Agent Design Consistency Checking via Planning

Nitin Yadav
University of Melbourne, Australia
nitin.yadav@unimelb.edu.au

John Thangarajah and **Sebastian Sardina**
RMIT University, Australia
{john.thangarajah, sebastian.sardina}@rmit.edu.au

Abstract

In this work we present a novel approach to check the consistency of agent designs (prior to any implementation) with respect to the requirements specifications via automated planning. This checking is essentially a *search* problem which makes planning technology an appropriate solution. We focus our work on BDI agent systems and the Prometheus design methodology in order to directly compare our approach to previous work. Our experiments in more than 16K random instances prove that the approach is more effective than previous ones proposed: it achieves higher coverage, lower run-time, and importantly, can handle loops in the agent detailed design and unbounded subgoal reasoning.

1 Introduction

As with any software development, verification and debugging are essential components of developing intelligent software systems. Here, we focus on agent systems developed following the BDI architecture [Rao *et al.*, 1995], a popular model that has influenced several agent design methodologies (e.g., Prometheus [Padgham and Winikoff, 2004], Tropos [Bresciani *et al.*, 2004], O-MaSE [DeLoach and Garcia-Ojeda, 2010]) and agent programming languages (e.g., JACK [Winikoff, 2005], Jason [Bordini *et al.*, 2007], JadeX [Bordini *et al.*, 2006]). There has been work on testing the correctness of BDI agent systems that have been implemented either via formal verification (e.g., [Dastani *et al.*, 2010; Dennis *et al.*, 2012; Shapiro *et al.*, 2002]) or via runtime testing (e.g., [Zhang *et al.*, 2009; Padgham *et al.*, 2013]) of *agent programs*. However, it is well accepted that identifying and correcting issues early in software development provides significant cost savings [Boehm, 1988, Page 1466]. To this end, more recent work [Abushark *et al.*, 2014; Abushark *et al.*, 2015; Yadav and Thangarajah, 2016] presents mechanisms for testing the correctness of designs *prior to any implementation*.

In [Abushark *et al.*, 2014], the authors provide an approach for checking the conformance of interaction protocols, by extracting all possible traces of the agent behavior model (comprising goals, plans and message exchanges) related to a particular protocol and report the ones that do not conform by checking against an executable structure of the protocol. In [Abushark *et al.*, 2015], they follow a similar approach for checking the consistency of requirements. Their approach however had no formal semantics and is not complete. In addition, in the design as the number of parallel steps increases, the time and space required to extract these traces grows at least exponentially. Yadav and Thangarajah [Yadav and Thangarajah, 2016] addressed these shortcomings and presented an approach to verifying the correctness of the agent detailed designs with respect to the requirements via model checking. Their technique is formal, sound and complete; it uses model checking rather than trace extraction, and presents the designer with a model as well as traces. They also empirically show that their approach is more scalable than the approach of Abushark et. al. However, there are some limitations in their work: despite the improved scalability compared to Abushark et. al. their approach still scales poorly as the design grows in size; they are unable to handle loops in the design; and they do not provide any means for *debugging designs* that are inconsistent with the requirements.

In this paper we present an approach to formally check the consistency of the requirements specification and the agent details at the design stage based on *automated planning*. Checking the consistency of an agent design is in fact a *search* problem, which makes planning technology a natural fit as we detail ahead. In order to perform a direct comparison with [Yadav and Thangarajah, 2016], we ground our approach also in the Prometheus agent design methodology, which is a popular and mature methodology that shares common principles to others [DeLoach *et al.*, 2009]. There are key advantages of our planning-based approach over that proposed in [Yadav and Thangarajah, 2016]. First, we are able to deal with loops and flexible goal hierarchies in the agent designs (which [Abushark *et al.*, 2014; Abushark *et al.*, 2015] also do not handle) as we describe in Section 4. Second, our approach is significantly more *efficient* as the designs scale up in size. We show this via experiments in more than 16K random instances (see Section 6). Finally, in our approach we also able to present a trace that is consistent with the requirements, but also go further by presenting techniques such as *swap step* and *skip step* that help in *debugging* inconsistencies between agent design and requirements (see Section 5).

Type	Name	Description
Percept	Purchase Request	A purchase order is received from the customer
Goal	Check Inventory	If the item is in stock request payment and validate it
Goal	Get Payment	Get and validate the payment
Goal	Get Payment	Invalid payment so get the Payment again
Goal	Send Item	Send the item to the customer

Figure 1: Example *Purchase Order Scenario* in Prometheus

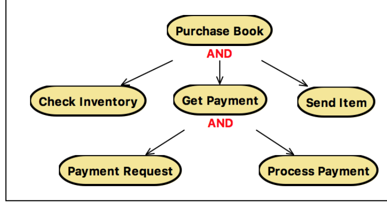

Figure 2: Example *Goal Overview* in Prometheus

2 Problem Description

There are several AOSE methodologies for developing BDI agent systems [Luck and Padgham, 2008]. Among them, the Prometheus methodology is a mature and popular methodology that also shares commonality with other AOSE methodologies [DeLoach *et al.*, 2009]. Prometheus [Padgham and Winikoff, 2004], together with the design tool (PDT) [Padgham *et al.*, 2008], supports the complete development of agent systems from specification and design through to implementation.

Two key phases of Prometheus (and other methodologies) are the *requirements* specification, and the *agent details* specification, where the internals of each agent is elaborated. The requirements are specified via *scenarios* and *goal diagrams*. Scenarios specify a particular run of the system akin to use cases in traditional Object-Oriented design. Figure 1 illustrates an example scenario related to a "Book Purchasing" system which lists the steps involved in a customer placing a purchase order. In this example, the system receives a $PurchaseRequest$ (a *percept* external to the system), forms a goal to $CheckInventory$ prior to the goal to $GetPayment$. The first attempt to get payment fails, which causes it to retry the goal and on success it continues to $SendItem$. In turn, the goal diagrams specify the functionality of the system and how they may be decomposed into smaller subgoals. The decomposition can be either "AND" or "OR" depending on whether all the subgoals must be achieved to satisfy the goal ("AND") or only one of the options ("OR"). Figure 2, illustrates a goal diagram for handling the $PurchaseBook$ goal related to the $PurchaseOrderScenario$ in Figure 1.

The *Agent Details* phase specifies the internals of each agent in terms of plans, messages, and goals that they handle and produce amongst other things. Figure 3 illustrates the detailed design for the $BookSeller$ agent that handles the $PurchaseRequest$ from the customer. Here the $PurchaseRequest$ percept activates a plan ($ProcessPurchaseOrderPlan$) that in turn triggers the $GetPayment$ and $SendItem$ subgoals.

It is these agent details that capture the behaviour of the agents in the system. The problem we address in this paper is

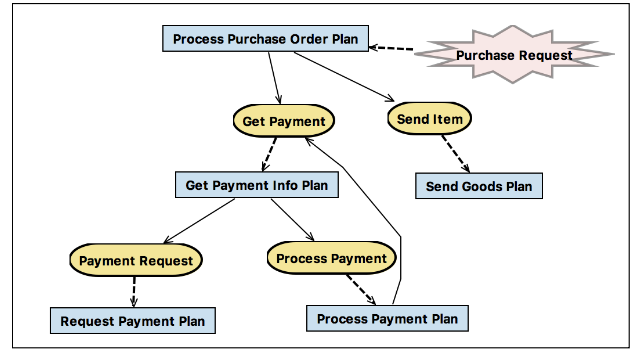

Figure 3: Example *BookSeller Agent Details* in Prometheus

whether the requirements specifications (scenarios and goal diagrams) and the agent detail diagrams are *consistent* with each other. In our example, we can see that although the requirements (Figures 1 and 2) specify that a purchase order requires a $CheckInventory$ step to be performed, the details of the $BookSeller$ agent does not include it. Our framework identifies such inconsistencies and also provides the designer with insights into possible source of errors - here, the scenario $CheckInventory$ step is missing in the agent details.

3 The Agent Design Consistency Problem

Let us make the above problem precise. To do so, we rely on the core notions in [Yadav and Thangarajah, 2016], but provide a more succinct problem definition. We begin by defining the main components, based on a given sets of goals G, percepts P, and actions A.

First, a goal overview diagram (e.g., Figure 2) is captured via a labelled ***goal tree*** $T = \langle G, g_0, \mathcal{R}, \mu \rangle$, where G is the set of goals, $g_0 \in G$ is the top level goal, relation $\mathcal{R} \subseteq G \times G$ defines the parent-child relationship between goals, and labeling function $\mu : G \to \{\mathsf{AND}, \mathsf{OR}\}$ states the achievement criteria for sub-goals. We use $\mathsf{sg}(T, g) = \{g' \mid \mathcal{R}(g, g')\}$ to denote the set of all (direct) sub-goals of goal g in tree T.

A ***scenario*** S (e.g., Figure 1) is a finite non-empty sequence of goals, percepts and actions, that is, $S \in (G \cup P \cup A)^+$. Often the first step of the scenario is a percept representing the external trigger of the behavior being represented. Given a scenario S, we use $|S|$ to denote the length of S and $S[i]$, with $1 \leq i \leq |S|$, to denote the i-th element step in S.

Then, a ***requirements specification*** is a pair (S, T) where S is a scenario and T a goal tree, as above. Note that the goal tree in a specification implicitly represents the various ways that goals can be fulfilled, by means of their sub-goals. This implies that a scenario can be realized in multiple ways.

A ***detailed design***, on the other hand, is a set $D = \{p_1, \ldots, p_n\}$, where each p_i is a plan. A ***plan***, in turn, is a tuple $p = \langle \mathsf{name}, \mathsf{trigger}, O \rangle$ where name is the unique plan identifier, the $\mathsf{trigger} \in G \cup P$ stands for the plan's trigger that may activate it, and $O \subseteq G \cup A$ is the set of outputs of the plan upon which its plan body will be built on.[1] Note that, at design level, information related to the internals of a plan body is not available. For convenience, we shall refer to the components of plan p as

[1]It is possible to handle percepts in plans via auxiliary goals.

p.name, p.trigger, and $p.O$, resp. Observe the actions required to fulfill a particular goal can be directly inferred from the agent's detailed design. Importantly, and in contrast to previous approaches [Yadav and Thangarajah, 2016; Abushark *et al.*, 2014], we *do not* preclude loops.

3.1 Problem Definition

The problem we are interested in is to *check if a detailed design D is consistent with a requirements specification* (S, T).

Requirement realization. We first define what it means for a trace to exactly capture a requirement specification, given that goals can be realized in various ways in the context of the goal tree in the specification. Consider a goal tree $T = \langle G, g_0, \mathcal{R}, \mu \rangle$, a goal $g \in G$, and a sequence of goals $\tau \in G^*$. We say that g is ***met*** by τ (relative to T) if either *(i)* $\tau = g$; *(ii)* $\mu(g) = \mathsf{AND}$, and there exists a set of goal (sub)sequences $X \subseteq G^*$ and a bijection f between X and $\mathsf{sg}(T, g)$ such that every subgoal $g' \in \mathsf{sg}(T, g)$ is met by subsequence $f(g')$ relative to T; or *(iii)* $\mu(g) = \mathsf{OR}$, and there exists a $g' \in \mathsf{sg}(T, g)$ that is met by τ relative to T.

Consider a requirement specification (S, T), with $S = o_1 \cdots o_n$, and a trace $\tau \in (G \cup A \cup P)^*$ of goals, percepts, and actions. We say that τ ***realizes*** the requirement specification (S, T) if there exist n subtraces $\tau_i \in (G \cup A \cup P)^*$, with $1 \leq i \leq n$, such that *(i)* $\tau = \tau_1 \cdots \tau_n$; and *(ii)* for all $j \in \{1, \ldots, n\}$, if $o_j \in G$, then $o_j \in G$ is met by τ_j; otherwise (i.e., $o_j \in P \cup A$), $\tau_j = o_j$.

Informally, a trace realizes a specification if it is a concatenation of sequences achieving each scenario step. For example, the requirement specification as shown through Figure 1 and Figure 2 can be met by the trace PurchaseRequest · CheckInventory · GetPayment · GetPayment · SendItem.

Design runs. Next we define what it means to "run" (i.e., execute) an agent detailed design. Intuitively, a run is an *interleaving* of various plan runs' such that each posted goal is *handled* by at most one plan.

A ***run of a plan*** $p = \langle \mathsf{name}, \mathsf{trigger}, O \rangle$ is a sequence of the form $\sigma = \mathsf{trigger} \cdot \mathsf{name} \cdot o_1 \cdots o_n$, where $o_1 \cdots o_n$ is a sequence permutation of O. This captures the fact that an agent activates a plan by handling its trigger, executing the plan body that results in the plan outputs. Let Σ_p be the set of all possible runs of a plan p. Since an agent may have more than one active plan at a time, a run for a whole agent design will consist of *interleaved* plan runs, under certain triggering constraints: a plan should be triggered for resolving a pending goal or addressing a percept.

Next, a ***run of an agent detailed design*** D is any sequence σ obtained by interleaving a finite number of runs from the set of plan runs $\bigcup_{p \in D} \Sigma_p$ such that there is a bijection f from the set of goal triggering indexes $\Delta_t = \{i \mid \sigma[i] \in G \text{ is a trigger step}\}$ into the set of goal plan outputs $\Delta_g = \{i \mid \sigma[i] \in G \text{ is a plan output step}\}$ where $f(i) < i$ for every $1 \leq i \leq |\sigma|$. Intuitively, the bijection between Δ_t and Δ_g captures the fact that every plan not started to address a percept is indeed started to resolve a (pending) goal posted by another plan, and that every posted goal (in a plan output) is addressed by some plan. The last requirement on the bijection requires the handling of every posted goal to happen after the posting in a run. For example, a run for the detailed design in Figure 3 is PurchaseRequest · ProcessPurchaseOrderPlan · GetPayment · SendItem · GetPayment · GetPaymentInfoPlan · PaymentRequest ··· There, the first GetPayment token is from the output of the plan ProcessPurchaseOrderPlan and the second GetPayment token is due to the triggering of the plan GetPaymentInfoPlan.

Design consistency. Next, we define when a run of a detail design is *consistent* with a requirement specification. Roughly speaking, this happens when the run includes, in the right order, all the steps of some realization of the specification. Formally, we say that a run σ of a detailed design D is ***consistent with a trace*** $\tau \in (G \cup P \cup A)^*$ if τ is a *subsequence*[2] of σ. Then, σ ***is consistent with a requirement specification*** (S, T) if σ is consistent with some trace that realizes (S, T). Note that a design trace will, generally, be longer than a trace for a scenario, as the former fleshes out *how* a particular requirement is achieved.

Finally, the problem we are interested is: *given a detailed design D and a requirement (S, T), is there a run σ of D that is consistent with (S, T).*

4 Technical Approach

In this section we provide a way to solve the consistency checking problem by resorting to automated planning. In classical planning, the model is provided in a *domain-independent* language and in a *succinct* manner. The most popular, and de-facto standard, representation model today is PDDL [McDermott *et al.*, 1998] (Planning Domain Definition Language), a language that allows one to formulate a so-called *planning problem* $\mathcal{P} = \langle I, G, \mathcal{D} \rangle$, where I is the initial state, G is the planning goal state, and $\mathcal{D}$ is a planning domain. We treat an agent details as a *directed goal-plan graph* where the graph's edges link goal and plan nodes. A goal node links to all the plans that can handle that goal and a plan node in turn links to all its sub-goals.

In this work we treat percepts and actions as agent goals. Technically, percepts and goals, both of them are triggers for plans. To encode actions as goals, we replace each action a with a goal g_a and add a plan $p_{g_a} = \langle p_a, g_a, \{\} \rangle$.

First we shall show how to encode a planning domain for acyclic plan libraries. Then, we resort to numerical planning to show how to deal with plan libraries that have loops. The core idea behind our encoding involves defining predicates that capture the structure of detailed design and the requirements. We rely on planning axioms to encode goal trees. The actions in the domain encode the semantics of the detailed design. To mark a completion of a scenario, we use a dummy step called *finish* that will serve as the achievement goal for the planning problem.

4.1 Planning Domain: Acyclic Plan Libraries

Our planning domain consists of three types of objects to encode agent goals, plans, and scenario steps: `(:types goal plan step)`. The predicates in our domain will encode two key aspects: the input of the conformance problem (i.e., the plan library, the goal hierarchy, and the scenario) and predicates to keep track

[2] A subsequence is a sequence that can be obtained from another sequence by deleting some elements (without changing the order).

of the exploration of this input. We use two predicates `can-handle` and `sub-goal` to encode a plan. The predicate `(can-handle ?p - plan ?g - goal)` associates a plan with its trigger. The second predicate `(sub-goal ?g - goal ?p - plan)` associates a plan with its sub-goals. To encode goal hierarchies, we require predicates to identify the goal type and to encode the goal decomposition. The predicates `(goal-and ?g)` and `(goal-or ?g)` encode AND and OR goal, respectively. A parent goal is associated with its children using the predicate `(child-goal ?child ?parent - goal)`. Similarly, to model a scenario we link a scenario step with its goal and the order in which the steps need to be achieved. The predicate `(step-goal ?s - step ?g - goal)` ties a step with a goal and the predicate `(next ?s ?s - step)` encodes the ordering of steps.

In our agent testing domain the planner will have to decide between progressing the scenario steps achieved and unfolding the detailed design. In order to keep track of these two objectives we define the following predicates. To keep track of active goals and plans we use the predicates `(active-goal ?g - goal)` and `(active-plan ?p - plan)`. It is not uncommon for a goal to be required by more than one step in a scenario. Hence, in order for a scenario step to be achieved, the goal associated with the scenario step may need to be met again. For this purpose we use a predicate `(accounted ?g - goal)` that tracks the sub-goals that are met for the current scenario step.

We use a derived predicate named `met-goal` to model the reasoning behind achieving a goal through meeting its subgoals. PDDL allows a natural translation of semantics behind achieving a goal. A goal can be said to be met if either of the following three conditions hold: (i) the goal itself is met by instantiating a plan that can handle it; or (ii) one of its children are met if the goal is of type OR; or (iii) all of this child goals are met if the goal is of type AND.

```
(:derived (met-goal ?g - goal)(or (exists (?p -
plan) (and (active-plan ?p) (can-handle ?p ?g)))
(and (goal-or ?g) (exists (?x - goal)(and
(met-goal ?x) (child-goal ?x ?g))))(and
(goal-and ?g)(not (exists (?x - goal)(and
(not (met-goal ?x)) (child-goal ?x ?g)))))))
```

In order to synchronize the progression of a step with the achievement of its goal we use the unary predicate `(canProgress)`. This predicate is true only when the goal associated with the current scenario step is met and not yet accounted for. We will use this predicate in the conditions of the actions that we define next.

```
(:derived (canProgress)(exists (?s - step ?
g - goal)(and (current ?s) (met-goal ?g) (
step-goal ?s ?g) (not (accounted ?g)))))
```

We only require three actions in our agent testing domain. Two actions are required to unfold the detailed design by handling active goals and generating new goals by activating plans; and one action for progressing the scenario steps.

The action `handleGoal` takes two parameters, a goal g and a plan p, such that goal g is pending and is the trigger of plan p. In addition, we only allow goals to be handled if the current scenario step is unachieved (this is to force the planner to progress the scenario if the current step is achieved). The effect of the action causes the plan p to be active and goal g to be inactive.

```
(:action handleGoal
:parameters (?g - goal ?p - plan)
:precondition (and (active-goal ?g)
(can-handle ?p ?g)(not (canProgress)))
:effect(and(not (active-goal ?g)) (active-plan ?p)))
```

An active plan will result in generation of its sub-goals. The action `generateGoal` is responsible for generating sub-goals of a plan in a step by step manner. Similar to the `handleGoal` action, we only generate further sub-goals if the current scenario step is unachieved. The effect of this action is to mark the new generated sub-goals as (i) active, (ii) generated from this plan, and (iii) not yet accounted for (since the current step has not yet progressed).

```
(:action generateGoal
:parameters (?p - plan ?g - goal)
:precondition (and (not
( goal-generated ?g ?p)) (active-plan ?p)
(sub-goal ?g ?p) (not (canProgress)) )
:effect (and (goal-generated ?g ?p)
(active-goal ?g)(not (accounted ?g))))
```

The last action is responsible for progressing the steps of the scenario that have been achieved. If the goal associated with the current step is met then the next step in sequence is marked as the current step and the goals met so far are set as accounted for the step achieved.

```
(:action progressStep
:parameters (?s - step ?g - goal)
:precondition (and(current ?s)
(met-goal ?g)(step-goal ?s ?g)(not (accounted ?g)))
:effect (and (not (current ?s))(forall (?x - step)
(when (next ?s ?x)(current ?x)))(forall (?x - goal)
(when (met-goal ?x)(accounted ?x)))))
```

We denote the planning domain consisting of the types, predicates, and actions we specified above as $\mathcal{D}_a$. Next we show how to construct a planning problem for this domain.

4.2 Planning Problem

Let $\mathcal{C} = \langle S, T, D\rangle$ be a consistency checking problem where S is the scenario, $T = \langle G, g_0, \mathcal{R}, \mu\rangle$ is a goal tree, and $D = \{p_1, \ldots, p_l\}$ is the detailed design where each p_i is a plan for $1 \leq i \leq l$. Also let $G = \mathcal{G}_O \cup \mathcal{G}_A$ where G_O and G_A are the goals of type OR and AND, respectively. We construct the planning problem $\mathcal{P}_\mathcal{C}$ for a consistency checking problem $\mathcal{C} = \langle S, T, D\rangle$ as follows:

Objects: The objects in the planning problem will consist of all the goals, plan names, and the steps of the scenario. We include a special scenario step labelled `finish` that encodes the achievement of all scenario steps. For the scenario $S = o_1 \cdots o_n$, we label the steps as $S1, \ldots, Sn$.

```
(:objects
 g0 ... gm - goal
 p0.name ... pl.name - plan
 S1 ... Sn  finish - step)
```

Initial condition: The initial condition consists of the first goal of the scenario as being active, encoding of the plan library, the hierarchies, and ordering of the scenario steps. The listing below shows a template to generate the initial conditions for our planning problem.

```
(:init
 (active-goal o1)
```

$\forall p \in D$ (can-handle $p.name$ $p.trigger$)
$\forall p \in D \forall g \in p.O$ (sub-goal g $p.name$)
$\forall o_i$ (step-goal S_i o_i)
$\forall g_1, g_2 \in G_i$ (child-goal g_2 g_1) s.t. $\mathcal{R}_i(g_1, g_2)$
$\forall g \in G_O$ (goal-or g)
$\forall g \in G_A$ (goal-and g)
(current o_1)
$\forall o_i, o_{i+1}$, where $i < n$ (next Si $Si+1$)
(next Sn finish))

Goal: The goal for the planning problem is to achieve the last scenario step, that is, `(:goal(current finish))`.

Proposition 1. *A detailed design D is consistent with a requirement specification, consisting of a scenario S and goal tree T, if and only if there are no loops in D and there exists a solution plan for the planning problem $\mathcal{P}_{\langle S,T,D \rangle}$ in the planning domain $\mathcal{D}_a$.*

4.3 Goal-Plan Cyclic Graphs

To account for loops in plan libraries we use planning *metrics* to keep a count of the pending goals. We introduce a function called *pending-goal* to keep a count of goals that have been produced from a plan but not yet handled.

```
(:functions(pending-goal ?g - goal)- number)
```

We update the `handleGoal` action to manage the `pending-goal` metric instead of considering the `active-goal` predicate. In the updated version, we check if the number of pending instances of a goal are at least 1 (instead of having the precondition that the goal is active). In the effect we decrease the pending goal instances.

```
(:action handleGoal
  :parameters (?g - goal ?p - plan)
  :precondition (and(>= (pending-goal ?g) 1)
   (can-handle ?p ?g)(not(active-plan ?p)))
  :effect (and(decrease (pending-goal ?g) 1)
     (active-plan ?p)))
```

We replace the `generateGoal` action by the `executePlan` action. In the `executePlan` action, we generate all sub-goals (by increasing their pending-goal count) of an active plan in a single step. In addition, since a goal can have multiple instances (due to loops in detailed design), we need to set the plan that handled a goal instance as inactive. Since we are keeping track of the number of goal that are pending, this also makes it simpler to increase the counts of all the sub-goals of an active plan in a single step.

```
(:action executePlan :parameters (?p - plan)
  :precondition (active-plan ?p)
  :effect (and (not (accounted ?g))(forall
  (?x - goal)(when (sub-goal ?x ?p)(increase
  (pending-goal ?x) 1 )))(not (active-plan ?p))))
```

In terms of the predicates, we do not require `active-goal` and the `goal-generated`. We refer to the domains constructed here to account of loops by $\mathcal{D}_c$. In terms of the problem encoding the initial condition will have the pending count of the goal associated with the first step of the scenario as 1, and for all other goals as 0. The rest of the planning problem definition will remain the same. We refer to the problem definition as defined here by $\mathcal{P}^c_{\mathcal{C}}$ where $\mathcal{C}$ is a consistency checking problem.

Proposition 2. *A detailed design D is consistent with a requirement specification, consisting of a scenario S and goal tree T, if and only if there exists a solution plan for the planning problem $\mathcal{P}^c_{\langle S,T,D \rangle}$ in the planning domain $\mathcal{D}_c$.*

5 Inconsistency Identification

From an agent designer's perspective, any feedback that points to a source of error causing the *inconsistency* will be useful in finding the potential faults in the design.

Iterative technique: In case the detailed design is inconsistent with the requirements, one can locate the first unachievable scenario step by running multiple planning problems in an iterative manner. For example, if a detailed design is inconsistent with a scenario $S = o_1 \cdots o_n$, then a designer can check for solution in $n-1$ planning problems with scenarios $\{o_1, o_1 \cdot o_2, \ldots, o_1 \cdots o_{n-1}\}$. The first planning problem that does *not* have a solution will indicate the first unachievable scenario step.

Mismatch identification actions: A more useful feedback for an agent designer would be if the framework can suggest modifications that will fix the errors. To incorporate such a feedback we extend the planning domain by introducing two actions `skipStep` and `swapStep`. We use action costs to ensure that these actions are only chosen when a plan cannot be found without them. The planning domain $\mathcal{D}_a$ is extended to include `(:functions(total-cost))` and the `skipStep` and `swapStep` actions.

The `skipStep` action takes the current step and marks it achieved. It takes a scenario step and its associated unmet goal as as input and sets the next scenario step as current. The action also has a cost of 100 (any large positive number will work). In effect this action renders the search as if a scenario step was achieved and the search can then progress towards achieving the next step of the scenario.

```
(:action skipStep
  :parameters (?s - step ?g - goal)
  :precondition (and (current ?s) (not (met-goal ?g))
  (step-goal ?s ?g) (not (canProgress)))
  :effect (and (not (current ?s))
     (forall (?x - step) (when (next ?s ?x)
             (current ?x)))
     (forall (?x - goal) (when (met-goal ?x)
             (accounted ?x)))
     (increase (total-cost) 100)))
```

The `swapStep` action swaps the sequential scenario steps. Given a scenario $o_1 \cdots \cdot o_i \cdot o_{i+1} \cdots o_n$, the action *swapStep* $o_i\, o_{i+1}$ will change the scenario to be $o_1 \cdots \cdot o_{i+1} \cdot o_i \cdots o_n$. In effect this action will try to re-sequence the goals in a scenario such that they can be achieved by the given detailed design.

```
(:action swapStep
    :parameters (?from - step ?to - step)
    :precondition (and (current ?from)
    (next ?from ?to) (not (canProgress)) )
    :effect (and (not (next ?from ?to))
        (next ?to ?from)
        (forall (?x - step) (when
          (next ?x ?from) (next ?x ?to)))
        (forall (?x - step) (when
          (next ?to ?x) (next ?from ?x)))
        (increase (total-cost) 100)))
```

In each planning problem we also specify the metric to minimize total cost of the plan: `(:metric minimize (total-cost)))`. The actions

p/g	$g/p=2$						$g/p=3$						$g/p=4$					
	h	$\#p$	$\#g$	fd	mcm	nmv	h	$\#p$	$\#g$	fd	mcm	nmv	h	$\#p$	$\#g$	fd	mcm	nmv
2	4	3.24	2.54	**100**	**100**	**100**	4	3.62	2.88	**100**	**100**	**100**	4	4.18	3.24	**100**	**100**	**100**
	8	16.68	12.24	**100**	**100**	96	8	26.76	20.14	**100**	94	62	8	39.34	29.22	**100**	56	35
	12	56.68	42.08	**87**	78	25	12	119.14	89.12	**82**	34	17	12	258.82	193.68	**79**	(-)	(-)
	16	187.44	140.10	**66**	22	(-)	16	587.52	441.02	**57**	(-)	(-)	16	2092.52	1571.22	**54**	(-)	(-)
3	4	5.14	3.18	**100**	**100**	**100**	4	5.86	3.66	**100**	**100**	**100**	4	7.02	4.26	**100**	**100**	**100**
	8	32.92	19.72	**100**	**100**	68	8	57.84	34.84	**100**	79	29	8	93.12	55.78	**100**	42	18
	12	167.16	99.84	**88**	40	(-)	12	387.78	233.72	**83**	(-)	(-)	12	1178.86	711.1	**78**	(-)	(-)
	16	792.96	477.32	**65**	(-)	(-)	16	3373.28	2029.04	**57**	(-)	(-)	16	7855.64	4718.42	**52**	(-)	(-)
4	4	7.16	3.58	**100**	**100**	**100**	4	9.38	4.62	**100**	**100**	**100**	4	11.04	5.44	**100**	**100**	**100**
	8	62.12	30.52	**100**	88	49	8	98.52	48.20	**100**	48	16	8	200.00	98.3	**100**	17	(-)
	12	361.28	180.02	**88**	8	(-)	12	1060.26	527.76	**82**	(-)	(-)	12	2771.48	1384.18	**78**	(-)	(-)
	16	3302.68	1649.20	**65**	(-)	(-)	16	13925.62	6970.72	**56**	(-)	(-)	16	38942.70	19466.78	**52**	(-)	(-)

Table 1: Percentage of cases terminated within 10 minutes. (-) indicates that less than 5% of the cases terminated.

that were in the domain $\mathcal{D}_a$ have an action cost of 1. We denote this extended domain with the repair actions as $\mathcal{D}_r$.

Proposition 3. *A detailed design D is not consistent with a requirement specification, consisting of a scenario S and goal tree T, if and only if there exists a minimal cost plan with actions* `skipStep` *or* `swapStep` *for the planning problem $\mathcal{P}_{\langle S,T,D\rangle}$ in domain $\mathcal{D}_r$.*

6 Empirical Results and Conclusion

In this section, we compare the efficiency of our planning approach with respect to the previous model checking approach [Yadav and Thangarajah, 2016]. As a planner we used fast-downward[3] and for model checking we used MCMAS[4] an ATL model checker, and NuSMV[5] a CTL model checker. We used the approach presented in [Yadav and Thangarajah, 2016] to encode the problems into MCMAS and NuSMV.

The benchmark was constructed in two steps - (i) build agent details; and (ii) for each agent detail generate multiple scenarios. The agent details were randomly generated based on three parameters: the height of the design (denoted by h), the maximum number of goals per plan (denoted by g/p), and the maximum number of plans per goal (denoted by p/g). The starting point of the agent details was always a goal and each goal was linked to at least one plan. The agent details we generated, technically, are trees with alternating levels of goal and plan nodes, and therefore we consider the height of the agent details as the height of its underlying tree. We generated both positive and negative test cases for 36 categories where the maximum number goals per plan and plans per goal were varied from 2 to 4 and the height was varied from 4 to 16 (in steps of 4). For each category we randomly generated 50 trees and for each tree, scenarios of different lengths were constructed. In total the benchmark had $16,088$ instances.

Table 1[6] shows the results for coverage of the three tools, fast downward (denoted by fd), MCMAS (denoted by mcm) and NuSMV (denoted by nmv), on the test cases. We ran the fast downward planner with `ff` as the heuristic with lazy search. The average size of the test cases in each category is represented by the average number of plans (denoted by $\#p$) and the average number of goals (denoted by $\#g$). A time limit of 10 minutes per problem instance was used for all the solvers. As evident from Table 1, the planning approach is able to solve significantly larger number of cases as compared to the model checking approach [Yadav and Thangarajah, 2016]. Observe that both the branching factors, the value of goals per plan and plans per goal, affect the number of problems that can be solved for MCMAS and NuSMV. In comparison, the coverage for the planning approach is affected more by the number of goals per plan than by the number of plans per goal. This is reasonable to expect, as it is the planner's decision when it comes to which plan node to expand for a goal, but all goals for a plan need to be catered for.

In order to closely analyse the time taken by these tools we looked at the test cases for the category with $h=8$, $g/p=2$ and $p/g=2$. We chose this category as the problem sizes were fairly complex and most of the test cases were solved within the time limit. In terms of the time consumed, the average time taken across all the test cases in this category for fast downward, MCMAS and NuSMV was $0.298s$, $1.27s$ and $22.329s$, respectively. The p-values for the two comparisons (fast downward vs MCMAS and fast downward vs NuSMV), of a two sample t-test for the hypothesis that the difference in population mean of time taken is equal to 0 was less than $2e^{-8}$. We observed that time taken by fast downward is affected less by the detailed design than by the length of the scenario. In comparison, the time taken by MCMAS and NuSMV showed greater variability with respect to the agent details. For a planner, the search effort increases with the length of the scenario, whereas model checkers first build a model and then verify properties on that model.

To conclude, (i) unlike previous approaches, our approach is able to deal with loops and flexible goal hierarchies in the agent designs; (ii) it achieves a higher coverage than the model checking approach [Yadav and Thangarajah, 2016] when tested on more than 16K random instances; (iii) the number of plans per goal affect the planner to a lesser degree than the goals per plan, whereas both these branching factors affect the model checking approach; (iv) the planning approach is more time efficient as compared to the model checking approach; and (v) the time taken by the planning approach is dependent more on the length of the scenario than the size of the detailed design.

[3] `http://www.fast-downward.org/`

[4] `http://vas.doc.ic.ac.uk/software/mcmas/`

[5] `http://nusmv.fbk.eu/`

[6] Experiments were conducted on a machine with 4Ghz corei7 CPU with 32GB RAM.

References

[Abushark *et al.*, 2014] Yoosef Abushark, John Thangarajah, James Harland, and Tim Miller. Checking consistency of agent designs against interaction protocols for early-phase defect location. In *Proceedings of the 13th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2014)*, pages 933–940, Paris, France, May 2014.

[Abushark *et al.*, 2015] Yoosef Abushark, John Thangarajah, Tim Miller, James Harland, and Michael Winikoff. Early detection of design faults relative to requirement specifications in agent-based models. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2015)*, pages 1071–1079, Istanbul, Turkey, May 2015.

[Boehm, 1988] Barry W. Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[Bordini *et al.*, 2006] R. Bordini, L. Braubach, H. Dastani, A. El-Fallah-Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.

[Bordini *et al.*, 2007] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[Bresciani *et al.*, 2004] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *JAAMAS*, 8(3):203–236, 2004.

[Dastani *et al.*, 2010] Mehdi Dastani, Koen V Hindriks, and John-Jules Meyer, editors. *Specification and Verification of Multi-agent systems*. Springer, Berlin/Heidelberg, 2010.

[DeLoach and Garcia-Ojeda, 2010] Scott A. DeLoach and Juan Carlos Garcia-Ojeda. O-MaSE: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

[DeLoach *et al.*, 2009] Scott A. DeLoach, Lin Padgham, Anna Perini, Angelo Susi, and John Thangarajah. Using three AOSE toolkits to develop a sample design. *IJAOSE*, 3(4):416–476, 2009.

[Dennis *et al.*, 2012] Louise A Dennis, Michael Fisher, Matthew P Webster, and Rafael H Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[Luck and Padgham, 2008] Michael Luck and Lin Padgham, editors. *Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, volume 4951 of *Lecture Notes in Computer Science*. Springer, 2008.

[McDermott *et al.*, 1998] Drew McDermott, Malik Ghallab, A. Howe, Craig A. Knoblock, A. Ram, M. Veloso, Daniel S. Weld, and David E. Wilkins. PDDL—The planning domain definition language. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, 1998.

[Padgham and Winikoff, 2004] Lin Padgham and Michael Winikoff. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons, Chichester, 2004.

[Padgham *et al.*, 2008] Lin Padgham, John Thangarajah, and Michael Winikoff. Prometheus design tool. In *Proceedings of The AAAI Conference on Artificial Intelligence*, pages 1882–1883, Chicago, USA, 2008.

[Padgham *et al.*, 2013] Lin Padgham, Zhiyong Zhang, John Thangarajah, and Tim Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Trans. Software Eng.*, 39(9):1230–1244, 2013.

[Rao *et al.*, 1995] Anand S Rao, Michael P Georgeff, et al. BDI agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[Shapiro *et al.*, 2002] Steven Shapiro, Yves Lespérance, and Hector J Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *AAMAS'02*, pages 19–26, 2002.

[Winikoff, 2005] Michael Winikoff. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 175–193. Springer, 2005.

[Yadav and Thangarajah, 2016] Nitin Yadav and John Thangarajah. Checking the conformance of requirements in agent designs using atl. In G. A. Kaminka, M. Fox, P. Bouquet, Hullermeijer E., Dignum F., Dignum V., and van Harmalen F., editors, *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-2016)*, pages 243–251, The Hague, The Netherlands, August 2016. ECCAI, IOS Press.

[Zhang *et al.*, 2009] Zhiyong Zhang, John Thangarajah, and Lin Padgham. *Model Based Testing for Agent Systems*, pages 399–413. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.