Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б07-мм

Мутация кода для фаззинга симулятора процессора на gem5

Лодыгин Леонид Александрович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
старший преподаватель кафедры ИАС, К. К. Смирнов

Консультант:
инженер-исследователь «Лаборатория технологий программирования инфраструктурных решений СПбГУ», В. А. Кутуев

Санкт-Петербург
2024

Оглавление

Введение

Процессоры являются ключевым компонентом современных вычислительных систем от персональных компьютеров и смартфонов до серверов. Он выполняет центральную роль в обработке данных. Ввиду своей важности процессоры должны быть максимально надёжными и безопасными. Малейшие ошибки в их работе могут привести к серьёзным последствиям, таким как сбои в работе систем, утечки данных или уязвимости [6] [1], которые могут быть использованы злоумышленниками.

Фаззинг представляет собой метод тестирования программного обеспечения, направленный на обнаружение ошибок, уязвимостей и дефектов. Он заключается в автоматической генерации случайных данных, которые подаются на вход тестируемой системе. Является инструментом для выявления критических ошибок, особенно в низкоуровневом программном обеспечении [8].

Одним из ключевых аспектов фаззинга является *мутирование кода*, которое позволяет эффективно генерировать разнообразные тестовые случаи. Мутирование кода подразумевает внесение изменений в исходный код для создания множества его вариаций. Это позволяет исследовать поведение системы в различных условиях, включая самые неожиданные.

Генетическое программирование представляет собой методику, вдохновлённую принципами естественного отбора и генетики. В контексте фаззинга и мутирования кода генетическое программирование позволяет эволюционно развивать тестовые случаи. Это достигается путём применения операторов мутации и скрещивания к наборам данных, создавая новые поколения тестов, которые постепенно улучшают свою эффективность в обнаружении ошибок. Генетическое программирование особенно полезно для сложных систем, таких как процессоры, где ручное создание тестов является крайне трудоёмким процессом.

Фаззинг процессоров — это метод тестирования, при котором процессор подвергается воздействию случайных данных, чтобы выявить скры-

тые ошибки и уязвимости в его работе. Этот подход особенно важен на фоне растущих требований к безопасности и надёжности современных вычислительных систем. Благодаря автоматизации и применению машинного обучения, фаззинг стал неотъемлемой частью верификации процессоров, позволяя разработчикам повысить устойчивость процессоров к потенциальным сбоям и атакам [12].

1. Постановка задачи

Целью данной работы является реализация различных вариантов мутирования кода в зависимости от его генерации для применения в фаззере `gem5` симулятора. Для её выполнения были поставлены следующие задачи:

1. Провести обзор существующих решений.
2. Разработать различные варианты мутирования кода.
3. Реализовать полученные варианты.

Генерацией кода занимался Яворовский А.А.

2. Обзор

2.1. Генетический алгоритм

Генетический алгоритм [10] является одним из методов эволюционных вычислений, предназначенных для решения задач оптимизации с помощью моделирования механизмов биологической эволюции. Одним из основных понятий в данном алгоритме является *хромосома* или *генотип* — возможное решение задачи. Совокупность таких хромосом составляет *популяцию*. Первоначальный шаг алгоритма — инициализация популяции, как правило случайным образом. Каждая хромосома в полученной популяции оценивается на качество решения изначальной задачи с помощью *функции приспособленности* [9]. По результатам проверки происходит отбор более подходящих решений, после чего над оставшимися хромосомами производятся операции *скрещивания* и *мутации*. Данный цикл повторяется, пока не будет достигнут *критерий остановки*. Критерием может являться приемлемое качество решения или же количество поколений. Схематично алгоритм может быть представлен следующим образом (рисунок 1).

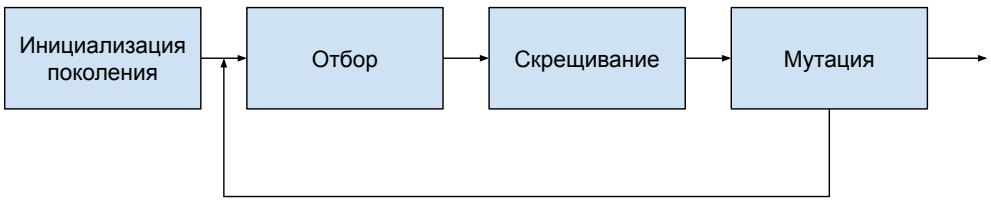

Рис. 1: Схема генетического алгоритма

Данный алгоритм применяется для оптимизации сложных систем, где традиционные методы могут работать неэффективно. Например, для настройки и обучения нейронных сетей, для решения задач на графах, для оптимизации запросов в базах данных.

2.2. Генетическое программирование

Генетическое программирование [11] — метод эволюционного моделирования, который используется для автоматического создания программ с помощью генетического алгоритма. Одной из особенностей данного подхода является выбор способа кодирования программы. Самым распространенным способом является представление программы в виде дерева разбора [5], так как на таком уровне абстракции легко вносить изменения в программу. В связи с такой особенностью, наиболее часто для генетического программирования используются функциональные яызки программирования, где программы естественным образом поддерживают такое представление.

При таком подходе операцией скрещивания является замена случайного поддерева одной хромосомы на случайное поддерево второй хромосомы (рисунок 2), а операцией мутации будет являться замена одного из поддеревьев на случайно сгенерированное (рисунок 3).

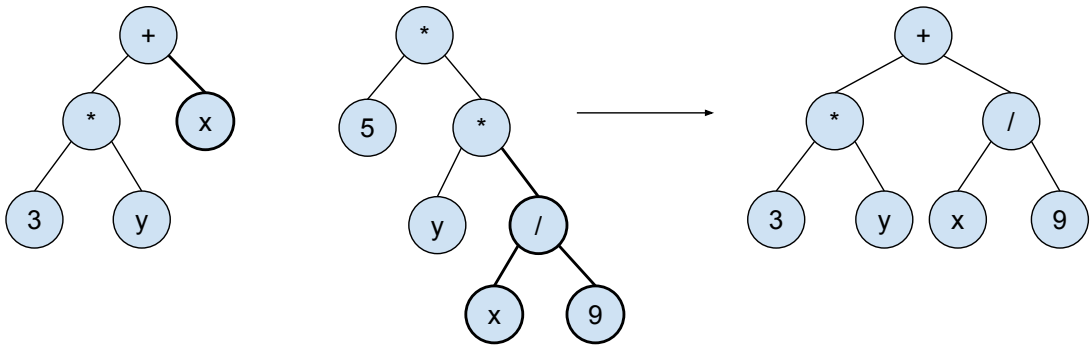

Рис. 2: Операция скрещивания

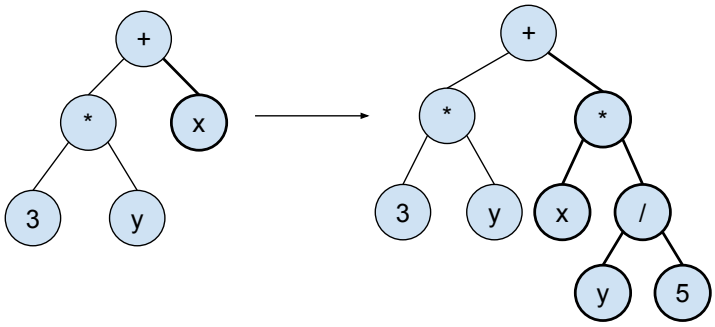

Рис. 3: Операция мутации

2.3. Существующие технологии

Для реализации генетических алгоритмов существует большое количество библиотек на различных языках программирования.

2.3.1. PyGAD

Библиотека на языке python, предназначенная для реализации генетических алгоритмов. Она ориентирована на простоту использования и подходит для решения различных задач оптимизации, подбора параметров и создания моделей, использующих принципы эволюции. `PyGAD` [7] поддерживает как генетические алгоритмы, так и простейшие версии генетического программирования через эволюцию строк или массивов. Основными особенностями библиотеки являются:

1. Поддержка различных типов данных: `PyGAD` может работать с массивами различных типов данных, таких как числа (целые, вещественные) или строки. Это делает библиотеку универсальной для применения к широкому кругу задач, от оптимизации математических функций до генетического программирования символьных выражений.

2. Поддержка различных стратегий отбора и эволюции.

2.3.2. DEAP

`DEAP` [2] предоставляет инструменты для создания, настройки и запуска эволюционных вычислений, позволяя пользователям интегрировать эти алгоритмы в свои проекты. Основные особенности `DEAP`:

1. `DEAP` предоставляет модульную архитектуру, которая позволяет настраивать каждый аспект эволюционных алгоритмов. Можно изменять и настраивать операторы отбора, мутации, кроссовера, а также определять собственные функции приспособленности и генетические представления.

2. `DEAP` поддерживает распределённые вычисления, что позволяет запускать задачи на нескольких процессорах для ускорения выполнения, особенно на больших популяциях или при сложных функциях отбора.

2.3.3. Jenetics

Библиотека [3] на языке Java, предназначенная для решения задач оптимизации и автоматизации поиска решений, где целевая функция трудно поддается анализу или имеет большой объем параметров. Данная библиотека содержит модуль `jenetics.prog` [4], предлагающий набор классов для построения, мутации и кроссинговера деревьев выражений. В `jenetics.prog` можно создавать выражения, состоящие из множества узлов, что позволяет включать комбинации математических операций, логические операторы, функции и условия. Используя большое число операций и терминалов, можно приблизить дерево выражений к структуре более сложной программы.

2.4. Использованные технологии

При использовании готовых библиотек для реализации генетического алгоритма серьезной задачей является описание представления хромосомы и возможностей для ее изменения. Так как для фаззинга процессоров предполагалось использовать множество различных подходов кодогенерации первоначальной популяции с использованием различных языков программирования, было принято решение не использовать готовые библиотеки для мутирования кода и написать для каждого случая свою реализацию, чтобы оценить, насколько конкретный подход "имеет право на жизнь". Помимо привязки к языку, многие готовые библиотеки для мутации кода ориентированы на задачи оптимизации функций и символьную регрессию, что делает их менее гибкими и подходящими для общего применения к произвольному коду.

3. Детали реализации

В данном разделе предлагается рассмотреть основные детали реализации мутации кода, предназначенного для фаззера симулятора `gem5`. Для кодогенерации были выбраны языки C и Assembly для избегания излишних трансформаций кода во время компиляции. В качестве основного языка разработки был выбран Python.

3.1. Мутация Csmith кода

Csmith генерирует код на C на основе задаваемого набора конфигураций. В связи с этим, вместо решения проблемы представления сгенерированного кода в виде дерева разбора для последующего мутирования, было принято решение применения мутации не к самому коду, а к набору конфигураций. Сам набор имеет большое количество параметров с различными диапазонами значений, например:

1. Ограничение на используемые типы данных.
2. Количество функций в коде.
3. Ограничение сложности выражений.
4. Выбор размерности массивов, используемых в коде и ограничение их длины.

Таким образом, под хромосомой подразумевается набор параметров для генерации кода (рисунок 4), операцией скрещивания является замена одного из параметров набора на параметр из другого набора (рисунок 5), а операцией мутации — замена значения одного из параметров на сгенерированный заново (рисунок 6).

```
Chromosome = {
    '--seed': range(1, 100),
    '--max-funcs': range(1, 100),
    '--max-expr-complexity': range(1, 100),
    '--max-array-len-per-dim': range(1, 100),
    '--max-funcs': range(1, 100),
    ...
}
```

Рис. 4: Шаблон для хромосомы

```
Chromosome1 = {
    '--seed': 5,
    '--max-funcs': 12,
    '--max-expr-complexity': 20,
    '--max-array-len-per-dim': 51,
    '--max-funcs': 3,
    ...
}

Chromosome2 = {
    '--seed': 9,
    '--max-funcs': 33,
    '--max-expr-complexity': 1,
    '--max-array-len-per-dim': 68,
    '--max-funcs': 30,
    ...
}
```

```
Child1 = {
    '--seed': 5,
    '--max-funcs': 12,
    '--max-expr-complexity': 1,
    '--max-array-len-per-dim': 51,
    '--max-funcs': 3,
    ...
}

Child2 = {
    '--seed': 9,
    '--max-funcs': 33,
    '--max-expr-complexity': 20,
    '--max-array-len-per-dim': 68,
    '--max-funcs': 30,
    ...
}
```

Рис. 5: Скрещивание конфигов

```
Chromosome = {
    '--seed': 5,
    '--max-funcs': 12,
    '--max-expr-complexity': 20,
    '--max-array-len-per-dim': 51,
    '--max-funcs': 3,
    ...
}

Child = {
    '--seed': 5,
    '--max-funcs': 12,
    '--max-expr-complexity': 100,
    '--max-array-len-per-dim': 51,
    '--max-funcs': 3,
    ...
}
```

Рис. 6: Мутация конфига

При генерации начальной популяции используется шаблон из набора доступных параметров. Каждый параметр из шаблона, кроме параметра seed, может как попасть в хромосому со случайным выбранным значением, так и вовсе отсутствовать. После генерации хромосом, по каждой из них `Csmith` генерирует код на C, который затем компилируется и отправляется в `gem5`. На основе полученной статистики из gem5, функция отбора, вычисляемая обратное отклонение полученного результата от желаемого, определяет приспособленность каждой хромосомы. Самые слабые хромосомы отсеиваются, над оставшимися хромосомами производятся операции скрещивания и мутации, для хромосом нового поколения `Csmith` генерирует код и все происходит по кругу, пока не будет достигнут необходимый результат или не будет совершено определенное количество итераций в алгоритме.

3.2. Мутация Assembly кода

Мутация ассемблерного кода происходит с помощью манипуляций над так называемыми базовыми блоками — последовательностями инструкций, имеющими одну точку входа и одну точку выхода. Полученный ассемблерный код разбивается на базовые блоки путем определения инструкций-лидеров по следующим критериям (рисунок 7):

1. Первая инструкция в коде — лидер.
2. Инструкция, являющаяся целью условного или безусловного перехода — лидер.
3. Инструкция, следующая за инструкцией условного или безусловного перехода тоже лидер.

Таким образом, инструкция–лидер и все последующие инструкции до следующей инструкции–лидера образуют базовый блок в ассемблерном коде. Тогда операцией скрещивания будет перестановка двух базовых блоков между хромосомами.

```
mov eax, 1      <- Лидер(первая инструкция)
add eax, 2
jmp label1
sub eax, 3      <- Лидер(инструкция после безусловного перехода)
label1:         <- Лидер(цель безусловного перехода)
cmp eax, 0
je label2
mul eax         <- Лидер(инструкция после условного перехода)
label2:         <- Лидер(цель условного перехода)
div eax
```

Рис. 7: Пример разбиения ассемблерного кода на базовые блоки

Во время операции мутации часть случайно выбранных команд в коде заменяются на другие из заранее подготовленного списка.

4. Тестирование

Для тестирования реализованных функций был написан набор тестов с использованием встроенной библиотеки `unittest`. Все тесты проходятся без ошибок.

Проект по фаззингу процессора разрабатывался в команде из нескольких человек, часть из которых в последствии его покинули. На момент присутствия полного состава было произведено исследование подхода с генерацией кода на C с помощью `Csmith`. Метрикой для данного исследования было выбрано `CPI` — среднее количество тактов процессора, затраченных на выполнение одной инструкции.

Исследование проводилось на рабочей станции с процессором Intel Core i5-10300H с тактовой частотой 2.50GHz, RAM DDR4 объемом 8Гб под управлением ОС Windows 10, на которой была запущена симуляция процессора на архитектуре `RISC-V` с помощью `gem5` версии 23.1.0.0. В качестве контроллера памяти использовался `DDR3`. Полная конфигурация приведена в приложении 1.

Проводимое исследование показало, что метод генетического программирования для мутации сгенерированных конфигов позволяет усиливать нагрузку на интересующий в тестах показатель `gem5`, при 100 итерациях был достигнут CPI в размере 88.9, однако подход с мутированием конфигураций для `Csmith` обладает явным недостатком — на каждой итерации затрачивается внушительное количество времени на генерацию кода для новых хромосом, в разы превышающее время, требующееся на прогон кода через `gem5` (рисунок 8). Причем время, требующееся на генерацию, сильно возрастает с увеличением количества параметров и увеличением их значений (рисунок 9, 10). Шаблоны хромосом, по которым генерировался код приведены в приложении 2

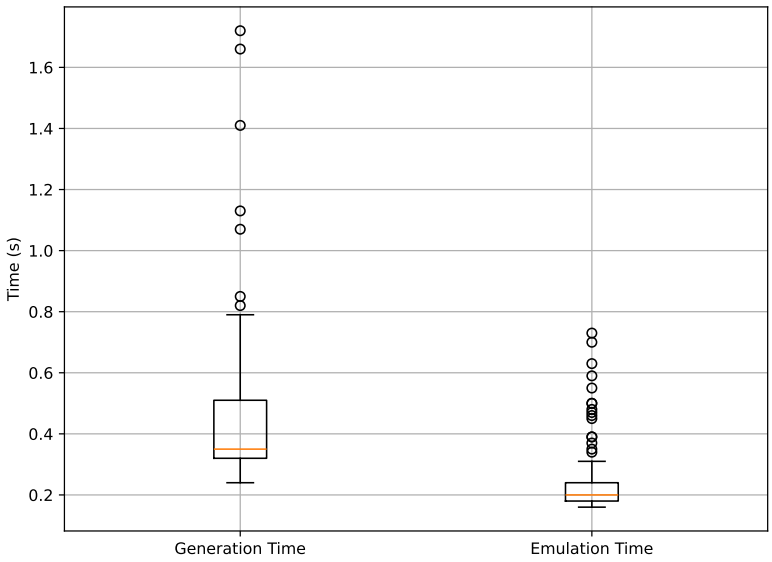

Рис. 8: Боксплоты времени, затраченного на генерацию и исполнение кода с малым диапазоном параметров

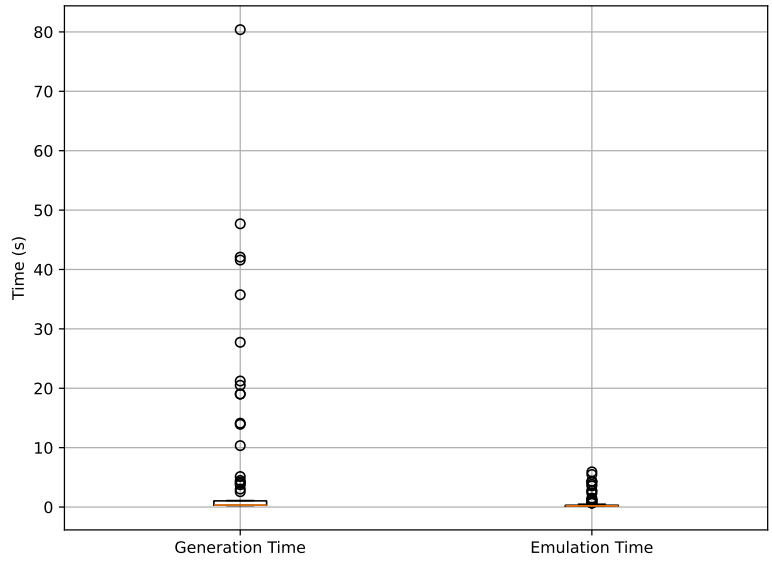

Рис. 9: Боксплоты времени, затраченного на генерацию и исполнение кода с увеличенным диапазоном параметров

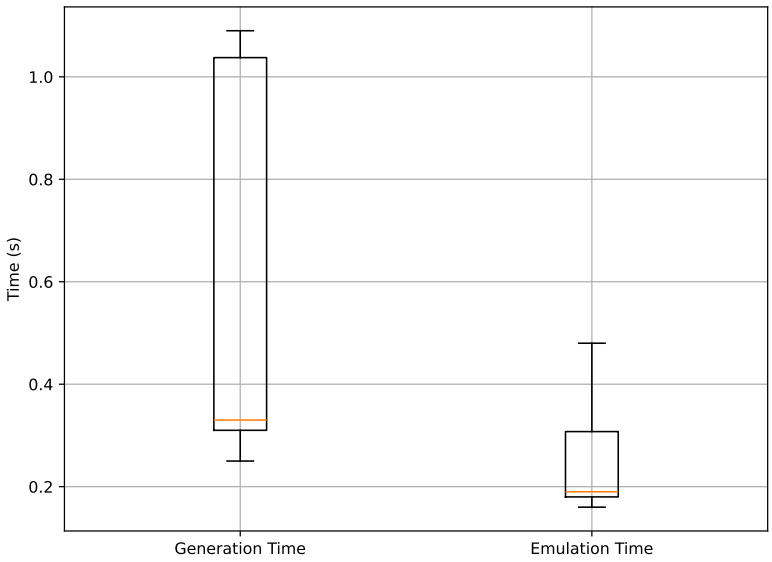

Рис. 10: Боксплоты времени, затраченного на генерацию и исполнение кода с увеличенным диапазоном параметров без аномалий

Заключение

В рамках проведения работы были получены следующие результаты.

- Проведен обзор существующих решений по применению генетического программирования.
- Реализована мутация кода, полученного с помощью `Csmith`.
- Реализована мутация ассемблерного кода.
- Проведено исследование подхода с использованием `Csmith`.

В качестве будущих задач можно выделить следующие пункты.

- Тестирование мутации ассемблерного кода с использованием новых полученных метрик.
- Исследование преобразования кода на C в `AST` с помощью `Clang` для последующего мутирования.

Полный код реализации доступен в публичном репозитории[1].

[1]Репозиторий с реализацией мутаций: https://github.com/SE-Processor-Fuzzing/pfuzz (дата доступа: 29 октября 2024 г.).

Список литературы

[1] 2023.4 IPU Out-of-Band (OOB) - Intel® Processor Advisory. — URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00950.html (дата обращения: 31.10.2024).

[2] DEAP documentation. — URL: https://deap.readthedocs.io/en/master/index.html (дата обращения: 29.10.2024).

[3] Jenetics: Java Genetic Programming library. — URL: https://jenetics.io/ (дата обращения: 29.10.2024).

[4] Jenetics.prog module. — URL: https://github.com/jenetics/jenetics/tree/master/jenetics.prog (дата обращения: 29.10.2024).

[5] Koza John R. Genetic programming as a means for programming computers by natural selection // Statistics and Computing. — 1994. — Vol. 4. — P. 87–112. — URL: https://api.semanticscholar.org/CorpusID:8750149.

[6] Praveen Kumar E, Priyanka S, Sudhakar T. A Review on Vulnerabilities to Modern Processors and its Mitigation for Various Variants // Procedia Computer Science. — 2022. — Vol. 215. — P. 91–97. — 4th International Conference on Innovative Data Communication Technology and Application. URL: https://www.sciencedirect.com/science/article/pii/S1877050922020816.

[7] PyGAD. — URL: https://pygad.readthedocs.io/en/latest/# (дата обращения: 29.10.2024).

[8] Solt Flavien, Ceesay-Seitz Katharina, Razavi Kaveh. Cascade: CPU Fuzzing via Intricate Program Generation // 33rd USENIX Security Symposium (USENIX Security 24). — Philadelphia, PA : USENIX Association, 2024. — . — P. 5341–5358. — URL: https://www.usenix.org/conference/usenixsecurity24/presentation/solt.

[9] Wikipedia contributors. Fitness function — Wikipedia, The Free Encyclopedia. — 2024. — [Online; accessed 29-October-2024]. URL: https://en.wikipedia.org/w/index.php?title=Fitness_function&oldid=1228046354.

[10] Wikipedia contributors. Genetic algorithm — Wikipedia, The Free Encyclopedia. — 2024. — [Online; accessed 29-October-2024]. URL: https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1248419193.

[11] Wikipedia contributors. Genetic programming — Wikipedia, The Free Encyclopedia. — 2024. — [Online; accessed 29-October-2024]. URL: https://en.wikipedia.org/w/index.php?title=Genetic_programming&oldid=1251885449.

[12] Уязвимость в процессорах Intel. — URL: https://habr.com/ru/companies/kaspersky/articles/775148/ (дата обращения: 31.10.2024).

Приложение 1. Конфигурация gem5

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
system.mem_mode = 'timing'
system.mem_ranges = [AddrRange('512MB')]
system.cpu = RiscvTimingSimpleCPU()
system.membus = SystemXBar()
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
system.cpu.createInterruptController()
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
```

Приложение 2. Шаблоны хромосом для тестирования

4.1. Шаблон с малым диапазоном

```
template_config = {
    '--seed': range(1, 10000),
    '--max-funcs': range(1, 10),
    '--max-expr-complexity': range(1, 10),
    '--max-array-len-per-dim': range (1, 10),
    '--max-array-dim': range (1, 10)
}
```

4.2. Шаблон с увеличенным диапазоном

```
template_config = {
    '--seed': range(1, 10000),
    '--max-funcs': range(1, 40),
    '--max-expr-complexity': range(1, 40),
    '--max-array-len-per-dim': range (1, 40),
    '--max-array-dim': range (1, 20)
}
```