ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-243

Checking the Conformance of Requirements in Agent Designs Using ATL

Nitin Yadav and **John Thangarajah** [1]

Abstract. Intelligent agent systems built using the BDI model of agency have grown in popularity for implementing complex systems such as UAVs, military simulations, trading agents and intelligent games. The robust and flexible behaviours that these systems afford also makes testing the 'correctness' of these systems a non-trivial task. Whilst the main focus on existing work has been on checking the correctness of agent-programs, in this work we present an approach to formally verify agent-based designs for a particular BDI agent design methodology. The focus is on verifying whether the detailed design of the agents conform to the requirements specification. We present a sound and complete approach, formally verifiable properties, and an evaluation with respect to time and effectiveness.

1 Introduction

Intelligent agent systems have been used to develop software systems in a variety of application areas [19]. Agent systems of this kind are often designed and implemented in terms of software structures that are based on metaphors of humans and human societies; for example, events, beliefs, goals, plans and intentions. While there are many agent development paradigms, the *Belief-Desire-Intention* (BDI) model [24] is a mature paradigm that has been adopted by several agent development platforms such as JACK [28], JASON [8] and JadeX [7]. As with any software development, testing the 'correctness' of these BDI-based agent systems is an important task. See [20] for a recent survey of the state of the art in testing agent systems.

Most existing work on verifying BDI agent systems has focused on formal verification (e.g. [10]), particularly using model checking techniques (e.g. [13]) and theorem proving (e.g. [25]), or on runtime testing (e.g. [30, 31]) of *agent programs*. That is, testing the correctness after the system (or part of it) has already been implemented. However, the notion that identifying and correcting errors early in the software development cycle is well accepted in software engineering [6, Page 1466]. In this work we present a formal model-checking based approach for verifying the correctness of the agent design models at the *design stage* prior to implementation.

There has not been much work on testing the correctness of detailed agent designs with the exception of the recent work by Abushark et al.[1, 2]. They provide an approach for checking the correctness of interaction protocols [1] and requirement models [2]. Their approach in [1] is to extract all possible behaviour traces of the detailed agent design (comprising goals, plans and message exchanges) related to a particular protocol and report the ones that do not conform by checking against an execution structure of the protocol. They adapt a similar approach for checking requirements in [2].

[1] RMIT University, Australia, email: firstname.lastname@rmit.edu.au

Although their approach is able to identify traces that do not conform to the interaction or requirement specifications, there are some significant limitations. The first is that the approach has no formal semantics and as mentioned by the authors in [2] the approach is neither sound nor complete. Secondly, in the design, plans that post two or more sub-goals where the execution can be interleaved can create a large number of traces caused by the interleaving of all the steps of those sub-goals. As the number of parallel steps increases the possible behaviour traces grows at least exponentially and hence the time and space to extract them. This is particularly problematic for checking the requirements using scenarios, where a scenario specifies a particular sequence of steps. If the sequence specified is one of the possible parallel interleaving, all possible traces must be extracted to find the one that matches. Finally, the trace extraction is carried out independent of the requirements specification. Hence, no matter how long or short the scenario is, the number and size of traces is only dependent on the detailed design, which can be inefficient.

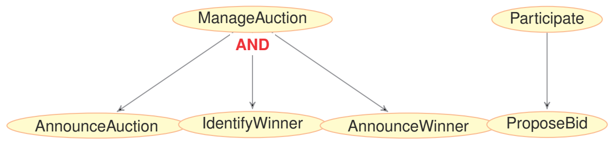
In this work, we present a formal approach to verifying the correctness of the detailed agent designs with respect to the requirement specifications via a model checking approach. Similar to the work of Abushark et al., we use Prometheus agent design models [22] as the basis of our work. The proposed approach however, is formal, sound and complete, uses model checking rather than trace extraction, and presents the designer with a model as well as traces. In addition, the approach is general and can be adapted to other agent design methodologies that follow the BDI model of agency [12] as they all share a set of common design concepts.

The key contributions of this paper are: (i) we formalise, rather informal and abstract, agent-oriented software design concepts; (ii) we provide precise semantics for the verification problem that we (and Abushark et.al [1]) are trying to address; (iii) we provide a formal design framework that is amenable to automatic testing and verification; (iv) we demonstrate how verification tools developed within the agent community can be used for checking conformance within agent designs; and (v) we provide an initial evaluation on the scalability of the proposed approach.

2 Background and related work

2.1 Prometheus- BDI agent design paradigm

The Prometheus methodology [22], together with the design tool (PDT) [23], supports the complete development of agent systems from specification and design through to implementation. The design methodology presents well-defined notation and processes for developing three key phases. The *System specification* where the interface of the system is specified in terms of inputs (percepts), outputs (actions), the external entities that interact with the system, and



(a) Goal overview for the auction example.

No	Type	Name	Role
1	Percept	StartAuction	Auctioneer
2	Goal	AnnounceAuction	Auctioneer
3	Goal	Participate	Bidder
4	Goal	IdentifyWinner	Auctioneer
5	Goal	AnnounceWinner	Auctioneer

(b) A scenario for the auction example.

Figure 1: Requirements spec. for the auction example.

the requirements of the system specified via scenarios and goal diagrams. Scenarios specify a particular run of the system akin to use cases in traditional Object-Oriented design. Table 1b, illustrates an example scenario related to an auction system. The Goal diagrams specify the functionality of the system and how they may be decomposed into smaller sub-goals. Figure 1a, illustrates a goal diagram for an auction system.

The *Architectural design* specifies the internals of the system in terms of agents and any communication between them. The *detailed design* details the internals of each agent in terms of plans, messages, and goals that they handle and produce amongst other things. Figure 2, illustrates the detailed design of an Auctioneer agent, in a simple auction system. The plan New_Auction is triggered by the percept StartAuction, produces the (sub)goal AnnounceAuction, this goal then triggers the execution of the plan BroadcastAnnouncement.

2.2 Alternating-time temporal logic

Alternating-time Temporal Logic (ATL) [4] is a logic for reasoning about the ability of agent coalitions in *multi-agent game structures*. ATL formulae are obtained by combining propositional formulas, the usual temporal operators—namely, $\bigcirc$ ("in the next state"), $\Box$ ("always"), $\Diamond$ ("eventually"), and $\mathcal{U}$ ("strict until")—and a *coalition path quantifier* $\langle\langle A \rangle\rangle$ taking a set of agents A as parameter. Conceptually, an ATL formula $\langle\langle A \rangle\rangle \phi$, where A is a set of agents, holds in an ATL model if the agents in A can *force* ϕ true, by choosing their actions, no matter how the agents *not* in A happen to move. The semantics of ATL is defined in concurrent game structures where, at each state, all agents simultaneously choose their actions from a finite set, and the next state deterministically depends on such choices. More concretely, a concurrent game structure is a tuple $\mathcal{M} = \langle \mathcal{A}, Q, \mathcal{P}, Act, d, \mathcal{V}, \sigma \rangle$, where $\mathcal{A} = \{1, \ldots, k\}$ is a finite set of agents, Q is the set of states, $\mathcal{P}$ is the set of propositions, Act is the set of all domain actions, $d : \mathcal{A} \times Q \mapsto 2^{Act}$ indicates all available actions for an agent in a state, $\mathcal{V} : Q \mapsto 2^{\mathcal{P}}$ is the valuation function stating what is true in each state, and lastly $\sigma : Q \times Act^{|\mathcal{A}|} \mapsto Q$ is the transition function mapping a state q and a joint-move $\vec{a} \in \mathcal{D}(q)$, where $\mathcal{D}(q) = \times_{i=1}^{|\mathcal{A}|} d(i, q)$ is the set of legal joint-moves in q, to the resulting next state q'.

A *path* $\lambda = q_0 q_1 \cdots$ in a structure $\mathcal{M}$ is a, possibly infinite, sequence of states such that for each $i \geq 0$, there exists a joint-move $\vec{a_i} \in \mathcal{D}(q_i)$ for which $\sigma(q_i, \vec{a_i}) = q_{i+1}$. To provide semantics to formulas $\langle\langle \cdot \rangle\rangle \varphi$, ATL relies on the notion of agent strategies. Technically, an ATL *strategy* for an agent *agt* is a function $f_{agt} : Q^+ \mapsto Act$, where $f_{agt}(\lambda q) \in d(agt, q)$ for all $\lambda q \in Q^+$, stating a particular action choice of agent *agt* at path λq. A *collective strategy* for group of agents $A \subseteq \mathcal{A}$ is a set of strategies $F_A = \{f_{agt} \mid agt \in \mathcal{A}\}$ providing one specific strategy for each agent $agt \in A$. For a collective strategy F_A and an initial state q, the set of all *possible outcomes* of F_A starting at state q, denoted $\mathit{out}(q, F_A)$, are the set of all computation paths that may ensue when the agents in A behave as prescribed by F_A, and the remaining agents follow any arbitrary strategy (see [4]). The semantics for the coalition modality is then defined as follows (here ϕ is a *path formula*; and $\mathcal{M}, \lambda \models \phi$ is defined in the usual way [4]):

$\mathcal{M}, q \models \langle\langle A \rangle\rangle \phi$ *iff* there is a collective strategy F_A such that for all computations $\lambda \in \mathit{out}(q, F_A)$, we have $\mathcal{M}, \lambda \models \phi$.

Given a concurrent game structure $\mathcal{M}$ and an ATL formula ϕ, the *model checking problem* of ATL asks for the set of states in $\mathcal{M}$ that satisfy formula ϕ. Let $[\phi]_{\mathcal{M}}$ denote the *maximal* set of states of $\mathcal{M}$ that satisfy ϕ. A state q in $\mathcal{M}$ is said to be *winning* for ϕ if $q \in [\phi]_{\mathcal{M}}$.

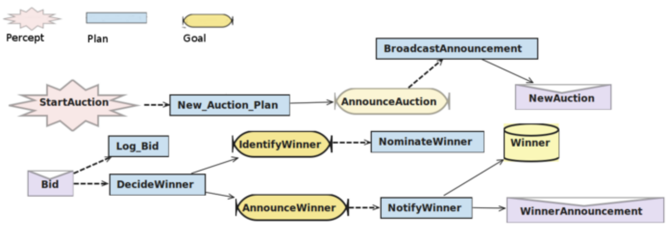


Figure 2: Auctioneer agent - detailed design.

2.3 Related work

The BDI agent-oriented paradigm is a popular and successful approach for building agent systems. We have a long history (20+ years) of collaboration with multiple industry partners that use BDI agents and the Prometheus methodology (or its variants) to design the agents. The overarching goal in this paper is to provide our user community with tools that would enable them to develop more reliable systems which is a need that has emerged from them.

As is with any software development approach, a complex system is generally conceptualized first using software design methodologies (e.g., Prometheus in our case), and then this design is implemented using a programming language (e.g., JACK, JASON, JADEX, etc). Although there is a large body of work on verifying BDI agent systems using formal verification techniques [17, 13, 26, 3, 11], these approaches either assume the presence of a formal BDI system or testing is done on an agent system that is already implemented. Though recent work [1, 2] has tried to address verification at the level of agent designs, that is before a system is even programmed, the use of formal verification techniques for this purpose is not widespread.

With respect to agent designs, the Tropos methodology supports validating requirements using T-Tool [14] and reasoning about agent goals using the GR-Tool [15]. However, Tropos does not provide support for verification of requirements against agent designs. The approach that comes closest to ours is that of Abushark et.al. [2]. In their work the authors verify requirements against detailed designs via an algorithmic approach. The key idea there is to construct a Petri net from the given requirements and verify the detailed designs by extracting the behavior traces and executing these traces against the constructed Petri net. The authors in [2] address the issue of requirements verification in an ad-hoc manner without formalising the problem, and leave the soundness and completeness of their approach as future work.

In [27], scenarios in Prometheus are extended such that they may be propagated to the detailed design, thus reducing the chance of error, and use them in generating scenario-based run-time test cases. Their approach complements the formal verification framework presented in this paper.

In this work we are interested in formally verifying requirements against agent details and on the way of achieving this provide a way to formalise agent designs based on Prometheus methodology. Further, as shown in [9], BDI design methodologies share similar structures, and hence we believe that the key formal notions developed here can be adapted to other BDI design methodologies.

3 Framework

We begin with formally defining the core elements of an agent design. Goals are usually captured by defining *goal trees*. A goal tree is a tree (in its usual sense) whose nodes are agent goals and branches are labelled either an AND (all sub-goals required to achieve the goal) or an OR (only one sub-goal required to achieve the goal). Formally, a *goal tree* is a tuple $T = \langle G, g_0, \mathcal{R}, \mu\rangle$ where G is the set of goals, $g_0 \in G$ is the top level goal, relation $\mathcal{R}$ defines the parent-child relationship between goals where $\mathcal{R}(g_1, g_2)$ implies that g_1 is the parent goal of g_2, and $\mu : G \to \{\mathsf{AND}, \mathsf{OR}\}$ provides AND-OR mapping for sub-goals. Given a goal g and its sub-goals $g_1, \ldots, g_n$, let $\mathsf{sg}(g) = \{g_1, \ldots, g_n\}$. That is, the function sg returns the set of all sub-goals of g.

A scenario consists of a sequence of steps that need to be achieved for it to be completed. Each step is either a *percept*, a *goal* or an *action* and the entity responsible for it is defined by an agent *role*. Formally, each step in a scenario is a pair (o, r) where o is a *step type* and r is an agent role. A scenario is then an ordered tuple $S = \langle (o_1, r_1), \ldots, (o_n, r_n)\rangle$ where $n \geq 1$. Given a scenario S consisting of n steps we denote the size of S by $|S|$ (i.e., $|S| = n$), its i^{th} step by $S[i]$, and the step type and role of the i^{th} step by $S[i].\mathsf{type}$ and $S[i].\mathsf{role}$, respectively. A *requirements specification* simply consists of a scenario S and a set of k goal trees $T_i = \langle G_i, g_{i0}, \mathcal{R}_i, \mu_i\rangle$, where $1 \leq i \leq k$.

A detailed design on the other hand consists of entities used in scenarios (i.e., goals, percepts, and actions) and additionally messages and plans. A *message* is of the form $\mathsf{from} \to \mathsf{to} : \mathsf{msg}$ where from and to are agents and msg is the name of the message. A plan in an agent design is defined in terms of its trigger and outputs such as messages, actions, and sub-goals. Details of a plan body are not defined at the design level (it is an implementation level detail). A *plan* is a tuple $p = \langle \mathsf{name}, \mathsf{trigger}, O\rangle$ where name is a unique identifier for p, $\mathsf{trigger}$ is its trigger, and O is the set of its outputs. For technical convenience, we shall refer to components of plan p as $p.\mathsf{name}$, $p.\mathsf{trigger}$, and $p.O$. In the rest of the paper, we refer to the set of all goals by Goals. Similarly, $\mathsf{Percepts}$, $\mathsf{Actions}$, $\mathsf{Messages}$, and Plans.

In order to track each possible plan instance we need to track *how* it was triggered. We assign each plan instance an id to track a sequence of plan activations that preceded it. Formally, the set of ids for plan $p = \langle \mathsf{name}, \mathsf{trigger}, O\rangle$ is defined inductively by $\Delta(p) = \{\mathsf{id} \cdot \mathsf{name} \mid \exists p'(\mathsf{trigger} \in p'.O), \mathsf{id} \in \Delta(p')\}$ where for the base case we have that $\Delta(p) = \{\mathsf{name} \mid \mathsf{trigger} \in \mathsf{Percepts}\}$. Generally, an id for a plan instance p will start with a plan name that handles a percept, followed by a sequence of plan names, subsequently ending with p's name, such that the trigger for a plan was in the output of plan preceding it in the id. Given an $\mathsf{id} = \mathsf{id}' \cdot \mathsf{name}$ for a plan $p = \langle \mathsf{name}, \mathsf{trigger}, O\rangle$ let $\mathsf{history}(\mathsf{id}) = \mathsf{id}'$ and let $\mathsf{active}(\mathsf{id}) = p$.

An agent for design purposes is a collection of plans along with its roles. A *design agent* is a tuple $\mathsf{Ag} = \langle n, R, Pl\rangle$ where n is agent's name, R is a set of agent's roles, and Pl is agent's plan library. A *detailed design* consists of a set of design agents. Formally, a detailed design D is a set $\{\mathsf{Ag}_1, \ldots, \mathsf{Ag}_n\}$ where Ag_i are design agents for $1 \leq i \leq n$. Observe that goals, actions, messages are closely linked to plans via its trigger and outputs and can be directly inferred from the agents plan library.

3.1 Conformance of requirements and designs

The problem we are interested in is to *automatically check if a detailed design D conforms to a requirements specification R.* Observe that though all scenario entities (i.e., step types) are also available in the detailed design there may not be an exact one to one mapping between them (else the verification task will be simple). For example, a scenario may specify a goal g whereas the detailed design may only have plans for g's sub-goals. Additionally, there may be multiple plans to handle a given goal, some of which may not conform to the requirements. In order to precisely define what it means for a detailed design to conform to a requirements specification we introduce a notion of *traces* for a requirement specification and detailed design.

Due to the subjective nature of the design process a requirement may be captured in multiple ways. Hence, in general, just by looking at the sub-goals one cannot infer if the designer has specified an ordering between sub-goals or that she is indifferent towards it. For consistency and uniformity we will assume the following interpretations:

- Listing the parent goal implies that ordering of sub-goals does not matter; and
- Listing of sub-goals without a parent goal implies that ordering of sub-goals matter.

Requirement traces: Informally, we say a goal is *met* if the goal itself is posted, or all its sub-goals are posted in case of AND sub-goal type, or at least one of its sub-goals is posted in case of a OR sub-goal type. Formally, a goal g from a goal tree $\langle G, g_0, \mathcal{R}, \mu\rangle$ is *met* by a sequence of goals $g_1, \ldots, g_n$ if either one of the following holds: (i) $n = 1$ and $g_1 = g$; or (ii) if $\mu(g) = \mathsf{AND}$, then $\mathsf{sg}(g) = \{g_1, \ldots, g_n\}$; or (iii) if $\mu(g) = \mathsf{OR}$, then $n = 1$ and $g_1 \in \mathsf{sg}(g)$.

Conceptually, a trace for a requirements specification is *one* possible way in which an underlying scenario can be achieved. Formally, a *trace for a requirement specification* with scenario $S = \langle (o_1, r_1), \ldots, (o_n, r_n)\rangle$ and goal trees $\{T_1, \ldots, T_k\}$ is a sequence of pairs $\tau = (o'_1, r_1) \cdots (o'_m, r_m)$ with $m \geq n$ such that for each step type o_i there exists indices s_i and e_i where:

1. if $o_i.\mathsf{type} \in \mathsf{Percepts} \cup \mathsf{Actions}$, then $s_i = e_i$ and $o_i = o'_{s_i}$;
2. if $o_i.\mathsf{type} \in \mathsf{Goals}$, then o_i is met through $o'_{s_i} \cdots o'_{e_i}$;
3. for all indices it holds that $e_j = s_{j+1} - 1$, $s_1 = 1$, and $e_n = m$, where $1 \leq j < n$.

Informally, a trace for a requirements specification is a concatenation of sequences (with start index s_i and end index e_i) such that each sequence achieves a scenario step. If the step is a percept or an action then the sequence is just one element containing the same percept or action. If a scenario step is a goal, then the sequence is such that the goal in the scenario step is met as per the assumptions discussed before.

Design traces: We define a trace for a detailed design incrementally by introducing traces for a plan, an agent, and finally for a group of agents. The idea is to define a trace for a design agent as an interleaving of its plan traces such that each posted goal is handled by at most one plan (goals are posted internally in an agent). We define a trace for a design as a collection of design agent traces such that each posted message is handled by at most one plan in the receiver agent (messages are posted across agents).

A *trace for a plan* $p = \langle \mathsf{name}, \mathsf{trigger}, O\rangle$ is a sequence of the form $\tau = \mathsf{trigger}\cdot\mathsf{name}\cdot o_1, \cdots, o_n$ where $|O| = n$ and $o_1, \cdots, o_n$ is a *permutation* of elements in O. (An agent activates a plan by acknowledging its trigger, executing the plan body, and then posting the plan outputs one by one.)

Since an agent may have more than one active plan at a time, a trace for a design agent will consist of interleaved active plan traces with certain constraints. We define an interleaved trace resulting from two traces $\tau_1 = t_1^1 \cdots t_n^1$ and $\tau_2 = t_1^2 \cdots t_m^2$ to be a trace $\tau_{1+2} = t_1 \cdots t_{m+n}$ such that $\tau_{1+2}^{\uparrow\tau_1} = \tau_2$ and $\tau_{1+2}^{\uparrow\tau_2} = \tau_1$ where $\tau_{1+2}^{\uparrow\tau}$ is a trace obtained by projecting out τ from τ_{1+2}. Given a set of plan traces $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a *trace for a design agent* is an interleaved trace $\tau = t_1, \ldots, t_\ell$ over agent's plan traces Γ such that:

1. if $t_i = \mathsf{g}$ is a trigger in plan trace τ' and $\mathsf{g} \in \mathsf{Goals}$, then there exists $t_j = \mathsf{g}$ where g is output in a plan trace $\tau'' \neq \tau'$ where $\tau', \tau'' \in \Gamma$, and $j < i$;
2. for any two $t_i = t_j = g$, where $i \neq j$, $g \in \mathsf{Goals}$ and t_i, t_j are triggers for plan traces τ^1 and τ^2, there must exist $t_{i'} = t_{j'} = g$ with $i' \neq j', i' < i, j' < j$ such that $t_{i'}, t_{j'}$ are plan outputs in traces $\tau^{1'}$, $\tau^{2'}$, respectively.

Intuitively:(1) a goal should be posted before it can be handled; and (2) only one plan gets activated for a posted goal.

A design trace then is simply a set of traces, one for each design agent, where we allow an agent to be inactive if it does not have any active plans. Formally, a *design trace* for a set of k agents is a sequence $\tau = (t_1^1, \ldots t_1^k) \cdots (t_\ell^1, \ldots t_\ell^k)$ such that each element t_j^{agt} can either be ϵ (here ϵ denotes an empty token) or from design agent agt's trace with the following constraints:

1. for all agents i it holds that $t_m^i \cdots t_n^i$ is agent i's trace such that either $m = 1$ or $t_{m-1}^i = \epsilon$, and $n = \ell$ or $t_{n+1}^i = \epsilon$;
2. for each $t_m^i = \mathsf{msg}$ where $\mathsf{msg} \in \mathsf{Messages}$ is a trigger of agent i, there exists a unique $t_n^{i'} = \mathsf{msg}$ where $t_n^{i'}$ is an output in a plan of agent $i' \neq i$, $n < m$, agent i is msg's receiver and agent i' is msg's sender;
3. if $t_m^i = \mathsf{msg}$ is a message (that is, $\mathsf{msg} \in \mathsf{Messages}$) and in output of agent i's plan, then for an agent i' such that i' is msg's receiver and for all indices $m < j < n$ where $t_n^{i'} = \mathsf{msg}$ is a trigger for a plan in agent i''s trace it is the case that $t_j^{i'} \neq \epsilon$.

Intuitively:(1) an agent trace cannot have empty tokens; (2) each message is handled by at most one plan; (3) the receiver agent must handle a message posted for it (i.e., it cannot choose to stay idle in presence of a pending message).

3.1.1 Comparing requirements and design traces

Conceptually, a trace for a detailed design is said to conform with a requirements trace if the design trace contains elements from the requirements trace in the right order. Formally, a design trace $\tau_D = (t_1^1, \ldots t_1^k) \cdots (t_\ell^1, \ldots t_\ell^k)$ for a design D *conforms* to a trace $\tau_R = (o_1, r_1) \cdots (o_m, r_m)$ of requirements specification R if there exists a set of indices $j_1, \ldots, j_m$ such that for all $1 \leq i \leq m$ there exists agent agt where $t_{j_i}^{\mathsf{agt}} = o_i$ and r_i is in agt's roles. We say a detailed design D conforms to a requirements specification R if there exists a trace τ_D of D for which there exists a trace τ_R of R such that τ_D conforms to τ_R.

A design trace will generally be much longer in length than a scenario trace because a detailed design fleshes out *how* a particular requirement is achieved.

4 Model checking agent designs

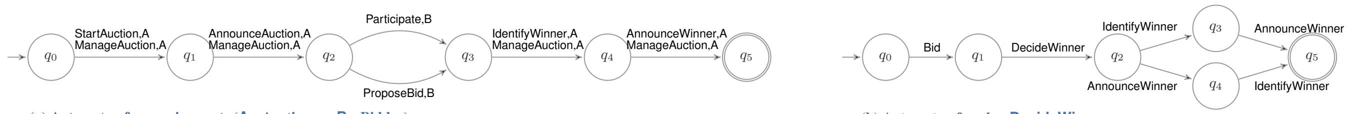
Conceptually, an ATL model (also known as concurrent game structure) consists of a set of agents that act concurrently in order for the game to progress. The game consists of a set of states that evolve based on agents' moves and one checks temporal formulae against these states to verify properties. For our conformance checking problem, the set of agents will consist of a requirements agent and a number of detailed design agents. The requirements agent will select its actions as per the underlying scenario and goal trees, whereas the design agents will move as per their active plans. In the ATL game structure we will match each design agent's action against the action of the requirements agent to check if a scenario step has been achieved. The objective then is to verify if all scenario steps have been achieved in the right order.

We do this in two steps: (i) we build two kinds of finite state automatons (see [16] for details on FSA.) one that will accept all traces for a requirements specification, and second that will accept all traces for a given plan; (ii) we use these automatons to build a concurrent game structure that will serve as our ATL model for model checking. A game state in our ATL model will consist of underlying states of these automatons (a requirements automaton and multiple plan automatons) and these states will be appropriately updated based on the agents' actions (that is, based on scenario steps achieved, plan triggers, and plan completions). Finally, to verify that a given detailed design conforms to the requirements specification, we will check the formula $\langle\langle \mathcal{A} \rangle\rangle \diamond \mathsf{final}$, where $\mathcal{A}$ is the set of all design agents and final captures the condition that the requirements automaton has reached its final state.

4.1 Requirements and agent plan automatons

We introduce two technical notations required for constructing the automatons. First, given a sequence $\tau = t_1 t_2 \cdots t_n t_{n+1}$ let a set of states for accepting τ indexed by a number be $\theta(\tau, i) = \{q_\epsilon^i, q_{t_1}^i, q_{t_1 t_2}^i, \ldots, q_{t_1 \cdots t_n}^i\}$. Intuitively, the role of states in $\theta(\tau, i)$ is to track the sequence τ and since we may need to track the same sequence more than once we index it by a number. Second, given a set of elements E, let the set of all possible sequences (that is, permutations) of length $|E|$ that can be generated from E be $\mathsf{perm}(E)$.

Next, we build an automaton such that its language is the set of traces for a given requirements specification. Formally, an automaton for a given scenario $S = \langle (o_1, r_1), \ldots, (o_n, r_n)\rangle$ and set of k goal trees $T_i = \langle G_i, g_{i0}, \mathcal{R}_i, \mu_i\rangle$, where $1 \leq i \leq k$, is a tuple $F_R = \langle Q, q_0, \Sigma, \delta, \{q_f\}\rangle$ where:



(a) Automaton for requirements (A:=Auctioneer, B:=Bidder)

(b) Automaton for plan DecideWinner

Figure 3: Automatons for the auction example.

1. $Q = \{q_{o_0}, q_{o_n}\} \cup \{q_o \mid \exists i\ S[i].\mathsf{type} = o\} \cup \{\theta(\tau, i) \mid \tau \in \mathsf{perm}(\mathsf{sg}(g)), \exists i (g = S[i].\mathsf{type}), g \in \mathcal{G}_S^{\wedge}\} \cup \{q_g \mid \exists i, j\ S[i].\mathsf{type} = g', g \in \mathsf{sg}(g'), g' \in \mathcal{G}_S^{\vee}\}$ where $\mathcal{G}_S^{\wedge}$ ($\mathcal{G}_S^{\vee}$) is the set of AND (OR) goals in scenario S;
2. q_{o_0} is the initial state and q_f is the final state where $q_f = q_{o_n}$;
3. $\Sigma = \{(o, r) \mid \exists i\, S[i] = (o, r)\} \cup \{(o, r) \mid \exists i, g\, S[i] = (g, r), o \in \mathsf{sg}(g)\}$ is the alphabet consisting of elements in traces of requirements R; and
4. $\delta \colon Q \times \Sigma \to Q$ is the transition function where $\delta(q, \sigma) = q'$ if:
   (a) $\sigma = (o_i, r_i)$, $q = q_{o_{i-1}}$, $q' = q_{o_i}$ for $1 \leq i \leq n$;
   (b) $\sigma = (o_i', r_i)$ such that there exists $o_i \in \mathcal{G}_S^{\vee}$, $o_i' \in \mathsf{sg}(o_i)$, $q = q_{o_{i-1}}$, $q' = o_i$ where $1 \leq i \leq n$;
   (c) $\sigma = (o_i', r_i)$ such that there exists $o_i \in \mathcal{G}_S^{\wedge}$, $o_i' \in \mathsf{sg}(o_i)$ and one of the following holds:
      i. $q = q_{o_{i-1}}$ and $q' = q^i_{o_i'}$;
      ii. $q = q^i_{\tau}$ and $q' = q_{o_i}$ where $\tau \cdot o_i' \in \mathsf{perm}(\mathsf{sg}(o_i))$;
      iii. $q = q^i_{\tau}$ and $q' = q^i_{\tau'}$ where $\tau \cdot o_i' = \tau'$ and $\tau' \cdot \tau'' \in \mathsf{perm}(\mathsf{sg}(o_i))$ for some $|\tau''| \geq 1$.

Intuitively, the set of states (1) of an automaton for a requirements specification consists of a state per each scenario step and additional states to cater for sub-goals where a scenario step has AND sub-goals. In addition, we index the states required for AND sub-goals (with the step number) as the same goal may appear more than once in a scenario. The alphabet (3) of the automaton consists of requirements trace elements, that is, a pair consisting of step type and step role. The first condition (4a) of the transition function connects each scenario step sequentially, the second (4b) provides alternatives where an OR goal could be achieved by one of its sub-goals, and the third (4c) caters for each permutation of an AND goal. Figure 3 (a) shows the finite state automaton for the auction requirements specification described in Figure 1b.

Theorem 1. *The language of automaton F_R for requirements specification R having a scenario S and goal trees $\{T_1, \ldots, T_k\}$ is the set of all possible traces of R.*

Proof (sketch). Let $w = \sigma_1 \cdots \sigma_\ell \in L(F_R)$, automaton $F_R = \langle Q, q_0, \Sigma, \delta, \{q_f\}\rangle$ and scenario $S = \langle (o_1, r_1), \ldots, (o_n, r_n)\rangle$. Hence, there exists a sequence of transitions $\lambda = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_\ell} q_\ell$ where $\delta(q_{i-1}, \sigma_i) = q_i$ for $1 \leq i \leq \ell$ and $q_\ell = q_f$. The sequence λ contains states $q^1, \ldots, q^n$ such that $q^i = q_{o_i}$ for $1 \leq i \leq n$ (from condition 4). From the states $q^1, \ldots, q^n$ one can extract index pairs (s_j, e_j) where $\lambda[s_j] = q^j$ and $e_j = s_{j+1} - 1$ where $1 \leq j < n$ and $e_n = \ell$. Each sub-sequence $\lambda[s_j] \cdots \lambda[e_j]$ achieves step o_j, for $1 \leq j \leq \ell$; hence w is a trace for requirements R. For the other side, assume $\tau = (o_1', r_1') \cdots (o_m', r_m')$ is a trace for requirements R. Then, there exists indices s_i, e_i for each step o_i, where $1 \leq i \leq m$, such that sequence $o_{s_i}' \cdots o_{e_i}'$ achieves step o_i. If o_i is a percept or action, for $1 \leq i \leq m$, then there exists transition from q_{o_i} to $q_{o_{i+1}}$ on symbol (o_i, r_i) (condition 4a in definition of F_R). If o_i is a goal, for $1 \leq i \leq m$, then either (i) $s_i = e_i$ and $\tau[s_i] = o_i$ (condition 4a in definition of F_R), or (ii) $s_i = e_i$, $\tau[s_i] \in \mathsf{sg}(o_i)$, and $\mu(g) = \mathsf{OR}$ (condition 4b in definition of F_R), or (iii) o_i is met through sequence $\tau[s_i] \cdots \tau[e_i]$ (condition 4c in definition of F_R). Observe that the first condition in transition function of F_S caters for condition (i), and second and third condition in transition function of F_S cater for condition (ii) above. Combined with the fact that q_{o_n} is the final state, we get that $\tau \in L(F_R)$. □

Next we construct an automaton for a plan such that the automaton will accept only the possible traces of the plan. An automaton for a plan $p = \langle n, t, O\rangle$ is a tuple $F_p = \langle Q, q_0, \Sigma, \delta, q_f\rangle$ where:

1. $Q = \{q_0, q_1, q_f\} \cup \{\theta(\tau, 1) \mid \tau \in \mathsf{perm}(O)\}$ is set of states;
2. q_0 and q_f are the initial and final states, respectively;
3. $\Sigma = \{n, t\} \cup O$ is the alphabet;
4. $\delta \colon Q \times \Sigma \to Q$ is the transition function where $\delta(q, \sigma) = q'$ if:
   (a) $q = q_0$, $\sigma = t$, and $q' = q_1$; or
   (b) $q = q_1$, $\sigma = n$, and $q' = q^1_{\epsilon}$; or
   (c) $q = q^1_{\tau}$, $\sigma \in O$, and $q' = q^1_{\tau \cdot \sigma}$ where $\tau \cdot \sigma \cdot \tau' \in \mathsf{perm}(O)$ for some $|\tau'| \geq 1$; or
   (d) $q = q^1_{\tau}$, $\sigma \in O$, and $q' = q_f$ where $\tau \cdot \sigma \in \mathsf{perm}(O)$.

The alphabet of the automaton is the elements of the plan.The transition function ensures that any trace accepted by the automaton starts with the plan trigger (4a), followed by its name (4b), and then any legal permutation of the plan's outputs (4c, 4d). Figure 3 (b) shows the finite state automaton for the DecideWinner plan of the Auctioneer. For the purpose of implementation the number of states of the automaton of plan $p = \langle n, t, O\rangle$ will be exponential in $|O|$.

Theorem 2. *The language of the automaton F_p for a plan $p = \langle \mathsf{name}, \mathsf{trigger}, O\rangle$ is the set of all traces of p.*

Proof (sketch). Observe that any word in the language of automaton F_p has the form $\mathsf{trigger} \cdot \mathsf{name} \cdot \lambda$ where λ is from the set $\mathsf{perm}(O)$. Hence, any word in the language of F_p is a trace of plan p. Similarly, it can be shown that the first two symbols in a trace τ of plan p result F_p to transition from its initial state to state q^1_{ϵ}; and subsequent symbols of τ cause F_p to transition from q^1_{ϵ} to q_f. □

These automatons provide a cleaner mapping to build the ATL game structure and also provide a straightforward translation to ISPL encoding for MCMAS [18] (please see the Appendix for the ISPL encoding for the auction example.)

4.2 ATL concurrent game structure

The ATL model that we will build consists of a requirements agent and a number of design agents. Observe that the automaton for a given scenario encapsulates all legal scenario traces (see Theorem 1), and hence models the behavior of the requirements agent. A design agent on the other hand consists of multiple plans, and therefore its behavior will be modelled by multiple automatons, one for each of its (potential) plans that might get activated.

A concurrent game structure for an agent design consists of an ATL requirements agent and one ATL agent per detailed design agent. The requirements agent executes actions as per the automaton obtained from the requirements specification and design agents execute actions as per automatons of their active plans. A game state captures the completion status of the requirements specification along with the status of plan instances. A key feature in our reduction is that the state of requirements specification progresses only when a design agent executes an action as expected by the requirements. Intuitively, this implies that a next step in a (partial) requirements trace has been achieved by one of the detailed design agents. We assume that the percept in the first scenario step is posted by default, and hence a plan that can handle it will be the first to get activated. Formally, given an automaton $F_R = \langle Q^R, q_0^R, \Sigma^R, \delta^R, \{q_f^R\}\rangle$ for a requirements specification R (consisting of a scenario and a set of goal trees) and a detailed design D containing k design agents $agt_i = \langle n_i, R_i, Pl_i\rangle$ and automatons $F_p = \langle Q^p, q^0, \Sigma^p, \delta^p, \{q^f\}\rangle$ for each plan $p \in Pl$ (let $Pl = \cup_{1\leq i\leq k} Pl_i$), a concurrent game structure for R and D is a tuple $\mathcal{M}_{\langle R,D\rangle} = \langle\{\mathsf{Req}, n_1, \dots, n_k\}, Q, \mathcal{P}, Act, d, \mathcal{V}, \delta\rangle$ where:

1. There are $k+1$ agents: Req is the requirements agent, and $n_1, \dots, n_k$ are design agents (one per detailed design agent);
2. States Q consist of the following finite range functions:
   (a) $\mathsf{scn} \in Q^R$ (Q^R is set of states for automaton F_R);
   (b) $\mathsf{plan}_i^{\mathsf{id}} \in Q^p \cup \{\mathsf{inact}\}$ where $\mathsf{id} \in \Delta(p), p \in Pl_i$ where $1 \leq i \leq k$, and inact is used to capture if a plan is inactive;
3. $\mathcal{P}$ is the set of propositions asserting value assignments to the above defined functions and $\mathcal{V}$ is the mapping from a game state q to the values returned by the above defined functions. For convenience, we will write $(\mathsf{scn}(q) = q_r) \in \mathcal{V}(q)$ as $\mathsf{scn}(q) = q_r$.
4. $Act = \{a \mid \exists r(a, r) \in \Sigma^R\} \cup \{\mathsf{id} \cdot a \mid a \in \Sigma^p, \mathsf{id} \in \Delta(p), p \in Pl\} \cup \{\mathsf{fin}, \mathsf{nop}\}$ is the set of domain actions, where Σ^R is the alphabet for the automaton F_R, Σ^p is the alphabet for the automaton for plan p, fin and nop are special actions to denote that a scenario has finished and an agent has no active plans, respectively. The action $\mathsf{id} \cdot a$ denotes symbol a of alphabet Σ^p prefixed by id of its plan instance p. Given an annotated action $\mathsf{id} \cdot a$, let $\mathsf{action}(\mathsf{id} \cdot a) = a$.
5. $d(j, q)$ defines the moves available for agent j in state q:
   (a) Requirements agent ($j = \mathsf{Req}$):
$$d(j,q) = \begin{cases} \{a \mid \exists r, q^R \langle \mathsf{scn}(q), (a, r), q^R\rangle \in \delta^R\}, & \text{if } \mathsf{scn}(q) \neq q_f^R \\ \{\mathsf{fin}\}, & \text{otherwise.}\end{cases}$$
   (b) Design agents ($j \in \{n_1, \dots, n_k\}$):
$$d(j,q) = \begin{cases} \mathsf{next\text{-}actions}(j,q), & \text{if } |\mathsf{next\text{-}actions}(j,q)| > 0 \\ \{\mathsf{nop}\}, & \text{otherwise.}\end{cases}$$
   where, $\mathsf{next\text{-}actions}(j, q) =$
   $\{\mathsf{id}\cdot a \mid \exists q_p \langle \mathsf{plan}_j^{\mathsf{id}}(q),\ \mathsf{action}(\mathsf{id}\cdot a), q_p\rangle \in \delta^p, \mathsf{id} \in \Delta(p)\}$
6. $\delta : Q \times Act^k \to Q$ is the transition function such that $\delta(q, \vec{a}) = q'$, where $\vec{a} = a_r, a_1, \dots, a_k$ is the move vector containing actions for requirements and design agents, and q' is as follows:
   (a) Requirements: updated if a design agent acts as expected:
$$\mathsf{scn}(q') = \begin{cases} s = \delta^R(\mathsf{scn}(q), (\ \mathsf{action}(a_i), \mathsf{r})), & \text{if } s \text{ is defined} \\ \quad \text{for some } 1 \leq i \leq k \text{ where } \mathsf{r} \in R_i; & \\ \mathsf{scn}(q), \text{otherwise} & \end{cases}$$
   (b) Plans: updated as follows ($j \in \{n_1, \dots, n_k\}$):
      i. if $\mathsf{plan}_j^{\mathsf{id}}(q) = q_0$ and $a_j = \mathsf{id} \cdot a$, then $\mathsf{plan}_j^{\mathsf{id}}(q') = q_1$ and $\mathsf{plan}_j^{\mathsf{id'}}(q') = \mathsf{inact}$ where $\mathsf{history}(\mathsf{id}) = \mathsf{history}(\mathsf{id'})$ and $\mathsf{active}(\mathsf{id}) \cdot \mathsf{trigger} = \mathsf{active}(\mathsf{id'}) \cdot \mathsf{trigger}$;
      ii. if $\mathsf{plan}_j^{\mathsf{id}}(q) = q_1$ and $a_j = \mathsf{id} \cdot a$, then $\mathsf{plan}_j^{\mathsf{id}}(q') = q_\epsilon^1$;
      iii. if $\mathsf{plan}_j^{\mathsf{id}}(q) = q^p$ and $a_j = \mathsf{id}\cdot a$, then $\mathsf{plan}_j^{\mathsf{id}}(q') = \delta^p(q^p, a)$ where $p = \mathsf{active}(\mathsf{id})$, and:
         A. if $a \in \mathsf{Goals}$, then $\mathsf{plan}_j^{\mathsf{id'}}(q') = q_0$ where $\mathsf{id'} = \mathsf{id} \cdot \mathsf{name}$ such that $p = \langle \mathsf{name}, a, O\rangle \in Pl_j$;
         B. if $a \in \mathsf{Messages}$, then $\mathsf{plan}_i^{\mathsf{id'}}(q') = q_0$ where $\mathsf{id'} = \mathsf{id} \cdot \mathsf{name}$ such that $p = \langle \mathsf{name}, a, O\rangle \in Pl_i$ where $i \neq j$;
      iv. if $a_j = \mathsf{nop}$, then $\mathsf{plan}_j^{\mathsf{id}}(q') = \mathsf{plan}_j^{\mathsf{id}}(q)$;

Given a requirements specification R and detailed design D, the states of the concurrent game structure $\mathcal{M}_{\langle R,D\rangle}$ consist of states from the automaton of requirements R, and states from the automatons of possible plan instances of design agents in D. The function scn returns the current state of automaton F_R whereas functions $\mathsf{plan}_j^{\mathsf{id}}$ returns the current state of plan $\mathsf{active}(\mathsf{id})$ of design agent j (2). Moves of the requirements agent (5a) from a state q consist of scenario step types that are part of outgoing transitions of current state of automaton F_R (that is, $\mathsf{scn}(q)$). If there are no outgoing transitions (because F_R has reached its final state) then the requirements agent's moves consist of the special action fin, implying that the requirements have been met. Moves of a design agent (5b) consists of a union of all possible symbols that are part of outgoing transitions of automatons of its active plans. A design agent j does the special action nop in a state q if it does not have any active plans, that is, the set of $\mathsf{next\text{-}actions}(j, q)$ is empty. The transition function of game structure $\mathcal{M}_{\langle R,D\rangle}$ models how the underlying automaton states are updated (6). For practical purposes, encoding a concurrent game structure to ISPL [18] is straightforward. We present the ISPL encoding for the auction example in the Appendix.

4.3 Verifiable design properties

The ATL model $\mathcal{M}_{\langle R,D\rangle}$ for a requirements specification R and detailed design D can be now used to verify design time properties such as *conformity* and *coverage*. Requirements conformity (as formalized in Section 3.1.1) deals with checking if a detailed design can achieve a given requirements specification, whereas coverage signifies in how many different ways can a requirements specification be achieved.

Requirements conformity: Given an ATL model $\mathcal{M}_{\langle R,D\rangle}$ for a requirements specification R and detailed design D we are interested in checking whether design D conforms to requirements R. Observe that the requirements agent in the ATL model has limited freedom in its behavior: it repeatedly selects one of its expected scenario steps until one of the agents with an appropriate role executes an expected action. What it implies is that we are trying to match a trace of detailed design with a trace of requirement. Hence, in order to check the conformity of requirements with detailed design we model check the formula $\varphi = \langle\langle\mathcal{A}\rangle\rangle \diamond \mathsf{final}$ where $\mathcal{A}$ is the set of all design agents in the ATL model and final is defined as $\mathsf{sch} = q_f^R$ where q_f^R is the final state of scenario automaton F_R. Finally, we check this formula from a state q_I (in the game structure) such that the requirements automaton and the plans that will handle that percept in the first scenario step are in their initial states, and all other design agent plan instances have the value inact. Formally, given a ATL model $\mathcal{M}_{\langle R,D\rangle}$ for requirements R with scenario $S = \langle(o_1, r_1), \dots, (o_n, r_n)\rangle$ and detailed design D it is the case that: (i) $\mathsf{sch}(q_I) = q_0^R$ where q_0^R is the initial state of requirements automaton F_R, (ii) $\mathsf{plan}_j^{\mathsf{id}}(q_I) = q^0$ for all design agents j such that $\mathsf{active}(\mathsf{id}).\mathsf{trigger} = o_1$, and (iii) $\mathsf{plan}_j^{\mathsf{id}}(q_I) = \mathsf{inact}$ for all design agents j such that $\mathsf{active}(\mathsf{id}) \cdot \mathsf{trigger} \neq o_1$.

Theorem 3. *A detailed design D conforms to requirements specification R iff $\mathcal{M}_{\langle R,D\rangle}, q_I \models \langle\langle\mathcal{A}\rangle\rangle \diamond (\mathsf{scn} = q_f^R)$ where $\mathcal{A}$ is the set of all detailed design agents in $\mathcal{M}_{\langle R,D\rangle}$ and q_f^R is the final state of requirements automaton F_R.*

PROOF (SKETCH). Suppose $\mathcal{M}_{\langle R,D\rangle}, q_I \models \langle\langle\mathcal{A}\rangle\rangle \diamond (\mathsf{scn} = q_f^R)$. Hence, there exists a collective strategy F_A, one for each agent in $\mathcal{A}$ such that $\diamond(\mathsf{scn} = q_f^R)$ is satisfied in all computations $\lambda \in \mathit{out}(q^I, F_A)$. Let $\lambda = q_0 \cdots q_\ell$ where $q_0 = q^I$, $\mathsf{sch}(q_\ell) = q_f^R$, and $\vec{a}_1 \cdots \vec{a}_\ell$ be the sequence of move vectors such that $\delta(q_{i-1}, \vec{a}_i) = q_i$, for $1 \leq i \leq \ell$, where δ is the transition function of $\mathcal{M}_{\langle R,D\rangle}$. Let each $\vec{a}_i = \langle a_i^r, a_i^1, \ldots, a_i^k\rangle$ where a_i^r is the action of requirements agent and $a_i^1, \ldots, a_i^k$ are actions of the design agents. From the definition of δ (condition 6) one can observe that by construction $(a_1^1, \ldots, a_1^k) \cdots (a_\ell^1, \ldots, a_\ell^k)$ is the design trace that conforms to the scenario trace $\tau = (a_1'^r, r_1) \cdots (a_m'^r, r_m)$ where τ is extracted from $a_1^r \cdots a_\ell^r$ by removing adjacent duplicate elements. For the other side, given a trace $\tau_D = (a_1^1, \ldots, a_1^k) \cdots (a_\ell^1, \ldots, a_\ell^k)$ of design that conforms to a trace $\tau_R = (a_1^r, r_1) \cdots (a_m^r, r_m)$ of requirements R (that is, there exists indices $w_1, \ldots, w_m$ where each step type was achieved for scenario), one can construct a path $\lambda = q_0 \cdots q_\ell$ following the transition function δ of $\mathcal{M}_{\langle R,D\rangle}$ such that (i) actions of design agents between states q_{i-1} and q_i are $\tau_D[i]$, (ii) requirements agent executes action (o_i, r_i) at w_i and repeats its action between each w_i's, for $1 \leq i \leq m$, (iii) $q_0 = q^I$, and (iv) $\mathsf{scn}(q_\ell) = q_f^R$. $\square$

Design nonconformity: Even though a given design may conform to a scenario, it may still contain traces that do not achieve any scenario trace. These design traces are not necessarily faulty as the ordering of plan outputs is not specified in the detailed design. Nonetheless, such design traces should be highlighted to the designer to spot potential errors in the agent design. Interestingly, we can extract all such traces from the winning states of the concurrent game structure $\mathcal{M}_{\langle R,D\rangle}$ by checking the formula $\tilde{\varphi} = \langle\langle\mathcal{A}\rangle\rangle\Box\neg\mathsf{final}$ where $\mathcal{A}$ is the set of all design agents in the ATL model. In general, such traces will have incorrect ordering of scenario steps or plan activations that do not achieve a scenario step. Intuitively, one extracts such traces by extracting actions of design agents from move vectors responsible for transitions in the winning states. We omit the details of the approach to extract such traces due to space limitations.

Requirements coverage: Given the set of maximal winning states $[\varphi]$ in ATL model $\mathcal{M}_{\langle R,D\rangle}$ one can extract *all* the requirements specification traces that can be achieved by at least one design trace. Since the language of automaton F_R ($L(F_R)$) is all the possible requirements traces, one can compute the words in $L(F_R)$ that are not in the traces extracted from the winning game states to denote requirements traces that a designer might want to cater for in the detailed design. For the discussed design properties, the set of maximal winning states serves as a model from which all the relevant design and requirements traces can be extracted.

5 Scalability Evaluation

In this section, we present initial scalability results to show the feasibility our approach. We generated test cases with a single scenario and multiple goal trees, and a single detailed design agent. The detailed designs were varied by: (i) goals per plan (g/p), (ii) plans per goal (p/g), and (iii) the depth of the design (d). (i) and (ii) were varied from 1 to 3, and depth was sequentially changed from 1 to 8, resulting in 72 test cases. Each scenario, with the number of steps matching the depth of the design, was constructed such that its corresponding design always conforms to it. These designs were then verified for conformance in McMAS [18] and their execution time was recorded from its output.[2] We had put an upper time limit of one hour.

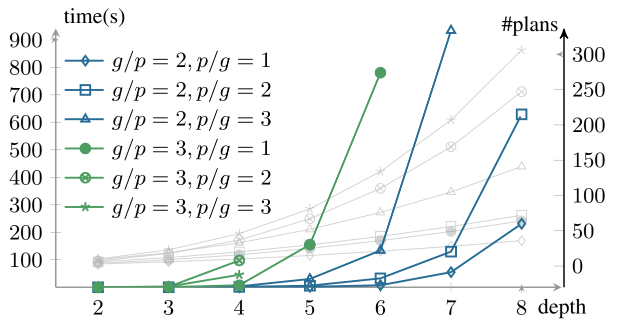


Figure 4: Scalability analysis of approach (see text for details).

In addition, to control the branching factor the percentage of target goals and plans at a depth d was always $1/d$. The rest of the goals/plans were simply one p/g and one g/p. If we do not control the branching factor, the number of possible traces (due to interleavings) grows at a rate factorial to the depth and branching. Given that in practice, not all plans and goals have high branching factors, our approach provides a balance between relevance and reliability.

Figure 4 shows the scalability results of our approach where on the x-axis is the depth, on the left y-axis is the time in seconds, and on the right y-axis is the number of plans in the design. For example, the line with square marks has: $g/p = 2$ and p/g=2; each point in the line is a test case; the last point is test case of depth 8 with an average execution time of more than 600 seconds and more than 70 plans. Note that the colored lines denote time and the faded gray lines denote the number of plans.

All cases with depth less than 4 were trivial to solve in terms of time (maximum time of 2.5 sec. for case 3-3-3). With depth 4 onwards the branching contributes significantly and as it appears from Figure 4 (respective cases with $g/p = 3$ have higher execution times than cases with $g/p = 2$) that goals per plan affect the execution time more than plans per goal. However, we point out that our approach tests one scenario at a time (along with its goal trees) against its relevant detailed design. A design that caters for one scenario, generally, will not have a large number of plans. Even so, our approach demonstrates that it can handle large design components within reasonable time bounds for design time verification.

To compare our technique with the one presented in [2], we requested the authors to run our test designs. For smaller test cases their approach was faster than ours (0.11 sec vs 1.2 sec. for case 2-2-4 and 0.13 vs 1.8 sec. for case 3-2-3). However, by increasing the depth by one in both test cases the trace based approach lagged considerably. For example, it took 70.57 sec. for the case 2-2-5 and it was still executing after 4.5 hours for the case 3-2-4. The same in our approach took 5.5 sec. and 97.5 sec., respectively. This is expected, as mentioned, since our approach relies on model checking, it scales significantly better than the trace based approach.

6 Conclusion

This paper presented a framework for formally verifying agent design models with respect to the requirements specifications. We show how informal and semi-structured design artefacts can be transformed into formal structures that can be model checked and we provide details on how to model check via ATL games.

[2] The system had a quad core i7 3.4GHz CPU with 8GB RAM.

```
1   Agent Requirements
2     Vars:
3           state: {s0, s1, s2, s3, s4, s5};
4     end Vars
5     Actions = {Ann_Auction, Start, Manage, finish,
6     Bid, Ann_Winner, Id_Winner, nop};
7     Protocol:
8           state = s0: {Start_Auction};
9           state = s1: {Ann_Auction, Manage};
10          state = s2: {Bid};
11          state = s3: {Id_Winner, Manage};
12          state = s4: {Ann_Winner, Manage};
13          state = s5: {finish};
14          Other: {nop};
15    end Protocol
16    Evolution:
17          state = s1 if (state = s0) and
18            (( Auctioneer.Action = Start));
19          state = s2 if (state = s1) and
20            (( Auctioneer.Action = Ann_Auction));
21          state = s2 if (state = s1) and
22            (( Auctioneer.Action = Manage));
23          state = s3 if (state = s2) and
24            (( P1.Action = Bid) or (P2.Action = Bid));
25          state = s4 if (state = s3) and
26            (( Auctioneer.Action = Id_Winner));
27          state = s4 if (state = s3) and
28            (( Auctioneer.Action = Manage));
29          state = s5 if (state = s4) and
30            (( Auctioneer.Action = Ann_Winner));
31          state = s5 if (state = s4) and
32            (( Auctioneer.Action = Manage));
33    end Evolution
34  end Agent
```

Figure 5: ISPL encoding for the requirements agent.

```
Agent Auctioneer
  Vars:
    P_NewAuction: {..};
    P_Broadcast: {..};
    P1_DecdWin: {init, s0, s1, s2, s3, s4, s5};
    P2_DecdWin: {..};
    P1_NotfyWin: {..};
    P2_NotfyWin: {..};
    ...
  end Vars
  Actions = {NewAuction, Auction, finish, Ann_Winner,
        Id_Winner, nop, NotfyWin, Ann_Auction, Start,
        LogBid, Bid, WinnerAnnouncement, Broadcast,
        DecdWin, NewAuction, Bid, NomWin};
  Protocol:
    P1_DecdWin = DecdWin0: {Bid};
    P1_DecdWin = DecdWin1: {DecdWin};
    P1_DecdWin = DecdWin2: {Id_Winner, Ann_Winner};
    P1_DecdWin = DecdWin4: {Ann_Winner};
    P1_DecdWin = DecdWin5: {Id_Winner};
    ...
  end Protocol
  Evolution:
    P1_DecdWin=s1 if (P1_DecdWin=s0) and (Action=Bid);
    P1_DecdWin=s2 if (P1_DecdWin=s1) and (Action=DecdWin);
    P1_DecdWin=s3 if (P1_DecdWin=s4) and (Action=Ann_Winner);
    P1_DecdWin=s4 if (P1_DecdWin=s2) and (Action=Id_Winner);
    P1_DecdWin=s5 if (P1_DecdWin=s2) and (Action=Ann_Winner);
    ...
  end Evolution
end Agent
```

Figure 6: ISPL encoding for the Auctioneer agent.

While there is existing work on formally verifying the correctness of agent programs via model checking (e.g. [13, 10, 29] and theorem proving (e.g. [25]) to the best of our knowledge this paper is the first to propose a formal verification of detailed agent designs, even prior to any implementation. This allows early detection of errors which is well known to save costs in software development. With respect to agent implementations recent work by Zhang et al.[21] has shown that, across 14 different agent programs, 34% of errors were due to faults in the design.

Although, in general, there is a lack of much work on checking the correctness of detailed agent designs with respect to the requirements specification, recent work by Abushark et al. [2] and Thangarajah et al. [27] are exceptions. We have highlighted some of the limitations of [2], and shown empirically that our approach is significantly more computationally effective as the parallelism within the design increases.

The problem we address also bears resemblance to specifications modelled using modal transition systems [5] in the sense that agent designs have parts that are "required" and parts that are "allowed". This required part is checked with respect to a scenario. However, due to the presence of goal hierarchies/decomposition we usually do not get a one to one mapping between the transitions.

Our main purpose in this paper was to demonstrate the use of *a* formal verification approach and we chose ATL for two key reasons. First, ATL model checking supports synthesis of strategies and this is essential for checking properties such as requirements coverage. Second, we are currently extending this framework to verifying AUML protocols and in that setting we require the ability to model an environment for messages that arise external to an agent. In addition, ATL's multi agent modelling allows for natural extensions that can be built in our framework, such as verifying agent capabilities. This work presents a base from which various other aspects of the agents' designs could be formally verified.

7 Appendix: ISPL Encoding

In this section we present the ISPL [18] encoding for the auction example used in the paper. Briefly, an agent in ISPL is modelled by four key elements: i) set of local states, ii) set of actions that the agent can execute, iii) a protocol that defines which actions are legal to execute based on its state, and iv) an evolution function that defines how states evolve. An ISPL file consists of the agent definitions, the winning condition, the initial states, and the coalition formula to check. In our case, the winning condition in the ISPL encoding is simply the last state of the requirements agent. In our auction example this is specified as `win if Requirements.state = s5;`. The formula we check for verifying requirements is `<g1> F win;` where `g1` is the coalition of agents.

The encoding for our auction example consists of 4 agents, a requirements agent named `Requirements` that models the underlying scenario, an auctioneer agent named `Auctioneer`, and two bidder agents named `P1` and `P2` (not shown here). Figure 5 shows the requirements agent. Observe the straightforward translation from the scenario (please see Figure 1b) in the agent design to its automaton (as shown in Figure 3) and finally to an ISPL encoding. The critical part in the encoding for the requirements agent is its evolution function. The requirements agent changes its state only when an expected action is executed by the correct design agent. For example, the requirements agent will update its state from `s2` to `s3` only when one of the bidder agents sends their bid (lines 23-24 in Figure 5).

Figure 6 shows the (partial) code for the auctioneer design agent. The state variables of design agents consist of variables to track the plans that the agent has. For example, the auctioneer agent has plans to start an auction, broadcast the announcement, handle bids, notify winner, etc. Multiple copies of a plan are required where its trigger can occur multiple times. For example, since multiple agents can send their bids, the auctioneer agent has multiple copies of the DecideWinner plan, one per bidder agent (that is, `P1_DecdWin` and `P2_DecdWin` in Figure 6). Also observe that the encoding for a design agent consists of an aggregation of plans it has and the plan automatons built earlier provide a clean way to map these to ISPL. The evolution function in the encoding of a design agent keeps a track of its active plans and allows the agent to progress one plan at at a time. The resulting behavior of a design agents emerges from the possible ways the active plans can be interleaved.

REFERENCES

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

[2] Yoosef Abushark, Michael Winikoff, Tim Miller, James Harland, and John Thangarajah. Checking the correctness of agent designs against model-based requirements. In *ECAI 2014*, volume 263, pages 953–954, Prague, 2014. IOS Press.

[3] N. Alechina, M. Dastani, B. S. Logan, and John-Jules Meyer. A logic of agent programs. pages 795–800, 2007.

[4] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, (49):672–713, 2002.

[5] Adam Antonik, Michael Huth, Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. 20 years of modal and mixed specifications. *Bulletin of the European Association for Theoretical Computer Science*, (95), 2008.

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

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

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

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

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

[11] M. Dastani and W. Jamroga. Reasoning about strategies of multi-agent programs. pages 997–1004, 2010.

[12] S. DeLoach, L. Padgham, J. Thangarajah, A. Perini, and A. Susi. Using three AOSE toolkits to develop a sample design. *IJAOSE*, 3(4):416–476, 2009.

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

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

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

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson Addison-Wesley, 3 edition, 2007.

[17] Wojciech Jamroga and Wojciech Penczek. Specification and verification of multi-agent systems. In *Lectures on Logic and Computation*, pages 210–263. Springer, 2012.

[18] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. pages 682–688, 2009.

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

[20] CuD. Nguyen, Anna Perini, Carole Bernon, Juan Pavn, and John Thangarajah. Testing in multi-agent systems. In Marie-Pierre Gleizes and JorgeJ. Gomez-Sanz, editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 180–190. Springer Berlin Heidelberg, 2011.

[21] 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.

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

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

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

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

[26] Steven Shapiro, Y Lespérance, and HJ Levesque. The cognitive agents specification language and verification environment. In *Specification and Verification of Multi-agent Systems*, pages 289–315. Springer, 2010.

[27] John Thangarajah, Gaya Buddhinath Jayatilleke, and Lin Padgham. Scenarios for system requirements traceability and testing. In *Proceedings of 10th International Conference on Autonomous Agents and Multiagent Systems AAMAS 2011*, pages 285–292, Taipei, Taiwan, 2011.

[28] Michael Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*, pages 175–193. Springer, 2005.

[29] Nitin Yadav and Sebastian Sardina. Reasoning about BDI agent programs using ATL-like logics. volume 7519, pages 437–449, 2012.

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

[31] Zhiyong Zhang, John Thangarajah, and Lin Padgham. *Automated Testing for Intelligent Agent Systems*, pages 66–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.