Tools for analyzing intelligent agent systems

Tibor Bosse\textsuperscript{a,\textdagger}, Dung N. Lam\textsuperscript{b} and K. Suzanne Barber\textsuperscript{b}

\textsuperscript{a} Vrije Universiteit Amsterdam, Department of Artificial Intelligence, De Boelelaan 1081a, 1081HV Amsterdam, The Netherlands

\textsuperscript{b} The University of Texas at Austin, Laboratory for Intelligent Processes and Systems, 1 University Station C5000, Austin, TX 78712-0240, USA

Abstract. When developing sophisticated multi-agent systems whose behaviors include collaboration, negotiation, and conflict resolution, analyzing and (empirically) verifying agent system behavior is a challenging task. To aid the developer in such tasks, this paper presents an approach that combines two software engineering tools – the Tracer Tool and the TTL Checker, which together record agent activities as execution traces and verify that the traces satisfy specified properties. The objective of the combined tool is to aid the user in redesigning, debugging, and maintaining the agent system. The Tracer Tool ensures that the user’s comprehension of the system behavior is accurate with respect to the execution traces and provides explanations of anomalous behavior, which can be detected as a failed behavioral property by the TTL Checker. The integrated approach has been applied to an agent-based system designed to coordinate unmanned aerial vehicles.

Keywords: Software comprehension, agent-oriented software engineering, reverse engineering, debugging, maintenance, tracer, formal analysis of agent systems

1. Introduction

Within the area of debugging and maintaining complex software systems, software comprehension (i.e., the process of gaining an understanding of how a software system functions) is crucial. Nevertheless, in the literature little attention is given to empirical analysis and verification of agent software systems. Most existing tools that are aimed at comprehending agent-based software can be classified either as reverse engineering or model checking tools. According to Chikofsky and Cross [7], reverse engineering is “the process of analyzing a subject system to create representations of the system at a higher level of abstraction”. A number of reverse engineering tools (e.g., Rational Rose [32]) have been developed to assist the designer in analyzing and perusing the source code of a software system (see Finnigan et al. [12] and Koskimies et al. [19]). In addition, various tools (e.g., Kronos [6], UPPAAL [22], SMV [24], and Spin [33]) have been developed to perform model checking (i.e., to prove that the formal specification of a system satisfies certain behavioral properties) (see Clarke et al. [8]). Despite such tools to aid the user in understanding software behaviors, software comprehension has still remained a time-consuming and mostly manual process. The obstacles faced by designers, developers, and end-users (who will be referred to simply as “users”) during software comprehension include (1) software complexity and sophistication, (2) the translation gap between low-level implementation structures and high-level design concepts, and (3) the large amount of time and effort demanded to build a comprehensive and accurate picture of the software and its behaviors.

This paper proposes an integrated approach to reduce the effort in comprehending and (empirically) verifying agent-based software systems, based on the collaboration between the Tracer Tool by [20] and the TTL Checker by [5]. To contend with the difficulties of software comprehension, the Tracer Tool records agent activities (including beliefs and intentions) as execution traces and displays a visualization of what the agents are doing in terms of high-level concepts that
are often used to describe agent behavior (e.g., beliefs, intentions, actions, and events [14,28,29]). In order to visualize the agent activities with respect to the user’s comprehension of the implemented agent system, the user’s comprehension (in terms of the relationship between these agent concepts) must first be explicitly modeled. Given this model and the execution traces, the Tracer Tool can also provide feedback on the accuracy of the user’s comprehension. The TTL Checker provides automated checking of agent system behavioral properties against the execution traces. Though similar to model-checking, this approach takes as input a set of observations from the system’s execution, rather than a formal model of the system. A user can discover behavioral anomalies from Tracer’s graphical interface or be alerted by the TTL Checker. To aid the user in tracking down a problem, the Tracer Tool can generate explanations that reveal, for example, why an agent performed a certain action.

In accordance with Edmonds’ emphasis on the need for empirical analysis of agent behavior, such as scenario-based analysis and field testing [11], the Tracer Tool and TTL Checker analyze the implemented system (using execution traces) rather than a model or design of the implemented system. The combination of tools presented in this paper is able to record agent behavior, verify properties of recorded agent behavior, detect anomalous behavior, and assist the user in diagnosing the cause of such behaviors. As will be discussed in detail later on, an important advantage of this approach is that the language used to express the properties under investigation can be more expressive. Similar research by Kalech and Kaminka [17] focuses on diagnosing inter-agent failures, where agents have models of other agents and uncertainties must be considered. Using a different approach, the tools in this paper take advantage of the fact that actual values can be recorded as true observations as the system is executing. In effect, the only uncertainties are those related to the user’s comprehension of the implemented system, which are addressed by the Tracer Tool’s ability to suggest updates to the user’s comprehension.

Sections 2 and 3 briefly describe the Tracer Tool and TTL Checker individually. Section 4 discusses the collaboration between these two tools, and Section 5 demonstrates the tools for software analysis in a specific case study, highlighting the ability to discover and analyze anomalous behavior in agent-based software. Section 6 shows the results of this case study. Finally, Section 7 reviews the contributions and related work.

2. Tracer Tool

Motivated by the difficulty of comprehending complex agent systems, the objective of the Tracing Framework [20] is to aid the user in understanding what is happening in the agent system and why such things are happening in terms of high-level agent concepts that are familiar to software designers, developers, and end-users. The Tracing Framework is established by the Tracer Tool, which provides automation to some of the tedious or arduous manual tasks involved for comprehension. The following subsections describe the major aspects of the Tracer Tool.

2.1. Logging execution traces

High-level agent concepts (e.g., beliefs, goals, intentions, action, events, and communication messages) are captured as run-time observations by logging agent activities. Each observation has attributes about time, data values (e.g., domain-specific data, and pre- and post-conditions), and stacktrace content in which the observation occurred. The Tracer Tool collects, sorts, and organizes the logged observations to be processed during or after run-time. By using a logging mechanism to record observations, the Tracer Tool can potentially be applied to any agent system implementation that can interface with Java’s logging API (directly or via a CORBA interface), regardless of agent or system architecture. The Tracer Tool uses attributes recorded for each observation to establish relationships between observations. The use of these attributes is demonstrated below in the heuristics when building the background knowledge. Note that the Tracer Tool is simply recording agent activity and does not perform any further reasoning on the beliefs, goals, or intentions of the agent, as the type of reasoning is specific to the agent system.

2.2. Building background knowledge

In order to help the user build and maintain an understanding of the system, the Tracer Tool requires an explicit model of what the user understands, called the background knowledge, which is represented as a relational graph where each node represents a high-level agent concept and an edge represents a relation (e.g., causal or temporal relation) between two agent concepts. The background knowledge can be seen as a causal graph of how the agents are expected to behave. Given an empty or incomplete background knowledge,
the Tracer Tool can offer suggestions about possible relations between agent concepts based on observations from the system’s execution, much like how humans impose causal connections between the action of flipping a switch and an observation of a light turning on. As the user reviews and accepts the suggested relations, the background knowledge is constructed from the actual implementation, which may be different from the original and possibly outdated design specification. In [20], Lam and Barber describe details about the background knowledge representation and experimental results on the effectiveness of the relation-suggesting algorithm. The results show that the 7 general heuristics used by the algorithm can suggest most of the relations needed in the background knowledge in the two case studies addressed in [20], and that some comprehension tasks can be automated to assist the user in building an understanding of the system.

The following heuristics are used to automatically suggest possible relations among observations. The objective of these heuristics is to connect observations together, creating a (causal/temporal) chain of observations. Some relations include relations between belief concepts to messages being sent and intentions being created, relations between intentions and resulting actions, relations between actions and subsequent events, and relations from events to agent beliefs.

1. If an observation is an intention, suggest relations from all previous beliefs (that have at least one common attribute with the intention) that occurred after the previous intention and do not already have an outgoing relation. If no belief is found, link to a previous intention with common attributes.

2. If an observation is a belief, suggest a relation from the latest message or event with common attributes that does not already have an outgoing relation.

3. If the observation is an event, suggest a relation from the latest action with common attributes that does not already have an outgoing relation.

4. If an observation is an action, suggest a relation from the latest action or intention with common attributes that occurred after the previous intention and does not already have an outgoing relation.

5. If an observation is a received message, suggest a relation from the latest message (that has common attributes) that was sent to the sender of the received message and does not already have an outgoing relation.

6. If an observation is a sent message, suggest a relation from the latest observation with common attributes and that does not already have an outgoing relation. When searching, skip all received messages whose sender is not the receiver of the sent message.

7. If an observation is an intention, suggest relations from all previous beliefs (regardless of the lack of common attributes) that occurred after the previous intention and do not already have an outgoing relation. If no belief is found, link to a previous intention or goal (regardless of the lack of common attributes).

Additional heuristics can be defined to identify more relations but perhaps at the expense of generality. The suggested relations are offered to the user to be added to the background knowledge. The user can modify the relations and add domain-specific ones. The resulting background knowledge is specific to the agent system being tested, but the aim of the heuristics is to be general enough to be applied to other agent systems. The background knowledge is used to interpret observations of agent activities as described in the next section.

2.3. Interpretation

From observations of agent behavior, actual agent behavior must be interpreted with respect to the user’s knowledge of expected behavior. The Tracer Tool can create an interpretation of the observations using the modeled background knowledge. Similar to the background knowledge, the interpretation is visualized as a relational graph, where nodes are observations and edges are (causal or temporal) relations among observations, as seen in Fig. 1. Graphical visualizations of what the agents are doing using familiar agent concepts offer a more palatable representation of agent behavior than lists of logged data that must be manually interpreted. The top of Fig. 1 shows observations as they are being logged by the agent system. An observation is characterized by the agent logging the observation, the type of observation, the subject or name for the observation, a timestamp, and other observation attributes. The graph in Fig. 1 shows the interpretation of the observations based on the relations defined in the background knowledge. In the graph, certain messages and beliefs (nodes annotated with “M” and “B”)
are used to create intentions (nodes labeled with “I”), which is the basis for actions (nodes labeled with “A”) being performed, which result in events (nodes labeled with “E”). If the events are sensed or inferred by the agent, then beliefs can be observed. In this particular interpretation, intentions are affected by previous intentions, which is shown by a directed edge between consecutive intention observations. The preference to show the edge is determined by the user, who specified the relation in the background knowledge. The background knowledge is the basis for interpretations and explanations. If there are problems with the background knowledge (which reflects the user’s comprehension), then they will be readily apparent in the visualization of the interpretation as unconnected nodes.

2.4. Explanation

Due to maintenance or redesign activities, the behavior of the implementation inevitably changes. The Tracer Tool provides suggestions to update the background knowledge, and the user must review and decide whether to accept modifications to the background knowledge. Hence, the user’s comprehension is kept updated with respect to the current implementation. To help the user decide whether a suggestion is valid or to further clarify causal factors of a particular observation, explanations can be generated by the Tracer Tool. An explanation is generated by following the graph of the interpretation backward starting from the observation in question. Abductive reasoning, which is typically used for explanation generation, is not needed. Since all observations are logged and an interpretation has been created using background knowledge, an explanation is simply a subgraph within the interpretation. The Tracer Tool presents the explanation in the form of a tree. For example, in Fig. 2, an explanation is given for action 531 (labeled as A:531), which is seen as a node in the interpretation (Fig. 1). The explanation tree shows that the flyToTarget action
Fig. 2. The Tracer Tool’s explanation of action A:531 ‘flyToTarget’.

(A:531) was caused by the removeCommitment intention (I:516), which was influenced by several beliefs (B:501, B:508, B:515) and a previous removeCommitment intention (I:479), all of which can be seen as nodes in the interpretation. The depth of the explanation continues until an observation without a cause is found, such as an exogenous environmental event or one of the initial observations.

2.5. Verification

The Tracer Tool provides a graphical interpretation of system behavior, so that patterns of agent behavior can be readily discovered by the user, a task at which humans are extremely good. Furthermore, anomalous behavior can also be detected as any observation that does not conform to some expected behavior pattern or constraint. Currently, anomaly detection is a manual task that involves discovering behavioral patterns in the interpretation, a task at which humans are extremely good. The Tracer Tool provides a capability to verify the expected behavior (represented as causal graphs) against the actual behavior of the implemented system. For more complex behavioral properties, the TTL Checker by Bosse et al. [5] provides automated verification of such properties against the observations collected by the Tracer Tool. By integrating the TTL Checker in the Tracing Framework, the user can specify properties to be automatically checked by the TTL Checker and can investigate any failed properties using the Tracer Tool’s explanation generator. Before presenting the case study and results of this collaboration, the next section provides an overview of the TTL Checker.

3. TTL Checker

The TTL Checker by Bosse et al. [5] is a tool that enables the automated verification of complex dynamic properties against execution traces. It is based on the predicate logical Temporal Trace Language (TTL), first introduced in Jonker and Treur [16]. First, in Section 3.1, the underlying modeling perspective is discussed. Next, in Section 3.2, the basic concepts of TTL are briefly introduced. Section 3.3 describes the operation of the TTL Checker.
3.1. Modeling perspective

Within the literature on analysis of properties (verification), generally two types of verification are distinguished, namely an empirical type of verification or validation (i.e., verifying whether a certain property holds for a (limited) set of traces), and an exhaustive type of verification (i.e., verifying whether a certain property holds for a model (thus for all possible traces) of a system). In the literature, much emphasis is put on the latter type of analysis, of which model checking is one of the most famous examples. This essentially comes down to the problem of justifying entailment relations between sets of properties defined at different aggregation levels of a system’s representation. In general, entailment relations can be established either by logical proof procedures or by checking properties of a higher aggregation level on the set of all theoretically possible traces generated by executing a system specification that consists of properties of a lower aggregation level. To make that feasible, expressivity of the language for these properties has to be sacrificed to a large extent (since for many relevant properties in the domain of agent-based systems, an exhaustive verification is simply undecidable). However, checking properties on a given set of traces (instead of all theoretically possible ones) is computationally much cheaper and more practical, and therefore the language for these properties can be more expressive (see [5] and [8] for an extensive discussion on this topic). TTL is an example of such an expressive language. For example, the possibility of explicit references to time points and time durations (see Section 3.2) enables modelling of the dynamics of continuous real-time phenomena, such as sensory and neural activity patterns in relation to mental properties (cf. [26]). This feature goes beyond the expressive power available in standard linear or branching time temporal (modal) logics such as LTL and CTL (e.g., [2,15]). Furthermore, the possibility to quantify over traces in TTL allows for specification of more complex adaptive behaviours, such as the property ‘exercise improves skill’. This is a relative property in the sense that it involves the comparison of two alternatives for the history. Such a property could be represented in TTL by translating it into a statement like: ‘For all agents a1, a2, for all traces γ1, γ2, and for all time points t, if at time point t agent a1 in trace γ1 has had more exercise than agent a2 in trace γ2, then a1 also has a higher skill than a2’. These kinds of relative properties can easily be expressed in TTL, whereas in standard forms of temporal logic different alternative histories cannot be compared.

3.2. TTL language

The TTL language is based on the assumption that dynamics can be described as evolution of states over time. The notion of state as used here is characterized on the basis of an ontology defining a set of physical and/or mental (state) properties that do or do not hold at a certain point in time. These properties are often called state properties to distinguish them from dynamic properties that relate different states over time. A specific state is characterized by dividing the set of state properties into those that hold, and those that do not hold in the state. Examples of state properties are ‘the agent is hungry’, ‘the agent has pain’, ‘the agent’s body temperature is 37.5° C’, ‘the environmental temperature is 7 °C’, ‘agent A believes that it is raining’, or ‘bot 15 communicates to bot 16 that it has serviced target 5’. Real value assignments to variables are also considered as possible state property descriptions.

To formalize state properties, ontologies are specified in a (many-sorted) first order logical format: an ontology is specified as a finite set of sorts (i.e., coherent groups of objects in the universe, cf. types in typeful programming), constants within these sorts, and relations and functions over these sorts (sometimes also called signatures). The examples mentioned above then can be formalized by nullary predicates (or proposition symbols) such as hungry, or pain, or by using n-ary predicates (with \( n \geq 1 \)) like has_temperature(body, 37.5°), has_temperature(environment, 7), belief(agentA, itsraining), or sending_message_to(bot15, bot16, belief(servicedTarget, (target, 5))). There is no specific restriction concerning the format of the predicates that can be chosen, as long as they are term structures, like in Prolog. In TTL, such predicates are called state ground atoms (or atomic state properties). For a given ontology Ont, the propositional language signature consisting of all ground atoms based on Ont is denoted by \( \text{APROP}(\text{Ont}) \). One step further, the state properties based on a certain ontology Ont are formalized by the propositions that can be made (using conjunction, negation, disjunction, and implication) from the ground atoms. Thus, an example of a formalized state property is hungry & pain. Moreover, a state \( S \) is an indication of which atomic state properties are true and which are false, i.e., a mapping \( S : \text{APROP}(\text{Ont}) \rightarrow \{\text{true, false}\} \). The set of all possible states for ontology Ont is denoted by \( \text{STATES}(\text{Ont}) \).
To describe dynamic properties of complex agent systems, explicit reference is made to *time* and to *traces*. A fixed time frame $T$ is assumed which is linearly ordered. Depending on the application, it may be dense (e.g., the real numbers), or discrete (e.g., the set of integers or natural numbers or a finite initial segment of the natural numbers). Dynamic properties can be formulated by stating a state at one point in time to a state at another point in time. A simple example is the following informally stated dynamic property for belief creation based on observation:

if the agent observes at $t_1$ that it is raining, then the agent will believe that it is raining.

A trace $\gamma$ over an ontology $\text{Ont}$ and time frame $T$ is a mapping $\gamma : T \rightarrow \text{STATES}(\text{Ont})$, i.e., a sequence of states $\gamma(t) \in \text{STATES}(\text{Ont})$. The temporal trace language TTL is built on atoms referring to, e.g., traces, time and state properties. For example, ‘in trace $\gamma$ at time $t$ property $p$ holds’ is formalized by $\text{state}(\gamma, t) |= p$. Likewise, ‘in trace $\gamma$ at time $t$ property $p$ does not hold’ is formalized by $\text{state}(\gamma, t) \not|= p$. Here $|=\!$ is a predicate symbol in the language, usually used in infix notation, which is comparable to the Holds-predicate in situation calculus. Thus, it denotes a satisfaction relation between a state and a state property. *Dynamic properties* are expressed by temporal statements built using the usual first-order logical connectives (such as $\neg$, $\land$, $\lor$, $\Rightarrow$) and quantification (the quantifiers $\forall$ and $\exists$; for example, over traces, time and state properties). For example, consider the following dynamic property:

in any trace $\gamma$, if at any point in time $t_1$ the agent $A$ observes that it is raining, then there exists a time point $t_2$ after $t_1$ such that at $t_2$ in the trace the agent $A$ believes that it is raining.

In formalized (TTL) form it looks as follows:

$$\forall t_1 \left[ \text{state}(\gamma, t_1) \models \text{observes}(A, \text{itsraining}) \Rightarrow \exists t_2 (t_2 \geq t_1 \land \text{state}(\gamma, t_2) \models \text{belief}(A, \text{itsraining}) \right]$$

In addition, language abstractions can be introduced by defining predicates for complex expressions. As TTL uses order-sorted predicate logic as a point of departure, it inherits the standard semantics of this variant of predicate logic. That is, the semantics of TTL is defined in a standard way, by interpretation of sorts, constants, functions and predicates, and a variable assignment. However, in addition the semantics involves some specialised aspects. As a number of standard sorts are present, the elements of these sorts are limited to instances of specified terms in these sorts, as is usual, for example, in logic programming semantics. For example, for the sort $\text{TIME}$ it is assumed that in its semantics its elements consist of the time points of the fixed time frame chosen. Moreover, for the sort $\text{TRACE}$, it is assumed that in its semantics its elements consist of a (limited) number of elements named by constants. Furthermore, for the sort $\text{STATPROP}$ for state properties it is assumed that in its semantics its elements consist of the set of terms denoting the propositions built in a chosen state language (this is called reification). A full description of the technical details of TTL's semantics is provided in [30].

### 3.3. Operation of the TTL Checker

To enable the automated verification of dynamic properties specified in TTL against formal traces, a dedicated tool has been developed. This tool (called the TTL Checker Tool) takes a dynamic property and one or more formal traces as input, and checks whether the dynamic property holds for the traces. Note that these checks can be performed irrespectively of who or what produced the formal traces: humans, simulators or an implemented (prototype) system. Thus, the TTL Checker Tool can be used to verify properties of empirical traces (produced by humans), simulated traces (produced by simulators) and execution traces (produced by implemented systems).

The Checker was implemented in Prolog (SWI-Prolog), and offers a user-friendly graphical editor to create and edit dynamic properties based on tree structures (by means of graphical manipulation and filling in slots, see Fig. 3) and a graphical user interface to visualize traces (using XPCE, see Fig. 4). A query to check some TTL formula against all loaded traces is compiled into a Prolog clause. Compilation is obtained by mapping conjunctions, disjunctions and negations of TTL formulae to their Prolog equivalents, and by transforming universal quantification into existential quantification. Thereafter, if this Prolog clause succeeds, the corresponding TTL formula holds with respect to all traces under consideration.

The complexity of the checking algorithm has an upper bound in the order of the product of the sizes of the ranges of all quantified variables. However, if a variable occurs in an atom of the form $\text{state}(\gamma, t) |= p$, the contribution of that variable is no longer its range size, but the number of times that the atom pattern...
occurs (with different instantiations) in trace(s) under consideration. The contribution of an isolated time variable is the number of time intervals into which the traces under consideration are divided. A number of specific optimisations make it possible to check realistic dynamic properties with reasonable performance. In practice, the duration of such checks usually varies from one second to a couple of minutes, depending on the complexity of the formula and the traces under consideration. With the increase of the number of traces, the checking time grows linearly. However, it is polynomial in the number of isolated time range variables occurring in the formula under analysis. For more (quantitative) details about the complexity of the checking algorithm, see [5].

The possibility to express dynamic properties in TTL and automatically check them against traces enables the user to formulate detailed requirements on the behavior of agent systems. See [4] for an earlier application in this area. That paper shows how a requirements analysis of an intelligent agent’s reasoning capability can be performed. The paper focuses in particular on a specific type of non-monotonic reasoning that involves dynamically introduced or retracted assumptions (as often used in diagnosis): reasoning by assumption. It is shown for this type of reasoning how relevant dynamic properties at different levels of aggregation can be identified as requirements that characterize the agent’s reasoning capability. Typical examples of such requirements are properties like reasoning termination (i.e., ‘eventually, the reasoning terminates’), prediction effectiveness (i.e., ‘for each assumption that is made, all relevant predictions are generated’), and correctness of rejection (i.e., ‘every assumption that is rejected does not hold in the world situation’). In [4], these kinds of requirements are formalized as dynamic properties in TTL. Moreover, it is shown how the TTL checker can be used to verify
these properties against different types of traces, and how this can be beneficial in the requirements analysis of reasoning capabilities of software agents. For example, checking properties against empirical traces may be useful in the context of requirements elicitation and validation in cooperation with stakeholders. Similarly, checking properties against simulated and prototype traces may be useful in the context of prototyping, by enabling the results of the requirements elicitation and specification phase to be formally analyzed and improved.

4. Integrated analysis approach

By combining the advantages of the Tracer Tool and TTL Checker, an integrated approach is presented for the analysis and comprehension of agent-based software. This section describes the constituent steps in this approach:

1. **Record execution traces by running the agent system through a set of testing scenarios.** This step is done using the Tracer Tool (in particular, using the logging mechanism described in Section 2.1). Normally, the logs are represented in XML format. To facilitate user understanding of the logs, a translation program has been written that translates the logs to traces in TTL format. As a result, the traces can be loaded into the TTL visualization tool. For an example screenshot, Fig. 4 shows time on the horizontal axis and state properties on the vertical axis. A mark on top of the line indicates that the state property is true during that time point or period. Note that a particular trace contains the information about only one agent. The information of other agents is stored in separate trace files.

2. **Build background knowledge of expected system behavior in terms of agent concepts such as beliefs, desires, and actions.** Based on the or-

Fig. 4. The TTL Checker’s visualization tool.
der and occurrence of the observations in the logs created in step 1, the Tracer Tool suggests relationships between the observations using heuristics (see Section 2.2). These suggestions help the user build up the background knowledge that is used to produce interpretations and explanations. The user can add relationships between certain observations that are specific to the agent system.

3. **Formalize relevant dynamic properties in TTL.** This step is done using the graphical editor for TTL introduced in Section 3.3. Two types of dynamic properties can be distinguished: generic dynamic properties and domain-specific dynamic properties. Generic dynamic properties are those that are independent of the domain. A simple example is ‘Communication Correctness’, i.e., “all messages sent from agent a1 to agent a2 are received by agent a2” (which is visualized in Fig. 3). The TTL checker contains a library of such generic dynamic properties, which can be re-used in any domain. Domain-specific dynamic properties have to be specified by hand for every new domain. In Section 5.3, some examples of both types of properties are given. Due to the expressiveness of the TTL language, very complex behavioral properties can be specified, e.g., relating events at many different time points and different traces to each other.

4. **Check the dynamic properties created in step 3 against the logs created in step 2.** This step is done using the TTL Checker (see Section 3.3 for details). As a result of this step, the user knows exactly which properties hold and which do not hold for the traces. Moreover, in case a property does not hold, the checker can provide detailed information about the position in the trace where the property fails. Usually this information has the form of a counter example. For example, if the aforementioned property ‘Communication Correctness’ fails, then the TTL Checker shows an example of a message that is sent from agent a1 to a2, but it is not received by a2.

5. **Explain failed properties.** This step is aided by the Tracer Tool (in particular, the explanation mechanism described in Section 2.4). Given a counter example from the TTL Checker due to a failed property, the user can examine the exact observations and its relation to other observations in the Tracer Tool’s visualization (see the interpretation graph and explanation tree in Figs 1 and 2). Observation details provide further evidence to track down the source of the problem, such as the type of communication message, who sent it, and why it was sent.

5. **Case study**

The integrated analysis approach mentioned above has been applied in a specific case study in the domain of UAV (Unmanned Aerial Vehicles). In Section 5.1, the UAV domain is introduced. In Section 5.2, the logs that were generated for this domain are described in detail. Section 5.3 provides some dynamic properties that are relevant for this domain. Section 6 summarizes results from this case study.

5.1. **Domain**

A multi-agent system has been implemented to simulate the target-monitoring behavior of Unmanned Aerial Vehicles (UAV) [21]. In this system, a two-dimensional area is simulated, in which a number of targets and UAVs are moving around. Figure 5 shows a visualization of 15 targets (labeled 0 through 14) and 3 UAVs (labeled 15, 16 and 17). Each UAV has an agent that determines the next mobile target the UAV should visit. From here on, ‘UAV’ and ‘agent’ is used synonymously. When a UAV visits a target, the target’s current location is broadcast to other UAVs. The overall objective is to minimize the time between visits for all targets. Each agent uses a Markov Decision Process that considers many factors, including beliefs about target locations, time of last visit, and preferences of other agents (see [21] for details). It would be a tedious task to manually check each observation to make sure certain behavioral properties hold – instead, the TTL Checker can automate this checking task.

5.2. ** Logs for the case study**

Using the mechanism described in Section 2.1, a number of logs of UAV simulations have been created. Since there is no completion state in the system, the logs can in theory be of infinite length. For the case study, it was decided to create logs of about 1250 time steps (which is sufficient for the current purpose, since each of these logs contains about 2000 observations on average). As mentioned earlier, the logs contain information about observations and at which time points they occur. Table 1 provides an overview of the agent concepts used in the UAV domain.
For empirical verification of agent systems, it is assumed that the user chooses scenarios that sufficiently cover the range of situations that the agent system will encounter. This implies that the user allows the system to run for an adequate amount of time so as to capture execution traces that encompass the scenarios. Thus, the completeness of analysis is reliant on the coverage of the scenarios tested, which is domain-dependent. Note that the objective of the combined tool is to aid in verifying agent behavior in a set of given execution scenarios. The presented tool cannot guarantee that agent behavior is correct for all possible scenarios unless they are included in the scenarios to be tested. This topic is further discussed in Section 7.
5.3. Dynamic properties for the case study

Based on the concepts in Table 1, several dynamic global properties (GP’s) have been specified for the UAV domain. Below, some of them are shown, both in an informal and in a formal (TTL) notation.

**GP1 “Target Diversity”**

“An agent does not visit the same target repeatedly without going to other targets”.

\( \forall \gamma \cdot \text{TRACES} \forall t_1, t_2 \cdot \text{TIME} \forall t_g \cdot \text{TARGET} \)

\[ \text{[ state}(\gamma, t_1) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land \text{state}(\gamma, t_2) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land t_1 < t_2 ] \]

⇒ \[ \text{[ } \exists t_3 \cdot \text{TIME} \exists t_{g1} \cdot \text{TARGET} \text{ state}(\gamma, t_3) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_{g1}) \land t_1 < t_3 < t_2 \land t_g \neq t_{g1} ] \]

Since the goal of the agents in the UAV system is to maintain updated beliefs about the positions of the moving targets, it is irrational for an agent to visit the same target twice in a row. Therefore, one would expect the above property GP1 to hold. Note that this property assumes a discrete time frame (i.e., each event has the duration of one time unit).

**GP2 “Belief Correctness”**

“All events of an agent visiting a target result in the belief of the agent that it serviced the target”.

\( \forall \gamma \cdot \text{TRACES} \forall t \cdot \text{TIME} \forall t_g \cdot \text{TARGET} \)

\[ \text{[ state}(\gamma, t) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land \text{state}(\gamma, t) \Rightarrow \text{belief}(\text{servicedTarget}, \text{target}, t_g) ] \]

Dynamic property GP2 can be used to check whether the agents’ belief generation mechanisms function adequately. Note that in the UAV system there is no delay between the moment that an agent visits a target and the moment that it believes that it has serviced the target.

**GP3 “Servicing Completeness”**

“All targets are eventually serviced by at least one agent”.

\( \forall t_g \cdot \text{TARGET} \)

\[ \exists t_1, t_2 \cdot \text{TIME} \text{ state}(\gamma, t_1) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land \text{state}(\gamma, t_2) \Rightarrow \text{belief}(\text{servicedTarget}, \text{target}, t_g) \land t_1 < t_2 \]

Property GP3 can be interpreted as a minimal requirement for the behavior of the system: if the agents behave correctly, eventually every target should at least be serviced once. Recall that the information about different agents is stored in different traces. Therefore, it is sufficient to check for all targets that there is at least one trace in which it is serviced.

**GP4 “Message Completeness”**

“Whenever an agent believes something, it sends a message about this to all other agents”.

\( \forall \gamma \cdot \text{TRACES} \forall t \cdot \text{TIME} \forall b \cdot \text{BELIEF} \)

\[ \text{[ state}(\gamma, t) \Rightarrow \text{belief}(b) \land \exists \text{self} \cdot \text{AGENT} \exists a \cdot \text{AGENT} \]

\[ \text{[ state}(\gamma, t) \Rightarrow \text{sending_message_to}(\text{self}, a, b) ] \]

In contrast to most other properties presented in this section (except GP5), GP4 is a generic dynamic property. However, it can be instantiated for the UAV domain by replacing the belief b by a specific belief for that domain (for example, a belief of a UAV that it has visited a certain target, or that it has made the commitment to visit a certain target).

**GP5 “Communication Correctness”**

“All messages sent from agent a1 to agent a2 are received by agent a2”.

\( \forall \gamma \cdot \text{TRACES} \forall t \cdot \text{TIME} \forall a_1, a_2 \cdot \text{AGENT} \forall m \cdot \text{MESSAGE} \)

\[ \text{[ state}(\gamma, t) \Rightarrow \text{sending_message_to}(a_1, a_2, m) \land a_1 \neq a_2 ] \]

⇒ \[ \exists \gamma \cdot \text{TRACES} \text{ state}(\gamma, t) \Rightarrow \text{receiving_message_from}(a_2, a_1, m) ] \]

GP5 is another generic property. It can be used to guarantee the delivery of communication messages. Again, it can also be instantiated for the UAV domain by replacing the belief m by a specific message for that domain.

**GP6 “Interval with two visits”**

“There is an interval with length i in which target tg is visited at least twice (by two different agents)”.

\( \exists \gamma, t_1, t_2 \cdot \text{TRACES} \exists t_3 \cdot \text{INTEGER} \exists t_{g1}, t_{g2} \cdot \text{TIME} \exists t_g \cdot \text{TARGET} \)

\[ \text{[ state}(\gamma, t_1) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land \text{state}(\gamma, t_2) \Rightarrow \text{event}(\text{uavScan}, \text{target}, t_g) \land t_{g1} \neq t_{g2} \land |t_{g1} - t_{g2}| = i ] \]

Note that, while checking property GP6, the TTL checker automatically returns the smallest i (if any) for which the property holds. Thus, it allows the user to find the shortest interval in which a certain target is visited twice by different agents. Finding such intervals could help the user in locating suboptimal behavior, because when an agent just visited a certain target, it is often not necessary that another agent visit the target again within a small time period. Obviously, similar properties can be created that state, for example, that a certain target is visited exactly twice in a certain period, or that a target is visited three times.
GP7 “Amount of Visits”
“For all targets there exists an amount of visits i”.
\[
\forall \gamma \in \text{TRACES} \exists t : \text{TIME} \quad \text{case}(\text{state}(\gamma, t) \models \text{event}(\text{uavScan}, (\text{target}, \text{tg})), 1, 0)
\]

In this property (and similar ones), the expression \(\text{case}(f, v_1, v_2)\) indicates the value \(v_1\) if \(f\) is true, and \(v_2\) otherwise. Note that the TTL Checker allows one to find the number \(i\) for which property GP7 holds for each target separately. Thus, it allows the user to check how often each target is visited. When there are big differences between the amounts of visits to different targets, something may be wrong since the agents in the system were designed to make sure that all targets are visited almost equally so as to minimize the time between visits of all targets.

GP8 “Belief Completeness”
“Eventually all agents believe for all targets that either they serviced it or another agent serviced it”.
\[
\forall \gamma \in \text{TRACES} \forall \text{tg} \in \text{TARGET} \quad \exists t : \text{TIME} \quad \text{state}(\gamma, t) \models \text{belief}((\text{servicedTarget}, (\text{target}, \text{tg}))) \lor \text{state}(\gamma, t) \models \text{belief}((\text{other_servicedTarget}, (\text{target}, \text{tg})))
\]

GP8 is another requirement for the behavior of the system: if the agents behave correctly, then eventually all agents should be informed that all targets have been serviced. It is important to realize that the selection of properties presented in this section is by no means complete. Obviously, depending on the user’s specific interests, the user is free to develop any desired property in TTL, using the graphical editor introduced in Section 3.3. To mention some interesting future directions, in addition to properties such as Belief Correctness (GP2) and Belief Completeness (GP8), a variety of properties can be formulated to check whether the traces under analysis satisfy some general properties of belief calculus. For example, it is not difficult to express the property “If an agent believes both ‘A or B’ and ‘not A’, then later it also believes ‘B’” in TTL. Similarly, TTL formulae may be designed for standard properties required in parallel and distributed systems, like fairness or freedom from starvation. However, in the current paper we restricted ourselves to a number of properties that are relevant for the UAV case study.

6. Results

This section discusses the results of the case study. Section 6.1 focuses on the results of the verification of properties against execution logs (step 4 of the method) and Section 6.2 focuses on explanation of unexpected results (step 5).

6.1. Verification

Using the TTL Checker, the properties shown in Section 5.3 have been checked against the execution logs mentioned in Section 5.2. Although most of the properties were satisfied, as expected, some of them failed. For example, property GP1 failed in a number of cases. First, because one of the agents (Agent 15) visited Target 4 twice in a row (at time points 661 and 662). And second, because another agent (Agent 17) visited Target 9 twice in a row (at time points 200 and 201). In these cases, the agents decided to visit the target again, because the costs of doing this were minimal. This is further explained in Section 6.2.

Moreover, note that property GP7 was successfully checked. As mentioned earlier, this is not very surprising, but it is more interesting to find the exact number \(i\) for which this property succeeds for different targets. These numbers are provided in Table 2. As indicated in Table 2, the number of visits is unevenly distributed over the targets. In particular, Target 11 is an exception, since it is visited only 5 times during the whole execution of the system, whereas the other targets are

<table>
<thead>
<tr>
<th>Target</th>
<th># Visited by Ag. 15</th>
<th># Visited by Ag. 16</th>
<th># Visited by Ag. 17</th>
<th># Visited in Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>29</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>0</td>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>5</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>18</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>1</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1</td>
<td>18</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
visited at least 12 times. The next subsection investigates this issue further.

6.2. Comprehension

This subsection describes how the Tracer Tool can help the user comprehend the failure of GP7. The developer of the UAV agents had not realized that Target 11 was being visited so few times relative to other targets. Since a performance metric of the system is the average time between visits for each target, visiting Target 11 more often would improve the overall system performance. To make such an improvement, the developer needs to understand why Target 11 is not being visited as frequently as other targets.

In determining which targets to service next, the agent controlling each UAV evaluates a reward function for each target, considering the target distance, the last time of visit, and other agents’ preferences and intentions. Using background knowledge for the UAV agents and the execution traces, the Tracer Tool creates a visual interpretation that can be readily analyzed. Investigating the initial observations in the interpretation, all agents have the initial belief that Target 11 is located at the side edge of the map; there is no other target that is further out on the map. Verifying this with the UAV Simulation, Target 11 and 15 are the farthest targets from the initial locations of the 3 UAVs. Since the UAVs commit to visit 3 targets at a time as shown by the intention nodes in the interpretation graphic, and there are 15 targets and 3 UAVs, only 9 targets are selected and partitioned among the UAVs. Thus, there are 6 targets (including Target 11) that have yet to be assigned to a UAV. The agent developer states that each UAV adds an extra bonus to the reward value for visiting a certain target so that the UAV will tend to revisit the same targets, thus creating a stable partitioning of targets to UAVs. Since Target 11 was not initially visited, it does not benefit from the bonus. The visual interpretation greatly aids in understanding each agent’s current and past internal beliefs and intentions and how they affect the agents’ actions.

Over the course of the simulation, targets move randomly. Target 11 happens to move towards the edge of the map away from the other UAVs, and thus decreasing the reward value of visiting Target 11. According to the reward function, the UAVs tend to visit clusters of targets because the overall cost of their intention commitments (which constitute the next 3 targets to visit) is minimized if the targets are closer together. Since Target 11 is further away from the UAVs and from other targets, the UAVs wait until it is more urgent to visit Target 11, i.e., when the reward value is higher than for visiting the closer targets. However, since the UAVs commit 3 targets into the future, they are also committed to moving back into the cluster of targets and thus away from Target 11.

In summary, it is a combination of the initial target assignment and the random movement of a target to isolated positions that caused this potentially adverse situation, where a target is not visited for long durations. Coefficients in the reward function can be adjusted to prevent such situations. Another solution is to commit to fewer targets at a time, thus allowing Target 11 to be visited when it is close enough to the last intended target.

7. Discussion

Software comprehension is an essential issue during the development, maintenance, and debugging of complex agent systems. Nevertheless, existing techniques (such as reverse engineering [7] and model-checking [8]) face a number of difficulties, such as the high complexity and sophistication of the software, the translation gap between low-level implementation structures and high-level design concepts, and the amount of effort demanded to gather a comprehensive picture of the software. To deal with these problems, this paper presents a novel approach to analyze the behavior of agent systems. The approach is a result of the collaboration between the Tracer Tool for comprehension of agent software [20] and the TTL Checker for automated checking of dynamic properties of agent systems [5]. The integrated approach comprises four important elements, i.e. (1) logging of agent system executions in terms of high-level design concepts, (2) formalization of desired dynamic properties, (3) automated verification of properties against logs, and (4) computer-aided explanation of unexpected properties.

To test the proposed approach, it was applied in an Unmanned Aerial Vehicles case study. Using the approach, the software developers succeeded in locating some undesired behavior, which subsequently allowed them to investigate the causes for such behavior. Unlike [27] who focus only on the verification of agent interaction protocols, the current approach can verify any aspect of the agent system, including interaction protocols.

As mentioned in Section 3.1, the perspective taken in this paper is to verify properties against finite sets
of traces. Although, as discussed, this has the advantage that the complexity of the checking process is relatively low, an inevitable drawback is the fact that there is no guarantee that the selected traces cover all possible scenarios. Nevertheless, various strategies may be helpful to increase the percentage of the scenarios that is covered. First of all, it helps to create a large number and variety of traces to increase the probability that unexpected events occur and to increase the coverage of possible case scenarios. This generation of traces may be done randomly, but another, more methodical approach is to do this partially controlled. In such an approach, some of the parameter values of the scenario are set manually, in order to increase the probability that interesting “borderline cases” occur. Selecting such appropriate parameter settings is a domain-dependent process. For the UAV domain, examples of such borderline cases would be situations in which one of the targets is deliberately placed far away from the other targets, or in which several clusters of targets are created. If the number of traces that are generated in this way becomes large, it does not always make much sense to check properties against sets of traces in an all-or-none manner. In many cases, the user is not so much interested in whether a specific property occurs at all, but rather in how often the property occurs within the traces. To perform such checks, TTL offers the capability (via the case construct introduced in property GP7) to generate properties with a probabilistic character. For an earlier application of such probabilistic TTL properties, see [3]. However, even these approaches do not guarantee that all unexpected situations are covered, particularly when the user does not have much insight in the range of possible behaviors. Therefore, another potential solution is to use testing techniques from traditional software engineering. An interesting approach in this direction is put forward by [10] and [18], who propose to use evolutionary learning methods to search for unwanted emergent behaviors in multi-agent systems.

Concerning other related work, existing research addresses the different elements of the presented approach separately. For example, in [13] several tools are suggested for the collection of trace files (consisting of important events and corresponding time parameters) in a distributed system. To examine the collected traces, techniques for querying distributed databases are proposed. This approach is similar to our approach in the sense that both make use of logging and analyzing real executions of an implemented system. A difference is that our approach focuses particularly on multi-agent systems, whereas [13] address distributed systems in general. Furthermore, various papers focus on tools for the visualization of multi-agent system behavior. Examples are [23] and [25], both claiming that visualization is essential to facilitate efficient debugging of multi-agent systems. In the current paper, this approach is adopted to aid rapid comprehension of agent concepts and their relationships in the implemented system.

Related work by Hamdan and Denzinger [9] enables agents to model and predict the actions of other agents based on previously observed situation-action pairs. Their approach is suited to reactive agents that do not change their behavior over time, in applications where the number of possible situations is finite (since all observed situations will be modeled), and in situations where features are completely observed. Unlike their approach, the Tracer tool is targeted for software verification and debugging from the developer’s perspective, rather than predicting agent behavior from an agent’s perspective. The logging approach used by the Tracer tool enables all relevant internal states (i.e., agent concepts) of an agent to be observed, not only externally observable features or actions. The environmental situation is irrelevant to the Tracer tool because all situation features that are used by an agent must be first modeled as a belief. Hence, the Tracer tool allows the case where an agent does not observe certain situation features, which is not considered in Hamdan and Denzinger’s work.

Despite our choice to focus on empirical verification, in an ideal situation the verification of dynamic properties is done exhaustively (i.e., against all theoretically possible traces instead of a practically given set of traces). However, due to the expressiveness of the TTL language, exhaustive search is not practical. One problem is that the use of real-valued time parameters in principle causes an explosion of the amount of states to be considered. This problem can be dealt with by assuming finite variability of state functions (i.e., partitioning time into a finite number of intervals with real-valued durations in which no change occurs). Work in this direction has been explored by, e.g., [1,8], and more recently [31]. Based on this idea, some initial model-checking tools for real-time systems have recently been developed (see Bozga et al. [6] and Larsen et al. [22]). Currently, the possibilities are investigated to combine such techniques with the presented approach.
Acknowledgements

The authors are grateful to Jan Treur for his constructive comments, and to David Han for the development and analysis of the agents in the UAV system. This research was funded in part by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0588. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References


