Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)

# Maintaining Evolving Domain Models[*]

**Dan Bryce**
SIFT, LLC
dbryce@sift.net

**J. Benton**
NASA ARC & AAMU-RISE Foundation
j.benton@nasa.gov

**Michael W. Boldt**
SIFT, LLC
mboldt@sift.net

## Abstract

When engineering an automated planning model, domain authors typically assume a static, unchanging ground-truth world. Unfortunately, this assumption can clash with reality, where domain changes often rapidly occur in best practices, effectors, or known conditions. In these cases, remodeling the domain causes domain experts to ensure newly captured requirements integrate well with the current model. In this work, we address this model maintenance problem in a system called `Marshal`. `Marshal` assists model maintainers by reasoning about their model as a (hidden) stochastic process. It issues queries, and learns models by observing query answers, plan solutions, and direct changes to the model. Our results indicate that anticipating model evolution leads to more accurate models over naive approaches.

## 1 Introduction

Most prior work on knowledge engineering, learning, and algorithms for planning assumes that the planning model is *static*. Knowledge engineering and learning approaches assume that a ground-truth model exists and focus on how to attain it. In practice, the ground-truth is ephemeral: organizational doctrine changes, sensors and effectors break, and knowledge engineers make mistakes. When the ground-truth and the model diverge, such *model drift* causes planning-driven applications to offer reduced functionality. In practice, knowledge engineers must maintain the model. While software engineering practices have recently gained traction in planning (Vaquero *et al.*, 2013b), maintenance has largely been ignored. Maintenance can represent 60% of the total life cycle cost of traditional software (Lientz and Swanson, 1980), and represents an unaddressed problem in planning.

The declarative and constraint-based representations used in planning lend themselves to automated forms of maintenance. We present `Marshal`, a system for maintaining evolving planning domain models. `Marshal` integrates with knowledge engineering tools, mixed-initiative planners, and plan executives to learn how the model of a planning task evolves. `Marshal` interacts with users to highlight potential plan flaws, ask clarifying questions, and explain model drift.

Consider an actual scenario taken from a recent occurrence within NASA ISS operations. In the scenario, two tasks require access to nitrogen that is stored in a tank. One of the tasks, `check(N2)` necessitates closing the valves from the nitrogen tank. Unfortunately, this requirement prevents the other task, `use(N2)`, from being completed because it requires opening the valves to use the nitrogen tank. Therefore, `use(N2)` can never overlap with `check(N2)`; it has a precondition that the leak check is not ongoing. The flight controllers could identify this condition (in their mental model), as they know the procedures for both tasks. However, operations planners cannot identify the condition that the tasks must not overlap (because the constraint is not formalized in their tools). Though not represented directly, the operations planners manage to avoid violation of the condition between these two tasks over several planning episodes. Unfortunately, on one plan the condition becomes violated. The flight controllers begin executing the plan. It fails because `check(N2)` occurs while executing `use(N2)`.

This example illustrates how `Marshal` can provide value. By observing the series of successful plans, `Marshal` hypothesizes variations of the domain model that are consistent with the plans. In the scenario, `use(N2)` has an unstated precondition that `check(N2)` is not in progress, and `Marshal` can include this as a hypothesis. As soon as `Marshal` is notified that a plan is invalid, it uses this unstated precondition hypothesis to explain the error. Instead of the user debugging the model, `Marshal` hypothesizes explanations for the plan failure, repairs the model, and confirms with the user.

`Marshal`'s capabilities stem from modeling the planning model as a first-order stochastic process with a hidden Markov model. That is, `Marshal` maintains a belief state $P(X_t|z_{1:t})$ over possible models $x_t$ given evidence $z_{1:t}$ received from the user. In the case of STRIPS models, this equates to tracking how for each action, each proposition can become (or be removed as) a precondition, add effect or delete effect. We discuss and evaluate alternative assumptions about how the model drifts.

In the following, we define the model maintenance problem (MMP), the MMP applied to planning models, and our `Marshal` solution. We then discuss an empirical evaluation of `Marshal` and related work. Our results demonstrate that

[*]This work was conducted under NASA contract NNX15CA19c.

Marshal learns more accurate models of planning domains if it expects and exploits model evolution. We also show that integrating interaction modalities beyond observing plans also helps to learn more accurate models. We illustrate these findings on several domains drawn from the learning track of the International Planning Competition.

## 2 Model Maintenance Problem

Marshal addresses the model maintenance problem (MMP) under the assumption that a user's understanding (mental model) of a domain evolves, drifting away from the formal computational model of the domain. Marshal is a model maintenance (MM) system that solves the MMP by updating the computational model to correct drift. In the following sections, we focus on the MMP with the assumption that we can observe model use and have access to a domain expert to provide answers to queries.

**Model Evolution**: Let $M_t^u$ denote a user's mental model of a domain that represents ground truth at time instant $t$. For example, $M_t^u$ might represent a set of planning operators, a Bayesian network, or a procedure. The model can evolve over time so that $M_{t+1}^u \neq M_t^u$ because the user misunderstood the domain or the domain changes. For example, a blocksworld planning user might realize that only clear blocks can be grasped (a misunderstanding) or a gripper might break (a domain change).

**Model Drift**: An inconvenient side-effect of evolving mental models is that their corresponding realizations as computational models no longer match reality as the user sees it. This model drift (a.k.a., "bit rot") renders (semi-)automated reasoners useless because their solutions are no longer generated from an accurate model.

Let $M_t$ denote a computational model corresponding to a user's mental model $M_t^u$, and $\epsilon(M_t, M_t^u)$, the modeling error. Let $E[\epsilon_t]$ denote the expected error, given a filtering density $P(M_t|z_{1:t})$, so that

$$E[\epsilon_t] = \int \epsilon(M_t, M_t^u)P(M_t|z_{1:t-1})dM_t$$

where $z_{1:t-1}$ signifies observations received by the MM system at times 1 to $t-1$. Fluctuations in this expected error characterize the degree to which the model drifts.

**Model Observations**: We assume that an observation $z_t$ accompanies each time step $t$. The observation provides clues to the MM system about how the user's mental model has evolved. For example, MM systems might receive observations of a hand coded computational model, a new constraint, or a deleted state fluent. The MM system can then use these observations to refine its knowledge about the model $M_{t+1}$.

**User Query Actions**: Beyond filtering observations of user interactions, MM systems can perform a query action $u_t$ at each step $t$ with the hope that it can influence which observation $z_t$ it receives. Ideally, the preferred observations will help it reduce model error. For example, the MM system might perform a query action where it (i) performs an inference and asks if it is correct, (ii) asks whether a constraint is still present, or (iii) critiques the user's solution to a problem and asks if the critique is valid.

Under these assumptions, we state the MMP as the problem of minimizing the average expected model error. MM systems that solve the MMP must define: (i) a prior distribution $P(M_0)$ over models, and (ii) a query policy $\mu = (\mu_0, \mu_1, \ldots, \mu_T)$ where $\mu_t$ defines $u_t$, and (iii) a model update function $f$ so that $P(M_{t+1}|z_{1:t}) = f(z_t, P(M_t|z_{1:t-1}))$.

**Definition** (Model Maintenance Problem) Find $(P(M_0), \mu, f)$ that minimizes the average expected model error:

$$c = \frac{1}{T+1}\sum_{i=0}^{T} E[\epsilon_t]$$

## 3 MMP for Planning

Our motivation to address planning in the context of an MMP is that manually creating and maintaining a planning domain model is extremely challenging. While many works have developed knowledge engineering approaches to creating planning models, model maintenance is a largely manual endeavor. MMP provides for an encompassing and fluid modeling process where an MM system can adapt the domain model to correct prior modeling mistakes or to incorporate new facets of the domain. We expect and allow for errors in the model, and accommodate cyclic or gradual evolution.

Knowledge engineering for planning provides several useful modalities for expressing planning domain knowledge. We adopt three: labeled plan examples, manual edits to the planning model, and answers to MM system queries. We expect that in any of these modalities the information provided by the user can be in error.

In the following, we define the elements of the model $M_t$ and an error function $\epsilon_t$ as they relate to STRIPS-based planning. We also describe the user interactions $z_t$ and queries $u_t$ noted above.

**Definition** (STRIPS Model) The grounded STRIPS planning model $M$ defines the tuple $(P, A)$, where $P$ is a set of state propositions, and $A$ is a set of actions. Each action $a \in A$ defines the tuple $(\mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a))$, where each element of the tuple is a subset of $P$. The subsets $A_{init} \subseteq A$ and $A_{goal} \subseteq A$ correspond to the possible initial states (using effects) and goals (using preconditions).

**Example** The (simplified) ISS scenario defines $M_0^u$ as $P = \{\texttt{avail(N2)}\}$, $A = \{\texttt{check(N2)}^{\vdash}, \texttt{check(N2)}^{\dashv}, \texttt{use(N2)}^{\vdash}, \texttt{use(N2)}^{\dashv}\}$ (using $\vdash$ and $\dashv$ to denote atomic actions for the respective start and end of durative actions), $\mathsf{del}(\texttt{check(N2)}^{\vdash}) = \{\texttt{avail(N2)}\}$, $\mathsf{add}(\texttt{check(N2)}^{\dashv}) = \{\texttt{avail(N2)}\}$, $\mathsf{pre}(\texttt{use(N2)}^{\vdash}) = \{\texttt{avail(N2)}\}$, and all other action preconditions, deletes, and adds are empty. While the action start and end notation resembles that used in temporal planning, our definitions assume atomic actions.

The error $\epsilon_t$ of a model $M_t$ with respect to the ground truth model $M_t^u$ is the normalized symmetric difference of the model features:

$$\epsilon_t = \frac{|(\mathcal{I}(M_t) \cup \mathcal{I}(M_t^u)) \backslash (\mathcal{I}(M_t) \cap \mathcal{I}(M_t^u))|}{|(\mathcal{I}(M_t) \cup \mathcal{I}(M_t^u))|}$$

where $\mathcal{I}(M)$ is a set of model features. We define $\mathcal{I}(M)$, where $M = (P, A)$ as a subset of $\{\mathsf{act}_*(a,p) | p \in P, a \in A, * \in \{\mathsf{pre}, \mathsf{add}, \mathsf{del}\}\}$ where, for example, if $\mathsf{act}_{\mathsf{pre}}(a,p) \in \mathcal{I}(M)$, then $p$ is a precondition of action $a$ in model $M$.

**Example** The flight controller's initial model defines:

$$\begin{aligned}\mathcal{I}(M_0^u) =\{&\mathsf{act}_{\mathsf{del}}(\mathtt{check(N2)}^{\vdash}, \mathtt{avail(N2)}),\\ &\mathsf{act}_{\mathsf{add}}(\mathtt{check(N2)}^{\dashv}, \mathtt{avail(N2)}),\\ &\mathsf{act}_{\mathsf{pre}}(\mathtt{use(N2)}^{\vdash}, \mathtt{avail(N2)})\}\end{aligned}$$

In our scenario, if the planning system used by operations planners has the model:

$$\begin{aligned}\mathcal{I}(M_0) =\{&\mathsf{act}_{\mathsf{del}}(\mathtt{check(N2)}^{\vdash}, \mathtt{avail(N2)}),\\ &\mathsf{act}_{\mathsf{add}}(\mathtt{check(N2)}^{\dashv}, \mathtt{avail(N2)})\}\end{aligned}$$

then $\epsilon_0 = \frac{|\{\mathsf{act}_{\mathsf{pre}}(\mathtt{use(N2)}^{\vdash}, \mathtt{avail(N2)})\}|}{|\mathcal{I}(M_0^u)|} = \frac{1}{3}$.

We note that in practice $M_t^u$ does not exist in any formal sense. Our evaluation uses a simulated user with a formally represented model $M_t^u$, thereby allowing us to compute $\epsilon_t$.

**Observations**: We discuss several user interactions that are representative of how MM systems can be used in our setting. Each user interaction $z_t$ at instant $t$ falls into one of the following categories:

- $M_t^{\sim u}$, a user-supplied model (based upon the mental model $M_t^u$)
- $(\pi, \mathcal{L})$, a user-labeled plan, where $\mathcal{L} \in \{\text{valid, invalid}\}$
- $(x, \mathcal{L})$ where $x$ is a model feature, and $\mathcal{L} \in \{\text{true, false}\}$

**Example** In our example, the failed plan corresponds to the observation at step 42: $z_{42} = (\pi, \text{invalid})$, where $\pi = (\mathtt{check(N2)}^{\vdash}, \mathtt{use(N2)}^{\vdash}, \mathtt{check(N2)}^{\dashv}, \mathtt{use(N2)}^{\dashv})$.

**Query Actions**: Given the STRIPS model and observations noted above, there are several forms of queries $u_t$ that the MM can use to control drift. We focus on a single type, model feature queries. Each model feature query is of the form $x$, where $x$ is $\mathsf{act}_{\mathsf{pre}}(a,p)$, $\mathsf{act}_{\mathsf{add}}(a,p)$, or $\mathsf{act}_{\mathsf{del}}(a,p)$. The user can either address a query by providing a label pair $(x, \mathcal{L})$ or take their own initiative to generate the next observation $z_t$. Each query $u_t$ involves zero or more specific questions that the user can answer. Our simulated user always answers the top ranked query.

## 4 Marshal

`Marshal` solves the MMP by combining a particle filter (Arulampalam *et al.*, 2002) with an information-gain-driven query generation policy. `Marshal` develops a prior distribution over possible models $P(\vec{X}_0)$ as a set of particles $\{\vec{x}_0^{(i)}\}_{i=1}^N$ and then repeats for each time step $t = 1 \ldots T$:

1. *Query* the user with $u_t$, and receive an observation $z_t$.
2. *Generate N samples from a proposal distribution* $\vec{x}_t^{(i)} \sim q(\vec{x}_t^{(i)} | \vec{x}_{t-1}^{(i)}, z_t)$, $i = 1, \ldots, N$, that accounts for both model drift and the observation.
3. *Weight the particles* with their likelihood $w_t^{(i)} = P(z_t|\vec{x}_t^{(i)})P(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)})/q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t)$.
4. *Re-sample the particles* from the set of normalized-weighted particles to create the next belief state $\{\vec{x}_t^{(i)}\}$.

In the following, we discuss how `Marshal` represents the space of domain models, how it develops a prior $\{\vec{x}_0^{(i)}\}_{i=1}^N$, how it selects queries $u_t$, and how it defines the proposal $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t)$, transition $P(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)})$ and observation $P(z_t|\vec{x}_t^{(i)})$ functions.

**Space of Domain Models**: `Marshal` represents its knowledge of the domain model using a set of domain features $\mathcal{X}$. Each possible model $M$ is defined in terms of $\mathcal{X}$ (i.e., $\mathcal{I}(M) \subseteq \mathcal{X}$). `Marshal` initializes $\mathcal{X}$ by generating sets $A$ and $P$ of skolem action and proposition symbols, so that $\mathcal{X} = \{\mathsf{act}_*(a,p) | p \in P, a \in A, * \in \{\mathsf{pre}, \mathsf{add}, \mathsf{del}\}\}$. As `Marshal` observes aspects of the model (e.g., action symbols appearing in a plan), it builds an interpretation that associates user provided symbols with its skolem symbols. We assume that the user is consistent in their use of symbols.

`Marshal` represents its knowledge as a probability distribution over $\vec{X}$ where each component $X$ denotes a random variable corresponding to an element of $\mathcal{X}$. Each $\vec{x}$ is an assignment to random variables in $\vec{X}$, where each assignment $x = \top$ indicates that $x \in \mathcal{I}(M)$ for the model $M$ represented by $\vec{x}$.

**Prior Distribution**: We formulate the prior distribution $P(\vec{X}_0)$ over models by assuming a starting model $M$ where every domain feature is false (i.e., $\mathcal{I}(M) = \{\}$). We initialize the particle set that approximates $P(\vec{X}_0)$ with a set of identical particles, where all domain features are false. While this prior is unlikely to reflect the user's mental model, it will change quickly as the user provides observations.

**Query Generation**: Given the approximation $\{\vec{x}_t^{(i)}\}_{i=1}^N$ of its filtering distribution $P(\vec{X}_t|z_{1:t})$, `Marshal` computes the priority of several model features so that $u_{t+1}$ is a ranked list. The priority for the query of model feature $x \in \mathcal{X}$ is defined by its relevance (higher relevance is better priority). We define the relevance of a query $x$ by the negative expected entropy of the posterior probability of the most recent plan $\pi$ upon observing $x$ or $\neg x$, i.e. $-H(P(\pi|\vec{X}_{t+1})P(\vec{X}_{t+1}|z_{1:t}, x))$, computed with respect to the particle set. This distribution represents the probability of the most recent plan observation assuming that $x$ ($\neg x$, resp.) is observed. That is, $x$ is relevant if it distinguishes cases where the plan fails or succeeds. If there has been no plan observations, then `Marshal` does not compute relevance.

`Marshal` breaks ties between features with the same relevance by computing their information gain. The information gain is the difference in entropy of the prior distribution $P(\vec{X}_t|z_{1:t})$ and the posterior distribution $P(\vec{X}_{t+1}|z_{1:t}, x)$.

**Sample Proposal**: The proposal distribution $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t)$ is an importance density for domain models. Drawing a sample from $q$ involves updating a particle approximating $\vec{X}_t$ with respect to the observation $z_{t+1}$. Intuitively, the update should make the resulting particle correspond more closely with the observation. In the next section, we evaluate five different approaches for updating the particles; these include:

- Verbatim (V): Update particles to be complicit with user domain update and query response observations. Ignore plan observations.
- Uniform Drift (U): Similar to verbatim, but for plan observations, uniformly sample a single domain model feature to add (remove).
- Uniform Drift Generalization (UG): In addition to uniform drift, also add (remove) *related* domain model features.
- Well-Formed Drift (W): Similar to uniform drift, but treat plans differently. If a particle agrees with the plan label, then non-uniformly sample a single domain model feature to add (remove). If the particle does not agree with the plan label, then repair or injure explanations of plans that are observed as valid or invalid (resp.).
- Well-Formed Generalization (WG): In addition to the well-formed drift, also add (remove) *related* domain model features.

The verbatim approach is a reference method where `Marshal` does not hypothesize model drift. Because the initial particles are identical and it does not update the particles (aside from matching domain models and query answers), each sampled particle will be identical. Thus, the importance $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = 1$ if $\vec{x}_t^{(i)}$ respects $\vec{x}_{t-1}^{(i)}$ aside from updates specified by $z_t$, and is 0 otherwise.

The uniform drift approach treats model and query observations identically to the verbatim approach. When observing a plan, it assumes that the domain model will change in a uniformly random way by one model feature per step. The importance is $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = 1/|\mathcal{X}|$ if $\vec{x}_t^{(i)}$ differs from $x_{t-1}^{(i)}$ by exactly one assignment and is 0 otherwise. This approach is able to hypothesize model drift, unlike the verbatim approach, because it changes the model in ways that are not necessarily stated by the user.

Generalization allows `Marshal` to effectively treat particles as tracking action schemas instead of ground actions. For example, say `Marshal` samples the domain feature $\mathsf{act}_{\mathsf{pre}}(a, p)$ to be set false, where $a$ is a grounding of action schema $go(?x, ?y)$ and $p$ is a grounding of predicate $at(?x)$. With generalization `Marshal` will also set false all $\mathsf{act}_{\mathsf{pre}}(a', p')$ where $a'$ is a different grounding of $go(?x, ?y)$ and $p'$ is a grounding of $at(?x)$. Note that with generalization, `Marshal` loses its ability to learn context-dependent model features (e.g., conditional effects).

The uniform drift with generalization defines importance as $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = 1/|\mathcal{X}_\mathcal{G}|$ if $\vec{x}_t^{(i)}$ differs from $x_{t-1}^{(i)}$ by exactly one group of assignments and as 0 otherwise. The set of groups $\mathcal{X}_\mathcal{G}$ partitions the set of model features $\mathcal{X}$ into related subsets that represent groundings of the same schema feature.

The well-formed drift approach treats model and query observations as the previous methods, but handles plans differently. If a model encoded by the particle agrees with the plan label (i.e., the plan is labeled valid and the plan is valid given the model), then the particle will change by one model feature. `Marshal` uses heuristics as bias, including:

- preconditions are likely delete effects, and vice versa.
- preconditions are unlikely add effects, and vice versa.
- add effects are unlikely delete effects, and vice versa.

The importance distribution $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = \alpha a$ if $\vec{x}_t^{(i)}$ differs from $x_{t-1}^{(i)}$ by exactly one assignment where $\alpha$ is a normalization constant, and is 0 otherwise. We define $a$ with respect to the heuristics so that if it satisfies one of the heuristics above, then $a = 0.5$. If it specifically violates one, then $a = 0.01$. If it does not satisfy or violate any, then $a = 0.1$.

The well-formed drift approach updates particles that disagree with plan observations so that they agree. `Marshal` does this by repairing or introducing *flaws*. Flaws are unsatisfied preconditions or goals. Repairing a flaw involves either removing the precondition from the domain model, or adding an add effect to an action preceding it and preserving the effect by removing clobbering delete effects of intermediate actions. Introducing a flaw involves adding an unsupported precondition, removing a supporting add effect, or clobbering it by adding a delete effect. If a plan is labeled valid, but is invalid under the domain model expressed by the particle, then for each plan flaw `Marshal` uniformly samples an update to the domain model to repair it. Similarly, if the plan is labeled invalid, but is valid given the particle, then `Marshal` uniformly samples an update that introduces a flaw. The importance distribution $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = \prod_{flaw \in Flaws(\pi, x_{t-1}^{(i)})} 1/|\mathcal{R}(flaw, \pi, x_{t-1}^{(i)})|$ if $z_t$ labels $\pi$ as valid and $\vec{x}_t^{(i)}$ applies exactly one repair from $\mathcal{R}(flaw, \pi, x_{t-1}^{(i)})$ for each plan flaw, and 0 otherwise. The importance distribution is $q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t) = 1/|\mathcal{F}(\pi, x_{t-1}^{(i)})|$ if $z_t$ labels $\pi$ as invalid and $\vec{x}_t^{(i)}$ applies exactly one injury from $\mathcal{F}(\pi, x_{t-1}^{(i)})$ to create a plan flaw, and 0 otherwise.

**Example** If a particle $\vec{x}_{42}^{(0)}$ at step 42 assigns the model features the same as $\mathcal{I}(M_0)$ (an incorrect model), and `Marshal` observes $z_{42}$ (the failing plan), then the well-formed technique may sample $x_{43}^{(0)}$ by assigning the model features the same as $\mathcal{I}(M_0^u)$ (i.e., setting $\mathsf{act}_{\mathsf{pre}}(\mathtt{use(N2)}^{\vdash}, \mathtt{avail(N2)})$ to true). The invalid plan is valid under $\vec{x}_{42}^{(0)}$ (because no preconditions are unsatisfied), and the possible injuries to the plan include setting $\mathsf{act}_{\mathsf{pre}}(\mathtt{use(N2)}^{\vdash}, \mathtt{avail(N2)})$ to true, or $\mathsf{act}_{\mathsf{pre}}(\mathtt{check(N2)}^{\dashv}, \mathtt{avail(N2)})$ to true. If we sample the former injury, then the resulting particle matches the observation and also $M_0^u$.

The well-formed drift with generalization approach employs generalization in the same way as the uniform drift with generalization method, but applies it to the well-formed drift technique (above). The importance distribution is defined with respective sets $\mathcal{R}_\mathcal{G}$, and $\mathcal{F}_\mathcal{G}$ in place of the same sets without subscripts. These sets group updates that would be in the original sets with the elements from their respective subsets of $\mathcal{X}$ in $\mathcal{G}$.

**Weighting Particles**: Weighting a particle requires that we define a transition and observation function. We define the transition function $P(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}) = 1/2^{|\mathcal{X}|}$ so that the probability of a transition between any two domain models is with uniform probability. We define the observation function $P(z_t|\vec{x}_t^{(i)})$ so that observations agreeing with a domain model have high probability (we use 0.99) and those disagreeing have low probability (we use 0.01). Thus, the weight is calculated as $w_t^{(i)} = P(z_t|\vec{x}_t^{(i)})P(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)})/q(\vec{x}_t^{(i)}|\vec{x}_{t-1}^{(i)}, z_t)$.
**Resampling Particles**: We sample particles with probability equal to their normalized weight $w_t^{(i)}/\sum_{j=1}^{N} w_t^{(j)}$. The resulting set of particles are re-indexed to define $\{x_t^{(i)}\}_{i=1}^{N}$.

## 5 Evaluation

In our evaluation, we show that `Marshal` can learn how the user's mental model has changed. We employ a simulated, scripted user agent capable of (1) evolving its mental model multiple times from an initially provided model, (2) sending `Marshal` answers to queries and simulated (scripted) plans based on those models, and (3) interfacing with `Marshal` to provide empirical data on the error between the `Marshal` learned model and the simulated user's model.

Using the simulated user agent, we evaluate `Marshal` by posing three questions:

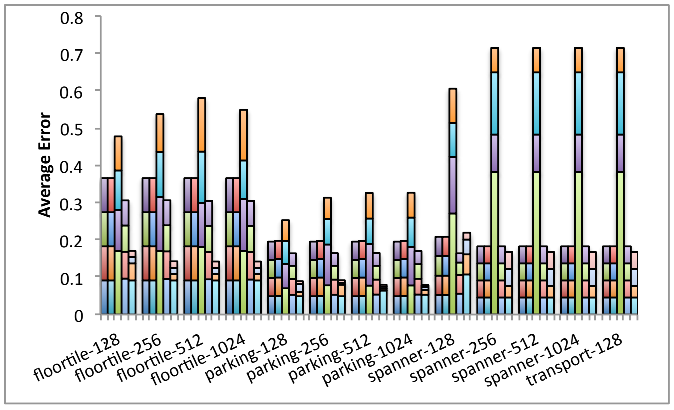
- Q1. Must `Marshal` assume a *evolving* model, or can static change be assumed?
- Q2. Do answers to *queries* help with the learning process, or are plan observations enough?
- Q3. Does a *uniform* transition function operate as effectively as a more *well-formed* transition function that captures common traits of domains?

**Experimental Setup**: Our experiments were run on a cluster containing Intel Xeon Harpertown quad-core CPUs, running at 2.83 Ghz with 2 GB of memory given to each `Marshal` instance. For each planning domain we assume that the user updates their mental model six times and after each change provides a series of 108 plans that they believe are valid. Each change is over a precondition, add or delete effect in an *action schema*. After each plan, the user answers a series of `Marshal`'s questions in the order that `Marshal` determines. After each series of plans, and just prior to the next drift in the user's model, we ask `Marshal` to calculate the probability (given its distribution over models) that each plan within a testing set of 28 plans is valid. The testing error is the difference in the actual probability that a plan is valid (given the user's ground-truth model) and the probability reported by `Marshal`. We report the average testing error over all six testing sessions in a single scenario. We also report the average model error. The testing error measures how well `Marshal` learns a useful model and the model error measures how close `Marshal` is to the current ground-truth. `Marshal` uses 128, 256, 512, or 1024 particles in its particle filter.

We note that the experiments are run faster than real-time. With each method, `Marshal` will complete a scenario as fast as the environment permits. We anticipate that actual usage will involve a human-user that will have considerably more latency than the simulated user. Each of our scenarios completes within a few hours, but represents actual usage over several months. As such, `Marshal` response time is on the order of a small number of seconds or less and within reasonable expectations for most applications.

To answer Q1 above, we evaluate the verbatim method (V) against the other methods. To answer Q2, we vary the number of (top ranked) queries answered by the user after each plan. To answer Q3, we compare the uniform (U, UG) to the well-formed (W, WG) methods.

We evaluate on the *parking*, *spanner*, *transport*, and *floortile* domains from the Learning Track of the 2014 International Planning Competition (IPC-2014). We generated 108 problems per version of the domain for each of the domains. We then produced plans using Fast Downward (Helmert, 2006) for each version of the domain and each of these problems.
**Parking**: The parking domain simulates parking multiple cars in a lot. The initial domain fed to `Marshal` has forgotten constraints over the car being at a curb before it is moved from curb-to-curb or curb-to-car, and does not include delete effects regarding this. It also excludes delete effects of the car being behind car when moving car-to-curb or car-to-car.
**Spanner**: The spanner domain simulates using a spanner to tighten nuts on a gate to fix it. A man must pick up a spanner, walk across several linked locations, then perform the appropriate fixes. Once a spanner is used, it cannot be used again. The initial domain neglects preconditions and effects over these constraints.
**Transport**: In the transport domain vehicles must transport packages between locations. Each vehicle has a limited capacity. The drifts in the domain involve reconfiguring the delete effects of the drive and pick-up actions.
**Floortile**: The floortile domain involves a robot painting tiles. The robot can move up, down, left, and right and paint going upward and downward. It also can change the color of paint it if the color is available. The initial domain does not include constraints on painting.
**Results**: Figure 1 illustrates the average model error and the average testing error for each method on each scenario, with the exception that `Marshal` ran out of memory on the transport domain instances when using more than 128 particles. We note that we generally have higher model and testing error when assuming no drift (Q1). The one exception is the uniform method with generalization (UG) has the greatest model error in all cases and surprisingly lower relative testing error. This is due to the UG method hypothesizing (potentially delusional) changes to the domain model that are not necessarily related to observations but sometimes explain plans. Increasing the number of queries does indeed help (or at least not harm) error in all cases (Q2), and the effect seems to be more pronounced as the number of particles increases. This trend is due to `Marshal` correctly hypothesizing which aspects of the model changed and then asking about them for confirmation. Of the methods that take drift into account (Q3), we see that the well-formed drift with generalization (WG) performs best, and increases performance with the number of particles and queries. It is followed by the well-formed



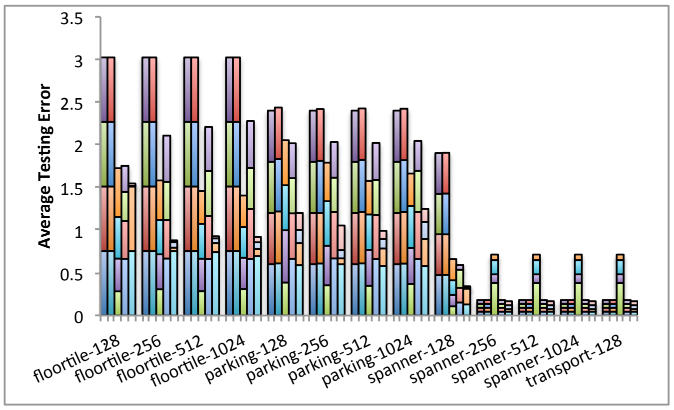


Figure 1: The average model error (left) and testing error (right) for each method and domain. Each group of columns lists results on a domain with a different number of particles. Each stacked column lists results for a method, from left to right (V, U, UG, W, WG). Within each column the results from the bottom to the top of the stack are for each number of queries per plan (0, 1, 2, 3). The data illustrates an anticipated trend that as the number of queries or particles increases, each method performs better. Across the methods, incorporating the well-formed heuristics and generalization (WG) results in the best performance.

method without generalization. The uniform methods have somewhat greater testing error, but less than the verbatim approach. Overall, it would seem that taking drift into account is useful, and that carefully applying bias in the well-formed methods is essential to tracking the evolving domain model.

## 6 Related Work

Our work on `Marshal` spans many areas, including mixed-initiative planning, expert systems, learning for planning, model-lite planning, and knowledge engineering.

Prior work on mixed-initiative planning (Chien *et al.*, 1998, Marquez *et al.*, 2013, Muscettola *et al.*, 1998, Myers, 2006, Schreckenghost *et al.*, 2014) largely treats model maintenance as an *ad-hoc* activity that relaxes or ignores constrains violated by the user.

Work on expert systems (Erman *et al.*, 1980, Freed *et al.*, 2011, Mailler *et al.*, 2009) and learning for planning has addressed model acquisition, but ignores the evolution of models. The emphasis has on acquiring the correct model from the outset rather than embracing the fluid nature of models.

Model-lite planning (Morwood and Bryce, 2012, Nguyen and Kambhampati, 2014, Weber and Bryce, 2011, Zhuo *et al.*, 2013) is more similar to our setting in that it assumes the model is only partially specified/known. However, it also assumes that the model is static. Attempts (Weber *et al.*, 2011a,b) to refine the knowledge about the model can only be effective if the model is static.

Knowledge engineering for planning (Cresswell *et al.*, 2013, Vaquero *et al.*, 2013b, Yang *et al.*, 2007) can be greatly enhanced with systems like `Marshal`. While considerable support for debugging plans and models is helpful, it can still be difficult for users to manually maintain their models. itSIMPLE does however acquire and re-use plan rationale to aid plan understanding (Vaquero *et al.*, 2011, 2013a) similar to how `Marshal` automatically adapts the model to match model usage. `Marshal` can integrate with such tools to provide quick-fixes, suggested errors, and other guidance to users. If so desired, `Marshal` could adapt the model with minimal input from users.

Other works have considered updating the domain model with input from the user. Plan post-optimization approaches adapt the model to eliminate irrelevant actions or speed up planning (Chrpa *et al.*, 2012, Nakhost and Müller, 2010), and `Marshal` could help users identify such actions.

## 7 Conclusion

We have presented a formalization of the model maintenance problem for STRIPS planning models and a system called `Marshal` that attempts to minimize the average expected error of the domain model. `Marshal` tracks the user's mental model with a stochastic process that reflects how models can evolve over time. `Marshal` uses observations of plans, domain models, and query responses to refine its knowledge about the user's model with Bayesian filtering. `Marshal` implements its model tracking as a particle filter.

Our experiments demonstrate that `Marshal` is effective at learning domain models that are evolving, and that using observations beyond just plans only improves its ability to correctly capture the user's mental model of a planning task. Our results across multiple domains from the learning track of the International Planning Competition reinforce our claims and highlight the unique capabilities afforded by `Marshal`.

In future work, we hope to extend the base planning language addressed by `Marshal` to include a broad set of relational, temporal, and resource constraints. This would require considerable extensions to the particle representation and use of the proposal distribution. We also foresee `Marshal` becoming a full mixed-initiative planner that not only provides feedback on the model, but also helps modify plans. Simultaneously modifying the domain model and plans given user interaction presents several unique and interesting challenges.

## References

M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174–188, Feb 2002.

Steve A. Chien, Nicola Muscettola, Kanna Rajan, Benjamin D. Smith, and Gregg Rabideau. Automated planning and scheduling for goal-based autonomous spacecraft. *IEEE Intelligent Systems*, 13(5):50–55, 1998.

Lukás Chrpa, Thomas Leo McCluskey, and Hugh Osborne. Optimizing plans through analysis of action dependencies and independencies. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI, 2012.

Stephen N. Cresswell, Thomas L. McCluskey, and Margaret M. West. Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28:195–213, 6 2013.

Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Comput. Surv.*, 12(2):213–253, June 1980.

Michael Freed, Daniel Bryce, Jiaying Shen, and Ciaran O'Reilly. Interactive bootstrapped learning for end-user programming. In *Artificial Intelligence and Smarter Living*, volume WS-11-07 of *AAAI Workshops*. AAAI, 2011.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26(1):191–246, July 2006.

Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

Roger Mailler, Daniel Bryce, Jiaying Shen, and Ciaran O'Reilly. Mable: a framework for learning from natural instruction. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors, *AAMAS (1)*, pages 393–400. IFAAMAS, 2009.

Jessica J Marquez, Guy Pyrzak, Sam Hashemi, Samia Ahmed, Kevin McMillin, Joseph Medwid, Diana Chen, and Esten Hurtle. Supporting real-time operations and execution through timeline and scheduling aids. In *43rd International Conference on Environmental Systems*, 2013.

Daniel Morwood and Daniel Bryce. Evaluating temporal plans in incomplete domains. In Jörg Hoffmann and Bart Selman, editors, *AAAI*, pages 1793–1801. AAAI Press, 2012.

Nicola Muscettola, P.Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(12):5 – 47, 1998.

Karen L. Myers. Metatheoretic plan summarization and comparison. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 182–192. AAAI, 2006.

Hootan Nakhost and Martin Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 121–128. AAAI, 2010.

Tuan A. Nguyen and Subbarao Kambhampati. A heuristic approach to planning with incomplete STRIPS action models. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, pages 181–189. AAAI, 2014.

Debra Schreckenghost, Tod Milam, and Dorrit Billman. Human preformance with procedure automation to manage spacecraft systems. In *Proceedings of IEEE Aerospace*, pages 1–16, 2014.

Tiago Stegun Vaquero, José Reinaldo Silva, and J. Christopher Beck. Acquisition and re-use of plan evaluation rationales on post-design. In *ICAPS workshop on knowledge engineering for planning and scheduling (KEPS)*, 2011.

Tiago Stegun Vaquero, José Reinaldo Silva, and J. Christopher Beck. Post-design analysis for building and refining AI planning systems. *Eng. Appl. of AI*, 26(8):1967–1979, 2013.

Tiago Stegun Vaquero, José Reinaldo Silva, Flavio Tonidandel, and J. Christopher Beck. itsimple: towards an integrated design system for real planning applications. *Knowledge Eng. Review*, 28(2):215–230, 2013.

Christopher Weber and Daniel Bryce. Planning and acting in incomplete domains. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*. AAAI, 2011.

Christopher Weber, Daniel Morwood, and Daniel Bryce. Goal-directed knowledge acquisition. In *ICML 2011 Workshop on Planning and Acting with Uncertain Models*, 2011.

Christopher Weber, Daniel Morwood, and Daniel Bryce. Reactive, proactive, and passive learning about incomplete actions. In *3rd ICAPS Workshop on Planning and Learning*, 2011.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artif. Intell.*, 171(2-3):107–143, February 2007.

Hankz Hankui Zhuo, Tuan Anh Nguyen, and Subbarao Kambhampati. Model-lite case-based planning. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.