Belief Programming with Map Family Decision Diagrams

Silvio Fossati
silvio.fossati@etu.unige.ch
University of Geneva
Switzerland

Aurélien Coet
aurelien.coet@unige.ch
University of Geneva
Switzerland

Dimitri Racordon*
dimitri.racordon@unige.ch
Northeastern University
United States

Abstract

Software that interacts with a physical environment must operate with a partial and imprecise knowledge, due to the inaccuracy of its sensors. *Belief programming* is a programming methodology that addresses this issue. It provides a framework to reason about the possible states of a system given a set of assumptions—or beliefs—about its variables. This technique is, however, prone to state space explosion, as pointed out by its authors. A straightforward implementation based on exhaustive search will run out of memory when assumptions are not sufficiently tight.

In this paper, we present a new model to overcome this issue. By using Map Family Decision Diagrams to maintain a compact representation of the possible states of the system, our approach scales to a large number of variables and is yet able to perform safety checks efficiently. We developed a belief programming framework based on that model, outperforming the original implementation by several orders of magnitude.

CCS Concepts

• Computing methodologies → Reasoning about belief and knowledge; **• Mathematics of computing → Decision diagrams**.

Keywords

Decision Diagrams; Partial Observability; State Space Explosion

ACM Reference Format:
Silvio Fossati, Aurélien Coet, and Dimitri Racordon. 2022. Belief Programming with Map Family Decision Diagrams. In *Proceedings of ICOOOLPS '22*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software must sometimes operate with only partial and/or imprecise knowledge about the environment. This kind of system has become pervasive, due in part to the advent of automated vehicles and the growing popularity of drones [6]. For example, unmanned aerial vehicles (UAVs) use sensors to control their speed and altitude, identify obstacles, or keep track of their battery level. Unfortunately, the exact state of the system is impossible to know due to imprecision. Consequently, applications must factor sensor errors in, leaving them difficult to test and model check. In effect, because only partial information is available to the system, the entire set of its possible states needs to be represented to reason about it [5].

Belief programming [1] is a methodology that supports the verification of systems with a partially observable state. It provides a framework to maintain a set of assumptions—or beliefs—during the execution of a program. Together, these assumptions make up a belief state that represents a set of possible actual states that can be queried at runtime to drive the system's behavior.

Though promising, the implementation presented in the original paper has a defect: it represents beliefs as an exhaustive enumeration of all possible states, leaving it unable to scale to large numbers of assumptions. As a result, the framework becomes unusable as the size of the model increases, due to state space explosion.

In this paper, we propose to address this issue with Map Family Decision Diagrams (MFDDs) [4]. Decision diagrams [2] are data structures that encode large sets of data into a memory-efficient representation by exploiting the similarities between elements. MFDDs are a particular variant designed to encode families of partial functions, which is well-suited to encode a more optimal and scalable representation of a belief set.

2 Belief Programming

To introduce the notions specific to belief programming, let us reuse the original paper's example of a controller for a UAV that must maintain its altitude at 500 feet [1]. In order to keep the drone at that altitude, the controller needs to take actions relative the measurements it receives from the vehicle's sensors, which are imprecise and hence convey only *partial information* about their real environment. To take such errors into account, the UAV's software must include a *state estimator* to determine all the possible states in which the system could be. Based on these possible states, a decision can be made by the controller to adapt the drone's position if it deviates too much from the target altitude.

In safety critical systems such as UAVs, one might further want to formally verify that the controller is guaranteed to never cause a crash. To do so, the common approach is to specify an *environment model* relating the system's measurements with their true values, to implement the system's state estimator and to then verify that its estimates are correct relative to the specified model.

In belief programming, the developer of a partially observable system can directly include a specification of the environment model in their program. In exchange, a state estimator is automatically provided by the runtime. With the help of a new logic called *Epistemic Hoare Logic* (EHL), the programmer can further verify that the actions taken in the program relative to the state estimator effectively meet the system's requirements. The methodology

*This author is also affiliated with the University of Geneva

ICOOOLPS '22, June 7, 2022, Online
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

is formalised in a new programming language called BLIMP, for BeLief IMPerative. To illustrate the capabilities of the language, we reintroduce in figure 1 the program that implements the drone controller example from [1].

```
1  alt = 500; cmd = 0; t_max = 10000; t = 0;
2  while (t < t_max)
3  { □(450 <= alt && alt <= 550) }
4  {
5      alt = choose(alt - 25 <= . && . <= alt + 25);
6      obs = choose(alt - 25 <= . && . <= alt + 25);
7      observe obs;
8
9      infer ◇(alt < 450) { cmd = 50 }
10     else infer ◇(alt > 550) { cmd = -50 }
11     else { cmd = 0 };
12
13     alt = alt + cmd;
14     t = t + 1
15 }
```

Figure 1: BLIMP program for a UAV controller [1]

In the program, the initial altitude of the drone is set to 500 and a loop is then executed a number of times. Attached to the loop, on line 3, is an invariant which indicates that the altitude of the drone must always remain between 450 and 550 feet. This invariant is expressed with the modal operator $\Box p$, which denotes that a property p must always be true.

On lines 5 and 6, the `choose(p)` statement is used to assign sets of possible values to variables `alt` and `obs`. The statement indicates that these variables might non-deterministically take any value that satisfies some predicate `p`. To keep track of all the possible states that partially observed variables might be in, the runtime maintains a *belief state* relating variables to their set of possible values. In our example, during the first iteration of the loop, after the execution of lines 5 and 6, the program's belief state includes all the states in which `alt` $\in [475, 525]$ and `obs` $\in [450, 550]$, with the additional constraint that `obs` must be within 25 feet of `alt`. This means, for instance, that the situation where `alt = 475` and `obs = 525` is not included in the belief state.

On line 7, the `observe` statement is used to assign the value of an actual measurement to `obs`, thus restricting the set of possible states for that variable to a single number and updating the belief state to only include values for `alt` that are consistent with that observation (that is, values within 25 feet of `obs`).

The `infer` statement of lines 9-11 evaluates conditions expressed with modal operators and branches according to their truth values. Modal operators include the $\Box p$ operator introduced previously, as well as a $\Diamond p$, which denotes that there exists an environment in the belief state such that p is true. In the example, if there exists a state where the altitude of the UAV is smaller than 450, then the first branch is selected, and the controller moves the vehicle up by 50 feet. Alternatively, if there is a state where the altitude is greater than 550, the drone is moved down by 50 feet. In all other cases, the drone is maintained at the same altitude.

A straightforward implementation of BLIMP, called CBLIMP, is presented in [1]. To maintain the program's belief state at runtime, CBLIMP simply enumerates all the possible values of partially observable variables in a set. While this approach works for small examples, its authors remark that it does not scale well with the number of variables in a program and that it quickly becomes too slow or runs out of memory when that number grows. In the following sections, we describe a more efficient encoding for belief states and show how our solution better scales to programs with large state spaces.

3 MFDDs

Map Family Decision Diagrams (MFDDs) [4] are a kind of decision diagrams [2] designed to efficiently encode and manipulate families of partial functions. To illustrate how they work, let us consider the following example. Imagine we wanted to encode a family F of partial functions with domain $\{x, y, z\}$ and co-domain $\mathbb{N}$, such that $F = \{[x \mapsto 1, y \mapsto 2, z \mapsto 1], [x \mapsto 1, y \mapsto 4, z \mapsto 10], [y \mapsto 3, z \mapsto 10]\}$ (where the notation $[x_1 \mapsto a_1, ..., x_n \mapsto a_n]$ represents a partial function mapping x_1 to a_1, x_2 to a_2, and so on). A naive approach could be to simply represent F as a set in which each partial function is represented by a dictionary. However, observing the structure of F, we notice that some of the mappings in the family's partial functions are the same, and that the naive encoding would therefore include redundant information (the mappings from x to 1 and z to 10 would be repeated). Furthermore, every time an operation would need to be performed on the functions in F, each dictionary would have to be processed separately and equivalent mappings would therefore need to be revisited every time.

In MFDDs, families of partial functions are represented as directed acyclic graphs in which the elements of the domain, called *keys*, correspond to nodes and elements of the co-domain, or *values*, are represented by labels on arcs. An arc labeled with a and leaving node x means that the key x is mapped to value a in some partial function represented in the MFDD. When a key doesn't belong to a partial function, a special *skip* arc (represented by a dashed line) is used to indicate that it doesn't take any value. At the end of every path in the MFDD is one of two possible nodes, $\top$ and $\bot$, representing the accepting and rejecting terminals, respectively. Each path starting from the root and ending with the accepting terminal corresponds to a partial function that is included in the family represented by the graph. Any path ending with $\bot$ is excluded. With this approach, mappings from keys to values that are shared between partial functions are also shared in the family's encoding and redundant information is thus avoided.

Figure 2 illustrates the MFDD for F. For simplicity, only paths ending with $\top$ are represented. The three different paths starting from node x and ending with $\top$ represent the three partial functions included in F. Notice how the mappings from x to 1 and z to 10 only appear once in the graph, but are each included in two of the paths and hence in two of the represented functions.

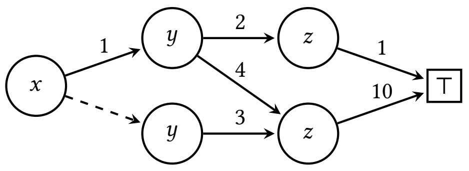

Figure 2: Representation of the MFDD for F

In addition to saving space, MFDDs also have the advantage that operations defined on them can be applied on all members of the family they represent simultaneously, rather than separately. Imagine for example that we wanted to filter out every function in F that maps the key z to the value 10. By simply making the arc labelled with 10 and leaving from z point to $\perp$, both $[x \mapsto 1, y \mapsto 4, z \mapsto 10]$ and $[y \mapsto 3, z \mapsto 10]$ could be excluded from F at the same time.

Operations like the filter described above are called *homomorphisms*, as they preserve the MFDD's structure. Formally, if m_1 and m_2 are MFDDs, $\cdot$ is a binary operation on them and Φ is a homomorphism, then $\Phi(m_1 \cdot m_2) = \Phi(m_1) \cdot \Phi(m_2)$. In practice, this means that homomorphisms can easily be composed, thus enabling efficient sequences of computations on MFDDs.

4 Proposed Approach

In order to reduce the memory footprint and the execution times of BLIMP's runtime, we have experimented with the encoding of belief states into MFDDs. While the translation of a program's possible states to an MFDD is fairly straightforward (the set of possible states is simply a family of partial functions mapping variables to their values), we proceed below to describe how the different statements of a BLIMP program can be translated into homomorphisms:

- `choose`: Choose statements update the belief state by adding new sets of possible values to partially observable variables. They simply correspond to the insertion of new arcs into an MFDD.
- `observe`: The `observe` statement restricts the set of possible mappings of a variable to a single, random value. This translates into two operations on the MFDD. First, a filter is applied on the key for the observed variable, to remove all states in which the mapping for the variable is different from the observed value. Second, filters are applied on the keys of all the other variables that are involved in constraints related to the observed value, so that they remain consistent with it.
- `infer`: `infer` statements check whether some condition on the belief state is true to make a decision. When the belief state is encoded as an MFDD, this amounts to querying the structure to determine if it contains mappings that satisfy the given condition. This can be done by applying a filter on the MFDD and checking if the result is empty.

5 Experimental Results

To assess the efficiency of our proposed approach, we have implemented the translation of the drone example from section 2 into a program using MFDDs, with the help of the DDKit Swift library [3]. In the following, we call this version of the program SBLIMPDD, for Swift BLIMP Decision Diagrams.

To determine the efficiency of our approach, we performed a series of benchmarks measuring the execution times of both SBLIMPDD and the original program implemented with CBLIMP. We experimented with the inclusion of two more variables representing the latitude and longitude of the UAV in the models and played with the ranges of values that the variables could take to assess the impact of these variations on the latency of the runtime in the two approaches.

The results of our experiments are presented in Table 1. All our measurements were performed on a machine with an Intel Xeon E5-2667 v4 processor @ 3.20GHz and 1TB of RAM. We computed the average, standard deviation and maximum time in seconds for multiple executions of the program with a fixed number of repetitions of the model's loop, both for the SBLIMPDD and CBLIMP implementations. Five variations of the UAV model were considered for the benchmarks:

1. The first one corresponds to the situation presented in the example of section 2, where only the altitude is considered with variations of ±25 feet in each execution of the loop.
2. In the second one, the longitude of the drone is added to the set of partially observable variables in the program, with variations of ±10.
3. The third one makes the longitude vary by 25 instead of 10.
4. In the fourth one, the latitude is further added to the set of partially observable variables, again with a variation of ±25.
5. Finally, in the fifth one, only the altitude and the longitude are considered, but they both start at an arbitrary value of 1000 and vary by ±500.

For each case, the table includes the total number of possible program states. In the last 2 variations of the model, the representation of the belief state became too large to compute statistics for CBLIMP, which explains the absence of results in the table.

Model	Stats [s]	CBLIMP	SBLMIPDD
alt: 500±25	Avg	0.001057	0.005216
51 states	Std. dev.	0.000155	0.001884
5 runs, 100 loop executions	Max	0.001995	0.010197
alt: 500±25, lon: 500±10	Avg	2.758465	0.022209
1,071 states	Std. dev.	0.533704	0.024409
5 runs, 100 loop executions	Max	3.884256	0.277797
alt: 500±25, lon: 500±25	Avg	18.403293	0.030730
2,601 states	Std. dev.	3.279009	0.018129
5 runs, 100 loop executions	Max	22.092539	0.199773
alt: 500±25, lat: 500±25, lon: 500±25	Avg	-	0.295050
132,651 states	Std. dev.	-	0.085851
5 runs, 100 loop executions	Max	-	0.926899
alt: 1,000±500, lon: 1,000±500	Avg	-	6.675489
1,002,001 states	Std. dev.	-	1.646441
3 runs, 20 loop executions	Max	-	9.557976

Table 1: Execution times of CBLIMP and SBLIMPDD on the UAV example

Figure 3 represents, with two curves in log-log scale, the average execution times relative to the total number of programs states for both SBLIMPDD and CBLIMP. We observe that for any model with more than a 100 different possible states, SBLIMPDD significantly outperforms CBLIMP.

In addition to execution times, we also measured the memory consumption of SBLIMPDD on 5 different variations of the UAV model. These variations were chosen to measure SBLIMPDD's robustness on examples of programs with very large belief states. The results, which include both the average execution times in seconds and the memory consumption of the program in KB, are reported in Table 2.

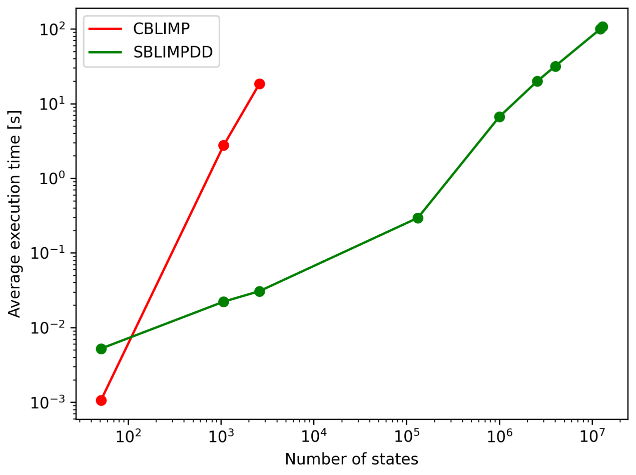

Figure 3: Average execution time (in seconds) relative to the total number of states in the program

Model	Stats	CBLIMP	SBLMIPDD
alt: 500±25 51 states 5 runs, 100 loop exections	Avg time [s]	0.001057	0.005216
	Memory [KB]	4,128	108,292
alt: 500±25, lon: 500±10 1,071 states 5 runs, 100 loop exections	Avg time [s]	2.758465	0.022209
	Memory [KB]	2,216,748	113,344
alt: 500±25, lon: 500±25 2,601 states 5 runs, 100 loop exections	Avg time [s]	18.403293	0.030730
	Memory [KB]	9,926,956	195,728
alt: 500±25, lat: 500±25, lon: 500±25 132,651 states 5 runs, 100 loop executions	Avg time [s]	-	0.295050
	Memory [KB]	-	413,388
alt: 1,000±500, lon: 1,000±500 1,002,001 states 5 runs, 100 loop executions	Avg time [s]	-	7.910374
	Memory [KB]	-	73,128,172
alt: 10,000±1,000, lon: 10,000±1000 4,004,001 states 5 runs, 100 loop executions	Avg time [s]	-	31.761087
	Memory [KB]	-	290,844,560
alt: 10,000±1,800, lon: 10,000±1,800 12,967,201 states 5 runs, 100 loop executions	Avg time [s]	-	107.006212
	Memory [KB]	-	946,584,524

Table 2: Average execution time and memory consumption of CBLIMP and SBLIMPDD on the UAV example

Figure 4 reports the memory consumption (in KB) of CBLIMP and SBLIMPDD relative to the number of states in the UAV example on a log-log scale. Focused on SBLIMPDD, we observe that beyond 10^{5} states, the curve begins to grow rapidly. This phenomenon might be explained by a lack of sharing between the representations for states beyond a certain point: as the state space grows, belief states might become more heterogenous, which could lead to more redundancies in the MFDD. Still, our approach manages to represent the belief states of programs with state spaces significantly larger than CBLIMP.

All of the results presented above confirm our initial intuition that the use of MFDDs to represent the belief states of partially observable systems are an efficient solution to the problem of state

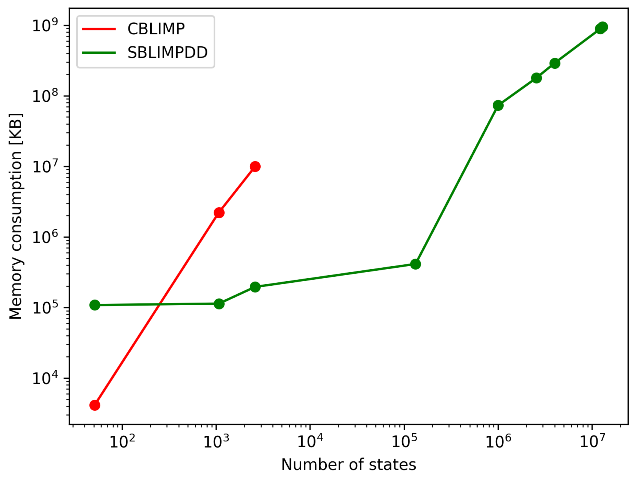

Figure 4: Memory consumption relative to the number of states in the UAV example

space explosion. Thanks to homomorphisms, MFDDs not only reduce the memory consumption of BLIMP's runtime, but also its latency on large problems.

6 Conclusion

In this paper, we proposed an alternative representation of belief states with MFDDs in the implementation of BLIMP's runtime. We described how BLIMP specific statements could be translated into efficient operations called homomorphisms, which could then be directly applied on MFDDs. Finally, we computed a series of benchmarks to validate our approach and showed that it did provide a more efficient implementation of BLIMP than the naive method.

While the results of our experiments are promising, they focus only on a specific example. Further work should therefore seek to generalise our proposed approach to make it automatically applicable to any BLIMP program, by providing a compiler from the language to our MFDD representation.

References

[1] Eric Atkinson and Michael Carbin. 2020. Programming and Reasoning with Partial Observability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 200 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428268

[2] Alban Linard, Emmanuel Paviot-adet, F. Kordon, Didier Buchs, and Samuel Charron. 2010. polyDD: Towards a Framework Generalizing Decision Diagrams. *Proceedings - International Conference on Application of Concurrency to System Design, ACSD*, 124–133. https://doi.org/10.1109/ACSD.2010.17

[3] Dimitri Racordon. 2020. DDKit. https://github.com/kyouko-taiga/DDKit.

[4] Dimitri Racordon, Aurélien Coet, Emmanouela Stachtiari, and Didier Buchs. 2020. Solving Schedulability as a Search Space Problem with Decision Diagrams. In *Search-Based Software Engineering*, Aldeida Aleti and Annibale Panichella (Eds.). Springer International Publishing, Cham, 73–87.

[5] Stuart J. Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach (4th Edition).* Pearson. http://aima.cs.berkeley.edu/

[6] Hazim Shakhatreh, Ahmad Sawalmeh, Ala Al-Fuqaha, Zuochao Dou, Eyad Almaita, Issa Khalil, Noor Othman, Abdallah Khreishah, and Mohsen Guizani. 2018. Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges. *IEEE Access* 7. https://doi.org/10.1109/ACCESS.2019.2909530