Dynamic Epistemic Logic

B. Nebel, R. Mattmüller, T. Engesser
Winter Semester 2016/2017

University of Freiburg
Department of Computer Science

Exercise Sheet P1

Due: November 28th, 2016, 20:00

Your task in this practical exercise is to implement a model checker that takes an epistemic model and some formulas as input and decides which of the formulas are satisfied, either for each single world in the model individually, or for a specific designated world.

The model and formulas are to be input into the program as a single JSON object, consisting of a dictionary with the two attributes `model` and `formulas`, where `model` specifies the epistemic model and `formulas` is a list of one or multiple formulas that are to be checked. Additionally, a third attribute `designated` can be specified which is used to point out a designated world of the model in which the formulas are to be exclusively checked. Have a look at the following example:

```
{"model": {"domain": ["w1", "w2", "w3"],
           "indist": [[2, "w1", "w2"], [1, "w2", "w3"], [2, "w3"]],
           "val": {"p": ["w1", "w2"], "q": ["w2", "w3"]}},
 "formulas": ["p&~q", "K1p", "K2p", "K1K2p", "K2K1p"],
 "designated": "w1"}
```

Both world and proposition names are represented as strings matching the regular expression `[a-z][a-zA-Z0-9]*` (they have to begin with a lowercase letter and may contain also arbitrary uppercase letters and numbers). Agent names are represented by positive integers (agent indices).

We then represent formulas as strings where, besides proposition and agent names, the characters `~`, `&`, `|`, `K`, and `C` are used for negation, conjunction, disjunction, the knowledge operator and the common knowledge operator (for the set of all agents). Since, by our naming conventions, proposition names start by lower case letters, there is no ambiguity. E.g., `CK1k2CK1` represents the formula describing that it is common knowledge between all agents that Agent `1` knows that proposition `k2CK1` holds. In practice, we will try to avoid confusing proposition names.

Furthermore, we allow the characters `(` and `)` for parenthesizing (sub-)formulas. In the absence of parentheses, we define the unary operators `~`, `C`, and `K` (for an arbitrary agent) to have precedence over the binary operators `&` and `|`. We don't define precedence between `&` and `|`. Instead, we assume that formulas are implicitly parenthesized to the left. E.g., `p&Cq|K1r&s` corresponds to the explicitly parenthesized formula `((p&Cq)|K1r)&s`.

The model specification then is a dictionary consisting of the following attributes:

- `domain`, a flat list containing all unique world names. Specifies the domain of the model.
- `indist`, a nested list specifying the epistemic indistinguishability relations for the agents. Each sublist defines an equivalence class for some agent. It contains the agent name first, followed by all worlds that belong to that particular equivalence class. Each world that is not listed explicitly within an equivalence class list for some agents is assumed to have its own equivalence class (and thus to be distinguishable from all other worlds) for that agent.
- `val`, a dictionary specifying the model's valuation function. It maps each proposition to a list containing exactly the names of the worlds for which the proposition is true.

If a designated world is specified, the output should be a list containing exactly the formulas from the formula list that are satified in that world. Otherwise the output should be a dictionary mapping each formula to a list of exactly the worlds in which the formula is satisfied.

E.g., the model checker should output the JSON object `["p&~q", "K1p", "K2p", "K1K2p"]` for the input above. If the designated world was left out, it should output the following object:

```
{"p&~q": ["w1"],
 "K1p": ["w1"],
 "K2p": ["w1", "w2"],
 "K1K2p": ["w1"],
 "K2K1p": []}
```

Your program is supposed to read the specification from the standard input and write the result to the standard output. Two additional input files, containing examples from the theoretical exercise sheets (Hanabi, Cheryl's birthday) will be provided. Your are supposed to work in groups of two students and use Python as the programming language. We will test and evaluate your submissions in terms of correctness and performance (only correctness is required to achieve a full score).

Exercise P1.1 (Implementation, 10 points)

Implement a model checker satisfying our specifications. You might first want to implement a parser that reads out the JSON input into appropriate Python data structures. Reading JSON objects can be easily done by Python's json module. For parsing the formula strings, you're allowed (but not required) to use third party libraries. Be aware that your choice of data structures (e.g., for model and formula representation) is critical for the performance of your model checker. For model checking, you can e.g. choose between a top-down or bottom-up approach (they both have their merits and drawbacks depending on the application scenario).

Exercise P1.2 (Benchmarking, 4 points)

Create a new problem instance that takes your model checker at least one second to solve. We will use your instances for comparing your implementations in terms of performance, so ideally they should be challenging, but efficently solvable by your implementation. Generate the outputs both for your instance as well as for the provided Hanabi and Cheryl's birthday instances.

Exercise P1.3 (Documentation, 4 points)

Briefly explain the important ideas behind your implementation. How do you represent models and formulas? How does your model checking approach work? Make a short presentation (maximally three or four slides for a five minute talk) describing your approach and discussing its advantages and disadvantages. Be prepared to present it in the exercise sessions.