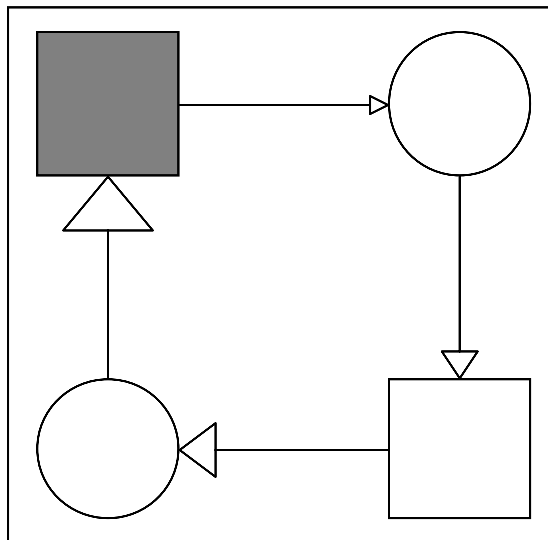
Design/CPN
ASK-CTL Manual

Version 0.9



University of Aarhus

Computer Science Department
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark
Tel: +45 89 42 31 88
Fax: +45 89 42 32 55

© 1996 University of Aarhus

© 1996 University of Aarhus

Computer Science Department
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark
Tel: +45 89 42 31 88
Fax: +45 89 42 32 55

e-mail: designCPN-support@daimi.aau.dk

Authors: Søren Christensen and Kjeld H. Mortensen.

Design/CPN is a trademark of Meta Software Corporation.

Macintosh is a registered trademark of Apple Computer, Inc.

Design/CPN ASK-CTL Manual

Version 0.9

Introduction

This library implements a CTL-like temporal logic called ASK-CTL. The logic is an extension of CTL [3] which is a branching time logic. ASK-CTL is interpreted over the state spaces (also called the occurrence graph or reachability graph/tree) of Coloured Petri Nets in the tool Design/CPN. We have extended CTL in order to take into account, both state information and transition information. Additionally the library also contains a model checker. It takes an ASK-CTL formula as argument, checks the formula against the current state space, and returns the truth value of the given formula. For efficiency reasons, the algorithm takes into account the Strongly Connected Component graph (SCC-graph).

The ASK-CTL logic and model checker is implemented in SML, and queries are formulated directly in SML syntax.

For more information about ASK-CTL, we refer to the paper presented at WoDES'96 [1] and the online technical report [2].

In the rest of this manual we assume that you know Design/CPN and the Occurrence Graph Tool, and have general knowledge about the temporal logic CTL and model checking.

How to Install the ASK-CTL Library

Assuming that you have downloaded the library archive, do the following:

UNIX

1) Uncompress the archive:

```
uncompress ASKCTL0_9.UNIX.tar.Z
```

2) Extract files from the archive:

```
tar xvf ASKCTL0_9.UNIX.tar
```

This will create the file called ASKCTLloader.sml and the directory called ASKCTL.

3) Copy the extracted files to the Design/CPN installation directory:

```
cp ASKCTLloader.sml <CPN_HOME>/OGMLfiles
cp -r ASKCTL <CPN_HOME>/OGMLfiles
```

If you do not have permissions to complete the commands above, then you need to modify the contents of ASKCTLloader.sml. You need to change the two lines:

```
use (ogpath^"ASKCTL/BitArray.sml");
use (ogpath^"ASKCTL/ASKCTL.sml");
```

such that you provide the alternative absolute path to the files:

```
use ("<abs. path>/ASKCTL/BitArray.sml");
use ("<abs. path>/ASKCTL/ASKCTL.sml");
```

Macintosh

If you have a file called

```
ASKCTL0_9.Mac.sea.hqx
```

then you need to decode the file using an application that can handle Binhex files. StuffIt and other archive applications are able to handle Binhex files. When you have decoded the Binhex file then you have an application called:

```
ASKCTL0_9.Mac.sea
```

This is an application which is a self extracting archive. Just double click on the icon to get the contents of the archive. The archive contains the following file and folder which need to be moved into the OGMLfiles folder of the Design/CPN installation folder:

```
ASKCTLloader.sml
ASKCTL
```

The installation is now completed.

How to Use the ASK-CTL Logic with Design/CPN

We assume that you have entered the simulator in Design/CPN, and the OG Tool, have generated the full state space, and the SCC-graph. First you need to load the library:

1) Create an auxiliary box (invoke Box from the Aux menu).

2) In the box type:

```
use (ogpath^"ASKCTLloader.sml");
```

If you have modified the path in ASKCTLloader.sml, then you need to give the absolute path to the file ASKCTLloader.sml instead:

```
use ("<abs. path>/ASKCTLloader.sml");
```

3) Invoke ML Evaluate from the Aux menu. The status bar will indicate when the library has been loaded successfully with the message "ASK-CTL library v0.9 loaded successfully".

If you re-enter the Occurrence Graph Tool, then you need to redo the steps above. Now you are ready to write ASK-CTL formulas and do model checking.

Basic Concepts

The ASK-CTL library has two parts: one which implements the language of the logic, and one which implements the model checker. In the sections following the contents of the two parts are explained. But first some notation and explanation of basic concepts.

Some of the ASK-CTL formulas are used to express properties about paths. A **path** is a sequence of states and transition occurrences – a walk-through of the state space constrained by the direction of arcs. A path may be infinite. From a state space it is possible to enumerate all paths.

An ASK-CTL formula is either interpreted either over the domain of states or transition occurrences in a path. One operator, the domain switch operator, MODAL, allows one to jump from one domain to the other. See the examples for more details of the semantics of this operator.

One of the more important operators in ASK-CTL is the until-operator, UNTIL. Assume we are currently in the domain of states, given a path, UNTIL(A_1,A_2) is true iff there exists a prefix of the path on which A_1 is true in every state and A_2 is true in the state immediately after. By quantifying universally and existentially over paths one gets the two ASK-CTL formulas FORALL_UNTIL and EXIST_UNTIL respectively.

Syntax and Semantics

Below find the syntax of ASK-CTL. As formulas are expressed over domains of either states or transition occurrences, the syntax actually has two categories called state and transition formulas respectively.

ASK-CTL formula syntax:

State formulas

```
A ::= TT
    | FF
    | NOT(A)
    | AND(A1,A2)
    | OR(A1,A2)
    | NF(<node expression>)
    | EXIST_UNTIL(A1,A2)
    | FORALL_UNTIL(A1,A2)
```

```
        | POS(A)
        | INV(A)
        | EV(A)
        | ALONG(A)
        | MODAL(B)
        | EXIST_MODAL(A1,B2)
        | FORALL_MODAL(A1,B2)
        | EXIST_NEXT(A)
        | FORALL_NEXT(A)
```

Transition formulas

```
B ::= TT
    | FF
    | NOT(B)
    | AND(B1,B2)
    | OR(B1,B2)
    | AF(<arc expression>)
    | EXIST_UNTIL(B1,B2)
    | FORALL_UNTIL(B1,B2)
    | POS(B)
    | INV(B)
    | EV(B)
    | ALONG(B)
    | MODAL(A)
    | EXIST_MODAL(B1,A2)
    | FORALL_MODAL(B1,A2)
    | EXIST_NEXT(B)
    | FORALL_NEXT(B)
```

It is the operator MODAL which changes domain. If MODAL(B) is in the state domain (a state formula), then B is in the transition domain (a transition formula). Likewise if MODAL(A) is a transition formula.

As a formula can be either a state or a transition formula, the semantics of a formula depends on which domain we currently are in. Below we give the semantics of the ASK-CTL operators for each domain. The operators are annotated with SML types, and A is the abstract ASK-CTL formula type (an SML constructor type).

Boolean constants:

```
val TT : A
val FF : A
```

The two constant values true and false respectively.

Standard boolean operators:

```
val NOT : A      -> A
val AND : A * A -> A
val OR  : A * A -> A
```

The three operators have the standard interpretation of the boolean functions ¬, ∧, and ∨.

Atomic predicates:

```
val NF : string * (Node -> bool) -> A
val AF : string * (Arc  -> bool) -> A
```

NF is the node function and makes only sense to use as a state sub-formula. Its arguments are a string and a function which takes a state space node and returns a boolean. The string us used when an ASK-CTL formula evaluates to false in the model checker. In this case the model checker will print a diagnostic message explaining why the formula is false, using the string in the message[1]. Thus the string is typically a short statement saying what the node function calculates. The function (Node -> bool) takes a state space node as argument and returns a boolean. It is typically used for identifying single states or a subset of the state space.

AF is the arc function and is analogous to NF, only that it is a transition formula and thus only makes sense to use as a transition sub-formula.

Path quantification operators:

```
val EXIST_UNTIL  : A * A -> A
val FORALL_UNTIL : A * A -> A
```

EXIST_UNTIL used as a state formula takes two arguments, A_1 and A_2, say. The operator is true if there exists a path, starting from where we are now such that A_1 is true for each state along the path until the last state on the path where A_2 must hold. Analogous as a transition formula, but now we consider the transitions on the paths instead.

[1] This feature is not implemented in v0.9 of the library.

FORALL_UNTIL is like EXIST_UNTIL, but now instead of looking for one path only, we require that all paths, starting from where we are now, fulfils the A_1 until A_2 property.

Derived path quantification operators:

```
val POS   : A -> A
val INV   : A -> A
val EV    : A -> A
val ALONG : A -> A
```

By setting the first argument, A_1, to true in the operators EXIST_UNTIL and FORALL_UNTIL we get a number of useful derived operators, POS, INV, EV, and ALONG. They are the most frequently used special case formulas.

POS, as a state formula, is true if it is possible, from the state we are now, to reach a state where the argument, A, is true. Analogous when POS is a transition formula, just for transitions instead.
POS(A) ≡ EXIST_UNTIL(TT,A)

INV, as a state formula, is true if the argument, A, is true for all reachable state, from the state we are at now. Thus the argument, A, is an invariant. Analogous when INV is a transition formula.
INV(A) ≡ NOT(POS(NOT(A)))

EV, as a state formula, is true if the argument, A, becomes true eventually, starting from the state we are now. The argument A must become true within a finite number of steps. Analogous when EV is a transition formula.
EV(A) ≡ FORALL_UNTIL(TT,A)

ALONG, as a state formula, is true if there exists a path for which the argument, A, holds for every state. The path is either infinite or ends in a dead state. Analogous when ALONG is a transition formula.
ALONG(A) ≡ NOT(EV(NOT(A)))

Domain change operators:

```
val MODAL        : A     -> A
val EXIST_MODAL  : A * A -> A
val FORALL_MODAL : A * A -> A
```

This class of operators is our extension of CTL. They make it possible to express properties about transition given that we start from a state with a state formula, or vice versa if we start from a transition with a transition formula.

MODAL, as a state formula, is true if there exists an immediate transition, from where we are now, and if the argument of MODAL, A, is true starting from this transition. The argument, A, must be a transition formula. MODAL, as a transition formula is a bit simpler because a transition always has an immediate destination state. In this case MODAL is true if its argument, A, is true in the destination state. The argument, A, is now a state formula.

EXIST_MODAL is related with MODAL, but takes two arguments, A_1, and A_2. EXIST_MODAL, as a state formula, is true if there exists an immediate successor state, M', in which A_2 holds and A_1 holds on the transition between the current state and M'. A_1 is a transition formula and A_2 is a state formula. Analogous when EXIST_MODAL is a transition formula.
EXIST_MODAL(A_1,A_2) ≡ MODAL(AND(A_1,MODAL(A_2)))

FORALL_MODAL, as a state formula, also looks at immediate successors, but now the argument, A_2, must hold for all immediate successor states and considering each successor state, M', A_1 must hold for the transition from the current state to M'. Analogous when FORALL_MODAL is a transition formula.

Immediate successor operators:

```
val EXIST_NEXT  : A -> A
val FORALL_NEXT : A -> A
```

These two operators are derived from the MODAL operator and are included as a convenient mnemonic notation.

EXIST_NEXT used as a state formula is true iff there exists an immediate successor state, from where we are now, in which the argument, A, is true. Analogous as a transition formula, just for

transitions instead.
EXIST_NEXT(A) ≡ MODAL(MODAL(A))

FORALL_NEXT also looks at immediate successors, but now the argument, A, must hold for all immediate successors.
FORALL_NEXT(A) ≡ NOT(EXIST_NEXT(NOT(A)))

Model Checking

```
val eval_node : A -> Node -> bool
val eval_arc  : A -> Arc  -> bool
```

Given an ASK-CTL formula we can check the formula given the semantics above. This is called model checking and for this purpose the two functions above are given.

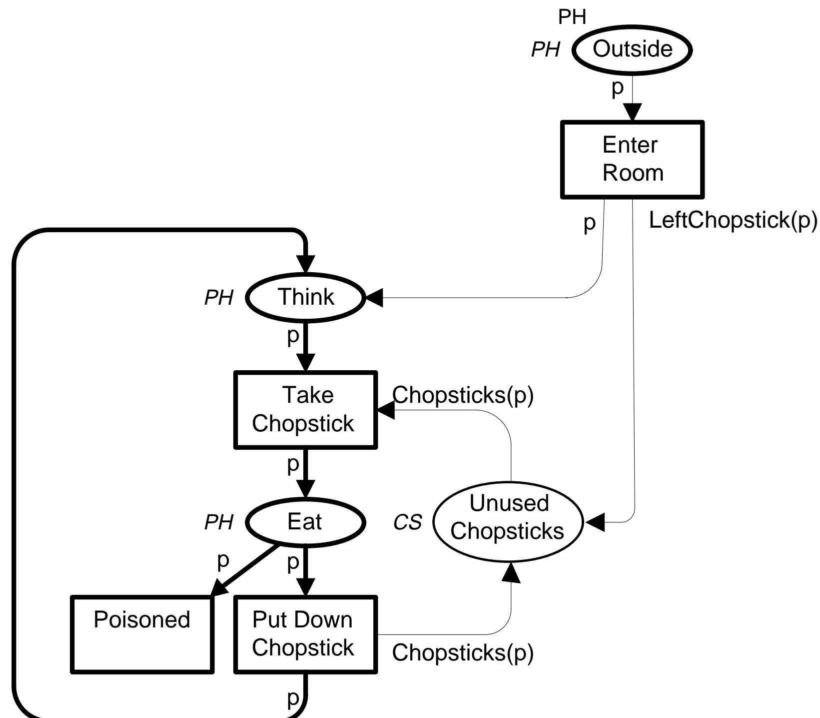
eval_node is used for state formulas. It takes two arguments, the ASK-CTL formula and a state from where the model checking should start. This state is typically the initial marking. The function returns true or false, and in the case of false it also prints out a diagnostic message which shows a counter example[2].

eval_arc is used for transition formulas, and is otherwise similar to the function eval_node.

Examples

Consider a variation of the Dining Philosophers example. A number of philosophers are initially outside a dining room. Each of them decides at some point to enter the room bringing along one chopstick (for the left hand) to be shared with the neighbor philosopher. Once in the room the philosopher sits down and starts thinking. If both left and right chopsticks are unused the philosopher can decide to start eating, thus occupying the two chopsticks. When finished eating the philosopher can decide to think again making available again the left and right chopsticks. As life is unpredictable, a philosopher can get poisoned and die while eating. In this case the chopsticks are forever lost. Obviously this model can end in a dead marking.

[2] This feature is not implemented in v0.9 of the library.




```
val n = 2 ;

color PH = index ph with 1..n declare ms ;
color CS = index cs with 1..n declare ms ;

var p : PH ;

fun Chopsticks(ph(i)) = 1‘cs(i)+1‘cs(i mod n + 1) ;
fun LeftChopstick(ph(i)) = 1‘cs(i) ;
```

We assume that you have completed the sections "How to Install the ASK-CTL Library" and "How to Use the ASK-CTL Logic with Design/CPN". In the following we provide a couple of typical illustrating examples of the use of ASK-CTL.

Below we list some useful example formulas. To make the examples more illustrative we apply formulas to the model above. You can take each of these examples and type them into an auxiliary box (Box from the Aux menu) and invoke ML Evaluate from the Aux menu.

Is the initial marking a home marking?

The following SML source code can be used to check whether the initial marking is a home marking:

```
fun IsInitialMarking n = (n=InitNode) ;

val myASKCTLformula =
       INV(POS(NF("initial marking",
                  IsInitialMarking))) ;

eval_node myASKCTLformula InitNode;
```

Is a given marking dead?

The following SML source code can be used to check whether a given marking is dead. We assume here that the marking in question has the number 8.

```
val myASKCTLformula = NOT(MODAL(TT)) ;

eval_node myASKCTLformula 8 ;
```

Is a given transition live?

Let us consider the take transition from the dining philosophers example. We would like to ask whether the binding element (take,<p=ph(2)>) is live.

```
fun IsConsideredBE a =
      (Bind.System'Take (1,{p=ph(2)})
     = ArcToBE a) ;

val myASKCTLformula =
       INV(POS(MODAL(AF("(take,<p=ph(2)>)",
                        IsConsideredBE))));

eval_node myASKCTLformula InitNode ;
```

Can philosopher 2 become poisoned?

```
fun IsPoisoned n a =
      (Bind.System'Poisoned (1,{p=ph(n)})
     = ArcToBE a) ;

val myASKCTLformula =
       MODAL(POS(AF("Is ph(2) poisoned",
                    IsPoisoned 2))) ;

eval_node myASKCTLformula InitNode ;
```

In general, you will find the manual for the Occurrence Graph Tools useful when writing ASK-CTL formulas [4].

Trouble Shooting

If you have problems with the installation or the use of the library, please contact the maintainer of the library or alternatively:

`designCPN-support@daimi.aau.dk`

Note that this library is a prototype for experimental studies of logics for Coloured Petri Nets. We are not aware of any bugs in the library, so if you discover misbehaviours, you are welcome to contact us.

For general information about Design/CPN, please consult the following page on the World Wide Web:

`http://www.daimi.aau.dk/designCPN/`

Bibliography

[1] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen, *"Model Checking Coloured Petri Nets Exploiting Strongly Connected Components",* in Proc. of the International Workshop on Discrete Event Systems, Institution of Electrical Engineers, University of Edinburgh, UK, August 1996, pp. 169-177.

[2] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen, *"Model Checking Coloured Petri Nets Exploiting Strongly Connected Components",* technical report, University of Aarhus, Denmark, 1996. `http://www.daimi.aau.dk/designCPN/`

[3] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla, *"Automatic Verification of Finite State Concurrent System Using Temporal Logic",* ACM Transactions on Programming Languages and Systems, vol. 8(2), 1986, pp. 244-263.

[4] Kurt Jensen, Søren Christensen, and Lars M. Kristensen, *"Design/CPN Occurrence Graph Manual",* University of Aarhus, Denmark, 1996. `http://www.daimi.aau.dk/designCPN/`

Appendix A: The Signature of the ASK-CTL Structure

The following is the signature of the ASK-CTL SML structure. Upon installation of the library the structure is opened, such that the names are available directly on the top-level, i.e., you do not need to write ASKCTL.eval_node but only eval_node.

```
structure ASKCTL :
  sig

    type A

    val TT  : A
    val FF  : A

    val NOT : A      -> A
    val AND : A * A -> A
    val OR  : A * A -> A

    val NF  : string * (Node -> bool) -> A
    val AF  : string * (Arc  -> bool) -> A

    val EXIST_UNTIL  : A * A -> A
    val FORALL_UNTIL : A * A -> A

    val POS          : A -> A
    val INV          : A -> A
    val EV           : A -> A
    val ALONG        : A -> A

    val MODAL        : A     -> A
    val EXIST_MODAL  : A * A -> A
    val FORALL_MODAL : A * A -> A

    val EXIST_NEXT   : A     -> A
    val FORALL_NEXT  : A     -> A

    val eval_node : A -> Node -> bool
    val eval_arc  : A -> Arc  -> bool

  end
```