NHK$^\sharp$ User's Manual

Noayuki Nagatou[1]
PRESYSTEMS Inc.

nagatou@presystems.xyz

Abstract

NHK$^\sharp$ *is a model checking tool developed by us and published according to GPLv3 and later.*

In general, there are two checking processes within the lifecycle of a system. One is verification of models describing functional specification, and the other is testing of real codes implementing the system. We developed a model checking tool can being used on both processes. Model checking systematically explores the state space of systems. The systems are expressed as the model or real code(or source code) written in programming languages such as C. There are several model checking tools that handle either the models or the real codes, but not both. In testing processes of the real codes, our model checking tool checks both the models and the real codes. The tool executes the binary code on GDB, then examines a composition with the model from which the portions of the model that correspond to the code being executed have been eliminated. A state space of the real code is a set of GDB's breakpoints corresponding to actions to communicate between the eliminated portion and others, so the model is written in a modeling language based on process algebra. This tool enables the re-use of the same model that was used in verification processes.

Contents

1 Installation

Compiling those files needs glib-1.2. You need to install it before doing this. Please refer the manual to install glib-1.2.

1. After installing glib-1.2,
   > tar -xvzf rccs.tar.gz

2. type the following
   > cd rccs/src

3. similarly,
   > make rccs

If you cannot compile then please edit the make file for your configuration. Perhaps, glib is installed into a different place.

2 Run-Time Options

NHK$^\sharp$ allow us to use the following option to control the behaviour of NHK$^\sharp$.

rccs [-tisq] [-f FORMULA] [-d TARGET [ARGUMENTS]] [-m MODEL].

When you give both -d and -f, then I examine the target in the verification mode, and give only -f then I verify a given property for a model. When you don't give both -f and -d, I will run the model in the emulation mode.

- -t : turns the trace flag on.
- -f "formula": specifies a formula and moves to verification mode.
- -d "target" ["arguments"]: specifies a debugging program and moves to collaboration mode.
- -i : provids the interactive execution mode.
- -g : stronG view of the semantics (default).
- -k : weaK view of the semantics.

3 Communication

Let us try to see a message media for the transmission of information in NHK$^\sharp$. One of implementations of message medias is a bounded buffer. The message media discipline is:

- The channel is a bidirection channel and a bounded buffer whose size is 1.
- A sender may always send a message, provided the buffer is not full.

- If the buffer is full then a sender is blocked.
- A receiver may always recive a message, provided the buffer is not empty.
- If the buffer is empty then a receiver is blocked.

The bounded buffer in NHK$^\sharp$ takes one message at a time. We call this buffer a register, and operations write/read. If a buffer is full/empty then write/read operation is blocked. To simplify the description, we consider that the buffer is FIFO. Moreover, communication is connected by bidirectional channel.

The concurrent object is defined by a set of oeprations and a specification that defines the meaning of the object. The object in NHK$^\sharp$ has the following specification.

- The order of recieving messages is equal to the order of sending messages.
- Every value read is written, but not overwritten.
- No value is written twice.
- No value is received twice.
- The order of recieving messages is equal to the order of sending messages.
- Every value read is written, but not overwritten.
- The first action is an output action.

Moreover, the shared object has atomicity (linearizability) property.

A pair of an input action and an matching output action in a linearization sequence atomically behaves. We regard the continious occurence of the matching actions as an atomic communication between an input action and an output action.

3.1 Atomicity Consistency (Linearizability)

Sequences of actions in NHK$^\sharp$ has linearizability property that Herlihy and Wing define in [HW87]. Linearizability requires each operation should appear to "take effect", and the order of nonconcurrent operations should be preserved.

An execution on concurrent objects is modeled by a history, which is a finite sequence of operation invocation and response events. An action involves two operations to communication media. One is to write messages to the media as an output action, and another is to read messgaes from it as an input action. Each operation is a pair consists of an invocation event and the matching response event. An invocation of an operation is written as $\mathrm{inv}(op)$, and also an response event is written as $res(op)$.

DEFINITION 3.1 (Sequencial History)
A history H is sequenctial if:

- The first event of history H is an invocation of an action.

- Each invocation event, except possibly the last, is immediately followed by a matching response event.
- Each response event, except possibly the last, is immediately followed by an invocation event.

Let a be any process P_i or any object X. $H|a$ denotes the prjection of H on a. Two histories are equivalent if for every process P_i, $H|P_i = H'|P_i$. A history is well-formed if a projection on each process is sequential. All histories considered in this paper are assumed to be well-formed.

A history H induces an irreflexive relation $\prec_H$ on a set of operations.

DEFINITION 3.2 (Partial Order on Oeprations)
$op_1 \prec_H op_2$ if $\mathrm{res}(op_1)$ precedes $\mathrm{inv}(op_2)$ in H.

If H is sequential then $\prec_H$ is total.

The next introduces the condition of linearizability.

DEFINITION 3.3 (Linearizability)
There is a sequential history S such that:

- history H is equivalent to some legal sequential history S,
- $\prec_H \subseteq \prec_S$, and
- $\prec_S$ is total order.

Given a linearizable history, there may be more than one linearization. A possiblity for liniarization is a pair $\langle S, P\rangle$, where S is a linearization of a given history and P is a set of pending operations which are not completed to construct S. *Poss* denotes a set of positilities. *Poss* is caputured by the following three axioms.

AXIOM 3.1 (Closure)
if $\langle S, P\rangle \in \mathit{Poss}$ then

$$\forall\,\mathrm{inv}(op) \in P \exists\,\mathrm{res}(op). S\,\mathrm{inv}(op)\,\mathrm{res}(op) \text{ is legal}$$
$$\Rightarrow \langle S\,\mathrm{inv}(op)\,\mathrm{res}(op), P - \{\mathrm{inv}(op)\}\rangle \in \mathit{Poss}$$

The axiom states that if S is a linearization of H , $\mathrm{inv}(op)$ is a pending invocation in H that is not completed to form S, and $S' = S\,\mathrm{inv}(op)\,\mathrm{res}(op)$ is a legal sequential history, then S' is also a linearization of H.

AXIOM 3.2 (Invocation)
$\{\langle S, P\rangle \in \mathit{Poss}\}\,\mathrm{inv}(op)\{\langle S, P \cup \{\mathrm{inv}(op)\}\rangle \in \mathit{Poss}\}$

Axiom Invocation states that any invocation of H is also a linearization of $H\,\mathrm{inv}(op)$.

AXIOM 3.3 (Response)
$\{\langle S, P\rangle \in \mathit{Poss} \text{ and } \mathrm{inv}(op) \notin P \text{ and } \mathrm{res}(op) = \mathrm{last}(S, A)\}\,\mathrm{res}(op)\{\langle S, P\rangle \in \mathit{Poss}\}$

Axiom Response states that any linearization of H in which the pending $\mathrm{inv}(op)$ is completed with $\mathrm{res}(op)$ is also a linearization of $H\,\mathrm{res}(op)$. $\mathrm{last}(S, A)$ is the response to A's last invocation in the sequential history S. An operation completion decides the order of the operation, and then the invocation of the operation in P on *Poss* is removed.

3.2 NHK$^\sharp$ Message Media

The bounded buffer in NHK$^\sharp$ takes one message at a time from that message media decipline. Operation enq, if the buffer is empty, places a message in the buffer, otherwise it is blocked. Operation deq, if the buffer is full, reads a message from the buffer, otherwise it is blocked. The message media decipline leads the axioms. m denotes a state of a message media, and $[v]$ denotes a queue list.

AXIOM 3.4 (Enqueue)

$\{m = \emptyset\}[\mathrm{inv}(\mathrm{enq}(v))/\,\mathrm{res}(\mathrm{enq}(v))]\{m = [v]\}$

AXIOM 3.5 (Dequeue)

$\{m \neq \emptyset \text{ and } m = [v]\}[\mathrm{inv}(\mathrm{deq}())/\,\mathrm{res}(\mathrm{deq}())]\{m = \emptyset \text{ and } v = \mathrm{res}(\mathrm{deq}())\}$

THEOREM 3.1
The message media with the enqueue and dequeue operations is an atomic object.

Proof Sketch. Let H is a history on the message media. NHK$^\sharp$ in current implementation immediately completes each operation. Therefore, every H is a sequential history, and for each history, $\prec_H$ is total order. Thus, the message media is an atomic object. Because atomicity is a local propety (see Theorem 1 in [HW87]) the whole of the message media is also atomic. ■

Moreover, we show that every transition between a sender and a receiver in NHK$^\sharp$ is atomic.

THEOREM 3.2
A transition between a sender and a receiver in NHK$^\sharp$ is atomic.

Proof Sketch. ■

4 Syntax of Model Description Language

4.1 Primitive Types

RCCS has two types, integer and string. Operator `+,-,/,*` are defined over these types.

4.2 Special Actions and Processes

Action `key` and `display` are special actions. `key` is used as an input action corresponding to the standard input, and `display` is used as an output action corresponding to the standard output. `accept` is used to represents accept states of an automaton produced from a given LTL formula in verification mode. Those special actions distiguish between upper letters and lower letters.

Process `ZERO` and `STOP` are a special process that means do nothing, not terminate the whole. NHK$^\sharp$ uses a special process `INIT_STATE` in verification mode. The process menas an initial state of an automaton for a LTL formula given by users. `ABORT` terminates the whole process for an automaton, but users cannot use in a model. Those special processes do not distinguish between upper letters and lower letters.

4.3 Scope

In expression (define P (x) body), the scope of x becomes body. In expression ($\overline{a}$(x): body), the scope of x becomes body.

Unfortinately, we use dynamic binding. In future, we will fix it.

4.4 Syntax of the Discription Language

```
Agent_Exp     ::= ( define ID ( ID_Seq ) Agent_Exp )
               |  ( bind ID Strings )
               |  ( globalvar IDs Strings )
               |  ( if ( B_exp ) Agent_Exp Agent_Exp )
               |  ( A_Binary_Exp )
               |  (  )
A_Binary_Exp  ::= A_Binary_Exp ++ A_Label_Exp
               |  A_Binary_Exp || A_Label_Exp
               |  A_Label_Exp
A_Label_Exp   ::= A_Label_Exp [ ID_Seq ]
               |  A_Label_Exp { Relabel_Seq }
               |  A_Unary_Exp
A_Unary_Exp   ::= ID ( Value_Seq ) : A_Unary_Exp
               |  ID : A_Unary_Exp
               |  ~ ID ( Value_Seq ) : A_Unary_Exp
               |  ~ ID : A_Unary_Exp
               |  ID ( Value_Seq )
               |  ID
               |  ( Agent_Exp )
ID_Seq        ::= ID_Seq , ID
               |  ID_Seq , ~ ID
               |  ID
               |  ~ ID
               |  epsilon
Value_Seq     ::= Value_Seq , B_Exp
               |  B_Exp
Relabel_Seq   ::= Relabel_Seq  , ID / ID
               |  Relabel_Seq  , ~ ID / ~ ID
               |  ID / ID
               |  ~ ID / ~ ID
B_Exp         ::= B_Exp | C_Exp
               |  B_Exp & C_Exp
               |  C_Exp
C_Exp         ::= C_Exp < V_Exp
               |  C_Exp <= V_Exp
               |  C_Exp > V_Exp
               |  C_Exp >= V_Exp
               |  C_Exp = V_Exp
               |  V_Exp
V_Exp         ::= V_Exp + V_Term
               |  V_Exp - V_Term
               |  V_Term
V_Term        ::= V_Term * V_Unary_Exp
               |  V_Term / V_Unary_Exp
               |  V_Term % V_Unary_Exp
```

```
                |  V_Unary_Exp
V_Unary_Exp     ::= ! V_Unary_Exp
                |  Fact
Fact            ::= Iconst | Strings | TRUE | FALSE | ID
                |  ( B_Exp )
```

4.5 Sorts and Derivatives of Processes

We make the notion of sort which is a little difference from syntactic sort in [Mil89]. If actions of a process P and all its derivatives

$$\begin{array}{rcl} \mathit{Sort}(\mathit{ZERO}) & = & \emptyset \\ \mathit{Sort}(a\colon P) & = & \{a\} \cup \mathit{Sort}(P) \\ \mathit{Sort}(\sim a\colon P) & = & \{\sim a\} \cup \mathit{Sort}(P) \\ \mathit{Sort}(P{+}{+}Q) & = & \mathit{Sort}(P) \cup \mathit{Sort}(Q) \\ \mathit{Sort}(P \| Q) & = & \mathit{Sort}(P) \cup \mathit{Sort}(Q) \\ \mathit{Sort}(\text{if } b\ P\ Q) & = & \mathit{Sort}(P) \cup \mathit{Sort}(Q) \\ \mathit{Sort}(A) & = & \mathit{Sort}(P) \text{ if } A \overset{\text{def}}{=} P \end{array}$$

We distiguish names and co-names, and the observation of pairs of matching actions is possible because it is used to determine a next transition of the entier of processes.

We make the notion of syntactic derivative of sort $\mathit{Sort}(P)$ which is used in the definition of operatinal semantics. All derivatives lie in $\mathit{Sort}(P)$ of P is syntactic immediate derivatives. Describing an action sequence $\alpha_1, \cdots, \alpha_n$ in which each element is an element of $\mathit{Sort}(P)$ of P, we call a process of the sequence and a set of them writes $\mathit{Derivative}(P)$. $\mathit{Derivative}(\mathit{ZERO})$ is $\{\mathit{ZERO}\}$.

4.6 RCCS Operatinal Semantics

In this section, we define the semantics of the discription language. For this purpose, we define a machine. Before we continue, we define context k. The following grammar defines a set of contexts. The context contains a hole, written in $\Box$, in the place of one subexpression.

$$\begin{array}{rcl} k & ::= & \Box \\ & | & P \| \Box \\ & | & \Box \| P \end{array}$$

$k[P]$ means to replace the hole in k with P, where P is a process defined in Section 4.4. Moreover, we define a function Env that maps all free variables to closures. This function is called a environment, and a closure is a pair of an expression and a environment. environments Env and colosures c have mutually recursive definitions.

$$\begin{array}{rcl} \mathit{Env} & ::= & \text{a list of pairs } \langle (X, c), \cdots \rangle \\ c & ::= & \{(P, \mathit{env}) | FV(P) \subset \mathrm{dom}(\mathit{Env})\} \end{array}$$

$\mathit{Env}[X \leftarrow c]$ means that (X, c) is added into Env, that is, $\{(X, c)\} \cup \{(Y, c') | (Y, c') \in \mathit{Env} \text{ and } X \neq Y\}$.

In addition to the above definition, we use the following sets.

$$ch \ ::= \ \text{a list of pairs } \langle (\mathit{Name}, \langle m_1, \cdots, m_n \rangle), \cdots \rangle$$

Channel ch represents a set of channels with which processes communicate each other. $ch[a \leftarrow v]$ means that (a, v) is added into ch, that is, $\{(a, v)\} \cup \{(b, v') | (b, v') \in ch \text{ and } a \neq b\}$.

A state of the machine is a triple $[(\mathit{exp}, \mathit{env}), k, ch]$ which is appended channel ch to machines in [FF02]. We define single steps of an evaluation function for the description language. A bijection function on names associtės a name a to a co-name, written with $\overline{a}$. Notice that $\overline{\overline{\alpha}} = \alpha$. $\mathit{Env}[\langle x \rangle \leftarrow \langle v \rangle, \langle y \rangle \leftarrow \langle w \rangle, \cdots]$ means $(\mathit{Env}[\langle x \rangle \leftarrow \langle v \rangle])[\langle y \rangle \leftarrow \langle w \rangle] \cdots$, and $ch[\alpha \leftarrow \langle v \rangle, \beta \leftarrow \langle w \rangle, \cdots]$ means $(ch[\alpha \leftarrow \langle v \rangle])[\beta \leftarrow \langle w \rangle], \cdots$.

$$\text{Output(1)}\,\frac{(a \leftarrow \langle v\rangle) \notin ch}{[(\sim a(v)\colon P, Env), \Box, ch] \overset{\sim a}{\to} [(P, Env), \Box, ch[a \leftarrow \langle v\rangle]]}$$

$$\text{Output(2)}\,\frac{(a \leftarrow \langle v\rangle) \notin ch}{[(\sim a(v)\colon P, Env_1), k[(Q, Env_2)\|\Box], ch] \overset{\sim a}{\to} [(Q, Env_2), k[\Box\|(P, Env)], ch[a \leftarrow \langle v\rangle]]}$$

$$\text{Output(3)}\,\frac{(a \leftarrow \langle v\rangle) \notin ch}{[(\sim a(v)\colon P, Env_1), k[\Box\|(Q, Env_2)], ch] \overset{\sim a}{\to} [(Q, Env_2), k[(P, Env_1)\|\Box], ch[a \leftarrow \langle v\rangle]]}$$

$$\text{Input}\,\frac{(a \leftarrow \langle v\rangle) \in ch}{[(a(x)\colon P, Env), k, ch[a \leftarrow \langle v\rangle]] \overset{a}{\to} [(k[P], Env[\langle x\rangle \leftarrow \langle v\rangle]), \Box, ch]}$$

$$\text{Sum(1)}\,\frac{[(P, Env_1), k, ch] \overset{\alpha}{\to} [(P', Env_1'), k', ch']}{[(P, Env_1){+}{+}(Q, Env_2), k, ch] \overset{\alpha}{\to} [(P', Env_1'), k', ch']}$$

$$\text{Sum(2)}\,\frac{\begin{array}{c}[(Q, Env_2), k, ch] \overset{\alpha}{\to} [(Q', Env_2'), k', ch']\\ Derivative([(P, Env_1), k, ch]) = \{P\}\end{array}}{[((P, Env_1){+}{+}(Q, Env_2), k, ch] \overset{\alpha}{\to} [(Q', Env_2'), k', ch']}$$

$$\text{Com(1)}\,\frac{[(P, Env_1), k, ch] \overset{\alpha}{\to} [(P', Env_1'), k[\Box\|Q], ch']}{[(P, Env_1)||(Q, Env_2), k, ch] \overset{\alpha}{\to} [(P', Env'), k[\Box\|Q], ch']}$$

$$\text{Com(2)}\,\frac{\begin{array}{c}[(Q, Env_2), k, ch] \overset{\alpha}{\to} [(Q', Env_2'), k[P\|\Box], ch']\\ Derivative([(P, Env_1), k, ch]) = \{P\}\end{array}}{[((P, Env_1)||(Q, Env_2), k, ch] \overset{\alpha}{\to} [(Q', Env_2'), k[P\|\Box], ch']}$$

$$\text{Com(3)}\,\frac{\begin{array}{c}[(P, Env_1), k[\Box\|(Q, Env_2)], ch] \overset{\overline{\alpha}}{\to} [(P', Env_1'), k', ch']\\ [(Q, Env_2), k[(P, Env_1)\|\Box], ch] \overset{\alpha}{\to} [(Q', Env_2'), k'', ch'']\end{array}}{[(P, Env_1)\|(Q, Env_2), k, ch] \overset{(\overline{\alpha}, \alpha)}{\to} [(k[(P', Env_1')\|(Q', Env_2')], \Box, ch]}$$

$$\text{Ins}\,\frac{[(P(x), Env[A \leftarrow P][\langle x\rangle \leftarrow \langle v\rangle]), k, ch] \overset{\alpha}{\to} [(P', Env[A \leftarrow P][\langle x\rangle \leftarrow \langle v\rangle]), k, ch]}{[(A(v), Env[A \leftarrow P]), k, ch] \overset{\alpha}{\to} [(P', Env[A \leftarrow P][\langle x\rangle \leftarrow \langle v\rangle]), k, ch]}$$

$$\text{If(1)}\,\frac{\begin{array}{c}\text{eval-val}(B, Env)\\ [(T, Env), k, ch] \overset{\alpha}{\to} [(T', Env'), k', ch']\end{array}}{[(\text{if } B\ T\ E, Env), k, ch] \overset{\alpha}{\to} [(T', Env'), k', ch']} \qquad \text{If(2)}\,\frac{\begin{array}{c}\neg\text{eval-val}(B, Env)\\ [(E, Env), k, ch] \overset{\alpha}{\to} [(E', Env'), k', ch']\end{array}}{[(\text{if } B\ T\ E, Env), k, ch] \overset{\alpha}{\to} [(E', Env'), k', ch']}$$

$$\text{Res}\,\frac{\alpha, \overline{\alpha} \notin \{\alpha, \cdots\} \qquad [(P, Env), k, ch] \overset{\alpha}{\to} [(P', Env'), k', ch']}{[(P[\alpha, \cdots], Env), k, ch] \overset{\alpha}{\to} [(P'[\alpha, \cdots], Env'), k', ch']}$$

$$\text{Rel(1)}\,\frac{[(P, Env), k, ch] \overset{\alpha}{\to} [(P', Env'), k', ch[\alpha \leftarrow \langle v\rangle]]}{[(P\{\alpha'/\alpha, \cdots\}, Env), k, ch] \overset{\alpha'}{\to} [(P'\{\alpha'/\alpha, \cdots\}, Env'), k', ch[\alpha' \leftarrow \langle v\rangle]]}$$

$$\text{Rel(2)}\,\frac{[(P, Env), k, ch[\alpha \leftarrow \langle v\rangle]] \overset{\alpha}{\to} [(P', Env'), k', ch]}{[(P\{\alpha'/\alpha, \cdots\}, Env), k, ch[\alpha \leftarrow \langle v\rangle]] \overset{\alpha'}{\to} [(P'\{\alpha'/\alpha, \cdots\}, Env'), k', ch]}$$

$$\text{ZERO(1)}\,\frac{}{[(\text{ZERO}, Env), \Box, ch] \not\to} \qquad \text{ZERO(2)}\,\frac{}{[(\text{ZERO}, Env), k, ch] \to [k[(\text{ZERO}, Env)], \Box, ch]}$$

5 Coroutine-Like Sequencing

An important application of coroutine is discrete event simulation, where coroutine may be used to simulate parallel processes within the framework of a sequential program.

6 Syntax of Formulae

We use LTL to describe goal properties of processes. We first assume that a trace has initial states and is a finite sequence of states. We write the length of trace $\sigma = s_0\, s_1\, \cdots\, s_n$ to $|\sigma|$ in which $|\sigma|$ is $n+1$. We write the suffix of $\sigma = s_0\, s_1\, \cdots\, s_i\, \cdots\, s_n$ starting at i as $\sigma^{i..} = s_i\, \cdots\, s_n$, and the i^{th} state as σ^i.

We assume a vocabulary $x, y, z, \cdots$ of variables for data values. For each state, variables are assigned to a single value. A state formula is any well-formed first-order formula constructed over the given variables. Such state formulas are evaluated on a single state to a boolean value. If the evaluation of state formula p becomes true over s, then we write $s[p] = \texttt{tt}$ and say that s satisfies p, where `tt` and `ff` are truth values, denoting *true* and $false$ respectively. Let φ and ψ be temporal formulas, a temporal formula is inductively constructed as follows:

- a state formula is a temporal formula,
- the negation of a temporal formula $\neg\varphi$ is a temporal formula,
- $\varphi \vee \psi$ and $\varphi \wedge \psi$ are temporal formulas, and
- $\Box\,\varphi$, $\Diamond\,\varphi$, $\circ\,\varphi$, and $\varphi\,\mathcal{U}\,\psi$ are temporal formulas.

We provide the formal syntax with BNF notation.

```
Start               ::= StartFormula

StartFormula        ::= StartFormula /\ PathFormula
                    |   StartFormula \/ PathFormula
                    |   StartFormula -> PathFormula
                    |   ! StartFormula
                    |   PathFormula

PathFormula         ::= <> PathFormula
                    |   [] PathFormula
                    |   PathFormula U StartFormula
                    |   X PathFormula
                    |   Proposition

Proposition         ::= Proposition &  Atom
                    |   Proposition |  Atom
                    |   Proposition -> Atom
                    |   ! Proposition
                    |   Atom

Atom                ::= Atom =  Exp
                    |   Atom <  Exp
                    |   Atom >  Exp
                    |   Atom <= Exp
                    |   Atom >= Exp
                    |   Boolean
                    |   Exp

Exp                 ::= Exp + Term
```

```
                  |   Exp - Term
                  |   Term

Term              ::= Digits
                  |   Strings
                  |   Id
                  |   ( StartFormula )

Boolean           ::= tt
                  |   ff

Action            ::= Id
                  |   ~ Id
```

7 Semantics of Property Description Language

We next define two semantics of temporal formulas over a finite trace according to [EFH$^+$03]. If trace σ satisfies property φ, then we write $\sigma \models \varphi$.

7.1 Strong Semantics

Furthermore,

- if p is a state formula, then $\sigma \models p$ iff $\sigma^0[\![p]\!] = \texttt{tt}$ and $|\sigma| \neq 0$,
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$,
- $\sigma \models \varphi \vee \psi$ iff $\sigma \models \varphi$ or $\sigma \models \psi$,
- $\sigma \models \varphi \wedge \psi$ iff $\sigma \models \varphi$ and $\sigma \models \psi$,
- $\sigma \models \Box\varphi$ iff for all $0 \leq i < |\sigma|$, $\sigma^{i..} \models \varphi$,
- $\sigma \models \Diamond\varphi$ iff there exists $0 \leq i < |\sigma|$ such that $\sigma^{i..} \models \varphi$,
- $\sigma \models \circ\,\varphi$ iff $\sigma' \models \varphi$ where $\sigma' = \sigma$ if $|\sigma| = 1$ and $\sigma' = \sigma^{1..}$ if $|\sigma| > 1$,
- $\sigma \models \varphi\,\mathcal{U}\,\psi$ iff there exists $0 \leq k < |\sigma|$ s.t. $\sigma \models \psi$ and for all $j < k$, $\sigma \models \varphi$.

A formula φ is satisfiable if there exists a sequence σ such that $\sigma \models \varphi$. Given set of traces T and formula φ, φ is valid over T if for all $\sigma \in T$, $\sigma \models \varphi$.

7.2 Weak Semantics

Furthermore,

- if p is a state formula, then $\sigma \models p$ iff $\sigma^0[\![p]\!] = \texttt{tt}$ or $|\sigma| = 0$,
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$,
- $\sigma \models \varphi \vee \psi$ iff $\sigma \models \varphi$ or $\sigma \models \psi$,
- $\sigma \models \varphi \wedge \psi$ iff $\sigma \models \varphi$ and $\sigma \models \psi$,
- $\sigma \models \Box\varphi$ iff for all $0 \leq i < |\sigma|$, $\sigma^{i..} \models \varphi$,
- $\sigma \models \Diamond\varphi$ iff there exists $0 \leq i < |\sigma|$ such that $\sigma^{i..} \models \varphi$,
- $\sigma \models \circ\,\varphi$ iff $\sigma' \models \varphi$ where $\sigma' = \sigma$ if $|\sigma| = 1$ and $\sigma' = \sigma^{1..}$ if $|\sigma| > 1$,
- $\sigma \models \varphi\,\mathcal{U}\,\psi$ iff there exists $0 \leq k < |\sigma|$ s.t. $\sigma \models \psi$ and for all $j < k$, $\sigma \models \varphi$.

8 Relationships between Models and Formlae

In this subsection, we describe the relationship between algebraic models and LTL formulas. The modeling language enables us to pass values via input prefix $\alpha(e)$ and output prefix $\overline{\alpha}(x)$ with the same name. Execution of $\alpha(e)$ produces value v of e. Execution of $\overline{\alpha}(x)$ causes a single assignment to x. Furthermore, the execution of two actions causes atomic assignment $x\colon = v$, that is, communication between two agents produces a new state by changing the values of the variables. This is similar to the first paragraph in Section 3.3 of [LS84, page 290].

This atomic assignment changes states, and we represent the change as $s[v/x]$, which denotes a change in the values of x in s to v. A state is a mapping from variables to values. Assuming that $\mathtt{Var_E}$ is a set of variables that appears in prefixes in agent E with range $\mathtt{V}$, $s\colon Var_E \to \mathtt{V}$. For example, the evaluation $s[\![x = y]\!]$ of $x = y$ at s becomes $s[\![x]\!] = s[\![y]\!]$, and at $s[v/x]$, $s[v/x][\![x]\!] = s[v/x][\![y]\!]$, i.e., $v = s[\![y]\!]$.

Therefore, communication between agents produces a sequence of assignments, which then produces a sequence of state changes called a trace. Let a set of traces produced by agent E be T. If for all traces $\sigma \in T$, $\sigma \models \varphi$, then we state that φ is valid over E and write $E \models \varphi$.

8.1 Transition of Automata

A Büchi automaton m contains of five components:

- A finite set of states, denoted Q.
- A finite set of input symboles, denoted Σ.
- A transition function δ that takes a state and an input symbol, and returns a next state. If q is a state, and s is an input symbol, then $\delta(q, a)$ returns state p.
- A start state q_0 is a state in Q.
- A set of accepting states Q_∞ is a subset of Q.

In this paper, an input symbol becomes a state of a model. We talk about an automaton m in *five-tuple* notation: $(Q, \Sigma, \delta, q_0, Q_\infty)$.

Now, we need to make the notion of the language that an automaton accepts. To do this, we define an extended transition function. The extended transition function constructed from δ is called $\hat{\delta}$. We define $\hat{\delta}$ by induction on the length of an input string σ, as follows:

$$\hat{\delta}(q, \sigma) = \begin{cases} q & \text{if } |\sigma| = 0 \\ \delta(\hat{\delta}(q, \sigma^{..n-1}), \sigma^n) & \text{if } 0 < |\sigma| < \omega. \end{cases}$$

We define the laguage $\mathcal{L}(m)$ of automaton m. Let $INF(\rho)$ be a set of automaton states that appear infinitely often in while reading σ, then σ is accepted by m if and only if $INF(\rho) \cap Q_\infty \neq \emptyset$. Thus,

$$\mathcal{L}(m) = \{\sigma \mid \rho^0 = q^0, \forall i\colon \rho^i = \hat{\delta}(q_0, \sigma^{..i}), \text{ and } INF(\rho) \cap Q_\infty \neq \emptyset\}.$$

η depends on weak or strong semantics. In weak semantics, η is q that is regarded as an element of Q_∞. In strong semantics, η is Λ, where Λ is inconsistency.

8.2 Correspondence between Models and Formulae

We describe a correspondence between a model and a Büchi automaton of a formulae of a property which the model are required. The correspondece is expressed with Hoare triple: {P}α{P'}, where P and P' are boolean predicates, and α is an action which a model performs.

An automaton m enters an automaton state q_j if there exists a history containing program state s and m is transformed from q_i into q_j by reading s. We define *correspondence invariant* by induction [AS87].

DEFINITION 8.1 (Correspondece Basis)

$\forall i\colon q_j \in Q$ and $(Init_\pi \wedge T_{0j}) \Rightarrow C_j$,

where $Init_\pi$ is the initial states of model π.

DEFINITION 8.2 (Correspondece Induction)
$\forall\alpha\colon\forall i\colon \alpha\in A\cup\overline{A}$ and $q_i\in Q$ and $\{C_i\}\alpha\{\wedge_{q_j\in Q}(T_{ij}\Rightarrow C_j)\}$.

8.3 Proving Safety Properties

A model π written by our model language has the form: $\pi=\pi_1\|\cdots\|\pi_n$. Processes synchronize and communicate using input actions and output actions. For a channel a, a value of exp and a variable var, execution of an output action

$$\overline{a}(exp)$$

causes the transfer of the value of exp, and execution of a matching input action

$$a(var)$$

,which some other process performs, causes recieve from channel a. an input action is deleyed until some matching output action.

Two matching actions are executed as an atomic action which causes an assignment to var:

$$var\ =\ exp.$$

The atomic assignment which consists of two actions $\overline{a}$ and a occurs as a free-standing statement. We regard the atomic assignment above as a fragment of models.

The set of atomic actions which make up program π is denoted $\alpha[\pi]$. If π is composed of fragments $\pi_1,\cdots,\pi_n$ then:

$$\alpha[\pi]=\alpha[\pi_1]\cup\cdots\cup\alpha[\pi_n]\cup\alpha[a_1]\cup\cdots\cup\alpha[a_n].$$

The tool gurantee matching semantics between input and output actions. For example, $P_1\|P_2$ in which each process is defined as follows does not produce input-output pairs $(\overline{a}(1),a(x))$ and $(\overline{a}(2),a(y))$:

$$\begin{array}{rcl} P_1 & = & a(x)\colon ZERO{+}{+}\overline{a}(0)\colon a(y)\colon ZERO,\\ P_2 & = & \overline{a}(2)\colon ZERO{+}{+}a(z)\colon \overline{a}(1)\colon ZERO.\end{array}$$

We present the following proof rules for all possible constructs of processes, according to [AFdR80].

$$\frac{}{\{p\}a(x)\{q\}}\ \text{Input}$$

This axiom corresponds to A.1 in [AFdR80]. The post assertion in this axiom will be checked against a corresponging output action with which some process cooperates.

$$\frac{}{\{p\}\overline{a}(y)\{q\}}\ \text{Output}$$

This axiom may look strange since it has no side-effect. We introduce this axiom corresponding to A.2' in [AFdR80] because the modelling language allows output actions without matching inputs. Communication $\{p\}\alpha(x)\colon P\|\overline{\alpha}(y)\colon Q\{q\}$ does not derivates an arbitrary predicate. The form of q restricts to the formula of $\{p\}\alpha(x)\colon P\|\overline{\alpha}(y)\colon Q\{x=y\wedge p\}$ where x is not free in p.

Action prefixes mean sequencial execiton of actions. This syntactical structure provids the following rule.

$$\frac{\{p\}\alpha\{p'\}\ \{'p\}P\{q\}}{\{p\}[\alpha\colon P]\{q\}}\ \text{Sequence}$$

The meaning of the following rule is that the post condition of summation must be estabilished along each possible path.

$$\frac{\{p\}P\{q\}\ \{p\}Q\{q\}}{\{p\}P{+}{+}Q\{q\}}\ \text{Summation}$$

Using these axioms and rules, we can establish the proof for a formula for each process.

We now present a proof rule and an axiom for communication among processes. The rule to used to deduce a property of $P \| Q$ has the following form:

$$\frac{\{p_1\}P\{q_1\} \ \{p_2\}Q\{q_2\}}{\{p_1 \wedge p_2\}P \| Q\{q_1 \wedge q_2\}} \text{ Composition}$$

The meaning of this rule is that proofs cooperate to help each other proof to validate the post conditions of input/output actions. We shall need the following axiom to establish cooperation:

$$\frac{}{\{True\}[a(x)\colon P \| \overline{a}(y)\colon Q]\{x = y\}} \text{ Communication}$$

References

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3):359–385, July 1980.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. In *Distributd Computing*, pages 117–126. Springer-Velag, 1987.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer Berlin Heidelberg, 2003.

[FF02] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi (Utah CS6520 Version), 2002.

[HW87] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM.

[LS84] Leslie Lamport and Fred B. Schneider. The "Hoare Logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):281–296, April 1984.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.