Verification of multi-agent negotiations using the Alloy Analyzer

Rodion Podorozhny[1], Sarfraz Khurshid[2], Dewayne Perry[2], and Xiaoqin Zhang[3]

[1] Texas State University, San Marcos, TX 78666,
rp31@txstate.edu
[2] The University of Texas, Austin, TX 78712,
{khurshid, perry}@ece.utexas.edu
[3] The University of Massachusetts, North Dartmouth, MA 02747,
x2zhang@umassd.edu

Abstract. Multi-agent systems provide an increasingly popular solution in problem domains that require management of uncertainty and a high degree of adaptability. Robustness is a key design criterion in building multi-agent systems. We present a novel approach for the design of robust multi-agent systems. Our approach constructs a model of the design of a multi-agent system in Alloy, a declarative language based on relations, and checks the properties of the model using the Alloy Analyzer, a fully automatic analysis tool for Alloy models. While several prior techniques exist for checking properties of multi-agent systems, the novelty of our work is that we can check properties of coordination and interaction, as well as properties of complex data structures that the agents may internally be manipulating or even sharing. This is the first application of Alloy to checking properties of multi-agent systems. Such unified analysis has not been possible before. We also introduce the use of a formal method as an integral part of testing and validation.

1 Introduction

Multi-agent systems provide an increasingly popular solution in problem domains that require management of uncertainty and high degree of adaptability. Robustness is a key design criterion in building multi-agent systems.

A common definition of a multi-agent system (MAS) [26] stipulates that an agent is an autonomous, interacting and intelligent (i.e. optimizing its actions) entity. Any MAS is a distributed system but not every distributed system can be categorized as a MAS by the above mentioned definition.

Management of uncertainty via adaptability and an ability to provide a satisficing solution to otherwise intractable problems are distinguishing features of multi-agent systems compared to centralized or other distributed systems. An agent knows of a great variety of methods to solve their local tasks, and it can tailor a method of achieving a goal according to resource availability for data processing, information exchanges and sources of information. Agents, due to their interactions, are capable of influencing the choices of methods both by themselves and by other agents due to recognition of various kinds of relationships between their subtasks that can be generalized as redundancy,

facilitation and enabling [19]. Agents can decide the degree to which an environment state, their own state, and their partial knowledge about states of other agents influence the amount of their contribution to the solution of a task imposed on the whole MAS. Unlike components of other distributed systems, an agent can refuse a request or can choose not to answer. At the same time, other agents are prepared to deal with a possibility that their requests will be refused or not answered. This freedom of choice, in a way, defines an agent's autonomy and distinguishes it from a component in a conventional distributed system. Thus, due to the above mentioned capabilities, agents are able to adapt their solution methods to the dynamics of the environment [17].

Some MAS have explicit specifications of interaction protocols between the agents. There has been a plethora of work on verification of MAS systems. Such approaches as model-checking ([27], [21], [16], [3]), Petri-nets and situation-calculus [8] have been applied to MAS verification. The vast majority of recent work on MAS verification are various applications of model checking that take into account peculiarities of properties that are desired to be verified in a MAS. The peculiarities of such properties usually are a consequence of bounded rationality in agents. Thus the set of operators (modalities) for property specifications is often extended to include such operators as agent *beliefs, desires, intentions*. Once such additional operators are introduced, usually a method is suggested to map a property specification that uses these MAS-specific operators into a formalism understood by off-the-shelf model-checkers, e.g. into the propositional LTL.

Examples of properties might be: "every request for a quote is answered within 4 time steps" [3], "for all paths in each state if agent *Train1* is in the tunnel then agent *Train1* knows that agent *Train2* is not in the tunnel" [16], "when sender is about to send an acknowledgment then it knows that the receiver knows the value of the bit that was most recently sent" [21] and "some agent A_i eventually comes to believe that A_j intends that A_i believes variable *a* has the value 10" [27].

As we can see from these examples most properties are some sort of reachability properties on a state transition model of a MAS. The use of model checking for these properties is understandable since it is essentially an efficient brute-force global state transition graph reachability analysis. ConGolog [8] uses situation calculus which is also most suited for the specification and analysis of event sequences, not data structures.

One lightweight formal method that is particularly suitable for checking properties of data structures is Alloy.

Most of the prior applications of the Alloy Analyzer have abstracted away from properties of multi-threaded systems. We explore the use of Alloy in designing, testing and validating a class of distributed systems known as the *multi-agent systems* (MAS). In particular we focus on exploring the suitability of the Alloy Analyzer to checking structurally rich properties of MAS.

In case of a model checking approach one needs to generate a number of particular instances of data structures either by hand or by writing a dedicated generator. For complex data structures the size of such an enumeration can be prohibitively large. Moreover, writing a generator correctly can itself be error-prone [22]. In contrast, the Alloy approach allows verification of rich structural properties, such as acyclicity of a

binary tree, via capturing them in a simple first-order logic formula based on intuitive path expressions.

We explore an application of Alloy with its relational logic specification language to multi-agent systems specifically focusing on properties of data structures in addition to event sequences. We expect to be able to check properties of the following format: "if agent *A* receives a data structure that satisfies property ϕ then eventually agent *A* will enter state σ_a if it believes that agent *B* is in state σ_b", "if agent *A* is in state σ_a and its task structure τ_1 satisfies property ϕ_1 then on reception of data structure *m* (from agent *B*) agent *A* will modify τ_1 with some part of *m* such that τ_1 will preserve property ϕ_1" and so on.

We also propose the use of a formal method for checking actual behavior of a system as exhibited by its execution traces against a behavior of its model. This is done in addition to the usual application of a formal method for verification of the system's model. Thus we integrate a formal method into testing and validation activities of a software design and analysis process.

We make the following contributions:

- **Checking multi-agent systems.** We present an approach to check a *utility-based reasoning* multi-agent system using a lightweight formal method;
- **Alloy application.** We present a novel application of the Alloy tool-set in checking rich properties that represent structural constraints in a multi-threaded scenario; and
- **Adequacy checking.** Our approach allows checking the adequacy of a given test suite against a relational specification.

2 Brief overview of Alloy

As software systems steadily grow in complexity and size, designing such systems manually becomes more and more error-prone. The last few years have seen a new generation of design tools that allow formulating designs formally, as well as checking their correctness to detect crucial flaws that, if not corrected, could lead to massive failures.

The Alloy tool-set provides a software design framework that enables the modeling of crucial design properties as well as checking them. Alloy [13] is a first-order, declarative language based on relations. The Alloy Analyzer [15] provides a fully automatic analysis for checking properties of Alloy models.

The Alloy language provides a convenient notation based on path expressions and quantifiers, which allow a succinct and intuitive formulation of a range of useful properties, including rich structural properties of software. The Alloy Analyzer performs a bounded exhaustive analysis using propositional satisfiability (SAT) solvers. Given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy formula into a boolean formula in conjunctive normal form (CNF), and solves it using an off-the-shelf SAT solver.

The Alloy tool-set has been used successfully to check designs of various applications, such as Microsoft's Common Object Modeling interface for interprocess communication [5], the Intentional Naming System for resource discovery in mobile networks [1], and avionics systems [7], as well as designs of cancer therapy machines [14].

The Alloy language provides a convenient notation based on path expressions and quantifiers, which allow a succinct and intuitive formulation of a range of useful properties, including rich structural properties of software. Much of Alloy's utility, however, comes from its fully automatic analyzer, which performs a bounded exhaustive analysis using propositional satisfiability (SAT) solvers. Given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy formula into a boolean formula in conjunctive normal form (CNF), and solves it using an off-the-shelf SAT solver.

We present an example to introduce the basics of Alloy.

Let us review the following Alloy code for a DAG definition:

```
module models/examples/tutorial/dagDefSmall

sig DAG {
  root: Node,
  nodes: set Node,
  edges: Node -> Node
}
sig Node {}
```

The keyword `module` names a model. A `sig` declaration introduces a set of (indivisible) atoms; the signatures `DAG` and `Node` respectively declare a set of DAG atoms and a set of node atoms. The *fields* of a signature declare relations. The field `root` defines a relationship of type `DAG x Node` indicating that only one node can correspond to a DAG by this relationship. The absence of any keyword makes `size` a total function: each list must have a size. The field `nodes` has the same type as `nodes` but maps a DAG onto a set of nodes defining a partial function. Alloy provides the keyword `set` to declare an arbitrary relation. The field `edges` maps a DAG onto a relationship, i.e. on a set of tuples `Node x Node`, thus defining edges.

The following *fact* constrains a graph to be a DAG:

```
fact DAGDef {
  nodes = root.*edges
  all m: Node | m !in m.^edges
}
```

The operator '`*`' denotes reflexive transitive closure. The expression `root.*edges` represents the set of all nodes reachable from the root following zero or more traversals along the `edge` field. A universally quantified (`all`) formula stipulates that no atom m of signature `Node` can appear in traversals originating for that atom m. The operator `^` denotes transitive closure.

Here are some other common operators not illustrated by this example. Logical implication is denoted by '`=>`'; '`<=>`' represents bi-implication. The operator '`-`' denotes set difference, while '`#`' denotes set cardinality and '`+`' - set union.

To instruct the analyzer to generate a DAG with 6 nodes, we formulate an empty predicate and write a `run` command:

```
pred generate() {}

run generate for 6 but 1 DAG
```

The scope of 6 forces an upper bound of 6 nodes. The `but` keyword specifies a separate bound for a signature whose name follows the keyword. Thus we restrict a generated example to 1 DAG.

3 Subject system details

As the subject of our analysis we have chosen a cooperative multi-agent system with explicit communication and with a utility-based proactive planning/scheduling.

A multi-agent system is cooperative if it can be assumed that agents strive to collectively contribute to reaching some common goal. In such a cooperative MAS, agents are willing to sacrifice their local optimality of actions if they are convinced (e.g. via a negotiation) that such a sacrifice will help increase the global optimality of the combined actions in the whole MAS. For simplicity we also assume there are no malicious agents in the chosen MAS.

3.1 Property examples derived from requirements

We can describe several properties informally at this stage, before we fix the assumptions of the MAS design further.

Some of the informal properties that are likely to be useful for such a negotiation:

1. negotiation must terminate;
2. the utility of the agreed upon combination of schedules must eventually increase throughout the course of negotiation even though occasional decreases are allowed; i.e. the negotiation must eventually converge on some choice of schedules that provides a local optimum of the combined utility (here local is used in the sense of restrictions on action set and time deadline, not in the sense of local to a single agent);
3. if agent *B* (the one who is requested to do an additional task) agrees to accomplish the task at a certain point in negotiation then it cannot renege on that agreement in the course of subsequent negotiation (somewhat related to the need to converge); and,
4. the beliefs of one agent about an abstraction of partial state of another agent obtained as a result of negotiation should not contradict the actual state of that other agent.

3.2 Experiment design

The experiment design is illustrated as derivation relationships between the software process artifacts in Fig. 1. The system requirements are used to derive a test suite and specify the intended behavior as properties. The subject MAS system is run on the test suite thus producing traces. The Alloy model of the system includes the representation

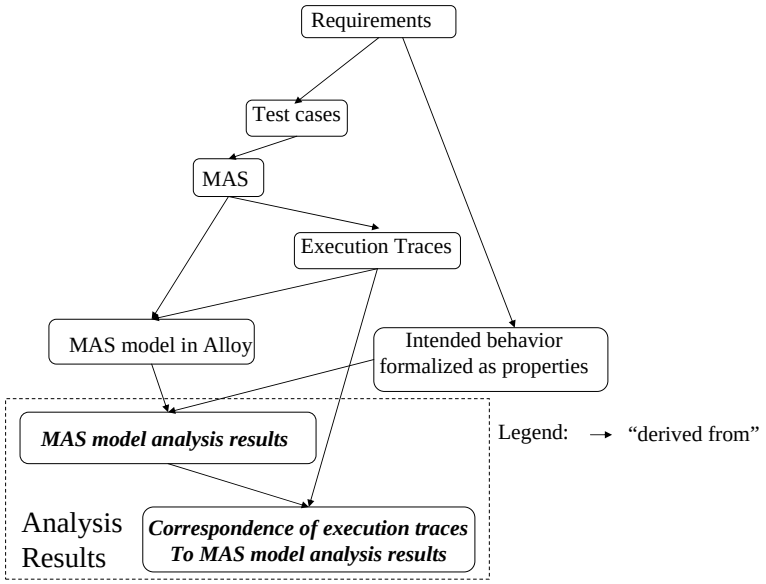

Fig. 1. Experiment design

of traces. This model is then verified against the formally specified properties and the properties that check correspondence of the traces to the results of the verification. Thus we check if the model satisfies the properties and if a sample of actual behavior highlighted by the test suite does not contradict the ideal behavior of the model.

3.3 Choice of the analyzed system

Next we will provide greater detail about the design of the chosen MAS. This detail will let us illustrate the task allocation problem introduced generally above and to formalize a property. The chosen system has been developed in the MAS laboratory headed by Prof. Victor Lesser at the University of Massachusetts, Amherst. This MAS is a mature utility-based reasoning multi agent system that has been extensively used and validated. It has been used as a testbed for a great number of experiments and technology transfer demonstrations in the area of MAS ([23], [24], [11], [12], [18], [9], [10]). This MAS is not restricted to a particular problem domain. It applies the utility-based reasoning to abstract tasks with generalized relationships. Thus we expect that the results obtained from its analysis can be useful for other utility-based systems. In this system an agent is combined of several components that include a problem solver and a negotiation component, among others. The problem solver provides a schedule based on a current set of task structures assigned for execution. The negotiation component drives the execution of negotiation protocols, it is aware of protocol specifications and keeps track of current states of negotiation instances undertaken by its agent. The task structures are specified in the TÆMS language [6]. The schedules are provided by the Design-To-Criteria (DTC) scheduler ([25]) developed by Dr. Tom Wagner which is invoked as

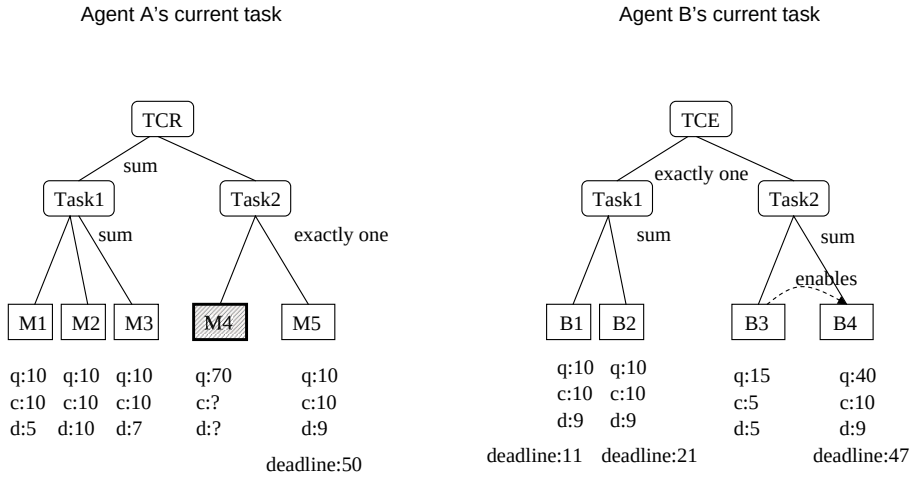

Fig. 2. Pre-negotiation task structures.

part of the agent's problem solver component operation. The DTC takes as input a task structure in TÆMS and a utility function specification and provides as output a set of schedules ranked by their utilities.

In this system a simplified description of an agent's cycle is as follows:

1. *Local scheduling*: in response to an event requesting a certain task to be performed, obtain a number of high ranked schedules by utility;
2. *Negotiation*: conduct negotiation(s) within a predefined limit of time; and,
3. *Execution*: start execution of the schedule chosen as a result of negotiation(s).

The actual cycle of agent's operation is more complex as an agent can react to various kinds of events that it can receive at any of the mentioned cycle stages.

3.4 Relation between protocol FSMs, task structures, offers and visitations

Next we describe the task allocation problem in terms of this design. More details about the cooperative negotiation example can be found in [28]. The negotiation protocol of an agent starting the negotiation (agent *A*), the contractor, is given in Fig. 4. The negotiation protocol of an agent responding to the request (agent *B*), the contractee, is given in Fig. 5.

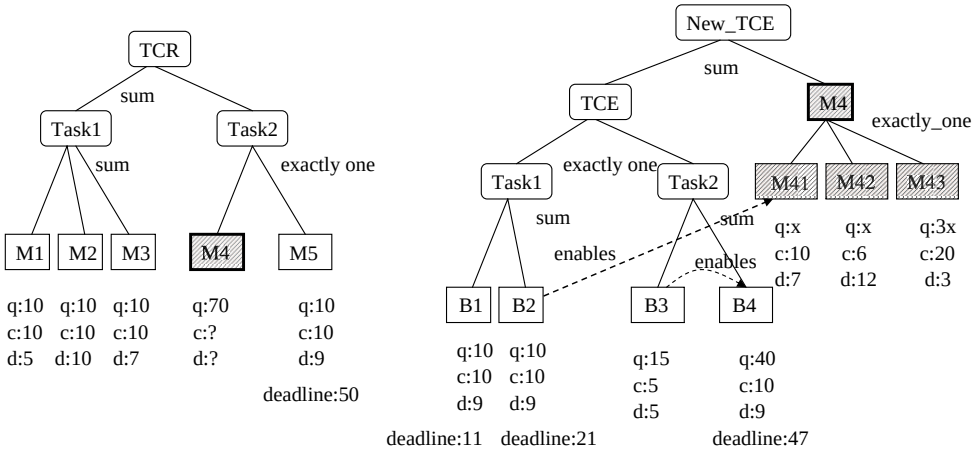

Fig. 3. Post-negotiation task structures.

Let us assume that agent *A* needs a certain non-local task (this means that an agent is not capable of doing that task even though it appears in one of its task structures) to be performed by some other agent. The negotiation's goal is to increase the combined utility of actions of both agents by choosing a particular way to perform the non-local task at a particular time.

In the description that follows we mention the concepts of a protocol FSM, task structures, offers and execution paths encoded in visitations. These concepts are related to one another in the following way.

The design of the particular MAS we are analyzing contains a module called an agent [23]. This module itself is an aggregate of several submodules. One of these submodules is the "Negotiation" submodule and it is responsible for encapsulating knowledge about various protocols known to an agent. These protocols are encoded as FSMs with states corresponding to abstractions of the states of an agent in negotiation and transitions attributed with trigger conditions and actions. A sequence of visitations corresponds to a path from a start node of such a protocol FSM to one of the final nodes.

A task structure of an agent captures its knowledge about multiple ways in which a certain task can be accomplished. The root of a task structure corresponds to a task that an agent is capable of accomplishing. The leaves of a task structure correspond to atomic actions in which both the set and partial order can vary to reflect the way to

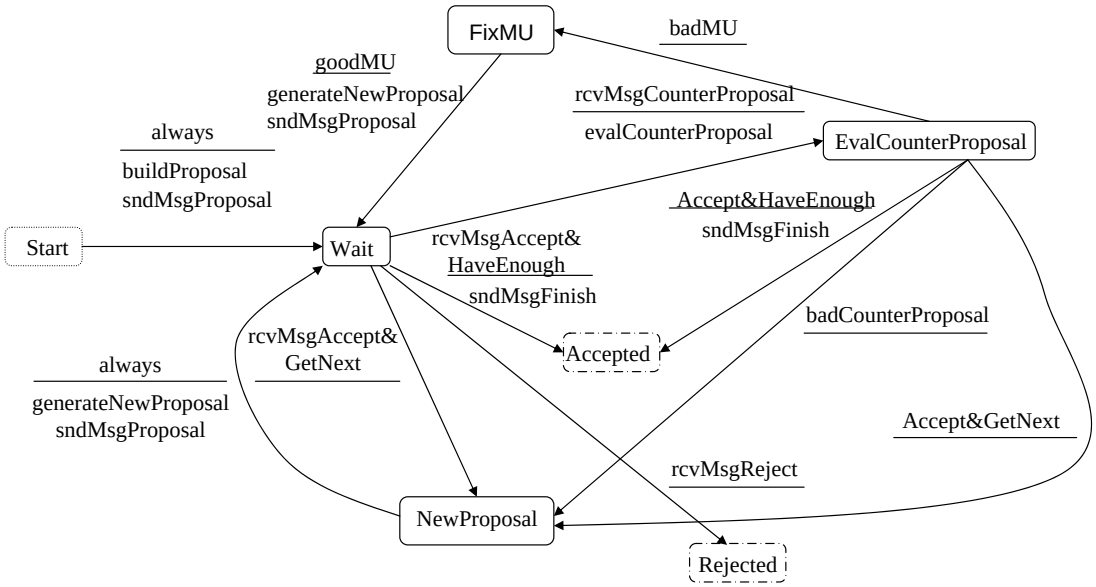

Fig. 4. Contractor's FSM.

accomplish an assigned task in a "utility-increasing" (but not guaranteed to be optimal) way. As an agent progresses through a negotiation protocol according to an FSM, the agent's task structure changes to reflect the agent's changing knowledge about other agent's state throughout that negotiation. Thus there are certain properties imposed on a task structure that must hold while an agent is in certain states of a negotiation protocol FSM.

A collection of task structures determines an agent's functionality analogously to a set of function signatures that define an interface of a module. The roots of task structures serve a similar purpose to function signatures at the agent level of abstraction of describing a software system. An outside event corresponding to a request to accomplish a certain task triggers an agent's reasoning about whether it can accomplish that task considering an agent's knowledge about the way to accomplish that task, that agent's state, the environment state and partial states of some other agents in the same MAS. The result of that reasoning is the current schedule that "interweaves" instances of atomic actions from various tasks currently assigned to that agent in a time-oriented partial order. That current schedule can be changed dynamically, as it is being executed,

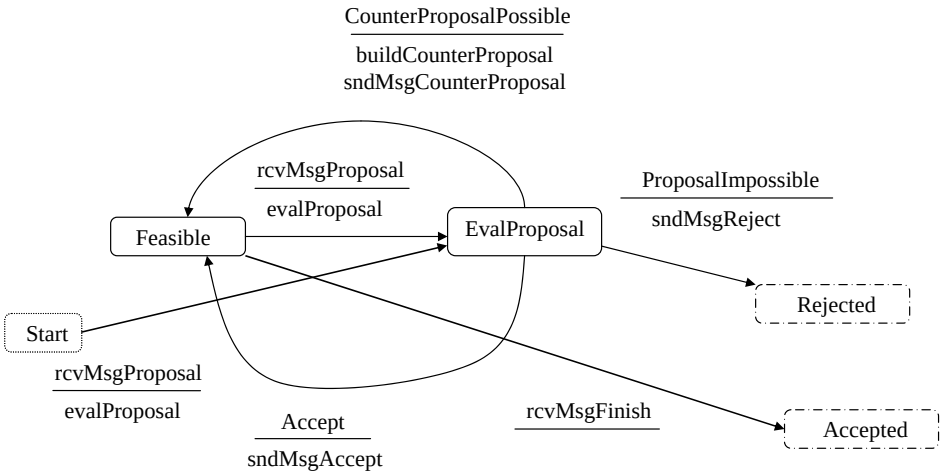

Fig. 5. Contractee's FSM.

in response to agents' changing opinion about the most reasonable schedule for a certain moment in time.

We do not consider execution of schedules, but focus only on the negotiation phase in which schedules always cover future time intervals.

An offer is a data structure generated by actions associated with FSM transitions. An offer encapsulates the parameters of a particular schedule formed on the basis of the agents' task structures, such as quality achieved, start time and finish time. The agents negotiate over these parameters.

Another submodule of an agent is "Communication". The "Negotiation" submodule relies on "Communication" in a fashion similar to how a networking application relies on TCP/IP protocols. The design intentionally separated the concern of ensuring reliable communication and naming mechanisms from the concern of ensuring that a certain "utility-increasing" protocol is followed during a negotiation between a pair of agents. Thus the issues of identifying agents to communicate with for a particular purpose were separated from the "negotiation" submodule by the authors of the MAS system we analyze. This was done to simplify their own analysis, to separate concerns. Our Alloy specification reflects that separation.

In a way, the task structure specifies all possible behaviors of an agent responsible for achieving the goal embodied by a task structure's root. During the stages of *Local scheduling* and *Negotiation* the task structure can be modified, thus modifying specification of a set of behaviors of an agent during an *Execution* stage. The behavior of an agent during the stages of *Local scheduling* and *Negotiation* is static, i.e. it is not modified during run-time. A schedule agreed upon as a result of *Negotiation* is a selected behavior (execution path) from a set of behaviors that was modified at run-time (represented by a task structure; to be performed in the *Execution* stage). Thus a property we describe below checks certain well-formedness of a behavior specification modified at run-time and correctness of an implementation responsible for the modification.

3.5 Details of the task allocation problem in the chosen design

Let us go over a possible scenario of agents' interactions in regard to a task allocation problem for the sake of illustration. This kind of interaction between agents is quite common in any utility-based reasoning MAS. In Fig. 2 we see two task structures. Suppose one task structure, with the root TCR, was assigned to agent *A*, the other, TCE, was assigned to agent *B*. Before the negotiation the striped methods (M4 and its children) are not part of the TCE structure. This assignment can be due to requests sent from the environment (e.g. a human or other automated system). TCE and TCR turned out to be non-leaf nodes with elaborations. Upon receiving task assignments agent *A* sent TCR structure to its local scheduler, agent *B* did the same for TCE.

Let us suppose agent *A* receives the following schedules from its scheduler:

- M1, M2, M3, M4 - highest utility
- M1, M2, M3, M5 - lower utility, feasible

Agent *B* receives the following schedule: B3, B4 that has the highest utility.

Next, agent *A* identifies M4 in its best schedule as non-local. It sends a request to agent *B* to do it. The fact that agent *A* knows that *B* can do M4 is hardwired for the example without loss of generality for the negotiation analysis results. The request initiates an instance of negotiation. Agent *A* plays the role of a *Contractor*, agent *B* - that of a *Contractee*. Agent *B* must see whether it can do M4 by the deadline agent *A* needs it, while accomplishing its current task TCE within the constraints. This is done by modifying the "currently reasoned about" structure and submitting it to the scheduler that will report if such a schedule is possible and, if yes, then with what utility.

The TCE structure must be modified preserving its well-formedness constraints (e.g. functional decomposition remains a tree); and forcing an M4 into a schedule by choosing appropriate quality of M4 that reflects the combined utility of both schedules (chosen by *A* and by *B*). Fig. 3 shows agent *B*'s task structure updated with an M4. The quality attribute of M4 must be such that the scheduler of agent *B* must produce feasible (though not necessarily high ranking) schedules that contain M4 and still accomplish the original TCE task.

Even if the agent *B*'s local scheduler returns an acceptable schedule (which has M4 in it and the original TCE is accomplished with the constraints on time and quality), agent *A* can request to make a tighter fit. Therefore if its scheduler found a feasible

schedule that includes M4, agent *B* (*Contractee*) is supposed to transition to state "Feasible" (Fig. 5) and wait for agent *A* to send another proposal with a "tighter" deadline on M4's execution or a "finish" message. This means that agent *B* must have modified its task structure to include M4. On the contrary, if there is not a single feasible schedule that can include M4 then agent *B* is supposed to transition to state "Rejected". If agent *B* reaches state "Rejected" then its post task structure TCE' is unchanged from the pre TCE.

With this description in mind we can rephrase this property in terms of the TÆMS structures and negotiation protocol specifications in Figures 4 and 5 as:

After agent *B* reaches state "Feasible" at least once its task structure must contain a subtree corresponding to task M4 and M4 must appear in a feasible schedule returned to agent *A*.

In this example the Contractee's FSM has been simplified for the sake of this illustration. For additional details about this example please refer to [28].

4 Alloy specification for the negotiation model

Our approach implies modeling particular paths traversed in the agents' negotiation finite state machines (FSMs) in response to certain testcases. Thus we check an abstraction of an execution path in a particular implementation. Both FSMs contain cycles. If a cycle diameter can be modeled with the scope that can be processed by the Alloy analyzer then we can iteratively check a certain property on an execution path that corresponds to multiple iterations of a cycle.

The negotiation protocols and task structures described in section 3 had to be simplified to have a tractable scope for the Alloy analyzer. The simplifications include:

1. ignoring attributes of task structures nodes (quality, duration, cost);
2. ignoring attributes of offers (mutual utility gain, cost, earliest start time);
3. ignoring attributes of schedules (start time and finish time of actions); and,
4. simplifying task structures by removing intermediate nodes (e.g. no Task_1, Task_2) and reducing the number of leaf nodes (e.g. only B1 and B3 left in agent *B*'s task structure).

The actual models used for analysis also contain only those atoms that are necessary for verifying a property at hand. Thus transitions that were not traversed by a modeled execution path and associated states were removed.

This amount of simplification was necessary to make the analysis feasible. Earlier we constructed a more detailed Alloy specification of the analyzed system. The Alloy analyzer was not able to cope with such a specification. We had to reduce its size gradually while still keeping the analysis useful. We expect that the next generation of the Alloy analyzer, Kodkod [20], would be able to deal with a larger specification.

The resultant Alloy model of the MAS for the purpose of verifying our assertions consists of 3 modules. One module, `negProtocol12_1abridgeDataProp`, models the FSMs, visitations of transitions through the FSMs (paths specified by transitions) and assertions. Two more modules model the data structures manipulated by the agents

- their task structures and schedules. Let us briefly go over the Alloy models in these modules.

The `negProtocol12_1abridgeDataProp` defines signatures for `State`, `Transition`, `Visitation` and `Offer`. Thus an FSM is modeled by constraining atoms of `State` and `Transition` signatures via the "fact" construct. A `Transition` signature contains fields for source and destination states, a set of visitations of that transition by a path and a set of transitions outgoing from the destination state of the transition. The `treeDefSmall` module models a task structure of an agent. The `schedDefSmall` module models a schedule data structure of an agent. It imports the `treeDefSmall` so that schedule items can reference the nodes of task structures. The consistency of the model has been successfully checked with an empty stub predicate. The analyzer found a solution.

```
abstract sig State {}

abstract sig Transition {
  source, dest: State,
  visit: set Visitation,
  nextTrans: set Transition
}

fact Injection { all t, t': Transition | t.source =
                  t'.source && t.dest =
                  t'.dest => t = t' }

abstract sig Visitation {
  trans: lone Transition,
  nextVisit: lone Visitation,
  offer: lone Offer
}
fact VisTransConsistent {
  all visitation: Visitation | visitation in
    visitation.trans.visit
}
```

5 Alloy specification for the properties

The paths of execution of the two negotiation protocols are represented by atoms of the `Visitation` signature. Thus it is via these atoms that we express a property that can be informally phrased as "If agent *A* is led to believe by a certain sequence of communications that agent *B* reaches a certain state then agent *B* should have indeed reached that state, having been subjected to the same changes of observed environment as agent *A*". This informal statement pinpoints such a feature of agents in a MAS as bounded rationality. The property checks for consistency between a certain abstraction of other agent's state (agent *B*) that a certain agent (*A*) obtains via communication. In the case of the particular system we used the communication is explicit. By modeling the environment sensed by agents we could allow for checking such properties based on implicit communication.

More specifically, in view of the simplifications we made, a property of this kind can be informally restated as "if agent *A* reaches state `EvalCounterProposal` then agent

```
module models/examples/tutorial/treeDefSmall

abstract sig Tree {
  root: Node,
  nodes: set Node,
  edges: Node -> Node
}

{
  nodes = root.*edges
  all m: Node | m !in m.^edges
}

abstract sig Node {}

one sig TCR,  M3, M4, M5, TCE, B1, B3, New_TCE extends Node{}

one sig AgentB_preTaskStrucTCE extends Tree {}
fact AgentB_preTaskStrucTCEDef {
  AgentB_preTaskStrucTCE.root = TCE
  AgentB_preTaskStrucTCE.nodes = TCE + B3
  AgentB_preTaskStrucTCE.edges = TCE->B3
}

one sig AgentB_postTaskStrucTCE extends Tree {}
fact AgentB_postTaskStrucTCEDef {
  AgentB_postTaskStrucTCE.root = New_TCE
  AgentB_postTaskStrucTCE.nodes = New_TCE + TCE + B1 +  M4
  AgentB_postTaskStrucTCE.edges = New_TCE->TCE +
   New_TCE->M4 + TCE->B1
}
```

B should have reached state `Wait2` and beginning since that state, agent *B*'s current schedule data structure should have contained an instance of atomic action M4". Below we can see how this property is formally expressed in the Alloy's relational algebra. The assertion has been successfully checked. No counterexamples were found for the path containing visitations that corresponded to the expected states and data structure conditions. Conversely, once an inconsistency between agent *A*'s belief and agent *B*'s state and data structures has been introduced into visitations, the analyzer pinpointed a possible counterexample.

We have also translated an Alloy specification of this property into a dynamic assertion in Java using a systematic translation approach [2]. Thus we were able to dynamically check the conformance of an implementation to the Alloy specification. We also showed the utility of the Alloy Analyzer by making sure that an assertion in the Alloy specification is right and then mechanically translating that assertion into a dynamic assertion in Java implementation.

```
module models/examples/tutorial/schedDefSmall
open models/examples/tutorial/treeDefSmall

abstract sig SchedItem {
  activity:  Node
}

one sig SchedItemM3 extends SchedItem{}
fact SchedItemM3Def {
  SchedItemM3.activity = M3
}

one sig SchedItemM4 extends SchedItem{}
fact SchedItemM4Def {
  SchedItemM4.activity = M4
}

one sig SchedItemM5 extends SchedItem{}
fact SchedItemM5Def {
  SchedItemM5.activity = M5
}

one sig SchedItemB1 extends SchedItem{}
fact SchedItemB1Def {
  SchedItemB1.activity = B1
}

one sig SchedItemB3 extends SchedItem{}
fact SchedItemB3Def {
  SchedItemB3.activity = B3
}

abstract sig Sched {
  items: set SchedItem,
  precedenceRel: SchedItem -> SchedItem
}

one sig AgentAschedWithNL extends Sched {}
fact AgentAschedWithNLDef {
  AgentAschedWithNL.items = SchedItemM3 + SchedItemM4
  AgentAschedWithNL.precedenceRel =
     SchedItemM3->SchedItemM4
}

one sig AgentAschedWithOutNL extends Sched {}
fact AgentAschedWithOutNLDef {
  AgentAschedWithOutNL.items = SchedItemM3 + SchedItemM5
  AgentAschedWithOutNL.precedenceRel =
     SchedItemM3->SchedItemM5
}

one sig AgentBschedWithNL extends Sched {}
fact AgentBschedWithOutNLDef {
  AgentBschedWithOutNL.items = SchedItemB1 + SchedItemM4
  AgentBschedWithOutNL.precedenceRel =
     SchedItemB1->SchedItemM4
}

one sig AgentBschedWithOutNL extends Sched {}
fact AgentBschedWithNLDef {
  AgentBschedWithNL.items = SchedItemB3
}
```

```
assert AgentAbeliefCompliesWithAgentBState {
  (some visitation: Visitation |
    visitation.trans.dest = EvalCounterProposal) =>
     (some visitation': Visitation |
       visitation'.trans.dest = Wait2 &&
         M4 in visitation'.offer.agentBTaskTree.nodes)
}
```

6 Specification difficulties

The main difficulty is keeping the Alloy model under a tractable scope while checking useful properties. In the case of the design of this particular MAS the protocols are specified via FSMs with loops. Thus we can check properties only within the scope of the FSM's diameter. Other difficulties are due to highly dynamic, hard to predict behavior of sensing agents. One has to classify the dynamics of the environment sensed by the agents and check the properties within each such situation. For instance, in the example used in this paper we can classify the situations based on combinations of "best" schedules of the 2 agents with regard to including the non local task (M4) into their schedules. Some of the possible combinations (for all cases agent *A* has M4 in its best schedule):

- agent *B* does not have M4 in its best schedule; the local utility of agent *B*'s schedule outweighs the combined utility if agent *B* is forced to do M4;
- agent *B* does not have M4 in its best schedule; the local utility of agent *B*'s schedule is below the combined utility if agent *B* is forced to do M4;
- agent *B* has M4 in its best schedule too, but not within the timeframe agent *A* needs M4 to be finished
- agent *B* has M4 in its best schedule too, it is within the timeframe agent *A* needs M4 to be finished

It should be possible to provide an Alloy model so that these combinations would not have to be specified explicitly. Instead, the Alloy analyzer itself would check over all the alternatives it sees in the model. A straightforward approach of modeling the attributes of the nodes in the agents' task structures results in too large a scope for the Alloy to handle. Perhaps the attribute values should be abstracted as features of the task structures, not as numerical values.

7 Conclusions and Future Work

We have created and validated a model for verifying data structure rich properties of a cooperative multi-agent system using a manually created execution path. To our knowledge, our work is the first application of the Alloy analyzer for checking properties of a multi-agent system. Moreover, this example illustrates how the use of Alloy's formal reasoning capability can be integrated into the testing and validation activities of software development.

Another step might be checking a property on all interior paths of a loop in an FSM. One more interesting property would involve checking if an elaboration of the non-local task is "interwoven" in one of the many alternative ways into the task structure of

an agent. We expect that checking such a more complicated and a more realistic case might highlight Alloy's advantage due to the declarative nature of its relational algebra. It would also be interesting to see whether CSP-based models and tools (FDR) or B CSP models would be useful for checking properties of negotiation in MAS systems with explicit communication.

Acknowledgments

We would like to express our deep gratitude to Prof. Victor Lesser (UMass, Amherst) for his help with the negotiation protocol example implemented in the multi-agent system simulator and helpful comments.

References

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, Dec. 1999.
2. B. Al-Naffouri. An algorithm for automatic generation of run-time checks from alloy specification. Advanced Undergraduate Project Report, Massachusetts Institute of Technology, 2002.
3. N. Alechina and B. Logan. Verifying bounds on deliberation time in multi-agent systems. In *EUMAS*, pages 25–34, 2005.
4. B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006, Shanghai, May 20-28)*, pages 72–81. ACM Press, 2006.
5. D. Box. *Essential COM*. Addison Wesley, 1998.
6. K. Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In *Foundations of Distributed Artificial Intelligence, Chapter 16*, pages 429–448. G. O'Hare and N. Jennings (eds.), Wiley Inter-Science, January 1996.
7. G. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Master's thesis, Massachusetts Institute of Technology, 2003.
8. G. Gans, M. Jarke, G. Lakemeyer, and T. Vits. SNet: A modeling and simulation environment for agent networks based on i* and ConGolog. In *CAiSE02*, pages 328–343, Canada, Toronto, May 2002. Springer.
9. B. Horling and V. Lesser. Using Diagnosis to Learn Contextual Coordination Rules. *Proceedings of the AAAI-99 Workshop on Reasoning in Context for AI Applications*, pages 70–74, July 1999.
10. B. Horling, V. Lesser, R. Vincent, A. Bazzan, and P. Xuan. Diagnosis as an Integral Part of Multi-Agent Adaptability. *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 211–219, January 2000.
11. B. Horling, R. Mailler, and V. Lesser. Farm: A Scalable Environment for Multi-Agent Development and Evaluation. *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003)*, pages 171–177, May 2003.
12. B. Horling, R. Vincent, R. Mailler, J. Shen, R. Becker, K. Rawlins, and V. Lesser. Distributed Sensor Network for Real Time Tracking. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 417–424, June 2001.

13. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.
14. D. Jackson and M. Jackson. *Separating Concerns in Requirements Analysis: An Example*, chapter Rigorous development of complex fault tolerant systems. Springer-Verlag. (To appear).
15. D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
16. M. Kacprzak, A. Lomuscio, and W. Penczek. Verification of multiagent systems via unbounded model checking. In *AAMAS*, pages 638–645, 2004.
17. V. Lesser. Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. *Autonomous Agents and Multi-Agent Systems*, 1:89–111, January 1998.
18. V. Lesser, M. Atighetchi, B. Benyo, B. Horling, A. Raja, R. Vincent, T. Wagner, X. Ping, and S. X. Zhang. The Intelligent Home Testbed. *Proceedings of the Autonomy Control Software Workshop (Autonomous Agent Workshop)*, January 1999.
19. V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. N. Prasad, A. Raja, R. Vincent, P. Xuan, and X. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.
20. E. Torlak. http://web.mit.edu/ẽmina/www/kodkod.html.
21. W. van der Hoek and M. Wooldridge. Model checking knowledge and time. *Proc. of the 9th Int. SPIN Workshop*, 2318 of LNCS:95–111, 2004.
22. M. Vaziri. *Finding Bugs Using a Constraint Solver*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.
23. R. Vincent, B. Horling, and V. Lesser. An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator. In *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, volume 1887, pages 102–127. Wagner and Rana (eds.), Springer,, January 2001.
24. T. Wagner and B. Horling. The Struggle for Reuse and Domain Independence: Research with TAEMS, DTC and JAF. *Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS (Agents 2001)*, June 2001.
25. T. Wagner and V. Lesser. Design-to-Criteria Scheduling: Real-Time Agent Control. *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, pages 89–96, March 2000.
26. G. Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, Mass., 1999.
27. M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with mable. In *AAMAS*, pages 952–959, 2002.
28. X. Zhang, R. M. Podorozhny, and V. Lesser. Cooperative, MultiStep Negotiation Over a Multi-Dimensional Utility Function. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing (ASC 2000)*, pages 136–142, 2000.