Welcome to the world of NHK♯

—Install Manual and Specification—

Naoyuki 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 **2**

2 Run-Time Options **2**

3 Communication **3**
3.1 Atomicity Consistency (Linearizability) . . . 3
3.2 NHK$^\sharp$ Message Media . . . 5
3.3 Synchronization . . . 6

4 Syntax of Model Description Language **8**
4.1 Primitive Type . . . 8
4.2 Special Actions and Processes . . . 8
4.3 Scope . . . 8
4.4 Syntax of the Description Language . . . 9
4.5 Sorts and Derivatives of Processes . . . 10
4.6 RCCS Operational Semantics . . . 10

5 Syntax of Formulae **11**

6 Semantics of Property Description Language **12**
6.1 Strong Semantics . . . 12
6.2 Weak Semantics . . . 12

7 Relationships between Models and Formulae **13**
7.1 Transition of Automata . . . 13
7.2 Correspondence between Models and Formulae . . . 13
7.3 A Proof System for The Modelling Language . . . 14
7.4 Proving Safety Properties . . . 17

8 Exploring State Space **19**

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 [-tigk] [-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. Option -i provides interactive mode for users. When evaluation reaches to a binary operator in process expression, you can choose a left or right operand. The evaluation resumes after that choice. Option -g and -k switch the semantics of LTL formulae. -g turns into strong view and -k turns into weak view.

- -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.
- -m "model" : specifies a source file for a model written in RCCS (Not Yet).
- -i : provides the interactive execution mode.
- -g : stronG view of the semantics of formulae (default).
- -k : weaK view of the semantics of formulae.

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 receive 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.

A concurrent object is defined by a set of operations and a specification that defines the meaning of the object. Message channels in NHK$^\sharp$ has the following specification.

- One output action matches to one input action.
- One input action match's to one output action.
- Every value read is written, but not overwritten.
- The first action of a history is an output action.
- An output action write values to a channel, provided a empty channel.
- An input action read values from a channel, provider a channel is not empty.
- No value is written twice.
- No value is read twice.

Moreover, the shared object has atomicity property (linearizability). A contiguous occurrence of the matching actions is an atomic communication among matching actions. A pair of an input action and an matching output action in a sequential history atomically behaves.

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

messages 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 inv(op), and also an response event is written as $res(op)$.

DEFINITION 3.1 (Sequencial History)
A history H is sequential 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 projection 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 possibility for linearization 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. $Poss_H$ denotes a set of possibilities for history H. $Poss$ is captured by the following three axioms.

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

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

The axiom states that if S is a linearization of H , 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)

$\{\langle S, P\rangle \in Poss\}\,\mathrm{inv}(op)\{\langle S, P \cup \{\mathrm{inv}(op)\}\rangle \in Poss'\}$

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

AXIOM 3.3 (Response)

$\{\langle S, P\rangle \in Poss \text{ and } \mathrm{inv}(op) \notin P \text{ and } \mathrm{res}(op) = \mathrm{last}(S)\}\,\mathrm{res}(op)\{\langle S, P\rangle \in Poss'\}$

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 discipline. 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. A specification is a set of axioms. The message media discipline leads the axioms. m denotes a state of a message media, $[v]$ denotes a queue list, and v is a items in a queue.

AXIOM 3.4 (Enqueue)

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

AXIOM 3.5 (Dequeue)

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

The above axioms imply the following Theorem 3.1.

THEOREM 3.1

If m is empty initially then

$$0 \leq \mathrm{length}(m) \leq 1,$$

where length provides the number of items in buffer m.

Proof Sketch. Operation enq, if the buffer is not empty then it is blocked, and deq, if the buffer is empty then it is blocked. When $\mathrm{length}(m)$ is empty initially then $\mathrm{length}(m)$ is at most 1. ■

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 property the whole of the message media is also atomic (see Theorem 1 in [HW87]). The message media with the enqueue and dequeue operations is an atomic object. Thereby, every histories of events for the message media is complete.

3.3 Synchronization

Synchronization in NHK$^\sharp$ starts at an output action and terminates at an input action. Output action a is an invocation $\mathrm{inv}(a)$, and input action is a response $\mathrm{res}(a)$. Partial order on channels $a \prec_H b$ is that for channel a, b, $a \prec_H b$ if $\mathrm{res}(a)$ precedes $\mathrm{inv}(b)$ in H. Each action is implemented by a sequence of events on the above bounded FIFO buffer: $\mathrm{inv}(a) \equiv \mathrm{inv}(\mathrm{enq}(v))\,\mathrm{res}(\mathrm{enq}(v))$ and $\mathrm{res}(a) \equiv \mathrm{inv}(\mathrm{deq}())\,\mathrm{res}(\mathrm{deq}())$. We show that the implementation is correct.

The following axioms provide the semantics of operations on channel.

AXIOM 3.6 (Output Action)
$\{\mathrm{length}(ch) = 0\}\,\mathrm{inv}(a)\{\mathrm{length}(ch) = 1\}$

AXIOM 3.7 (Input Action)
$\{\mathrm{length}(ch) = 1\}\,\mathrm{res}(a)\{\mathrm{length}(ch) = 0\}$

where $\mathrm{length}(ch)$ is the number of messages on a channel.

For any history H, $\mathrm{inv}(a)$ is not contiguous to other output action $\mathrm{inv}(a)$. For example, if events $\mathrm{inv}(a(v))$, $\mathrm{inv}(a(v'))$, $\mathrm{res}(a(v))$, $\mathrm{res}(a(v'))$ lie within H then $\mathrm{inv}(a(v))\,\mathrm{res}(a(v))\,\mathrm{inv}(a(v'))\,\mathrm{res}(a(v'))$, and $\mathrm{inv}(a(v'))\,\mathrm{res}(a(v'))\,\mathrm{inv}(a(v))\,\mathrm{res}(a(v))$ are acceptable. But, $\mathrm{inv}(a(v))\,\mathrm{inv}(a(v'))\,\mathrm{res}(a(v'))\,\mathrm{res}(a(v))$, and $\mathrm{inv}(a(v'))\,\mathrm{inv}(a(v))\,\mathrm{res}(a(v))\,\mathrm{res}(a(v'))$ are not acceptable.

LEMMA 3.1
An invocation $\mathrm{inv}(a)$ for channel a does not contiguous to $\mathrm{inv}(a)$, and also a response $\mathrm{res}(a)$ does not contiguous to $\mathrm{res}(a)$.

Proof Sketch. Describe a sequential history H for channel a. $\mathrm{inv}(a)$ is partial. If $\mathrm{length}(a) = 1$ then the invocation is blocked. From Axiom 3.6, if $\mathrm{length}(a) = 0$ then $\mathrm{length}(a) = 1$ after $\mathrm{inv}(a)$ occurs. From Axiom 3.7, $\mathrm{res}(a)$ is invoked only if $\mathrm{length}(a) = 1$. If $\mathrm{length}(a) = 0$ then an response of a is blocked. Thereby, for any history H, H is sequential. Thus, $\mathrm{inv}(a)$ ($\mathrm{res}(a)$) is not contiguous. ■

Lemma 3.1 stats that all history must be sequential. The opposite of Lemma 3.1, i.e. there are no contiguous invocations in a sequential history, is explicit by the definition of sequential history.

THEOREM 3.2
One output action corresponds with one input action, and one input action corresponds with one output action.

Proof Sketch. Pick a derivation $\langle S, P\rangle \in Poss_m$ showing that H is linearizable. From Lemma 3.1 $\mathrm{res}(a)$ (is an input action) is contiguous to $\mathrm{inv}(a)$ (is an output action), and $\mathrm{inv}(a)$ is contiguous to $\mathrm{res}(a)$ on S. From Axiom 3.1 Closure, if $\mathrm{inv}(a) \in P$, then $\langle S\,\mathrm{inv}(a)\,\mathrm{res}(a), P - \{\mathrm{inv}(a)\}\rangle \in Poss_m$. Thus, all output action has a matching input action, and all input action has a matching output action. ■

Theorem 3.2 implies the following theorem.

THEOREM 3.3
No value is written twice, and no value is read twice.

Proof Sketch. From Lemma 3.1 all history is sequential. From Theorem 3.2 subhistory $\mathrm{inv}(a(v))\,\mathrm{inv}(a(v))$ is not invoked. A subhistory $\mathrm{res}(a(v))\,\mathrm{res}(a(v))$ is also invoked. Thus, same values are not written into a channel and read from a channel. ■

THEOREM 3.4
If an invocation $\mathrm{inv}(a(v))$ precedes $\mathrm{inv}(a(v'))$ and $\mathrm{res}(a(v'))$ has occurred, then $\mathrm{res}(a(v))$ must precede $\mathrm{inv}(a(v'))$.

Proof Sketch. Pick a derivation $\langle S, P\rangle \in Poss_m$ showing that H is linearizable. From Theorem 3.1, $\mathrm{res}(a(v))$ must precedes $\mathrm{inv}(a(v'))$, and must adapt Axiom 3.1 for $\mathrm{inv}(a(v))$ and $\mathrm{res}(a(v))$ in S. Thus, v on channel a by $\mathrm{res}(a(v))$ does not vanish until $\mathrm{res}(a(v))$ is invoked. ■

Theorem 3.4 implies that messages do not spontaneously vanish form a channel.

THEOREM 3.5
If a response $\mathrm{res}(a)$ is invoked then $\mathrm{inv}(a)$ has been invoked before $\mathrm{res}(a)$ and $\mathrm{res}(a)$ does not precede $\mathrm{inv}(a)$.

Proof Sketch. Pick a derivation $\langle S, P\rangle \in Poss_m$ showing that H is linearizable. $\mathrm{res}(a)$ occurs at m in H. From Axiom 3.3 Response, $\{\langle S, P\rangle \in Poss$ and $\mathrm{inv}(op) \notin P$ and $\mathrm{res}(op) = \mathrm{last}(S)\}\,\mathrm{res}(op)\{\langle S, P\rangle \in Poss'\}$. H is legal only if $\mathrm{inv}(a)$ occur at $i(< m)$. Axiom 3.1 application is included. Then, for $i \leq j \leq m$, $\langle S', P'\rangle \in Poss_j$ and $\mathrm{inv}(a) \in P'$. Also, $\langle S'\,\mathrm{inv}(a)\,\mathrm{res}(a), P' - \{\mathrm{inv}(a)\}\rangle \in Poss_j$. Thus, $\mathrm{res}(a)$ does not precede $\mathrm{inv}(a)$. ■

Channels in NHK$^\sharp$ are implemented by queue. Each output action is implemented by enq operation on queue, and each input action is implemented by deq operation. A history is an event sequence interleaved actions with operations of the implementation.

An implementation is correct if for every history H $H|a$ is linearizable. Let H be a interleaved history. We focus on showing that for all $H|m$

$$\langle S_m, P_m\rangle \in Poss_{H|m} \Rightarrow \langle S_a, P_a\rangle \in Poss_{H|a}.$$

We define an abstract function: $\mathcal{A}(m) : M \to Ch$ so that it maps each state of the implementation to a state of an abstract data. M is a set of states of queue and Ch is a set of states of channel.

$$\mathcal{A}(m) = \{a|\,(m = a = [v]) \vee (m = a = \emptyset)\},$$

The order of items in buffer does not need because the queue implementation must satisfy Theorem 3.1 $0 \leq \mathrm{length}(m) \leq 1$.

THEOREM 3.6
A transmission via action is atomic.

Proof Sketch. Association between a message media m and a channel a

$$\begin{aligned}\mathrm{inv}(a(v)) &\equiv \mathrm{inv}(\mathrm{enq}(v))\,\mathrm{res}(\mathrm{enq}(v)),\\ \mathrm{res}(a(v)) &\equiv \mathrm{inv}(\mathrm{deq}(v))\,\mathrm{res}(\mathrm{deq}(v)).\end{aligned}$$

Then, by Axiom 3.4 and Axiom 3.5

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

and

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

These sequences satisfy Axiom 3.6 and Axiom 3.7.

We need to show that if $H|m$ is linearizable then $H|a$ is also linearizable. It suffices to show that $\mathcal{A}(m)$ is included in a set of results which S_a causes. Pick $\langle S_m, P_m\rangle \in Poss_{H|m}$. If the last inference is Axiom 3.1 for enq(v), then the later events of S_m are $\mathrm{inv}(\mathrm{enq}(v))\,\mathrm{res}(\mathrm{enq}(v))$, and the last event of $H|a$ is $\mathrm{inv}(a(v))$. Also $m = [v]$, so $\mathcal{A}(m) = [v]$. At this moment $\langle S_a, P_a\rangle \in Poss_{H|a}$ and $\mathrm{inv}(a(v)) \in P_a$. By Theorem 3.1 $\mathrm{inv}(\mathrm{deq}(v))\,\mathrm{res}(\mathrm{deq}(v))$ is contiguous to $\mathrm{inv}(\mathrm{enq}(v))\,\mathrm{res}(\mathrm{enq}(v))$. Then, $H|a\ \mathrm{res}(a(v))$ and $S_a\ \mathrm{inv}(a(v))\,\mathrm{res}(a(v))$, thereby $a = \emptyset$ or $[v]$. Therefore $\mathcal{A}(m) \in \{\emptyset, [v]\}$.

Moreover, if the last inference is Axiom 3.1 for deq(v), then the later events of S_m are $\mathrm{inv}(\mathrm{deq}(v))\,\mathrm{res}(\mathrm{deq}(v))$, and the last event of $H|a$ is $\mathrm{res}(a(v))$. Also $m = \emptyset$, so $\mathcal{A}(m) = \emptyset$. At this moment $\langle S_a, P_a\rangle \in Poss_{H|a}$, thereby $a = \emptyset$. Therefore $\mathcal{A}(m) \in \{\emptyset\}$.

By Theorem 3.1 $\mathrm{inv}(\mathrm{enq}(v'))\,\mathrm{res}(\mathrm{enq}(v'))$ is contiguous to $\mathrm{inv}(\mathrm{deq}(v))\,\mathrm{res}(\mathrm{deq}(v))$. Thus $\mathcal{A}(m)$ is included in a set of states which linearization cause. ■

4 Syntax of Model Description Language

4.1 Primitive Type

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 distinguish 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 means 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.

`bind` and `define` are special forms. `bind` is used to connect external systems. For example, to connect servers or clients via socket interface, and to bind an action to a command in source codes via GDB debugger. `ID` in form `(bind ID String)` becomes an action in agent expressions.

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.

4.4 Syntax of the Description Language

```
Command_Form  ::= ( define ID ( ID_Seq ) Agent_Exp )
                |  ( definit ( ID_Seq ) ( Value_Seq ) )
                |  ( load ( Strings ) )
                |  ( quit )
                |  ( bind ID Strings )
                |  ( Agent_Exp )
                |  ( )
Agent_Exp     ::= 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

$$\begin{aligned}
\mathrm{Sort}(\mathrm{ZERO}) &= \emptyset \\
\mathrm{Sort}(a\colon P) &= \{a\} \cup \mathrm{Sort}(P) \\
\mathrm{Sort}(\sim a\colon P) &= \{\sim a\} \cup \mathrm{Sort}(P) \\
\mathrm{Sort}(P{+}{+}Q) &= \mathrm{Sort}(P) \cup \mathrm{Sort}(Q) \\
\mathrm{Sort}(P \| Q) &= \mathrm{Sort}(P) \cup \mathrm{Sort}(Q) \\
\mathrm{Sort}(\text{if } b\ P\ Q) &= \mathrm{Sort}(P) \cup \mathrm{Sort}(Q) \\
\mathrm{Sort}(A) &= \mathrm{Sort}(P) \texttt{if } A \stackrel{\mathrm{def}}{=} P
\end{aligned}$$

We distinguish 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 entire of processes.

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

4.6 RCCS Operational Semantics

In this section, we define the semantics of the model discription language. For this purpose, we define an abstract machine. 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.

$$\begin{aligned}
k \quad ::= \quad & \Box \\
| \quad & P \| \Box \\
| \quad & \Box \| P
\end{aligned}$$

$k[P]$ 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. environment *env* and colosure c have mutually recursive definitions.

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

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

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

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

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 $[(exp, env), k, ch]$ 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 assocites a name a to a co-name, written with $\overline{a}$. Notice that $\overline{\overline{\alpha}} = \alpha$. $env[\langle x \rangle \leftarrow \langle v \rangle, \langle y \rangle \leftarrow \langle w \rangle, \cdots]$ means $(env[\langle x \rangle \leftarrow \langle v \rangle])[\langle y \rangle \leftarrow \langle w \rangle] \cdots$, and $ch[\alpha \leftarrow \langle v \rangle, \beta \leftarrow \langle w \rangle, \cdots]$ means $(ch[\alpha \leftarrow \langle v \rangle])[\beta \leftarrow \langle w \rangle], \cdots$. We show the operational semantics of RCCS in Fig. 1

5 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
```

6 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 $\sigma \models \varphi$.

6.1 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 $0 \leq i < |\sigma|$, $\sigma^{i\cdot\cdot} \models \varphi$,
- $\sigma \models \Diamond\varphi$ iff there exists $0 \leq i < |\sigma|$ such that $\sigma^{i\cdot\cdot} \models \varphi$,
- $\sigma \models \circ\varphi$ iff $\sigma' \models \varphi$ where $\sigma' = \sigma$ if $|\sigma| = 1$ and $\sigma' = \sigma^{1\cdot\cdot}$ 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 $\sigma \in T$, $\sigma \models \varphi$.

6.2 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 $0 \leq i < |\sigma|$, $\sigma^{i\cdot\cdot} \models \varphi$,
- $\sigma \models \Diamond\varphi$ iff there exists $0 \leq i < |\sigma|$ such that $\sigma^{i\cdot\cdot} \models \varphi$,
- $\sigma \models \circ\varphi$ iff $\sigma' \models \varphi$ where $\sigma' = \sigma$ if $|\sigma \models 1$ and $\sigma' = \sigma^{1\cdot\cdot}$ if $|\sigma| > 1$,
- $\sigma \models \varphi \,\mathcal{U}\, \psi$ iff there exists $0 \leq k < |\sigma|$ s.t. $\sigma \models \psi$ and for all $j < k$, $\sigma \models \varphi$.

7 Relationships between Models and Formulae

In this section, 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. The execution of $\alpha(e)$ produces value v of e, and the execution of $\overline{\alpha}(x)$ causes a single assignment to x. Furthermore, the execution of two actions causes atomic assignment $x := v$, 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 $\mathrm{Var}_{\mathtt{E}}$ is a set of variables that appears in prefixes in agent E with range $\mathtt{V}$, $s\colon \mathrm{Var}_E \to \mathtt{V}$. 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 $E \models \varphi$.

7.1 Transition of Automata

A LTL formula by which a property of a model is given transforms to a Büchi automaton. A Büchi automaton m contains of five components:

- A finite set of states, denoted Q.
- A finite set of input symbols, 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:

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

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

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

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.

7.2 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 correspondence is expressed with Hoare triple: $\{\mathrm{p}\}\alpha\{\mathrm{q}\}$, where p and q are boolean predicates, and α is an action which a model performs. To show that a program π satisfies m, we demonstrate a correspondence between m and π. We define a correspondence invariant C_i for each state q_i. the correspondence invariant C_i is a predicate that holds on a model state s if 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 formally define *correspondence invariant* by induction [AS89].

THEOREM 7.1 (Correspondece Basis)
$\forall j\colon q_j \in Q$ and $(Init_\pi \wedge T_{0j}) \Rightarrow C_j$,
where $Init_\pi$ is the initial states of model π.

THEOREM 7.2 (Correspondece Induction)
$\forall \alpha\colon \forall i\colon \alpha \in A \cup \overline{A}$ and $q_i \in Q$ and $\{C_i\}\alpha\{\wedge_{q_j \in Q}(T_{ij} \Rightarrow C_j)\}$.

In order to establish that model π satisfies property p, we must show that every history σ of π accepts by m for a property. There are two ways that m fails to accept a history σ of π:

1. m attempts an undefined transition when reading σ.
2. m never enters an accepting state after some finite prefix of σ.

Thus, it suffices that (1) and (2) are impossible. Two obligations ensure that (1) is impossible. First, we must show that m can make some transition from its start state upon reading the first state in a history.

THEOREM 7.3 (Transition Basis)
$Init_\pi \Rightarrow \vee_j(q_j \in Q$ and $T_{0j})$

Second, we must show that m can always make a transition upon reading subsequent states in a history. If m is in state q_i then the model state read by m satisfies a correspondence invariant C_i. To avoid an undefined transition, any atomic action α that is then executed must transform the model state so that one of the transition predicates T_{ij} emanating from q_i holds. This is guaranteed by

THEOREM 7.4 (Transition Induction)
For all $\alpha(\in \mathcal{A})$, and for all i $(q_i \in Q)$,

$$\{C_i\}\alpha\{\vee_{j:q_j \in Q} T_{ij}\}$$

Finally, we present conditions to ensure that m cannot reject σ by failing to enter accepting state infinitely often (scenario (2) above). A set Q' of automaton states is a subset of Q that contains no accepting states. Let Q_{INF} be a set of states that infinitely often appear on a history of a program, if m reject a history then Q_{INF} restricts to Q'. To ensure that Q_{INF} of a history is not restricted to Q', we construct a variant function $v_k : Q \times S \to N$, where N is Natural numbers. When $v_k(q, s) = 0$, then q is in Q'.

THEOREM 7.5 (Knot Exit)
$\forall q_i \in Q'\colon (v_k(q_i, s_i) = 0) \Rightarrow \neg C_i$.

THEOREM 7.6 (Knot Variance)
For all $\alpha(\in \mathcal{A})$, and for all $q_i \in k$,

$$\{C_i \wedge 0 < v_k(q_i) = V\}\alpha\{\vee_{j:q_j \in Q} T_{ij} \wedge C_j \Rightarrow v_k(q_j) < V\}$$

We show later that those theorems hold in NHK$^\sharp$.

7.3 A Proof System for The Modelling Language

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 receive from channel a. An input action is delayed 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:

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

If π is clear by context then π is omitted.

The modelling language guarantees matching semantics between input and output actions. All change of variables appears at execution of action, not in body. For example, consider $P_1 \| P_2$ in which each process is defined as follows:

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

It does not produce a change of variables except input-output pairs. If changes except the pairs are available, axiom Communication is meaningless.

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 corresponding 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 $\{p\}\alpha(x)\colon P \parallel \overline{\alpha}(y)\colon Q\{q\}$ does not derives an arbitrary predicate. The form of q restricts to the formula of $\{p\}\alpha(x)\colon P \parallel \overline{\alpha}(y)\colon Q\{x = y \wedge p\}$ where x is not free in p.

Generalizing axiom Output provides the following axiom:

$$\frac{}{\{p\}P\{p\}}\ \text{Preservation,}$$

if free variables in p is not changed in execution of P.

Action prefixes mean sequential execution of actions. This syntactical structure provides the following rule.

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

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

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

Subsequently we use the following well-known axioms and proof rules.

$$\frac{}{\{p\}\,\mathrm{ZERO}\{p\}}\ \text{Zero}$$

$$\frac{\{p \wedge b\}P_1\{q\} \quad \{p \wedge \neg b\}P_2\{q\}}{\{p\}[(\text{if } b\ P_1\ P_2)]\{q\}}\ \text{Alternation}$$

$$\frac{p \to p' \quad \{p'\}P\{q'\} \quad q' \to q}{\{p\}P\{q\}} \text{ Consequence}$$

$$\frac{\{p\}P\{q\} \quad \{p\}P\{r\}}{\{p\}P\{q \wedge r\}} \text{ Conjunction}$$

$$\frac{\{p[t/z]\}P\{q\}}{\{p\}P\{q\}} \text{ Substitution}$$

$$\frac{\{p\}P\{q\} \quad A \stackrel{\text{def}}{=} P}{\{p\}A\{q\}} \text{ Omission}$$

Using these axioms and rules, we can establish a proof for formulae of a process. A proof start by the following form:

$$\frac{\{p_1\}P_1\{q_1\} \quad \{p_2\}P_2\{q_2\} \quad \cdots}{\{p_1 \wedge \cdots \wedge p_n\}P_1 \| P_2 \| \cdots \{q_1 \wedge \cdots \wedge q_n\}}$$

Each proof cooperates if they help to validate the post-conditions of input-output mentioned in those proofs. The meaning of this rule is that proofs cooperate to help each other proof to validate the post conditions of input/output actions. More formally, the proofs of $\{p_i\}P_i\{q_i\}$ for $1 \leq i \leq n$ cooperate if

1. the assertions used in the proof contain no variables to change in P_j where $j \neq i$, and
2. $\{\text{pre}_1 \wedge \text{pre}_2\}\overline{a}\colon P \| a\colon Q\{\text{post}_1 \wedge \text{post}_2\}$ holds whenever $\{\text{pre}_1\}\overline{a}\colon P\{\text{post}_1\}$ and $\{\text{pre}_2\}a\colon Q\{\text{post}_2\}$ are taken from the proof of $\{p_i\}P\{q_i\}$ and $\{p_j\}Q\{q_j\}$.

We shall need the following axiom to establish cooperation:

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

We introduce inference rules for cooperation because they cannot produce proof trees.

$$\frac{\{\text{True}\}[a(x) \| \overline{a}(y)]\{x = y\}}{\{p\}a(x)\{p \wedge (x = y)\}} \text{ Cooperation1,}$$

and

$$\frac{\{\text{True}\}[a(x) \| \overline{a}(y)]\{x = y\}}{\{p\}\overline{a}(y)\{p \wedge (x = y)\}} \text{ Cooperation2.}$$

The modelling language in NHK$^\sharp$ allows self-communication, for example $(\overline{a}(1)\colon a(x)\colon \text{ZERO})$. Using the system above, we can prove it:

$$\frac{\text{True} \to \text{True} \quad \dfrac{}{\{\text{True}\}\overline{a}(1)\{\text{True}\}} \quad \dfrac{\dfrac{\{\text{True}\}[a(x) \| \overline{a}(1)]\{x = 1\}}{\{\text{True}\}a(x)\{x = 1\}} \quad \dfrac{}{\{x = 1\}\,\text{ZERO}\{x = 1\}}}{\{\text{True}\}a(x)\colon \text{ZERO}\{x = 1\}} \quad (x = 1) \to (x = 1)}{\dfrac{\{\text{True} \wedge \text{True}\}\overline{a}(1)\colon a(x)\colon \text{ZERO}\{x = 1 \wedge \text{True}\}}{\{\text{True}\}\overline{a}(1)\colon a(x)\colon \text{ZERO}\{x = 1\}}}$$

7.4 Proving Safety Properties

Using axioms and rules in Section 7.3, we establish several proofs of assertions in Section 7.2. Variables occur in input and output actions present state variables of a model. Let $\mathrm{Var} = \{x, y, \cdots\}$ be a set of state variables. When two processes cooperate through an input action $a(x)$ and a matching output action $\overline{a}(v)$, a state of a model is changed by the assignment $x := v$ which is atomic. The assignment as $x := v$ denotes a state change from env to $env[v/x]$.

The following explains the correspondence between instruments in Section 7.2 and in the operational semantics in Section 4.6. Correspondence invariant C_i predicates:

- a model changes its state to s_i and
- an automaton m reaches to q_i reading s_i.

A state s_i can be described by giving values of state variables in Var. A valuation is a function that maps variables in Var to values in a domain D. State s_i corresponds to a valuation. Environment env provided by transition within matching communication in a model is also a function that maps variables in Var to values in a domain D. Therefore, we regards an environment env_i in NHK$^\sharp$ interpreter as a state s_i. We can write a formula that is true on a valuation. Therefore, we define a formula Env_i for an environment as follows:

$$Env_i \equiv (\wedge_i (v_i = a_i)).$$

Env_i is equivalent to C_i. Env_0 is {True} and corresponds to $Init_\pi$.

Consider a pair of a model state and an automaton state for a property that the model should satisfy. Let $\langle env_i, q_i \rangle$ be a pair of a model state and an automaton state. When the pair exists, C_i clearly holds on the pair. If a model changes states according to Fig. 1, env_i appears in a configuration on the execution history of the model. In verification mode, NHK$^\sharp$ translates a LTL formula to a process expression which has a special constant process as initial state to represent an automaton for the LTL formula. Each transition of an automaton is indicated with form (`if` (B_i) (E_T) (E_F)). If $env_j \models B_i$ then the automaton reaches to E_T, otherwise E_F. B_i on env_j is checked when an input and an output action are matched. The semantics of `if` operator in verification mode are provided as follow:

$$\mathrm{IF}(1)\frac{(P, env) \overset{[\alpha,\overline{\alpha}]}{\rightarrow} (P', env') \quad env' \models B \quad \neg\text{is-if}(E_F)}{\langle [(P, env), k, ch], (\texttt{if}\ B\ E_T\ E_F) \rangle \rightarrow \langle [(P', env'), k', ch'], E_T \rangle}$$

$$\mathrm{IF}(2)\frac{(P, env) \overset{[\alpha,\overline{\alpha}]}{\rightarrow} (P', env') \quad env' \models B \quad \text{is-if}(E_F)}{\langle [(P, env), k, ch], (\texttt{if}\ B\ E_T\ E_F) \rangle \rightarrow \langle [(P', env'), k', ch'], \mathrm{eval}(E_T, env) \rangle}$$

$$\mathrm{IF}(3)\frac{(P, env) \overset{[\alpha,\overline{\alpha}]}{\rightarrow} (P, env')' \quad env' \not\models B \quad \neg\text{is-if}(E_F)}{\langle [(P, env), k, ch], (\texttt{if}\ B\ E_T\ E_F) \rangle \rightarrow \langle [(P', env'), k', ch'], E_F \rangle}$$

$$\mathrm{IF}(4)\frac{(P, env) \overset{[\alpha,\overline{\alpha}]}{\rightarrow} (P, env')' \quad env' \not\models B \quad \text{is-if}(F)}{\langle [(P, env), k, ch], (\texttt{if}\ B\ E_T\ E_F) \rangle \rightarrow \langle [(P', env'), k', ch'], \mathrm{eval}(E_F, env) \rangle}$$

where is-if(E) holds if the most outer operator of E is `if`. If `Abort` which is a special constant process without arguments occurs, NHK$^\sharp$ explores other execution paths of a model.

$$\mathrm{ABORT}\frac{}{\langle [(P, env), k, ch], (\texttt{Abort}) \rangle \rightarrow \langle [\emptyset, \emptyset, \emptyset], (\texttt{Abort}) \rangle},$$

where $[\emptyset, \emptyset, \emptyset]$ means to terminate exploration on a path.

Transition predicate T_{ij} is derived from transition function $\delta : (Q \times S) \rightarrow 2^Q$ such that $q_j \in \delta(q_i, s_j)$. T_{ij} holds for all s_j such that $q_j \in \delta(q_i, s_j)$. T_{ij} holds on $\langle env_j, (\texttt{if}\ B_i\ P_i\ Q_i) \rangle$ if and only if $env_j \models B_i$. When the continuation in the configuration contains a closure, then $env_j = env_j^P \cup env_j^Q$. If pairs of the same variable name exist then the value of a variable becomes non-deterministic.

We show that NHK$^\sharp$ satisfies Theorem 7.1. The predicate depends on T_{0j} because we assume $Env_0 = \mathrm{True}$. If T_{0j} then $env_j \models B_i$. Env_j obviously holds at model state s_j. Thus, C_j holds.

Theorem 7.2 is shown by a proof system in Section 7.3. Let $Env_j \equiv Env_i \wedge (x = y)$ and $env_j = env_i[x \leftarrow y]$. We show the proof tree, and then retrieve the following inferences on $a(x)\colon P$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathrm{True}\}[a(x)\|\overline{a}(y)]\{x = y\}}{\{Env_i\}a(x)\{env_j \models B_i\}}}{\dfrac{B_i \rightarrow Env_j}{\{Env_i\}a(x)\{Env_j\}}} \quad \dfrac{\vdots}{\{Env_j\}\langle[(P, env_j^P), k', ch'], T_i\rangle\{Env_k\}}}{\{Env_i\}\langle[(a(x)\colon P, env_i^P), k, ch], (\text{if } B_i\ T_i\ F_i)\rangle\{Env_k\}} \quad env_j \models B_i}{\{Env_i\}\langle[(a(x)\colon P, env_i^P)\|(\overline{a}(y)\colon Q, env_i^Q), k, ch], (\text{if } B_i\ T_i\ F_i)\rangle\{Env_k\}} \quad \vdots$$

In addition, we retrieve the following inferences on $\overline{a}(x)\colon Q$.

$$\dfrac{\vdots \quad \dfrac{\dfrac{\dfrac{\dfrac{\{\mathrm{True}\}[a(x)\|\overline{a}(y)]\{x = y\}}{\{Env_i\}\overline{a}(y)\{env_j \models B_i\}}}{\dfrac{B_i \rightarrow Env_j}{\{Env_i\}\overline{a}(y)\{Env_j\}}} \quad \dfrac{\vdots}{\{Env_j\}\langle[(Q, env_i^Q), k', ch'], T_i\rangle\{Env_k\}}}{\{Env_j\}\langle[(\overline{a}(y)\colon Q, env_i^Q, k, ch]\rangle\{Env_k\}} \quad env_j \models B_i}{\{Env_i\}\langle[(a(x)\colon P, env_i^P)\|(\overline{a}(y)\colon Q, env_i^Q), k, ch], (\text{if } B_i\ T_i\ F_i)\rangle\{Env_k\}}$$

We can ignore the inferences of P and Q because we now discuss the one step of matching actions.

We must show that every history of a model is accepted by automaton m. There are two ways that m fails to accept history σ of model M:

(i) a model attempts an undefined transition when reading σ, or

(ii) a model never reaches to an accepting state after some finite prefix of σ.

In order to prove that every history of M, it suffices to show that (i) and (ii) are impossible. The above proof tree shows it in addition to the correspondence invariant. If $env_j \models B_i$ then env_j at which B_i holds exists. By the definition of Env_j, Env_j clearly holds at env_j. Therefore, if $env_j \models B_i$ then $env_j \models Env_j$. To avoid an undefined transition, an automaton for a property must transform from q_i to q_j. The one of conditions must holds at model state env_j that the execution of a pair of matching actions produces from env_i. Therefore, if $env_j \models Env_j$ then $env_j \models B_i$. Thus, any state of a model relates to a state of an automaton that stay in a range while reading a history of a model.

We derive conditions to ensure that m cannot reject a history by (ii). To do this, we define a relation $\ll$ on a set of automaton states. $q_{i-1} \ll q_n$ if for each i from 1 to n, there is n such that $\delta(q_{i-1}, s) = q_i$. Clearly, $\ll$ is a well-founded relation on a set of automaton states. Since $\ll$ is well-founded, the following function of which the range is an initial segment of ordinals exists:

$$\rho(q_j) = \sup\{\rho(q_i) + 1\colon q_i \ll q_j\}.$$

Assume that a subset Q' of automaton states contains no accepting states. Since subset Q' is well-ordered by $\ll$, the least element q_ξ of Q' exists. If m rejects a history by (ii) then there exists a path from q_η to q_ξ. We construct a variant function $v(q_\eta, s)$ which is zero when $\rho(q_\xi) = \rho(q_\eta) + 1$.

A property P is liveness if and only if;

$$\forall\alpha \in S^*\colon \exists\beta \in S^\omega\colon \alpha\beta \models \varphi.$$

Get the negation of it,

$$\exists\alpha \in S^*\colon \forall\beta \in S^\omega\colon \alpha\beta \not\models \varphi.$$

Then, An automaton specifies a liveness property has infinite sequences of automaton states, reading β. In this case, a sequence of automaton states by reading β contains no accepting states. Let Q' be a set of automaton states in the sequence. Q' has the $\ll$-minimal element which is then the least element. If an automaton accepts a history then the automaton does not enter into Q'. Thus, let q_ξ be the least element, $\delta(q_\eta, s_\xi) \neq q_\xi$

at state q_η such that $\rho(q_\xi) = \rho(q_\eta) + 1$. Given $v(q_\eta, s)$ with $\xi - \eta - 1$, $v(q_\eta, s) = 0 \Rightarrow \neg C_\xi$. In NHK$^\sharp$, $v(q_\eta, env_\eta) = 0 \Rightarrow \neg Env_i$.

Liveness properties require a subset containing no accepting states since an automaton specifying a liveness property moves to some state by all reading input. To ensure that α such that for any infinite sequence β the automaton rejects $\alpha\beta$, does not exist. we show that the variant function is decreased by every matching action pair, and introduce the axiom for communication;

$$\frac{}{\{\mathrm{True} \wedge v(q_i, s_i)\}[a(x) \| \overline{a}(y)]\{x = y \wedge v(\delta(q_i, s_i), s_j) = v(q_i, s_i) - 1\}}\ \text{Communication}^*$$

Let s_i be env_i, C_i Env_i, T_{ij} $env_j \models B_i$, and q_i (if B_i T_i F_i) in NHK$^\sharp$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\{\mathrm{True} \wedge v(q_i, env_i)\} \\ [a(x) \| \overline{a}(y)] \\ \{x = y \wedge v(\delta(q_i, env_i), env_j) = v(q_i, env_i) - 1\}\end{array}} \\ \begin{array}{ll}\{Env_i \wedge 0 < v(q_i, env_i)\} & \\ a(x) & v(T_i, env_j) = v(q_i, env_i) - 1 \\ \{env_j \models B_i & \rightarrow v(T_i, env_j) < v(q_i, env_i) \\ \wedge v(T_i, env_j) = v(q_i, env_i) - 1\} & \end{array}} \quad \dfrac{\vdots}{\{Env_j\}}}{\begin{array}{l}\{Env_i \wedge 0 < v(q_i, env_i)\} \\ a(x) \\ \{(env_j \models B_i \wedge Env_j) \rightarrow (v(T_i, env_j) < v(q_i, env_i))\}\end{array} \quad \begin{array}{l}\{Env_j\} \\ \langle [(P, env_j^P), k', ch'], T_i \rangle \\ \{Env_k\}\end{array}}}{\{Env_i \wedge 0 < v(q_i, env_i)\}\langle [(a(x) \colon P, env_i^P), k, ch], q_i \rangle \{Env_k\}} \quad \vdots}{\{Env_i \wedge 0 < v(q_i, env_i)\}\langle [(a(x) \colon P, env_i^P) \| (\overline{a}(y) \colon Q, env_i^Q), k, ch], q_i \rangle \{Env_k\}}$$

In addition, we retrieve the following inferences on $\overline{a}(x) \colon Q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\{\mathrm{True} \wedge v(q_i, env_i)\} \\ [a(x) \| \overline{a}(y)] \\ \{x = y \wedge v(\delta(q_i, env_i), env_j) = v(q_i, env_i) - 1\}\end{array}} \\ \begin{array}{ll}\{Env_i \wedge 0 < v(q_i, env_i)\} & \\ \overline{a}(x) & v(T_i, env_j) = v(q_i, env_i) - 1 \\ \{env_j \models B_i & \rightarrow v(T_i, env_j) < v(q_i, env_i) \\ \wedge v(T_i, env_j) = v(q_i, env_i) - 1\} & \end{array}} \quad \dfrac{\vdots}{\{Env_j\}}}{\begin{array}{l}\{Env_i \wedge 0 < v(q_i, env_i)\} \\ \overline{a}(x) \\ \{(env_j \models B_i \wedge Env_j) \rightarrow (v(T_i, env_j) < v(q_i, env_i))\}\end{array} \quad \begin{array}{l}\{Env_j\} \\ \langle [(Q, env_j^Q), k', ch'], T_i \rangle \\ \{Env_k\}\end{array}}}{\{Env_i \wedge 0 < v(q_i, env_i)\}\langle [(a(x) \colon Q, env_i^Q), k, ch], q_i \rangle \{Env_k\}} \quad \vdots}{\{Env_i \wedge 0 < v(q_i, env_i)\}\langle [(a(x) \colon P, env_i^P) \| (\overline{a}(y) \colon Q, env_i^Q), k, ch], q_i \rangle \{Env_k\}}$$

The variant function follows obligations O3–O4 in [AS89], and the existence of C_i and T_{ij} means the existence of an invariant I which follows obligations O1–O2 in [AS89]. Thus, the method of NHK$^\sharp$ is sound and complete by [AS89].

8 Exploring State Space

The algorithm in NHK$^\sharp$ consists of two depth-first searchs [CVWY91]. The two searchs are performed by evaluation on the modeling language. The second search is performed after the other. The first depth-first search is called `firstDFS()`, and the second depth-first search `secondDFS()`.

Listing 1: FirstDFS

```
1 firstDFS(m,f,S1,S2){
2    if [m,f] is not a member of S1
3       push([m,f],S1)
4       if m' s.t. m->m' exists
5          firstDFS(m',f',S1,S2)
6       else
7          return(S1)
8       if [m,f] is an accepting state
9          secondDFS(m,f,S,S2)
10   else
11      return(S1)
12 }
```

Listing 2: SecondDFS

```
1 secondDFS(m,f,S1,S2){
2    if [m,f] is not a member of S2
3       push([m,f],S2)
4       if [m,f] is a member of S1
5          terminate(True)
6       else
7          if m' s.t. m->m' exists
8             secondDFS(m',f',S1,push([m,f],S2))
9    else
10      return(S2)
11 }
```

`m'` and `f'` are produced following the semantics in Section 4.6 and Section 6.

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.

[AS89] Bowen Alpern and Fred B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Program. Lang. Syst.*, 11(1):147–167, January 1989.

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification: 2nd International Conference, CAV '90 New Brunswick, NJ, USA, June 18–21, 1990 Proceedings*, pages 233–242, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[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.

$$\text{Output}(1)\ \frac{(a \leftarrow \vec{v}) \notin ch}{[(\sim a(\vec{v})\colon P), env, \Box, ch] \overset{\sim a}{\rightarrow} [P, env, \Box, ch[a \leftarrow \vec{v}]]}$$

$$\text{Output}(2)\ \frac{(a \leftarrow \vec{v}) \notin ch}{[(\sim a(\vec{v})\colon P), env, k[Q \| \Box], ch] \overset{\sim a}{\rightarrow} [Q, env, k[\Box \| (P, env)], ch[a \leftarrow \vec{v}]]}$$

$$\text{Output}(3)\ \frac{(a \leftarrow \vec{v}) \notin ch}{[(\sim a(\vec{v})\colon P), env, k[\Box \| Q], ch] \overset{\sim a}{\rightarrow} [Q, env, k[(P, env) \| \Box], ch[a \leftarrow \vec{v}]]}$$

$$\text{Input}\ \frac{}{[(a(\vec{x})\colon P), env, k, ch[a \leftarrow \vec{v}]] \overset{a}{\rightarrow} [k[(P, env[\vec{x} \leftarrow \vec{v}])], env, \Box, ch]}$$

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

$$\text{Sum}(2)\ \frac{[Q, env, k, ch] \overset{\alpha}{\rightarrow} [Q', env', k', ch'] \quad P = 0}{[P + + Q, env, k, ch] \overset{\alpha}{\rightarrow} [Q', env', k', ch']}$$

$$\text{Com}(1)\ \frac{[P, env, k, ch] \overset{\alpha}{\rightarrow} [P', env', k[\Box \| (Q, env)], ch']}{[P \| Q, env, k, ch] \overset{\alpha}{\rightarrow} [P', env', k[\Box \| (Q, env)], ch']}$$

$$\text{Com}(2)\ \frac{[Q, env, k, ch] \overset{\alpha}{\rightarrow} [Q', env', k[(P, env) \| \Box], ch'] \quad P = 0}{[P \| Q, env, k, ch] \overset{\alpha}{\rightarrow} [Q', env', k[(P, env) \| \Box], ch']}$$

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

$$\text{Closure}\ \frac{[P, env', k, ch] \overset{\alpha}{\rightarrow} [P', env', k, ch]}{[(P, env'), env, k, ch] \overset{\alpha}{\rightarrow} [(P', env'), env, k, ch]}$$

$$\text{Constant}\ \frac{\begin{array}{c}[A(\vec{v}), env[A \leftarrow P(\vec{x})], k, ch] \rightarrow [(P, env[A \leftarrow P(\vec{x})][\vec{x} \leftarrow \vec{v}]), env[A \leftarrow P(\vec{x})], k, ch] \\ [(P(\vec{x}), env[A \leftarrow P][\vec{x} \leftarrow \vec{v}]), env[A \leftarrow P], k, ch] \overset{\alpha}{\rightarrow} [(P', env[A \leftarrow P(\vec{x})][\vec{x} \leftarrow \vec{v}]), env[A \leftarrow P], k, ch]\end{array}}{[A(\vec{v}), env[A \leftarrow P(\vec{x})], k, ch] \overset{\alpha}{\rightarrow} [(P', env[A \leftarrow P(\vec{x})][\vec{x} \leftarrow \vec{v}]), env[A \leftarrow P(\vec{x})], k, ch]}$$

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

$$\text{Res}\ \frac{\alpha, \overline{\alpha} \notin \{\alpha, \cdots\} \quad [(P, env), k, ch] \overset{\alpha}{\rightarrow} [(P', env'), k', ch']}{[(P[\alpha, \cdots], env), k, ch] \overset{\alpha}{\rightarrow} [(P'[\alpha, \cdots], env'), k', ch']}$$

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

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

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

Figure 1: Semantics of RCCS