Checking Consistency of Agent Designs Against Interaction Protocols for Early-Phase Defect Location

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.au

ABSTRACT

Multi-agent systems are increasingly being used in complex applications due to features such as autonomy, proactivity, flexibility, robustness and social ability. However, these very features also make verifying multi-agent systems a challenging task. In this paper, we propose a mechanism, including automated tool support, for early phase defect detection by comparing agent interaction specifications with the detailed design of the agents participating in the interactions. The basic intuition of our approach is to extract sets of possible traces from the agent design and to verify whether these traces conform to the protocol specifications. Our approach is based on the Prometheus agent design methodology but is applicable to other similar methodologies. Our initial evaluation shows that even simple protocols developed by relatively experienced developers are prone to defects, and our approach is successful in uncovering some of these defects.

Categories and Subject Descriptors

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

Keywords

AOSE; verification; multi-agent systems

1. INTRODUCTION

Multi-agent systems are gaining popularity for building complex applications ranging from critical systems used in crisis management, to non-critical systems such as information exchange [12]. Developing multi-agent systems is challenging, as these systems are often required to operate under conditions not conceived by their designers. Several architectures have been proposed to support the building of multi-agent systems, including the popular Belief-Desire-Intention (BDI) agent architecture [20]. In addition, several agent-oriented software engineering (AOSE) methodologies and notations, such as Prometheus [17], Tropos [2], O-MaSE [6], ROADMAP [21], and GAIA [23], have been proposed to help organise the development activities of multi-agent systems. While these methodologies differ in many ways, they are common in their inclusion of a number of activities, namely: analysis, design, implementation, and testing.

Recent work related to these methodologies has proposed several testing and debugging techniques [5, 14, 16]. While testing plays a crucial role in software verification, it is well accepted in software engineering that defects found and fixed late in a project cost considerably more than those found earlier [19]. Thus, the ability of find defects in design artefacts before implementation commences will positively impact the cost of developing multi-agent systems.

In this paper, we focus on one aspect of design-time verification of BDI agent designs: agent interaction protocols. An interaction protocol strictly defines the way in which agents ought to communicate in order to accomplish their own objectives in a joint setting. For example, the way buyers and sellers should interact in an auction setting.

We describe a method with automated tool support for checking agent designs against the specification of the interaction protocols. This consists of checking execution traces of the designs to ensure that they are consistent with the protocol specification. A failure to do so may indicate a defect in either the protocol specification or the agent design. This method uses only design-level artefacts, so can be used before any implementation is produced. However, this means the main technical challenge is to find an appropriate mechanism for evaluating the computational properties of the design artefacts. We do this by translating both the protocol specification and the design artefacts into Petri Nets [13], and checking traces of the execution of the Petri Nets for consistency.

We have implemented this method and developed tool support for automating the consistency checks. Our tools are based on the Prometheus AOSE methodology [17], however, many of the concepts and tools used are general enough to be used with any AOSE methodology that supports the BDI model of agency.

We evaluate our method and tool support on two interaction protocols with multiple designs. These designs were developed by relatively experienced developers and based

[*]Acknowledges King Abdulaziz University for scholarship.

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

on the Prometheus methodology. Our results show that the approach is successful in revealing defects in agent designs with respect to the interaction protocol specification, however, some of these defects were false positives.

In Section 2, we briefly highlight the modelling of interaction protocols in some AOSE methodologies, and mention how they each support checking design artefacts. We present our method in Section 3, describe our evaluation in Section 4, and report results in Section 5. Section 6 concludes the paper.

2. BACKGROUND & RELATED WORK

This section focuses on briefly describing and explaining the topics related to this paper. We start by describing the AUML (Agent UML) sequence diagrams [15], which are used to specify agent interactions by some of the more popular AOSE methodologies, including Prometheus, Tropos, O-MaSE, INGENIAS and GAIA. Even though GAIA does not use AUML sequence diagrams for modelling interaction protocols directly, there are many proposals for integrating it. We also provide an overview of these methodologies with respect to their support for checking design artefacts.

2.1 Modelling Interaction Protocols

AUML sequence diagrams, also called protocol diagrams, illustrated in Figure 1, specify the allowable sequences of messages between agents. Specifications consist of vertical lines identifying agents, horizontal directed arrows representing messages, and boxes representing a range of control constructs. Some of the more common control structures are alternative (alt, as illustrated in Figure 1), optional (opt) and parallel (par) with the usual meanings. Dashed lines separate options in choice or parallel boxes. These control structures can also have guards, which describe a relevant condition such as alternatives or the termination condition for a loop. For a full specification of AUML protocols see Odell et al. [15].

In addition to the graphical representation, methodologies such as Prometheus adapts a textual notation [22] to generate the protocol diagram. This textual notation provides a handle to the various elements of the protocol that can be used computationally.

2.1.1 Net-Bill Protocol Example

The ‘Net-Bill’ protocol models transactions in the electronic commerce systems [4]. The protocol specifies the interaction between three agents, ‘Customer’, ‘Merchant’ and ‘Bank’ as shown in Figure 1. The customer optionally sends a request to the merchant at the start of the interaction. Then, the merchant responds by sending the ‘Quote’. After that, the customer may refuse or accept the quote. In the case where the customer accepts the quote through posting an ‘Accept’ message to the merchant, the delivering of the goods along with the payment transaction will take place. Figure 2 represents one possible design, specified using the Prometheus methodology, which satisfies the protocol. The figure shows the *detailed design* of the three agents, where the plans that handle incoming messages, and produce the outgoing messages are specified. Note that the messages are shown as events.

Despite this being a well-known protocol, it is surprisingly difficult to verify the correctness of a design based on this protocol by hand. As we shall see, our method is able to

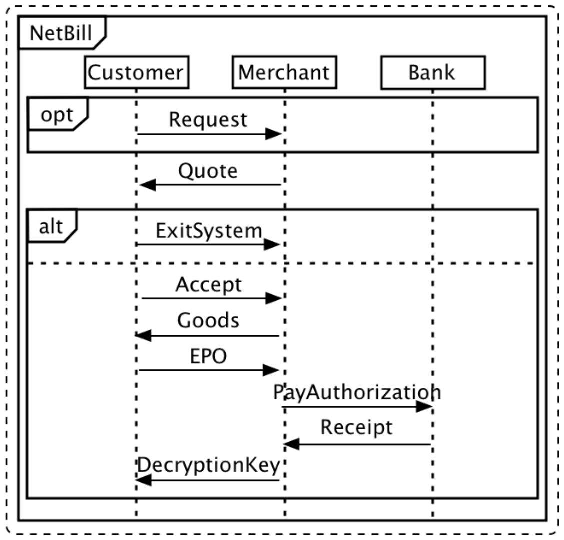


Figure 1: Net-Bill AUML protocol

detect certain types of defects in the designs based on this protocol, in particular when messages are sent in the wrong order, or when too many messages are sent.

2.2 AOSE Methodologies and Design Artefacts

In the context of AOSE, whilst testing [16, 14] and formal validation of agent programs [7] have received attention, there has been little work into checking the correctness of BDI designs in the design phase.

In this section, we investigate the support of checking the correctness of design artefacts in five of the most commonly used AOSE methodologies: Prometheus, Tropos, O-MaSE, INGENIAS and GAIA; although, only three provide support for verifying design artefacts to some limited extent.

Tropos offers two frameworks, each including tool support, for: (i) validating formal requirements specification (‘T-Tool’ [8]); and (ii) reasoning with formal goal models (‘GR-Tool’ [9]). These two frameworks verify the goal diagrams that are part of the analysis stage of Tropos. As far as we are aware, there is no support for validating that agent designs conform to the interaction protocol specifications.

The INGENIAS Development Kit (IDK) [10] integrates a tool called ACLAnalyser for analysing the interaction specifications between agents by executing the system and logging the run-time interactions [1]. Even though this gives valuable feedback about the interactions specified in the design phase, it requires an implementation.

O-MaSE offers an environment for developing agent-based systems through agentTool III (aT^3) [6]. aT^3 provides a verification framework that maintains the consistency between the related models. The check is done based on a set of the predefined rules that are specified by the designer.

The Prometheus and the GAIA methodologies provide agent engineers with a graphical environment for developing agent-based systems (PDT [17] and GAIA4E [3]). These tools do not offer any framework for verifying and validating the agent design artefacts, although earlier versions of PDT supported limited static consistency checking of the design for warnings such as events that are not handled, and messages that should be sent/received according to a protocol.

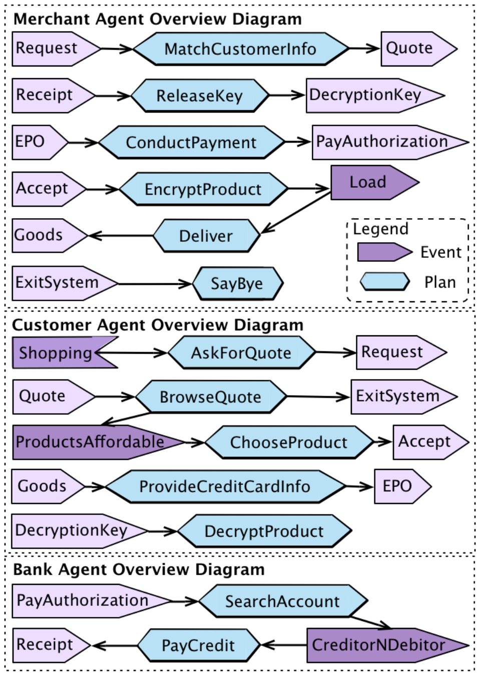


Figure 2: Net Bill detailed design in PDT

These checks were based purely on static relationships between the design entities. In contrast, our work assessed the dynamic behaviour of designs.

3. TECHNICAL APPROACH

In this section, we present our method for consistency checking between interaction protocols and agent designs, and briefly discuss tool support for automating this. We discuss the limitations of the method and propose ways to structure designs to mitigate these limitations.

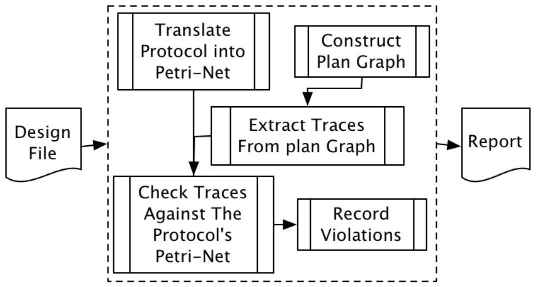
Our approach is to use Petri Nets, both as a means of analysing the specification of a given protocol, and as a means of determining the computational properties of a given design. Note that in principle a design may incorporate multiple protocols, or may be incomplete. Hence it may not always be possible to classify a given design as either correct or incorrect, but our analysis can find certain types of defects along the way. We use the previously described Net-Bill protocol as a running example in this section.

3.1 Overview

Figure 3 shows a high-level overview of the proposed method. The input to our method is a design file that includes two parts of the design:

1. protocols that model the interactions between agents; and
2. designs for agents in the system, which provide a semi-formal definition of the agents' plans, percepts (including incoming messages), actions (including outgoing messages), and causal relationships between these.

Each protocol is analysed and translated into a Petri Net [13]. Using the design artifacts, a *plan graph* is constructed, which outlines the causal relationships between plans and



Translate Protocol into Petri-Net
Construct Plan Graph
Design File
Extract Traces From plan Graph
Report
Check Traces Against The Protocol's Petri-Net
Record Violations

Figure 3: Process for our method

messages for all agents/roles involved in the protocol. The set of possible message traces are derived from the plan graph, and are dynamically run against the protocol's corresponding Petri Net, logging any violations.

3.2 Process

The intention of our approach is to find design defects, either in the agent detail design artifacts or in the protocols, by checking the consistency between the two. As shown in Figure 3, there are five steps involved in the method: (1) translate a protocol into a Petri Net representation, preserving its semantics; (2) construct a plan graph from the agent detailed design, including only those agents who are participants in the protocol; (3) extract all possible traces permitted by the plan graph; (4) check all traces against the protocol's Petri Net; and (5) record all violations.

3.2.1 *Translating AUML protocols to Petri Nets*

Our tool support is written for the Prometheus Design Tool, therefore, it assumes protocols are specified using AUML [15]. However, AUML does not have a precise semantics, so we choose to transform interaction protocols into a more general model, specifically Petri Nets [13], as they can capture the essential fragments of interactions such as selection, loops and parallelism. This generalises the proposed method to include other interaction modelling notations that can be translated into that general model.

As an example, Figure 4 depicts the corresponding Petri Net that models the semantics for the Net-Bill protocol described in Figure 1. We perform this transformation by directly adopting the AUML to Petri Net translation algorithm proposed by Poutakidis et al. [18].

3.2.2 *Constructing a plan graph from agent designs*

Interaction protocols define the possible interactions between agents within an agent-based system. Thus, each participant (agent) should be designed in a way that captures its role in a particular protocol. In the Prometheus methodology, protocols are reflected in the participants' detailed designs in terms of outgoing and incoming events that are associated with plans. According to the design of the Net-Bill system in Figure 2, the 'Customer' agent sends the 'Request' message using the 'AskForQuote' plan. On the receiver side, this message is handled by one plan 'MatchCustomerInfo' that posts the 'Quote' message.

As this example demonstrates, agent designs corresponding to a protocol are scattered across the multiple detailed designs of its participants. To generate the set of possible traces for a design over more than one agent, all design

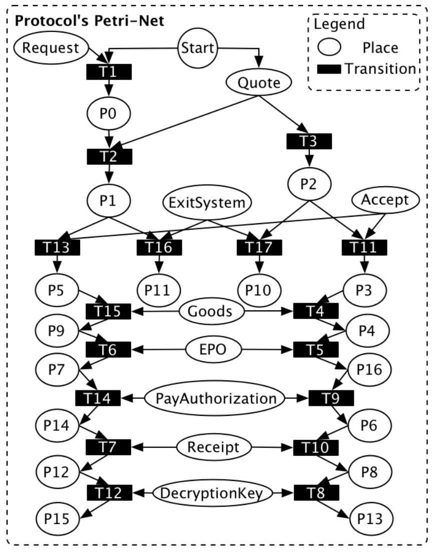


Figure 4: The Petri Net of the Net-Bill protocol

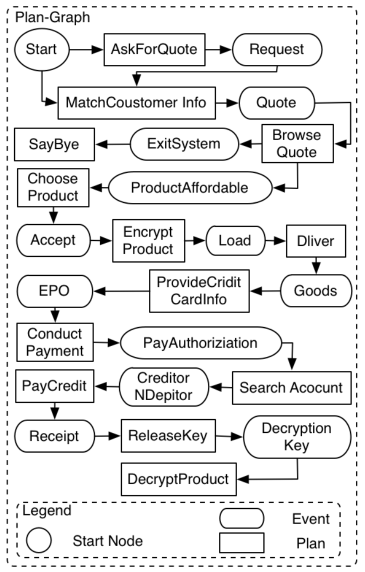


Figure 5: Plan graph for the Net-Bill design from Figure 2

constructs (plans and events) within these designs that are related to a protocol need to be located and organised into a single coherent artifact. Thus, following Miller et al. [11], we construct a *plan graph* for representing the information relevant to a given protocol.

Definition 1. *A* plan graph *is a directed bipartite graph, $G = (P, M, E)$, where P and M are the sets of plans and messages directly, and E the set of edges in the graph, which represent causal relationships between plans and messages. That is, the edge $p \mapsto m$ represents that plan p sends message m, and the edge $m \mapsto p$ represents that message m is a trigger for plan p.*

A plan graph for a given protocol and set of agent designs is constructed by extracting each plan that sends or receives the messages of the specified protocol, and the messages that are associated with these plans from the detailed designs. Since it is possible for a protocol to have more than one starting point (e.g. more than one plan can send the initial message, or there can be more than one possible initial message), we add a single dummy 'Start' message as the first node of the plan graph, which is linked to all nodes that represent the protocol's starting points.

Figure 5 shows the plan graph corresponding to the design of the Net-Bill system. This plan graph effectively merges the designs of the three agents in Figure 2, but includes only the information that is relevant to the specified protocol.

3.2.3 Extracting execution traces from a plan graph

The generated plan graph acts as a static view of the combined designs for the participants in the relevant protocol. As a result, each path of that plan graph represents a possible sequence of plans and corresponding messages of the protocol. Considering the plan graph in Figure 5, the trace 'Start, MatchCustomerInfo, Quote, BrowseQuote, ExitSystem, SayBye ' is one possible path of that plan graph, and represents an instance of dynamic behaviour of the system. After filtering out the plans from that execution trace, we are left with one possible trace of observable messages: 'Quote, ExistSystem'.

A plan graph may capture many execution flow fragments including: sequential composition, selection (choice), loops and parallel execution; each outlined in Figure 6. The parallel fragment is captured when one plan posts multiple messages or multiple plans run in parallel, whilst the selection fragment appears when a message can trigger more than one plan, but the execution semantics specify that exactly one plan will be selected. Due to the non-deterministic nature of BDI agent frameworks[1], plan graphs must be traversed to extract all possible traces.

In our method, we assume that parallelism represents non-deterministic interleaving of both plans and messages, as this would be the observable behaviour demonstrated by most BDI frameworks. Consequently, a simple depth-first-traversal is not sufficient to generate traces.

To extract traces, we translate our plan graphs into Marked Petri Nets [13]. We choose this formalism because Petri Nets are expressive enough to model the semantics of our plan graphs, and because our plan graphs are syntactically similar to Petri Nets. Further, we can make use of existing theories for analysing the generated model.

Definition 2. *A* marked Petri Net *is a tuple $M = \{P, T, I, O, \mu\}$, in which P is a finite set of places, T is a finite set of transitions, such as $P \cap T = \emptyset$, I is an input function, O an output function, and μ is the marking of the Petri Net defined as an n-vector, $\mu = \{\mu_1, \mu_2, \mu_3, ..., \mu_n\}$, in which $n = |P|$ and each $\mu_i \in N, i = 0, 1$.*

The transformation of a plan graph into a marked Petri Net is straightforward for most plan graph constructs: messages are translated into Petri Net places, and plans are translated into transitions. This simple mapping is sufficient for sequential composition, selection, and loops, but for parallelism, a synchronisation fragment is used to capture the interleaving parallelism in the cases where plans post multiple messages. Figure 7 shows an example of such a case.

[1]Plan choices are dependent on the context at the time of deliberation.

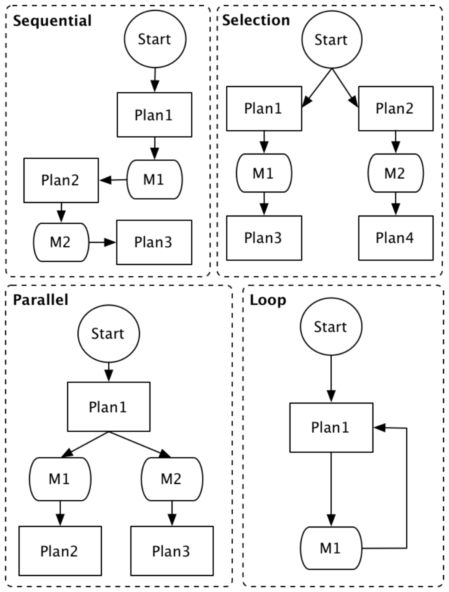


Figure 6: Plan graph execution flow fragments

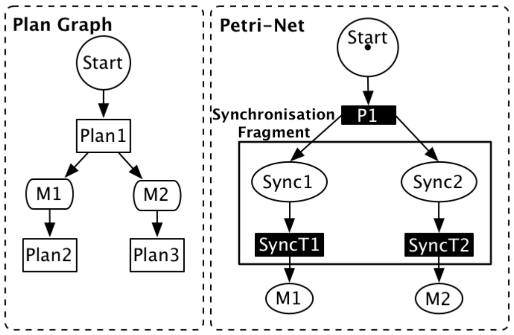


Figure 7: Plan graph with plan that posts multiple messages

If more than one plan can trigger the commencement of the protocol, a Petri Net for each starting point is created.

To extract the traces, we calculate the *reachability graph* of the Petri Net, which is a transition relation that defines the state and transitions of the Petri Net.

Each state within the reachability graph is in fact a *marking vector* that shows the distribution of tokens across the places (messages) of the Petri Net. Figure 8 shows the reachability graph for the Petri Net from Figure 7. When the reachability graph is in state S0, this represents the marking $\{1, 0, 0, 0, 0\}$. Using the legend at the bottom of Figure 8, this means that a token is in the Petri Net place 'Start', because the first slot in the marking is 1, while the remainder are 0. When the graph moves to state S1, there are tokens in the Petri Net places 'Sync1' and 'Sync2'. State S2 represents the state in which there are tokens in 'Sync1' and 'M2', indicating that message 'M2' has been sent. By analysing the changes in markings between transitions, we can represent which messages are sent and in which order, as defined in the plan graph.

Each path of the reachability graph represents a possible trace of the Petri Net. From the reachability graph, we can use a standard depth-first traversal to extract the set of all possible traces defined by the Petri Net (and therefore its corresponding plan graph). As a final step, the places added in the synchronisation fragment, such as 'Sync1' and 'Sync2' in Figure 7, must be filtered from the traces.

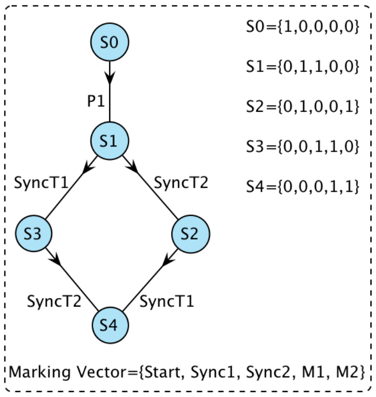


Figure 8: Reachability Graph of the Petri Net in Figure 7

3.2.4 *Running traces against the protocol*

Finally, each possible trace is verified by checking that it is a valid execution trace of the protocol's Petri Net (note: not the plan graph's Petri Net), using the messages from the trace as tokens on the protocol Petri Net.

As an example, consider the protocol Petri Net from Figure 4. At the start, there is a token in the 'Start' place. The first message is taken from the trace, and a token is put into the Petri Net place that corresponds to that message. If the first message is 'Request', then a token is placed here, and the Petri Net is executed. The transition $T1$ can fire because both of its input places contain tokens, and the result is a token in place $P0$. The process then repeats for the next message in the trace. An algorithm for performing this check is shown in Algorithm 1.

Algorithm 1 The algorithm for verifying traces against a Petri Net

```
Require: traces (a set of sequences of message labels)
Require: petri_net (a Petri Net representing a protocol)
  while $traces \neq \emptyset$ do
    choose a $trace \in traces$
    while $trace \neq \langle\rangle$ do
      $next\_message \leftarrow head(trace)$
      place next_message in petri_net
      if $petri\_net.execute = fail$ then
        record the fault
      end if
      $trace \leftarrow tail(trace)$
    end while
    remove trace from traces
  end while
```

When the Petri Net fails to execute, it indicates an inconsistency between the trace and the Petri Net; representing an inconsistency between the agent design and the protocol. For example, if the first message in the trace had been 'ExitSystem', a token would be placed on the 'ExitSystem' place, meaning that the Petri Net could not be executed because no transition has tokens in all of its input places.

Table 1: Categorisation of causes for failures

	Failure (executing protocol 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 less messages than it should, relative to the protocol.
2	The Petri Net has terminated, but the remainder of the trace is non-empty.	2. The trace contains more messages than it should, relative to the protocol.
3	A token is placed into the Petri Net, but the Petri Net cannot be executed.	3.(a) The message that needs to be sent is missing in the trace; or 3.(b) Ordering between messages within the trace is not as it should be.

3.2.5 Recording violations

A trace is recorded as "passed" if and only if the execution of the Petri Net hits a termination place and the remaining trace is empty. Otherwise, it is recorded as failed. To improve reporting, a cause for a failure can be used to provide debugging information. There are four broad reasons why failures occur, outlined in Table 1.

3.3 Implementation

We have implemented an eclipse plug-in that integrates with the Prometheus Design Tool (PDT) to automate our approach. The tool takes the PDT's design file (including at least the AUML protocols and agent detailed designs) as an XML file, generates the protocol Petri Net and plan graph, extracts the traces from the plan graph, executes these traces against the protocol Petri Net, recording and categorising all failures. The output is a report that provides: (1) detailed logs of the erroneous traces, including their categorisation (Table 1); and (2) an execution summary. The summary includes information such as the execution time and the number of traces that passed and those that failed.

3.4 Discussion

The method presented in this section checks the consistency between design artefacts to flag potential defects. However, given the partial nature of the agent detailed designs used in Prometheus and other semi-formal AOSE methodologies, neither soundness nor completeness are possible.

Specifically, our method may raise false positives due to the underspecification of the designs. Given an agent plan that posts several messages, our method assumes that all messages are posted by the plan. However, a designer may intend only some of these messages to be posted for any single execution of the plan, depending on some logic internal to the plan. In Prometheus, such logic is not captured at the design level. As such, the set of possible traces generated by our method could be larger than the set of traces intended by the designer, and some of the additional traces may violate the corresponding protocol.

With regards to the categorisation in Table 1, some causes can result in false positives, but others will not. Causes 1 and 3(a) (short trace and missing message) will always be true positives, because the partial nature of the designs will result in more traces than may be intended, but never less. Additional traces result from designs in which plans post multiple messages, but the designer intended only some of these to be posted at any time. Our method assumes that all messages must be posted, which will not result in shorter traces or missing messages. On the other hand, causes 2 and 3(b) in Table 1 can result from underspecification, so may contain false positives.

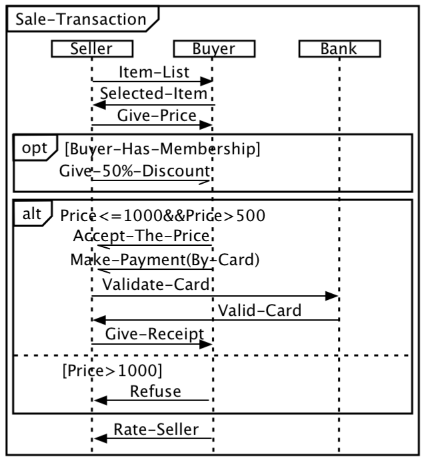


Figure 9: Sale Transaction AUML Diagram

To avoid false positives, agent detailed designs can be structured such that a plan intended to post only a subset of its specified messages is broken into sub-plans, in which all messages specific are intended to be posted, thus providing a deterministic way to calculate the messages that will be sent by each plan.

4. EVALUATION

This section details the empirical evaluation of our mechanism. The goals of this evaluation are:

1. to assess whether the proposed mechanism is able to detect defects in agent designs with respect to interaction protocols;
2. to determine the level of false positives generated by the proposed approach; and
3. to determine whether the time taken to run the tools is reasonable considering the complexity of the protocol and the design of its participants.

4.1 Objects of analysis

Two interaction protocols were used in this evaluation; namely, 'Sale-Transaction' and the 'Net-Bill' protocol. We investigated three designs for the 'Sale-Transaction' protocol and one for the 'Net-Bill'. Each design was produced by a person outside of our research team who was familiar with BDI modelling and the Prometheus methodology.

Table 2: Number of design units

Design	#Plans	#Messages	Total
ST1	17	27	44
ST2	11	17	28
ST3	9	17	26
NB	13	22	35

Table 3: Categorisation of defects over all iterations

Design	True Pos	False Pos	Unknown	Total
ST1	3	0	2	5
ST2	2	0	2	4
ST3	4	0	0	4
NB	0	1	0	1
Total	9	1	4	14

Figure 9 shows the AUML protocol for the 'Sale-Transaction' system, which was designed by one of the authors. The corresponding system models an online store as a multi-agent system with three agents ('Seller Agent', 'Buyer Agent' and 'Bank Agent') that interact with each other. We asked three participants to complete detailed designs for all the agents involved in the protocol, resulting in three different designs.

A complete design of the agent system following the 'Net-Bill' protocol (see Section 2.1.1) was produced by a research programmer with extensive experience in the Prometheus methodology. Details about the number of plans and messages in these designs are summarised in Table 2.

4.2 Experimental Process

After manually checking the consistency of the message names against the messages of the protocol, we followed an iterative process for checking each design. This iterative process involves the following three steps for each design:

1. *Execution:* We ran our tools over the PDT design file (including both the protocols and agent design) to produce a report. We used a laptop running a 64-bit Intel core i7 processor clocked at 2.4 GHz. 3 GB of RAM is dedicated to be used by the Java Virtual Machine.
2. *Inspection:* Using the reports generated, we analysed the causes of failed traces, categorising each cause as either a true positive, a false positive, or unknown. A false positive is a warning raised that we believe is not a defect in the design, but is caused by a clear and valid assumption made by the designer. An unknown categorisation implies that the design produces traces that fail, perhaps because the designer made explicit design assumptions that we do not understand.
3. *Modification:* Some defects mask the presence of other defects. As such, we modified the design in a way to rectify the causes of the reported failures, including false positives and unknowns. The changes involve adding, removing, and modifying plans, events, and associations.

We iterated through each design until our tool reported no problems. In each iteration, we recorded the following: (1) the number of warnings raised by the tool; (2) the number of true positive, false positive, and unknown defects; (3) the time taken for extracting the traces from the reachability graph; and (4) the number of the extracted traces.

Table 4: Summary of potential problems raised by the tool per iteration (Iter: Iteration, MM: missing messages, TS: trace too short, TL: trace too long, OE: misordering between messages)

Design	Iter	MM	TS	TL	OE
ST1	1	3	0	0	0
	2	0	0	0	2
	3	0	0	0	0
ST2	1	1	0	2	13
	2	1	0	0	1
	3	0	0	0	2
	4	0	0	0	0
ST3	1	0	4	0	0
	2	0	0	0	0
NB	1	0	0	4	5
	2	0	0	0	0

Table 5: Time Analysis of Traces

Design	Iter	Traces	Extract Time	Exec Time	# Failed Traces
ST1	1	4	0.33s	0.01s	4
	2	6	0.36s	0.02s	4
	3	4	0.4s	0.03s	0
ST2	1	154440	6.2s	335m	154440
	2	14	0.32s	0.09s	12
	3	14	0.32s	0.09s	10
	4	4	0.48s	0.04s	0
ST3	1	4	0.36s	0.02s	4
	2	4	0.4s	0.03s	0
NB	1	20	0.30s	0.06s	20
	2	4	0.32s	0.02s	0

5. RESULTS

This section presents the results of the empirical evaluation outlined in Section 4. Following the iterative process described we were able to identify and categorise the total number of defects found (true positives), the number of false positive defects found, and the number of potential defects that could not be accurately categorised for each of the designs. The findings are shown in Table 3. As the table shows, we were able to detect nine defects in four different designs with only one false positive and four that we were unable to categorise. Although a more comprehensive evaluation is necessary to provide concrete claims, this preliminary set of experiments provide an indication that the tool does indeed detect defects in agent designs with respect to interaction protocols, with few false positives and in a reasonable amount of time as we show ahead.

Table 4 shows information about the potential problems raised by the tool in each iteration of each design. As this table demonstrates, some defects manifest themselves as more than one reported problem. At the end of each iteration, a known category of problems were addressed, for example in 'Sale Transaction 1', the missing messages were fixed after iteration one. This fix may cause the number of defects found in the next iteration to decrease (defects are removed), or increase (a defect is removed but it was masking others).

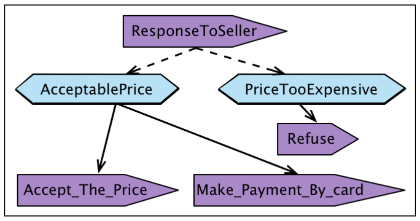


Figure 10: Buyer Agent Overview Diagram

For example, in 'Sale Transaction 2', the correction of the missing message and the two long traces from the first iteration revealed an additional missing message in the next iteration, but also removed 12 of the 13 message ordering related warning messages.

As mentioned in Section 4.2, there will be some failures that do not represent defects in the agent designs. In 'Sale Transaction 2', there are still two problems raised in the third iteration (Table 4) that are categorised as unknown. Figure 10 shows one of these. The participant designed both messages 'Accept_The_Price' and 'Make_Payment_By_Card' to be sent by one plan. Our approach considered that these messages could be sent in any order. However, 'Accept_The_Price' must be sent first to correspond to the protocol. Despite this, we have not classified this as a true positive defect, because it is unclear if the participant assumed that the ordering is implicit.

Table 5 shows the time-cost for extracting traces from the reachability graph and executing them against the protocol's Petri Net. The cost is low for all iterations, except for iteration 1 of 'Sale Transaction 2'. Although the time for extracting such a large number of traces was relatively low, a defect in that design caused an explosion in the number of traces, which consequently took over five and a half hours to execute on the Petri Net. This indicates a potential for explosion in execution time, and that further work is needed to mitigate this problem.

6. CONCLUSIONS

In this paper, we proposed an approach and tool support for finding defects in agent designs with respect to interaction protocol specifications. This approach involves generating the set of possible traces permitted by the agent detailed designs, and checking whether the sequences of messages in these traces are valid with respect to a given protocol specification. Although our tool supports only designs written using the Prometheus methodology, we believe the approach is general enough to work with other AOSE methodologies that follow the BDI model of agency.

We evaluated our approach on four designs developed by relatively experienced developers. The results showed that the proposed approach is able to detect defects in agent designs, with a low number of false positives and generally in a reasonable amount of time.

In future work, we will refine the approach to reduce false positives. One step would be to allow context conditions in BDI plans, which could then be used to filter out some traces that currently lead to false positives. Our evaluation indicated that some designs can result in an explosive number of traces, and therefore, long execution times. To mitigate this problem, we plan to *prioritise* traces, with the goal of finding and reporting defects early in the execution.

7. REFERENCES

[1] J. Botía, J. Gómez-Sanz, and J. Pavón. Intelligent data analysis for the verification of multi-agent systems interactions. *IDEAL*, pages 1207–1214, 2006.

[2] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent oriented software development methodology. *AAMAS*, 8(3):203–236, 2004.

[3] L. Cernuzzi and F. Zambonelli. GAIA4E: A tool supporting the design of MAS using Gaia. In *ICEIS*, pages 82–88, 2009.

[4] B. Cox, J. Tygar, and M. Sirbu. NetBill security and transaction protocol. In *First USENIX Workshop on e-Commerce*, pages 77–88, 1995.

[5] M. Dastani, J. Brandsema, A. Dubel, and J.-J. Meyer. Debugging bdi-based multi-agent programs. *Programming Multi-Agent Systems*, pages 151–169, 2010.

[6] S. A. DeLoach and J. C. Garcia-Ojeda. O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

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

[8] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *Proceedings of 5th IEEE International Symposium on Requirements Engineering*, pages 174–181. IEEE, 2001.

[9] P. Giorgini, J. Mylopoulos, and R. Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Eng. App. of AI*, 18(2):159–171, 2005.

[10] J. J. Gomez-Sanz, R. Fuentes, J. Pavón, and I. García-Magariño. INGENIAS development kit: a visual multi-agent system development environment. In *AAMAS*, pages 1675–1676. IFAAMAS, 2008.

[11] T. Miller, L. Padgham, and J. Thangarajah. Test coverage criteria for agent interaction testing. *AOSE XI*, pages 91–105, 2011.

[12] 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. *KER*, 21(4):345, 2006.

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

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

[15] J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *AOSE*, pages 201–218. Springer, 2001.

[16] 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.

[17] L. Padgham and M. Winikoff. *Developing intelligent agent systems: a practical guide*, volume 1. Wiley, 2004.

[18] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging mass using design artifacts: The case of interaction protocols. In *AAMAS*, pages 960–967. ACM, 2002.

[19] R. Pressman. *Software engineering: a practitioner's approach*, volume 7. McGraw-Kill New York, 2009.

[20] A. Rao and M. Georgeff. BDI agents: From theory to practice. In *AAMAS*, pages 312–319, 1995.

[21] L. Sterling and K. Taveter. *The Art of Agent-Oriented Modeling*. MIT Press, 2009.

[22] M. Winikoff. Towards making agent UML practical: A textual notation and a tool. In *Quality Software, 2005.(QSIC'5)*, pages 401–406. IEEE, 2005.

[23] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent oriented analysis and design. *AAMAS*, 3(3):285–312, 2000.