Symbolic Synthesis of Knowledge-based Program Implementations with Synchronous Semantics

Xiaowei Huang and Ron van der Meyden

University of New South Wales, Australia
Model Checking vs. Synthesis

Past Work:

Concrete program

Knowledge-based program implementation conditions as formulas of a logic of knowledge and time

Model checking algorithm

Holds

Fails

This Paper:

Knowledge-based program

Synthesis algorithm

Concrete program
Outline

1. Epistemic Model Checking
2. Knowledge-based Programs
   - extend the Unity style programs to encompass use of knowledge in assignment statements, as well as sequential structure
3. Symbolic Synthesis (Implemented in MCK)
   - extend epistemic model checking to automated synthesis of atemporal knowledge-based programs with respect to clock semantics and synchronous perfect recall semantics
4. Examples
   1. Muddy children
   2. Leader election in a ring of processes
5. Conclusions
Epistemic Model Checking

**Specification Logic**

\[ \phi = p \mid \phi_1 \land \phi_2 \mid \neg \phi \mid X\phi \mid K_i\phi \]

**Interpreted System**

\[ \mathcal{I}(M) = (\mathcal{R}, \pi), \text{ where} \]

- \( \mathcal{R} \) is a set of runs over a set \( S \) of underlying states.
  - A point is a pair \((r, m)\) where \( r \in \mathcal{R} \) is a run and \( m \in \mathbb{N} \).
- \( \pi : \mathcal{R} \times \mathbb{N} \rightarrow \mathcal{P}(\text{Prop}) \) is an assignment.

Let \( r(m) \) be the underlying state of point \((r, m)\), and \( r_i(m) \) be agent \( i \)'s local state at \((r, m)\). Define \( \mathcal{K}_i(r, m) = \{(r', m') \in \mathcal{R} \times \mathbb{N} | r_i(m) = r_i'(m')\} \)

\[ \mathcal{I}, (r, m) \models X\phi \text{ if } \mathcal{I}, (r, m + 1) \models \phi \]

\[ \mathcal{I}, (r, m) \models K_i\phi \text{ if } \mathcal{I}, (r', m') \models \phi \text{ for all } (r', m') \in \mathcal{K}_i(r, m) \]
Epistemic Model Checking

Let $O_i : S \rightarrow O$ be a function representing the observation that agent $i$ makes at each state.

**View $V$: the local state of agent $i$ at time $m$**

- $V = \text{clk}: r_i^{\text{clk}}(m) = (m, O_i(r(m)))$, representing that the agent $i$ knows the current time when making an observation.
- $V = \text{spr}: r_i^{\text{spr}}(m) = O_i(r(0))...O_i(r(m))$, representing that the agent $i$ remembers all its observations.

**Model Checking Problem**

Given a system $M$, a view $V$, and a formula $\phi$ of the logic of knowledge and time, MCK builds an interpreted system $I$, and checks whether $\phi$ holds at time 0 in all runs of $I$, i.e., whether $I, (r, 0) \models \phi$ for all $r$ of $I$. 
A group of $n$ children have been playing outside, and some have mud on their foreheads. Each child can see the forehead of the others but cannot see his or her own forehead. Father says to group, “At least one of you has mud on your forehead”. He then repeated asks following question: “Do you know whether or not you have mud on your forehead?” The children give their answers (‘Yes” or “No” ) simultaneously each time the question is asked, and each child observes the answers given by the other children.
Standard Programs

standard program over a set of variables \( V \)

\[
Prog ::= \epsilon \mid stat_1 ; \ldots ; stat_k
\]

\[
stat ::= a \mid \text{if } g_1 \rightarrow a_1 \[] g_2 \rightarrow a_2 \[] \ldots \[] g_k \rightarrow a_k \text{ fi}
\]

where \( g_i \) are guards and \( a_i \) are atomic statements

System model \( S = (M_e, \text{Prot}) \)

1. \( M_e = (\text{Ags}, \text{Acts}, \text{Var}_e, \text{Init}_e, \tau) \), where \( \tau \) is a standard program.
2. \( \text{Prot}_i = (\text{PVar}_i, \text{LVar}_i, \text{OVar}_i, \text{Init}_i, \text{Acts}_i, \text{Prog}_i) \) for \( i \in \text{Ags} \), where \( \text{Prog}_i \) is a standard program.
3. parallel program \( \text{Prog}(S) = \tau \parallel_{i \in \text{Ags}} \text{Prog}_i \)

Transition: \( (s, \tau\|_{i \in \text{Ags}} P_i) \rightarrow (s', \tau\|_{i \in \text{Ags}} P'_i) \)
Example: muddy children (2 agents)

muddy: Bool[Agent]
said: Bool[Agent]

init_cond = (Exists x:Agent() (muddy[x]))

agent Child0 "child" ( said, muddy[Child1]; Child1 )
agent Child1 "child" ( said, muddy[Child0]; Child0 )

deprecated

transitions
begin
said[Child0] := Child0.SayYes;
said[Child1] := Child1.SayYes
end

protocol "child" ( said: observable Bool[], see: observable Bool; opponent: Agent("child"))
begin
% first step
if neg muddy[opponent] -> << SayYes >>
[] otherwise -> skip
fi;
% second step
if said[self] \ (neg said[self] \\ muddy[opponent]) -> << SayYes >>
[] otherwise -> skip
fi
end
In $\text{Prog}_i$, the guards $g$ in conditional statements and the expressions $e$ in the assignments may be formulas of the logic of knowledge.

Example: muddy children (2 agents, clock semantics)

```plaintext
muddy: Bool[Agent]
said: Bool[Agent]

init_cond = (Exists x:Agent() (muddy[x])) \ Forall x:Agent() (info[x] == muddy[x])

agent Child0 "child" ( said, muddy[Child1] )
agent Child1 "child" ( said, muddy[Child0] )

transitions
begin
said[Child0] := Child0.SayYes; said[Child1] := Child1.SayYes
end

protocol "child" ( said: observable Bool[], see1: observable Bool )
begin
  if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes >>
  [] otherwise -> skip fi;
  if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes >>
  [] otherwise -> skip fi
end
```

Xiaowei Huang and Ron van der Meyden
transform a knowledge-based program $\text{Prog}_i$ into its skeleton, denoted $\text{skell}(\text{Prog}_i)$, by replacing each knowledge formula $\phi$ in a guard $g$ or assigned expression $e$, occurring at time $t$, by a new variable $v^t_\phi$.

$\theta$: a substitution mapping each skeleton variable $v^t_\phi \in \text{skellVar}(\text{Prog}_i)$, for $i \in \text{Ags}$, to a boolean expression on the observable variables of agent $i$'s protocol $P_i$. If we apply this substitution to $\text{skell}(\text{Prog}_i)$, we obtain a standard program $\text{skell}(\text{Prog}_i)\theta$.

We now define $\text{skell}(\mathcal{P})\theta$ to be an implementation of the joint knowledge-based protocol $\mathcal{P}$ with respect to the view $\text{clk}$ if $I^{\text{clk}}(\mathcal{M}_e, \text{skell}(\mathcal{P})\theta) \models X^t(\phi \iff \theta(v^t_\phi))$ for all $v^t_\phi \in \text{skellVar}(\mathcal{P})$. 

Xiaowei Huang and Ron van der Meyden () Synthesis Knowledge-based Programs 10 / 22
The following theorem states that in fact, given our assumptions, there is essentially a unique implementation.

**Theorem**

*If* $P$ *is a joint atemporal knowledge-based protocol for environment* $M_e$, *then there exists a substitution* $\theta$ *such that* $\text{skell}(P)\theta$ *is an implementation of* $P$ *in* $M_e$ *with respect to* $\text{clk}$. *Moreover, for all substitutions* $\theta, \theta'$ *such that* $\text{skell}(P)\theta$ *and* $\text{skell}(P)\theta'$ *are implementations of* $P$ *in* $M_e$ *with respect to* $\text{clk}$, *we have that*

$$I^{\text{clk}}(M_e, P\theta) \models X^t(\theta(v^t_\phi) \Leftrightarrow \theta'(v^t_\phi)) \text{ for all } v^t_\phi \in \text{skellVar}(P).$$
Given a joint knowledge-based program $P$ of length $m$, let $P^h$ be the knowledge based program obtained after making the following modifications to $P$:

1. Add to agent protocols:
   1. observable history variables
   2. statements that record observations in these variables

Intuitively, each variable $v@k$ is a new local observable variable that records the value of the original observable variable $v$ of agent $i$ at time $k$. We now define an implementation of $P$ in $M_e$ with respect to the synchronous perfect recall semantics to be an implementation of $P^h$ in $M_e$ with respect to the clock semantics.
Example: muddy children (2 agents, synchronous perfect recall)

muddy: Bool[Agent]
info: Bool[Agent]
init_cond = (Exists x:Agent() (muddy[x])) /\ Forall x:Agent() (info[x] == muddy[x])
agent Child0 "child" ( info[Child1] )
agent Child1 "child" ( info[Child0] )
transitions
begin
info[Child0] := Child0.SayYes; info[Child1] := Child1.SayYes
end

protocol "child" ( info: observable Bool )
begin
if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes >>
[] otherwise -> skip fi;
if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes >>
[] otherwise -> skip fi
end

Transformed into muddy children (2 agents, clock)

protocol "child" ( info: observable Bool )
histinfo1, histinfo2: observable Bool
begin
if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes | histinfo1 := info >>
[] otherwise -> << skip | histinfo1 := info >> fi;
if (Knows Self muddy[Self]) \ (Knows Self neg muddy[Self]) -> << SayYes | histinfo2 := info >>
[] otherwise -> << skip | histinfo2 := info >> fi
end
We work with epistemic Kripke structures $M(S) = (S, \{\sim_i\}_{i \in \text{Ags}}, \pi)$, where

- $S$ is a set of assignments to agent local and environment variables
- $\sim_i$ is an indistinguishability relation of agent $i$, defined as
  - for all $s_1, s_2 \in S$, $s_1 \sim_i s_2$ if and only if $s_1 \upharpoonright \text{OVar}_i = s_2 \upharpoonright \text{OVar}_i$,
- $\pi$ is just the trivial interpretation on $S$, i.e., $v \in \pi(s)$ iff $s(v) = 1$.

Given a model $M$, a state $s \in S$, and an atemporal formula $\phi$, the relation $M, s \models \phi$ can be recursively defined as follows:

- $M, s \models p$ if $p \in \pi(s)$
- $M, s \models \neg \phi$ if not $M, s \models \phi$
- $M, s \models \phi_1 \land \phi_2$ if $M, s \models \phi_1$ and $M, s \models \phi_2$
- $M, s \models K_i \phi$ if $M, s' \models \phi$ for all states $s'$ such that $s \sim_i s'$
Synthesis

Main Idea

For $k = 0 \ldots N$ we define structures $M_k = M(S_k)$ by defining the sets $S_k$. 

\[ \begin{align*}
M_0 & \rightarrow M_1 & \rightarrow \cdots & \rightarrow M_{k-1} & \rightarrow M_k \\
M_{k-1} & \rightarrow M_k
\end{align*} \] 

\[ \begin{align*}
\phi_1 & \rightarrow \phi_2 & \rightarrow \cdots & \rightarrow \phi_m \\
\theta(\phi_1) & \rightarrow \cdots & \rightarrow \theta(\phi_m) \\
\tau |_{i \in Ags_{stat}^t} & \rightarrow \theta \rightarrow M_k
\end{align*} \]
Synthesis

Recursive Procedure

- Initially, we define $S_0$ to be the set of assignments $s$ such that $s$ is a possible initial state. This determines $M_0$.
- For every $k > 0$,
  - For $v^k_{\phi} \in \text{skellVar}(\text{Prog}_i)$, we let $\theta(v^k_{\phi})$ be any formula such that $M_k \models \theta(v^k_{\phi}) \iff \phi$
  - $S_{k+1} = \{ t \mid t \text{ is a state resulting from running } \tau ||_{i \in \text{Ags}} \text{stat}^i_k \theta \text{ (the } k\text{-th step joint program) from a state in } S_k \}$. This determines $M_k$.

Theorem

Let $\theta$ be the substitution defined above. Then $P\theta$ implements $P$ in $M_e$ with respect to the view $\text{clk}$.
Symbolic Implementation

Symbolic Data Structure – Binary Decision Diagram (BDD)

- compact representation
  - heuristic variable reordering, e.g., sifting, etc
- efficient operations, e.g., $\land$, $\neg$, $\exists$, $\forall$, etc.

Implementation

- symbolically use BDD’s to represent the structures $M_k$
- evaluate the applicable knowledge formulas $\phi$ in the structures $M_k$
- extract $\theta(v^k_\phi)$ as a boolean expression over observable variables
In the special case where actions do not change the values of propositions (one example where this holds is the Muddy Children problem) we can encode each stage of a knowledge-based program as an update. Differences include

- DEMO does not include knowledge-based programs as an explicit construct, it does not attempt to synthesize a concrete implementation of such programs.
- In DEMO, the epistemic model $M$, the update structure $U$, and the structure $M \circ U$ are all represented by an explicit enumeration of their states, with a bisimulation quotient is applied to reduce the state space.

It’s interesting to investigate whether the symbolic representation of our approach has the benefit of mitigating the state-space explosion in some interesting cases.
Examples – Muddy Children

Size of Problem

For \( n \) muddy children, we are dealing with an initial state space of \( 2^n - 1 \) states and a deterministic solution protocol that runs for \( n \) steps, giving \( n \cdot (2^n - 1) \) points in the relevant part of the interpreted system.

Performance Evaluation (run time in seconds)

<table>
<thead>
<tr>
<th>No. of Children</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMO</td>
<td>0.54</td>
<td>5.79</td>
<td>71.11</td>
<td>897.28</td>
<td>9,995.10</td>
<td>&gt; 36,000</td>
<td></td>
</tr>
<tr>
<td>MCK clk</td>
<td>0.32</td>
<td>0.88</td>
<td>2.03</td>
<td>6.32</td>
<td>9.09</td>
<td>20.23</td>
<td>57.30</td>
</tr>
<tr>
<td>MCK spr</td>
<td>1.14</td>
<td>5.96</td>
<td>13.13</td>
<td>58.92</td>
<td>96.12</td>
<td>484.22</td>
<td>1,239.60</td>
</tr>
</tbody>
</table>
Examples – Leader Election

Description

- There are \( n \) agents numbered \( i = 1 \ldots n \)
- The leader at time \( t \) is the highest numbered agent that hasn’t crashed.
- An agent can crash at any time, and once crashed, remains crashed.
- Let agent \( i \)'s presumed leader be the largest agent number \( m_i \) for which agent \( i \) considers it possible that \( m_i \) is the leader.
- Agent \( i \) able to send messages to agent \( (i \mod n) + 1 \). In each round, each (noncrashed) agent \( i \) sends its neighbour a message “from \( i \): \( j \)” stating that its presumed leader is \( j \).
- The network delivers any message to the intended recipient, provided that the sender has not crashed.
- If agent \( i \) has crashed, then the network delivers the message that was in agent \( i \)'s buffer to agent \( (i \mod n) + 1 \).

Actions do change the values of propositions (e.g., the presumed leader of agent \( i \)). So we can’t directly use DEMO to model.

(factual changing?)
Examples – Leader Election

protocol "elect" (crashed : Bool, my_num: observable LeaderNum, from_field: observable LeaderNum, message: observable LeaderNum)

presumed: LeaderNum

init_cond = presumed == 3

begin
if (neg crashed) /
  neg Knows Self neg leader == 3 -> <<Send3 | presumed := 3 >>
[] (neg crashed) /
  (Knows Self neg leader == 3 )
    /
    neg Knows Self neg leader == 2 -> <<Send2 | presumed := 2 >>
[] (neg crashed) /
  (Knows Self neg leader == 3 )
    /
    (Knows Self neg leader == 2 )
    /
    neg Knows Self neg leader == 1 -> <<Send1 | presumed := 1 >>
[] otherwise -> skip fi;
(repeat if statement)
end

Performance Evaluation (run time in seconds)

<table>
<thead>
<tr>
<th>No. of Agents, Semantics</th>
<th>Length of Run</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3, MCK clk</td>
<td>2.35</td>
</tr>
<tr>
<td>3, MCK spr</td>
<td>11.26</td>
</tr>
</tbody>
</table>
Conclusions

- the first step towards the goal of a practical tool, based on symbolic methods, for knowledge-based program implementation.
- demonstration on two modest scale examples.

Future Work

- Further application case studies
- Optimisation on the current implementation.
- Handling probabilistic knowledge.