EpiPEn: A DSL for Solving Epistemic Logic Puzzles with Z3

Josh Horowitz, Jack Zhang

December 10, 2021

1 Introduction

There's a class of logic puzzles that, at first sight, looks impossible to solve. A classic example is *Cheryl's Birthday* [1]

> Albert and Bernard just met Cheryl. "When's your birthday?" Albert asked Cheryl.
>
> Cheryl thought a second and said, "I'm not going to tell you, but I'll give you some clues". She wrote down a list of 10 dates:
>
> > May 15, May 16, May 19, June 17, June 18, July 14, July 16, August 14, August 15, August 17.
>
> "My birthday is one of these", she said.
>
> Then Cheryl whispered in Albert's ear the month—and only the month—of her birthday. To Bernard, she whispered the day, and only the day.
>
> "Can you figure it out now?" she asked Albert.
>
> > Albert: I don't know when your birthday is, but I know Bernard doesn't know either.
> >
> > Bernard: I didn't know originally, but now I do.
> >
> > Albert: Well, now I know too!
>
> When is Cheryl's birthday?

This puzzle is particularly difficult, because it requires us to reason at a higher level than first order logic. When Albert and Bernard talks about what they know and don't know, they implicitly reveals properties of their knowledge. This class of logical puzzle, where reasoning about knowledge and ignorance is required, is called epistemic puzzles.

We designed and implemented a domain specific language called **EpiPEn** (**Epi**stemic **P**rogramming **En**vironment) in which these puzzles can be expressed and solved, using Z3 [2] as an underlying SMT solver. In this paper, we will describe the logical encoding underlying EpiPEn, give some details on the design and implementation of the DSL itself, and present the results we found applying this DSL to six different puzzles.

2 Logical encoding

To encode epistemic logic puzzles formally, we need to encode statements about what agents do or do not know. Traditional logics, such as first-order logic, are not sufficient – they can say that φ is true, but not that agent A knows φ. If we introduce a new operator $K_A\varphi$ meaning "agent A knows φ", we arrive at a new (modal) logic called *epistemic logic* [3]. The other wrinkle, in modelling epistemic-logic puzzles, is that they often involve tracking changes in an agent's knowledge as new information is revealed – usually through public announcement. To model this, we might for instance introduce an operator $[\varphi]\psi$ meaning "after φ is publicly announced, ψ holds". This gives us a particular *dynamic epistemic logic* known as *public announcement logic* [1] [4].

While this formalism is a helpful start, it is not obvious how to make use of it in a world with plenty of first-order SMT solvers but no dynamic-epistemic-logic solvers. Fortunately, we discovered that a significantly limited formalism, relying only on first-order logic, covered nearly all puzzles.

Surveying epistemic-logic puzzles, we observed a common structure. Each puzzle is centered around a set of unknown constants (e.g., the day and month of Cheryl's birthday). Each character is given access to a specific set of constant values. Then there is a series of public announcements, each either by an omniscient storyteller (e.g. Cheryl) or by one of the various characters. These public announcements add to a pool of *common knowledge*, which affects what the characters do and do not subsequently know. Public announcements can, of course, include statements about characters' knowledge at that time – that's what makes these puzzles epistemic.

To roughly formalize this:

A "story" starts with a set $\{A_i\}$ of characters and a set $\{x_i\}$ of unknown constants. As the story proceeds, we will track a changing "knowledge state". This state consists of:

- a map $\mathsf{knows} : \{A_i\} \to \mathcal{P}(\{x_i\})$ stating which constants character A_i has been given direct access to, and
- a common-knowledge formula κ, which all characters know.

Two basic events can occur in a story which change the knowledge state:

- A character A can be given direct access to a constant x (e.g., when Cheryl whispers the month in Albert's ear). This simply results in x being added to the set $\mathsf{knows}(A)$.

- A character A can make a public announcement ψ. This results in the common knowledge κ being replaced with $\kappa \wedge K_A\psi$.

But we said we were working solely in first-order logic. What does $K_A\psi$ mean? In the context of a given knowledge state, we define:

$$K_A\psi := \forall\,(\text{constants not in } \mathsf{knows}(A)) : \kappa \to \psi.$$

Here's a helpful way to understand this condition. An assignment of values to all the constants would usually be called a "model", but we will call it a "possible world". A character's knowledge tells them that not all "possible worlds" are actually possible – only those where A's known constants have the values A knows they do, and κ holds. Call these A's "accessible worlds" – worlds consistent with A's knowledge. Then A knows ψ if ψ holds in all worlds accessible to A. (This is essentially the "partition principle" of epistemic logic.)

Note that $K_A\psi$ has, as free variables, (all or some subset of) the constants in $\mathsf{knows}(A)$. This means that A can evaluate this formula to true or false, to know whether or not they know ψ. This also means that, if A announces $K_A\psi$ (or a formula containing it), some other agent A' can reason about the implications of this announcement without necessarily having direct access to the values of the constants in $\mathsf{knows}(A)$.

Now we know how to update a knowledge state in response to an event in a puzzle's story. Two more notes before we move on:

1. Knowing values, rather than facts

We now know how to say "A knows ψ" in first-order logic. How do we say "A knows the value of an expression e"? Define

$$KV_A e := \exists v : K_A\,(e = v)\,.$$

This correctly captures the meaning of "A knows the value of e" – there's some value v (perhaps unknown to us) for which A knows that the expression equals v.

2. Iterating knowledge operators

We said that, when A makes a public announcement of ψ, κ is augmented by $K_A\psi$, not just ψ. This is important – when a character says something, they don't just reveal the truth of that statement, they also reveal that they could infer that truth from their prior knowledge.

We only needed to add $K_A\psi$ to κ, not $\psi \wedge K_A\psi$, because $K_A\psi \to \psi$ is a theorem for all ψ. (We confirmed this with our SMT solver.) This means our logic satisfies the modal axiom M [5].

But why stop there? Why don't we need to add $K_AK_A\psi$, etc.? Because $K_A\psi \to K_AK_A\psi$ is also a theorem for all ψ. (We also confirmed this with our SMT solver.) This means our logic satisfies the modal axiom 4 (a.k.a. *positive introspection*) [5].

3 DSL design & implementation

We implemented this puzzle formalism and logical encoding as a domain-specific language in Python called **EpiPEn** (**Epi**stemic **P**rogramming **En**vironment).

We knew we wanted to embed our language in a general scripting language, to avoid re-implementing common programming primitives which a user might wish to use in describing puzzles, such as loops. We chose Python due to its popularity and ease of use, and because there are well-supported Python bindings for Z3, our SMT solver.

However, as Python does not support general metaprogramming features (as Racket does), we were restricted to a shallow embedding. It was a challenge to implement our library's API in a way that let users input puzzles concisely and idiomatically, given this constraint.

Here is a complete listing for how the Cheryl's Birthday puzzle can be expressed and solved with EpiPEn:

```python
from epipen import *
from z3 import *

start_story()

Albert, Bernard = Characters('Albert Bernard')

month, day = Ints('month day')

# C gives A and B the possible dates.
announce(storyteller,
    Or([
        And(month == m, day == d)
        for (m, d) in
        [(5, 15), (5, 16), (5, 19), (6, 17),
         (6, 18), (7, 14), (7, 16), (8, 14),
         (8, 15), (8, 17)]
    ])
)
# C whispers to A and B.
learn_constant(Albert, month)
learn_constant(Bernard, day)

# A: I don't know when your birthday is...
announce(Albert, Not(know(value_of=day)))
# A: but I know B doesn't know either.
announce(Albert, Not(know(Bernard, value_of=
    month)))

# B: I didn't know originally, but now I do.
announce(Bernard, know(value_of=month))

# A: Well, now I know too!
announce(Albert, know(value_of=day))

print_possible_worlds()
# > printing up to 10 possible world(s)
# >   [day = 16, month = 7]
# >   (and that's all)
```

We hope the code mostly speaks for itself. A few quick notes of clarification:

1. The two "basic events" that transform the knowledge state are performed here with `learn_constant` and `announce`.

2. On line 25, the function `know` (with `value_of=`) is used to express $KV_{\texttt{Albert}}\texttt{day}$. The `Albert` here is implicitly assumed, since `Albert` is the one making the announcement. (Some hacks were needed to make this possible – an example of the challenges of embedding a DSL shallowly in Python.) On line 27, `Bernard` is provided explicitly, since here `Albert` is speaking about `Bernard`'s knowledge, not his own.

3. To express $K_A\psi$, rather than KV_Ae, a user can use the variant `know(`A`, that=`ψ`)`. (Surprisingly, this version was not used directly in any of the puzzle solutions we constructed.)

3.1 Implementation

Implementation was relatively straightforward.

One tricky aspect was speed. The most important responsibility of our system, performance-wise, is to avoid unnecessarily inflating the size of the common-knowledge formula κ. A large, complex κ will take our SMT solver a long time to solve. Keeping κ small is especially important because every use of `know()` adds an additional copy of the old common-knowledge formula inside the new one. For instance, an announcement of `know(that=`ψ`)` replaces κ with $\kappa \wedge K_A\psi = \kappa \wedge (\forall(\cdots) : \kappa \rightarrow \psi)$. This final expression has two occurrences of κ; we have at least doubled the size of our common-knowledge formula. So a small improvement in the size of κ early on can avoid huge blow-ups later.

Here are two tricks our system uses to try to keep κ small:

1. We make use of Z3's quantifier-elimination and simplification capabilities. In particular, rather than replacing κ with $\kappa \wedge \psi$, we replace it with $\mathsf{simplify}(\kappa \wedge \mathsf{quantifier\text{-}elimination}(\psi))$.

   Quantifier elimination is absolutely essential for Z3 to be able to handle puzzles – even on a puzzle as simple as Cheryl's Birthday, it leads to a 20x speedup. Simplification leads to a more modest but still helpful 2x speedup on more complex puzzles.

2. Our model says that an announcement of `know(that=`ψ`)` should replace κ with $\kappa \wedge K_A\psi$. However, quite often the $K_A\psi$ here could be replaced with just ψ. This will be the case, for instance, when $\psi = K_A\varphi$, since, as mentioned earlier, the "positive introspection" theorem asserts $K_A\varphi \rightarrow K_AK_A\varphi$. It will also be the case when $\psi = \neg K_A\varphi$, since an analogous "negative introspection" theorem asserts $\neg K_A\varphi \rightarrow K_A\neg K_A\varphi$. To cover these cases, and more, we add an extra check. When character A announces ψ, we ask Z3 whether $\psi \rightarrow K_A\psi$ is a tautology. If it is, we only add ψ to κ, not $K_A\psi$.

   This check is usually quite fast: 0.05 seconds or so per announcement. It leads to significant speedups – we've seen more-than-100x improvements for complex puzzles.

4 Results

We investigated some of the more well-known puzzles in this area, and found various degree of success in encoding them in our language and solving them. Here we present our encodings and results.

4.1 Cheryl's Birthday

Our encoding of the Cheryl's Birthday puzzle is used as the central example in the "DSL design & implementation" section above. The encoding is, in our opinion, straightforward, readable, and ergonomic for the encoder.

Note how a list comprehension is used to turn the initial list of possible dates into a logical constraint. This is a nice example of using a feature of the embedding language (Python) to improve expressiveness of puzzle encoding.

We use "`# >`" comments to show lines printed out by code samples. As the output lines at the end of the listing above show, EpiPEn is able to solve the Cheryl's Birthday puzzle, outputting the solution July 16th. This solution takes a bit less than a second on a 2016 Apple MacBook Pro.

4.2 Consecutive Numbers

> Anne and Bill get to hear the following: "Given are two natural numbers. They are consecutive numbers. I am going to whisper one of these numbers to Anne and the other number to Bill." This happens. Anne and Bill now have the following conversation.
>
> > Anne: "I don't know your number."
> >
> > Bill: "I don't know your number."
> >
> > Anne: "I know your number."
> >
> > Bill: "I know your number."
>
> First they don't know the numbers, and then they do. How is that possible? What surely is one of the two numbers? [4]

```
from epipen import *
from z3 import *

start_story()
Anne, Bill = Characters('Anne Bill')
A, B = Ints('A B')

announce(storyteller,
    And(A >= 0, B >= 0,
        Or(A == B + 1, B == A + 1)))
learn_constant(Anne, A)
learn_constant(Bill, B)

# A: 'I don't know your number.'
announce(Anne, Not(know(value_of=B)))
# B: 'I don't know your number.'
```

```
announce(Bill, Not(know(value_of=A)))
# A: 'I know your number.'
announce(Anne, know(value_of=B))
# B: 'I know your number.'
announce(Bill, know(value_of=A))

print_possible_worlds()
# > printing up to 10 possible world(s)
# >    [A = 2, B = 3]
# >    [A = 1, B = 2]
# >    (and that's all)
```

Here, we start with an infinite number of possible worlds, as A and B can be any pair of positive consecutive integers. In fact, the set of possible worlds stays infinite through the first two announcements, as a user can observe by adding additional print_possible_worlds commands earlier in the program. But reasoning about this infinite domain isn't any real trouble for Z3, and EpiPEn is still able to solve for a solution: one of their numbers must certainly be 2.

4.3 Muddy Children

Imagine a large family of n children. Exactly k become muddy while playing outside. A child can see which other children are muddy, but not if he or she is muddy. Father says that at least one of them is muddy, and then says, "In a moment I will clap my hands. If you know whether you are muddy, please step forward." He then repeats this request over and over again, until all children have stepped forward. What will happen? [4, paraphrased]

```
 1 from epipen import *
 2 from z3 import *
 3 
 4 n, k = 6, 3
 5 
 6 start_story()
 7 
 8 children = [Character(f"children[{i}]") for i in range(n)]
 9 is_muddy = [Bool(f"is_muddy[{i}]") for i in range(n)]
10 
11 actual_world = World({is_muddy[i]: i < k for i in range(n)})
12 
13 # Each child sees which others are muddy
14 for i in range(n):
15     for j in range(n):
16         if i != j:
17             learn_constant(children[i], is_muddy[j])
18 
19 # Father: 'At least one of you is muddy'
20 announce(storyteller, Or([is_muddy[i] for i in range(n)]))
21 for step in range(1, 1000):
22     print(f"Step {step}")
23     # Father: 'If you know whether you are muddy, step forward'
24     all_step_forward = True
25     with simultaneous():
26         for i in range(n):
27             is_muddy_formula = know(children[i], value_of=is_muddy[i])
28             is_actually_muddy = actual_world.value_of(is_muddy_formula)
29             if is_actually_muddy:
30                 print(f"child {i} steps forward")
31                 announce(children[i], is_muddy_formula)
32             else:
33                 announce(children[i], Not(is_muddy_formula))
34                 all_step_forward = False
35     if all_step_forward:
36         break
37 
38 # > Step 1
39 # > Step 2
40 # > Step 3
41 # > child 0 steps forward
42 # > child 1 steps forward
43 # > child 2 steps forward
44 # > Step 4
45 # > child 0 steps forward
46 # > child 1 steps forward
47 # > child 2 steps forward
48 # > child 3 steps forward
49 # > child 4 steps forward
50 # > child 5 steps forward
```

This encoding is the most complex yet. After initial set-up, we enter a loop of "steps". In each step, each child considers whether or not they know if they are muddy. They then step forward, or do not step forward, acting as an announcement of this fact.

This requires two features we have not discussed yet:

- This puzzle is a bit different than previous ones. In previous puzzles, we are told what actions took place and are asked to infer the previously-hidden state of the world. In this puzzle, we know the state of the world and are asked to simulate what actions will take place as a result of this state.

To reflect this new structure, on line 11, we construct a `World` object which contains the actual settings of the constants in the puzzle. We use this object on line 28 to determine if a child will actually step forward or not, based on what they see.

- This puzzle also involves simultaneous actions – each child chooses to step forwards or not simultaneously, without taking notice of the other children's actions in that step.

  To express this simultaneity, we use a `simultaneous()` block, starting on line 25. Here's what this means operationally: Entering this block, the common knowledge is frozen. All uses of `know()` inside the block will refer to this frozen common knowledge. All uses of `announce()` inside the block have their effects deferred until the block closes.

4.4 Who Has the Sum?

> Anne, Bill, and Cath all have a positive integer on their forehead. They can only see the foreheads of others. One of the numbers is the sum of the other two. All the previous is common knowledge. They now successively make the truthful announcements:
>
> > Anne: "I don't know my number."
> >
> > Bill: "I don't know my number."
> >
> > Cath: "I don't know my number."
> >
> > Anne: "I know my number. It is 50."
>
> What are the other numbers?

This puzzle is very straightforward to encode. However, when we ran the code, the solver wasn't able to solve it within a reasonable time. When we examined the intermediate states of the puzzle, we discovered that the first announcement:

> Anne: "I don't know my number."

yielded a very complicated Boolean expression about the three integer variables. A human analyst could infer that this announcement should really only contribute a simple formula to the common knowledge: $B \neq C$. (If Bill and Cath's numbers were equal, then Anne would have known that her number must be the sum, as her number can't be zero). Our SMT solver was unable to perform simplifications like this, leading to extremely large formulas that it could not solve efficiently. Ideally, we would find means to automatically simplify formulas constructed by our system. Until then, we decided to add a feature to our DSL to let the user provide hand-written simplification hints. We can still use the solver to verify these hints, ensuring the correctness of the analysis.

```
1  from epipen import *
2  from z3 import *
3
4  start_story()
5  Anne, Bill, Cath =Characters('Anne Bill Cath')
6  A, B, C = Ints('A B C')
7  announce(storyteller, And(
8      A > 0, B > 0, C > 0,
9      Or(A == B + C, B == A + C, C == A + B)
10 ))
11 learn_constants(Anne, [B, C])
12 learn_constants(Bill, [A, C])
13 learn_constants(Cath, [A, B])
14
15 with assert_adds_ck(Not(B == C)):
16   announce(Anne, Not(know(value_of=A)))
17 announce(Bill, Not(know(value_of=B)))
18 announce(Cath, Not(know(value_of=C)))
19 announce(Anne, A == 50)
20
21 print_possible_worlds()
22 # > printing up to 10 possible world(s)
23 # >    [C = 30, A = 50, B = 20]
24 # >    (and that's all)
```

On line 15, a block is introduced with `assert_adds_ck` together with a hint about what the contents of the block are expected to add to the common knowledge ($B \neq C$, in this case). Our system uses our SMT solver to verify that the effect of running the contents of this block is equivalent to the effect of ignoring the contents and just adding the hint onto the common knowledge (with $\wedge$) instead.

With this hand-written simplification, EpiPEn is able to generate the correct solution within a minute – slower than most, but manageable.

4.5 Limitations

Here we discuss two puzzles that EpiPEn is not currently able to solve.

The unexpected hanging paradox

> At a trial a prisoner is sentenced to death by the judge. The verdict reads "You will be executed next week, but the day on which you will be executed will be a surprise to you." The prisoner reasons as follows. "I cannot be executed on Friday, because in that case I would not be surprised. But given that Friday is eliminated, then I cannot be executed on Thursday either, because that would then no longer be a surprise. And so on. Therefore the execution will not take place." And so, his execution, that happened to be on Wednesday, came as a surprise. So, after all, the judge was right. What error does the prisoner make in his reasoning? [4]

The problem here lies with the expressive power of our formalism. The notion of a surprise could seemingly be encoded as $\neg K V_A \psi$. However, here we are dealing with a claim that the prisoner will be surprised, even after being informed that they will be surprised. This means that $\neg K V_A \psi$ must be interpreted relative to a knowledge state that includes $\neg K V_A \psi$. This is a twisty bit of self-reference that EpiPEn is simply unable to model.

Sum and Product

> A says to S and P: I have chosen two integers x, y with $1 < x < y$ and $x + y \leq 100$. In a moment I will inform S of their sum $s = x + y$, and I will inform P of their product $p = xy$. These announcements will remain secret. You are required to make an effort to determine the numbers x and y. He does as announced. The following conversation now takes place:
>
> P says: I don't know the numbers.
>
> S says: I knew you didn't know the numbers.
>
> P says: Now I know the numbers.
>
> S says: Now I also know the numbers. [4]

This one, on the other hand, although easy to encode, proves to be very difficult for the underlying solver. As it involves non-linear operations on the integers, we were not able to solve this puzzle within a reasonable time. It's also not feasible to provide the solver with hand-written simplification, as the reasoning of the puzzle requires us to express that, for example, "p is not a prime". To express this, we need two for-alls over an integer multiplication, which is not something the solver performs very well on.

5 Future work

Future work on EpiPEn should include more performance tuning. This would include careful profiling, to find unnecessarily expensive computations. Significant performance improvements could also come from more sophisticated logical analysis – our current approach uses very little knowledge about the specific properties of $K_A\psi$.

We are also interested in applications. It would be fascinating to see how puzzle designers make use of our system to double-check their work, and perhaps to explore spaces of novel puzzles more creatively. And formal modeling and analysis of knowledge and communication might have applications beyond just puzzles:

- In the linguistic field of pragmatics, the "cooperative principle" asserts that people structure their communications to convey the information desired, with all participants taking into account knowledge held by other participants.
- The field of cryptography is centered around controlling access to secret information, even as communication occurs.

Perhaps the methods we developed here could be applicable to problems in these domains.

6 Logistics

6.1 Teamwork

Both team members worked together to solve puzzles and encode them in our DSL. Josh designed the logical encoding (in terms of unknown constants, knowledge states, and first-order $K_A\psi$). Josh also designed and implemented the DSL. Jack led writing the paper, and investigated providing hand-written simplification to the puzzles.

6.2 Course Topics

In our project, we mainly focused on the application of SMT solvers. We used the Z3 SMT solver as the underlying solver for our DSL in multiple places – both to solve common-knowledge formulas for concrete models, and to check for tautologies along the way.

References

[1] Alexandru Baltag and Bryan Renne. Dynamic Epistemic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2016 edition, 2016.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Rasmus Rendsvig and John Symons. Epistemic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.

[4] Hans van Ditmarsch and Barteld Kooi. *One Hundred Prisoners and a Light Bulb*. Springer International Publishing, 2015.

[5] James Garson. Modal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.