Early Detection of Design Faults Relative to Requirement Specifications in Agent-Based Models

Yoosef Abushark*
RMIT University
Melbourne, Australia
yoosef.abushark@rmit.edu.au

John Thangarajah
RMIT University
Melbourne, Australia
johnt@rmit.edu.au

Tim Miller
University of Melbourne
Melbourne, Australia
tmiller@unimelb.edu.au

James Harland
RMIT University
Melbourne, Australia
james.harland@rmit.edu

Michael Winikoff
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

ABSTRACT

Agent systems are used for a wide range of applications, and techniques to detect and avoid defects in such systems are valuable. In particular, it is desirable to detect issues as early as possible in the software development lifecycle. We describe a technique for checking the plan structures of a BDI agent design against the requirements models, specified in terms of scenarios and goals. This approach is applicable at design time, not requiring source code. A lightweight evaluation demonstrates that a range of defects can be found using this technique.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*methodologies*

General Terms

Reliability; Algorithm; Design

Keywords

AOSE; Requirements specification; multi-agent systems

1. INTRODUCTION

Software systems based on autonomous agents are used in a large number of applications in which the domain is highly dynamic [17, 13]. A popular model for such development is the paradigm known as the *Belief-Desire-Intention* (BDI) model [27], which has been used as the basis for a number of agent programming languages, such as JACK, Jason and JadeX [4]. These have been used in application areas such as robotics, automated manufacturing, financial management and flight management.

There are a number of methodologies for the development of BDI systems (see [31] for a recent survey). Many of these share a common set of design concepts, usually modelling the requirements of a system by use cases and goals. These are then used to develop plans which achieve these goals, according to the process appropriate to each methodology.

The complexity of the systems developed provides a natural motivation for verification methods to be applied to such systems. A large amount of existing work on verifying BDI agent systems focuses on formal verification [10], particularly using model checking techniques [12] and theorem proving [28], or on runtime testing [23]. These methods can generally only be applied once the application has been fully developed, as they require complete agent programs to be written, and then analysed.

It has long been established in software engineering that *early* detection and resolution of software defects saves time and money [3, Page 1466]. However, comparatively little work has been done on the development of methods for detection of defects in BDI systems during the design phase. There is some support for such methods in Tropos [8], which provides methods for verifying formal requirements, and for reasoning with formal models of goals. However, there is no method provided for checking the artifacts developed during the detailed design phase against the original requirements. INGENIAS [26] provides a mechanism for checking the execution of agent programs against design artifacts, but clearly this requires the programs to be written prior to the verification.

This work is the full version of an earlier extended abstract [2], and provides detail, formalisation, and evaluation. The aim of this work is to develop a suite of techniques which can used to detect defects in BDI systems during the design phase, i.e. *prior to implementation.* In our previous work, we developed techniques for checking the functional correctness of agent-based designs with respect to communication protocol models [1]. In this paper, we build on that work and develop a technique for checking the functional correctness of agent designs with respect to *requirements models*. As with [1], our approach requires only design-phase models.

We propose a mechanism, grounded in the Prometheus methodology [24], for checking that the detailed plan structures conform to the requirements specified in terms of scenarios (the use-case representation in Prometheus) and goals. It is worth noting that what we are proposing goes beyond the simple static consistency checks that tools such as the Prometheus Design Tool (PDT) [21] perform, as our proposal considers the dynamic behaviour of a system.

We chose the Prometheus methodology due to our expertise in it and the ease of access to the supporting tool [21]. However, we believe our approach is generalisable to other BDI methodologies.

A preliminary evaluation of this technique has shown that it can be used to find a number of defects, using the conference manage-

*Acknowledges King Abdulaziz University for scholarship.

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, 2015, Istanbul, Turkey.*

	Type	Name	Role
1	Percept	Review Phase	Review Management
2	Goal	Invite Reviewers	Review Management
3	Action	Send_Invitations	Review Management
4	Percept	Reviewers_Preferences	Review Management
5	Goal	Collect Prefs	Assignment
6	Goal	Assign Reviewers	Assignment
7	Action	Give_Assignments	Assignment
8	Percept	Review_Report	Review Management
9	Goal	Collect Reviews	Review Management
10	Goal	Get PC Opinions	Review Management

Figure 1: Review Scenario Description

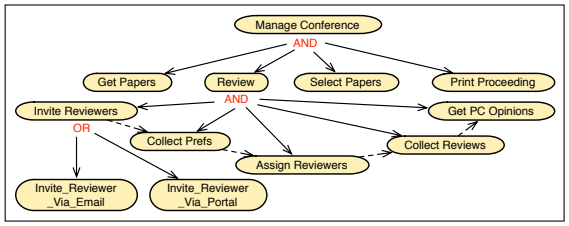


Figure 2: Goal Diagram for the Conference Management System

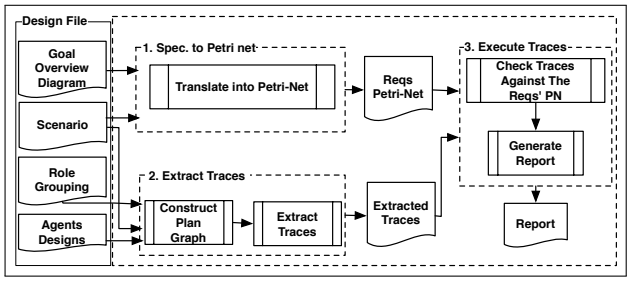


Figure 3: Process for the proposed framework

ment system as an example. We have also performed some scalability testing, to see how well our technique can handle large systems. There is still more evaluation that can be done, but we believe this shows our technique is promising.

This paper is organised as follows. Section 2 introduces the necessary background. Our approach to checking the requirements specification is detailed in Section 3, and a (limited) evaluation of its scalability and effectiveness is discussed in Section 4. A brief comparison to related work is presented in Section 5, and we conclude with discussion and future work in Section 6.

2. BACKGROUND

We now briefly introduce the relevant parts of the Prometheus models, using the conference management system [22] as a running example. This system helps in managing the different phases of the conference review process, including submission, review, decision and paper collection. In the submission phase, the system should be able to assign a number to each submission and provide receipts to authors. After the specified submission deadline, the system assigns papers to the reviewers, who review the paper. After receiving the reviews, the system supports making decisions on whether to accept or reject each paper, notifying the authors. Then, the system collects the accepted papers and prints them as conference proceedings.

A fundamental aspect of any software engineering methodology is the specification of requirements. In agent-oriented software engineering (AOSE), requirements specifications generally include [31, Section 4] *scenarios*, which are instances of the desired execution behaviour, and *goals*, which are intended states of the system. In the Prometheus methodology, scenarios consist of a sequence of steps, where each step can be an action (i.e. something an agent does), a percept (i.e. an input from the environment), a goal to achieve, or a sub-scenario.[1] Each step is associated with a role. Figure 1 shows a scenario for the conference management system. Note that the aim of the scenario is to capture an example trace through the system's behaviour, and it therefore does not specify a complete set of execution traces. However, as we shall see, we can use the information in scenarios and goal overview diagrams to construct constraints that must be met by the detailed design.

Goals are commonly modelled using a *goal diagram* that shows the relationship between goals, including how goals are decomposed into sub-goals, and whether the relationship is disjunctive (a parent goal is achieved if one of its children is achieved, "OR"), undirected conjunctive (a parent goal is achieved if all its children are achieved in some, unspecified, order, "undirected-AND"), or directed conjunctive, i.e. sequential (a parent goal is achieved if all its children are achieved in a specified order, "directed-AND"). Figure 2 shows the goal overview diagram for the conference management system, which outlines the goals and sub-goals required to successfully review the papers. The notation used distinguishes between *undirected-AND* ("AND") and *directed-AND* by using dashed arrows to indicate the ordering constraints, e.g. *Invite Reviewers* to *Collect Prefs*).

During the design process, roles are assigned to the agents in the system. For example, we assign the two roles 'Review Management' and 'Assignment' to the 'Review_Manager' agent. Scenarios are realised by designing the agents that are assigned roles that are involved in the scenario. Thus, the 'Review_Manager' agent should be designed in a way to cover all steps that are associated with the 'Review Management' and 'Assignment' roles.

The internals of agents are modelled in terms of goals, actions, percepts, and messages, which are associated with plans. Each agent type has a detailed design diagram that shows its plans, the trigger for each plan (a percept, message or goal), and the actions performed, messages sent and subgoals posted by each plan.

3. TECHNICAL APPROACH

To check the agent designs (i.e. the detailed structure of plans within agents), we compare all possible executions of the agent designs against the desired traces specified by the scenarios. We transform the design models from Prometheus-specific informal models into Petri nets. This has two benefits. First, it generalises the approach, making it applicable to other methodologies. Second, it allows us to leverage existing tools and techniques. Specifically, we transform scenarios, with additional information from the goal overview diagram, into Petri nets, and also translate agent designs into Petri nets. We then verify that all traces of the design Petri net are valid with respect to the scenario Petri net.

Our process takes a design file as input, and outputs a report of potential defects in the design. The process has three steps (see Figure 3):

1. *Transforming specification models into Petri nets:* In addition to considering the scenario, we also need to consider the goal overview diagram because a goal in the scenario may be realised in the agent design by achieving its sub-goals, or through its parent.

[1]There is also an "other" step type, the details of which are not important here.

2. *Extracting all possible execution traces from agent designs:* For each scenario, we automatically construct a *plan graph* [16] from the agents' detailed designs, which is an intermediate structure that defines the control flow between and within plan structures over all of the agents. We use information from both the scenario (to determine the starting point for the plan graph) and assignment of roles to agents so we can map scenario steps, which are assigned to roles, to the correct agents.
3. *Executing design traces against the Petri nets:* We check each of the possible traces of the design's Petri net against the scenario Petri net, and note any discrepancies, such as a failure in executing the Petri net that was due to an incorrect ordering between goals in the detailed design.

3.1 Transforming Scenarios to Petri nets

The first step of the process is translating the scenario into a Petri net, taking into account the goal overview diagram. Although a scenario is a single sequence of steps, the result is a set of sequences, because a goal step can be realised in the detailed design by implementing its children (more generally, its descendants), or its parent. For example, the goal step *Invite Reviewers* in Figure 1 could be implemented through its children, *Invite_Reviewer_Via_Email* or *Invite_Reviewer_Via_Portal* (see Figure 2).

In our approach, we translate scenarios to a variant of Petri net control fragments including: *Choice*, *Parallel* and *Sequence*.

When mapping a goal step we need to consider its parent and its children. To model the possibility that a goal step in a scenario could be realised by implementing its parent goal, we map a goal G with parent P as a choice fragment, with the goal G as one option in the choice, and P as the other option (except when P is OR decomposed - see below). For example, in Figure 4 the goal *Collect Prefs* is in a choice fragment with its parent, the goal *Review*. For completeness, we should include not just the parent goal, but all ancestor goals. However, in practice, it would be highly unusual for a scenario step to be implemented in terms of its grandparent, since this corresponds to a situation where the implementation operates at a much higher level of abstraction than the scenario.

We also need to consider the child goals of a scenario goal step. If the goal has no children, the goal is mapped simply as an alternative with its parent. However, there are three different cases for goals with children:

1. *OR-composed children:* The goal, G, has children G_1 OR ... OR G_n. In this case, the mapping is to an alternative between the goal G, its parent (if there are no OR siblings), or one of its children.
2. *Undirected-AND-composed children:* The goal, G, has children G_1 AND ... AND G_n. In this case, the mapping is to an alternative between the goal G, its parent, or ALL of its children, which can be achieved in parallel.
3. *Directed-AND-composed children:* The goal, G, has children G_1 AND $\rightarrow$ ... AND $\rightarrow$ G_n. In this case, the mapping is to an alternative between the goal G, its parent, or ALL of its children in the sequence specified.

For example, in Figure 2, the goal *Invite Reviewers* is OR-decomposed with two children, and so in Figure 4, it is mapped to a choice with its parent ('Review'), and with its children, where the two children are in *choice* with each other.

We now formalise the translation. Let G be the goal corresponding to the goal step being mapped, let $parent(G)$ denote its parent, $children(G)$ denote its children as a sequence of names (where the order is the order of execution for a directed-AND decomposition, and is arbitrary otherwise), and let $decompose(G)$ denote the decomposition type of G, i.e., one of LEAF, OR, AND (undirected-AND), SEQ (directed-AND). For brevity we also define a simple textual notation for depicting Petri net control fragments: a name is short hand for a step and we use $seq(p_1; \ldots; p_n)$ to denote the Petri net where the p_i are joined sequentially; $choice(p_1; \ldots; p_n)$ to denote the Petri net where there is a choice, and exactly one of the p_i is selected; and $par(p_1; \ldots; p_n)$ to denote the Petri net where all the p_i are triggered to run in parallel.

We map a scenario consisting of steps named $S_1 \ldots S_n$ to the Petri net denoted by $seq(\widehat{S_1}; \ldots; \widehat{S_n})$. If S_i is an action or percept then $\widehat{S_i} = S_i$. If S_i is the name of a goal step then $\widehat{S_i}$ depends on the decomposition type of G, formalised as:

$$\widehat{G} = \begin{cases} choice(\overline{G}) & \text{if } DG = LEAF \\ choice(\overline{G}; M) & \text{if } DG = OR \\ choice(\overline{G}; par(M)) & \text{if } DG = AND \\ choice(\overline{G}; seq(M)) & \text{if } DG = SEQ \end{cases}$$

where $DG = decompose(G)$
and $\langle G_1, \ldots, G_n \rangle = children(G)$
and $M = \widehat{G_1}; \ldots; \widehat{G_n}$

We also define the auxiliary function $\overline{G}$, which maps the scenario step itself, G, to $parent(G); G$, and all other goals to just themselves. This is used to allow $\widehat{S_i}$ to be applied recursively to the descendants of G without introducing parents of descendants. The definition of $\overline{G}$ has a special case where the scenario step being mapped is a specific child of an OR-decomposed goal (e.g. if the scenario step was 'Invite_Reviewer_Via_Email'). In this case we want to honour that choice, and not allow the design to realise the goal in terms of its parent.

$$\overline{G} = \begin{cases} G; parent(G) & \text{if } G \text{ is the scenario step being mapped} \\ & \text{and } decompose(parent(G)) \neq OR \\ G & \text{otherwise} \end{cases}$$

This definition can be easily implemented, and for the running example, yields
seq(Review Phase ; choice(Invite Reviewers ; Review ; Invite_Reviewer_Via_Email ; Invite_Reviewer_Via_Portal) ; Send_Invitations ; Reviewers_Preferences ; choice(Collect Prefs ; Review) ; choice(Assign Reviewers ; Review) ; Give_Assignments ; Review_Report ; choice(Collect Reviews ; Review) ; choice(Get PC Opinions ; Review)),
the start of which is shown in expanded form in Figure 4.

3.2 Extracting Execution Traces

To verify a particular scenario against the designs of the agents involved in that scenario, we extract all possible execution traces that the agents can realise for the scenario, which is an extension to our earlier work [1]. It is not unusual that a design of a scenario is scattered across multiple agents, since roles that are associated with steps of the scenario might be assigned to multiple agents. As a result, the extraction of the execution traces may require considering the detailed designs of multiple agents together. However, in the scenario we are using as an example, only a single agent is involved (the *Reviewer Manager* agent).

3.2.1 Merging multiple agent designs

Our starting point is a detailed design, which consists of, for each agent, a collection of plans. Each plan has a defined trigger,

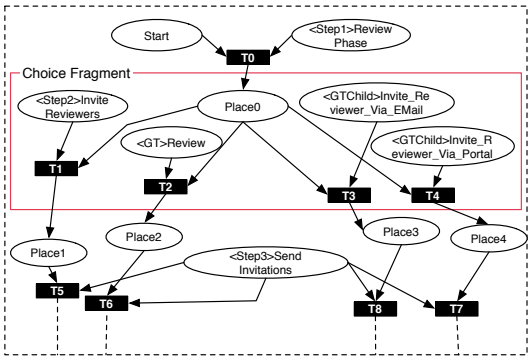


Figure 4: Part of the equivalent Petri net for the Review Scenario in Figure 1 (GT: Goal Tree)

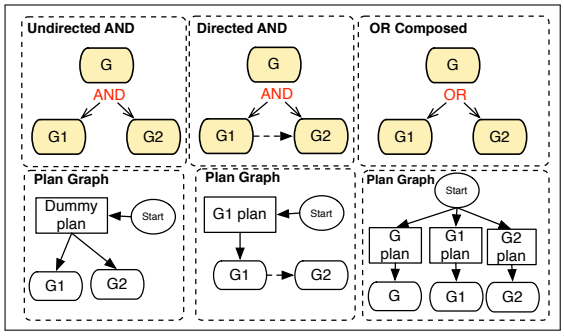


Figure 5: Plan graphs for the three types of goal compositions

which can be a goal, message, or percept. Plans can create sub-goals, send messages, and perform actions. A detailed design can be represented as a graph with distinguishable node types for actions, percepts, goals, messages and plans, and with edges between nodes indicating relationships (e.g. a plan posting a sub-goal, or a plan being triggered by a percept).

We create a *plan graph* [16] from the detailed design by starting with the entity that corresponds to the first step of the scenario, and then recursively traversing links in the agent designs. These links may be spread over several agents; for example, via the sending of a message. Therefore, the extracted plan graph is a subset of the agents' (merged) designs.

There are three specific challenges that need to be addressed, and that require a deviation from a simple subset of the agents' designs. The plan graph is based on the work of Miller *et al.* [16]. However, Miller *et al.* define a plan graph with respect to an interaction protocol, so their plan graphs capture only the relationship between plans and messages. By contrast, we are interested in the relationship between plans and actions, percepts, messages, and goals. This introduces additional challenges.

First, there may not be a single entity that corresponds to the first step of a scenario. If the first step is an abstract goal, then a subset of the goal's descendants could start the scenario, meaning there may be a number of possible starting points in the scenario:

1. *OR-composed children:* In this case, the starting point is the goal or *one of* its children. The plan graph is derived by following links from each of the possible starting points, and then having these starting points be alternatives.
2. *Undirected-AND children:* In this case, the starting point is any of the sub-goals, but all sub-goals must occur. The plan graph is derived by introducing a dummy plan that links to all of the sub-goals, and that dummy plan is triggered by the start of the process.
3. *Directed-AND children*: The first sub-goal in the sequence is the starting point for the plan graph.

Figure 5 presents examples of each of these relationships, and their corresponding plan graphs, where G is the (abstract) goal that is the first step of the scenario being considered.

Second, we need to deal with actions. In a detailed design, an action is not followed by another step (does not have any outgoing arcs), and hence terminates the plan graph. However, in scenarios an action does not always end the process. Thus, an action step in the scenario creates a "gap" in the plan graph relative to the scenario. For example, in the *Review* scenario the *Send_Invitations* action creates a gap, since, in the detailed design, there are no outgoing arcs from the action, but the action is not the end of the scenario. We rectify this by using the scenario to "bridge the gap". If an action is not the final step of the scenario, then we use the steps defined in the scenario to determine the link to continue the flow of the plan graph. For example, if the step following an action is a percept, then we use the percept as a continuation of the plan graph, and if it is a goal then we use the goal. We add a dashed link from the action to the corresponding continuation point. For instance, *Reviewers_Preferences* follows *Send_Invitations* in the scenario, so we add a dashed link from the *Send_Invitations* action to the *Reviewers_Preferences* percept in Figure 6a. This approach also applies when the first step of a scenario is an action.

Third, it is possible for the first step to be assigned to a role that is associated with multiple agents, meaning that multiple plan graphs need to be considered. We therefore consider the detailed designs of all agents that are linked to that role, deriving multiple plan graphs, and positioning each plan graph as an alternative. Thus the starting nodes of each plan graph will be children of a single starting node to form one large plan graph from all designs.

Figure 6a shows the plan graph corresponding to the design of the *Review Manager* agent, which fulfils the role from the *Review* scenario. For brevity, we omit the agent internal design from which the plan graph was extracted.

3.2.2 *Extracting execution traces from the plan graphs*

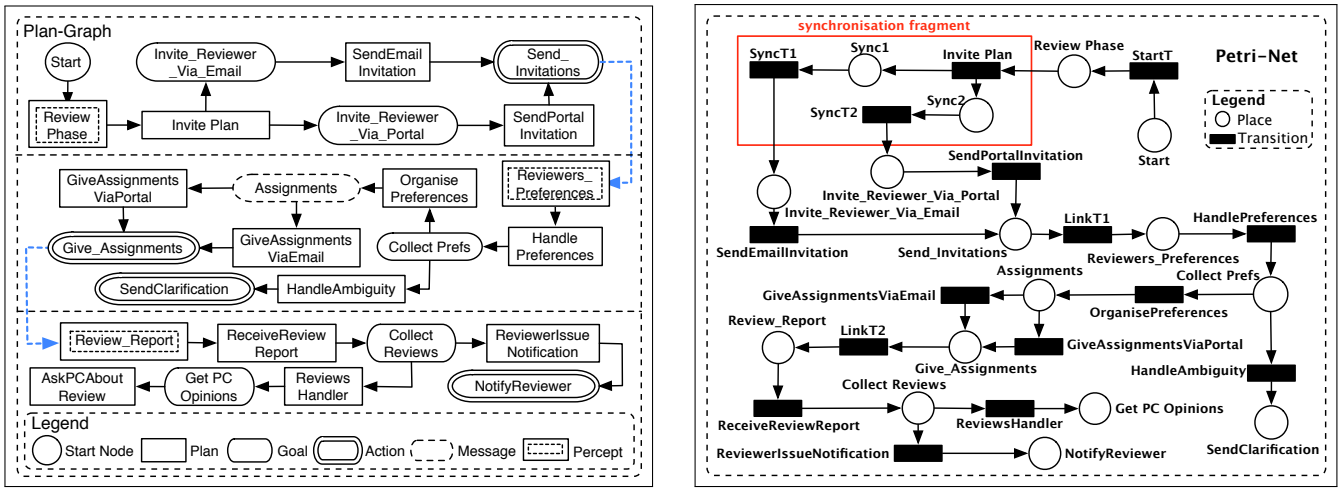
Plan graphs are structures that show the *static* view of agents with respect to a particular scenario. Each path through a plan graph represents one execution trace of the system, and each of these traces should conform to the requirements specified by the scenario.

To execute traces, we translate the plan graph into a Petri net, and then use the standard reachability graph construction [18] to obtain all possible traversals of the plan graph. Translating the plan graph to a Petri net is straightforward: plans are mapped to transitions, and other node types are mapped to places. Edges between nodes are mapped across. However, there are a few exceptions to this.

First, dashed edges, which are from an action to a percept, are handled by creating a new transition node, which sits in between the two place nodes (e.g. *LinkT1* and *LinkT2* in Figure 6b).

Second, if the first node in the plan graph is a plan, then we create a start node (place) that links to it. However, if the first node is not a plan, then we create a start node (place) that links to a new transition called *StartT* that links to the first node. As noted earlier, there can also be cases where there are a number of possible alternative starting nodes. In this case we create for each possible starting point that is not a plan a corresponding transition, and the Start place node links to each of these transitions.

Third, when a plan posts multiple steps (e.g. goals or actions),



(a) Plan graph

(b) Petri net

Figure 6: Plan graph and Petri net for the *Review* scenario

the order is not specified in the design. In particular, with subgoals, the ordering of the steps to achieve those subgoals may be interleaved. Therefore, we treat these steps of a plan as if they were executed in parallel to allow for the different possible interleaving – that is, we create a separate asynchronous process for each step. The *synchronisation fragment* in Figure 6b represents this process. Without this fragment the steps will not be executed concurrently, and hence all traces will capture the same order between the respective steps. For instance, without the sync parts in Figure 6b, a token would be simultaneously deposited on *InviteReviewerViaEmail* and *InviteReviewerViaPortal*, resulting in all traces with this order, whilst the plan graph does not specify such an order.

Finally, we exclude the plans that handle the last entities in the plan graph, as they do not affect the trace (the scenario does not include plans), and would result in transitions without output places. Figure 6b shows the Petri net resulting from translating the plan graph of Figure 6a.

After transforming the plan graph into a Petri net, we need to extract all possible trace paths for that Petri Net. This is done using the *reachability graph* of the Petri net, which is a directed graph that shows all reachable states of the Petri net.

3.3 Execution and Reporting

The previous steps have resulted in a Petri net N_S that corresponds to the scenario, taking into account the possibility of goals being realised through their parent or descendent goals, and another Petri net N_P that corresponds to the plan graph for that scenario. In this step, we take each trace from the reachability graph of N_P, and check it against N_S. Any cases where the trace does not correspond to an execution of N_S is reported.

The possible traces are extracted by traversing the reachability graph using a standard depth-first search. Each path of the reachability graph represents a possible trace of N_P, and therefore its corresponding plan graph. As mentioned earlier, we exclude any entities not related to the specified scenario, since the N_S Petri net does not include such entities. Specifically, we filter out of each trace any elements that do not correspond to a place in N_S (i.e. are included in the $seq(\widehat{S_1}; \ldots; \widehat{S_n})$).

Each trace is then verified by checking that it can be successfully executed by N_S. This is done using the entities of the trace as tokens on N_S. Given a trace of N_P of the form $S_1, S_2, \ldots, S_m$ where each S_j is the name of a step, we perform the following process: (1) set j to 1; (2) place a token on the place in N_S corresponding to S_j; (3) if there are no enabled transitions, and the execution is not at a termination state then this indicates an error, otherwise repeatedly fire enabled transitions in N_S until there are no more enabled transitions; (4) if $j = m$ then stop, otherwise increment j by 1 and go to (2).

A failure in firing a transition (in step 3) indicates a defect in the design with respect to the scenario. A trace is considered successful when its execution consumes all entities and hits a termination state of N_S. Based on the execution of all traces on the Petri net, the framework provides informative feedback about agent detailed designs with respect to a particular scenario. Table 1, modified from [1], lists possible failures and their causes.

For example, consider Figure 6a, which contains four intentionally seeded defects: the plan (*Invite Reviewers*) posts two subgoals which should be an 'OR' decomposition, yet as specified in the design they are considered as 'AND' decompositions; a missing scenario step (*Assign Reviewers* goal step); and introducing two new entities (*HandleAmbiguity* and *SendClarification* actions) that result in the existence of some traces that only cover part of the scenario.

After executing the 12,951 traces of N_P, four defects in the design were revealed: one inconsistent ordering, one missing step and two incomplete paths. For instance, *Review Phase*, *Invite_Reviewer_Via_Email*, *Send_Invitations*, *Reviewers_Preferences*, *Collect Prefs* and *Give_Assignments* ... is one of the traces of N_P. The execution of N_S would be terminated at the transition that is associated with the *Assign Reviewers*[2] goal step place, since it is missing in the trace. Since that the execution was terminated without reaching a termination place, an error was recorded.

3.4 Tool Support

We have implemented the proposed approach as an eclipse plug-in that integrates with the Prometheus Design Tool (PDT). The plug-in takes the design as an XML file, and asks the user to select a scenario to analyse. It then generates: (1) the requirements

[2]Recall that this goal has been deliberately removed from Fig 6b.

Table 1: Categorisation of causes for failures

	Failure (executing specification Petri net)	Cause (in plan graph)
1	The remaining trace is empty, but the Petri net has not terminated (there is no token in a termination place).	1. The trace contains fewer steps than it should, relative to the scenario.
2	The Petri net has terminated, but the remainder of the trace is non-empty.	2. The plan graph trace contains more steps than it should, relative to the scenario.
3	A token is placed into the Petri net, but the Petri net cannot be executed.	3.(a) The step that needs to be executed is missing in the trace; or 3.(b) The ordering between steps within the trace is inconsistent with the scenario.

Petri net N_S; (2) the plan graph; (3) filtered execution traces; and (4) a textual report that lists potential defects in agent designs.

4. EVALUATION

In this section we evaluate the framework first with respect to its scalability as the number of goals and plans in the design increases, and second with respect to its effectiveness in detecting errors.

4.1 Scalability evaluation

In the first part of the evaluation we examine how well the proposed approach scales as the size of the agent design grows. The complexity of our approach rests on the trace extraction from the resulting plan graphs, in terms of its size, but more importantly, from the level of parallelism that it contains. Therefore, we generate synthetic plan graphs systematically varying the size and the amount of parallelism up until the time taken to extract the traces are still "reasonable". We define "reasonable" to be within 24 hours.

4.1.1 Experiment design

We first generate synthetic plan graphs that are a combination of plans and other design entities including goals, messages, actions and percepts. We generated a total of 21 different plan graphs using the systematic process described below. We extracted and executed the traces recording the following measures: (i) the number of traces corresponding to a plan graph; (ii) the time taken to extract all traces of plan graph; and (iii) the time taken to execute all traces against the requirements Petri net model.

Plan graph generation. In order to increase the size of the plan graph, we increase the number of the design entities including plans, actions, percepts, goals and messages. For simplicity, rather than generating graph structures we generate *trees* (i.e., a single root and acyclic). However, for consistency we will continue to refer to them as plan graphs.

Plan graphs capture two control fragments: choice and parallel. It is common for plan graphs to capture both control types of fragments, so we defined the following systematic process to generate a variety of plan graphs while systematically increasing the scale of the graphs.

1. We started all plan graphs with one plan that is linked to a number of posted entities (parallelism)
2. The types of each branch (choice or parallelism) was the opposite of its parent.
3. The number of nodes N_L at a level L is $N_L = N_{L-1} + c$, in which N_{L-1} is the number of nodes at the parent level, and c is a constant.
4. All nodes had one child, except the left most node, which had $c + 1$ children.

We created 21 plan graphs with different sizes, by varying the breadth and depth of the graph as illustrated in Table 2. The column "Breadth" refers to the constant c in the formula above, "Depth" refers to the total number of "plan node" levels, which are the executable constructs (i.e. plans), and "#Paths" refers to the number of paths in the plan graph, not the corresponding reachability graph (see Figure 7 for case 2 as an example).

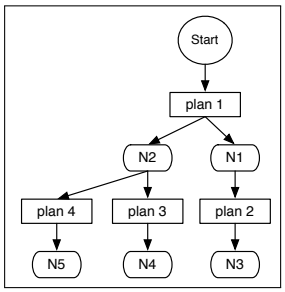


Figure 7: Plan graph for case 2 in Table 2

Table 2: Structures of the plan graphs used in the experiments

	Breadth	Depth	#Paths	#Plans	#Nodes
Case 1	2	1	0	1	2
Case 2	2	2	2	4	5
Case 3	2	3	4	8	9
Case 4	2	4	6	13	14
Case 5	2	5	8	19	20
Case 6	2	6	10	26	27
Case 7	2	7	12	34	35
Case 8	2	8	14	43	44
Case 9	2	9	16	53	54
Case 10	2	10	18	64	65
Case 11	3	1	0	1	3
Case 12	3	2	3	6	8
Case 13	3	3	6	13	15
Case 14	3	4	9	22	24
Case 15	3	5	12	33	35
Case 16	3	6	15	46	48
Case 17	3	7	18	61	63
Case 18	4	1	0	1	4
Case 19	4	2	4	8	11
Case 20	4	3	8	18	21
Case 21	4	4	12	31	34

As is shown in Table 2, by varying the size of a plan graph, we vary the size of the agent design. For instance in case 3, the plan graph is of breadth 2 and depth 3 which results in a design with 8 plans and 9 other nodes (actions, percepts, goals and messages). In case 10, with depth 10, the design had 64 plans and 65 other nodes.

We incremented the breadth and depth by 1 for each case, omitting cases for which execution took more than one day (e.g. breadth 3, depth 8, and breadth 4, depth 5).

Execution environment. We ran all the experiments on a desktop running a 64-bit Intel core i7 processor clocked at 3.4 GHz. 1 GB of RAM is dedicated to be used by the Java Virtual Machine. We ran no other tasks on the machine. We ran each case 6 times, and took the average time for both extracting and executing all traces.

4.1.2 Results

Table 3 shows our findings. We see that the number of traces grows exponentially as the plan graph increases in size, as expected. Similarly, we see that the extraction time is proportionally to the number of traces. However, the execution time, which also grows proportionally to the number of traces, is significantly lower than the extraction time.

Table 3: Summary of the scalability analysis results (B: breadth, D: Depth, #Traces: number of traces generated, Extraction: time taken in seconds to extract all traces, Execution: time taken in seconds to run all traces against the requirements Petri net

	B	D	#Traces	Extraction	Execution
Case 1	2	1	2	0.000	0.000
Case 2	2	2	12	0.001	0.000
Case 3	2	3	60	0.002	0.000
Case 4	2	4	280	0.012	0.002
Case 5	2	5	1260	0.034	0.005
Case 6	2	6	5544	0.095	0.031
Case 7	2	7	24024	0.167	0.040
Case 8	2	8	102960	0.524	0.050
Case 9	2	9	437580	2.221	0.104
Case 10	2	10	1847560	3.870	0.260
Case 11	3	1	6	0.000	0.000
Case 12	3	2	270	0.010	0.000
Case 13	3	3	8400	0.110	0.030
Case 14	3	4	242550	1.272	0.720
Case 15	3	5	6810804	36.500	1.060
Case 16	3	6	188684496	1129.600	26.401
Case 17	3	7	5187948480	60982.540	172.421
Case 18	4	1	24	0.001	0.000
Case 19	4	2	10080	0.090	0.040
Case 20	4	3	2587200	13.500	0.380
Case 21	4	4	630630000	4603.390	94.330

Figure 8a plots the extraction time of all the cases listed in Table 3. Note the Y-axis is logarithmic. The exponential nature of this problem is hardly surprising. What we consider significant is the number of traces that can be analysed within a reasonable time (in our case, within one day). For instance, the time taken to extract over 5 billions traces in case17 took around 17 hours. However, as Table 2 shows, the equivalent design to the plan graph of case 17 consists of 61 plans and 63 other nodes (actions, percepts, goals and messages). Further, these nodes include both parallelism and choice decompositions.

Whilst the plan graphs analysed were generated artificially, we believe that this shows our approach can, at least under some circumstances, scale up relatively well. It is of course impossible to know what sizes will occur in practice without a real design of some significant size. Based on our own experience over the last 15 years of building agent systems in collaboration with industry partners, case 15 (33 plans, 35 nodes) is quite large and this system took just 38 seconds to check.

4.2 Effectiveness Evaluation

In this section, we present a preliminary evaluation to validate that our approach is able to detect defects in a design, and to learn about the types of problems it cannot detect.

4.2.1 Method

Two participants from outside our research team, both experienced in BDI agent design, were given the corrected version of the plan graph in Figure 6a. Each participant was asked to make several small changes (called *mutants*), such as adding, removing, replacing and renaming entities. In total, we received 11 mutants. We ran our approach on each mutated version and investigated whether the introduced defects were detected or not.

4.2.2 Results

The resulting mutants all fell into three categories (although more are possible). First, the addition of new (irrelevant) entities *(ADD)*. For instance, one participant added a new goal in between the *Assign Reviewers* and *Collect Prefs* steps. Further, they made the plan that posts the "Assign Reviewers" goal posts a new goal "Optimise Allocation". Then, they added new plan to handle the "Optimise Allocation" goal that posts the "Assign Reviewers" goal.

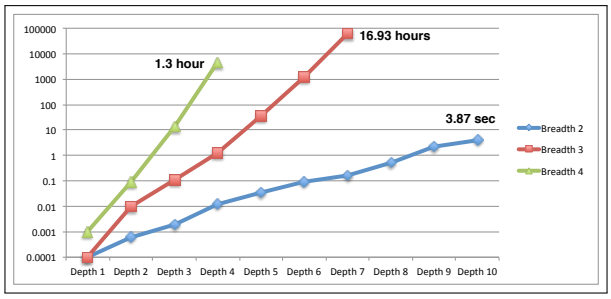
Second, the removal (deletion) of entities *(DEL)* from the plan graph including renaming them inconsistently with respect to the scenario. Third, the replacement of goal steps *(REP)* with related goals from the goal tree (parents or children).

Table 4 summarises the results obtained by applying our approach on the 11 mutants. Only one mutant (MA0) out of the five in the *ADD* category was detected. This is because a new goal was introduced that created a new step in the scenario, which was not originally specified. The remaining four added new functionality in the design that was filtered because it did not occur in the scenario (recall that traces have elements that do not occur in the scenario filtered out). This filtering is motivated by the possibility that a plan graph may include behaviour that is related to another scenario. However, the filtering does limit our ability to detect issues caused by adding elements to the design. Overall, we argue that this limitation may be acceptable in order to avoid too many false positives. However, in future work, we will investigate how to consider all scenarios in a design together to eliminate this issue.

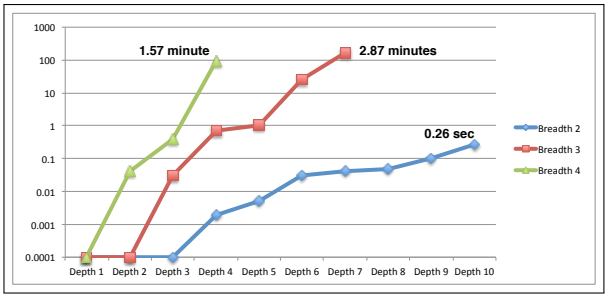
Table 4: Summary of potential problems raised by the plan graph mutation per mutants category (ID: mutant ID,MS: missing steps, TS: trace too short, TL: trace too long, OE: misordering between steps; Def.: modified design contains a defect with respect to the scenario; Det.: the defect is detected.)

Category	**ID**	**MS**	**TS**	**TL**	**OE**	**Def.**	**Det.**
ADD	MA0	0	0	1	0	✓	✓
	MA1	0	0	0	0	✓	✗
	MA2	0	0	0	0	✓	✗
	MA3	0	0	0	0	✓	✗
	MA4	0	0	0	0	✓	✗
DEL	MR0	0	1	0	0	✓	✓
	MR1	0	4	0	0	✓	✓
	MR2	0	2	0	0	✓	✓
	MR3	0	0	0	0	✗	✗
REP	MP0	0	0	0	0	✗	✗
	MP1	0	0	0	0	✗	✗
Total	11	0	7	1	0	8	4

In the *DEL* category, three of the four mutants were detected. In the undetected one, the participant modified the plan graph such that the goal *Assign Reviewer* was handled by one plan instead of



(a) Extraction Time



(b) Execution Time

Figure 8: Time taken to extract traces for all the cases in Table 3

two. This simply reduced the number of (correct) traces, and hence the design is still correct with respect to the scenario.

None of the *REP* mutants were discovered. This is because goals in the design were replaced with child/parent goals from the goal hierarchy that were valid for the given scenario. Therefore, the mutants were not defective.

This shows that our approach is promising, despite the small numbers involved. In particular, if we consider our technique as a method of classifying mutants as correct or incorrect, we correctly identified 8 out of 12 mutants as either defective or correct. As mentioned above, the 4 incorrectly classified mutants were due to the filtering process, which is the subject of future work. It is also encouraging to note that there were no false positives, i.e., all detected defects corresponded to actual design defects.

5. RELATED WORK

In the context of agent design methodologies, there has been little research into checking the correctness of agent design artifacts. There is a large body of work on verifying BDI agent systems using formal verification such as model checking (e.g. [10, 5, 6, 12]) and theorem proving (e.g. [28]). While these approaches can provide a higher level of assurance than our work, they assume the existence of either a formal model that abstracts the behaviour of the system, or an implementation of the system itself. In contrast, we aim to detect defects in semi-formal models.

Similarly, researchers have investigated specific methods for runtime testing of multi-agent programs [19, 20], including BDI agents [23, 16], assertion-based verification with static analysis [29], and run-time debugging of agent interaction [7, 25]. Clearly, testing or verifying an implementation also provides some level of verification of the corresponding design, however, there is clear value in being able to detect design defects before implementing a design.

Many AOSE methodologies offer development environments via their supporting tools, including frameworks for verification of design artifacts. Tropos [8] provides perhaps the most complete support for requirements verification. Tropos offers two frameworks, each including tool support, for: (i) validating formal requirements specification using *T-Tool* [14]; and (ii) reasoning with formal goal models using *GR-Tool* [15]. Using symbolic reasoning, these frameworks provide a high-level of assurance, but neither support verification of agent design artifacts. Both the O-MaSE [11] and Prometheus methodologies [24] offer support for agent design artifacts. O-MaSE offers the *agentTool III* (aT^3), which checks for consistency between the related models using results that check static relationships between design entities, while early versions of the Prometheus Design Tool offered basic static consistency checking feature. Such ideas are complementary to our approach.

Further, there has been some related work in the area of traceability for supporting verification of agent models. Cysneiros and Zisman [9] introduce a rule-based traceability approach grounded in the Prometheus methodology and JACK platform. The approach automatically generates traceability relations for verifying the models (agent designs and JACK code), and checking their completeness by identifying missing elements. Thangarajah *et al.* [30] propose a mechanism that facilitates the traceability between requirements (via scenarios) and other design entities, which is used to support test case generation. They establish the concept of *traceability links*, which relate scenarios steps to other design entities including goals, actions, percepts and events. Their work is complementary to our approach.

6. CONCLUSION

In this paper, we presented a method, with tool support, that is able to find defects in agent designs with respect to the requirements that are specified in the form of scenarios and goal hierarchies. The tool extracts all possible execution traces from the detailed agent designs, which represent the behaviour of the system, and checks each trace against the requirements models.

We evaluated our approach on 11 versions of one design (via mutations) that were done by two participants. Our evaluation shows that our approach was able to successfully detect defects, and raised no false positives. In addition, we performed a scalability analysis on the approach, with results showing that our approach can verify large plan graphs in under 24 hours, despite the exponential explosion in traces as designs get larger.

Our approach is grounded in the Prometheus design methodology, however it can be generalised to others. This generalisation is possible due to the intermediate representations that we use. More specifically, both the requirement specifications and the detailed agent designs are translated to executable Petri net structures. The latter is used to extract traces and the former to execute the traces extracted. Thus, applying our techniques to other methodologies would simply involve transforming their requirements and detailed designs into equivalent Petri net structures.

In future work, we plan to improve the approach to consider multiple scenarios at once, which will mitigate the problem of filtering out unrelated steps that led to some defects not being detected in our evaluation. We will also apply our approach to an extensive set of Prometheus case studies to determine its effectiveness at finding defects in practice.

REFERENCES

[1] Y. Abushark, J. Thangarajah, T. Miller, and J. Harland. Checking consistency of agent designs against interaction protocols for early-phase defect location. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14*, pages 933–940, Paris, France, May 2014.

[2] Y. Abushark, M. Winikoff, T. Miller, J. Harland, and J. Thangarajah. Checking the correctness of agent designs against model-based requirements. In *Proceedings. European Conference on Artificial Intelligence*, pages pp. 953–954, 2014.

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

[4] 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.

[5] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416. ACM, 2003.

[6] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[7] J. Botía, A. López-Acosta, and A. Skarmeta. ACLAnalyser: A tool for debugging multi-agent systems. In *Proceedings. European Conference on Artificial Intelligence*, pages pp. 967–968, 2004.

[8] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[9] G. Cysneiros and A. Zisman. Traceability and completeness checking for agent-oriented systems. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 71–77. ACM, 2008.

[10] M. Dastani, K. Hindriks, and J. Meyer, editors. *Specification and Verification of Multi-agent systems*. Springer, Berlin/Heidelberg, 2010.

[11] S. A. DeLoach and J. C. Garcia-Ojeda. O-MaSE: a customisable approach to designing and building complex, adaptive multi-agent systems. *Agent Oriented Software Engineering*, 4(3):244–280, 2010.

[12] L. Dennis, M. Fisher, M. Webster, and R. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[13] R. Evertsz, J. Thangarajah, and T. Ly. An agent oriented software engineering application for modelling military tactics. In *AAMAS'14*, 2014. (To appear).

[14] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 174–181. IEEE, 2001.

[15] P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159–171, 2005.

[16] T. Miller, L. Padgham, and J. Thangarajah. Test coverage criteria for agent interaction testing. In *Agent Oriented Software Engineering XI*, volume 6788 of *LNCS*, pages 91–105. Springer, 2011.

[17] S. Munroe, T. Miller, R. Belecheanu, M. Pechoucek, P. McBurney, and M. Luck. Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. *Knowledge engineering review*, 21(4):345, 2006.

[18] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[19] C. D. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. *AAMAS*, 25(2):260–283, 2012.

[20] C. D. Nguyen, A. Perini, and P. Tonella. A goal-oriented software testing methodology. In *Agent-Oriented Software Engineering VIII*, pages 58–72. Springer, 2008.

[21] L. Padgham, J. Thangarajah, and M. Winikoff. Tool support for agent development using the Prometheus methodology. In *International Conference on Quality Software*, pages 383–388. IEEE, 2005.

[22] L. Padgham, J. Thangarajah, and M. Winikoff. The Prometheus design tool – a conference management system case study. In *Agent Oriented Software Engineering VIII*, volume 4951 of *LNCS*, pages 197–211. Springer, 2008.

[23] L. Padgham, J. Thangarajah, Z. Zhang, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39(9):1230–1244, 2013.

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

[25] L. Padgham, M. Winikoff, and D. Poutakidis. Adding debugging support to the Prometheus methodology. *Journal of Engineering Applications in Artificial Intelligence*, 18(2), March 2005.

[26] J. Pavón and J. Gómez-Sanz. Agent oriented software engineering with INGENIAS. In *Multi-Agent Systems and Applications III*, volume 2691 of *LNCS*, pages 394–403. Springer, 2003.

[27] A. S. Rao, M. P. Georgeff, et al. Bdi agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[28] S. Shapiro, Y. Lespérance, and H. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *AAMAS'02*, pages 19–26, 2002.

[29] J. Sudeikat, L. Braubach, A. Pokahr, W. Lamersdorf, and W. Renz. Validation of BDI agents. In *Programming Multi-Agent Systems*, pages 185–200. Springer, 2007.

[30] J. Thangarajah, G. Jayatilleke, and L. Padgham. Scenarios for system requirements traceability and testing. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 285–292. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[31] M. Winikoff and L. Padgham. Agent Oriented Software Engineering. In G. Weiß, editor, *Multiagent Systems*, chapter 15. MIT Press, 2 edition, 2013.