Checking The Correctness of Agent Designs Against Model-Based Requirements

Yoosef Abushark\textsuperscript{1} and Michael Winikoff\textsuperscript{2} and Tim Miller\textsuperscript{3} and James Harland\textsuperscript{1} and John Thangarajah\textsuperscript{1}

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.

1 Introduction

Autonomous agents are widely-used for developing systems that are highly dynamic in nature in a broad range of domains [6]. While there are many agent development paradigms, the Belief-Desire-Intention (BDI) model [12] is a mature paradigm that has been adopted by several agent development platforms. Most existing work on verifying BDI agent systems has focused on formal verification ([3]), particularly using model checking techniques ([4]) and theorem proving ([13]), or on runtime testing [10]. Such work tends to focus on the verification of complete agent programs, requiring source code. However, it is long established that early detection and resolution of software defects saves time and money [2, Page 1466].

Our aim therefore is to develop a suite of lightweight techniques, supported by tools, for detecting defects at the design phase, prior to implementation.

We have recently developed techniques for checking the functional correctness of agent-based designs with respect to communication protocol models [1]. The novelty of our approach is that it requires only design-phase models, and no source code. In this paper, we build on this previous work with a technique for checking the functional correctness of agent designs with respect to requirements models.

In this work, we propose a mechanism, grounded in the Prometheus methodology [11, 8], for checking that the detailed plan structures conform to the requirements specified in terms of scenarios and goals. We believe our approach is generalisable to other BDI methodologies (see [14] for a recent survey).

We use the conference management system [9] as an 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 for 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.

Requirements specifications in agent-oriented software engineering [14, Section 4] generally include 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, a percept, a goal to achieve, or a sub-scenario. 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 of a multi-agent system designed to meet these requirements.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Type & Name & Role \\
\hline
1 & Percept & Review Phase \\
2 & Goal & Involve Reviewers \\
3 & Action & Send Invitations \\
4 & Percept & Reviewers Preferences \\
5 & Goal & Collect Pref \\
6 & Goal & Assign Reviewers \\
7 & Action & Give Assignments \\
8 & Percept & Review Report \\
9 & Goal & Collect Reviews \\
10 & Goal & Get PC Opinions \\
\hline
\end{tabular}
\caption{Review Scenario Description}
\end{table}

2 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 a Petri net (via a plan graph). We then verify that all traces of the design Petri net are valid with respect to the scenario Petri net.

Our process has three steps:

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. Although a scenario is a single sequence of steps, the result

\footnotesize
\textsuperscript{1} RMIT University, Melbourne, Australia.
\textsuperscript{2} University of Otago, New Zealand.
\textsuperscript{3} University of Melbourne, Australia.
is a set of sequences, because a goal step can be realised in the
detailed design by implementing its descendants, or its parent.
2. Extracting all possible execution traces from agent designs: We
create a plan graph [5] 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. 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
to, we translate the plan graph into a Petri net, and then use the
standard reachability graph construction [7] to obtain all possible
traversals of the plan graph.
3. Executing design traces against the Petri nets: We check each of
the possible traces of the plan graph’s Petri net \( N_P \) against the
scenario Petri net \( N_S \), and note any discrepancies.

3 Example

Consider Figure 2a, which contains intentionally seeded four de-
defects: a plan (Invite Reviewers) that posts multiple goals (non-
determinism), resulting in traces with wrong ordering of steps; 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 miss-
ing step and two incomplete paths. For instance, Review Phase, In-
vite_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 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.

We have implemented the proposed approach as an eclipse plug-in
that integrates with the Prometheus Design Tool (PDT).

REFERENCES
tency of agent designs against interaction protocols for early-phase
agent interaction testing. In AOSE XI, volume 6788 of LNCS, pages
[6] 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 engi-
development using the Prometheus methodology. In International Con-
tool – a conference management system case study. In AOSE VIII,
oracle generation for automated unit testing of agent systems. IEEE
ification language and verification environment for multiagent systems.
In AAMAS’02, pages 19–26, 2002.
G. Weiß, editor, Multiagent Systems, chapter 15. MIT Press, 2 edition,
2013.