КРАТКОЕ РУКОВОДСТВО ПО TLA+

1. ВВЕДЕНИЕ

TLA+ – это язык для написания моделей распределенных систем и параллельных алгоритмов. Он основан на теории множеств, логике первого порядка и темпоральной логике действий (TLA, temporal logic of actions). Программы, написанные на TLA+ не выполняются компьютером, как программы на обычном языке программирования: они запускаются через средство проверки моделей. Средство проверки моделей (model checker) – это программа, которая исследует все возможные варианты выполнения системы. Вы можете задать поведение и свойство, а model checker проверит, выполнены ли оно для данной модели или нет. В случае, если свойство нарушается, средство предоставит контрпример.

TLA+ использовался для построения моделей различных программ. Самые известные из них это:

- блокировки в ядре Linux;
- диспетчер контейнеров в Docker SwarmKit;
- протоколы консенсуса Paxos, Raft, Tendermint, HotStuff.

Этот документ – небольшое введение в TLA+. В результате ознакомления с ним, читатель научится писать собственные модели алгоритмов, а также проверять их свойства, в том числе и темпоральные (учитывающие временной аспект).

Модель в TLA+ – это **набор состояний и переходов между этими состояниями**, которые задаются в терминах предикатов и действий. Модель описывает поведение системы как последовательность состояний, удовлетворяющих логическим условиям и изменяющихся в результате действий. Для того, чтобы написать модель на TLA+ необходимо задать начальное состояние и систему переходов. Проверить модель на выполнение инвариантов можно задав эти инварианты (в виде предикатов) и запустив model checker.

2. УСТАНОВКА

Простейший способ – установка расширения для Visual Studio Code. Также, консольную версию TLA+ можно установить с помощью команд:

```
$ sudo apt install -y default-jre
$ git clone https://github.com/pmer/tla-bin.git
$ cd tla-bin
$ ./download_or_update_tla.sh
$ sudo ./install.sh
```

3. ПРИМЕР МОДЕЛИ ПРОТОКОЛА НА TLA+

В качестве примера, построим модель с двумя процессами: A и B. Они могут обмениваться сообщениями по ненадежной сети, в которой сообщения могут быть потеряны или переупорядочены. A может отправить B два сообщения: “hello” и “world” в произвольном порядке. В случае, если B получает сначала сообщение “hello”, а затем “world”, он устанавливает статус “Ok”.

Построим модель такого протокола в TLA+. Замечу, что нам нужны будут два файла: первое с расширением .tla (оно содержит саму модель) и второе с расширением .cfg – файл конфигурации TLA+. Файл .tla должен начинаться с линии вида
---- *MODULE Name*----
и заканчиваться линией вида:
==========.
Между этими линиями пишется спецификация. С помощью ключевого слова *EXTENDS* можно импортировать модули из стандартной библиотеки TLA+: Integers, Sequences, FiniteSets, RealTime и др. В нашем примере для работы с последовательностями нам понадобится модуль Sequences:

MODULE *hello_world*

EXTENDS *Sequences*

Как уже было сказано выше, модель в TLA+ – это набор состояний и переходов между этими состояниями. Состояние программы – это набор значений ее переменных. Для объявления переменных используется ключевое слово *VARIABLES:*

VARIABLES
A_msgs,
network,
status,
B_inbox

Переменная *A_msgs* – это множество исходящих сообщений от A, *networks* – множество сообщений в сети, *status* – статус B, *B_inbox* – последовательность сообщений, полученных B. Обратите внимание, мы объявляем только имена переменных без указания их типа. Начальные значения переменных (начальное состояние) пользователь задает оператором *Init*, одновременно задавая тип переменных*:*

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land A\_msgs = \{\} \\
&\quad \land network = \{\} \\
&\quad \land status = \text{“None”} \\
&\quad \land B\_inbox = \langle\rangle
\end{aligned}
$$

В TLA+ возможные переходы между состояниями описываются предикатом над парой из двух состояний: текущего и следующего. Если в какой-либо паре предикат истинен, то model checker проверит его. Фактически, он проверит все переходы, соответствующие предикату. Именно это позволяет средству проверки модели полностью исследовать поведение системы.

Таким образом, мы определяем переходы путем написания действий: операторов, учитывающих текущее и следующее состояние. Поскольку имена переменных одинаковы в обоих состояниях, обновленная переменная *var* обозначается *var'*, то есть добавляется символ “штрих”. Все возможные действия объединяются оператором *Next*:

$$
\begin{aligned}
&Send(m) \triangleq \\
&\quad \land m \notin A\_msgs \\
&\quad \land A\_msgs' = A\_msgs \cup \{m\} \\
&\quad \land network' = network \cup \{m\} \\
&\quad \land \text{UNCHANGED } \langle status,\ B\_inbox \rangle \\
&NetworkLoss \triangleq \\
&\quad \land \exists\, e \in network : network' = network \setminus \{e\} \\
&\quad \land \text{UNCHANGED } \langle status,\ B\_inbox,\ A\_msgs \rangle \\
&NetworkDeliver \triangleq \\
&\quad \land \exists\, e \in network : \\
&\qquad \land B\_inbox' = B\_inbox \circ \langle e \rangle \\
&\qquad \land network' = network \setminus \{e\} \\
&\quad \land \text{UNCHANGED } \langle status,\ A\_msgs \rangle \\
&Receive \triangleq \\
&\quad \land status' = \text{IF } B\_inbox = \langle \text{“hello”},\ \text{“world”} \rangle \text{ THEN “Ok” ELSE “None”} \\
&\quad \land \text{UNCHANGED } \langle network,\ B\_inbox,\ A\_msgs \rangle \\
&Next \triangleq \\
&\quad \lor Send(\text{“hello”}) \\
&\quad \lor Send(\text{“world”}) \\
&\quad \lor NetworkLoss \\
&\quad \lor NetworkDeliver \\
&\quad \lor Receive
\end{aligned}
$$

Рассмотрим действие *Send* с входным параметром *m*. Оно выполняется, когда выполнена конъюнкция:
m не входит во множество A_msgs в текущем состоянии
И A_msgs' (A_msgs в следующем состоянии) *равно объединению A_msgs* (в текущем состоянии) *с m*

И множество network' равно объединению network с m
И status, B_inbox не изменяются между состояниями.

Ключевое слово *UNCHANGED* должно быть использовано для всех неизменяемых переменных чтобы исключить ошибки при проверке, связанные с неоднозначностью.

Действие *NetworkLoss* выполняется, когда существует (квантор существования обозначается *\E*) сообщение в *network*, которое не является элементом *network'* (*network* в следующем состоянии), и остальные переменные не изменяются.

Приведем пример того, как model checker выбирает переходы на примере двух этих действий. Пусть у нас есть текущее состояние: *<<A_msgs = {"hello"}, network = {"hello"}>>*. В свою очередь, *Next == \/ Send ("hello") \/ Send ("world") \/ NetworkLoss*. Другие действия и переменные пока опустим. В таком случае, может быть выполнено действие *Send ("world")*, и тогда следующее состояние будет: *<<A_msgs = {"hello", "world"}, network = {"hello", "world"}>>*. Действие *Send ("hello")* пока не может быть выполнено, так как *"hello"* уже есть во множестве *A_msgs*. Однако, также может быть выполнено действие *NetworkLoss*, что приведет к новому состоянию *<<A_msgs = {}, network = {}>>*. Получается, Next будет истинным в двух случаях и model checker проверит инварианты в обоих соответствующих состояниях.

Вернемся, к исходной модели. Действие *NetworkDeliver* задает переход, при котором сообщение удаляется из сети и добавляется в последовательность принятых сообщений (для добавления элемента в последовательности используется оператор *\o*). Действие *Receive* задаёт переход, при котором B устанавливает статус с помощью оператора *IF-THEN-ELSE*.

4. СПЕЦИФИКАЦИЯ СВОЙСТВ

После построения модели, необходимо специфицировать ее свойства. Глобально, свойства можно разделить на 2 класса:

- *Свойства безопасности (Safety)* гарантируют, что в системе ничего плохого не случится. Они утверждают, что система никогда не перейдёт в недопустимое состояние.

- *Свойства живучести (Liveness)* утверждают, что если система находится в некотором состоянии, то в будущем она обязательно придет к состоянию, которое удовлетворяет нужным условиям. Нарушение свойства живучести не может быть замечено в отдельный момент времени — оно связано с бесконечными временными последовательностями.

Средство проверки модели пробегает по всем возможным состояниям модели и проверяет выполнение заданных свойств в каждом состоянии. Если свойство не выполняется, model checker вернёт контрпример – последовательность состояний, конечный член которой не удовлетворяет заданному свойству. Стандартное средство проверки моделей для спецификаций на TLA+ – TLC Model Checker. TLC пробегает по всем возможным состояниям; в отличие от других средств (например Apalache), мы не можем задать глубину проверки. В случае успешной проверки TLC вернет сообщение: Model checking completed. No error has been found. В ином случае, выдаст контрпример.

Приведем пример инварианта. Инвариант *NothingUnexpectedInNetwork* является Safety-свойством и гарантирует, что каждое сообщение в сети является элементом множества отправленных сообщений:

$$NothingUnexpectedInNetwork \triangleq \forall e \in network : e \in A\_msgs$$

Имея спецификацию модели и инварианта, мы можем запустить проверку. Для этого нам также необходимо создать конфигурационный файл с расширением .cfg следующего содержания:

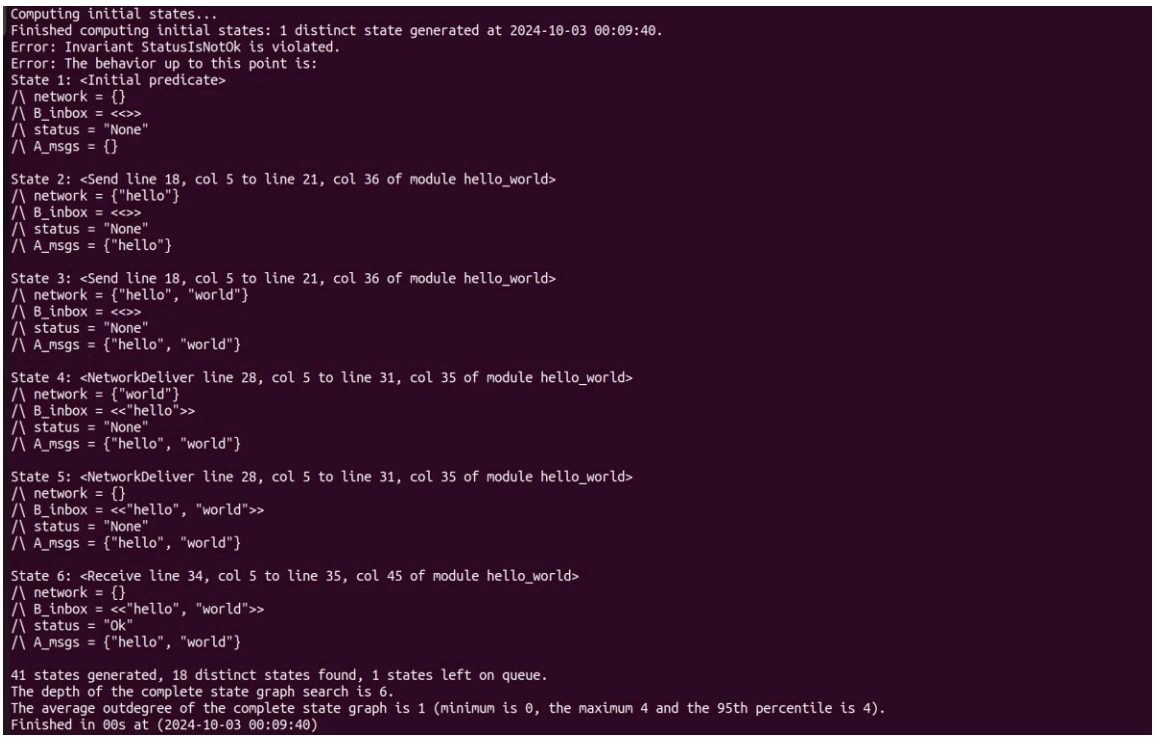
INIT Init
NEXT Next
INVARIANT NothingUnexpectedInNetwork

Запустим проверку командой *tlc hello_world.tla*. В результате получим вывод: Model checking completed. No error has been found, то есть сигнал об успешном выполнении свойства для нашей модели.

Теперь напишем другой инвариант, используя оператор *LET ... IN*.

$$\begin{array}{l} StatusIsNotOk \triangleq \\ \quad \text{LET } IsStatusOk \triangleq status = \text{“Ok”} \\ \quad \text{IN } \neg IsStatusOk \end{array}$$

В этом случае, TLC выведет трассу состояний (см. рисунок). Действительно, в состоянии 6 переменная *status* равна “*Ok”*, то есть инвариант не выполнен.



Также в связи с отсутствием проверки типов в TLC, зачастую задается инвариант *TypeOK*, в котором описываются допустимые значения и типы переменных. Он не является обязательной частью спецификации, но полезен для отладки, а также для повышения читабельности модели.

$$\begin{aligned} TypeOK \triangleq\ & \land A\_msgs \in \text{SUBSET } \{\text{“hello”}, \text{“world”}\} \\ & \land network \in \text{SUBSET } \{\text{“hello”}, \text{“world”}\} \\ & \land status \in \{\text{“Ok”}, \text{“None”}\} \end{aligned}$$

В данном примере можем задать *TypeOK* вышеуказанным способом, где *SUBSET* – показательное множество.

4.1. Liveness-свойства

Немного модифицируем модель, предположив, что сообщения не могут быть потеряны в сети. Также допустим, что B хранит полученные сообщения в виде множества, а не в виде последовательности. Статус “Ok” устанавливается в случае если B получил все возможные сообщения. Модель теперь выглядит следующим образом:

```
VARIABLES
    A_msgs,
    network,
    status,
    B_inbox

CONSTANT
    msgs

vars ≜ ⟨A_msgs, network, status, B_inbox⟩

Init ≜
    ∧ A_msgs = {}
    ∧ network = {}
    ∧ status = "None"
    ∧ B_inbox = {}

Send(m) ≜
     ∧ m ∉ A_msgs
     ∧ A_msgs' = A_msgs ∪ {m}
     ∧ network' = network ∪ {m}
     ∧ UNCHANGED ⟨status, B_inbox⟩

NetworkDeliver ≜
    ∧ ∃ e ∈ network :
        ∧ B_inbox' = B_inbox ∪ {e}
        ∧ network' = network \ {e}
    ∧ UNCHANGED ⟨status, A_msgs⟩

Receive ≜
    ∧ status' = IF B_inbox = msgs THEN "Ok" ELSE "None"
    ∧ UNCHANGED ⟨network, B_inbox, A_msgs⟩

Steps(m) ≜ Send(m) ∨ NetworkDeliver ∨ Receive

Next ≜ ∨ ∃ m ∈ msgs : Steps(m)
       ∨ UNCHANGED vars
```

Для такой модели можно сформулировать Liveness-свойство: система в конце концов (eventually) придет к состоянию, в котором статус будет равен “Ok”. Для его спецификации нам необходим темпоральный оператор <> (eventually).

```
EventuallyStatusIsOK ≜
    LET IsStatusOk ≜ status = "Ok"
    IN  ◇IsStatusOk
```

Темпоральные свойства задаются в файле конфигурации с помощью ключевого слова *PROREPTIES*. При проверке свойств такого типа необходимо учитывать, что в TLA+ по умолчанию любое поведение допускает stuttering, то есть выполнение действия, в котором ничего не происходит и переменные не изменяются. Более того, TLA+ допускает бесконечное количество таких переходов. Это позволяет симулировать работу реальных систем, которые могут остановиться в произвольный момент времени. В связи с этим, для проверки свойств живучести нам необходимо добавить условие честности (fairness condition), то есть предположение о том, что система не останавливается.

- *Weak fairness* означает, что если процесс всегда может прогрессировать, то в конечном итоге он будет прогрессировать.

- S*trong fairness* означает, что если процесс всегда может *периодически* прогрессировать (то есть действия могут выполняться или не выполняться), то в конце концов он добьется прогресса.
- *Weak fairness* влечет за собой s*trong fairness.*

Добавим fairness-условие в нашу спецификацию. *Weak fairness* записывается как темпоральная формула *WF_vars(Action)*, где *vars* – кортеж всех переменных модели.

$$\begin{aligned} Spec \triangleq\ & Init \\ & \wedge \Box[Next]_{vars} \\ & \wedge \forall\, m \in msgs : \mathrm{WF}_{vars}(Steps(m)) \end{aligned}$$

Темпоральный оператор *[]* (always) означает, что *Next* будет выполнен всегда. Это стандартный вид записи спецификации. Запустим проверку новой модели с обновленным файлом конфигурации.

CONSTANT msgs = {"hello", "world"}
SPECIFICATION Spec
INVARIANTS NothingUnexpectedInNetwork
PROPERTIES EventuallyStatusIsOK

В результате TLC не найдет ошибок, model checking будет успешно завершен. Код рассмотренного примера можно найти в репозитории.

5. TLA+ ДЛЯ ПРОТОКОЛОВ КОНСЕНСУСА

В задаче на хакатоне Вам будет предложено построить модель протокола консенсуса и проверить её свойства в средстве проверки модели. Протокол консенсуса – это протокол, решающий проблему достижения консенсуса в распределенных системах, когда все корректные процессы должны принять решение о некотором общем предлагаемом значении.

Пусть у нас есть множество участников *V*, большинство (*2f+1*) из которых подчиняются протоколу, а *f* участников – византийские, то есть могут вести себя произвольно, например, предоставлять ложные или противоречивые сообщения или самопроизвольно отключаются от сети. Пусть также имеется множество значений *S*, и каждый процесс из *V* начинает работу с каким-то начальным значением из *S*. Тогда *протокол решает задачу достижения византийского соглашения (консенсуса)*, если в итоге каждый процесс выбирает значение f_v из множества *S*, и выполнены следующие условия:

1. **Согласованность: все корректно работающие узлы принимают одинаковое значение f_v.**

2. **Корректность: принятое значение было предложено одним из имеющихся узлов.**

3. **Завершаемость: все корректно работающие узлы в конце концов достигают определённого значения.**

Примеры верификации таких протоколов на TLA+:

1) Tendermint
   Ссылна на спецификацию.
2) HotStuff
   Ссылка на статью о формальной верификации протокола HotStuff
   Ссылка на спецификацию
3) Описание и верификацию протокола консенсуса Paxos можно найти в классическом видеокурса по TLA+:
   https://lamport.azurewebsites.net/video/video7.html.

6. ДОПОЛНИТЕЛЬНЫЕ ИСТОЧНИКИ

1) Полная памятка по синтаксису TLA+
2) TLA+ Video Course by Leslie Lamport
3) www.learntla.com
4) Официальный репозиторий с большим количеством примеров спецификаций на TLA+
5) L. Lamport. Specyfying Systems.