Writing a CTLSKD Model-Checker in Ocaml

AURÈLE BARRIÈRE

ACM Reference Format:

Aurèle Barrière. 2018. Writing a CTLSKD Model-Checker in Ocaml. 1, 1 (December 2018), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Temporal Logics are a convenient formalism when it comes to reasoning about dynamic systems. Similarly, Epistemic Logics allow to describe and reason about knowledge in distributed systems, or systems with imperfect information. Combining the two kinds of logics for Epistemic Temporal Logics is a popular field with many uses, as in Artificial Intelligence or Game Theory. In many real-world applications, several agents are involved, each with its own view of the world, and things evolve dynamically.

In an earlier work [4], we defined a new Epistemic Temporal Logic, CTL*KΔ. It extends the CTL*K logic, a well-known epistemic extension of CTL*, a popular temporal logic. While CTL*K added to CTL* the K operator to model agents knowledge, CTL*KΔ adds the operator Δ, that allows agents to change their "observational power", or their point of view on the system. For instance, the formula $\Delta^o KAXp$ means that after changing to an observation o, the agent knows that, on the next step, the proposition p will hold, whatever path has been taken by the system. To the best of our knowledge, this was the first time that such a change was enabled in an Epistemic Temporal Logic.

For Epistemic and/or Temporal Logics, model-checking (deciding if a formula is true in a model) is an important problem, as it allows to confront a specification and the modelization of a system. In [4], we showed that model-checking CTL*KΔ could be reduced to the model-checking of CTL*. The model-checking of CTL* is itself known to be reducible to the model-checking of LTL [7], another classical Temporal Logic. We previously defined and model-checked CTL*KΔ in a multi-agent setting. However, this work will focus on the implementation of a single-agent CTL*KΔ model-checker. This choice was made for simplicity and the algorithm is almost identical.

In this report, I present my work on implementing a model-checker of CTL*KΔ. To this end, we use NuSMV [6], a SAT-based LTL model-checker, and build a CTL* model-checker, then a CTL*KΔ model-checker, using the reductions. The implementation is done in OCaml, and can be found online [1]. In a first section, we present the logics and their formalization, then the reductions. We then discuss the implementation itself. Finally, we present a way to evaluate our implementation.

Author's address: Aurèle Barrière.

2 FORMALLY MODEL-CHECKING CTL*KΔ

Most of this section comes from our previous work [4, 5]. More details can be found there. The following sections will only focus on new work.

2.1 Single agent CTL*KΔ

2.1.1 Syntax. We begin by introducing the syntax of CTL*KΔ. We consider O to be a set of *observations*, that each represent a possible observational power of the agent. AP is a set of atomic propositions. Formulas of CTL*KΔ can be *history formulas* φ or *path formulas* ψ.

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\psi \mid E\psi \mid K\varphi \mid \Delta^o\varphi$$
$$\psi := \varphi \mid \neg\psi \mid \psi \wedge \psi \mid X\psi \mid \psi U \psi$$

Where $p \in AP$ and $o \in O$. The temporal operators X and U are meant to represent the typical *next* and *until* operators of temporal logics. A is a path quantifier, similar to those found in branching-time logics. Intuitively, $A\psi$ should hold for a history if ψ is true in every possible future. E is A's dual. K is an epistemic operator. Intuitively, $K\varphi$ should be true whenever the agent knows that φ is true. We introduce a new one, Δ^o, to represent a change of observation. φ is called a history formula, as we only need to know what happened in the past to decide if the formula is true. ψ is called a path formula as it also requires the future evolution of the system.

Models. The models on which such formulas can be interpreted are classical Kripke Structure, with several equivalence relations between states (one for each observation). We note $M = (S, I_s, o_I, T, V, \sim_{o_1}, \dots, \sim_{o_m})$. A *run* or *path* is an infinite sequence of states $\pi = \pi_0\pi_1 \dots$. A *history* is a finite sequence of states $h = h_1 \dots h_n$.

2.1.2 Record Semantics. We first define the natural semantics of CTL*KΔ. To this end, we first define Observation Records.

Observation records. Given O a set of observations, we define *observations records* to be ordered lists of pairs of observations and natural numbers. Intuitively, an observation record represents changes of observations.

Example: $r = [(o_1, 0), (o_2, 3), (o_3, 3)]$ means that the player starts at time 0 with observation o_1. It keeps this observation, then at time 3, it first changes to o_2 and then to o_3. We use observation records in the semantics to remember the previous observations of the agent.

We write $r[(o, n)]$ to append a new pair (o, n) to the observation record r. We write $r_{\leq n}$ the record r without the pairs (o, m) where $m > n$. We write r_n the record r without the pairs (o, m) where $m \neq n$. We define a function $O(r, n)$ which gives a tuple of the observations used at time n.

On a given model, with an observation record we can define an equivalence relation between histories, with regard to a record. Two histories are equivalent with regard to the record if the player can't distinguish them by using the observations in the record.

$h \approx_r h'$ *iff* $\forall i < |h|, \forall o \in O(r, i), h(i) \sim_o h'(i)$ *and* $|h| = |h'|$.

Finally, the semantics are defined as follows: History formulas need a history h (finite sequence of previous states) and an observation record r to be interpreted, to know which history might be considered possible for the agent. Path formulas are interpreted on a run π (infinite sequence of states of the model), a point in time (natural number), and an observation record.

$M, h, r \models p$	*iff*	$p \in V(last(h))$
$M, h, r \models \neg\varphi$	*iff*	$M, h, r \not\models \varphi$
$M, h, r \models \varphi_1 \wedge \varphi_2$	*iff*	$(M, h, r \models \varphi_1$ and $M, h, r \models \varphi_2)$
$M, h, r \models \varphi_1 \vee \varphi_2$	*iff*	$(M, h, r \models \varphi_1$ or $M, h, r \models \varphi_2)$
$M, h, r \models A\psi$	*iff*	$\forall\pi$ that extends h, we have $M, \pi, \lvert h\rvert - 1, r \models \psi$
$M, h, r \models E\psi$	*iff*	$\exists\pi$ that extends h, we have $M, \pi, \lvert h\rvert - 1, r \models \psi$
$M, h, r \models K\varphi$	*iff*	$\forall h'$ such that $h' \approx_r h$, we have $M, h', r \models \varphi$
$M, h, r \models \Delta^o\varphi$	*iff*	$M, h, r[(o, \lvert h\rvert - 1)] \models \varphi$
$M, \pi, n, r \models \varphi$	*iff*	$M, (\pi_0 \dots \pi_n), r \models \varphi$
$M, \pi, n, r \models \neg\psi$	*iff*	$M, \pi, n, r \not\models \psi$
$M, \pi, n, r \models \psi_1 \wedge \psi_2$	*iff*	$(M, \pi, r, n \models \psi_1$ and $M, \pi, r, n \models \psi_2)$
$M, \pi, n, r \models X\psi$	*iff*	$M, \pi, (n + 1), r \models \psi$
$M, \pi, n, r \models \psi_1 U \psi_2$	*iff*	$\exists m \geq n$ such that $\forall k \in [n, m[, M, \pi, k, r \models \psi_1$ and $M, \pi, m, r \models \psi_2$

Finally, we say that $M = (S, I_s, o_I, T, V, \sim_{o_1}, \dots, \sim_{o_m})$ models φ (written $M \models \varphi$), if $\forall s \in I_s, M, s, [(o_I, 0)] \models \varphi$. This definition corresponds to the model-checking problem.

2.2 Model-Checking CTL*KΔ

Because of perfect-recall semantics, it may seem that we have to remember the complete history and records when evaluating a formula. However, we can extract some information from the history that is sufficient for the evaluation of the formula. Intuitively, to evaluate a history formula, it is enough to know the current state, the current observation and the set of states that the agent believes the system might be in (called the *Information Set*). This new structure to represent the knowledge is more succinct than remembering entire histories and records, as there is a finite number of information sets.

We define two functions to update information sets. U_Δ updates the set when a player goes through a change of observation and U_T updates the set when the player moves to a new state.

$$\begin{array}{lcl} U_\Delta(I, s, o) & = & \{x \in I \mid x \sim_o s\} \\ U_T(I, s, o) & = & \{x \in S \mid \exists t \in I, t \longrightarrow x \text{ and } x \sim_o s\} \end{array}$$

A few rules from this new semantics are:

$M, s, I, o \models p$	*iff*	$p \in V(s)$
$M, s, I, o \models A\psi$	*iff*	$\forall\pi$ such that $\pi_0 = s$, we have $M, \pi, I, o \models \psi$
$M, s, I, o \models K\varphi$	*iff*	$\forall s' \in I$, we have $M, s', I, o \models \varphi$
$M, s, I, o' \models \Delta^o\varphi$	*iff*	$M, s, U_\Delta(I, s, o), o \models \varphi$
$M, \pi, I, o \models X\psi$	*iff*	$M, \pi_{1\dots}, U_T(I, \pi_1, o), o \models \psi$

We proved in [4] the following theorem: $\forall\varphi, h, r, s, I, o$ such that $FH(h, r) = (s, I, o)$, $M, h, r \models \varphi$ *iff* $M, s, I, o \models \varphi$. Where *FH* is a function that relates histories and observation records to the corresponding current state, information set and observation. This proves that the two semantics are equivalent, and it suffices to model-check the information set semantics.

To this end, from $M = (S, I, o_I, T, V, \sim_{o_1}, \dots, \sim_{o_m})$ we define the augmented model $\hat{M} = (S', T', V')$, a Kripke Structure.

- $S' = S \times 2^S \times O$: states are state of the original model, an observation set and an observation.
- $(s, I, o)\ T'\ (s', I', o)$ *iff* $s\ T\ s'$ and $I' = U_T(I, s', o)$
- $V'(s, I, o) = V(s)$. As the algorithm is executed, new atomic propositions will appear.

Finally, any formula of CTL*KΔ can be model-checked with a marking algorithm on the (finite) augmented model. Intuitively, for each $\Delta^o\varphi$ where φ is a CTL* formula, we model-check φ on each state of $\hat{M}$. Then, we mark each state

(s, I, o') of $\hat{M}$ with a new atomic proposition if φ holds on $(s, U_\Delta(I, s, o), o)$. Similarly for formulas $K\varphi$ where φ is a CTL* formula, a state (s, I, o) is marked with a new atomic proposition if φ holds in each (s', I, o) where $s' \in I$.

Finally, we replace either $\Delta^o\varphi$ or $K\varphi$ in the formula with the new atomic proposition, and start from the beginning until there are no more K or Δ operators in the formula. The remaining formula can be model-checked with the CTL* model-checker.

The definition of the Information Set semantics makes the correctness of the algorithm almost trivial.

3 OUR APPROACH TO IMPLEMENTING A CTL*KΔ MODEL-CHECKER

3.1 Types

Our models can either be standard models, where each state can be numbered, or augmented models for the model-checking of CTL*KΔ. States of augmented models include a standard state, an information set and an observation

We thus define our states as such:

```
type std_state = int
type inf_set = std_state list
type state = I of std_state
           | A of std_state * inf_set * observation
```

When building augmented states, we make sure that the information sets have no duplicates and are ordered, which guarantees that each augmented state has a unique representation.

Kripke structures are then defined. The `kripke` type represents the states and transition of a Kripke Structure: each element of the list is a state and its list of successors. A marking represents the valuation function. Each element of the list contains an atomic proposition and the list of states where it holds. Decoupling the two types allows to add new atomic propositions without changing the underlying structure, which will prove useful for marking algorithms. Observations equivalence relations are described with the `obs_marking` type. It maps observations and states to a value.

```
type kripke = (state * state list) list
type std_kripke = (std_state * std_state list) list
type marking = (atp * state list) list
type std_marking = (atp * std_state list) list
module ObsMap = Map.Make(struct type t = (observation * std_state) let compare = compare end)
type obs_marking = int ObsMap.t
let eq_state (om:obs_marking) (o:observation) (s1:std_state) (s2:std_state): bool =
  ObsMap.find (o,s1) om = ObsMap.find (o,s2) om
```

We then define the 3 logics' syntaxes:

```
type ltl =
  | LTL_TRUE
  | LTL_AP of atp
  | LTL_NEG of ltl
  | LTL_OR of ltl * ltl
  | LTL_AND of ltl * ltl
  | LTL_X of ltl
  | LTL_U of ltl * ltl

type state_ctls =
  | ST_CTLS_TRUE
  | ST_CTLS_AP of atp
  | ST_CTLS_NEG of state_ctls
  | ST_CTLS_OR of state_ctls * state_ctls
  | ST_CTLS_AND of state_ctls * state_ctls
  | ST_CTLS_A of path_ctls
  | ST_CTLS_E of path_ctls
 and path_ctls =
   | P_CTLS_S of state_ctls
   | P_CTLS_NEG of path_ctls
   | P_CTLS_OR of path_ctls * path_ctls
   | P_CTLS_AND of path_ctls * path_ctls
   | P_CTLS_X of path_ctls
   | P_CTLS_U of path_ctls * path_ctls
```

```
type history_ctlskd =
  | H_CTLSKD_TRUE
  | H_CTLSKD_AP of atp
  | H_CTLSKD_NEG of history_ctlskd
  | H_CTLSKD_OR of history_ctlskd * history_ctlskd
  | H_CTLSKD_AND of history_ctlskd * history_ctlskd
  | H_CTLSKD_A of path_ctlskd
  | H_CTLSKD_E of path_ctlskd
  | H_CTLSKD_K of history_ctlskd
  | H_CTLSKD_D of observation * history_ctlskd
 and path_ctlskd =
   | P_CTLSKD_H of history_ctlskd
   | P_CTLSKD_NEG of path_ctlskd
   | P_CTLSKD_OR of path_ctlskd * path_ctlskd
   | P_CTLSKD_AND of path_ctlskd * path_ctlskd
   | P_CTLSKD_X of path_ctlskd
   | P_CTLSKD_U of path_ctlskd * path_ctlskd
```

Given these types, we can define the signature of our 3 model-checking procedures:

```
let ltl_mc (k:kripke) (init:state) (m:marking) (spec:ltl): bool = ...
let ctls_mc (k:kripke) (init:state) (m:marking) (spec:state_ctls): bool = ...
let ctlskd_mc (k:std_kripke) (state_init:std_state) (m:std_marking) (obs_init:observation)
              (om:obs_marking) (spec:history_ctlskd): bool = ...
```

Notice how LTL and CTL* model-checkers take as input generic kripke structure (either standard or augmented), while the CTL*KΔ function takes a standard kripke structure. It will then build the augmented model and give it to the CTL* function, which in turn calls the LTL one on the same model. Similarly, as only CTL*KΔ has imperfect information, its function is the only one requiring an observation marking and an initial observation.

3.2 LTL Model-Checking

As we use NuSMV, a LTL and CTL model-checker, our model-checking procedure can simply call it. However, NuSMV only works on an input file. We then write several functions to translate a LTL model-checking problem as a file following the syntax of NuSMV. We then call the NuSMV executable on this file. Because its output is a long string, we write a small wrapper in C to analyze it and simply return 1 or 0 depending on the result. This returned value can be obtained in Ocaml by using the `Sys.command` function.

3.3 CTL* Model-Checking

An algorithm to create a CTL* model-checker from a LTL one can be seen in [7]. This is a simple marking algorithm on formulas of the form $E\varphi$.

However, in this algorithm, the authors assume that the LTL model-checking procedure returns true if the formula is true in at least one path starting from the initial state (LTL formulas are evaluated on paths). NuSMV returns true if the formula holds in all paths starting from the initial state. We thus adapt the algorithm to mark on all sub-formulas $A\varphi$, and replace each $E\varphi$ with $\neg A \neg \varphi$.

Finally, all the marking algorithms from [7] and [4] were written in an imperative way. We design a marking recursive function for functional languages. We use the pattern-matching feature of OCaml to match on the inductive structure of formulas to model-check. In practice, our function has the following signature:

```
let rec marking_update (k:kripke) (m:marking) (spec:state_ctls): (marking * state_ctls) = ...
and path_marking_update (k:kripke) (m:marking) (spec:path_ctls): (marking * path_ctls) = ...
```

This function takes a structure (with its marking of atomic propositions) and a specification. It returns a new marking and a new specification, where in the new formula, each sub-formula $A\varphi$ has been replaced with a fresh atomic proposition, and the new marking contains the truth assignment for each of these new propositions.

Most cases simply call the function recursively on subformulas:

```
  | ST_CTLS_OR (s1,s2) -> let (newm1, newspec1) = marking_update k m s1 in
                          let (newm2, newspec2) = marking_update k newm1 s2 in
                          (newm2, ST_CTLS_OR (newspec1, newspec2))
```

When dealing with an $A\varphi$, we call the model-checker on each states to mark the correct states (function `check_ltl` calls `ltl_mc` on all states), and replace $A\varphi$ with the proposition `fresh`:

```
| ST_CTLS_A p ->
   begin match is_linear_ctls p with
   | true -> let fresh = fresh_atp m in
             let newm = (fresh, check_ltl k m (linear_ctls_to_ltl p))::m in
             (newm, ST_CTLS_AP fresh)
   | false -> let (newm, newspec) = path_marking_update k m p in
              marking_update k newm (ST_CTLS_A newspec)
   end
```

Note that, if φ is not a linear formula, we don't replace it right away, as the LTL procedure only deals with LTL formulas. In that case, we first replace the inner sub-formulas.

We have one such case for each possible inductive case. We don't define any for $E\varphi$ as we replaced them earlier. Finally, the `marking_update` function returns a new marking and a new specification. This formula is linear (no more $A\varphi$), and checking it in LTL with the new marking is equivalent to model-checking the original problem.

3.4 CTL*KΔ Model-Checking

As seen on section 2, model-checking CTL*KΔ can be done once again with a marking algorithm. However, this marking operates on the augmented model. We first define functions to build an augmented model (along with its valuation of atomic propositions) from a standard kripke structure.

```
let augmented_kripke (k:std_kripke) (obs:observation list) (om:obs_marking): kripke = ...
let rec augmented_marking (m:std_marking) (states: state list): marking = ...
```

We can then define the marking algorithm in a similar way as CTL*. This time, the formulas that will be replaced by fresh atomic propositions are $K\varphi$ and $\Delta^o\varphi$, where φ is a CTL* formula. More details can be found in [4]. For instance, for $\Delta^o\varphi$ formulas:

```
| H_CTLSKD_D (o,h) ->
   begin match (is_history_ctls h) with
   | true -> let states_h_true = check_ctls k m (history_ctlskd_to_ctls h) in
             let fresh = fresh_atp m in
             let newm = (fresh, List.filter (filter_delta om states_h_true o) (get_states k))::m in
             (newm, H_CTLSKD_AP fresh)
   | false -> let (newm, newspec) = aug_marking_update k m om h in
              aug_marking_update k newm om (H_CTLSKD_D (o,newspec))
   end
```

4 EVALUATION

Our implementation was successfully written in Ocaml. It is available online [1], but requires a NuSMV executable (also available online for free [2]). It required over 800 lines of commented OCaml and a small NuSMV wrapper written in C.

Only the model-checking procedures were implemented, meaning that there is currently no parser to get a formula and a model. One has to write directly in the Ocaml syntax, which makes for unnecessary long definitions.

To evaluate our implementation, we first wrote a few generic tests on small models, such that we could know the answer. All of these tests return the expected result.

We then created parametric tests for each of our logics. These take 2 arguments (the model's size and the formula's size), build a problem and solve it. This allows to study the time complexity of our procedures. We picked these models and formulas for their simplicity and our ability to predict their results (to check for any errors in the implementation).

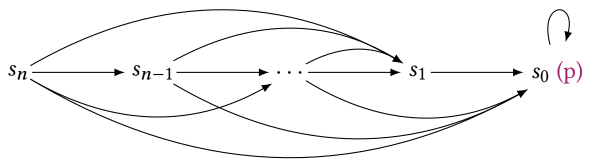

Fig. 1. Parametric Model for LTL

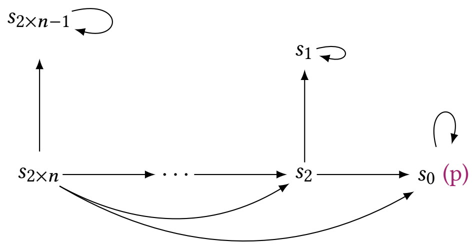

Fig. 2. Parametric Model for CTL*

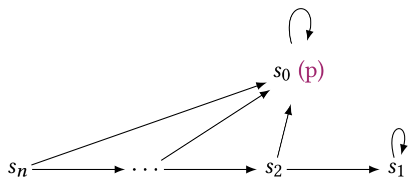

Fig. 3. Parametric Model for CTL*KΔ

4.1 Evaluation Setting

Evaluating LTL. Given a size n, we build the model that can be seen on Figure 1. Given a formula size m, we build the following specification: $\phi_0 = (true\ U\ p), \quad \phi_{m+1} = true\ U\ X(\phi_m)$. We can easily check that, for any formula size and model size, the formula holds (with initial state s_n).

The Figure 4 gives the time it took for our solver to build the example and find that result, for different values of m and n.

Evaluating CTL.* The parametric model is given Figure 2. The formulas are $\phi_0 = E(true\ U\ p) \quad \phi_{m+1} = EX\phi_m$. The Figure 5 gives the results.

*Evaluating CTL*KΔ.* The parametric model is given Figure 3. The formulas are $\phi_0 = \Delta^{o_2} K \neg p \quad \phi_{m+1} = EX\phi_m$. The Figure 5 gives the results. The initial observation is o_1, the blind observation where all states are indistinguishables. The observation o_2 is the perfect one, where the agent can distinguish each state. The initial state is s_n.

4.2 Result analysis

As we only checked our procedures on one example each, the depicted results are not enough to make general assumptions about the performance and complexity of our algorithms, but they provide a valuable insight. First, we know that both LTL and CTL* model-checking problems are in PSPACE-complete [8]. We showed in [5] that the single-agent CTL*KΔ model-checking problem is in EXPTIME. However, no lower-bounds have been established yet.

We can see that both marking algorithm (for CTL* and CTL*KΔ) seem to behave linearly in the size of the formula. Indeed, in our examples, the constructors are all nested. Consider the formulas $\phi_0 = E(true\ U\ p) \quad \phi_{m+1} = EX\phi_m$. When model-checking ϕ_m on a model, we first model-check ϕ_0 on each states of the augmented model, and add a new marking. We then model-check the formula EXa, where a is a new atomic proposition (true on the states where ϕ_0 holds). In general, to model-check ϕ_{m+1}, we model-check the formula EXq where q is an atomic proposition true on states where ϕ_m holds. This amounts to using NuSMV on the LTL formula $\neg Xq$. The time to model-check this formula does not depend on m, as the entire formula ϕ_m has been reduced to a simple atomic proposition. This explains why our marking algorithms are linear in the size of the formulas in our examples.

Finally, the number of states in the augmented model is easily computable: $|S| \times 2^{|S|-1} \times |O|$ (we only build the states (s, I, o) where $s \in I$, as the others are not reachable). This explains why the time of model-checking CTL*KΔ explodes so quickly in Figure 6.

5 FUTURE WORKS

Our implementation seems to give correct results so far, but could be improved in numerous ways.

Parsing. We could allow the user to write an input file in a given syntax (maybe the same as NuSMV's).

Multi-Agent. In [5], we define CTL*KΔ for multi agent settings. Each K and Δ operators takes an additional parameter to indicate the relevant agent. We showed that the model-checking algorithm is almost identical, but needs to use k-trees instead of Information Sets (a tree that represents one agent's knowledge of the other agents' knowledge, introduced in [9]). This would be an easy extension to our implementation.

Polymorphism. We didn't use all the polymorphic features of OCaml. A more elegant way to define models would have been to use a more abstract type for states. We kept it simple for this proof-of-concept implementation.

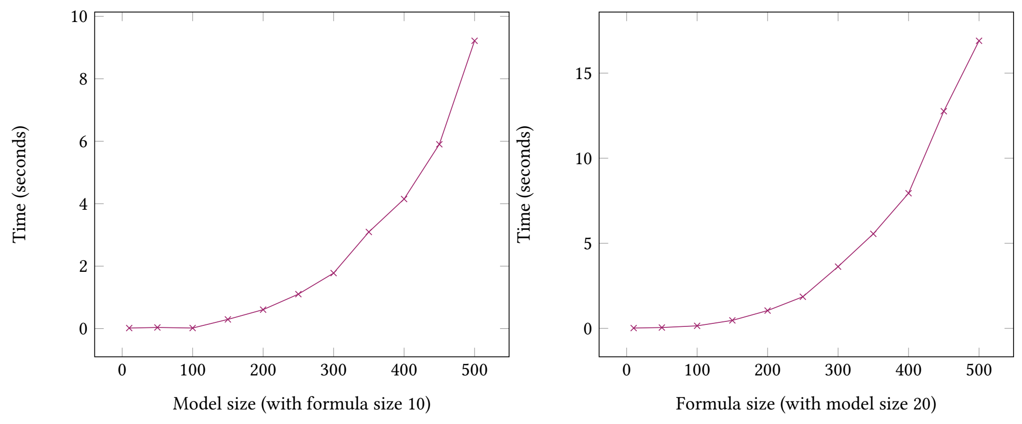

Fig. 4. LTL Time Results

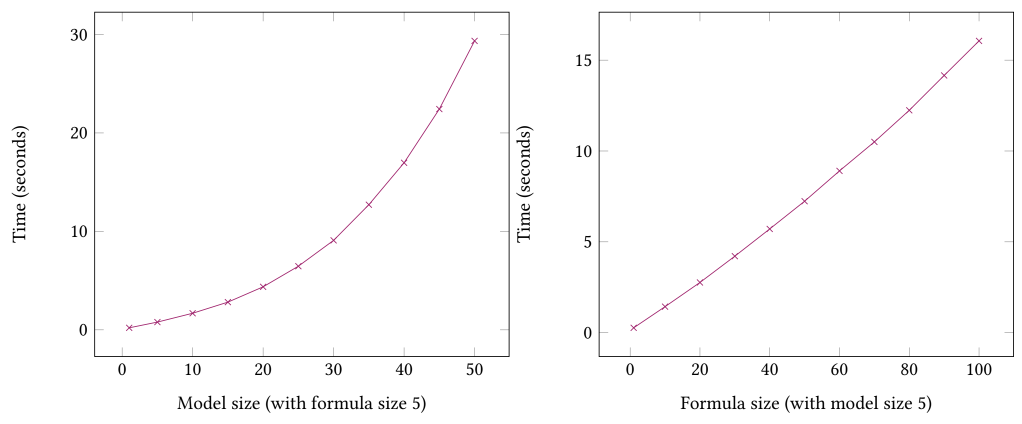

Fig. 5. CTL* Time Results

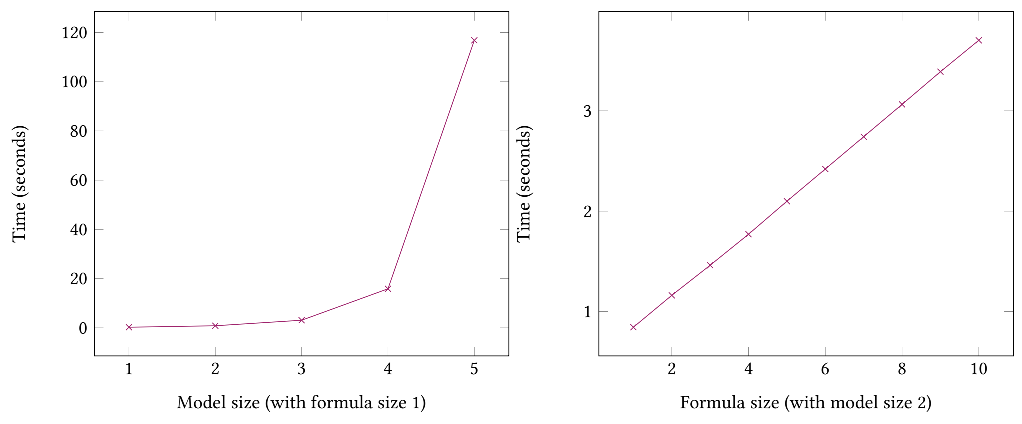

Fig. 6. CTL*KΔ Time Results

Other Solvers. We currently only use NuSMV, but there are numerous other LTL solvers. Another good choice would have been SPIN [3]. It would be interesting to compare the results in our setting.

Binding NuSMV. In this version, we write to a file each LTL problem for NuSMV to parse and solve. We might be losing a lot of time doing useless writing and parsing (our representation is already parsed). A good solution would be to bind our representation to NuSMV's, and call their own functions directly. Such C-OCaml interfacing can be done, but would require some time to understand NuSMV's code.

Data structures. As the augmented model grows exponentially in the size of the original model, keeping a list of states and transitions might not be the most optimal choice.

Splitting the augmented model. In our CTL*KΔ model-checking procedure, we give the entire augmented model to the CTL* model-checker. However, the augmented model is actually composed of o disjoint components, where o is the number of observations. As CTL* and LTL model-checking complexity depends on the size of the model, we could probably save some time by giving them the appropriate component of the augmented model. If NuSMV performs reachable state analysis, this would only save the time of writing and parsing too many states and transitions.

Counter-Example generation. NuSMV finds a counter-example in false specifications. We believe that, using these counter-examples, we could build one for the CTL*KΔ model-checking problem. However, NuSMV's output when printing a counter-example isn't directly exploitable (states are renamed for instance).

6 ACKNOWLEDGMENTS

I am grateful to Bastien Maubert, Sasha Rubin and Aniello Murano for working with me on [4, 5], and advising me on this project. In particular, they redirected me to NuSMV and SPIN, and towards the reduction from CTL* to LTL in [7].

7 RELATED WORKS AND CONCLUSION

Our previous work already defined CTL*KΔ. Our implementation is based on the model-checking procedure we wrote in [4]. The reduction technique resembles other algorithms from model-checking. Marking algorithms have been used previously (in [7] for instance). However, all implementations in the literature seem to be written in an imperative way. Information Sets are a particular case of k-trees [9].

We implemented a CTL*KΔ model-checker in OCaml. We make use of its functional features to implement functional marking algorithms. The model-checking problem was solved in [4] by means of a reduction to CTL* model-checking.

The functional style of OCaml makes for elegant definitions of logics and marking algorithms. However, we still have a long way to go before this can check large models in a reasonable time. Part of it comes from the complexity of CTL*KΔ model-checking in itself, but we could also optimize our implementation. Nevertheless, our implementation has given a correct results on all of our tests.

REFERENCES

[1] A CTLSKD Model-Checker in Ocaml. https://github.com/Aurele-Barriere/CTLSKD_MC.

[2] Getting NuSMV. http://nusmv.fbk.eu/NuSMV/download/getting-v2.html.

[3] The spin model-checker. http://spinroot.com/spin/whatispin.html.

[4] Aurèle Barrière. Defining and Model-Checking an Epistemic Temporal Logic with Changes of Observations, 2017.

[5] Aurèle Barrière, Bastien Maubert, Aniello Murano, and Sasha Rubin. Changing observations in epistemic temporal logic. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, pages 621–622, 2018.

[6] 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 *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 359–364, 2002.

[7] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[8] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse, France, 30 September - 2 October 2002*, pages 393–436, 2002.

[9] Ron van der Meyden and Nikolay V. Shilov. Model checking knowledge and time in systems with perfect recall (extended abstract). In *Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings*, pages 432–445, 1999.