# Разработка GC, модель

# План разработки модели

- Определимся с вопросами, сформулированными в конце прошлой лекции, которые относятся к модели.
- Для этого, нужно определиться со степенью детализации.
- Обычно, все количественные детали (конкретные числа, адреса, смещения, представления данных и тд), для моделирования структуры не нужны, от них можно легко абстрагироваться.
- А важны следующие вещи: классификация сущностей + отношения между сущностями (что и формирует абстрактную структуру), определение нужных нам свойств этой структуры (обычно это инварианты, то есть логические формулы, которые должны быть всегда истинны при работе со структурой и из истинности которых мы делаем вывод, что у нас всё корректно), определение нужных нам операций над этой структурой и проверка сохранения свойств операциями (сохранение инвариантов).

# Вопросы с прошлой лекции

- *Как узнать/задать множество корневых объектов?*
- Как узнать где в данных объекта лежат возможные указатели на другие объекты?
- **Как определить корректность алгоритма?**
- **Каким основным свойствам (инвариантам) должен удовлетворять алгоритм, чтобы быть корректным?**
- *Какой контекст сборщика мусора у нас будет?*
- **Какие основные примитивные операции над этим контекстом будут использоваться?**
- **Каким свойствам (инвариантам) должны удовлетворять эти примитивные операции, чтобы общие свойства алгоритма удовлетворялись?**

# Определение корректности

- Тут всё просто: корректным будем считать тот алгоритм, который сохраняет инварианты корректности ☺
- Основная сложность будет определить эти важные инварианты.
- Подумаем, что мы ожидаем от сборщика мусора?
  - Как необходимое условие, наверное, хотелось бы безопасного сборщика мусора – он не должен удалять объекты, с которыми наша программа работает, иначе программе будет тяжело ☺
  - Ещё хотелось бы, чтобы у нас неиспользуемые объекты не засоряли память, то есть сборщик мусора должен удалять неиспользуемые объекты и возвращать память в пул свободной памяти.
- Исходя из этих двух основных наших требований сформулируем свойства GC

# Свойства GC

- При заданном множестве корневых объектов, GC должен точно классифицировать объекты по их достижимости: достижимые и недостижимые
- Какие свойства объектов нам нужны в модели?
  - Ну, так как мы говорим о достижимости и недостижимости, очевидно, что нам нужно отношение достижимости ☺
  - То есть, определим отношение того, что один объект связан с множеством других (оно может быть и пустым) и назовём его Link (Link – можно считать множеством пар объектов (a,b), каждая пара говорит о том что в объекте ‘a’ есть указатель на объект ‘b’ и они связаны)
  - Как мы будем определять достижимые объекты? Наверняка идти по цепочкам указателей в объектах. А как понять, какие объекты мы уже обошли? Введём множество Marked и будем туда складывать объекты, которые мы уже обошли.

# Свойства GC

- Хотелось бы во время обхода объектов GC, давать программе тоже работать, поэтому будем проектировать модель инкрементальной маркировки объектов, а это значит, что нам потребуется понятие времени.
  На этот раз, чтобы время не моделировать в явном виде, возьмём расширение Alloy, которое называется Electrum и которое сочетает возможности статичного моделирования с динамическим.
- Это позволит нам сделать спецификацию чуть проще и понятнее.

# Свойства GC

- Так как между операциями может происходить изменение объектов и их связей между ними пользователем, то нужно будет:
  - Ввести понятие контекста GC, где будем сохранять информацию между вызовом операций. Например, введём множество ToBeChecked, куда будем складывать все объекты, которые собираемся просмотреть и промаркировать.
  - Контроллировать операции пользователя, которые затрагивают объекты, с которыми мы работаем. Это в сборщиках мусора называется барьерами. Например, у уже промаркированного объекта пользователь меняет указатель, значит нам это нужно отследить и тот объект, на который начинает указывать указатель, нужно поместить в множество ToBeChecked, для дальнейшего обхода и проверки.

# Свойства GC

- Итого, мы имеем сущности:
  - Objects – множество всех объектов
  - Link – отношение связи между объектами
  - Marked – множество промаркированных объектов
  - ToBeChecked – множество объектов для проверки
  - RootObjects – множество корневых объектов
- Проверка инвариантов важна только на стадии сборки мусора, то есть, когда на очередной вызов GC ToBeChecked пусто. На этом этапе важно не удалять потенциально используемые объекты и удалить все неиспользуемые.

# Свойства GC

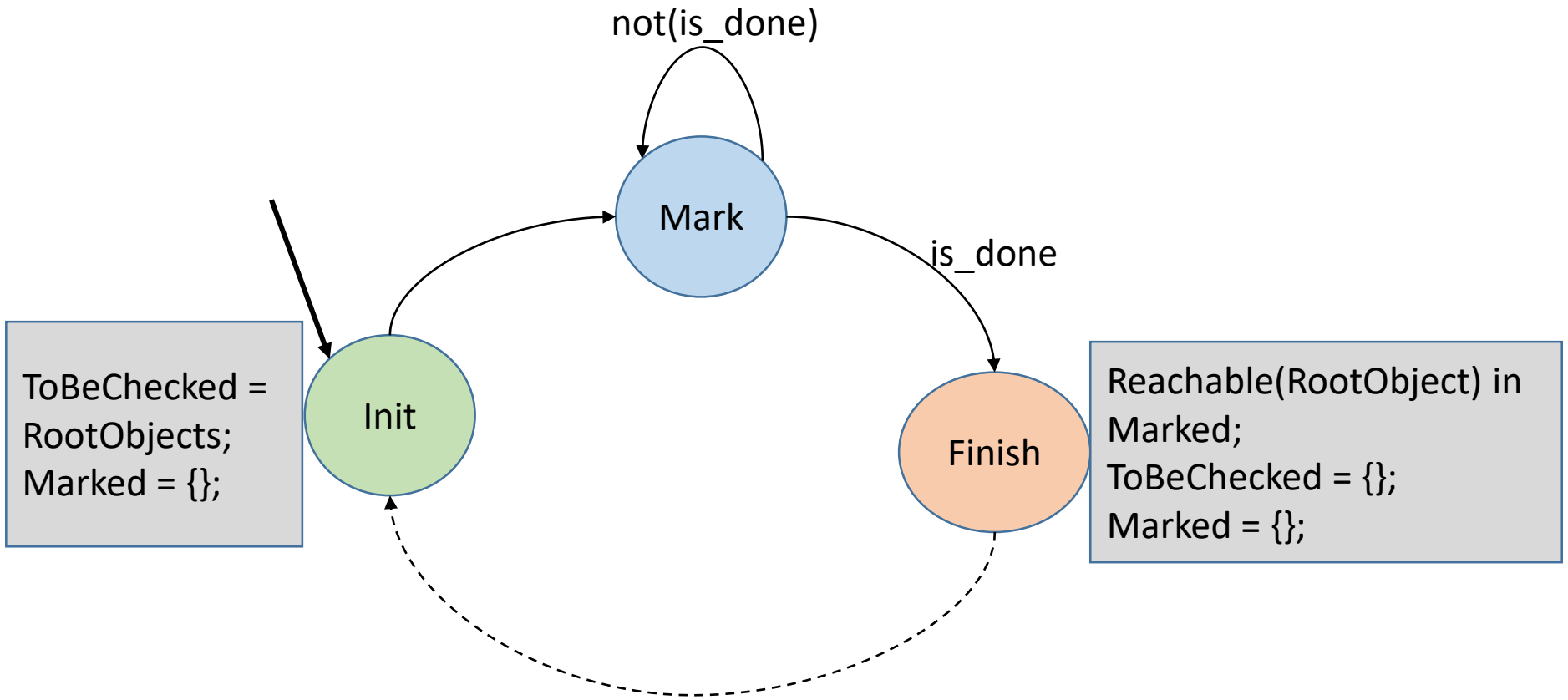
- Предикаты корректности:
  - Все достижимые из RootObjects должны быть в Marked
  - ToBeChecked должно быть пусто
  - Если за весь цикл сборки не было перелиновки объектов пользователем, то все достижимые из RootObjects должны быть равны множеству в Marked.
    Таким образом, из Reachable(RootObject) = Marked следует, что GC не будет оставлять неиспользуемые объекты и память будет возвращаться в пул свободной.

# Операции GC

- Операция, по-сути, одна: gc ☺, но для удобства, разобъём её на более мелкие:
  - init_step – инициализирует цикл gc, помещая всё из RootObjects в ToBeChecked
  - mark_step – маркирует ровно один объект из ToBeChecked, добавляет его в Marked, а связанные с ним в ToBeChecked
  - finish_step – операция удаления всех недостижимых и запуск нового цикла
- Предикаты:
  - is_done – проверяет завершение цикла маркировки

# Барьеры GC

- Перелинковка пользователем:
  - user_link(object_from, object_to) – связывает два объекта
  - user_unlink(object_from, object_to) – разлинковывает объекты
  - user_add_root_object(object) – добавляет в множество корневых
  - user_del_root_object(object) – удаляет




# Автомат GC, упрощенная схема

# Особенности моделирования

- Для проверки условия удаления мусорных объектов нужно будет сделать режим поиска моделей, где бы GC работал без интерференции со стороны пользователя, то есть, где бы отношение Link на протяжении моделирования не менялось бы и не менялось множество RootObjects.
- То есть, моделировать будем два режима:
  - Static – где множество корневых объектов и связи между ними не меняются на протяжении работы сборщика мусора.
  - Dynamic – где множество корневых и связи могут меняться.
- Почему именно два эти режима?
  - Static – позволит проверить, что сборщик мусора не оставляет недостижимых объектов.
  - Dynamic – проверка того, что сборщик мусора не удалит достижимые объекты из-за интерференции с процессом пользователя
- Можно было бы сделать и в одном режиме, но это потребовало бы введения дополнительных механизмов отслеживания изменяемых объектов и связей для учёта в условиях корректности, что сильно бы усложнило модель (и её уже сложно было бы считать простой учебной).

# Особенности моделирования

- Electrum имеет режим работы с бесконечными трассами при использовании бэкенда electrod/NuSMV, но для простоты и наглядности в нашей модели мы будем моделировать всего один цикл работы сборщика мусора
- После инициализации в начальный момент времени, будет выполняться маркировка объектов, пока есть обрабатываемые объекты
- После маркировки будет  произведена сборка недостижимых объектов и сборщик мусора остановится в финальном состоянии

# Отличие Electrum от Alloy

- Наличие ключевого слова ***var*** для обозначения тех сигнатур и отношений, которые могут меняться во времени
- Наличие темпоральных операторов, как для LTL логики будущего, так и для PLTL логики прошлого:
  - always **p** – с данного момента и всегда в будущем
  - eventually **p** – начиная с данного момента, когда-либо в будущем
  - after **p** – в следующий момент времени будет p
  - **p** until **q** – p выполняется до тех пор пока не выполняется q, затем выполняется q
  - **p** releases **q** – q должно выполнятся до p, затем может невыполнятся
  - historically **p** – до текущего момента всегда выполнялось p
  - once **p** – в прошлом хоть раз выполнялось p
  - before **p** – в прошлом моменте p выполнялось
  - **p** since **q** – в прошлом когда-то выполнялось q, а потом постоянно выполнялось q
  - **p** triggered **q** – если когда-то выполнилось p, то после этого выполнялось q

# Преимущества Electrum

- Моделирование динамических процессов намного удобнее и естественнее, чем в Alloy
- Полностью включает в себя всю семантику Alloy
- Содержит более совершенный движок генератора контр-примеров (в некоторых случаях быстрее на порядки)
- Во многих случаях способен заменить пару Alloy + TLA+ или подобную.
  То есть в рамках одного формализма и инструмента можно удобно вести моделирование как структуры системы, так и динамику её развития.

# Ссылки

- Страница проекта Electrum2
- Документация по Electrum2
- Trustworthy Software Design with Alloy