1	0 0	2 0	1	3
Келсен и Ма (2008) [32]	2 1	1 1	2 1	1 2	2 1	2 2	1 1	1 1	2 2	2	9
Смединг (2009) [33]	2 0	0 2	1 2	1 2	2 1	2 2	1 1	1 1	2 1	2	9
Кел (2021) [34]	2 1	0 2	1 1	2 2	2 1	2 2	1 2	1 1	2 2	2	11
Фостер и др. (2025) [35]	0 2	0 2	0 0	0 0	0 0	0 0	0 2	0 0	0 2	0	4
Шенг и др. (2019) [36]	1 0	0 2	0 2	2 2	2 0	0 0	0 0	0 0	0 0	0	5
Хуанг и др. (2019) [37]	2 1	1 2	1 2	2 2	2 1	2 2	2 2	1 1	2 2	2	13 ✓
Йон и др. (2024) [38]	2 0	0 2	2 2	2 2	2 0	2 2	0 2	0 2	2 2	2	14
Муллиган и др. (2014) [39]	2 2	1 2	2 2	1 2	2 1	2 2	2 2	1 2	2 2	2	15 ✓
Наш подход (2025)	2 2	2 2	2 2	2 2	2 2	2 2	2 2	2 2	2 2	2	19 ✓

шого шага для команд перемещения и управляющих конструкций, а также реализуется абстрактная машина, компилирующая программы в форму, близкую к байт-коду. Выполнение этих команд визуализируется при помощи учебного симуляционного инструмента, что обеспечивает интерактивное понимание семантического поведения. Хотя данный подход хорошо адаптирован к целевым предметно-ориентированным языкам, ему не хватает универсальной нотации, применимой к разным языкам, что затрудняет построение комбинированной семантики для нескольких языков. Кроме того, тесная связь с конкретной языковой архитектурой ограничивает возможность повторного использования семантики в иных контекстах.

Гончаров с соавторами [5] предложили абстрактный GSOS-фреймворк высшего порядка, представляющий собой категориальный подход, в котором операционные правила задаются посред-

ством так называемых динатуральных преобразований (GSOS-законов высшего порядка). Несмотря на то, что предлагаемый фреймворк отличается высоким уровнем абстракции и композиционности, он не обеспечивает непосредственного исполнения семантики и не предоставляет практических инструментов для программирования и отладки. Кроме того, использование развитых математических понятий, в частности теории категорий, затрудняет его освоение для лиц, не имеющих углубленной подготовки в области формальных методов.

Maxe с соавторами [6] предложили структурную операционную семантику (SOS) для разработанного ими языка взаимодействий — формализма в стиле алгебры процессов, предназначенного для описания диаграмм последовательности сообщений и диаграмм последовательности в целом. Несмотря на то, что данный подход дает прочное теоретическое обоснование, устанавливая соответствие между операционной и денотационной семантиками, он не рассчитан на непосредственное исполнение и не поддерживает реализацию или отладку семантики на практике. Кроме того, семантика выражается в формальном математическом стиле, а не непосредственно в виде конструкций языка программирования, что ограничивает её практическую применимость.

В работе Джоно [7] представлена адаптивная структурная операционная семантика (ASOS) — метаязык, развивающий идеи MSOS и поддерживающий как правила по умолчанию, так и правила адаптации для предметно-ориентированных языков (например, RobLANG). Спецификации на ASOS транслируются посредством системы SEALS в исполняемые интерпретаторы. Хотя данный подход обеспечивает гибкое средство динамической настройки семантики, он не вводит универсальной унифицированной нотации, применимой к различным языкам, что затрудняет построение комбинированной семантики для многоязыковых систем. Кроме того, концептуальная сложность модели снижает её доступность для лиц, не имеющих основательной подготовки в области математики и программирования.

Проенца и Эдиксховен [8] представили фреймворк Caos — инфраструктуру на языке Scala для определения структурной операционной семантики (SOS) применительно к небольшим языкам программирования и хореографиям, включая язык While, хореографические языки и лямбда-исчисление. В основе подхода лежат типы данных Scala и определения правил. Спецификации компилируются в интерактивный веб-аниматор, который визуализирует выполнение программных переходов, предоставляя пользователю наглядное пошаговое представление о семантическом поведении. Несмотря на преимущества в реализации, визуализации и модульности, фреймворк не предлагает универсальной унифицированной нотации, легко применимой к разным языкам, что усложняет комбинирование семантик нескольких языков. Кроме того, зависимость от Scala и концепций функционального программирования может стать барьером для пользователей с ограниченным опытом в программировании и формальных методах.

Чен с соавторами [9] разработали исполняемую мелкошаговую операционную семантику для Verilog в рамках фреймворка K. Данный фреймворк обеспечивает выполнение и отладку кода Verilog на основе этой семантики. Кроме того, семантика поддерживает модульное рассуждение и повторное использование, позволяя изолировать и адаптировать отдельные языковые конструкции и подъязыки. Вместе с тем некоторые аспекты, в частности сложная семантика синтеза или недетерминированное поведение, остаются трудными для представления или полной верификации в границах этого фреймворка. Предлагаемый подход дает строгую и исполняемую операционную семантику, заточенную под Verilog, однако не вводит универсальной нотации для применения к другим языкам и не поддерживает комбинирование семантик нескольких языков. Более того, из-за тесной привязки к устоявшемуся синтаксису Verilog такой подход менее пригоден для определения семантики языков, находящихся в стадии разработки или не имеющих стабильного синтаксиса.

Сюй с соавторами [10] предложили операционную семантику малог шага, адаптированную для Crystality — предметно-ориентированного языка, предназначенного для параллельного исполнения в среде виртуальных машин Ethereum. Предлагаемый подход задает точную и исполняемую семантику для Crystality в контексте параллельной EVM, однако ему не хватает унифицированной нотации, пригодной для описания семантики различных языков, а также он не поддерживает удобного объединения семантик разных языков. Кроме того, дополнительная сложность, вносимая параллельным выполнением и специфическими для блокчейна особенностями, снижает доступность этого подхода для лиц, не имеющих углубленной технической подготовки.

Рехенбергер и Фрухвирт [11] предложили формальную операционную семантику для FreeCHR — варианта языка обработки правил с ограничениями (CHR), поддерживающего более гибкое и обобщенное применение правил. Хотя данный подход дает точную исполняемую семантику, ориентированную именно на FreeCHR, ему недостаёт унифицированной нотации, применимой к различным языкам, и он не обеспечивает легкого объединения семантик нескольких языков. Кроме того, его строго формальный и технический характер затрудняет освоение разработчиками программного обеспечения, не имеющими глубоких познаний в математике и программировании.

Ли с соавторами [12] предложили новый фреймворк для операционной семантики, в котором ограничения честности (fairness) включаются непосредственно в семантику конкурентных программ. Несмотря на формальную выразительность и математическую строгость, данная семантика не является исполняемой и не обладает интерактивной поддержкой или средствами отладки, что ограничивает её применение для практического прототипирования. Вместе с тем она позволяет вести рассуждения о честности в широком спектре областей. Предлагаемый подход не вводит унифицированной нотации, применимой к разным языкам, и не поддерживает легкого объединения семантик нескольких языков. Более того, его специализированный характер и сложность делают его менее доступным для пользователей, не имеющих глубоких математических знаний и подготовки в области программирования.

Лемридже с соавторами [13] предложили слабую операционную семантику для AuDaLa — предметно-ориентированного языка, предназначенного для моделирования и анализа бизнес-процессов, учитывающих данные. Хотя этот подход ориентирован именно на AuDaLa, а не на языки общего назначения, он отличается расширяемостью и модульностью, что облегчает адаптацию и частичные определения. Тем не менее его применимость за пределами данного DSL ограничена. Подход не предоставляет унифицированной нотации, применимой к разным языкам, и не поддерживает легкого объединения семантик нескольких языков. Более того, семантика задается с использованием формальных методов спецификации, а не непосредственно на языке программирования, что снижает её доступность и простоту реализации.

Ченчиарелли с соавторами [14] (1997) и [15] (1999) предложили основанную на событиях операционную семантику как для последовательных, так и для многопоточных программ на языке Java. Данная семантика отличается точностью и выразительностью, однако не является исполняемой и не встроена в какой-либо инструмент, позволяющий проводить прямую симуляцию. Она задается в рамках математического формализма, предназначенного для рассуждений и верификации, и поэтому менее пригодна для прототипирования или инструментальной разработки. Событийно-ориентированная семантика ориентирована исключительно на Java и не предоставляет унифицированной нотации, применимой к другим языкам, равно как и не поддерживает простой интеграции семантик нескольких языков. Помимо этого, семантика определяется с использованием формальных правил, а не выражается непосредственно на языке программирования, что ограничивает простоту её реализации и практическое применение.

Дроссопулу и Айзенбах [16] представили формальную операционную семантику малого шага для базового подмножества Java, а также соответствующую систему типов и доказательство кор-

ректности типизации. Несмотря на строгую формализацию операционной семантики Java и системы типов, предлагаемый подход не дает универсальной нотации, применимой к разным языкам, и не позволяет просто объединять семантики нескольких языков. Кроме того, данная семантика ориентирована на формальную верификацию, а не на исполнение, и потому не является непосредственно выполняемой.

Коция и Реджио [17] разработали структурированную операционную семантику малого шага для фрагмента Java. Данный подход не предоставляет универсальной нотации, пригодной для описания семантики различных языков, и не поддерживает простого объединения семантик разных языков. Кроме того, семантика не задаётся непосредственно на языке программирования.

В работе Весонги [18] предложена операционная семантика для Javalite — упрощённого фрагмента Java. Данному подходу недостает универсальной нотации, применимой к различным языкам, и он не поддерживает легкого объединения семантик нескольких языков. Поскольку подход строится вокруг фиксированного подмножества Java, он мало подходит для определения семантики языков, находящихся в разработке и не имеющих стабильного синтаксиса.

Белблидия и Дебаби [19] разработали операционную семантику, ориентированную на язык виртуальной машины Java (JVML). Данный подход не предоставляет унифицированной нотации для нескольких языков и не позволяет просто объединять семантики разных языков. Он также мало пригоден для определения семантики новых языков, не имеющих стабильного синтаксиса, а также для работы с отдельными языковыми фрагментами. Кроме того, подход может быть сложен для освоения теми, кто не обладает математической подготовкой и навыками программирования, и не способствует быстрой разработке операционной семантики.

Аттали с соавторами [20] предложили фреймворк операционной семантики, предназначенный для языка программирования Eiffel. Данный подход не обеспечивает унифицированной нотации для разных языков и не поддерживает легкого объединения семантик нескольких языков. Он также сталкивается с трудностями при определении семантики для новых языков, не имеющих стабильного синтаксиса, и не упрощает и не ускоряет процесс разработки семантики. Кроме того, подход остается малодоступным для пользователей без серьёзной математической подготовки и навыков программирования, что ограничивает его практическую применимость.

Гуревич и Хаггинс [22] представили фреймворк операционной семантики, задающий строгое описание поведения языка C. Данный подход сконцентрирован преимущественно на языке C и потому не обеспечивает ни простоты модификации отдельных языковых конструкций, ни унифицированной нотации, применимой к различным языкам, ни поддержки комбинирования семантик нескольких языков. Кроме того, ему недостает надёжных средств для выполнения и отладки семантики, и он не упрощает семантическое описание за счёт добавления дополнительной информации о программе. Более того, подход не является в достаточной степени доступным для пользователей, не обладающих серьёзными математическими навыками, и не ориентирован на быструю разработку или модульное проектирование семантики, что ограничивает его гибкость и практическую применимость.

Уоллес [23] построил операционную семантику для основных конструкций языка C++, охватывающую объектно-ориентированные возможности, наследование, полиморфизм, шаблоны и обработку исключений. Данный подход не предоставляет унифицированной нотации, применимой к различным языкам, и не позволяет объединять семантики нескольких языков. Ему также недостает поддержки выполнения, отладки и изолированного тестирования семантики; кроме того, он мало пригоден для определения семантики языков без стабильного синтаксиса, равно как и для непосредственного выражения семантики на языке программирования. Вопросы повторного использования уже существующей семантики, её настройки под конкретные среды исполнения, а также упрощения семантического описания за счет дополнительной информации о программе остаются нерассмот-

ренными. Наконец, подход может оказаться сложным для освоения лицами без фундаментальной математической подготовки, что ограничивает скорость разработки семантики.

Камиллери [24] разработал операционную семантику для моделирования параллельных и коммуникационных возможностей языка OCCAM. Данный подход не предоставляет унифицированной нотации для нескольких языков и не ориентирован на объединение семантик различных языков. Он также мало пригоден для определения семантики новых языков, не имеющих стабильного синтаксиса, для непосредственного выражения семантики на языке программирования и для упрощения семантических описаний за счет дополнительной информации о программе. К тому же подход обеспечивает лишь ограниченную поддержку повторного использования существующей семантики и её настройки под конкретные среды выполнения, а также является достаточно сложным для освоения, что замедляет процесс разработки.

Мэтьюз и Финдлер [25] разработали фреймворк операционной семантики, ориентированный на многоязычные программы и обеспечивающий бесшовную совместимость между различными языками программирования в рамках унифицированной модели выполнения. Вместе с тем данный подход не приспособлен для языков, не имеющих стабильного синтаксиса, для непосредственной спецификации семантики на языке программирования общего назначения, а также для упрощения семантики за счёт дополнительной информации о программе. Кроме того, высокая сложность освоения этого подхода может замедлять быструю разработку.

Вердехо и Марти-Олиэт [26] предложили исполняемую структурную операционную семантику, реализованную в системе Maude — высокопроизводительном фреймворке, базирующемся на логике переписывания. Предлагаемый подход может оказаться сложным для понимания без соответствующей математической подготовки, что снижает его доступность и замедляет процесс разработки. Кроме того, он не обеспечивает поддержки определения семантики для языков, не имеющих стабильного синтаксиса, объединения семантик нескольких языков, упрощения семантики за счёт дополнительной информации о программе, а также адаптации семантики к различным средам выполнения.

Штродер с соавторами [27] предложили линейную операционную семантику для ISO Prolog, предназначенную для анализа завершимости и сложности вычислений путем моделирования последних в виде линейных последовательностей переходов между состояниями. Данный подход не упрощает модификацию отдельных языковых конструкций, не вводит унифицированной нотации для разных языков и не поддерживает объединения семантик нескольких языков. Основное внимание уделяется формальному анализу, а не выполнению или отладке; кроме того, подход не предоставляет возможностей для задания семантики непосредственно на языке программирования, а также для её повторного использования и модульной адаптации. Следует также отметить, что предлагаемая семантика носит довольно специализированный характер и отличается математической сложностью, что затрудняет её освоение и, как следствие, замедляет быструю разработку. Поддержка настройки семантики для различных сред выполнения и её упрощения за счёт дополнительной информации о программе является лишь ограниченной.

Росу и Сербнат [28] представили семантический фреймворк K — модульную систему, основанную на правилах и логике переписывания, предназначенную для задания исполняемой операционной семантики. Данный фреймворк является сложным для начинающих, что может замедлять быструю разработку; кроме того, он предлагает лишь ограниченную встроенную поддержку упрощения семантики за счет дополнительной информации о программе и её настройки для различных сред выполнения. Хотя фреймворк K поддерживает работу с частичными языковыми фрагментами и семантикой, ориентированной на конкретные задачи, его сложность может препятствовать эффективному использованию этих возможностей.

Нипков и Клейн [29] использовали Isabelle/HOL для определения и верификации операционной семантики, преимущественно в стиле малых и больших шагов, применительно к императивным и функциональным языкам. Данный подход требует серьезной математической подготовки. Он также не предоставляет полностью унифицированной нотации, одинаково пригодной для различных языков, и не упрощает семантику автоматически за счет добавления дополнительной информации о программе. Ввиду своего формального характера этот подход оказывается менее гибким при работе с новыми языками, не имеющими стабильного синтаксиса, и может быть более медленным при быстрой разработке по сравнению с более легкими и специализированными инструментами.

Непомнящий с соавторами [30] представили язык C-light — упрощенное подмножество языка программирования C, предназначенное для формальных рассуждений и верификации. Данный подход не предлагает унифицированной нотации для нескольких языков и не предоставляет простых способов объединения семантик разных языков. Он также не обеспечивает поддержки выполнения и отладки семантики и мало пригоден для определения семантики новых языков, не имеющих устоявшегося синтаксиса. Подход может оказаться сложным для изучения лицами без основательной подготовки в области формальных методов и не делает акцента на быстрой разработке или упрощении семантики за счёт добавления дополнительной информации о программе. Кроме того, повторное использование или адаптация семантики для различных сред выполнения не являются его основной задачей.

Норриш [31] представил формальную операционную семантику для обширного подмножества языка C с использованием системы HOL. Данный подход не упрощает модификацию семантики применительно к отдельным конструкциям, не обладает унифицированной нотацией для нескольких языков и не поддерживает объединение семантик разных языков. Он предоставляет лишь незначительные возможности для выполнения или отладки семантики в интерактивном режиме, не приспособлен для быстрого определения семантики новых языков и может быть сложен для освоения без подготовки в области формальных методов. Кроме того, повторное использование или настройка семантики под конкретные среды выполнения, а также упрощение семантики за счет добавления дополнительной информации о программе не относятся к сильным сторонам этого подхода.

Келсен и Ма [32] предложили операционную семантику, задаваемую посредством исполняемых метамоделей с использованием объектно-ориентированных фреймворков моделирования (например, EMF). Данный подход не предоставляет унифицированной нотации для различных языков и не поддерживает объединение семантик нескольких языков. Он даёт лишь ограниченные инструменты для выполнения и отладки семантики и мало пригоден для языков с нестабильным синтаксисом. Кроме того, ему недостаёт надёжной поддержки повторного использования существующей семантики, её настройки под конкретные среды выполнения, а также упрощения семантики за счет добавления дополнительной информации о программе. Хотя этот подход отличается относительной легкостью, он все равно может оказаться сложным для тех, кто не имеет математической подготовки.

Смединг [33] предложил формальную и исполняемую операционную семантику для подмножества Python, реализованную на функциональном языке OCaml. Данный подход не обеспечивает действительно унифицированной нотации для описания семантики различных языков, поскольку фреймворк тесно привязан к Python, что ограничивает его общую применимость. Кроме того, он предполагает наличие подготовки в области программирования и формальной семантики.

Кел [34] предложил операционную семантику для Python, реализованную в рамках SOS-подхода как формальную и исполняемую. Работа с фреймворком K и формальными методами спецификации может представлять сложность для пользователей, не имеющих фундаментальной подготовки в области языков программирования и формальной семантики. Несмотря на то, что семантика

является исполняемой, поддержка отладки отдельных семантических правил остается ограниченной и для эффективного использования может требовать углубленных знаний либо привлечения дополнительных инструментов.

Фостер с соавторами [35] предложили подход, направленный на устранение разрыва между программными моделями и программным кодом. В его основе лежит моделирование программного обеспечения с помощью коммуникативных коиндуктивных процессоподобных структур, известных как деревья взаимодействий (ITrees). Формализация данного подхода в Isabelle/HOL позволяет верифицировать получаемые модели с использованием логики Хоара, а также выполнять извлеченный код через операционную семантику. Рассмотрение наряду с процессно-ориентированными языками также традиционного императивного кода демонстрирует широкую применимость предложенного метода. Вместе с тем использование трасс в качестве основы для операционной семантики смещает акцент в сторону дедуктивной верификации поведения программного обеспечения, а не его анализа непосредственно через операционную семантику. Другим ограничением выступает необходимость освоения нетривиальной теории деревьев взаимодействий.

Шенг с соавторами [36] предложили подход для вывода операционной семантики на основе алгебраической. Данный подход представляет алгебраическую семантику в нормальной форме, опирающейся на охраняемые выборы, и определяет в помощнике доказательств Coq теоремы, служащие стратегиями для вывода правил операционной семантики из нормальной формы. Вместе с тем подход ориентирован в первую очередь на язык моделирования многопоточных дискретных событий (MDESL), что ставит под вопрос его применимость к языкам программирования, не основанным на потоках. В частности, могут возникнуть трудности при определении нормальной формы для неохраняемого кода. Кроме того, формализация данного подхода в помощнике доказательств налагает дополнительную нагрузку на обучение, поскольку требует знакомства с основами системы Coq.

Хуанг с соавторами [37] предложили использовать фреймворк K для определения операционной семантики языка программирования ST, в результате чего была получена семантика, получившая название KST. Данный подход применяет грамматику в форме Бэкуса — Наура для задания правил переписывания применительно к языковым конструкциям. Эти правила описывают, каким образом подпрограммы изменяют состояния программы и как они могут быть возобновлены в контексте программ на ST. Существенным преимуществом подхода выступает естественная поддержка фреймворком K определения незавершающегося цикла сканирования, в рамках которого обновляются входные данные, выполняется программа на ST и регистрируются выходные данные. Вместе с тем заметным недостатком является тесная связь между грамматикой языка и его семантикой, что усложняет внесение изменений в набор конструкций языка программирования.

Йон с соавторами [38] представили предметно-ориентированный язык SpecTec, предназначенный для спецификации формальной семантики WebAssembly (Wasm). SpecTec позволяет определять правила системы типов и операционную семантику в виде отношений редукции, а также направлен на упрощение существующего процесса определения и расширения стандарта Wasm. Этот подход также поддерживает генерацию псевдокодовых представлений стандарта Wasm и соответствующего интерпретатора. Вместе с тем применимость SpecTec к другим языкам программирования остается неопределенной, поскольку до настоящего времени он использовался исключительно для Wasm. Другим ограничением выступает то, что правила отношений редукции для языковых конструкций могут быть заданы в различных стилях, а отсутствие их унификации усложняет автоматическую генерацию интерпретаторов на основе этих правил. Хотя предлагаемый подход включает алгоритм для унификации правил редукции, его способность справляться с унификацией в общих случаях может быть ограничена.

Муллиган с соавторами [39] представили фреймворк Lem для определения формальной семантики языков программирования. Данный фреймворк основан на задании семантики с использованием функционального языка Lem. Полученная семантика может быть извлечена в функциональный язык OCaml или в теории для помощников доказательств Isabelle/HOL, HOL4 и Coq. Другой бэкенд фреймворка обеспечивает автоматическую генерацию HTML- или LaTeX-представлений, предназначенных для описания стандартов языков. Применение Lem для формализации многопроцессорных архитектур IBM Power и ARM, стандарта C/C++11, протоколов сети TCP/IP и языка OCaml демонстрирует широкую применимость этого подхода. Вместе с тем определение семантики в виде индуктивных отношений вносит трудности при формировании требуемых выводов для генерации представления операционной семантики в OCaml. В связи с этим пользователь должен явно задавать «входные» и «выходные» компоненты каждого индуктивного отношения. Другим ограничением выступает необходимость ручного редактирования автоматически сгенерированных теорий для помощников доказательств — например, добавления мер для обоснования завершимости рекурсивных функций в некоторых случаях.

Продемонстрируем, что предложенный подход удовлетворяет всем требованиям 1–19:

1. Определяется локальная операционная семантика для каждой отдельной конструкции языка программирования посредством семейства деклараций аспекта `opsem`, индексированного типами конструкций.
2. Работа с языками программирования осуществляется через их модели, благодаря чему семантика языковых конструкций задается в унифицированной нотации, предоставляемой языком ABML.
3. Если программа включает конструкции из нескольких языков, сначала строятся модели каждого языка, после чего составляется единая модель самой программы. Семантика этой составной модели выводится на основе семантики отдельных языковых моделей — всё в рамках ABML.
4. Язык ABML, используемый для задания операционной семантики, является лексическим расширением SBCL (диалекта Lisp), что обеспечивает исполнимость семантики.
5. Подход поддерживает разработку операционной семантики даже для языков с развивающимся синтаксисом, поскольку семантика задается непосредственно на уровне онтологических моделей без привязки к конкретному синтаксису.
6. Любая стандартная среда разработки Lisp может применяться для программирования и отладки операционной семантики в ABML.
7. Семантика отдельной языковой конструкции в конкретном окружении может быть отлажена в интерпретируемом режиме путем вычисления аспекта `"opsem"` в соответствующей модели окружения.
8. Благодаря тщательному подбору имен атрибутов онтологическая модель может отражать свой реальный аналог как структурно, так и терминологически.
9. Операционная семантика определяется на языке ABML, который является языком программирования.
10. Подход позволяет упростить операционную семантику путем обогащения онтологической модели языка дополнительными атрибутами. Например, в разделе 4 модель языка C была расширена атрибутом `nonshared`, что дало возможность использовать упрощенную модель памяти для переменных, помеченных этим атрибутом. Результаты семантического анализа (например, статический анализ, дедуктивная верификация, проверка моделей), выраженные в виде значений атрибутов, могут применяться для обогащения модели. В более простых случаях внешние инструменты не требуются — анализ может быть реализован через введение новых аспектов.

11. Операционная семантика легко расширяема: новые программные конструкции могут быть добавлены, а существующие изменены путем создания или корректировки их моделей и обновления соответствующих деклараций аспекта `opsem`.
12. Подход поддерживает инкрементальную разработку операционной семантики. Так, в разделе 4 семантика для подмножества конструкций C, используемых в конкретной программе, была разработана независимо. Это позволяет определять семантику лишь для тех конструкций, которые имеют отношение к решаемой задаче.
13. Наш подход даёт возможность создавать варианты семантики для одного языка, адаптированные к разным задачам, путем построения различных аспектных деклараций для одной и той же программной конструкции.
14. Существующая семантика может быть повторно использована при разработке новой, поскольку все они определены в рамках единой концептуальной основы.
15. Семантика легко адаптируется к изменениям в среде выполнения, так как среда также моделируется с помощью типов ABML. Для адаптации достаточно изменить соответствующие атрибуты этих типов.
16. Подход отличается относительной простотой освоения, поскольку: (а) ABML имеет небольшой набор конструкций; (б) их понимание не требует глубоких математических знаний или навыков программирования; (в) наш опыт работы с фрагментами различных языков (C, C++, Python, Rust, Verilog, Go, ST) показывает, что набор необходимых внешних функций Lisp невелик и может быть изолирован — нет необходимости полностью осваивать весь язык Lisp.
17. Подход поддерживает как низкоуровневое, так и высокоуровневое представление о поведении программы. Низкоуровневое поведение отражается через выполнение стадий аспекта с использованием сопоставителей, тогда как высокоуровневая структура представлена концептуальными построениями, основанными на атрибутах.
18. Концептуально определение операционной семантики включает моделирование языка программирования и его среды выполнения с использованием типов и атрибутов объектов, а также описание эволюции значений атрибутов во время выполнения. Реализация сводится к программированию на ABML — лаконичном и доступном языке. Таким образом, как этапы проектирования, так и реализации являются достаточно простыми, что позволяет быстро разрабатывать семантику.
19. Процесс разработки семантики естественным образом декомпозируется. Каждая языковая конструкция может обрабатываться независимо благодаря принципу локальности: семантика конструкции определяется исключительно через атрибуты, которые она требует, без необходимости знать другие атрибуты или конструкции в системе.

Результаты проведенного сравнительного анализа, обобщённые в таблицах 1 и 2, показывают, что 8 из 36 рассмотренных подходов (включая наш) могут служить основой для полноценного фреймворка; соответствующие подходы отмечены знаком ✓. Отбор проводился по критерию отсутствия нулевых оценок и наличия оценки 2 более чем по десяти требованиям. Ключевыми преимуществами нашего подхода являются независимость от специализированных инструментов, сравнительная лёгкость освоения и более тонкая настройка среды выполнения.

Заключение

В нашей работе были получены следующие новые результаты:

- усовершенствованная версия языка ABML — нового предметно-ориентированного языка, предназначенного для эффективного моделирования дискретных систем, включая различные языки программирования;
- новый подход к построению исполняемой операционной семантики различных языков программирования с использованием ABML;

- пример, демонстрирующий применение предложенного подхода к практически значимому фрагменту широко используемого языка программирования C;
- Набор из 19 требований к фреймворку, способному работать с операционной семантикой языков программирования как с исполняемым программным кодом;
- анализ 35 современных подходов к семантике и оценку их соответствия указанным 19 требованиям;
- выявление наиболее подходящих подходов для разработки такого фреймворка.

Сравнительный анализ показывает, что предложенный подход работает на уровне ведущих существующих решений и превосходит их в нескольких ключевых аспектах: он не опирается на специализированные инструменты, отличается относительной простотой освоения и допускает более тонкую настройку среды выполнения. Это позволяет рассматривать наш подход как сильного кандидата для создания всеобъемлющего фреймворка, что и составляет один из главных вкладов данной работы. Другим значительным вкладом статьи является сам язык ABML. Потенциальные области его применения выходят за рамки операционной семантики. Он может использоваться для проектирования и прототипирования программных и информационных систем на основе легковесной онтологии типов и атрибутов объектов, для реализации формальных методов (таких как статический анализ, дедуктивная верификация и проверка моделей), в качестве унифицированного интерфейса для систем, основанных на этих методах, а также для прототипирования цифровых двойников.

Текущим ограничением нашего подхода — общим для всех подходов к построению исполняемой операционной семантики — являются допущения, принимаемые при моделировании поведения конструкций целевого языка. Например, в рассмотренном тематическом исследовании результат операции + в C может отличаться от своего Lisp-аналога, используемого в операционной семантике на основе ABML.

В рамках дальнейшего развития данной работы предполагается решить две задачи. Во-первых, реализовать полноценный фреймворк для разработки операционной семантики языков программирования и применить его для построения семантики нескольких промышленно значимых языков — в первую очередь C, C++, C#, Java, Python, Go, Reflex, poST, ST, ONNX и Verilog. Во-вторых, адаптировать предложенный подход для поддержки построения аксиоматической семантики, что позволит создавать специализированные системы дедуктивной верификации, ориентированные на конкретные задачи или классы программ.

References

[1] I. S. Anureev, “A specification language for discrete dynamic systems based on ontologically structured knowledge”, *System Informatics*, no. 29, pp. 137–158, 2025, in Russian. DOI: 10.31144/si.2307-6410.2025.n29.p137-158

[2] C. Rhodes, “SBCL: A sanely-bootstrappable Common Lisp”, in *Workshop on Self-sustaining Systems*, 2008, pp. 74–86.

[3] N. Chappe, L. Henrio, and Y. Zakowski, “Monadic interpreters for concurrent memory models: Executable semantics of a concurrent subset of LLVM IR”, in *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2025, pp. 283–298.

[4] W. Steingartner and V. Novitzká, “Operational semantics in a domain-specific robot control language: A pedagogical use case”, *Computer Science and Information Systems*, vol. 21, no. 3, pp. 1077–1095, 2024.

[5] S. Goncharov, S. Milius, L. Schröder, S. Tsampas, and H. Urbat, “Towards a higher-order mathematical operational semantics”, *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 632–658, 2023.

[6] E. Mahe, C. Gaston, and P. L. Gall, “Equivalence of denotational and operational semantics for interaction languages”, in *International Symposium on Theoretical Aspects of Software Engineering*, 2022, pp. 113–130.

[7] G. Jouneaux, “Self-adaptable operational semantics”, Ph.D. dissertation, Université de Rennes, 2024. [Online]. Available: https://theses.hal.science/tel-05136513/

[8] J. Proença and L. Edixhoven, “Caos: A reusable SCALA web animator of operational semantics”, in *International Conference on Coordination Languages and Models*, 2023, pp. 163–171.

[9] Q. Chen et al., “The essence of Verilog: A tractable and tested operational semantics for Verilog”, *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 234–263, 2023.

[10] Z. Xu, H. Wang, and M. Sun, “Operational semantics for crystality: A smart contract language for parallel evms”, in *International Symposium on Theoretical Aspects of Software Engineering*, 2025, pp. 303–321.

[11] S. Rechenberger and T. Frühwirth, “A refined operational semantics for FreeCHR”, *Electronic Proceedings in Theoretical Computer Science*, vol. 439, pp. 278–293, 2026.

[12] D. Lee, M. Cho, J. Kim, S. Moon, Y. Song, and C.-K. Hur, “Fair operational semantics”, *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 811–834, 2023.

[13] G. P. Leemrijse, T. T. Franken, and T. Neele, “Formalisation of a new weak semantics for AuDaLa”, in *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*, 2024, pp. 93–116.

[14] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, “From sequential to multi-threaded Java: An event-based operational semantics”, in *Proceedings of the Algebraic Methodology and Software Technology: 6th International Conference*, 1997, pp. 75–90.

[15] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, “An event-based structural operational semantics of multi-threaded Java”, in *Formal syntax and semantics of Java*, Springer, 1999, pp. 157–200.

[16] S. Drossopoulou and S. Eisenbach, *Towards an operational semantics and proof of type soundness for Java*, https : / / www . researchgate . net / profile / Sophia - Drossopoulou / publication / 2274336_Towards_an_Operational_Semantics_and_Proof_of_Type_Soundness_for_Java / links / 0deec53398518f1a48000000 / Towards - an - Operational - Semantics - and - Proof - of - Type - Soundness-for-Java.pdf, 1998.

[17] E. Coscia and G. Reggio, *An operational semantics for Java*, https://www.researchgate.net/profile/Eva-Coscia/publication/2292715_An_Operational_Semantics_for_Java/links/004635150d44744df9000000/An-Operational-Semantics-for-Java.pdf, 1998.

[18] S. O. Wesonga, “Javalite — an operational semantics for modeling Java programs”, M.S. thesis, 2012.

[19] N. Belblidia and M. Debbabi, “A dynamic operational semantics for JVML”, *Journal of Object Technology*, vol. 6, no. 3, pp. 71–100, 2007.

[20] I. Attali, D. Caromel, and S. O. Ehmety, “An operational semantics for the Eiffel language”, INRIA, Tech. Rep. RR-2732, 1995.

[21] I. Attali, D. Caromel, and M. Russo, “A formal executable semantics for Java”, in *Proceedings of Formal Underpinnings of Java, an OOPSLA*, vol. 98, 1998, pp. 1–13.

[22] Y. Gurevich and J. K. Huggins, “The semantics of the C programming language”, in *International Workshop on Computer Science Logic*, 1992, pp. 274–308.

[23] C. Wallace, “The semantics of the C++ programming language”, in *Specification and validation methods*, Oxford University Press, Inc., 1995, pp. 131–164.

[24] J. Camilleri, "An operational semantics for OCCAM", *International Journal of Parallel Programming*, vol. 18, pp. 365–400, 1989.

[25] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 3, pp. 1–44, 2009.

[26] A. Verdejo and N. Marti-Oliet, "Executable structural operational semantics in Maude", *The Journal of Logic and Algebraic Programming*, vol. 67, no. 1-2, pp. 226–293, 2006.

[27] T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs, "A linear operational semantics for termination and complexity analysis of ISO Prolog", in *International Symposium on Logic-Based Program Synthesis and Transformation*, 2011, pp. 237–252.

[28] G. Roșu and T. F. Șerbănuță, "An overview of the K semantic framework", *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[29] T. Nipkow and G. Klein, *Concrete semantics: with Isabelle/HOL*. Springer, 2014, 298 pp.

[30] V. A. Nepomniaschy, I. S. Anureev, I. Mikhailov, and A. Promskii, "Towards verification of C programs. C-light language and its formal semantics", *Programming and Computer Software*, vol. 28, pp. 314–323, 2002.

[31] M. Norrish, "C formalised in HOL", University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-453, 1998.

[32] P. Kelsen and Q. Ma, "A lightweight approach for defining the formal semantics of a modeling language", in *International Conference on Model Driven Engineering Languages and Systems*, 2008, pp. 690–704.

[33] G. J. Smeding, "An executable operational semantics for Python", M.S. thesis, Universiteit Utrecht, 2009. [Online]. Available: http://gideon.smdng.nl/wp-content/uploads/thesis.pdf

[34] M. A. Köhl, *An executable structural operational formal semantics for Python*, 2021. arXiv: 2109.03139 [cs.PL].

[35] S. Foster, C.-K. Hur, and J. Woodcock, "Unifying model execution and deductive verification with interaction trees in Isabelle/HOL", *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–40, 2025. DOI: 10.1145/3702981

[36] F. Sheng, H. Zhu, J. He, Z. Yang, and J. P. Bowen, "Theoretical and practical aspects of linking operational and algebraic semantics for MDESL", *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 1–46, 2019. DOI: 10.1145/3295699

[37] Y. Huang, X. Bu, G. Zhu, X. Ye, X. Zhu, and J. Shi, "KST: Executable formal semantics of IEC 61131-3 structured text for verification", *IEEE Access*, vol. 7, pp. 14 593–14 602, 2019. DOI: 10.1109/ACCESS.2019.2894026

[38] D. Youn et al., "Bringing the WebAssembly standard up to speed with SpecTec", vol. 8, no. PLDI, p. 210, 2024. DOI: 10.1145/3656440

[39] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics", in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, 2014, pp. 175–188. DOI: 10.1145/2628136.2628143