MCTK 1.0 User Manual

Kaile Su[1], Xiangyu Luo[2,3,*], Abdul Sattar[4]

[1] School of Electronics Engineering and Computer Science, Peking University, Beijing, China
sukl@pku.edu.cn

[2] Key Laboratory of Security for Information System of Ministry of Education
School of Software, Tsinghua University, Beijing, China

[3] Department of Computer Science, Guilin University of Electronic Technology, Guilin, China
shiangyuluo@gmail.com

[4] Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
a.sattar@gu.edu.au

April 10, 2009

1 Introduction

MCTK is a symbolic model checker for temporal logic of knowledge, developed by Kaile Su and Xiangyu Luo, at Department of Computer Science at Sun Yat-sen University. MCTK is developed based on NuSMV [3] by CUDD under the Open Source license that allows free academic and non-commercial usage. It functionally extends the NuSMV 2.1.2 by dealing with CTL^* and more worthy to point out, the CKL_n formulas raised by Halpern and Vardi, which integrates knowledge and time with arbitrary order. In other words, the tool can check the more complicated $ECKL_n$ that is an extension of CKL_n by adding the path quantifier A and E. The main algorithm [6, 5] is given by Kaile Su and implemented by Xiangyu Luo. The current version 1.0 of MCTK interprets epistemic modalities under observable semantics.

The traditional model checking is mostly focused on the specification of temporal formulas. But as the epistemic logic is getting more and more concern in Multi-Agent system research, combining of the two modalities may give us more expressive power to formalize more complex situation. So, this work may mean a further step to putting the pure theory of Multi-Agent into implementation for the whole community.

1.1 What's the difference between MCTK and NuSMV 2.1.2?

1. In the package of MCTK, we add subdirectory "eckl" to "∼/nusmv/src/", modified some original source code of NuSMV 2.1.2 in "∼/nusmv/src/", and mod-

[*] The developer of MCTK. Please email bugs and suggestions on MCTK at shiangyuluo@gmail.com.

ified "∼/nusmv/Makefile.in" by adding "eckl" to the end of the assignment of key "ALL_PKGS".

2. In the input language of MCTK, we extend the CTL specification (with keyword "SPEC") with knowledge modality K and common knowledge modality C, so that we can check $CTLK$ properties in MCTK. A new kind of specification with keyword "ECKLNSPEC" is added to allow one to specify $ECKL_n$ properties. We support the agent definition by allowing one to specify some input parameters of the agent's module observable. All local variables of an agent's module are observable to the agent.

3. An example "mwoven.smv" for testing CTL^* properties is available in the directory "∼/nusmv/examples/ctlstar/"; Some examples for testing epistemic properties are available in the directory "∼/nusmv/examples/knowledge/".

1.2 Installation and Invocation of MCTK

The installation of MCTK is the same as that of NuSMV 2.1.2. Please see the file README in the distribution package of MCTK. We give a quick way to build MCTK in command line as follows:

1. decompress the MCTK package and move to the decompressed directory "MCTK-#.#.#"
2. move to the directory "MCTK-#.#.#/cudd-2.3.0.1"
3. % make
4. move to the directory "MCTK-#.#.#/nusmv"
5. % ./configure
6. % make

After compilation, the executable file NuSMV is built in the directory "MCTK-#.#.#". In the command line you can execute NuSMV followed with the smv file you want to verify. You can also get the following usage of MCTK by command "% ./NuSMV -help".

The usage of MCTK

```
Usage:  ./NuSMV [-s] [-ctt] [-lp] [-n idx] [-v vl] [-cpp] [-is] [-ic] [-ils]\
        [-ii] [-r] [-f] [-int] [-h | -help] [-i iv_file] [-o ov_file]\
        [-AG] [-load script_file] [-reorder] [-dynamic] [-m method]\
        [[-mono]|[-thresh cp_t]|[-cp cp_t]|[-iwls95 cp_t]] [-coi]\
        [-noaffinity] [-iwls95preorder] [-bmc] [-bmc_length k]\
        [-ofm fm_file] [-obm bm_file] [input-file] [-ctl*_alg vl]
Where:
         -s              does not read any initialization file
                         (/usr/local/share/nusmv/master.nusmvrc, ~/.nusmvrc)
                         default in batch mode.
         -ctt            enables checking for the totality of the transition
                         relation
         -lp             lists all properties in SMV model
         -n idx          specifies which property of SMV model should be checked
         -v vl           sets verbose level to "vl"
         -cpp            runs preprocessor on SMV files
         -is             does not check SPEC
         -ic             does not check COMPUTE
         -ils            does not check LTLSPEC
         -ii             does not check INVARSPEC
         -r              enables printing of reachable states
         -f              computes the reachable states (forward search)
         -int            enables interactive mode
         -h | -help      prints out current message
         -i iv_file      reads order of variables from file "iv_file"
         -o ov_file      prints order of variables to file "ov_file"
         -AG             enables AG only search
         -load sc_file   loads NuSMV commands from file "sc_file"
         -reorder        enables reordering of variables before exiting
         -dynamic        enables dynamic reordering of variables
         -m method       sets the variable ordering method to "method".
                         Reordering will be activated
         -mono           enables monolithic transition relation
         -thresh cp_t    conjunctive partitioning with threshold of each
                         partition set to "cp_t" (DEFAULT, with cp_t=1000)
         -cp cp_t        DEPRECATED: use -thresh instead.
         -iwls95 cp_t    enables Iwls95 conjunctive partitioning and sets
                         the threshold of each partition to "cp_t"
         -coi            enables cone of influence reduction
         -noaffinity     disables affinity clustering
         -iwls95preorder enables iwls95 preordering
         -bmc            enables BMC instead of BDD model checking
         -bmc_length k   sets bmc_length variable, used by BMC
         -ofm fm_file    prints flattened model to file "fn_file"
         -obm bm_file    prints boolean model to file "bn_file"
         input-file      the file both the model and
                         the spec were read from
         -ctl*_alg n     sets the type of CTL* model checking algorithm to "n"
            -ctl*_alg 1  uses the default CTL* model checking algorithm
            -ctl*_alg 2  checks the CTL subformula of CTL* formula by using
                         CTL model checking algorithm
         -bddstats       sets to print BDD status after checking an ECKLn
                         specification
```

2 Software architecture

A multi-agent system contains an *environment* and a finite number of *agents*. The basic form of an agent is "agent $A : M$", where A is the *name* of the agent and M is the agent's *program module* ("module" for short). Each agent in a multi-agent system is assumed to have a unique name. The program module M is the main part of an agent, which determines its observable variables and its behavior.

Each agent is able to perform its actions according to its own local state, which is encoded by its observable variables. The change of the *state* of the system occurs as a result of *joint actions* performed by the agents and the environment. In the environment and each agent's module, we thus include the descriptions of how the actions change the state of the system. Note that agent's actions do not need to be represented explicitly for each system.

The architecture of MCTK is an extension of NuSMV 2.1.2. Before model checking an $ECKL_n$ specification, the following modules should be invoked in turn to obtain the symbolic representation of a finite-state program with n agents:

Parser builds a parse tree representing the internal format of a MCTK input file using Lex and Yacc.

Instantiation processes the parse tree, and performs the instantiation of the declared modules, building a description of the finite state machine (FSM) representing the model (e.g., the transition relation, the initial states and the fairness).

Encoder performs the encoding of data types and finite ranges into boolean domains. The state variables $\mathbf{x}$ of the system and the observable variables O_i of each agent $i (1 \leq i \leq n)$ can be retrieved in this phase.

FSM Compiler provides the routines for constructing and manipulating FSMs at the BDD level. We can get the BDDs that represents the initial condition θ and the transition relation τ of the system in this phase.

2.1 Input language

The MCTK input language is extended from the NuSMV input language and its syntax is described as follows. We refer to NuSMV v2.1 user manual [1] for more details of its input language.

```
MCTK_program ::= EnvDef AgentDefList
EnvDef ::= "MODULE" "main" "(" ")"
  EVarDef        ;; environmental variables
  AgentList
  [EVarInitDef]
  [EVarTransDef]
  [CTLKSpecDef]  ;; CTLK specification
  [ECKLnSpecDef] ;; ECKLn specification
  ...
AgentList ::= atom ":" AgentType ";" |
  AgentList atom ":" AgentType ";"
AgentType ::= atom [ "(" AParaList ")" ] |
  "array" number ".." number "of" AgentType
AParaList ::= simple_f |
  AParaList "," simple_f
AgentDefList ::= AgentDef |
  AgentDefList AgentDef
AgentDef ::=
  "MODULE" atom [ "(" FParaList ")" ]
    [LVarDef]      ;; local variables
    [ActDef]       ;; action variables
    [ActInitDef]
    [ActTransDef]
    [OVarInitDef]  ;; INIT of observable variables
    [OVarTransDef] ;; TRANS of observable variables
    ...
FParaList ::= FPara | FParaList "," FPara
ActDef ::= atom ": {" AtomList "};"
AtomList ::= atom | AtomList "," atom
atom ::= [A-Za-z_][A-Za-z0-9_\$#-]*
```

```
FPara ::= atom |                 ;; unobservable variable
  ["ObsPrm_"][A-Za-z0-9_\$#-]* ;; observable variable
```

2.1.1 Environment declaration

We use the `main` module to define the environment. `EVarDef` declares some state variables of the environment, which may be used by some agents as shared variables for the purpose of inter-agent communication. `AgentList` defines a number of agents. Note that, for conveniently describing a set of agents with the same module definition, we allow `AgentType` to be an array of `AgentType` itself. The set of the initial states and the next states relate current or next states, encoded by the environmental variables in `EVarDef`, are defined in `EVarInitDef` and `EVarTransDef` respectively. `CTLKSpecDef` and `ECKLnSpecDef` gives some specifications in $CTLK$ and $ECKL_n$ logics to be checked. Their syntaxes are listed as follows:

```
CTLKSpecDef :: "SPEC" f [";"]
f ::
  simple_f                  ;; a simple boolean expression
  | "(" f ")"
  | "EG" f                  ;; exists globally
  | "EX" f                  ;; exists next state
  | "EF" f                  ;; exists finally
  | "AG" f                  ;; forall globally
  | "AX" f                  ;; forall next state
  | "AF" f                  ;; forall finally
  | "E" "[" f "U" f "]"     ;; exists until
  | "A" "[" f "U" f "]"     ;; forall until
  | name "K" f              ;; agent name knows f
  | "(" NameList ")" "C" f  ;; agents NameList commonly know f
  ...

ECKLnSpecDef ::= "ECKLNSPEC" g [";"]
g ::
  simple_f
  | "(" g ")"
  | "A" g                      ;; for all paths
  | "E" g                      ;; for some path(s)
  | "X" g                      ;; next state
  | "G" g                      ;; globally
  | "F" g                      ;; finally
  | g "U" g                    ;; until
  | name "K" g                 ;; agent name knows g
  | "(" NameList ")" "C" g  ;; agents NameList commonly know g
  ...

name :: var_id                ;; the name of an agent
NameList :: name | NameList "," name
```

Simple expressions simple_f are constructed from variables, constants, and a collection of operators, including boolean connectives, integer arithmetic operators, case expressions and set expressions, and logic operators ! (not), & (and), | (or), xor (exclusive or), -> (implication) and <-> (equivalence). We have $\mathbf{F}f \equiv true\mathbf{U}f$ and $\mathbf{G}f \equiv \neg\mathbf{F}\neg f$. "(" NameList ")" "C" f express that formula f is the common knowledge among the agents in NameList. Note that each agent's "Name" must be an *instance* of an agent's module.

2.1.2 Agent declaration

As is shown in the syntax of AgentDef, each agent is defined as a MODULE. We explain some important parts of an agent's module below:

Formal parameters: The formal parameters of an agent's module are defined in FParaList. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated. The difference between the syntax of the formal parameters of an agent's module and that of a NuSMV module is that the former allows some formal parameters to be specified *observable*. In the current implementation, all of the formal parameters with prefix "ObsPrm_" are observable. We only allow the type of the observable actual parameter, simple_f declared in AParaList, to be *variable*. Thus, an agent is able to observe or modify these observable actual parameters, i.e. some variables of environment and even some local variables of other agents.

Observable variables: The set O_i of observable variables of agent i is defined to be the set of agent i's local variables and observable actual parameters. In our framework, an agent's local state is identified by its observable variables. If a local variable defined in LVarDef is a *module instance*, then all of the local variables except those instances of other modules in the module instance are recursively inserted into the set of local variables of the agent. So the variable defined in ActDef is also in the set of the agent's observable variables.

Protocol: The protocol for agent i is a description of what actions agent i may take. It can be formally defined as a function from the set of its local states to nonempty sets of its actions. If at a given step of the protocol there is more than one action that may be chosen, then only one of them can be actually performed. The choice of the action to perform is nondeterministic. All of the allowable actions can be encoded by a variable of an enumerated type, which is defined in ActDef. The evolution of the variable that represents action is defined in ActInitDef and ActTransDef, which define an agent's initial actions and the next actions relate current or next local states respectively. The variables appearing in ActInitDef and ActTransDef are restricted observable. Note that the protocol is not necessary because we are mainly concerned about the evolution of an agent's observable variables.

Evolution of observable variables: Agent i's evolution function for observable variables is the description of how the values of agent i's observable variables change.

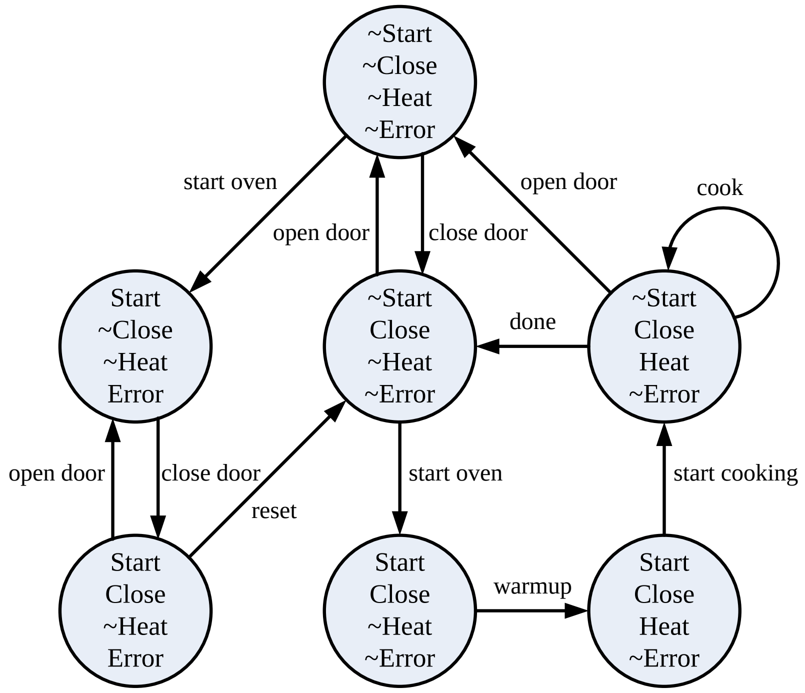

Figure 1: A microwave oven

It is a function from the set of its local states to the set of its local states. The evolution function is defined in `OVarInitDef` and `OVarTransDef`.

3 Case Studies

3.1 A Microwave Oven

In this subsection we first illustrate the model checking algorithm in MCTK for CTL^* on a small example that describes the behavior of a microwave oven in [4]. Figure 1 gives the Kripke structure for the oven. We can model the Kripke structure in the following MCTK input file.

A microwave oven

```
MODULE main
VAR
  start : boolean;
  close : boolean;
  heat : boolean;
  error : boolean;

INIT
  start=0 & close=0 & heat=0 & error=0

TRANS
```

```
  --s0->s1
  ( (!start & !close & !heat & !error) &
    (next(start) & !next(close) & !next(heat) & next(error)) )
  |        --s0->s2
  ( (!start & !close & !heat & !error) &
    (!next(start) & next(close) & !next(heat) & !next(error)) )
  |        --s1->s4
  ( (start & !close & !heat & error) &
    (next(start) & next(close) & !next(heat) & next(error)) )
  |        --s2->s0
  ( (!start & close & !heat & !error) &
    (!next(start) & !next(close) & !next(heat) & !next(error)) )
  |        --s2->s5
  ( (!start & close & !heat & !error) &
    (next(start) & next(close) & !next(heat) & !next(error)) )
  |        --s3->s0
  ( (!start & close & heat & !error) &
    (!next(start) & !next(close) & !next(heat) & !next(error)) )
  |        --s3->s2
  ( (!start & close & heat & !error) &
    (!next(start) & next(close) & !next(heat) & !next(error)) )
  |        --s3->s3
  ( (!start & close & heat & !error) &
    (!next(start) & next(close) & next(heat) & !next(error)) )
  |        --s4->s1
  ( (start & close & !heat & error) &
    (next(start) & !next(close) & !next(heat) & next(error)) )
  |        --s4->s2
  ( (start & close & !heat & error) &
    (!next(start) & next(close) & !next(heat) & !next(error)) )
  |        --s5->s6
  ( (start & close & !heat & !error) &
    (next(start) & next(close) & next(heat) & !next(error)) )
  |        --s6->s3
  ( (start & close & heat & !error) &
    (!next(start) & next(close) & next(heat) & !next(error)) )

JUSTICE start & close & !error
```

We check the following three specifications with MCTK, the first one is in CTL, the second one is in $ECKL_n$ but is semantically equivalent to the first one. The third one is in CTL^*, which cannot be expressed in CTL or LTL.

```
SPEC        AG(start -> AF heat)
ECKLNSPEC   A G(start -> A F heat)
ECKLNSPEC   !E F(!close & start & E((F heat) & (G error)))
```

If the fairness constraint "`JUSTICE start & close & !error`" is specified, then the first two specifications both are true, otherwise both are false. The third CTL^* specification is true no matter whether the fairness constraint is specified or not.

3.2 The Dining Cryptographers Protocol

In this subsection, we apply our model checker MCTK to the verification of the Dining Cryptographers Protocol [2], a protocol for anonymous broadcast. Chaum introduces this protocol by the following story:

> Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made

with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been the NSA (US National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if the NSA is paying.

Chaum shows that the following protocol can be used to solve the dining cryptographers' quandary. The protocol assumes that at most one cryptographer is paying.

1. Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer to his right, so that only the two of them can see the outcome.

2. Each cryptographer then states aloud whether the two coins that he can see – the one he flipped and the one his left-hand neighbor flipped – fell on the same side or different sides.

2e. As an exception to the previous step, if one of the cryptographers is the payer, he states the opposite of what he sees.

It can be shown that after running this protocol, all the cryptographers are able to determine whether it was the NSA or one of the cryptographers who paid for dinner. Specifically, an even number of differences uttered at the table indicates that NSA is paying, an odd number of differences indicates that a cryptographer is paying. Significantly, if a cryptographer is paying neither of the other two learns anything from the utterances about which cryptographer it is.

3.2.1 Model description in MCTK

To formalize this protocol with three cryptographers in MCTK input language, we should first define the environment of this protocol as follows:

Environment Declaration

```
MODULE main()
VAR
  coin1 : boolean;  --cryptographer C1
  coin2 : boolean;  --cryptographer C2
  coin3 : boolean;  --cryptographer C3

  C1 : DC(coin1, coin2, C2.said, C3.said);
  C2 : DC(coin2, coin3, C1.said, C3.said);
  C3 : DC(coin3, coin1, C1.said, C2.said);

INIT  (C1.paid + C2.paid + C3.paid) <= 1;
TRANS next(coin1)=coin1 &
      next(coin2)=coin2 &
      next(coin3)=coin3
```

In the environment declaration, we define three boolean variables, coin1, coin2 and coin3, to respectively indicate the coins the cryptographers C1, C2 and C3 flipped. For example, if the value of variable coin1 is *true*, then it means that the coin the cryptographer C1 flipped is on its obverse side, otherwise it is on its reverse side. Next expressions under the "TRANS" keyword relate current and next state variables to express transition relation of the model. For example, expression next(coin1)=coin1 in line 12 expresses that the next value of variable coin1 is equal to its current value. So line 12-14 has kept variables coin1, coin2 and coin3 unchangeable since their initial values are assigned non-deterministically.

Then, we define the module of the dining cryptographers as the following source code.

Agent's Module

```
1  MODULE DC(
2    Observable coin_self,
3    Observable coin_left,
4    Observable said1,
5    Observable said2)
6
7  VAR
8    paid : boolean;--true=pay for the dinner
9    said : boolean;--true=say "different side"
10
11 ASSIGN
12   init(said) := paid xor
13                 (coin_self xor coin_left);
14   next(said) := said;
15   next(paid) := paid;
```

In the declaration of agent's module, "DC" is the name of the module used by the cryptographers. The variables defined under the "VAR" keyword are viewed as the local ones in the agent's module, where boolean variable paid represents whether the cryptographer pays for the dinner. The set of initial states of the model is determined by a boolean expression under the "INIT" keyword, so line 11 of the environment declaration initially restricts that at most one cryptographer pays for the dinner. Another local boolean variable said represents the cryptographer's utterance.

Line 12-13 of agent's module simulates Step 2 and 2e of this protocol. Note that we assign the value of variable said in the initial state of the model, so that we can determine the agents' knowledge not considering the temporal dimension. Line 14-15 of agent's module has kept variables said and paid unchangeable since their initial values are determined.

Besides the local variables paid and said, an agent's observable variables also include his own coin and the coin his left-hand neighbor flipped and the other two agents' utterances. So, we should define four observable formal parameters, coin_self, coin_left, said1 and said2, respectively for the four observable variables in line 2-5 of agent's module. In the environment declaration, for example, Line 7 define a cryptographer C1 with the actual parameters coin1, coin2, C2.said and

C3.said. So the observable variables of C1 are C1.paid, coin1, coin2, C1.said, C2.said and C3.said.

3.2.2 $ECKL_n$ specification

Now let us consider the specification of the problem. We see that the following $ECKL_n$ formula expresses Chaum's requirements in the case of cryptographer C1:

```
ECKLNSPEC
G(
  !C1.paid -> (
     (C1 K (!C1.paid & !C2.paid & !C3.paid)) |
     ( (C1 K (C2.paid | C3.paid)) &
       !(C1 K C2.paid) & !(C1 K C3.paid) ) )
)
```

That is, if cryptographer C1 does not pay, then he knows either that no cryptographer pays, or knows that one of the other two pays, but does not know which. The specifications for the other cryptographers are similar. To verify the protocol, it suffices to consider only the specification for cryptographer C1 above, because of symmetry considerations.

3.3 Russian Cards

From a pack of seven known cards two players each draw three cards and a third player gets the remaining card. How can the players with three cards openly inform each other about their cards, without the third player learning from any of their cards who holds it?

The "Russian Cards" problem [7] was originally presented at the Moscow Math Olympiad 2000. The problem is to find solutions that allow the sender and receiver to learn each other's hand of cards, without revealing this information to the eavesdropper.

Call the players Anne, Bill and Cath, and the cards $0, \ldots, 6$, and suppose Anne holds $\{0, 1, 2\}$, Bill $\{3, 4, 5\}$, and Cath card 6. For the cards $\{0, 1, 2\}$, write 012 instead, for the card deal, write 012.345.6, etc. Assume from now on that 012.345.6 is the actual card deal. Anne and Bill exchange each other's information by some announcements. All announcements must be public and truthful. Obviously, the first requirement for a solution to the problem is that Cath's ignorance of Anne and Bill's hand of cards is always commonly known to the three players after any announcement. Such announcements are called safe.

A solution to the Russian Cards problem is a sequence of safe announcements. The second requirement for a solution is that after these safe announcements, it is commonly known to Anne and Bill (not necessarily including Cath) that Anne knows Bill's hand and Bill knows Anne's hand. The following five hands protocol is a solution to the problem:

Anne says ”My hand of cards is one of 012, 034, 056, 135, 246.” after which Bill says “Cath has card 6.” (in other words, Bill announces that his hand of cards is one of 345, 125, 024).

3.3.1 Russian Cards in MCTK

In this section, we specify the five hands protocol in MCTK and verify the two requirements above, which are expressed in $ECKL_n$ logic. We first define the environment of this protocol as follows.

```
Environment Declaration
MODULE main
VAR
  stage:0..3;

  a0:0..1;  a1:0..1;  ...  a6:0..1;
  b0:0..1;  b1:0..1;  ...  b3:0..1;
  c0:0..1;  c1:0..1;  ...  c3:0..1;

  a_ann:0..1;
  b_ann:0..1;

  PA: Anne(a0,...,a6,a_ann,b_ann,stage);
  PB: Bill(b0,...,b6,a_ann,b_ann,stage);
  PC: Cath(c0,...,c6,a_ann,b_ann,stage);

TRANS
  (stage=0 ->
    (a0+a1+a2+a3+a4+a5+a6+
    b0+b1+b2+b3+b4+b5+b6+
    c0+c1+c2+c3+c4+c5+c6)=0)  &
  (stage=1 ->
    ((a0+a1+a2+a3+a4+a5+a6)=3&
    (b0+b1+b2+b3+b4+b5+b6)=3&
    (c0+c1+c2+c3+c4+c5+c6)=1&
    (a0+b0+c0)=1&(a1+b1+c1)=1&(a2+b2+c2)=1&
    (a3+b3+c3)=1&(a4+b4+c4)=1&(a5+b5+c5)=1&
    (a6+b6+c6)=1))  &
  (stage>=1 ->
    (next(a0)=a0 & ... & next(a6)=a6 &
    next(b0)=b0 & ... & next(b6)=b6 &
    next(c0)=c0 & ... & next(c6)=c6))

ASSIGN
  init(stage):=0;
  init(a_ann):=0;
  init(b_ann):=0;
```

```
37
38   next(stage):=case
39     stage<3: stage+1;
40     stage=3: stage;
41   esac;
42
43   next(a_ann):=case
44     next(PA.announce)=1: 1;
45                       1: a_ann;
46   esac;
47
48   next(b_ann):=case
49     next(PB.announce)=1&c6=1: 1;
50                            1: b_ann;
51   esac;
```

In the environment declaration, variable `stage` is the ‘clock tick’, it stands for the execution stage of this protocol. The initial value of `stage` is 0. `stage=3` indicates that the execution of the protocol finishes. Boolean variables `a0,...,a6`, `b0,...,b6` and `c0,...,c6` stand for Anne, Bill and Cath’s hand of cards, respectively. For example, `b3=1` means that Bill holds card 3, `b3=0` otherwise. Line 17-31 describe how the hand of cards of the three players change in each stage. Line 17-20 express that cards aren’t dealt to those players in stage 0. Line 21-27 restrict that Anne and Bill each can receive three cards and Cath can receive one in stage 1. But the values of `a0,...,a6`, `b0,...,b6` and `c0,...,c6` are uncertain, so by this restriction, we can model $\binom{7}{3}\binom{4}{3}\binom{1}{1} = 140$ deals. Line 28-31 keep all players’ hands of cards invariant after stage 1. Boolean variables `a_ann` denotes whether Anne announce some information or not. The value of `a_ann` is initially assigned 0 in Line 35. Line 43-46 assign 1 to `a_ann` after Anne’s announcement (`PA.announce=1`). Similarly, boolean variable `b_ann` stands for Bill’s announcement. Line 48-51 assign 1 to `b_ann` after Bill announces that Cath holds card 6 (`PB.announce=1` and `c6=1`).

Agent `PA`, for Anne, is declared by Line 12. The name of the agent is `PA`. It uses module “Anne”. It can observe the variables between parentheses. So, the set of Anne’s observable variables is {`a0,a1,a2,a3,a4,a5,a6,a_ann,b_ann,stage`} plus the local variables of module “Anne”. Variables `a_ann,b_ann,stage` are also the actual parameters of Agent `PB` and `PC`, as they are publicly observable.

Anne’s protocol is defined as the following module.

```
Anne's Module
MODULE Anne(Observable a0,...,Observable a6,
        Observable a_ann,Observable b_ann,
        Observable stage)
VAR
  announce:0..1; --announcement action

ASSIGN
  init(announce):=0;
```

```
9   next(announce):=case
10    stage=1 &
11    ((a0 & a1 & a2)|(a0 & a3 & a4)|
12     (a0 & a5 & a6)|(a1 & a3 & a5)|
13     (a2 & a4 & a6)) : 1;
14    1 : announce;
15  esac;
```

All of the formal parameters in Anne's module are observable. Boolean variable `announce` is used to denote whether Anne take the action of announcement or not. `announce=1` means that Anne has taken the action of announcement, `announce=0` otherwise. Line 9-14 means that Anne announces "My hand is one of 012, 034, 056, 135, and 246" in stage 1.

Bill and Cath's protocols are defined as the following modules. Line 9-12 means that if in stage 2, Anne has already announced that her hand is one of the five cases, then Bill will take the action of announcement. Note that there is not any code in the body of Cath's module, as Cath does not act.

Bill and Cath's Modules

```
1  MODULE Bill(Observable b0,...,Observable b6,
2          Observable a_ann,Observable b_ann,
3          Observable stage)
4  VAR
5    announce:0..1; --announcement action
6
7  ASSIGN
8    init(announce):=0;
9    next(announce):=case
10     stage=2 & a_ann=1 : 1;
11                     1 : announce;
12   esac;
13
14 MODULE Cath(Observable c0,...,Observable c6,
15         Observable a_ann,Observable b_ann,
16         Observable stage)
```

3.3.2 $ECKL_n$ specifications

Now let's formalize the two requirements for the five hands protocol in $ECKL_n$ logic.

We use `a_know_b`, `b_know_a` and `c_ignorant` to denote "Anne knows Bill's hand", "Bill knows Anne's hand" and "Cath does not know any of Anne's or Bill's hand", respectively. We say that an agent knows whether formula ψ holds if it either knows ψ or its negation. `c_ignorant` also means that "if Cath does not hold a card, Cath can imagine both Anne and Bill to have it", it follows that Cath does not know Anne to have it, and does not know Bill to have it. Therefore, we have the following $ECKL_n$ formulas

```
a_know_b := ((PA K b0)|(PA K !b0))
                     & ... &
            ((PA K b6)|(PA K !b6));
b_know_a := ((PB K a0)|(PB K !a0))
                     & ... &
            ((PB K a6)|(PB K !a6));
c_ignorant := (!(PC K a0) & !(PC K b0))
                         & ... &
              (!(PC K a6) & !(PC K b6)).
```

Thus, we can formalize the first and the second requirements as the following two $ECKL_n$ specifications:

```
ECKLNSPEC A G((PA,PB,PC) C c_ignorant)
ECKLNSPEC A G((a_ann & b_ann) ->
          ((PA,PB) C (a_know_b & b_know_a)))
```

4 Contact Us

The basic theory of MCTK is proposed by Kaile Su (sukl@pku.edu.cn) and the tool is implemented by Xiangyu Luo (shiangyuluo@gmail.com). Welcome to email bugs and suggestions on MCTK to help us improve it.

References

[1] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. *NuSMV 2.1 User Manual*. ITC-IRST: Istituto Trentino di Cultura / Istituto Ricerca Scientifica e Tecnologica, Trento, Italy, Available at http://nusmv.irst.itc.it/NuSMV/userman/v21/nusmv.pdf, 2002.

[2] David Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[3] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV-2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[4] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 2000.

[5] Kaile Su. Model checking temporal logics of knowledge in distributed systems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 98–103. AAAI Press / The MIT Press, 2004.

[6] Kaile Su, Abdul Sattar, and Xiangyu Luo. Model checking temporal logics of knowledge via obdds. *Comput. J.*, 50(4):403–420, 2007.

[7] Hans P. van Ditmarsch. The russian cards problem. *Studia Logica*, 75(1):31–62, 2003.