MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

THEORY OF SOFTWARE

Model Checking Programs in Process-Oriented IEC 61131-3 Structured Text

N. O. Garanina[1], S. M. Staroletov[1], V. E. Zyubin[1], I. S. Anureev[1]

DOI: 10.18255/1818-1015-2024-1-32-53

[1]Institute of Automation and Electrometry SB RAS, Novosibirsk, Russia

MSC2020: 68N30
Research article
Full text in Russian

Received January 17, 2024
Revised February 6, 2024
Accepted February 14, 2024

The process-oriented programming is a paradigm based on the process concept where each process is a concurrent finite state machine inside. The paradigm is intended for PLC (programmable logic controllers) developers to write Industry 4.0-enabled software. The poST language is a promising process-oriented extension of the IEC 61131-3 Structured Text (ST) language designed to provide a conceptual consistency of the PLC source code with technological description of the process under control. This language combines the advantages of FSM-based programming with the standard syntax of the ST language. We propose transformational semantics of poST providing rules for translation of poST language statements to Promela — the input language of the SPIN model checker. Following these semantic rules, our Xtext-based translator outputs a Promela model for the poST program. Our contribution is the poST transformational semantics and the method for automatic generation of the Promela code from poST control programs. The resulting Promela program is ready to be verified with SPIN model checker against linear temporal logic requirements to the source poST program.

In the paper we provide an overview of related work, as well as a brief description of the poST and Promela languages. Further, the Promela poST translation rules cover control flow statements, process creation and state management constructs, and timeout management. Then we define service processes for modeling the external environment and managing high-level LTL specifications. Then we present the main ideas of implementing the translator poST to Promela. We also illustrate our approach using the example of a system for managing electricity consumption and production, including renewable sources.

Keywords: control software; model checking; process-oriented programming; LTL; SPIN; Structured Text

INFORMATION ABOUT THE AUTHORS

Garanina, Natalia O. (corresponding author)	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Senior researcher, PhD
Staroletov, Sergey M.	ORCID iD: 0000-0001-5183-9736. E-mail: serg_soft@mail.ru Senior researcher, PhD
Zyubin, Vladimir E.	ORCID iD: 0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su Head of Laboratory, Dr. Sc.
Anureev, Igor S.	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@iis.nsk.su Senior researcher, PhD

Funding: State task IAaE SB RAS, project No. 122031600173-8, state task No. AAAA-A19-119120290056-0.

For citation: N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, "Model checking programs in process-oriented IEC 61131-3 Structured Text", *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 32–53, 2024. DOI: 10.18255/1818-1015-2024-1-32-53.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

THEORY OF SOFTWARE

Верификация моделей программ на процесс-ориентированном расширении языка Structured Text стандарта IEC 61131-3

Н. О. Гаранина[1], С. М. Старолетов[1], В. Е. Зюбин[1], И. С. Ануреев[1]

DOI: 10.18255/1818-1015-2024-1-32-53

[1]Институт автоматики и электрометрии СО РАН, Новосибирск, Россия

УДК 004.822+681.51
Научная статья
Полный текст на русском языке

Получена 17 января 2024 г.
После доработки 6 февраля 2024 г.
Принята к публикации 14 февраля 2024 г.

Процессно-ориентированное программирование — это парадигма, основанная на концепции процесса. Каждый процесс представляет собой конечный автомат. Эта парадигма предназначена для разработчиков ПЛК (программируемых логических контроллеров) для написания программного обеспечения с поддержкой Индустрии 4.0. Язык poST является многообещающим процессно-ориентированным расширением языка структурированного текста (ST) МЭК 61131-3, предназначенным для обеспечения концептуальной согласованности исходного кода ПЛК с технологическим описанием управляемого процесса. Этот язык сочетает в себе преимущества программирования на основе конечных автоматов со стандартным синтаксисом языка ST. Мы предлагаем трансформационную семантику poST, заданную правилами перевода операторов языка poST в Promela — входной язык средства проверки моделей SPIN. Следуя этим правилам, наш транслятор, основанный на технологии Xtext, строит модель Promela для программы poST.

Основной вклад нашей статьи — это трансформационная семантика poST и метод автоматической генерации кода Promela из программ управления PoST. Полученная модель Promela готова к проверке с помощью системы верификации моделей SPIN на соответствие требованиям к исходной программе poST, выраженных в терминах линейной темпоральной логики LTL. В статье мы приводим обзор связанных работ, а также краткое описание языков poST и Promela. Представленные далее правила трансляции poST в Promela покрывают операторы потока управления, конструкции создания процессов и управления их состояниями, а также операторы для таймаутов. Отдельно определены сервисные процессы для моделирования внешней среды и задания высокоуровневых LTL спецификаций. Затем мы останавливаемся на основных идеях реализации транслятора poST в Promela, и далее иллюстрируем наш подход на примере системы управления потреблением и производством электроэнергии, в том числе из возобновляемых источников.

Ключевые слова: управляющее программное обеспечение; проверка моделей; процесс-ориентированное программирование; LTL; SPIN; Structured Text

ИНФОРМАЦИЯ ОБ АВТОРАХ

Гаранина, Наталья Олеговна (автор для корреспонденции)	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Старший научный сотрудник, к.ф.-м.н.
Старолетов, Сергей Михайлович	ORCID iD: 0000-0001-5183-9736. E-mail: serg_soft@mail.ru Старший научный сотрудник, к.ф.-м.н.
Зюбин, Владимир Евгеньевич	ORCID iD: 0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su Заведующий лаборатории, д.т.н.
Ануреев, Игорь Сергеевич	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@iis.nsk.su Старший научный сотрудник, к.ф.-м.н.

Финансирование: Госзадание ИАиЭ СО РАН, проект № 122031600173-8, госзадание № АААА-А19-119120290056-0.

Для цитирования: N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, “Model checking programs in process-oriented IEC 61131-3 Structured Text”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 32–53, 2024. DOI: 10.18255/1818-1015-2024-1-32-53.

Введение

При значительном прогрессе в формальных методах доказательства моделей программ остается вопрос их применимости к реальному программному обеспечению. Можно констатировать, что применение таких методов к произвольным программам не представляется целесообразным из-за трудоемкости перевода языковых конструкций с неоднозначной семантикой, большого пространства состояний, неопределенности требований или сложности их задания.

С другой стороны, существует такой класс, как программы для ПЛК (Программируемые Логические Контроллеры), предназначенные для реализации алгоритмов управления. Цена ошибки в управляющем ПО высока, что указывает на то, что управляющие программы нуждаются в формализации и последующем доказательстве их корректности относительно требований, которые также нужно формализовать. Программы на языках для ПЛК лаконичны, сами языки содержат небольшое количество конструкций, а требования выражаются с помощью небольшого количества ключевых переменных. Эти особенности языков ПЛК делают применение формальных методов для них более перспективным.

В настоящее время для программируемых промышленных контроллеров используются языки стандарта МЭК 61131-3 [1]. Они различаются представлением программы: в виде текста, лестничных схем, функциональных блок-схем и списков команд. В частности, широкое распространение имеет язык структурированного текста (Structured Text, ST) [2]. Этот язык похож на Паскаль и имеет простой синтаксис с дополнительными конструкциями (например, таймаутами и интервалами), специфичными для программ ПЛК. Однако большинство программ управления цикличны и зависят от состояния, что приводит к большому количеству операторов переключения в тексте программы.

Чтобы избежать громоздкости текста программ для ПЛК в работе [3] мы предложили использовать понятие процесса как основной логической сущности программы для предоставления разработчикам удобных средств работы с состояниями и переходами. Мы называем программирование по этой методологии процесс-ориентированным, а язык, который включает средства работы с процессами и их состояниями, — процесс-ориентированным языком программирования. Разработанный нами язык poST (process-oriented Structured Text) [4] является одним из процесс-ориентированных языков. Он расширяет стандартизированный язык ST, для которого имеются среды разработки, моделирования и создания встроенного ПО реальных контроллеров.

Использование формальных методов верификации программ ПЛК оправдано прежде всего тем, что такие программы могут работать с дорогостоящим оборудованием предприятий и некорректное поведение программы может привести как к финансовым потерям, так и к серьезным последствиям для окружающей среды и персонала. Поэтому необходимо предусмотреть как можно большее разнообразие средств верификации таких программ. Поэтому в нашей работе мы предлагаем трансляцию из языка poST во входной язык инструмента верификации SPIN [5]. Эту трансляцию можно также назвать транспиляцией, поскольку она использует только исходные коды программ, и в настоящее время это удобный способ переключения между различными языками [6].

Инструмент SPIN использует один из формальных методов — проверку моделей [7]. Обзор практики применения проверки моделей представлен в статье Оватмана и др. [8]. Метод проверки моделей, применяемый в SPIN, состоит в следующем: на вход системе проверки подаётся формальное описание конечной системы переходов и требование корректности этой системы, представленное в виде темпоральной формулы; по системе переходов и требованию строится автомат Бюхи (композиция системы и отрицания требования), язык которого методом DFS или BFS проверяется на непустоту. Если находится допустимый путь в автомате-композиции — это означает, что язык композиции системы и отрицания требования не пуст, и найденный путь (трасса) служит контрпримером, т. е. историей изменения значений переменных системы, которая приводит

к нарушению требования. Начиная с первых работ Кларка [9], в качестве источника системы переходов предлагалось использовать формализованный язык, из которого система переходов получается единственным способом. Инструмент SPIN отвечает всем нашим требованиям: (1) имеет формальный входной язык для описания моделей; (2) требования выражаются в виде формул линейной темпоральной логики LTL [7] над ключевыми переменными программы; (3) инструмент одобрен сообществом, используется в большом количестве реальных проектов и реализует эффективные алгоритмы проверки моделей с большим пространством состояний.

Входным языком SPIN является Promela (Protocol Meta-Language), который наследует семантику алгебры процессов CSP (Communicating Sequential Processes) [10], расширяя её некоторыми акторными возможностями. Он описывает параллельные и распределенные системы как взаимодействующие процессы. В силу близости этой парадигмы к процесс-ориентированному подходу можно было бы описывать процесс-ориентированные системы управления сразу на Promela, однако в этом языке нет средств работы с состояниями процессов и таймаутами. Хотя состояния, процессы и переключение между ними могут быть представлены с помощью условных конструкций, дополнительных переменных и передачи активирующих сообщений, это делает код более громоздким и запутанным. Кроме того, возникает необходимость разработки для Promela компилятора или транслятора в ST.

В этой статье мы описываем правила трансформации кода из poST в Promela, а также вопросы реализации транспайлера poST2Promela в соответствии с этими правилами. Реализация представляет собой решение, основанное на инструменте синтаксического анализа Eclipse Xtext на Java и Xtend. Мы добавили обзор связанных работ, пример трансляции и детали реализации к предварительной версии работы, опубликованной в [11].

Оставшаяся часть статьи имеет следующую структуру. В разделе 1 мы обсуждаем современное состояние в области создания моделей Promela для верификации различных свойств программных систем. В разделе 2 мы рассматриваем особенности языков poST и Promela. Раздел 3 определяет правила трансформационной семантики языка poST. Вопросы реализации обсуждаются в разделе 4. Процесс трансформации продемонстрирован на примере солнечной электростанции в разделе 5, а выводы приведены в Заключении.

1. Обзор связанных работ

Тема трансляции разнообразных программных моделей и программ во входные языки верификаторов моделей, в частности в Promela, довольно популярна и в разное время создавались трансляторы программ на различных языках для их последующей верификации. Рассмотрим некоторые из них.

В статье [12] предлагается трансформация предметно-ориентированного языка управления роботами в Promela. Поскольку конструкции языка относительно просты и язык не допускает даже рекурсии, перевод на Promela относительно прост: 1) получение AST (абстрактного синтаксического дерева) из программы управления роботом, 2) получение CFG (графа потока управления) из AST, 3) трансляция CFG в полученный код. Перед этим выполняется процесс абстрактной интерпретации [13], чтобы избавиться от символьных переменных и оптимизировать CFG. Авторы также используют специальный процесс PLC для координации между роботами, а требования задаются матрицами возможных запросов от пользователя. Все необходимые конструкции выражаются в Promela. Процесс трансляции не формализован, и при этом авторы отмечают, что абстрактная интерпретация и оптимизация могут привести к неправильному генерированию программ, если в исходной программе присутствуют некоторые типы goto-переходов.

В статье [14] Леу и Хольцманн представили концепцию визуального языка v-Promela, который генерирует код Promela из моделей, созданных пользователями в графической среде Hybrid FSM. Вводится понятие капсулы как логического состояния с возможностью описания связей, сообщений между капсулами, при этом поддерживается декомпозиция. В этом случае состояния преобразуются в метки в коде Promela с генерацией встроенных функций для поддержки семантики входа и выхода из состояния. Для обеспечения корректности переходов используются конструкции `d_step` и `atomic`, чтобы сгенерированный код согласно модели работал как положено во всех допустимых ситуациях чередования. При этом семантика языка не формализована. Похожий подход можно наблюдать в статье Бенерекетти и др. [15], где код Promela генерируется из расширенного представления автомата, в данном случае из динамических конечных автоматов, путем определения семантики операторов (частей) конечного автомата и последующего описания алгоритма генерации кода в виде текста алгоритма, который представляет собой лишь полуформальное описание.

Лион и др. [16] предложили подход с преобразованием кода из языка описания протоколов Reo в код Promela для проверки LTL-требований к протоколам. Поскольку в Reo есть концепция портов, связь через них транслируется по каналам, встроенным в язык Promela, и для каждого порта используется два канала: один для данных, другой для синхронизации. Поскольку протокол Reo описывается как агент, выполняющий последовательность защищенных условий и действий (их также можно получить из графических схем), авторы формально определяют семантику преобразования спецификации таких условий и действий в код, где каждый агент — это процесс Promela.

Кларл [17] предлагает язык Helena — предметно-ориентированную спецификацию агентов и их отношений (например, для описания протоколов p2p) в редакторе на основе Eclipse. Предлагаются алгоритмы перевода в Promela для дальнейшей проверки свойств, выраженных в LTL\X (LTL без оператора neXt). Сам перевод описан неформально и для одного из операторов языка дан шаблон Xtend, поскольку для преобразования используется структура Xtext. По используемым инструментам трансляции наша работа близка к этой.

В работах [18, 19] Дилли и Ланге предлагают подход Gomela проверки программ на подмножестве языка Go путем генерации соответствующего кода на Promela. Проверяется корректность функций передачи сообщений. Преобразование описывается в функциональном стиле с использованием рекурсивной функции TransStmts, которая переводит все операторы MiniGo в конструкции Promela. В результате по прилагаемым бенчмаркам можно проверить корректность простых алгоритмов, а не готовых программ.

Работу Бринксмы, Мадера и Фенкера [20] можно считать новаторской работой по верификации программ ПЛК. Здесь впервые была введена формальная концепция цикла управления ПЛК. На примере бетонного завода строится модель процесса управления в Promela и требования к ней выражаются в LTL. Однако модель системы не генерируется автоматически, а создается вручную.

Что касается верификации программ на языках семейства 61131, то работа [21] предлагает получить промежуточный код Promela для блоков FBD (Function Block Diagram) с целью их дальнейшей проверки. Поскольку код предполагается генерировать по исходному представлению графа, сначала выполняется топологическая сортировка графа. Применимость метода продемонстрирована на одном примере [22].

Формальное моделирование и верификация программ ПЛК на основе схем МЭК 61499 представлены в статьях Лях и др. [23], а также Шатрова и Вяткина [24]. Они используют системы верификации NuSMV и SPIN, и отмечают более короткое время проверки и большую пригодность для циклических моделей как преимущества системы SPIN.

Недавняя статья Эбненасира [25] направлена на отказоустойчивый анализ программ ПЛК, выраженных в виде схем лестничной логики. Представлены не до конца уточненные способы транс-

ляции языковых конструкций (импульс, инструкции установки/сброса, времени) в конструкции Promela. В качестве источника требований предлагается использовать временные диаграммы, по которым автор генерирует простые последовательные отношения с помощью логики LTL. Автор не приводит примеры использования этого подхода для спецификации и верификации требований к реальным программам.

Подводя итог вышесказанному, большинство этих статей в основном являются работами proof-of-the-concept или диссертациями, и их применимость в индустрии неизвестна. Как правило, в рассматриваемых статьях недостаточно проработаны сложные вопросы, связанные с особенностью семантики программ ПЛК, а требования к примерам программ не выражены в терминах циклов управления ПЛК [26]. В нашем подходе мы используем предметно-ориентированный язык, удобный для выражения процессов и их состояний, создаем проекции Promela для всех синтаксических конструкций этого языка и детально описываем условия возможности трансляции poST-программ для ПЛК в язык Promela.

2. Языки poST и Promela

Язык poST представляет собой процесс-ориентированное расширение языка ST МЭК 61131-3, которое обеспечивает концептуальную согласованность исходного кода ПЛК с технологическим описанием управляемого процесса. Язык сочетает в себе преимущества программирования на основе конечных систем переходов с традиционным синтаксисом языка ST, что облегчает его внедрение. В основе его семантики лежит понятие гиперпроцесса [3].

Программа poST, отражая цикл управления ПЛК, включает в себя процессы, активность которых организована в цикл в порядке их появления в программном коде. Эта схема предполагает модели ПЛК, которые абстрагируются от времени цикла управления [26], при этом само время цикла управления задано с помощью параметра INTERVAL. Каждый процесс определяется упорядоченным набором своих состояний. Когда в цикле управления очередь активации переходит к процессу, он выполняет конечную последовательность действий, связанную с его текущим состоянием. Для описания этих действий процесса мы используем стандартные конструкции ST (объявления переменных, операторы потока управления и т. д.) и специфические процесс-ориентированные функции: операторы управления состояниями процесса (START/STOP PROCESS, SET NEXT и другие) и операторы таймаута. Взаимодействие процессов в poST происходит через общие переменные. Семантика языка poST предполагает автоматическую реализацию низкоуровневых конструкций для отображения сигналов ввода-вывода на программные переменные, состояний процесса, операторов таймаута и цикла управления, имеющего определенную длительность, заданную в INTERVAL. Грамматика языка poST в формате Xtext доступна в репозитории[1].

Язык Promela[2] используется для описания параллельных процессов, основанных на формализме CSP. Программа на Promela состоит из параллельных процессов, исполнение которых реализует семантику чередования. Чередование может быть ограничено операторами `atomic` и `d_step`, которые не позволяют прерывать последовательность действий процесса. В Promela переменные могут принимать значения следующих конечных типов: логические, перечислимые, целочисленные значения (в диапазоне от $-(2^{31}-1)$ до $2^{31}-1$). Процессы Promela взаимодействуют через общие переменные, а также через каналы сообщений, реализующие синхронные и асинхронные способы связи. Язык Promela включает блокирующие операторы потока управления `if` и `do`, в то время как аналогичные операторы потока управления poST являются неблокирующими. Программа на Promela может быть проверена на соответствие требованиям, сформулированным средствами

[1] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/blob/main/poST_grammar.xtext
[2] http://spinroot.com/spin/Man/grammar.html

темпоральной логики LTL, с помощью инструмента проверки моделей SPIN [5]. Для проверки этот инструмент использует композицию автоматов Бюхи для программы и формулы, поэтому в Promela возможны только конечные типы переменных модели. Следовательно, типы poST, представляющие действительные числа, не могут быть напрямую преобразованы в типы Promela, и в настоящей работе мы ограничиваемся программами poST, которые не содержат переменные действительных типов и библиотечные функции, возвращающие действительные значения. Ещё одна особенность poST, требующая специального внимания при трансляции — это работа с таймаутами, поскольку в Promela нет переменных типа времени.

Язык poST определяет алгоритмы управления, которые взаимодействуют с некоторой средой. Мы предполагаем, что исходный код poST кроме управляющей программы содержит и программу управляемого объекта для целей анализа поведения системы управления, которую они составляют. Транслируя код обеих программ, мы строим модель Promela, соответствующую порядку активации процессов, структуре состояний процессов, управлению таймаутами и типам переменных с точностью до абстрагирования от типов poST, представляющих действительные числа.

Сформулируем следующие принципы преобразования poST-программ в Promela-модели:

1. **Переменные**. Каждая переменная poST-программы имеет взаимно-однозначное соответствие некоторой переменной Promela-модели. Для каждого процесса p определена локальная переменная его состояния $state_p$. Она принимает значения перечислимого типа $State_p$, включающего специальные состояния *Stop* и *Error*. Глобальная переменная $time_p$ объявляется для каждого процесса p, в poST-коде которого есть операторы таймаута.
2. **Процессы**. Процесс poST p отображается в Promela-процесс p, бесконечно выполняющийся в цикле `do-od`. Пребывание процесса p в состоянии $s \in State_p$ задается с помощью оператора $(state_p == s)$, за которым следует тело состояния s. Состояние процесса p изменяется путем присвоения переменной $state_p$ значения из $State_p$. Тело Promela-процесса p состоит из условий пребывания в состояниях и их тел. Это тело заключено в оператор `atomic`, чтобы гарантировать последовательное выполнение действий согласно семантике poST.
3. **Таймауты**. В случае, если poST-процесс p выполняет действия по таймауту T, в соответствующем Promela-процессе запуск (сброс) связанного таймера $time_p$ транслируется в присваивание $time_p = 0$, и выполнение этих действий ограничено защитным условием $(time_p == T_p)$. Значения таймера увеличиваются на единицу в каждом цикле, пока не достигнут значения T_p или не произойдёт сброс таймера. При вычислении значения T_p используется значение соответствующего таймаута и значение INTERVAL.
4. **Синхронизация**. Существует несколько способов моделирования последовательного выполнения процессов poST в Promela. Один из них основан на передаче активирующего сообщения от процесса к процессу по каналу *cur* единичной емкости. Это сообщение содержит уникальный идентификатор процесса. После завершения действий в текущем состоянии функции процесс p передает ход процессу q, отправляя сообщение q в канал *cur*. Процесс q начинает выполняться после получения этого сообщения. Благодаря блокирующей семантике Promela-выражений последовательное выполнение процессов обеспечивается тем, что процесс может запуститься только после получения сообщения со своим именем.
5. **Моделирование входных данных**. Входные данные для алгоритмов управления поступают из трех источников: данные внешней среды, которые могут не зависеть от работы алгоритма управления, данные управляемого объекта, которые зависят от выходов алгоритма управления, а также выходные данные процессов, обеспечивающие их внутреннее взаимодействие. Сервисный Promela-процесс `Gremlin` моделирует данные неуправляемой среды. Этот процесс присваивает случайные значения соответствующим входным переменным. Мо-

делирование данных объекта управления и выходных данных других процессов задается процессами Promela, соответствующими их poST-коду, при этом сервисный Promela-процесс `OutInput` корректно сопоставляет имена входных и выходных данных процессов.

6. **Структура выполнения итоговой Promela-модели**. Согласно семантике poST, передача активирующего сообщения в результирующей Promela-модели организована следующим образом. Сначала входные данные обновляются процессом `Gremlin`, затем процессом `OutInput`. Затем очередь переходит к процессам контроллера, которые образуют последовательность выполнения посредством передачи активирующих сообщений. Последний процесс контроллера снова передает ход процессу `Gremlin`.

В следующем разделе правила трансляции конструкций языка poST в конструкции языка Promela описаны более подробно.

3. Правила трансляции языка poST в язык Promela

3.1. Именование, типы, объявления и операции

В языке poST есть глобальное пространство имен, пространство имен программ и процессов, тогда как в Promela есть только глобальное пространство имен и пространства имен процессов. Чтобы избежать конфликтов имен, мы формируем полные имена объектов в модели Promela программы poST, учитывая (1) имя объекта; (2) вид объекта (процесс, переменная и т. д.); и (3) имя процесса или программы, в которой находится объект. Чтобы улучшить читаемость программы, мы используем режим именования по умолчанию, который учитывает только вид объекта и может добавить порядковый номер, если в глобальном пространстве имен Promela есть несколько одинаковых имен.

Типы переменных в большинстве случаев транслируются тривиально. Здесь мы рассматриваем poST-программы без переменных действительных типов, поскольку в Promela нет типов, представляющих действительные числа, а абстракция типов данных выходит за рамки данной статьи. Трансляция типа TIME обсуждается ниже при описании трансляции выражения TIMEOUT.

В соответствии с политикой именования и трансляции типов, приведенными выше, все переименованные переменные poST объявляются в модели Promela как глобальные переменные, а константы poST тривиально транслируются с помощью директивы Promela `#define`.

Promela включает в себя те же арифметические и логические операции, что и poST, за исключением возведения в степень, которое можно смоделировать операциями побитового сдвига Promela.

3.2. Операторы потока управления

В таблице 1 мы приводим метод трансляции трех видов операторов потока управления poST в код Promela. Пусть *code′* — образ Promela для poST *code*, созданный нашим алгоритмом трансляции. В силу блокирующей семантики Promela и неблокирующей семантики poST для операторов потока управления, необходимо использовать ветку `else` и оператор `skip` при трансляции poST оператора IF, поскольку Promela-оператор `if` блокирует исполнение процесса, если условие *cond* ложно. По семантике Promela, ветвь `else` выбирается, когда другие условия в операторе `if` невыполнимы. Оператор `skip` в этой ветке не выполняет никаких действий. poST оператор CASE оценивает выражение *expr* (тип INT) и выполняет ветвь, соответствующую списку значений, содержащему результат вычисления. Списки значений не пересекаются. Promela-оператор `do` также является блокирующим, поэтому при трансляции poST оператора DO мы добавляем ветвь `else` с оператором `break` для моделирования завершения цикла в Promela.

3.3. Специфические операторы процесс-ориентированных программ

Система управления в poST задается набором poST-программ. Следовательно, нам необходимо перевести несколько poST-программ в единую Promela-модель. Программа на poST состоит из про-

Table 1. Control-Flow statements **Таблица 1.** Операторы потока управления

poST	Promela	poST	Promela

IF *cond* THEN
  body
END_IF
	if
	:: *cond′* -> { *body′* }
	:: else -> skip;
	fi;
		CASE *expr* OF
		  $list_1$: $body_1$
		...
		  $list_n$: $body_n$
		ELSE
		  body
		END_CASE
			int v = *expr′*;
			if
			:: v == l_{11}||...||v == l_{1m_1} ->
			  { $body'_1$ }
			...
			:: v == l_{n1}||...||v == l_{nm_n} ->
			  { $body'_n$ }
			:: else ->
			  { *body′* }
			fi;

WHILE *cond* DO *body* END_WHILE	do :: *cond′* -> { *body′* } :: else -> break; od;		

цессов, которые циклически активируются один за другим. Несколько poST-программ запускаются в едином цикле управления в порядке их появления в коде. В Promela мы моделируем этот циклический запуск, последовательно передавая *активирующее сообщение* от процесса к процессу через Promela-канал в том порядке, в котором poST-процессы появляются в коде исходных программ. Promela-каналы поддерживают блокировку чтения и записи. Мы используем канал cur емкостью 1 для передачи псевдонимов процессов в активирующих сообщениях. В результирующей Promela-программе каждый процесс заблокирован до тех пор, пока не прочтет свой псевдоним из этого канала. После выполнения своего тела процесс передает в канал псевдоним следующего процесса для его активации. Promela-процесс для последнего poST-процесса передаёт ход в сервисный процесс Gremlin, описанный в разделе 3.4, который является первым процессом результирующей модели Promela. Следуя семантике последовательной активации poST-программ мы используем Promela-оператор atomic для тела каждого транслируемого poST-процесса. Этот оператор уменьшает степень чередования исполнения в Promela-программе, что значительно упрощает верификацию. В левом блоке таблицы 2 представлена трансляция верхней структуры процессов poST-программ.

Тело poST-процесса состоит из *состояний*. На каждой итерации цикла управления poST-процесс выполняет код, соответствующий некоторым его состояниям, за исключением состояний STOP и ERROR, когда он ничего не делает. Для каждого транслируемого процесса n с Promela именем n' мы используем специальный счетчик состояний c_n', чтобы сохранить имя его текущего состояния. В начале программы poST её первый объявленный процесс находится в своем первом состоянии, а все остальные процессы находятся в состоянии STOP. В правом блоке таблицы 2 приводится трансляция отдельного poST-процесса в язык Promela.

poST-процессы могут проверять статус активности других процессов с помощью операторов ACTIVE и INACTIVE. Также каждый процесс может перевести себя или другой процесс в другое состояние с помощью операторов RESTART, STOP, START PROCESS, STOP PROCESS и других. Отметим, что имеет место инкапсуляция процессов в том смысле, что другой процесс можно только остановить или запустить, а доступа к отдельным именованным состояниям посторонние процессы не имеют. Перечисленные операторы poST тривиально транслируются в Promela, если целевое состояние не включает оператор TIMEOUT. См. примеры в левом блоке таблицы 3.

Последним оператором состояния poST-процесса может быть оператор TIMEOUT. Инструкции тела этого оператора выполняются по истечении указанного времени с момента перехода процесса в это состояние. Чтобы смоделировать такое поведение в Promela, мы вводим переменную счетчика времени t_n' — по одной на каждый процесс n, содержащий состояния с TIMEOUT. Если процесс

Table 2. Process-oriented features

Таблица 2. Процессно-ориентированные структуры

poST	Promela	poST	Promela

PROGRAM $prog_1$
PROCESS n_{11}
$body_{11}$
END_PROCESS
...
PROCESS n_{n1}
$body_{n1}$
END_PROCESS
END_PROGRAM
...
PROGRAM $prog_m$
PROCESS n_{1m}
$body_{1m}$
END_PROCESS
...
PROCESS n_{nm}
$body_{nm}$
END_PROCESS
END_PROGRAM
	mtype : P =
	{p_n'_{11}, ... p_n'_{nm}}
	chan cur=[1] of {mtype:P}
	active proctype n'_{11}() {
	do :: cur ? p_n'_{11} ->
	atomic {
	$body'_{11}$;
	cur ! p_n'_{21};}
	od;
	}
	active proctype n'_{12}() {
	do :: cur ? p_n'_{21} ->
	atomic {
	$body'_{12}$;
	cur ! p_n'_{31};}
	od;
	}
	...
	active proctype n'_{nm}() {
	do :: cur ? p_n'_{nm} ->
	atomic {
	$body'_{nm}$;
	cur ! Gremlin;}
	od;
	}
		PROCESS n
		STATE s_1
		$body_1$
		END_STATE
		...
		STATE s_m
		$body_m$
		END_STATE
		END_PROCESS
			mtype:S_n' = s_s'_1,...
			s_s'_n,s_$Stop'$,s_$Error'$}
			mtype:S_n' c_n';
			active proctype n'() {
			c_n' = s_$Stop'$;
			do :: cur ? p_n' ->
			atomic {
			if
			:: c_n'==s_s'_1 ->
			{ $body'_1$ }
			...
			:: c_n'==s_s'_n ->
			{ $body'_n$ }
			:: else -> skip;
			fi;
			cur ! nxt_p;}
			od;
			}

находится в состоянии с таймаутом, то в каждом цикле управления его счетчик времени увеличивается на единицу. Согласно семантике poST, этот счетчик времени обнуляется, когда (1) процесс сбрасывает таймер (RESET); (2) процесс переходит в другое состояние; и (3) происходит таймаут. Следуя семантике poST, мы используем отношение > в Promela-операторе `if` для таймаута, поскольку выполнение блока таймаута начинается в следующем цикле после истечения времени таймаута. Мы приводим типичные конструкции модели для таймаутов в правом блоке таблицы 3.

Чтобы уменьшить размер Promela-модели, мы проводим в ней следующую оптимизацию счетчиков времени. Во-первых, мы используем значение INTERVAL, которое задаёт физическое время исполнения цикла управления в poST, чтобы уменьшить все значения таймаута до ближайшего кратного этому интервалу. Во-вторых, мы делим все значения таймаутов poST-программы на их наибольший общий делитель. Кроме того, для счетчиков времени выбираем минимально достаточный размер *nb* беззнакового типа. Например, мы добавляем один счетчик времени размером 4 бита, если результирующий Promela-процесс имеет два состояния с таймаутами, отсчитывающими 5 (101b) и 9 (1001b) итераций цикла управления.

Table 3. State and Timeout Statements **Таблица 3.** Операторы состояний и таймаутов

poST	Promela	poST	Promela

IF (PROCESS n INACTIVE)
THEN $body$
END_IF
	if
	:: c_n' == s_$Stop'$ ||
	c_n' == s_$Error'$ ->{ $body'$ }
	:: else -> skip;
	fi;
		PROCESS n_1
		STATE s_1
		$body_1$
		START PROCESS n_2
		END_STATE
		...
		END_PROCESS
			active proctype n_1'(){
			...
			:: c_n_1' == s_s_1' -> {
			$body_1'$
			c_n_2' = s_2';
			t_n_2' = 0; }
			...
			}
			unsigned t_n_2' : nb

PROCESS n
STATE s_1
$body_1$
SET NEXT
END_STATE
STATE s_2
$body_2$
END_STATE
...
END_PROCESS
	active proctype n'() {
	do :: cur ? p_n' ->atomic {
	if
	:: c_n' == s_s_1' ->{ $body_1'$
	c_n' = s_s_2'; }
	:: c_n' == s_s_2' ->{ $body_2'$ }
	...
	:: else -> skip;
	fi;
	cur ! nxt_p;}
	od;
	}
		PROCESS n_2
		STATE s_2
		$body_2$
		TIMEOUT T#$tout$
		THEN $body_t$
		END_TIMEOUT
		END_STATE
		...
		END_PROCESS
			active proctype n_2'(){
			...
			:: c_n_2' == s_s_2' -> {
			if
			:: t_n_2' > $tout'$ ->$body_t'$
			:: else -> t_n_2'++;
			fi;}
			...
			}

3.4. Сервисные процессы

Мы вводим три специальных процесса Promela: процесс `BOC` для маркера начала цикла управления, процесс `Gremlin` для моделирования неопределенного поведения среды и процесс `OutInput` для корректного взаимодействия программ poST.

Требования к функционированию реактивных систем, определенные для poST-программ, часто естественным образом выражаются в терминах взаимосвязи между входными и выходными данными программы. Соответственно, проверка таких высокоуровневых требований происходит *в промежутке* между циклами управления системы, а программные процессы рассматриваются как черные ящики. Для обеспечения такой проверки мы вводим сервисный Promela-процесс `BOC`, который фиксирует начало цикла управления для проверки требований. Используя значение «импульсной» переменной `cycle_u` этого процесса, мы формулируем аналоги модальностей LTL — *циклические темпоральные операторы* — для высокоуровневых требований в Promela, как показано в таблице 4. Эти темпоральные операторы использует логика cycle-LTL, разработанная нами в [27].

Разберём, как используется импульсный сигнал `cycle_u` на примере оператора `G_cltl`. В [27] доказано, что $\mathbf{G}^{\mathbf{i}}\varphi \equiv \mathbf{G}(\mathit{Input} \rightarrow \varphi)$, где $\mathbf{G}^{\mathbf{i}}$ — циклический глобальный оператор, $\mathbf{G}$ — стандартный глобальный LTL-оператор и Input — булева переменная, истинная только в начале цикла управления. В Promela-коде таблицы 4 *Input* представлен импульсным сигналом `cycle_u`. Таким образом макрос `c_imp(expr)` кодирует импликацию $(\mathit{Input} \rightarrow \varphi)$, где φ соответствует `(expr)`. Поэтому при подстановке `c_imp(expr)` в `G_cltl(expr)` получаем: `G_cltl(expr) [] c_imp(expr) = [](cycle_u ->`

Table 4. High-level LTL modalities and service process BOC

Таблица 4. Высокоуровневые модальности LTL и сервисный процесс BOC

LTL модальности	Процесс BOC

`#define c_imp(expr) (cycle_u -> (expr))`
`#define c_and(expr) (cycle_u && (expr))`
`#define G_cltl(expr) [] c_imp(expr)`
`#define F_cltl(expr) <> c_and(expr)`
`#define U_cltl(expr1, expr2)`
`  c_imp(expr1) U c_and(expr2)`
`#define W_cltl(expr1, expr2)`
`  c_imp(expr1) W c_and(expr2)`
`#define V_cltl(expr1, expr2)`
`  c_and(expr1) V c_imp(expr2)`
`#define next_cltl(expr) (cycle_u ->`
`  (cycle_u U (!cycle_u W c_and(expr))))`
	`bool cycle_u;`
	`active proctype BOC () {`
	`  do :: current ? p_BOC ->`
	`  cycle_u = true;`
	`  atomic {`
	`  cycle_u = false;`
	`  current ! p_`$name_{11}$`; }`
	`  od;`
	`}`

`expr)`, что и даёт нужное выражение циклического темпорального оператора через обычный темпоральный LTL-оператор. Для остальных операторов рассуждения аналогичны.

Для взаимодействия процессов с окружением в poST-программах объявляются переменные INPUT, OUTPUT и IN_OUT. Процессы в poST могут включать переменные INPUT, не связанные с переменными OUTPUT и IN_OUT других процессов. Такие переменные реализуют входы от внешней среды, поведение которой считается неопределенным. Мы моделируем неопределенное поведение среды с помощью процесса `Gremlin`, представленного в левом блоке таблицы 5 сверху. При этом в силу ресурсоемкости верификации моделей в SPIN, рассматриваются только переменные «небольших» типов BOOL, USINT и SINT. Более подробно идея спецификации неопределенного поведения среды изложена в разделе 4.3.

Программы poST взаимодействуют через переменные с одинаковыми именами. Политика именования при трансляции такова, что эти переменные имеют разные имена в результирующей Promela-модели. Поэтому необходимо явно обновлять входы одной программы выходами другой программы между итерациями цикла управления с помощью Promela-процесса `OutInput`, который приведён в левом блоке Таблицы 5 снизу.

3.5. Общая модель Promela для программ poST

В общем случае, наш алгоритм трансляции принимает на вход несколько poST-программ, описывающих систему управления в одном файле. Эта система управления может включать в себя алгоритм управления и его окружение: управляемые и неуправляемые объекты. В правом блоке таблицы 5 представлена результирующая Promela-модель, включающая три сервисных процесса и процессы, соответствующие исходным poST-процессам. Деятельность этих процессов образует цикл управления посредством передачи между ними активирующих сообщений, начиная с недетерминированного процесса `Gremlin`, моделирующего неконтролируемый объект с неопределенным поведением, и заканчивая результатом трансляции последнего poST-процесса, который снова передает активирующее сообщение процессу `Gremlin`. Внутри цикла активность процессов упорядочена, как описано в таблицах 2 и 5.

Table 5. Service processes and the Promela model for a poST program

Таблица 5. Сервисные процессы и модель Promela для программы poST

poST	Promela	poST	Promela

PROCESS n
VAR_INPUT
v_b : BOOL;
v_u : USINT;
v_s : SINT;
...
END_VAR
...
END_PROCESS
	active proctype Gremlin(){
	do :: cur ? p_Gremlin ->
	atomic {
	if
	:: v'_b = true;
	:: v'_b = false;
	fi;
	select (v'_u: 0..255);
	select (v'_s: -128..127);
	cur ! p_OutInput;}
	od;
	}
		PROGRAM $prog_1$
		Var_Decl_1
		PROCESS n_{11}
		...
		PROCESS n_{n1}
		END_PROGRAM
		...
		PROGRAM $prog_m$
		Var_Decl_m
		PROCESS n_{1m}
		...
		PROCESS n_{nm}
		END_PROGRAM
			$Var\_Decl'_1$
			...
			$Var\_Decl'_m$
			$Service\_Decl$
			init{
			cur ! p_Gremlin;}
			active proctype
			Gremlin(){...}
			active proctype
			OutInput(){...}
			active proctype
			BOC(){...}
			active proctype
			$name'_{11}$(){...}
			...
			active proctype
			$name'_{nm}$(){
			do :: cur ? p_n'_{nm} ->
			atomic {
			...
			cur ! p_Gremlin;
			}
			od;
			}

PROGRAM n_1
VAR_OUTPUT
var : $type$;
END_VAR
...
END_PROGRAM
...
PROGRAM n_2
VAR_INPUT
var : $type$;
END_VAR
...
END_PROGRAM
	active proctype OutInput(){
	do :: cur ? p_OutInput ->
	atomic {
	$var'(n_2)$ = $var'(n_1)$;
	...
	cur ! p_BOC;}
	od;
	}		

4. Реализация транслятора poST в Promela

4.1. Общий подход к реализации

Для реализации нашего транслятора мы следуем ранее заданным архитектурным подходам к проектированию трансляторов языка poST. В частности, мы используем фреймворк Xtext[3] [28] в качестве базового инструмента для разработки парсеров. При этом код описанной трансформационной семантики реализован на языке Xtend[4], который в дальнейшем преобразуется в Java с помощью Eclipse IDE для DSL разработчиков[5]. Мы используем этот язык прежде всего потому, что на нем удобно описывать шаблоны генерации языковых конструкций, параметризованных с использованием подготовленного контекста в объектах. Что касается модели ввода, мы используем представление программы poST в виде EMF объектов [29], которые естественным образом обрабатываются в Xtend. Парсинг внутри Xtend проектов осуществляется с использованием сред-

[3] http://eclipse.org/Xtext
[4] https://www.eclipse.org/xtend
[5] https://www.eclipse.org/downloads/packages/release/2022-12/r/eclipse-ide-java-and-dsl-developers

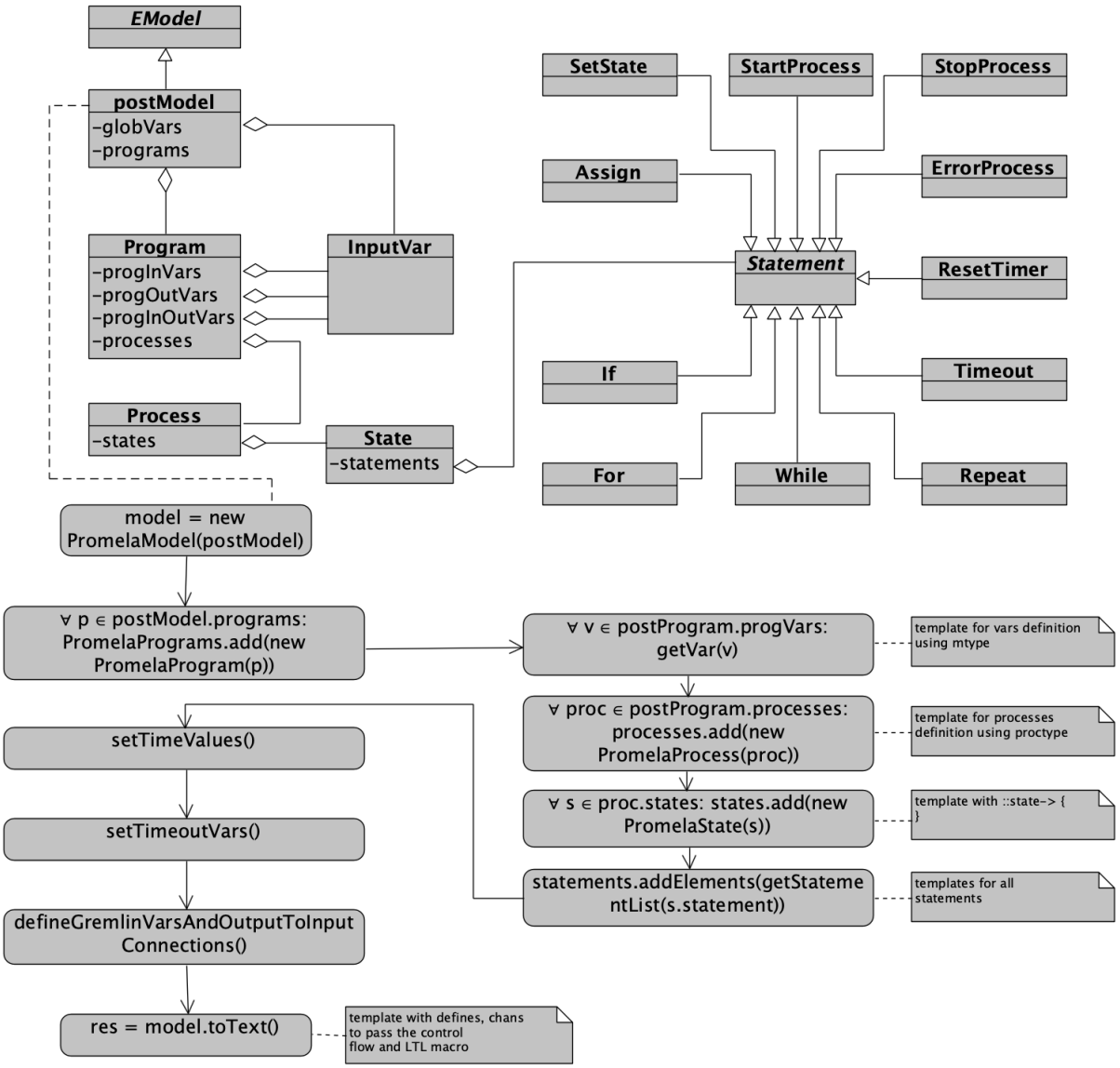

Fig. 1. Implementation of the translation of poST statements to Promela

Рис. 1. Реализация трансляции выражений poST в язык Promela

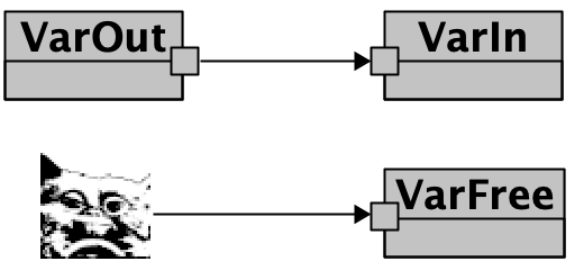

Fig. 2. A normal and a gremlin-based variable binding

Рис. 2. Связывание переменных стандартное и с гремлином

ства ANTLR [30], но мы работаем с объектами языка и их связями на высоком уровне. Обсуждаемый транслятор представляет собой JVM-приложение, которое запускается из командной строки.

4.2. Детали реализации

Рассмотрим, каким именно образом мы получаем код на языке Promela из poST-программы. В верхней части рис. 1 представлены EMF объекты программы на poST. В соответствии с процесс-ориентированной структурой модель, полученная из исходного poST-кода, определяет несколько программ с процессами (например, Plant и Controller). Процессы этих программ взаимодействуют через значения входных и выходных переменных, которыми они обмениваются по завершении цикла управления. Сам процесс является конечной системой переходов, при этом каждое состояние представлено атомарной последовательностью действий, которая определяется операторами языка poST (на рисунке показаны основные из них). Задача транслятора — корректно перевести операторы, описания процессов, переменных и т. п. языка poST в код на языке Promela, согласно семантике, определенной таблицах раздела 3.

Транслятор (в нижней части рис. 1) итеративно перебирает структуры данных процессов, переменных, состояний и операторов исходного кода poST для создания соответствующих языковых конструкций в Promela. Данные о текущем процессе, состоянии и т. п. сохраняются в текущем контексте. Он используется в шаблонах, которые описываются в виде конструкций языка Xtend, где удобно работать со строками. Приведём пример такого шаблона для процесса на poST:

```
>>’
<<IF !vars.isEmpty()>>
<<vars.toText()>>
<<ENDIF>>

active proctype <<Context.getName(name)>>()
{
  do ::
  <<Context.getName(__currentProc)>> ?
  <<Context.getName(nameMType)>> ->
    atomic {
    if
      <<states.toText()>>
      :: else -> skip;
    fi;
     <<Context.getName(__currentProc)>> !
     <<Context.getName(nextMType)>>;
    }
  od;
}
>>’
```

Во французских угловых кавычках здесь находится код в Xtext, который выполняется и подставляет результат в шаблон. Сначала генерируются переменные процесса. Затем объявляется процесс с именем, порожденным согласно вышеопределенной политике именования. Для порождённого таким образом процесса генерируется (1) получение активирующего сообщения по ранее сгенерированному каналу, (2) тело процесса и (3) передача хода следующему процессу, информация о котором предварительно была получена из poST-программы.

Отметим на рис. 1 следующие функции трансляции реализующие работу с таймаутами, которые позволяют уменьшить размер результирующей модели (см. конец раздела 3.3). Первый способ редуцирования размера модели состоит в сокращении количества итераций цикла при работе с таймаутами: функция *setTimeValues* корректирует значения таймаутов процессов, разделив их на НОД (аналогичная техника применяется в [31]). Второй способ редуцирования уменьшает размер необходимой памяти для хранения пространства состояний во время проверки модели: функция *setTimeoutVars* находит в выражениях все заданные значения таймаута в стиле ST[6] и вычисляет количество битов для минимизации их представления, используя беззнаковый тип данных Promela[7].

Функция трансляции *defineGremlinVarsAndOutputToInputConnections()* вставляет служебные процессы обработки переменных из таб. 5 в список процессов. Процесс работы с переменными активируется после очередного цикла управления. Этот процесс присваивает значения выходных переменных значениям тех входных переменных, для которых определена такая привязка. Переменные, которые не имеют программных связей с выходными переменными, считаются переменными, значения которых задаёт окружение программируемой системы управления, поведение которого недетерминировано. Это окружение моделируется с помощью подхода *«гремлин»*.

4.3. Недетерминизм на основе подхода «гремлин»

Алгоритм управления обычно взаимодействует с внешним устройством, которое может иметь сложную логику изменения переменных в результате действия физических законов. Однако не всегда необходимо знать и моделировать эту логику, чтобы обеспечить верификацию заданных требований при активации определенных структур в управляющем коде. Для моделирования взаимодействия алгоритма управления и такой внешней среды, мы используем *гремлина*, который представляет собой нерассуждающую силу, способную непредсказуемым образом изменять значения входных переменных перед выполнением следующего цикла управления (подобно гремлинам из знаменитого фильма Джо Данте и Стивена Спилберга, которые буянили в магазине, хватая и бросая всё, что попадалось на глаза). Насколько нам известно, такой гремлин-подход впервые был использован при тестировании пользовательского интерфейса программ Palm OS [32]: здесь гремлины активируются в произвольных местах экрана, обеспечивая рандомизированное тестирование и огромное количество вариантов переходов между экранными формами приложения.

Если входная переменная в программе poST не связана с какой-либо другой переменной (см. рис. 2), считается, что она может принимать произвольное значение из диапазона типа переменной. При моделировании это означает, что переменной будет присвоено случайное значение, а при верификации — что переменной будут присвоены все возможные значения. Такой подход позволяет получить представление обо всех возможных случаях входных данных для работы системы управления. Развитием этого подхода является создание ограниченных гремлинов посредством установки условий на свободные переменные.

5. Пример трансляции

Рассмотрим задачу моделирования потребления и производства электроэнергии, в том числе из возобновляемых источников. Компонентами системы являются:

- **солнце (sun)**: днем может светить, а может и не светить из-за появления облаков; ночью не светит;
- **солнечная панель (solar Panel)**: производит одну единицу энергии в цикл, если светит солнце;

[6] См., например, https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528280971.html.
[7] http://spinroot.com/spin/Man/datatypes.html

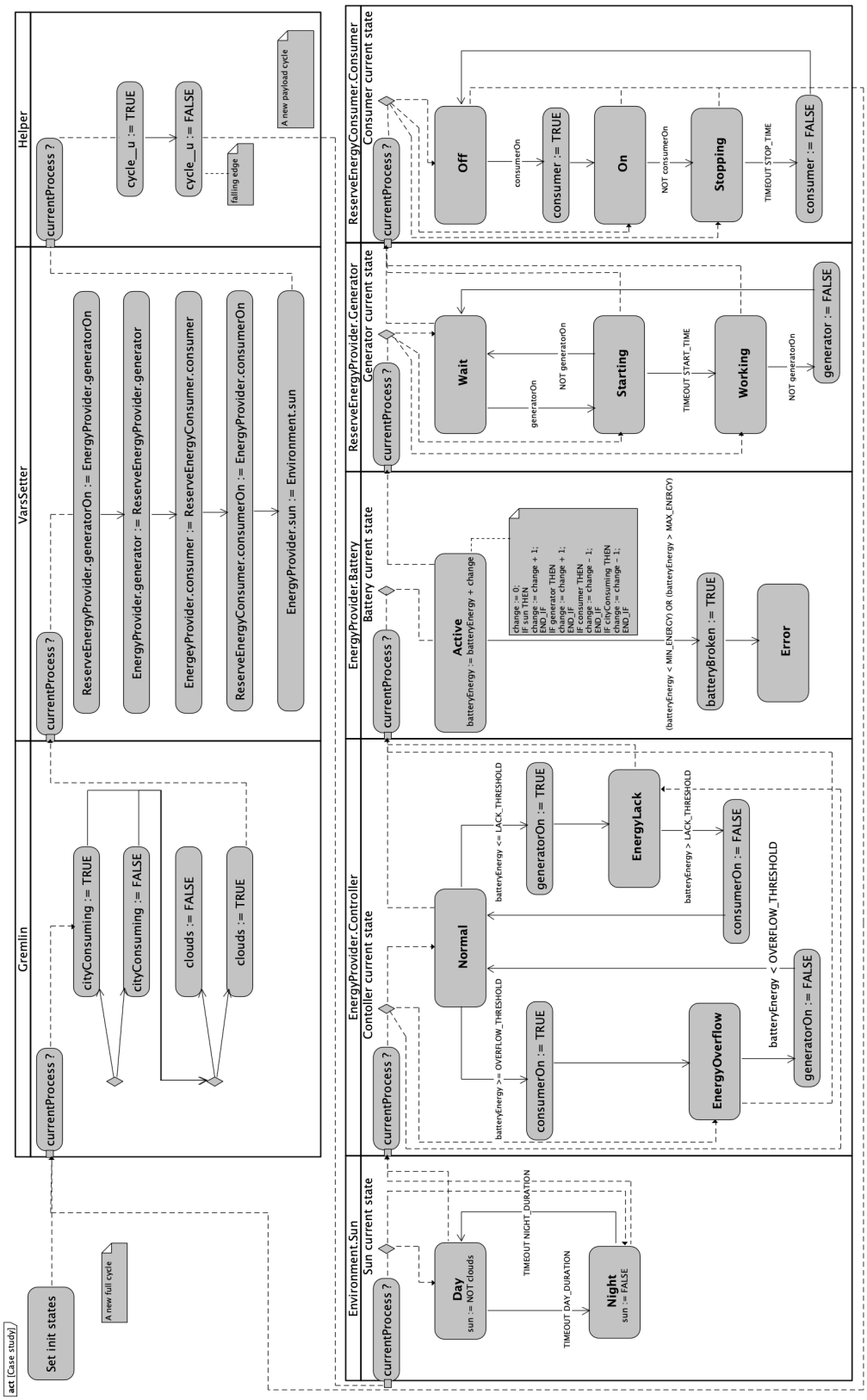

Fig. 3. UML modeling of the case study

Рис. 3. UML-модель иллюстративного примера

- **батарея (battery)**: накапливает определенное количество единиц энергии; если единиц энергии мало или много, то батарея выходит из строя (случай ошибки);
- **потребитель (consumer)**: может потреблять или не потреблять одну единицу энергии в цикл;
- **нагрузка (load)**: можно включить для рассеивания лишней энергии, если светит солнце и город не потребляет энергию;
- **генератор (generator)**: может включаться при недостатке электроэнергии, когда не светит солнце и потребитель потребляет энергию;
- **контроллер (controller)**: включает и выключает генератор и нагрузку, чтобы поддерживать заряд батареи в допустимых пределах.

UML-моделирование работы системы в процесс-ориентированной парадигме показано на рис. 3. В данном случае используется диаграмма деятельности, предназначенная для отображения нескольких процессов с их системами переходов [33]. Однако здесь нет процессов, исполняющихся параллельно: видно, что в самом начале каждый процесс ждёт активирующего сообщения в свой канал («ожидание очереди»), чтобы продолжить работу в своём текущем состоянии. Такое исполнение в системе процессов аналогично циклическому планированию с состояниями и заданной последовательностью процессов. Пунктирными линиями мы отметили необходимый поток управления средой исполнения (в соответствии с заданной семантикой), а простыми линиями отмечаем переходы по логике системы, описанной в poST-программе.

Работа системы является циклической. Получив ход, процесс предпринимает свои действия исходя из текущего состояния и передает ход следующему. При этом вначале действуют три сгенерированных служебных процесса: (1) для установки значений свободных переменных (здесь: гремлин для переменных `cityConsuming` и `clouds`); (2) для установки значения входных переменных в соответствии с заданными выходными переменными (здесь: согласование значений пяти переменных, которые задаются одними процессами и должны быть доступны другим после завершения итерации цикла управления); и (3) установка импульса начала цикла управления, который используется в спецификациях требований.

Рассмотрим наш метод трансляции на одном из процессов примера, а именно на процессе `Consumer`. Пусть его исходный код в poST выглядит следующим образом:

```
PROCESS Consumer
  VAR CONSTANT
    STOP_TIME : TIME := T#1h;
  END_VAR
  STATE Off
    IF consumerOn THEN
      consumer := TRUE;
      SET STATE On;
    END_IF
  END_STATE
  STATE On
    IF NOT consumerOn THEN
      SET STATE Stopping;
    END_IF
  END_STATE
  STATE Stopping
    TIMEOUT STOP_TIME THEN
      consumer := FALSE;
```

```
      SET STATE Off;
    END_TIMEOUT
  END_STATE
END_PROCESS
```

Результат трансляции этого процесса в код на языке Promela показан ниже:

```
#define STOP_TIME 3600000 //1h
active proctype Consumer() {
do :: __currentProcess ? Consumer__p ->
 atomic {
 if :: Off__s == Consumer__cs ->
        if :: consumerOn__0 ->
                consumer__1 = true;
                Consumer__cs = On__s;
           :: else -> skip;
        fi;
    :: On__s == Consumer__cs ->
        if :: !consumerOn__0 ->
                Consumer__t = 1;
                Consumer__cs = Stopping__s;
           :: else -> skip;
        fi;
    :: Stopping__s == Consumer__cs ->
        if :: Consumer__t > STOP_TIME ->
                consumer__1 = false;
                Consumer__cs = Off__s;
           :: else ->
                Consumer__t = Consumer__t + 1;
        fi;
    :: else -> skip;
  fi;
  __currentProcess ! Gremlin__sp;
 }
od;}
```

В результирующем процессе объявлен бесконечный цикл, в начале которого процесс ждёт активирующее сообщение в канале передачи активирующих сообщений. Сгенерированные имена состояний процесса `Off__s`, `On__s` и `Stopping__s` являются ключами при выборе текущего состояния процесса `Consumer__cs`. `Consumer__1` — это сгенерированное Promela-имя переменной `Consumer` в poST с добавленным номером, поскольку в других программах в модели тоже есть такая переменная. Эта переменная будет также участвовать в обмене значениями в начале цикла управления в сгенерированном процессе `OutInput`, как описано в разделе 3.4. Наконец, генерируется переменная таймера `Consumer__t` для подсчета времени, в течение которого процесс находится в этом состоянии, чтобы реализовать переход в результате таймаута. В конце тела процесса ход передается процессу следующему по схеме.

В терминах логики линейного времени LTL с использованием циклического темпорального оператора `G_cltl` (раздел 3.4) мы сформулировали следующие требования:

1. `G_cltl (!batteryBroken)` — означает, что батарея никогда не сломается.
2. `G_cltl (!(generatorOn && ConsumerOn))` — означает, что потребитель и генератор никогда не работают одновременно.

Исходная poST-программа и ее представление в Promela доступны в нашем репозитории[8].

Заключение

В статье мы рассмотрели правила трансляции процесс-ориентированного языка poST в язык формальной верификации Promela. Таким образом, наша трансляция является транспиляцией, которая оправдана для языков программирования ПЛК с небольшим числом операторов, поскольку все конструкции входного языка могут быть преобразованы в конструкции языка, для которых уже хорошо разработаны методы формальной верификации (в нашем случае мы используем метод проверки моделей и инструмент верификации SPIN). Автоматическая транспиляция из процесс-ориентированного языка poST позволяет писать верифицируемые программы для ПЛК, не кодируя на Promela повторяющиеся языковые конструкции, связанные с семантикой переключения процессов, переходом по состояниям, обменом значениями переменных и генерацией импульса для верификации свойств для циклов управления. Проект находится в публичном доступе в репозитории[9]. Среди недостатков текущего подхода к трансляции можно отметить неполную поддержку типов данных. В частности, не поддерживаются переменные, принимающие действительные значения. Наш подход можно развить, реализуя библиотеки арифметики как с фиксированной, так и с плавающей запятой, однако это повлечет за собой очень большое количество состояний модели и верификацию результирующих программ затруднительно будет осуществить без подходящих методов абстрагирования. Также мы не реализовывали библиотечные функции poST. Однако разработка транспилятора poST2Promela является значительным шагом в построении инструментальной цепочки создания верифицируемых процесс-ориентированных программ.

References

[1] IEC, *IEC 61131-3: 2013 Programmable controllers-Part 3: Programming languages*, https://webstore.iec.ch/publication/4552, International Standard, 2013.

[2] T. M. Antonsen, *PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming*. Books on Demand, 2020, 208 pp.

[3] V. E. Zyubin, “Hyper-automaton: A model of control algorithms”, in *2007 Siberian Conference on Control and Communications*, 2007, pp. 51–57. DOI: 10.1109/SIBCON.2007.371297.

[4] V. E. Zyubin, A. S. Rozov, I. S. Anureev, N. O. Garanina, and V. Vyatkin, “poST: A process-oriented extension of the IEC 61131-3 structured text language”, *IEEE Access*, vol. 10, pp. 35 238–35 250, 2022. DOI: 10.1109/ACCESS.2022.3157601.

[5] G. J. Holzmann, *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003, 608 pp.

[6] L. Schneider and D. Schultes, “Evaluating Swift-to-Kotlin and Kotlin-to-Swift transpilers”, in *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2022, pp. 102–106.

[7] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of model checking*. Springer, 2018, 1212 pp.

[8] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/tree/main/samples
[9] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev

[8] T. Ovatman, "An overview of model checking practices on verification of PLC software", *Software & Systems Modeling*, vol. 4, no. 15, pp. 937–960, 2016.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[10] C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[11] N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, "Model checking programs in process-oriented IEC 61131-3 structured text", *System Informatics*, vol. 22, pp. 21–30, 2023. DOI: 10.31144/si.2307-6410.2023.n22.p21-30.

[12] M. Weißmann, S. Bedenk, C. Buckl, and A. Knoll, "Model checking industrial robot systems", in *Model Checking Software. SPIN 2011*, Springer, 2011, pp. 161–176.

[13] P. Cousot, "Abstract interpretation", *ACM Computing Surveys*, vol. 28, no. 2, pp. 324–328, 1996. DOI: 10.1145/234528.234740.

[14] S. Leue and G. Holzmann, "v-Promela: A visual, object-oriented language for SPIN", in *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE, 1999, pp. 14–23.

[15] M. Benerecetti *et al.*, "From dynamic state machines to Promela", in *International Symposium on Model Checking Software*, Springer, 2019, pp. 56–73.

[16] B. Lion, S. Chouali, and F. Arbab, "Compiling protocols to Promela and verifying their LTL properties", in *Proceedings of 21st MODELS Workshops*, 2018, pp. 31–39.

[17] A. Klarl, "From Helena ensemble specifications to Promela verification models", in *Model Checking Software. SPIN 2015*, Springer, 2015, pp. 39–45.

[18] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in Go using Promela and SPIN", in *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software*, EPTCS, 2010, pp. 34–45. DOI: 10.4204/EPTCS.314.4.

[19] N. Dilley and J. Lange, "Automated verification of Go programs via bounded model checking", in *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1016–1027.

[20] E. Brinksma, A. Mader, and A. Fehnker, "Verification and optimization of a PLC control schedule", *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 1, pp. 21–33, 2002. DOI: 10.1007/s10009-002-0079-0.

[21] M. Xia, M. Sun, G. Luo, and X. Zhao, "Design and implementation of automatic verification for PLC systems", in *IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing*, IEEE, 2013, pp. 374–379.

[22] P. Liu, G. Luo, M. Xia, and M. He, "Automatic verification of event-driven control programs: A case study", in *The Fourth International Workshop on Advanced Computational Intelligence*, IEEE, 2011, pp. 249–256.

[23] T. Liakh, R. Sorokin, D. Akifev, S. Patil, and V. Vyatkin, "Formal model of IEC 61499 execution trace in FBME IDE", in *IEEE 20th International Conference on Industrial Informatics (INDIN)*, IEEE, 2022, pp. 588–593.

[24] V. Shatrov and V. Vyatkin, "Promela formal modelling and verification of IEC 61499 systems with comparison to SMV", in *IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, 2021, pp. 1–6.

[25] A. Ebnenasir, "Formalizing ladder logic programs and timing charts for fault impact analysis and verification of fault tolerance", Michigan Technological University, Tech. Rep. CS-TR-23-01, Jan. 2023.

[26] A. Mader, "A classification of PLC models and applications", *Discrete Event Systems: Analysis and Control*, vol. 569, pp. 239–246, 2000.

[27] N. O. Garanina *et al.*, "A temporal logic for programmable logic controllers", *Automatic Control and Computer Sciences*, vol. 55, no. 7, pp. 763–775, 2021. DOI: 10.3103/S0146411621070038.

[28] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way", in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 307–309.

[29] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse modeling framework*. Pearson Education, 2008, 744 pp.

[30] "ANTLR: A predicated-LL(k) parser generator", *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[31] D. Lepri, E. Ábrahám, and P. C. Ölveczky, "Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories", *Science of Computer Programming*, vol. 99, pp. 128–192, 2015.

[32] N. Rhodes and J. McKeehan, *Palm OS programming: the developer's guide*. O'Reilly Media, Inc., 2002, 681 pp.

[33] R. Miles and K. Hamilton, *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc., 2006, 290 pp.