Programming and Reasoning with Partial Observability

ERIC ATKINSON, Massachusetts Institute of Technology, USA
MICHAEL CARBIN, Massachusetts Institute of Technology, USA

Computer programs are increasingly being deployed in partially-observable environments. A partially observable environment is an environment whose state is not completely visible to the program, but from which the program receives partial observations. Developers typically deal with partial observability by writing a state estimator that, given observations, attempts to deduce the hidden state of the environment. In safety-critical domains, to formally verify safety properties developers may write an environment model. The model captures the relationship between observations and hidden states and is used to prove the software correct.

In this paper, we present a new methodology for writing and verifying programs in partially observable environments. We present *belief programming*, a programming methodology where developers write an environment model that the program runtime automatically uses to perform state estimation. A belief program dynamically updates and queries a belief state that captures the possible states the environment could be in. To enable verification, we present *Epistemic Hoare Logic* that reasons about the possible belief states of a belief program the same way that classical Hoare logic reasons about the possible states of a program. We develop these concepts by defining a semantics and a program logic for a simple core language called BLIMP. In a case study, we show how belief programming could be used to write and verify a controller for the Mars Polar Lander in BLIMP. We present an implementation of BLIMP called CBLIMP and evaluate it to determine the feasibility of belief programming.

CCS Concepts: • **Software and its engineering** → **General programming languages**; *Semantics*; • **Theory of computation** → *Operational semantics*; **Pre- and post-conditions**; **Invariants**; *Modal and temporal logics*; **Hoare logic**; **Program reasoning**; • **Computing methodologies** → **Reasoning about belief and knowledge**.

Additional Key Words and Phrases: programming languages, logic, partial observability, uncertainty

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

1 INTRODUCTION

Computer systems are increasingly deployed in *partially observable* environments in which the system cannot exactly determine the environment's true state [Russel and Norvig 2020; Smallwood and Sondik 1973]. For example, the software that controls an uncrewed aerial vehicle (UAV) cannot exactly determine the vehicle's true altitude above the ground. Instead, the vehicle's software receives a measurement from a GPS altimeter that estimates the vehicle's altitude.

This measurement or *observation* reveals only partial information about the environment's true state, such as that the UAV's true altitude is within 25 feet of the reported measurement. The primary challenge for a system deployed in such an environment is therefore that it must leverage

Authors' addresses: Eric Atkinson, Massachusetts Institute of Technology, USA, eatkinson@csail.mit.edu; Michael Carbin, Massachusetts Institute of Technology, USA, mcarbin@csail.mit.edu.

2475-1421/2020/11-ART200
https://doi.org/10.1145/3428268

the partial information provided by an observation to meet its goals, which, in contrast, are typically expressed in terms of the environment's true state.

Consider, for example, a UAV tasked with avoiding a collision with the ground. Its controller software will include a *state estimator* that must infer if it is possible that the vehicle may soon have an altitude of 0 given only estimated altitude measurements. If the vehicle is indeed at risk, then the controller must take action to ensure the vehicle's altitude stays strictly positive. However, the discrepancy between the measurements and the vehicle's true altitude introduces the risk that the state estimator's inferences may indicate a strictly positive altitude when the true altitude is in fact 0. Moreover, the controller's reasoning must soundly work with the state estimator's inferences to intervene whenever the true altitude is dangerously near 0.

In safety-critical domains that desire formal guarantees, such as robotics and vehicle navigation, the best-available approach to formally verify the system is to 1) formally specify an *environment model*: the specific relationship between an observation and the system's true state; 2) implement the state estimator and verify the correctness of its inferences in relation to the environment model; and 3) implement the remainder of the controller and verify that the composition of the environment model and controller meets the system's requirements.

1.1 Belief Programming

In this paper, we present *belief programming*, a programming methodology in which the developer writes a program for the controller that includes a specification of the environment model. From that specification, the program runtime automatically provides a state estimator, eliminating the need to manually implement the state estimator and verify its behavior against the environment model.

To instantiate the concepts of this programming methodology, we present a new language, BeLief IMP (BLIMP), a variant of the pedagogical language IMP [Winskel 1993]. BLIMP provides first-class abstractions for environment modeling, for observations, and to interface with the automatically generated state estimator.

Environment Model. The belief programming methodology extends IMP with a nondeterministic choice statement, `x = choose(`p`)`, that nondeterministically updates the program variable `x` to a value that satisfies the predicate p. Unlike a traditional `choose` statement [Back 1978], the value of `x` is not immediately observable to the controller. Instead the semantics of `x` is the set of all possible values that satisfy p. The programming methodology permits such nondeterministic values to be composed with additional computation to produce a jointly nondeterministic and unobserved set of program variables whose potential values correspond to the partially observable values of the system's physical state.

Observations. To reveal the true value of an unobserved program variable, the controller must explicitly perform an observation. Belief programming extends IMP with an observation statement, `observe y`, that makes the value of an unobserved variable `y` visible to the program. If, for example, `y` is a measurement that is derived from another unobservable value `x`, such as an altitude measurement derived from the UAV's true altitude, then the true value of `y` reveals partial information about the true value of `x`.

State Estimation. The belief programming runtime system dynamically maintains a *belief state* that captures the set of all possible values of all unobserved variables. The belief programming methodology also extends IMP with an inference statement, `infer` $p_{\square}$, that computes a boolean inference over the program's belief state and enables the controller to guide its actions given the validity of a proposition. For example, if $\Diamond$`(altitude < 1)` is true, which states that is possible that the UAV's altitude is less than 1 foot, then the controller can intervene to avoid a collision.

Together our abstractions for environment modeling, observations, and state estimation enable belief programming to provide runtime capabilities for state estimation that eliminate the need to implement and verify the state estimator itself.

1.2 Epistemic Hoare Logic

While the belief programming methodology automates the construction of a sound state estimator, a developer must still verify that the contoller's actions and state estimator soundly work together to meet the system's requirements. To address this problem, we present the *Epistemic Hoare Logic* (EHL), a variant of Hoare Logic that supports modal propositions in its assertion logic that model a belief program's dynamically tracked belief state.

EHL includes the modal propositions $\Diamond p$, "it is possible that p is true", and $\Box p$, "it is always the case that p is true", that quantify p over the set of all possible values of the program's variables as captured by its belief state. These propositions, along with EHL's inference rules, enable a developer to represent the state estimator's inferences as propositions in the logic – e.g. `◇(altitude < 1)` meaning "it is possible that the true altitude is less than 1 foot" – and also specify and verify the system's requirements – e.g. `□(altitude >= 1)` meaning "it is always the case that the true altitude is at least 1 foot."

1.3 Contributions

In this paper, we present the following contributions:

- *Belief Programming.* We introduce belief programming, a programming methodology that makes it possible for the program runtime to automatically provide a state estimator given an environment model specification. Specifically, the program runtime tracks the program's belief state: all possible values of the program's unobservable state.
- *Language.* We present the syntax and semantics of Belief IMP (BLIMP), a language designed for belief programming. We establish basic properties of BLIMP semantics that should be true of any belief programming language. Namely, we show the state estimator that a BLIMP program provides soundly and precisely captures the environment's true state.
- *Epistemic Hoare Logic.* We present Epistemic Hoare Logic for verifying properties of BLIMP programs. We show that our logic is sound with respect to BLIMP's semantics.
- *Case Study.* We present a case study showing how belief programming can be used to develop a verified implementation of the Mars Polar Lander's flight control software. The Mars Polar Lander is a lost space probe, hypothesized to have crashed into the surface of Mars during descent due to a control software error [JPL Special Review Board 2000]. We present a controller implemented with belief programming, and formally prove using EHL that it does not have the error that caused the MPL crash.
- *Implementation.* We evaluate the feasibility of belief programming by presenting an implementation of BLIMP in C called CBLIMP. Our results show that belief programming is feasible for problems in robotics and vehicle navigation domains.

The dual contributions of belief programming and Epistemic Hoare Logic enable developers to more easily program in partially observable environments where correctness is paramount. Such developers must currently hand-write an environment model and a state estimator, and belief programming enables them to omit the state estimator. Epistemic Hoare Logic enables developers to reason about the correctness of the resulting belief program, just as Hoare logic allows them to reason about the correctness classical hand-written control software.

2 EXAMPLE

In this section we show how a developer uses belief programming to implement a controller for a UAV. The controller's objective is to maintain the UAV's altitude at 500 feet above the ground. While the UAV can precisely control its altitude, it has to contend with measurement error from its altitude sensing equipment and wind gusts that can blow it off course.

```
1  cmd = 0
2  t_max = 10000; t = 0;
3  while (t < t_max)
4  {
5      input obs;
6      // Controller loop start
7      if (obs < 475) { cmd = 50 }
8      else if (obs > 525) { cmd = -50 }
9      else { cmd = 0 }
10     // Controller loop end
11     t = t + 1
12 }
```

Fig. 1. An altitude controller for a UAV.

The listing in Figure 1 shows how a developer can write such a controller in a traditional programming language. At every time step, up to a maximum of `10000` steps, the controller receives an altitude observation `obs` (Line 5). We assume the observation comes from a sensor which has some inherent measurement error, so that the value stored in `obs` is not precisely the UAV's true altitude. If the observation is sufficiently low, the controller issues a command to climb by 50 feet (Line 7). Conversely, if the observation is sufficiently high, the controller issues a command to descend (Line 8). The conditions on Lines 7 and 8 form a coarse-grained state estimator that determines if the UAV is too high, too low, or at an acceptable altitude. We assume the command is stored in `cmd`, and that there is an external process that reads `cmd` and modifies the UAV's altitude by exactly `cmd`.

```
1  alt = 500; cmd = 0;
2  t_max = 10000; t = 0;
3  while (t < t_max)
4  { 450 <= alt && alt <= 550 }
5  {
6      alt = choose(alt - 25 <= . &&
7          . <= alt + 25);
8      obs = choose(alt - 25 <= . &&
9          . <= alt + 25);
10     // Controller loop start
11     ...
12     // Controller loop end
13     alt = alt + cmd
14     t = t + 1;
15 }
```

Fig. 2. Environment model for the UAV altitude controller.

The developer needs to ensure the UAV maintains a consistent altitude for safety reasons. We will assume that the developer wants to provide this assurance using formal verification, which requires an environment model of how the true and observed altitude are related. The developer can write such a model as a program that specifies the set of environments the UAV may be in, including the set of values the UAV's true and observed altitude may take on.

The listing in Figure 2 shows how a developer can write such a model. The model is composed with the controller by inlining the annotated lines in Figure 1 into Line 11 of Figure 2. Note that this replaces the input `obs` in Figure 1 with the value of `obs` specified by the model, and we have added a *loop invariant* on Line 4.

The modeling language includes a nondeterministic assignment operator `choose`, which takes a predicate and nondeterministically

```
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 }
```

Fig. 3. Implementation of the UAV controller using belief programming in BLIMP.

chooses a value that satisfies that predicate. The placeholder `.` stands for the new value of the variable, so that `x = choose(x - 1 <= . && . <= x + 1)` picks a new value for `x` that is within a distance of 1 of its previous value.

At every time step, the model chooses the current altitude `alt` from a value within a distance of 25 of the previous altitude (Line 6). This models a wind gust causing a change of up to 25 feet of altitude per step. The model then chooses the observation `obs` from within a distance of 25 of the true altitude (Line 8). This models the altitude instrumentation as having a measurement error of up to 25 feet. After the controller runs, the model alters the altitude by adding to it the resulting command `cmd` (Line 13).

The condition the developer needs to ensure is given by the loop invariant on Line 4: `450 <= alt && alt <= 550`. This means that the UAV maintains its target altitude of 500 feet within an error margin of 50 feet. The developer can prove that the composition of the environment model and the controller satisfy this condition using classical verification techniques.

2.1 Belief Programming

We will now explain, alternatively, how the developer implements this program using belief programming. The listing in Figure 3 shows the code to implement the controller in our belief programming language BLIMP. As this program executes, it maintains a *belief state* of the set of possible environments it could be in. Instead of the conditions over concrete observations in Figure 1, the code in Figure 3 uses conditions over belief states to determine control actions.

To explain how belief programming operators work, we will walk through the execution of the first iteration of the while loop in Figure 3. At the start of the loop, the belief state contains a single environment that has `alt` $= 500$, `cmd` $= 0$, `t_max` $= 10000$, and `t` $= 0$.

Choose Statements. The `choose` statements on Lines 5 and 6 expand the belief state to include all possible environments the nondeterminism could generate. After the first assignment to `alt` on Line 5, the belief state contains all environments such that `alt` $\in [475, 525]$. After the assignment to `obs` on Line 6, the belief state contains all environments such that `obs` $\in [450, 550]$, with the

additional constraint that the distance between `alt` and `obs` is less than or equal to 25. This means that, for example, the belief state does not contain the environment where `alt` = 500 and `obs` = 550.

Observe Statements. The `observe` statement on Line 7 implicitly receives an input that is the observed altitude `obs`. It updates the belief state to contain only environments that are consistent with that observed altitude. For example, if the program receives the value 525, then `observe` modifies the belief state to only contain environments where `obs` = 525 and, correspondingly, where `alt` $\in [500, 525]$.

Infer Statements. The `infer` statements on Lines 9-11 branch based on belief state conditions. Conditions use the $\Box$ and $\Diamond$ *modal operators* to quantify over environments in the belief state, with $\Box$ meaning "for all environments in the belief state" and $\Diamond$ meaning "there exists an environment in the belief state". The condition $\Diamond$`(alt < 450)` means that there is an environment in the belief state such that `alt` is smaller than 450. Similarly $\Diamond$`(alt > 550)` means that there is an environment such that `alt` is larger than 550. Compared to the state estimator on Lines 7 and 8 of Figure 1, the infer statements provide more intuition for why the controller was constructed this way. If one of these conditions is true, then it is possible for the true environment to be outside of the desired range of 500 ± 50 feet, and immediate action is needed to correct the situation. Since, assuming the example observation of 525, our belief state contains only the environments where `alt` $\in [500, 525]$, neither of these conditions is true. Thus, the `cmd = 0` branch of the infer statements is executed. This constrains every environment in the belief state to include `cmd = 0`. Note that under the assumption that the observation localizes `alt` to within 25 feet, at most one condition of the infer statements will be true in any given belief state.

Assignments. The final line of the loop, Line 13, updates `alt` to be its previous value plus the value of `cmd` in every environment in the belief state. Because the belief state contains the environments such that `cmd` = 0 and `alt` $\in [500, 525]$, after the update the belief state contains all environments such that `alt` $\in [500, 525]$. The loop invariant on Line 3 states that any environment in the belief state must have a value for `alt` in the range $[450, 550]$. Because our belief state constrains `alt` to the smaller range $[500, 525]$, our belief state satisfies the invariant.

2.2 Reasoning with Epistemic Hoare Logic

The previous section describes a single execution of the belief program given concrete observations, but in general developers need to reason about all potential executions given any observations that are allowed under the environment model. In this section, we show how a developer can use Epistemic Hoare Logic to reason about the potential executions of the belief program in Figure 3. We first present a small example that showcases how Epistemic Hoare Logic can be used to reason about a single statement in the program. We then explain at a high level how similar reasoning can be used to verify the program maintains its altitude within limits, with the details in Appendix[1] C.

2.2.1 Small Example. In this section, we consider proving a simple property about the statement on Line 5 of Figure 3 that specifies altitude updates. We will assume, in accordance with the loop invariant, that the altitude immediately before executing this statement is larger than 450. We will then show that after executing this statement, the altitude is larger than 425. This property is expressed by the following judgment in Epistemic Hoare Logic:

$$D \vdash \{\Box\texttt{(450 <= alt)}\}\ \texttt{alt = choose(alt - 25 <= . \&\& . <= alt + 25)}\ \{\Box\texttt{(425 <= alt)}\}$$

The context D indicates that the judgement applies when the statement is executed under deterministic control flow. The pre-condition, $\Box$`(450 <= alt)`, states that in any environment in the

belief state, the variable `alt` is at least `450`. The post-condition, `□(425 <= alt)`, states that in any environment in the belief state, the variable `alt` is at least `425`.

Epistemic Hoare Logic Rules. To prove this deduction, we first instantiate the Epistemic Hoare Logic rule for `choose` statements that we have designed. We present the general rule in Figure 10 of Section 4. By default, the rule allows for nondeterministic control flow, but we can specialize it to the D context via a subtyping rule. Instantiating the choose rule and subtyping with our original pre-condition yields the following judgement, using the notation s^* for the statement on Line 5:

$$D \vdash \{\texttt{□(450 <= alt)}\}\ s^*\ \{\texttt{□(450 <= a) \&\& □(a - 25 <= alt \&\& alt <= a + 25)}\}$$

The rule produces a post-condition by first renaming all instances of the assigned variable `alt` with a fresh variable `a` in the pre-condition. It then conjuncts this with the `choose` statement's predicate under a `□` with `alt` replaced with `a` and the placeholder `.` replaced with `alt`.

Implications. To rewrite the post-condition into the desired form, we prove that another predicate is implied by the post-condition and use the rule of consequence (formalized in general in Figure 10) to replace the post-condition with the new predicate.

First, we use the general principle that the `□` operator commutes with `&&`. We have formalized this principle as Theorem B.1 in Appendix[1] B. This gives us the following judgment:

$$D \vdash \{\texttt{□(450 <= alt)}\}\ s^*\ \{\texttt{□(450 <= a \&\& a - 25 <= alt \&\& alt <= a + 25)}\}$$

We then use the principle of lifting theorems about environments to theorems about belief states. This states that if we have an implication over environments, we can wrap the premise and conclusion of this implication with `□`, and obtain an implication over belief states. We have formalized this principle as Theorems B.4 and B.5 in Appendix[1] B. In this example, we use the fact that if the environment satisfies `450 <= a && a - 25 <= alt && alt <= a + 25`, then it also satisfies `425 <= alt`. The principle of lifting says that as a result, if the belief state satisfies `□(450 <= a && a - 25 <= alt && alt <= a + 25)`, then it also satisfies `□(425 <= alt)`. Applying the rule of consequence gives us the original judgment we set out to prove:

$$D \vdash \{\texttt{□(450 <= alt)}\}\ s^*\ \{\texttt{□(425 <= alt)}\}$$

2.2.2 Verification. The developer would like to ensure that the loop body in Figure 3 maintains an altitude of 500 feet within a tolerance of 50 feet. This means that the loop body must preserve its invariant. This corresponds to the Epistemic Hoare Logic deduction

$$D \vdash \{\texttt{□(450 <= alt \&\& alt <= 550)}\}\ s\ \{\texttt{□(450 <= alt \&\& alt <= 550)}\}$$

where the pre- and post-conditions are both equal to the loop invariant on Line 3 of Figure 3. The program s is the loop body on Lines 5-14 of Figure 3. Thus, this deduction states that if the loop body starts in a belief state satisfying the loop invariant, it produces a belief state that also satisfies the loop invariant. We consider the case of deterministic control flow, as signified with the D context. This is appropriate because the loop condition, `t < t_max`, is consistently either true or false across all environments in the belief state.

The high-level procedure is to assume the loop invariant as the pre-condition for the loop body. We then use the logic's rules to derive an appropriate post-condition for the loop body, and finally prove that the post condition implies the loop invariant using the reasoning principles in Section B. The details of this process are in Appendix[1] C.

$$
\begin{aligned}
E &::= x \mid c \mid E + E \mid E - E \mid E * E \mid E \,/\, E \mid y \mid \,.\,\\
P &::= b \mid E < E \mid E == E \mid P \;\&\&\; P \mid P \;||\; P \mid \,!P\\
P_\Box &::= \Box P \mid \Diamond P \mid P_\Box \;\&\&\; P_\Box \mid P_\Box \;||\; P_\Box \mid \,!P_\Box\\
P_\exists &::= \exists y^*.\; P_\Box
\end{aligned}
$$

Fig. 4. Syntax of expressions and propositions.

3 LANGUAGE: SYNTAX AND SEMANTICS OF BLIMP

In this section, we present the belief programming language BLIMP. We first present BLIMP's syntax and semantics, and then state and prove properties of the semantics. We then present an *execution model*. While the semantics describes both the environment modeling and state estimation behavior of BLIMP programs, the execution model projects out the state estimation operations.

3.1 Syntax

In this section, we present the syntax of BLIMP, which gives the constructs of belief programming.

3.1.1 Expressions and Propositions. Figure 4 gives the syntax of expressions and propositions.

Expressions. We use the notation E to refer to expressions. An expression may be a variable x, a numeric constant c, or formed using one of the binary operators `+`, `-`, `*`, `/`, which have standard interpretations. An expression may also contain a quantified variable y or the placeholder `.`. Quantified variables only make sense when the variable is bound by an outer quantifier, and the placeholder value only makes sense in the context of an enclosing `choose` statement.

Propositions. We use the notation P to refer to propositions. Propositions may be boolean constants $b \in \{\texttt{true}, \texttt{false}\}$ or comparisons between expressions using the comparison operators `<` and `==`. Propositions may also be combined by conjunction, disjunction, and negation through the boolean operators `&&`, `||`, and `!`, respectively.

Modal Propositions. We use the notation $P_\Box$ to refer to modal propositions. Modal propositions are propositions that are modified using the $\Box$ and $\Diamond$ operators. They quantify over environments in the belief state, and are used to query the belief state for state estimation. Modal propositions may be combined by conjunction, disjunction, and negation, using the same syntax as for non-modal propositions. Note that in contrast to many modal logics, BLIMP's modal operators may only be applied once; hence, propositions such as $\Box\Box P$ are not in the language. This is a design decision we have made to succinctly capture properties of the domain. The alternative would be to allow propositions such as $\Box\Box P$, and include a theorem of the form $\Box\Box P \Rightarrow \Box P$.

Existential Propositions. We use the notation $P_\exists$ to refer to existentially quantified modal propositions. An existential proposition is a modal proposition prepended with the $\exists$ symbol and a comma-separated list of quantified variables y^* (which could potentially be empty). Existential propositions form the core propositional language for reasoning with Epistemic Hoare Logic, and appear in BLIMP in assertions and loop invariants.

3.1.2 Statements. Figure 5 presents the syntax of statements. We use the notation S to refer to the set of statements, which is specified by the grammar in Figure 5. A statement may be an assignment, a choose statement, an assertion, an observation, an if statement, an infer statement, a while loop, a composition of two statements, or a skip statement.

$$
\begin{aligned}
S ::= \; & x = E \\
\mid \; & x = \texttt{choose(}\ P\ \texttt{)} \\
\mid \; & \texttt{assert}\ P_{\exists} \\
\mid \; & \texttt{observe}\ x \\
\mid \; & \texttt{if}\ P\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} \\
\mid \; & \texttt{infer}\ P_{\Box}\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} \\
\mid \; & \texttt{while}\ P\ \{\ P_{\exists}\ \}\ \{\ S\ \} \\
\mid \; & S\ ;\ S \\
\mid \; & \texttt{skip}
\end{aligned}
$$

Fig. 5. Syntax of statements

Assignment and Choose. An assignment statement $x = E$ and a choose statement $x =$ `choose(` P `)` both assign to the program variable x. The assignment statement does so using an expression while the choose statement does so using a proposition that contains the `.` placeholder value.

Assertions. The keyword `assert` signifies an assertion statement. An assertion includes an existential proposition specifies a property that must be true at the statement's program point.

Observation. The keyword `observe` signifies an observation statement. An observation includes the program variable, x, to be observed at the statement's program point.

If and Infer Statements. The keywords `if` and `infer` signify an if statement and an infer statement, respectively. Both statements select branches based on a condition. The distinction between an if and an infer statement is that an if statement branches on an ordinary proposition, whereas an infer statement branches on a modal proposition. If statements facilitate conditional environment models, whereas infer statements facilitate state estimation by branching on belief state queries.

While Loops. The keyword `while` signifies a while loop. Such a loop consists of a body, a condition regarding whether to continue or not, and a loop invariant. The loop invariant is an existential proposition that must be true at the start and end of each loop, and may be used to express a safety property that must be true at every iteration of a time-step loop.

Composition and Skip. Statements may be sequentially composed with the `;` operator. The `skip` keyword signifies a skip statement, a null statement that performs no operations.

3.2 Semantics

In this section, we illustrate how belief programming works by presenting the semantics of BLIMP. The semantics precisely specify how BLIMP manipulates belief states with the goal that the belief states always capture all possible behaviors and only capture realizable behaviors.

3.2.1 *Preliminaries.*

Environments. An environment $\sigma \in \Sigma = (X \to C)$ is a finite map from program variable names to their numeric values, where each value belongs to a finite subset C of the integers. We use the notation $\sigma[x \mapsto c]$ to mean the environment σ with the variable x mapped to the value c. An *optional environment* $\mu \in \Sigma \cup \{\cdot\}$ may be either an environment σ or a null value $\cdot$. The use of null optional environments is described in more detail in Section 3.2.5.

Belief States. A belief state $\beta \in \mathcal{P}(\Sigma)$ is an element of the powerset of Σ, i.e. it is a set of environments. The interpretation is that if an environment σ is in β, then the program believes that σ is a possibly true environment.

3.2.2 Expressions. Figure 6 presents the semantics of expressions. Our approach is a big-step operational semantics that states what value an expression computes when evaluated under a given environment. We use the notation $\langle \sigma, e \rangle \Downarrow c$ to mean that the expression e evaluated under the environment σ yields the value c. The meaning of a variable x is the value of the variable in the input

$$\frac{}{\langle \sigma, x \rangle \Downarrow \sigma(x)} \qquad \frac{}{\langle \sigma, c \rangle \Downarrow c} \qquad \frac{\langle \sigma, e_0 \rangle \Downarrow c_0 \quad \langle \sigma, e_1 \rangle \Downarrow c_1}{\langle \sigma, e_0 + e_1 \rangle \Downarrow c_0 + c_1} \qquad \frac{\langle \sigma, e_0 \rangle \Downarrow c_0 \quad \langle \sigma, e_1 \rangle \Downarrow c_1}{\langle \sigma, e_0 - e_1 \rangle \Downarrow c_0 - c_1}$$

$$\frac{\langle \sigma, e_0 \rangle \Downarrow c_0 \qquad \langle \sigma, e_1 \rangle \Downarrow c_1}{\langle \sigma, e_0 * e_1 \rangle \Downarrow c_0 * c_1} \qquad \frac{\langle \sigma, e_0 \rangle \Downarrow c_0 \qquad \langle \sigma, e_1 \rangle \Downarrow c_1}{\langle \sigma, e_0 \ / \ e_1 \rangle \Downarrow c_0 / c_1}$$

Fig. 6. Semantics of expressions. We use the notation $\langle \sigma, e \rangle \Downarrow c$ to mean that the expression e evaluated under the environment σ yields the value c.

$$\begin{aligned}
\sigma \vDash b &\iff b = \texttt{true} \\
\sigma \vDash p_1 \ \texttt{\&\&} \ p_2 &\iff \sigma \vDash p_1 \wedge \sigma \vDash p_2 \\
\sigma \vDash p_1 \ \texttt{||} \ p_2 &\iff \sigma \vDash p_1 \vee \sigma \vDash p_2 \\
\sigma \vDash \ \texttt{!}p &\iff \sigma \nvDash p \\
\sigma \vDash e_1 < e_2 &\iff \text{there exist } c_1, c_2 \text{ s.t. } \langle \sigma, e_1 \rangle \Downarrow c_1 \wedge \langle \sigma, e_2 \rangle \Downarrow c_2 \wedge c_1 < c_2 \\
\sigma \vDash e_1 \texttt{ == } e_2 &\iff \text{there exist } c_1, c_2. \text{ s.t. } \langle \sigma, e_1 \rangle \Downarrow c_1 \wedge \langle \sigma, e_2 \rangle \Downarrow c_2 \wedge c_1 = c_2
\end{aligned}$$

Fig. 7. Semantics of propositions. We use the notation $\sigma \vDash p$ to mean σ, which must be an environment, satisfies p, a proposition.

environment σ. The meaning of a constant c is the value of the constant. The arithmetic operators +, -, *, and / have their standard interpretation. We assume that / denotes integer division.

3.2.3 Propositions. Figure 7 presents the semantics of propositions. The meaning of a proposition is whether or not, for a given environment, the environment satisfies the proposition (i.e. the proposition is true in the environment). We use the notation $\sigma \vDash p$ to mean that the environment σ satisfies the proposition p. The meaning of an equality == or size < comparison is that an environment satisfies the comparison iff evaluating the expressions yields numbers that satisfy the comparison. Note that the existential quantifiers in the definition are trivial because the expression semantics is a total function of the environment. The meaning of each of the operators &&, ||, and ! is its standard interpretation in propositional logic.

$$\begin{aligned}
\beta \vDash \Box p &\iff \text{for every } \sigma \in \beta, \ \sigma \vDash p \\
\beta \vDash \Diamond p &\iff \text{there is some } \sigma \in \beta \text{ s.t. } \sigma \vDash p \\
\\
\beta \vDash \exists.\ p_\Box &\iff \beta \vDash p_\Box \\
\beta \vDash \exists y_0, \hat{y}'.\ p_\Box &\iff \text{there is some } c \text{ s.t. } \beta \vDash \exists \hat{y}'.\ p_\Box[c/y_0]
\end{aligned}$$

Fig. 8. Semantics of modal and existential propositions. We use the notation $\beta \vDash p_\Box$ and $\beta \vDash p_\exists$ to mean that the belief state β satisfies $p_\Box$ or $p_\exists$.

3.2.4 Modal and Existential Propositions. Figure 8 presents the semantics of modal and existential propositions. The meaning of a modal or existential proposition is whether or not a given belief state

satisfies the proposition (i.e. the proposition is true in the belief state). We use the notation $\beta \vDash p_{\square}$ and $\beta \vDash p_{\exists}$ to mean that the belief state β satisfies $p_{\square}$ or $p_{\exists}$. The meaning of $\square$ is to universally quantify over all environments in the belief state, and the meaning of $\diamond$ is to existentially quantify over environments in the belief state. The meaning of the operators &&, ||, and ! (elided in Figure 8) is their standard propositional logic interpretations, and is the same as Figure 7 with σ replaced with β. The meaning of $\exists$ is its standard meaning in first-order logic. We use notation $p_{\square}[c/y_0]$ to mean the proposition $p_{\square}$ with c substituted for y_0, and the notation $\exists.\ p_{\square}$ to mean an existential proposition with an empty set of quantified variables.

3.2.5 Statements. Figure 9 presents the semantics of statements. We follow a big-step operational approach, where every statement updates the belief state as well as an optional true environment. While the program execution only updates the belief state, we model programs as simultaneously nondeterministically updating the true environment to provide an easy specification of the set of legal observation inputs. Namely, a legal observation is one that could have come from the true environment. A null true environment signifies that, due to nondeterministic control flow, the true environment took a different branch than the belief state. Observations under a null true environment are illegal. The nondeterminism of the true environment impacts the belief state through the observations, but for fixed observation inputs, the belief state update is deterministic. Every statement produces, given an initial belief state and true environment, a *configuration*, which is either a new belief state and new optional true environment or an error $\bot$. We augment the result with an observation list, which documents all observations and is an element of the grammar

$$O ::= x : c :: O \mid \texttt{nil}$$

In other words, an observation list is a list of associations of variable names to values. We use the notation $\langle \beta, \mu, s \rangle \Downarrow \langle C \mid o \rangle$ to mean that the belief state β and optional true environment μ produce the configuration C augmented with the observation list o.

Assignment and Choose. The meaning of either an assignment or choose statement is a new environment, if an original environment was present, and belief state with the statement variable rebound to a new value. In the case of assignment, the value is given by evaluating the expression. In the case of a choose statement the value must be consistent with the proposition, meaning that if we replace the placeholder . with the new value, the proposition must hold. In both cases, the new belief state is obtained by applying this process to each environment in the initial belief state. A choose statement is nondeterministic with respect to the true environment, but deterministic with respect to the belief state.

Assertions. The meaning of an assertion, if its predicate is true, is to return the input belief state and environment. If its predicate is false, the assertion returns an error.

Observations. An observation `observe` x does not modify the true environment. It does modify the belief state to be consistent with the true environment on the observed variable x by only keeping those environments in the initial belief state that have the same value for x as in the true environment. The semantics also specify that the value of x is in the observation list, and that an error occurs if the true environment is null.

If Statements. If the if statement's condition is deterministic (i.e., it is either true in all environments in the belief state or false in all environments), then the execution takes the appropriate branch. This is specified by semantic rules that require $\beta \vDash \square p$ or $\beta \vDash \square(!p)$, where β is the initial belief state and p is the if statement's condition.

$$\frac{\beta' = \{\sigma_\beta[x \mapsto c_\beta] \mid \sigma_\beta \in \beta \wedge \langle \sigma_\beta, e\rangle \Downarrow c_\beta\}}{\langle \beta, \cdot, x = e\rangle \Downarrow \langle \beta', \cdot \mid \texttt{nil}\rangle} \qquad \frac{\langle \beta, \cdot, x = e\rangle \Downarrow \langle \beta', \cdot \mid \texttt{nil}\rangle \qquad \langle \sigma, e\rangle \Downarrow c}{\langle \beta, \sigma, x = e\rangle \Downarrow \langle \beta', \sigma[x \mapsto c] \mid \texttt{nil}\rangle}$$

$$\frac{\beta' = \{\sigma_\beta[x \mapsto c_\beta] \mid \sigma_\beta \in \beta \wedge \sigma_\beta \vDash p[c_\beta/.]\}}{\langle \beta, \cdot, x = \texttt{choose}(p)\rangle \Downarrow \langle \beta', \cdot \mid \texttt{nil}\rangle} \qquad \frac{\langle \beta, \cdot, x = \texttt{choose}(p)\rangle \Downarrow \langle \beta', \cdot \mid \texttt{nil}\rangle \quad \sigma \vDash p[c/.]}{\langle \beta, \sigma, x = \texttt{choose}(p)\rangle \Downarrow \langle \beta', \sigma[x \mapsto c] \mid \texttt{nil}\rangle}$$

$$\frac{\beta \vDash p_\square}{\langle \beta, \mu, \texttt{assert } p_\square\rangle \Downarrow \langle \beta, \mu \mid \texttt{nil}\rangle} \qquad \frac{\beta \nvDash p_\square}{\langle \beta, \mu, \texttt{assert } p_\square\rangle \Downarrow \langle \bot \mid \texttt{nil}\rangle}$$

$$\frac{\beta' = \{\sigma_\beta \mid \sigma_\beta \in \beta \wedge \sigma_\beta(x) = \sigma(x)\}}{\langle \beta, \sigma, \texttt{observe } x\rangle \Downarrow \langle \beta', \sigma \mid x : \sigma(x)\rangle} \qquad \frac{}{\langle \beta, \cdot, \texttt{observe } x\rangle \Downarrow \langle \bot \mid \texttt{nil}\rangle}$$

$$\frac{\begin{array}{c}\beta \vDash \Diamond p \;\&\&\; \Diamond(!p) \\ \beta_T = \{\sigma_\beta \mid \sigma_\beta \in \beta \wedge \sigma_\beta \vDash p\} \qquad \beta_F = \{\sigma_\beta \mid \sigma_\beta \in \beta \wedge \sigma_\beta \nvDash p\} \\ \mu_T = \sigma \text{ if } \mu = \sigma \wedge \sigma \vDash p \text{ else } \cdot \qquad \mu_F = \sigma \text{ if } \mu = \sigma \wedge \sigma \nvDash p \text{ else } \cdot \\ \langle \beta_T, \mu_T, s_1\rangle \Downarrow \langle \beta'_T, \mu'_T \mid \texttt{nil}\rangle \qquad \langle \beta_F, \mu_F, s_2\rangle \Downarrow \langle \beta'_F, \mu'_F \mid \texttt{nil}\rangle \\ \mu' = \begin{cases} \mu'_T & \mu = \sigma \wedge \sigma \vDash p \\ \mu'_F & \mu = \sigma \wedge \sigma \nvDash p \\ \cdot & \text{else} \end{cases}\end{array}}{\langle \beta, \mu, \texttt{if } p \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle \beta'_T \cup \beta'_F, \mu' \mid \texttt{nil}\rangle}$$

$$\frac{\begin{array}{c}\beta \vDash \Diamond p \;\&\&\; \Diamond(!p) \\ \beta_T = \{\sigma_\beta \mid \sigma_\beta \in \beta \wedge \sigma_\beta \vDash p\} \qquad \beta_F = \{\sigma_\beta \mid \sigma_\beta \in \beta \wedge \sigma_\beta \nvDash p\} \\ \mu_T = \sigma \text{ if } \mu = \sigma \wedge \sigma \vDash p \text{ else } \cdot \qquad \mu_F = \cdot \text{ if } \mu = \sigma \wedge \sigma \vDash p \text{ else } \sigma \\ \langle \beta_T, \mu_T, s_1\rangle \Downarrow \langle \beta'_T, \mu'_T \mid o_T\rangle \qquad \langle \beta_F, \mu_F, s_2\rangle \Downarrow \langle \beta'_F, \mu'_F \mid o_F\rangle \\ o_T \neq \texttt{nil} \vee o_F \neq \texttt{nil}\end{array}}{\langle \beta, \mu, \texttt{if } p \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle \bot \mid \texttt{nil}\rangle}$$

$$\frac{\beta \vDash \square p \qquad \langle \beta, \mu, s_1\rangle \Downarrow \langle C_1 \mid o\rangle}{\langle \beta, \mu, \texttt{if } p \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle C_1 \mid o\rangle} \qquad \frac{\beta \vDash \square(!p) \qquad \langle \beta, \mu, s_2\rangle \Downarrow \langle C_2 \mid o\rangle}{\langle \beta, \mu, \texttt{if } p \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle C_2 \mid o\rangle}$$

$$\frac{\beta \vDash p_\square \qquad \langle \beta, \mu, s_1\rangle \Downarrow \langle C_1 \mid o\rangle}{\langle \beta, \mu, \texttt{infer } p_\square \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle C_1 \mid o\rangle} \qquad \frac{\beta \nvDash p_\square \qquad \langle \beta, \mu, s_2\rangle \Downarrow \langle C_2 \mid o\rangle}{\langle \beta, \mu, \texttt{infer } p_\square \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}\rangle \Downarrow \langle C_2 \mid o\rangle}$$

$$\frac{\langle \beta, \mu, \texttt{assert } p^I_\exists \texttt{ ; if(}p\texttt{) \{}s \texttt{ ; while } p \texttt{ \{ } p^I_\exists \texttt{ \} \{ } s \texttt{ \}\} else \{ skip \} }\rangle \Downarrow \langle C \mid o\rangle}{\langle \beta, \mu, \texttt{while } p \texttt{ \{ } p^I_\exists \texttt{ \} \{ } s \texttt{ \}}\rangle \Downarrow \langle C \mid o\rangle}$$

$$\frac{\begin{array}{c}\langle \beta, \mu, s_1\rangle \Downarrow \langle \beta', \mu' \mid o_1\rangle \\ \langle \beta', \mu', s_2\rangle \Downarrow \langle \beta'', \mu'' \mid o_2\rangle\end{array}}{\langle \beta, \mu, s_1 \texttt{ ; } s_2\rangle \Downarrow \langle \beta'', \mu'' \mid o_1 +\!\!+ o_2\rangle} \qquad \frac{}{\langle \beta, \mu, \texttt{skip}\rangle \Downarrow \langle \beta, \mu \mid \texttt{nil}\rangle}$$

Fig. 9. Semantics of statements. We use the notation $\langle \beta, \mu, s\rangle \Downarrow \langle C \mid o\rangle$ to mean that the belief state β and optional true environment μ produce the configuration C augmented with the observation list o.

If the statement's condition is nondeterministic (as specified by requiring $\beta \vDash \Diamond p \;\&\&\; \Diamond(!p)$), the if statement executes both branches, sending as belief-state input to each the set of environments in which the condition has the appropriate value. It sends the true environment as input to the branch it actually takes and the null environment to the other branch. The resulting belief state is then the union of environments resulting from either branch, and the resulting true environment is from the branch that the initial true environment actually took.

If an if statement's condition is nondeterministic, then neither of its branches can make observations, or else the result is an error. This is because it is unclear what interaction with the true environment means within a branch that environment did not necessarily take.

Infer Statements. The semantics of infer statements are similar to the semantics of if statements in an ordinary language, where infer operates solely on the belief state. If the belief state satisfies the condition, it evaluates the first branch and otherwise evaluates the second branch.

While Loops. The semantics of while loops is defined recursively using if statements. This is similar to a standard equivalence notion for while-loop programs (see [Winskel 1993] Section 2.5). We additionally include an assertion that requires the loop invariant to be true.

Composition and Skip. The semantics of statement composition and skips are standard, except that they have been appropriately extended to include the observation list. Sequencing concatenates observation lists using the $+\!\!+$ operator.

Errors. We have elided for clarity from Figure 9 the full semantics of how errors propagate. We assume that errors propagate maximally throughout the program, so if at any point the semantics produce $\bot$, the whole program produces $\bot$.

3.3 Semantic Properties

In this section, we establish several properties of the semantics in Figure 9 that we posit any belief programming system should satisfy. These properties constrain the semantics so that the belief state updates respect the true environment updates.

The first property is that beliefs should be sound. This means that for each environment in the belief state, and each new environment and new belief state that are reachable according to the semantics, the new environment is in the new belief state. We formalize this as follows:

Theorem 3.1 (Belief Soundness).

$$\textit{If } \sigma \in \beta \textit{ and } \langle \beta, \sigma, s\rangle \Downarrow \langle \beta', \sigma' \mid o\rangle, \textit{ then } \sigma' \in \beta'.$$

Proof. By structural induction on derivations of $\Downarrow$. Details are in Appendix[1] A.1. □

This means that it is impossible for the true environment to lie outside of the belief state.

In addition, beliefs should be precise. This means that for every environment in a new belief state reachable from an initial belief state and the semantics, there is an environment in the initial belief state from which the new environment is reachable. We formalize this as follows:

Theorem 3.2 (Belief Precision).

$$\textit{If } \langle \beta, \mu, s\rangle \Downarrow \langle \beta', \mu' \mid o\rangle, \textit{ then for every } \sigma'_\beta \in \beta', \textit{ there is some } \sigma_\beta \in \beta \textit{ such that } \langle \beta, \sigma_\beta, s\rangle \Downarrow \langle \beta', \sigma'_\beta \mid o\rangle$$

Proof. By structural induction on derivations of $\Downarrow$. Details are in Appendix[1] A.2. □

This means that every environment in the belief state could be the true environment.

Finally, beliefs should be deterministic given observations. This means that the belief state depends on the true environment only through observations. We formalize this as follows:

Theorem 3.3 (Belief Determinism).

If $\langle \beta, \mu_1, s \rangle \Downarrow \langle \beta_1', \mu_1' \mid o_1 \rangle$ *and* $\langle \beta, \mu_2, s \rangle \Downarrow \langle \beta_2', \mu_2' \mid o_2 \rangle$ *and* $o_1 = o_2$, *then* $\beta_1' = \beta_2'$.

Proof. By structural induction on derivations of $\Downarrow$ using a strengthened induction hypothesis. Details are in Appendix[1] A.3. □

3.4 Execution Model

Here, we discuss how belief state updates execute in absence of the true environment. This models how the belief program executes. According to Theorem 3.3, the semantics in Figure 9 depends on the true environment μ only through the observation list o. By projecting out the operations on belief states, we can use the semantics in Figure 9 to compute the new belief state using only the initial belief state and the sequence of observations. In other words, if we define the belief execution $\Downarrow_\beta$ as follows

$$\frac{\langle \beta, \mu, s \rangle \Downarrow \langle \beta', \mu' \mid o \rangle}{\langle \beta, o, s \rangle \Downarrow_\beta \beta'}$$

then $\Downarrow_\beta$ is a partial function of β, o, and s. The function $\Downarrow_\beta$ gives the semantics of the belief program's execution on a concrete sequence of observations o.

4 EPISTEMIC HOARE LOGIC

In this section, we present Epistemic Hoare Logic and sketch a proof of its soundness. Figure 10 gives the inference rules for Epistemic Hoare Logic. Each rule yields a deduction $PC \vdash \{p_\exists\}\ s\ \{p_\exists'\}$, where the context PC is drawn from the grammar $PC ::= N \mid D$. The purpose of the context is to ensure observations do not occur under nondeterministic control flow, as that would result in an error according to the semantics. The deduction $D \vdash \{p_\exists\}\ s\ \{p_\exists'\}$ meas that, assuming the statement s is executed under deterministic control flow and terminates, s maps belief states satisfying the pre-condition $p_\exists$ to new belief states satisfying the post-condition $p_\exists'$. The deduction $N \vdash \{p_\exists\}\ s\ \{p_\exists'\}$ means the same thing, but where the enclosing control flow may be nondeterministic.

Assignment and Choose. Assignment and choose statements conjunct the pre-condition with a new proposition. In the case of assignment, the new proposition encodes that the value of the variable is equal to the result of the expression. In the case of a choose statement, the new proposition encodes that the value of the variable must be consistent with the choose statement's proposition. In either case the previous value of the variable is encoded with a fresh variable.

Assertions. An assertion has identical pre- and post-condition propositions. In order to apply the rule, the developer must show that the pre-condition implies the asserted proposition.

Observations. An observation adds a new existentially quantified variable y_{n+1} to the variable list from the pre-condition. The post-condition ensures that the value of the observed variable is deterministic by stating that in every environment it is equal to y_{n+1}. Moreover, y_{n+1} is not completely unrestricted; it must satisfy all properties that the observed variable satisfied in the precondition. Observations always require that the enclosing control flow is deterministic.

Composition and Skip. The rules for statement sequencing and skip statements are standard; they are the same as in classical Hoare logic [Floyd 1967; Hoare 1969]

If and Infer. Both if and infer statements require that both branches satisfy the same post-condition. For if, the pre-condition of each branch includes the statement's condition under the □ modality. For infer, the pre-condition of each branch includes the statement's condition, which is itself a modal proposition.

$$\frac{x_0 \text{ fresh in } p_\Box, e \qquad p'_\Box = p_\Box[x_0/x] \qquad e' = e[x_0/x]}{N \vdash \{\exists \hat{y}.\ p_\Box\}\ x = e\ \{\exists \hat{y}.\ p'_\Box\ \&\&\ \Box(x == e')\}} \qquad \frac{N \vdash \{p_\exists\}\ s\ \{p'_\exists\}}{D \vdash \{p_\exists\}\ s\ \{p'_\exists\}}$$

$$\frac{x_0 \text{ fresh in } p_\Box, p \qquad p'_\Box = p_\Box[x_0/x] \qquad p' = p[x_0/x][x/.]}{N \vdash \{\exists \hat{y}.\ p_\Box\}\ x = \texttt{choose}(p)\ \{\exists \hat{y}.\ p'_\Box\ \&\&\ \Box(p')\}} \qquad \frac{\forall \beta.\ \beta \vDash p_\exists \Rightarrow \beta \vDash p^a_\exists}{N \vdash \{p_\exists\}\ \texttt{assert}\ p^a_\exists\ \{p_\exists\}}$$

$$\frac{PC \vdash \{p_\exists\}\ s_1\ \{p'_\exists\} \qquad PC \vdash \{p'_\exists\}\ s_2\ \{p''_\exists\}}{PC \vdash \{p_\exists\}\ s_1\ ;\ s_2\ \{p''_\exists\}} \qquad \frac{}{N \vdash \{p_\exists\}\ \texttt{skip}\ \{p_\exists\}}$$

$$\frac{y_{n+1} \text{ fresh in } p_\Box \qquad p'_\Box = p_\Box[y_{n+1}/x]}{D \vdash \{\exists y_0, y_1, \ldots, y_n.\ p_\Box\}\ \texttt{observe}\ x\ \{\exists y_0, y_1, \ldots, y_n, y_{n+1}.\ p'_\Box\ \&\&\ \Box(x == y_{n+1})\}}$$

$$\frac{N \vdash \{\exists \hat{y}.\ (\Box p)\ \&\&\ p_\Box\}\ s_1\ \{\exists \hat{y}'.\ \Box p'\} \qquad N \vdash \{\exists \hat{y}.\ (\Box !p)\ \&\&\ p_\Box\}\ s_2\ \{\exists \hat{y}'.\ \Box p'\}}{N \vdash \{\exists \hat{y}.\ p_\Box\}\ \texttt{if}\ p\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ \{\exists \hat{y}'.\ \Box p'\}}$$

$$\frac{\forall \beta.\ \beta \vDash \exists \hat{y}.\ p_\Box \Rightarrow \beta \vDash \Box p\ ||\ \Box(!p) \qquad D \vdash \{\exists \hat{y}.\ (\Box p)\ \&\&\ p_\Box\}\ s_1\ \{p'_\exists\} \qquad D \vdash \{\exists \hat{y}.\ (\Box !p)\ \&\&\ p_\Box\}\ s_2\ \{p'_\exists\}}{D \vdash \{\exists \hat{y}.\ p_\Box\}\ \texttt{if}\ p\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ \{p'_\exists\}}$$

$$\frac{PC \vdash \{\exists \hat{y}.\ p^i_\Box\ \&\&\ p_\Box\}\ s_1\ \{p'_\exists\} \qquad PC \vdash \{\exists \hat{y}.\ !(p^i_\Box)\ \&\&\ p_\Box\}\ s_2\ \{p'_\exists\}}{PC \vdash \{\exists \hat{y}.\ p_\Box\}\ \texttt{infer}\ p^i_\Box\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ \{p'_\exists\}}$$

$$\frac{\begin{array}{c} N \vdash \{p'_\exists\}\ s\ \{p''_\exists\} \\ \forall \beta.\ \beta \vDash p_\exists \Rightarrow \beta \vDash \exists \hat{y}.\ \Box p^I \qquad \forall \beta.\ \beta \vDash \exists \hat{y}.\ \Box p^I\ \&\&\ \Box p \Rightarrow \beta \vDash p'_\exists \qquad \forall \beta.\ \beta \vDash p''_\exists \Rightarrow \beta \vDash \exists \hat{y}.\ \Box p^I \end{array}}{N \vdash \{p_\exists\}\ \texttt{while}\ p\ \{\ \exists \hat{y}.\ \Box p^I\ \}\ \{\ s\ \}\ \{\exists \hat{y}.\ \Box(!p\ \&\&\ p^I)\ \}}$$

$$\frac{\begin{array}{c} D \vdash \{p'_\exists\}\ s\ \{p''_\exists\} \\ \forall \beta.\ \beta \vDash \exists \hat{y}.\ p^I_\Box \Rightarrow \beta \vDash \Box p\ ||\ \Box(!p) \qquad \forall \beta.\ \beta \vDash p_\exists \Rightarrow \beta \vDash \exists \hat{y}.\ p^I_\Box \\ \forall \beta.\ \beta \vDash \exists \hat{y}.\ p^I_\Box\ \&\&\ \Box p \Rightarrow \beta \vDash p'_\exists \qquad \forall \beta.\ \beta \vDash p''_\exists \Rightarrow \beta \vDash \exists \hat{y}.\ p^I_\Box \end{array}}{D \vdash \{p_\exists\}\ \texttt{while}\ p\ \{\ \exists \hat{y}.\ p^I_\Box\ \}\ \{\ s\ \}\ \{\exists \hat{y}.\ \Box!(p)\ \&\&\ p^I_\Box\}} \qquad \frac{\begin{array}{c} PC \vdash \{p'_\exists\}\ s\ \{p''_\exists\} \\ \forall \beta.\ \beta \vDash p_\exists \Rightarrow \beta \vDash p'_\exists \\ \forall \beta.\ \beta \vDash p''_\exists \Rightarrow p'''_\exists \end{array}}{PC \vdash \{p_\exists\}\ s\ \{p'''_\exists\}}$$

Fig. 10. Epistemic Hoare Logic rules. We use the notation $PC \vdash \{p_\exists\}\ s\ \{p'_\exists\}$ to mean that in a context PC the statement s maps belief states satisfying $p_\exists$ to new belief states satisfying $p'_\exists$.

Furthermore, the post-condition of an if statement must use the $\Box$ modality, whereas the post-condition of an infer statement may be any existential proposition. A more conventional approach would be to have both branches of the if statement imply the same predicate as the post-condition. For this to be sound, we would need to show that if $\beta_1 \vDash p_\exists$ and $\beta_2 \vDash p_\exists$ then $\beta_1 \cup \beta_2 \vDash p_\exists$, which is in fact false. However, it is true that if $\beta_1 \vDash \Box p$ and $\beta_2 \vDash \Box p$, then $\beta_1 \cup \beta_2 \vDash \Box p$.

To preserve a deterministic context, developers must ensure that the the if statement's condition has the same value in all environments.

While. To verify a while loop with Epistemic Hoare Logic, the developer must prove several properties of the initial pre-condition $p_\exists$, the loop invariant $p^I_\exists$, the loop body's pre-condition $p'_\exists$, and the loop body's post-condition $p''_\exists$.

The developer must show that the pre-condition implies the loop invariant, the invariant implies the body's pre-condition, and the body's post-condition implies the invariant. Implications such as these are a standard feature of proving properties using Hoare rules, although the overall proof rules are sometimes structured differently.

To preserve a deterministic context, developers must also ensure that the the while loop's condition has the same value in all environments.

Rule of Consequence. The rule of consequence that any proposition that implies the pre-condition may be substituted for the pre-condition. Conversely, any proposition implied by the post-condition may be substituted for the post-condition.

4.1 Soundness

In this section, we formalize and establish the soundness of Epistemic Hoare Logic with respect to the semantics of BLIMP. We start by explaining what supporting lemmas are needed, and then state the main theorem and sketch the proof.

4.1.1 Substitution. The main theorem depends on a number of lemmas that relate the substitutions in Figure 10 to the environment mapping in Figure 9.

The following lemma gives the required property for expressions. It states that if we evaluate an expression under a new environment with a fresh variable x_0 that is a rebinding of x, we are free to rename x to x_0 in the expression without changing the result.

Lemma 4.1 (Expression Substitution).

$$\textit{If } x_0 \textit{ is fresh in } e, \textit{ then } \langle \sigma, e\rangle \Downarrow c \iff \langle \sigma[x_0 \mapsto \sigma(x)], e\rangle \Downarrow c \textit{ and}$$
$$\langle \sigma, e\rangle \Downarrow c \iff \langle \sigma[x_0 \mapsto \sigma(x)], e[x_0/x]\rangle \Downarrow c$$

Proof. By structural induction on expressions. □

By similar means, we establish analogous properties for propositions, modal propositions, and existential propositions. The full set of lemmas is in Appendices A.4.1-A.4.3.

4.1.2 Observation List Emptiness. The main theorem depends upon a lemma that states that observations cannot happen under nondeterministic control flow. We have formalized this property as follows:

Lemma 4.2 (Observation List Emptiness).

$$\textit{If } N \vdash \{p_\exists\}\ s\ \{p'_\exists\} \textit{ and } \langle \beta, \mu, s\rangle \Downarrow \langle C \mid o\rangle, \textit{ then } o = \texttt{nil}.$$

Proof. By structural induction on derivations of $\Downarrow$. The case of sequencing uses the fact that $\texttt{nil} +\!\!+ \texttt{nil} = \texttt{nil}$ □

4.1.3 Soundness Theorem. In this section, we establish the soundness of the logic. Our approach is to show partial correctness, meaning that if the program terminates, its final state is described by the post-condition.

The theorem has two parts. First, we establish the soundness of the logic in the case where control flow may be nondeterministic. The theorem states that if the belief state satisfies the pre-condition, and the semantics produces a configuration, the configuration includes a new belief state that satisfies the post-condition. Second, we establish the soundness of the logic under deterministic control flow. The theorem states that if the initial belief state satisfies the pre-condition, and the

true environment is in the belief state, then the program executes without error and the new belief state satisfies the post-condition.

Theorem 4.3 (Logic Soundness).

(1) *If* $N \vdash \{p_\exists\}\ s\ \{p'_\exists\}$, $\beta \vDash p_\exists$, *and* $\langle \beta, \mu, s\rangle \Downarrow \langle C \mid o\rangle$, *then* $C = (\beta', \mu')$ *and* $\beta' \vDash p'_\exists$.
(2) *If* $D \vdash \{p_\exists\}\ s\ \{p'_\exists\}$, $\sigma \in \beta$, $\beta \vDash p_\exists$, *and* $\langle \beta, \sigma, s\rangle \Downarrow \langle C \mid o\rangle$, *then* $C = (\beta', \sigma')$ *and* $\beta' \vDash p'_\exists$.

Proof. (1) By structural induction on derivations of $\Downarrow$. The cases for assignments and choose statements rely on substitution lemmas. The cases for deterministic if, infer, and while follow from induction hypotheses, although while loops require destructing the semantic rule. For non-deterministic if statements, we show that $\beta_1 \vDash \exists \hat{y}.\ \Box p \wedge \beta_2 \vDash \exists \hat{y}.\ \Box p \implies \beta_1 \cup \beta_2 \vDash \exists \hat{y}.\ \Box p$, which follows from standard principles of first-order logic. The details are in Appendix[1] A.4.4.

(2) By structural induction on derivations of $\Downarrow$. The cases except for `observe` are similar to those above, except that we use Theorem 3.1 to ensure the premises of the induction hypotheses. The case for `observe` relies on the assumption that $\sigma \in \beta$ to instantiate y_{n+1}. The details are in Appendix[1] A.4.5.

□

5 CASE STUDY: THE MARS POLAR LANDER

In this section, we show how belief programming and Epistemic Hoare Logic could be used to implement and verify the control software of the Mars Polar Lander (MPL). The MPL is a lost space probe, hypothesized to have crashed into the surface of Mars during descent due to a control software error [JPL Special Review Board 2000]. We do not claim that belief programming is the first or only technique for preventing the loss of the MPL. However, the notoriety and subsequent investigation of the MPL's loss has resulted in ample documentation [JPL Special Review Board 2000] useful for illustrating in detail how belief programming and Epistemic Hoare Logic work.

The code presented in this section is written in BLIMP, except that for convenience we define two pieces of syntactic sugar. The syntax p_1 `=>` p_2 desugars to `!`p_1 `||` p_2 and the syntax x `=` p desugars to `if` p `{` x `= 1 } else {` x `= 0 }`.

The code in Figure 11 is the piece of MPL's software [JPL Special Review Board 2000] responsible for the final phase of its Martian descent. The code uses a radar altimeter as well as two touch sensors on the landing legs to monitor its progress along the descent. Note that this is a simplification from the original software, which used three touch sensors. The code consists of a state estimator to determine when it reaches the Martian surface, and control code to shut off its engine once it does.

Reading Observations. On Line 8 the controller reads the value of the radar altimeter into the variable `radar_alt`. On Lines 10 and 11 the code reads the values of the touchdown sensors into `cur_td_1` and `cur_td_2`. It also stores their previous values in `prev_td_1` and `prev_td_2`.

State Updates. The block of code from Line 13-21 sets the state variables `state_1` and `state_2` based on the values of the touchdown sensors. Specifically, if an individual sensor has indicated a `1` for two iterations in a row, its state is set to `1`. Otherwise, its state is set to `0`.

Notably the annotated lines (Lines 13 and 14) were missing from the original software, meaning that any two positive sensor readings in a row were sufficient to permanently set the state to `1`. It is hypothesized that this was part of the sequence of events that caused the MPL to crash, and these two lines are the recommended fix [JPL Special Review Board 2000].

Engine Shutdown. The block of code from Line 22-26 determines when to shut down the engine. If the health check has completed (see below) and at least one of the healthy indicators registers a touchdown, then the program sets `engine_enabled` to `0`, shutting down the engine.

```
1  //Controller Initialization
2  prev_td_1 = 0; cur_td_1 = 0; prev_td_2 = 0; cur_td_2 = 0;
3  health_1 = 1; health_2 = 1;
4  engine_enabled = 1; event_enabled = 0;
5
6  while (engine_enabled == 1) {
7      //Controller loop start
8      input radar_alt;
9
10     prev_td_1 = cur_td_1; input cur_td_1;
11     prev_td_2 = cur_td_2; input cur_td_2;
12
13     state_1 = 0; //Missing, probable cause of crash.
14     state_2 = 0; //Missing, probable cause of crash.
15
16     if (prev_td_1 == 1 && cur_td_1 == 1) {
17         state_1 = 1
18     };
19     if (prev_td_2 == 1 && cur_td_2 == 1) {
20         state_2 = 1
21     };
22     if ((state_1 == 1 && health_1 == 1) ||
23         (state_2 == 1 && health_2 == 1) &&
24         event_enabled == 1) {
25         engine_enabled = 0;
26     }
27     //Indicator health check
28     if (radar_alt < 40 && event_enabled == 0) {
29         if (prev_td_1 == 1 && cur_td_1 == 1) {
30             health_1 = 0;
31         };
32         if (prev_td_2 == 1 && cur_td_2 == 1) {
33             health_2 = 0;
34         }
35         event_enabled = 1;
36     }
37 }
```

Fig. 11. Code of the Mars Polar Lander

Health Check. The block of code from Line 28-36 performs a health check on the touchdown indicators. The health check assigns into the health variables `health_0` and `health_1`, and is performed the first time the radar altimeter indicates an altitude lower than 40 meters. At this point, the lander is off of the ground so both touchdown indicators should read `0`. If an indicator reads 1 on both the current and previous time step, it is assumed to be defective, and its health variable is set to `0`. After the health check completes, the program sets the flag `event_enabled` to 1 to allow the touchdown indicators to shut down the engine.

5.1 Error Model

From the above code, we can deduce several sources of error the MPL control software was designed to be robust with respect to:

- *Transient False Positives.* If a touch sensor is momentarily triggered, the software will not immediately assume the lander has contacted the ground. The software requires two time steps in a row where the sensor was positive before it sets its state variable to true and thereby registers that it has contacted the ground. The assumption here is that the duration of the transient false positive is no longer than one time step.
- *Permanent False Positives.* If a touch sensor is defective, it may constantly send out an indication that the lander has contacted the ground. This is detected and corrected for by the indicator health check on Lines 28-36. This block of code checks if the touch sensor yields a non-transient positive contact signal when the lander is just under 40 meters above the ground. If so, the code assumes that sensor is defective and ignores its output for the engine shutoff decision. The assumption here is that at most one sensor will be defective.
- *Permanent False Negatives.* If a touch sensor is defective, it may constantly send out an indication that the lander has not contacted the ground when in fact it has. The above code accounts for this by using two sensors and shutting off the engine when either one indicates a touchdown. The assumption here is that at most one sensor will be defective.

Landing Leg Deployment. Another source of error for the MPL that is not obvious from the code above but that has been well-documented is landing leg deployment. When the landing legs deploy about 1500 meters above the surface, the process can result in false positives that exceed the one time step assumed for transients [JPL Special Review Board 2000]. Without the two annotated lines (Lines 13 and 14), this would cause the sensor's state variable to be permanently set to 1, causing the engine to shut down immediately after the health check completed.

5.1.1 Formalization. The nondeterministic program in Figure 12 formalizes these sources of error. It provides inputs to the control software by setting the variables `cur_td_1`, `cur_td_2`, and `radar_alt`. It can be composed with the control software by inlining the code in Figure 11 on Lines 11 and 43. After composing with the control software, the resulting overall program could then be verified to prove the loop invariant on Lines 13 and 14 of Figure 12 always holds. We now describe each piece of the formal model in more detail.

Permanent Errors. The block of code on Lines 6-9 of Figure 12 models permanent errors, both false positive and false negative. Each of the variables `perm_1` and `perm_2` is 1 if its sensor has suffered a permanent failure, with the assumption being that neither of these two variables will be 1 at the same time. The variables `perm_1_v` and `perm_2_v` store the permanent error values, which are the constant values that each of the sensors will read when after suffering a permanent error.

Loop Condition. The code on Lines 12-14 of Figure 12 specifies the same loop condition as in Figure 11, but also adds a loop invariant. The invariant specifies that when the lander is off the ground, its engine is enabled. Verifying this property would ensure that the software does not have the bug that caused the MPL to crash. The invariant also specifies on Line 14 that the lander should spend less than 2 time steps on the ground with the engine on. This is another constraint the MPL was designed to satisfy [JPL Special Review Board 2000]. However, the model in Figure 12 admits transient false negative errors, which can violate this constraint in extreme cases. Thus, on Line 14 of Figure 12, we assume this holds in the case where there are no transient false negatives.

```
//Model Initialization
prev_err_1 = 0; prev_err_2 = 0; trans_td = 0;
alt = 8000;
time_on_ground = 0;
//Permanent errors
perm_1 = choose(. == 0 || . == 1);
perm_2 = choose((. == 0 || . == 1) && (perm_1 == 1 => . == 0));
perm_1_v = choose(. == 0 || . == 1);
perm_2_v = choose(. == 0 || . == 1);
//Controller initialization
...
while(engine_enabled == 1)
{ (alt > 0 => engine_enabled == 1) &&
  (trans_td == 0 => time_on_ground < 2) }
{
    //Model start
    if (alt == 0 && engine_enabled == 1) {
        time_on_ground = time_on_ground + 1
    };
    alt_rate = choose(0 <= . && . <= 39 && . <= alt);
    alt = alt - alt_rate;
    radar_alt = choose (38 <= . - alt && . - alt <= 38);

    err_1 = choose((prev_err_1 == 1 => . == 0) && (. == 0 || . == 1));
    prev_err_1 = err_1;
    err_2 = choose((prev_err_2 == 1 => . == 0) && (. == 0 || . == 1));
    prev_err_2 = err_2;
    if (alt == 0 && (err_1 == 1 || err_2 == 1)) { trans_td = 1; };

    leg_err = 1400 <= alt && alt <= 1600;
    if perm_1 { cur_td_1 = perm_1_v }
    else if (leg_err == 1 || err_1 == 1) {
        cur_td_1 = choose(. == 0 || . == 1)
    } else { cur_td_1 = alt == 0; };

    if perm_2 { cur_td_2 = perm_2_v }
    else if (leg_err == 1 || err_2 == 1)  {
        cur_td_2 = choose(. == 0 || . == 1)
    } else { cur_td_2 = alt == 0; };

    //Model end
    //Controller loop start
    ...
}
```

Fig. 12. Model for verification of the Mars Polar Lander.

Time on Ground. The code on Lines 17-19 of Figure 12 measures the amount of time the lander has spent on the ground with the engine on and stores it in `time_on_ground`. This is measured purely to evaluate the loop invariant and is not passed to the controller.

Altitude. The code on Lines 20-22 of 12 specifies how altitude changes and the error model of the radar altimeter. It stores the rate of altitude change in `alt_rate`, the new altitude in `alt`, and the altimeter reading in `radar_alt`. We assume that the altitude changes by at most 39 meters and that the altimeter is accurate to within 38 meters. The original motivation for including touchdown sensors on the MPL was that the radar altimeter is inaccurate below about 40 meters [JPL Special Review Board 2000]. This model is designed to conservatively capture this property while still ensuring that the condition on Line 28 of Figure 11 triggers the indicator health check.

Furthermore, note that on line 3 of Figure 12 we have specified that the entry point to the program is when the lander is at an altitude of 8 kilometers.

Transient Errors. The code on Lines 24-28 of Figure 12 models transient errors. The variables `err_1` and `err_2` are set to be 1 if a transient error occurred for the first or second touchdown sensor, respectively. The previous values of these variables are stored in `prev_err_1` and `prev_err_2`. This code specifies that a transient error can occur for a sensor if it did not occur at the previous time step. Furthermore, if a transient occurs after touchdown (i.e. a false negative), the code specifies that the variable `trans_td` is set to 1.

Landing Leg Deployment. To account for landing leg deployment errors, the model sets the flag `leg_err` whenever the landing gear are deploying. This occurs at about 1500 meters for the MPL [JPL Special Review Board 2000]. We have modeled a deployment window of 100 meters around this nominal value, so that landing leg deployment occurs between 1400 and 1600 meters.

Touchdown Indicators. The code on Lines 31-34 of Figure 12 specifies how the first touchdown indication is generated from the various sources of error. The result is stored in `cur_td_1`. If the indicator suffered a permanent error, then it returns its permanent error value. Otherwise, during landing leg deployment or a transient error, it may output either `0` or `1`. If none of these errors are present, then it indicates `1` iff the lander has touched the surface (i.e. the altitude is 0 meters).

The code on Lines 36-39 of Figure 12 specifies how the second touchdown indication is generated and is entirely symmetric.

5.2 Belief Programming

We now show how to use the model in Figure 12 to construct a belief program to control the MPL. We show a fragment of the belief program in Figure 13. This fragment can be completed by inlining the appropriate parts of Figure 12 into Lines 2 and 9 of Figure 13.

The belief program executes by observing each of the sensor readings generated from the model. It then determines, on Line 15, whether these are sufficient to guarantee the lander is on the ground. If so, it shuts down the engine. The belief program also modifies the loop invariant to have a $\Box$ modality, stating that it must be true in every environment (Lines 5 and 6).

5.3 Verification

In this section, we will explain how to verify the loop invariant on Lines 5-6 of Figure 13. We simplify the problem here by only considering the first condition on Line 5. The remaining condition is considered in Appendix[1] D.

Initialization Post-condition. As specified by the rules in Figure 10, we must show that the initialization code generates a post-condition that satisfies the loop invariant. This post-condition

```
1  // Model Initialization
2  ...
3  engine_enabled = 1;
4  while (engine_enabled == 1)
5  { □((alt > 0 => engine_enabled == 1) &&
6      (trans_td == 0 => time_on_ground < 2)) }
7  {
8      // Model start
9      ...
10     // Model end
11     observe radar_alt;
12     observe cur_td_1;
13     observe cur_td_2;
14
15     infer □(alt == 0) {
16         engine_enabled = 0
17     }
18 }
```

Fig. 13. Implementation of the Mars Polar Lander with belief programming

can be written as `□(engine_enabled == 1) && ...`, where the □-proposition is generated by the code on Line 3 of Figure 13. Now, we can apply the fact that

$$\forall \sigma.\ \sigma \vDash \texttt{(engine\_enabled == 1)} \Rightarrow \sigma \vDash \texttt{(alt > 0 => engine\_enabled == 1)}$$

and Theorems[1] B.4 and B.5 to see that

$$\beta \vDash \texttt{□(engine\_enabled == 1) \&\& } \cdots \Rightarrow \beta \vDash \texttt{□(alt > 0 => engine\_enabled == 1)}$$

Loop Body Post-condition. According to the rules in Figure 10, we must show that the loop body's post-condition implies the loop invariant. We can summarize the post-condition as

```
□(engine_enabled_0 == 1) &&
(□(alt == 0) && □(engine_enabled == 0)) ||
(◇(alt != 0) && □(engine_enabled == engine_enabled_0))
```

where the first line comes from the loop condition on Line 4 and the second and third line come from the infer statement on Line 15. We assume we have applied the standard rule for strongest-postcondition predicates and taken the disjunction of each branch of the `infer` [Floyd 1967].

We can now show the post-condition implies the invariant. The disjunction gives us two cases. In the first, we can assume that `□(alt == 0)`. Now, we can apply the fact that by vacuous truth,

$$\forall \sigma.\ \sigma \vDash \texttt{(alt == 0)} \Rightarrow \sigma \vDash \texttt{(alt > 0 => engine\_enabled == 1)}$$

and Theorems[1] B.4 and B.5 to see that

$$\beta \vDash \texttt{□(alt == 0)} \Rightarrow \beta \vDash \texttt{□(alt > 0 => engine\_enabled == 1)}$$

In the second case, we follow a similar logic to the initialization post-condition argument above, with the additional premise that `□(engine_enabled == engine_enabled_0)`.

6 IMPLEMENTATION

We have demonstrated the feasibility of a belief programming implementation that directly represents the belief state with a set. Specifically, we wrote the implementation in C using a hash set data structure to represent beliefs.

Called CBLIMP, our implementation is a shallow embedding of BLIMP into C; i.e. it is a C library that implements each of the core BLIMP primitives as a C function. CBLIMP's `observe` function takes, in addition to the current belief state and the variable to be observed, a parameter that is the observed value of the variable. CBLIMP's `infer` function returns a boolean to be used as a branching condition by the C program, and CBLIMP's `if` function takes as parameters callbacks for each branch that execute on modified belief states. All regular C variables in a CBLIMP program can be considered to be deterministic with respect to the belief state (i.e. they have the same value in all environments in the belief state).

CBLIMP includes functions that extend BLIMP primitives to simulate the true environment via random sampling. The simulated `observe` function presents a different interface: instead of taking the observed value as a parameter, it uses the value from the simulated true environment.

We enforce that environments are finite by augmenting `choose` statements to include a range that the newly assigned variable must belong to. For example, as part of the MPL model, Line 24 of Figure 12 gives the choose statement

```
err_1 = choose((prev_err_1 == 1 => . == 0) && (. == 0 || . == 1));
```

This specifies that `err_1` is a boolean value (i.e. it is either 0 or 1) and that whenever `prev_err_1` is 1, `err_1` must be 0. In our implementation, we have alternatively specified this as

```
err_1 = choose(0,1, prev_err_1 == 1 => . == 0);
```

where the arguments `0` and `1` of the `choose` statement are the lower and upper bounds of the range that `err_1` must belong to. Note that these statements are equivalent; whereas in the original statement we specified that `err_1` is a boolean using the choose statement's proposition, in the new statement we have instead specified this using the range bounds.

Research Question. We evaluated CBLIMP to answer the question: does the direct implementation achieve practical performance given the latency requirements of the domain? For the UAV example, a step latency of 1 second is required to match the latency of common GPS receivers [Sparkfun 2020]. For the MPL example, a latency of 10ms is required [JPL Special Review Board 2000].

Benchmarks. We used the UAV example with a 100-step time horizon and the MPL example as benchmarks. The code is nearly identical to that in the paper, with the following changes:

- The MPL benchmark includes an additional intermediate variable that captures the error between the true altitude and the radar altitude.
- We manually determined bounds for every variable specified by a choose statement to facilitate use of our implementation's augmented choose statements.

We also implemented another version of MPL that uses a different *grid size*. Note that the original version of MPL defines a uniform discretization grid for both true and radar altitude at a resolution of 1 meter. We modified this to be a non-uniform grid on both true and observed altitude with higher-altitude grid cells exponentially larger than lower-altitude grid cells. We call this benchmark "MPL-Exp". We have provided its BLIMP source code and verified it in Appendix[1] E.

Methodology. We ran each benchmark 5 times and recorded the mean and standard deviation of the time to execute a single iteration of the main while loop. We also recorded the maximum time taken by an iteration across all runs.

To construct observations to send as inputs to the belief program, we simulated the true environment alongside the belief program's execution, sampling the new true environment uniformly at random when we encountered a choose statement. While the latency measurements include both the time to update the belief state and run the simulation, we expect them to be dominated by belief state operations.

Table 1. Results of performance experiments

Benchmark	Mean +- Std. Dev.	Maximum
UAV	2.49 +- 0.37 ms	4.41 ms
MPL	2.45 +- 1.42 s	11.9 s
MPL-Exp	0.76 +- 0.56 ms	2.37 ms

Results. Table 1 summarizes the step latency for each benchmark. We can see that with the UAV benchmark, the direct implementation of belief programming is practical in the sense that the step latency is well under the 1s threshold. However, with the MPL example, the direct implementation is not practical as-is. The latency requirement of 10ms is about 1000x faster than the worst-case latency of our implementation. By contrast, our modified "MPL-Exp" benchmark is practical for the MPL problem because its worst-case latency is considerably below the 10ms threshold.

Threats to validity. We ran these benchmarks on a 2017 MacBook Pro with an i7-7920HQ CPU at 3.10GHz and 16 GB of 2133 MHz DDR3. Both of our benchmarks operate in domains that necessitate embedded computers that are less powerful. The UAV example enjoys a comfortable margin over the required latency; it is likely that the processors common on larger drones could meet the latency requirements. The MPL-Exp benchmark would typically be run on a small embedded processor, both for reliability benefits and because such processors are typically the ones hardened against cosmic radiation. Although we expect such a processor to be slower than a modern laptop computer, with a comfortable 5x slowdown margin until it violates the latency requirement, we speculate that this benchmark could meet the requirement with standard performance engineering techniques.

7 FUTURE WORK

7.1 Implementation Efficiency

There is an open question of how to design an efficient implementation for the belief programming runtime. CBLIMP directly implements the semantics of Figure 9 using an exhaustive representation of belief states. This implementation scales poorly with the number of variables in the program, and we discuss here more efficient potential runtime implementation approaches.

Runtime SMT. One approach could symbolically execute BLIMP constructs to collect constraints that an SMT solver would use to evaluate modal propositions. This approach would make use of the enhanced performance of SMT solvers compared to an exhaustive approach, and bears similarities to other languages that deploy solvers at runtime (e.g. [Samimi et al. 2010; Yang et al. 2012]).

Restricted belief states. Known efficient implementations exist when the belief states admitted by the language are more restricted than the full powerset of environments. Examples of restricted classes of belief states studied in the literature include ellipsoids [Bertsekas 1971] and polytopes [Wan et al. 2018].

Synthesis. Using the semantics of Figure 9 as a specification, a variety of synthesis tools [Delaware et al. 2015; Lau et al. 2000; Solar-Lezama et al. 2006] exist that could potentially generate more efficient implementations than naive enumeration. The goal would be to translate `infer` statements to ordinary `if` statements, using synthesis to construct a predicate for the `if` statement that is a function of the observed values and is semantically equivalent to the `infer` statement's predicate.

7.2 Logic Automation

The logic in Figure 10 and the theorems in Section B enable sound, manual reasoning about the behavior of belief programs. As with many program logics, the gap from manual to automated reasoning is the need for automated techniques for invariant inference and implication checking.

Invariant Inference. As in traditional program logics, a while loop requires a loop invariant. Although propositions in the Epistemic Hoare Logic include modalities, classic approaches such as template-based invariant inference may be directly applicable [Flanagan and Leino 2001] via templates that include modalities. An additional distinct difference from many traditional program logics is that the rules for if statements require developers to manually determine a suitable post-condition or, in other words, provide an invariant. Here too template-based techniques may be directly applicable. In either case, there may also be new opportunities for analysis-based invariant inference techniques that account for modalities.

Discharging Implications. A classic approach to discharge implications that appear in the premises of Hoare logic rules is to employ an automatic theorem prover such as Z3 [Moura and Bjørner 2008]. To apply this approach to Epistemic Hoare Logic, one would need to contend with the modalities in propositions. Recent work holds out the promise of automated reasoning techniques for modal implications via a reduction to SMT [Areces et al. 2015; Caridroit et al. 2017].

8 RELATED WORK

Set-based Uncertainty. [Combettes 1993] is a survey paper that gives an overview of how set-based uncertainty is used in the signal processing domain. It explains how programs can over-approximate the true belief state (using e.g. ellipsoids [Schweppe 1973]; more recent work has studied polytopes [Wan et al. 2018]), and how the quality of approximation can be measured [Schweppe 1973] to determine if the resulting belief state is too large. It also gives efficient algorithms [Cimmino 1938; Kaczmarz 1937] for a restricted set of operations on approximate belief states . By contrast, belief programming reasons about the exact belief state and provides a richer set of operations. However, it cannot achieve the same computational efficiency as an approximation.

Classical Verification. In Sections 2 and 5, we alluded to how the UAV and MPL examples could be verified using classical techniques. Here, we explain this process in more detail.

The developer would first compose their handwritten environment model (Figures 2 and 12) with their handwritten state estimator (Figures 1 and 11). The resulting program is in the language IMP [Winskel 1993] with the addition of choose statements that provide nondeterminism. The developer could obtain a Hoare logic for this language by either extending the logic of IMP [Winskel 1993] to include choose statements, or by rewriting it to a language such as GCL [Dijkstra 1975] which supports nondeterminism natively and also has a Hoare logic. Finally, the developer would apply the Hoare logic to the program, which requires discharging verification conditions. Because the proposition language is the standard propositional calculus, we expect there are many techniques in the literature that the developer could apply to this end.

The advantage of classical verification, relative to belief programming, is that developers can expect to draw on a wide variety of verification literature to solve the problem, as the formulation is relatively standard. The disadvantage is that developers must write the raw code of the state estimator by hand, which is a tedious process. By contrast, in belief programming, the state estimator consists of `infer` statements, which are more intuitive to use.

Epistemic and Belief Revision Logics. The Epistemic Hoare Logic we present in Section 4 is similar to dynamic Epistemic logics such as public announcement logic [Plaza 2007] and action-based

logics [Baltag and Moss 2004]. Whereas these logics typically use either propositions or abstract action spaces to modify the belief state, our logic uses a belief program to do so.

Synthesis. To avoid writing state estimators by hand, developers might generate a state estimator by synthesizing it directly from the environment model [Delaware et al. 2015; Lau et al. 2000; Solar-Lezama et al. 2006]. Such a problem would be challenging due to having hidden state encoded in the belief program that must be explicit in the state estimator.

For example, in the desired handwritten state estimator in Figure 11, the program contains the variables `prev_td_1` and `prev_td_2`, which are not related to any of the variables in the model in Figure 12. We anticipate that existing synthesis tools will have difficulty automatically inferring the existence of such hidden variables.

Dynamic Constraint Solving. Some existing programming languages [Samimi et al. 2010; Yang et al. 2012] perform constraint solving at runtime using an SMT solver. Such approaches are necessarily similar to a belief program, which also represents a constrained set of program states at runtime. Existing systems were designed for other domains, and do not articulate complete set of `choose`, `observe`, and `infer` constructs that BLIMP has.

8.1 Relationship to Probabilistic Programming

Probabilistic programming languages (PPLs) are also designed to enable developers to reason about uncertainty. We compare BLIMP to PPLs on three core axes: language features, contemporary reasoning techniques, and the practicality of inference (i.e., the implementation of `infer`) .

Language Features. BLIMP's primary programming constructs are `choose`, `observe`, and `infer` and have probabilistic analogs in probabilistic programming languages. BLIMP's `choose` has the classic interpretation of non-probabilistic, nondeterministic choice. The analog in probabilistic programming is the probabilistic `sample` construct that randomly samples a value according to a distribution. BLIMP's `observe` has a similar semantics to `observe` constructs in PPLs. BLIMP's `infer`, which enables the program itself to perform inference, has some support in PPLs as well [Baudart et al. 2020; Staton 2017].

In general, probabilistic programming provides a more flexible modeling mechanism than BLIMP's set-based uncertainty, enabling a developer to specify distributions for nondeterministic outcomes. For applications for which probabilistic models of outcomes are available, probabilistic programming can be an appropriate choice. However, probabilistic models of outcomes are not available for all applications (e.g., our MPL application) and introducing distributions can complicate reasoning, as we discuss below. BLIMP is an additional design point for applications that do not necessarily benefit from probabilistic modeling.

Reasoning. The verification of probabilistic programs is an active area of research [Sampson et al. 2014; Sato et al. 2019]. This work typically provides the ability to express and verify the probability that an assertion is true of the program. In principle, it is possible to verify BLIMP-like modal assertions in such a framework. Specifically, $\Box$ maps to an assertion that a proposition is true with probability 1; $\Diamond$ maps to an assertion that a proposition is true with positive probability.

However, a significant challenge with reasoning about probabilistic programs is that the distribution over states at any given point in a program may not have an analytical characterization. Specifically, the composition of a standard, well-known probability distribution (e.g., a Gaussian distribution) with a computation can result in a distribution that is not well-characterized by a standard, well-known distribution for which standard statistical quantities (such as mean and variance) are easily accessible.

Therefore the full modeling, programming, and reasoning workflow must carefully consider the distributions used in `sample` statements such that they adhere to the application's uncertainty model and that the resulting distributions in the rest of the program can be precisely reasoned about at an acceptable level of complexity.

BLIMP is, instead, an additional design point for modeling uncertainty that need not rely on an appropriate selection of `sample` distributions or, more generally, the complexity of techniques for reasoning about probabilistic constructs.

Practicality of Inference. BLIMP's runtime implementation to support inference – i.e. `infer` – tracks all possible environments. The resulting implementation is sound. A probabilistic programming language can take a similar strategy – i.e., exact inference – if it desires sound inference.

However, probabilistic programming languages can also leverage approximate inference algorithms for probabilistic programs such as Sequential Monte Carlo methods [Del Moral et al. 2006] that need only track a high-probability subset of the possible environments. These methods are approximate in that the probability of an event is estimated with a fidelity that is a function of the size and diversity of the tracked state. Designing algorithms to select this state efficiently is application-specific. Contemporary diagnostics for Monte Carlo methods (e.g. [Cowles and Carlin 1996; Liu 1996]) are typically not sound in that they can indicate a good approximation when the true approximation is poor. Establishing bounds on the quality of the resulting approximation is still an active area of research [Chatterjee and Diaconis 2018].

Therefore, while approximate inference methods can in practice provide empirically good results, reasoning about the soundness of their results is still an active area of research.

9 CONCLUSION

In this paper, we presented belief programming and Epistemic Hoare Logic. Belief programming enables developers to write programs that can be directly executed to give state estimators that are derived from environment models. Epistemic Hoare Logic enables developers to reason about belief programs. We discussed both by reference to the BLIMP language, with belief programming described by BLIMP's semantics and Epistemic Hoare Logic operating over BLIMP statements. We determined that belief programming is feasible by evaluating our BLIMP implementation, CBLIMP. Taken together, this work lays new foundations for soundly reasoning about the behavior of software that executes in partially-observable environments.

ACKNOWLEDGMENTS

We would like to thank Alex Renda, Deokhwan Kim, Ben Sherman, Jesse Michel, Cambridge Yang, Jonathan Frankle, and anonymous reviewers for their helpful comments and suggestions. This work was supported in part by the Office of Naval Research (ONR-N00014-17-1-2699). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Office of Naval Research.

NOTES

1: All appendices can befound in the supplementary materials section of the ACM Digital Library.

REFERENCES

Carlos Areces, Pascal Fontaine, and Stephan Marz. 2015. Modal Satisfiability via SMT Solving. In *Software, Services, and Systems*. Springer.

Ralph-Johan Back. 1978. *On the Correctness of Refinement Steps in Program Development.* Ph.D. Dissertation. University of Helsinki.

Alexandru Baltag and Lawrence S. Moss. 2004. Logics for Epistemic Programs. *Synthese* 139 (03 2004), 165–224.

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Conference on Programming Language Design and Implementation*.

Dimitri P. Bertsekas. 1971. *Control of uncertain systems with a set-membership description of the uncertainty*. Ph.D. Dissertation. MIT.

Thomas Caridroit, Jean-Marie Lagniez, Daniel Le Barre, Tiago de Lima, and Valentin Montmirail. 2017. A SAT-based Approach to Solving the Modal Logic S5-Satisfiability Problem. In *AAAI Conference on Artificial Intelligence*.

Souray Chatterjee and Persi Diaconis. 2018. The Sample Size Required in Importance Sampling. *Annals of Applied Probability* 28 (04 2018), 1099–1135. Issue 2.

G. Cimmino. 1938. Calcolo aprossimato per le souzioni dei sistemi di equazioni lineari. *La Ricera Scientifica, Series II* 9 (1938), 326–333.

P.L. Combettes. 1993. Foundations of Set-Theoretic Estimation. *Proc. IEEE* 81 (02 1993), 182–208. Issue 2.

Mary Kathryn Cowles and Bradley P. Carlin. 1996. Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review. *J. Amer. Statist. Assoc.* 91 (06 1996), 883–904. Issue 434.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. 2006. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68 (06 2006), 411–436. Issue 3.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Symposium on Principles of Programming Languages*.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy, and Formal Derivations of Programs. *Commun. ACM* 18 (08 1975), 453–457. Issue 8.

Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium on Formal Methods Europe*.

R.W. Floyd. 1967. Assigning Meanings to Programs. In *Symposium in Applied Mathematics*.

C.A.R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12 (10 1969), 576–580. Issue 10.

JPL Special Review Board. 2000. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*. Technical Report. Jet Propulsion Laboratory.

Stefan Kaczmarz. 1937. Angenaherte Auflosung von Systemen linearer Gleichungen. In *Bulletin International de l'Academie Polonaise des Sciences et des Lettres. Classe des Sciences Mathematiques et Naturelles. Serie A, Sciences Mathematiques*.

Tessa Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *International Conference on Machine Learning*.

Jun S. Liu. 1996. Metropolized Independent Sampling with Comparisons to Rejection Sampling and Importance Sampling. *Statistics and Computing* 6 (06 1996), 113–119.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*.

Jan Plaza. 2007. Logics of Public Communications. *Synthese* 158 (09 2007), 165–179.

Stuart Russel and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach* (4 ed.). Pearson.

Hesam Samimi, Ei Darli Aung, and Todd Millstein. 2010. Falling Back on Executable Specification Languages. In *European Conference Object-Oriented Programming*.

Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *Conference on Programming Language Design and Implementation*.

Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. 2019. Formal Verification of Higher-order Probabilistic Programs: Reasoning About Approximation, Convergence, Bayesian Inference, and Optimization. In *Symposium on Principles of Programming Languages*.

Fred C. Schweppe. 1973. *Uncertain Dynamic Systems*. Prentice-Hall.

Richard D. Smallwood and Edward J. Sondik. 1973. The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon. *Operations Research* 21 (10 1973), 1071–1088. Issue 5.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2006. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Sparkfun. 2020. GPS Buying Guide - SparkFun Electronics. https://www.sparkfun.com/GPS_Guide. Accessed 2020-07-16.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming*.

Jian Wan, Sanjay Sharma, and Robert Sutton. 2018. Guaranteed State Estimation for Nonlinear Discrete-Time Systems via Indirectly Implemented Polytopic Set Computation. *IEEE Trans. on Automatic Control* 63 (12 2018), 4317–4322. Issue 12.

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages*. MIT Press.

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Symposium on Principles of Programming Languages*.