Model Checking Rational Agents

Rafael H. Bordini, *University of Durham*

Michael Fisher and Michael Wooldridge, *University of Liverpool*

Willem Visser, *Research Institute for Advanced Computer Science*

The authors use model-checking techniques to automatically verify multiagent systems. Furthermore, property-based slicing can reduce a multiagent system's state space, increasing model-checking efficiency. The analysis of a typical scenario of an autonomous Mars rover illustrates this approach.

A cornerstone of autonomous-agents research is that we can model—and ultimately program—such agents through the concept and principles of *rational action*. In other words, agents are systems that attempt to accomplish their goals by acting rationally in a shared environment. This view of agents manifests itself in many ways, but perhaps the most significant manifestation is the development of rational agents that use practical reasoning to make decisions about what to do.[1]

Agent-oriented programming techniques seem appropriate for developing systems that operate in complex, dynamic, and unpredictable environments. Such application areas include airtraffic control, autonomous-spacecraft control, health care, and industrial-systems control, all of which require dependable and, more importantly, verifiable computational systems. We aim to address this requirement by developing model-checking techniques for the (automatic or semiautomatic) verification of rational-agent systems written in a logic-based agent-oriented programming language.

Typically, developers apply model-checking techniques to abstract models of a system rather than the system implementation (see the "Model Checking" sidebar). Although this is important for detecting design errors at an early stage, developers might still introduce errors during coding.[2] In contrast, developers can directly apply our model-checking techniques to systems implemented in an agent-oriented programming language, automatically verifying agent systems without the usual gap between design and implementation.

Our approach

We developed our techniques for *AgentSpeak*, a rational-agent programming language based on the AgentSpeak(L) abstract agent-oriented programming language.[3] AgentSpeak shares many features of the agent-oriented programming paradigm (see the "Agent-Oriented Programming" sidebar for more information). More importantly, it supports the principle that we can program agents in terms of "mentalistic" notions such as beliefs, desires, and intentions, and these mental attitudes feature prominently in the language (which is why AgentSpeak is often called a BDI language).

Besides using some mentalistic notions in programming the system, we also want to be able to refer to such notions in the specifications that we check against our agent programs. Therefore, we use a simple form of BDI logic as our property specification language.[1] We can automatically translate this language into linear temporal logic (LTL, the property specification language of various widely used, freely available model checkers) with special expressions for interpreting the BDI modalities in terms of the state of AgentSpeak agents.

Similarly, we've developed techniques for automatically translating AgentSpeak programs into the model specification language of existing model-checking systems. In this way, we reduce the problem of verifying that an AgentSpeak system has certain BDI logic properties to a conventional LTL model-checking problem.

Avoiding the need to develop a model checker from scratch has its advantages. Some existing model checkers have more than a decade of continuous development behind them. They can therefore offer reliable implementations as well as sophisticated techniques to cope with the substantial computational requirements of model-checking experiments. The idea, in this case, is to translate specific notations into input and specification languages that those model checkers understand.

Following that approach, and on the basis of work

Model Checking

Imagine we have a logical formula, φ, that specifies some program's property that we wish to check. How can we check this property against a program description?

One (deductive) approach is to have another logical formula, say Γ, that exactly specifies the program (derived, for example, through logical semantics). Thus, Γ must characterize the program's (possibly infinite number of) models (or executions). To check that the property holds, we must prove that we can infer φ from Γ. Effectively, this means establishing that the set of models that satisfy Γ is a subset of the set of models that satisfy φ. If the sets of models are infinite, this can be difficult.

If, on the other hand, we have a small (finite) set of models (say, Σ) that represent the only possible program executions, we only need to check that all these models satisfy φ—that is, that each individual model in Σ satisfies φ. Because only a finite number of such models exist, this can be checked automatically.

This (algorithmic) approach is called *model checking*.[1] It has been particularly successful where the programs being checked are reactive systems and where the properties are given using temporal logics. We can characterize model checking in terms of the relationship between sets of models. In the case of reactive systems being checked with respect to temporal formulas, these models are infinite sequences. So, we can also characterize the model-checking relationship in terms of automata that accept such infinite sequences, particularly Büchi automata.

Although model checking has been successful in several areas, it has two key problems. First, for any nontrivial program and property, the space required to check the property can be very large. Second, basic model checking is restricted to finite-state programs (in particular, programs with finite numbers of possible executions). Researchers have been focusing on these two problems and have recently made significant advances. Using symbolic approaches to represent the model-checking problem in a more compact way has helped alleviate the space problem. Also, techniques that let the model-checking procedure be applied to finite abstractions of infinite systems (see the "Abstraction and Slicing" sidebar) allow the verification of programs with infinite state spaces.

(Also see the related article in this issue, "Using Model Checking to Assess the Dependability of Agent-Based Systems.")

Reference

1. E.M. Clarke Jr., O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999.

initially reported elsewhere,[4] we developed a set of tools called CASP (*C*hecking *A*gent-*S*peak *P*rograms). Using CASP, we can translate an AgentSpeak agent or multiagent system into both PROMELA (the input language of Spin (http://spinroot.com), a publicly available model-checking system) and Java. We can then use Spin or Java PathFinder (http://ase.arc.nasa.gov/visser/jpf) to verify our systems. JPF is an on-the-fly model checker that works directly on Java bytecodes.[2]

Our approach to dependable multiagent systems applies particularly to agents for which precise (that is, formal) notions of belief, desire, and intention exist. Although they can coexist in a system, agents programmed using traditional techniques should be verified with equally traditional verification techniques; in particular, their properties must be specified in plain, linear temporal logic rather than the property specification language we consider here.

Programming multiagent systems

A first key step in our research was to restrict AgentSpeak to finite-state systems, resulting in the AgentSpeak(F) language.[4]

Agent-Oriented Programming

The notion of agents as rational actors is a leitmotif in autonomous-agents research. In the early 1990s, Yoav Shoham articulated a vision of programming agents that makes rational action a first-class component of the programming model.[1] Shoham envisioned systems composed of multiple interacting agents (a "societal model of computation"), where the agents were directly programmed in terms of mental states such as beliefs, desires, and intentions.

The concept shares some of the attributes of declarative programming: instead of directly programming an agent in terms of low-level instructions relating to its precise course of action, we give it goals (or desires) to achieve and information about its environment in the form of beliefs: the goals in turn determine the agent's intentions. The agent then makes a rational decision about what action to perform: it behaves in the way any rational individual would, given its beliefs and goals.

For example, a rational individual with a given intention would be inclined to try again if the first attempt to achieve that intention failed. Similarly, an agent with a particular intention wouldn't choose to adopt other intentions that were inconsistent with this one. Programming an agent then amounts to giving it appropriate beliefs about its environment and giving it some goal to achieve.

There are several compelling arguments for this model. Ideally, it abstracts away from control issues: we simply present some goal that we wish to be achieved, and we expect the computational system to act as a rational agent would given such a goal. Also, because we're used to understanding and predicting the behavior of rational agents, the behavior of agent programs should be relatively easy for humans to understand and predict. To exemplify his proposal, Shoham presented a prototype agent-oriented programming language called AGENT0; researchers have subsequently proposed many other similar languages.

Reference

1. Y. Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, Mar. 1993, pp. 51–92.

Reactive Planning Systems

A key component of rational action in humans is planning: we can generate from a goal a recipe for action such that following this recipe will achieve the goal. Accordingly, much AI research has addressed the issue of *automatic planning*: the synthesis of plans by agents from first principles. Unfortunately, planning is, like so many other AI problems, prohibitively expensive computationally. Although researchers have made great strides in developing efficient planners, the process's inherent complexity casts some doubt on whether you can use plan-synthesis algorithms to develop plans at runtime.

Many researchers have instead considered approaches to developing agents that use *precompiled* plans—that is, plans developed offline, at design time. Michael Georgeff and Amy Lansky's *procedural-reasoning system* is a common ancestor of such approaches.[1] On one level, you can view a PRS simply as an architecture for executing precompiled plans. However, the control structures in the architecture incorporate additional features that provide a sophisticated environment for runtime practical reasoning.

First, the architecture's control structure might invoke plans by their effect rather than by name (as is the case in conventional programming languages). Second, plans are associated with a context, which must match the agent's current situation for the plan to be considered viable. These two features mean that an agent may have multiple potential plans for the same end and can dynamically select between them at runtime, depending on its current circumstances. In addition, plans are associated with *triggering events*, the idea being that a plan is made "active" by the occurrence of such an event, which can be external to the agent or internal. An example of an internal event might be the creation of a new subgoal or a plan's failure to achieve its desired effect. External events, on the other hand, are generated by perceived changes in the agent's environment. Overall, plans can be invoked in a goal-driven manner (to satisfy a subgoal) or in an (external) event-driven manner. AgentSpeak represents an attempt to distill PRS's essential features into a simple, unified programming language.

Reference

1. M.P. Georgeff and A.L. Lansky, "Reactive Reasoning and Planning," *Proc. 6th Nat'l Conf. Artificial Intelligence* (AAAI 87), AAAI Press/MIT Press, 1987, pp. 677–682.

By translating multiagent systems defined in this language into the existing model checkers' input language, we could exploit the extensive range of tools and techniques available with such model checkers.

An AgentSpeak(L) agent is defined by a set of beliefs (ground predicates) and a set of plans. In this language, we distinguish two types of goals: achievement and test. Achievement goals are predicates prefixed with the **!** operator; test goals are prefixed with the **?** operator. An achievement goal states that the agent wants to achieve a state of the world where the associated predicate is true. A test goal states that the agent wants to test whether the associated predicate matches one of its beliefs.

An AgentSpeak agent is a *reactive planning system* (see the related sidebar). The agent reacts to events that change either its beliefs (because its perception of the environment has changed) or its goals (because an executing plan requires the execution of another plan to achieve a subgoal). A *triggering event* defines which events may lead to the execution of a particular plan. The programmer writes the plans so that the addition (+) or deletion (–) of beliefs or goals (the mental attitudes of AgentSpeak agents) can trigger them.

An AgentSpeak(L) plan has a triggering event that denotes the plan's purpose, a context (given by a conjunction of belief literals), and a body. The conjunction of literals in the context must be a logical consequence of that agent's current beliefs if the plan is to execute. A *plan body* is a sequence of basic actions or (sub)goals that the agent must achieve (or test) when the plan is triggered. Plan bodies refer to the basic actions that an agent can perform on its environment. Such actions are also written as predicates, using special predicate symbols called *action symbols*. We later present plans that illustrate these notions. As in Prolog, an uppercase initial letter is used for variables and lowercase for constants and predicate symbols.

The main difference in syntax between AgentSpeak(F) and AgentSpeak(L) is that the former doesn't allow first-order terms—that is, terms are either constants or variables. Other restrictions—such as using uninstantiated variables in triggering events or specifying bounds for the data structures that our AgentSpeak interpreter uses—apply only when the target model checker is Spin. JPF can cope with data structures that don't have a predefined bound. Of course, if the model is such that the data structures grow too much, this will prevent full state-space search (it's a well-known cause of state-space explosion).

Researchers have proposed various extensions to AgentSpeak(L) to make it a more practical programming language—for example, the construct of *internal actions* (somewhat similar to native methods in Java). These are arbitrary programs that the agents run internally ("internal" here distinguishes these actions from those that change the environment, thus having an effect that's external to the agent). Action symbols with a "." character denote internal actions: users can then define libraries of such actions. The interpreter includes a library of predefined (standard) internal actions to account for important extensions of the language for general practical use (from arithmetic expressions to more sophisticated mechanisms such as dropping intentions).

The standard internal action **.send** is used for speech-act-based interagent communication and is interpreted as follows. If an AgentSpeak(F) agent l_1 executes **.send** (l_2, *ilf*, *at*), a message will be inserted into the mailbox of agent l_2, having l_1 as sender, illocutionary force *ilf*, and propositional content *at* (an atomic AgentSpeak(F) formula). To keep things simple, AgentSpeak(F) has only three predefined illocutionary forces: **tell**, **untell**, and **achieve**. They have the same informal semantics as in the usual agent communication languages. These communicative acts change the agent's mental state only after user-defined functions confirm the source of information's trustworthiness or the rights of other agents to request tasks.

Although we don't provide a detailed description of the interpretation of AgentSpeak(L) programs, we give some related concepts. *Intentions* are particular courses of

actions to which an agent has committed so as to achieve a particular goal: each intention is a stack of *partially instantiated plans*—that is, plans in which some variables have been instantiated. An *event*, which can trigger a plan's execution, can be external when originating from the agent's perception of its environment. Or, the event can be internal when generated from one of the agent's executing plans (for example, an achievement goal in a plan body generates a goal-addition event).

In our approach, we program a multiagent system by writing a collection of AgentSpeak source codes (one for each agent in the system) and the definition of the shared environment. To define the shared environment, we need to represent all the facts (in the form of predicates) about the environment's state: we use the target model checker's input language rather than AgentSpeak to do this. Each agent has its own percepts based on sensing the environment, thus letting us model the fact that agents might have incorrect or incomplete information about the world. Agents also have a belief revision function that generates the appropriate external events (the perceived changes in the environment). The available agent architecture adopts a simple belief revision function, unless the user provides a more specific one.

Specifying required properties

We've presented a way of interpreting the informational, motivational, and deliberative modalities of BDI logics in terms of an AgentSpeak agent's state; this is based on the operational semantics of AgentSpeak. We use this framework to interpret the BDI modalities in terms of data structures in the model of an AgentSpeak(F) agent. This way, we can translate (temporal) BDI properties into LTL formulas.

The logical property specification language for our model-checking approach is a simplified version of LORA (Logic of Rational Agents),[1] which is based on modal logics of intentionality, dynamic logic, and CTL* (a well-known branching temporal logic). In the restricted version of the logic used here, we limit the underlying temporal logics to LTL rather than CTL*, given that our target model checkers can automatically process LTL formulas (excluding the "next" operator $\bigcirc$). Let *pe* be any valid Boolean expression in the model specification language of the model checker, *l* be any agent label, *x* be a variable ranging over agent labels, and *at* and *a* be atomic and action formulas defined in the AgentSpeak(F) syntax, but with no variables allowed. Then we define inductively the set of well-formed formulas (*wff*) of our property specification language as we show in Figure 1.

1. *pe* is a *wff*.
2. *at* is a *wff*.
3. (Bel *l* at), (Des *l* at), and (Int *l* at) are *wff*.
4. $\forall x.(M\ x\ at)$ and $\exists x.(M\ x\ at)$ are *wff*, where $M \in$ {Bel, Des, Int} and *x* ranges over a finite set of agent labels.
5. (Does *l a*) is a *wff*.
6. If φ and ψ are *wff*, so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$, always $(\Box\varphi)$, eventually $(\Diamond\varphi)$, until $(\varphi\ \mathcal{U}\ \psi)$, and "release," the dual of until $(\varphi\ \mathcal{R}\ \psi)$.
7. nothing else is a *wff*.

Figure 1. The property specification language used in our model-checking approach.

In the syntax used in Figure 1, agent labels denoted by *l*, and over which variable *x* ranges, are the ones associated with each AgentSpeak(F) program during translation. That is, the labels given as input to the translator form the finite set of agent labels over which the quantifiers are defined. The only unusual operator in this language is (Does *l a*), which holds if the agent denoted by *l* has requested action *a* and that's the next action the environment will execute. An AgentSpeak(F) atomic formula *at* refers to what's actually true of the environment (rather than what's true from the agent's viewpoint).

The concrete syntax used in the system for writing formulas also depends on the underlying model checker. Before we pass the LTL formula on to the model checker, we translate *Bel*, *Des*, and *Int* modalities into predicates accessing the AgentSpeak(F) data structures modeled in the model checker's input language. An intention requires an available applicable plan, whereas a desire doesn't. In BDI theory, intentions are desired states of affairs that an agent has committed itself to achieving (in practice, by executing a plan). The term *goal* often refers to a desire, but BDI theory assumes an agent's goals—but not necessarily its desires—are compatible with each other.

Autonomous Mars rover: An illustrative scenario

Consider a typical day for an autonomous rover such as NASA's Spirit and Opportunity, which landed on Mars in early 2004. The ground team might tell the rover to traverse toward a certain rock, place its spectrometer arm on the rock, carry out extensive measurements, then perform a long traverse to another distant rock. Before sending the rover to Mars, the team instructs it to give priority to rocks with "green patches," even when traveling to another target, because such patches provide an interesting opportunity for NASA scientists.

Also, the rover's batteries only work when there's sunlight, so activities are constrained by the amount of energy stored during the day. The rover must transmit its collected data back to Earth before it runs out of energy, so it must interrupt an activity if finishing it means the rover won't have enough energy to downlink collected data back to Earth.

Previous Mars exploration rovers, such as Sojourner, didn't have flexible control software. Researchers have reported, for example, that during one operation, the rover didn't position itself correctly to approach a certain rock with the spectrometer arm.[5] The misplaced spectrometer meant the rover couldn't collect any useful data. NASA thus lost an opportunity because the rover didn't revisit that particular rock. Reactive planning systems are particularly suitable for providing flexible control for autonomous rovers.

Here we provide some AgentSpeak plans for the autonomous Mars rover scenario. According to the first example, whenever the rover believes it has observed a green patch on a rock, unless its batteries are too low, it'll try to examine the rock:

```
+green_patch(Rock) :
   not battery_charge(low) <-
      ?location(Rock,Coordinates);
      !traverse(Coordinates);
      !examine(Rock).
```

The rover must retrieve, from its own belief base, the coordinates associated with that rock (this is the test goal in the beginning of the plan's body), then achieve the goal of traversing to those coordinates, and finally examining the rock. Recall that each of these achievement goals will trigger the execution of some other plan.

The next plans provide alternative courses of actions for traversing to a certain location that the rover must choose according to what it believes about the environment:

```
+!traverse(Coords) :
   safe_path(Coords) <–
      move_towards(Coords).

+!traverse(Coords) :
   not safe_path(Coords) <–
      ...
```

If the rover believes there's a safe path for traversing toward the given coordinates, it simply moves toward those coordinates (this is a basic action through which the rover can effect changes in its environment). We don't show the alternative plan here, in which the rover searches for an alternative route, avoiding any unsafe paths.

The next example tells the rover how to examine a certain rock:

```
+!examine(Rock) :
   correctly_positioned(Rock) <–
      place_spectrometer(Rock);
      !extensive_measurements(Rock).

+!examine(Rock) :
   not correctly_positioned(Rock) <–
      !correctly_positioned(Rock);
      !examine(Rock).
```

If the rover believes it's correctly positioned to examine the rock, it executes the action of placing the spectrometer arm on that rock (recall that basic actions denote the hardwired means available in the rover for changing its environment), then achieving the goal of doing extensive spectrometric measurements. If it doesn't believe it's correctly positioned, then it should first achieve a state of affairs in which it believes it to be so before attempting again to examine the rock.

Next, we show examples of properties that the Mars rover is expected to satisfy, written in our specification language. The specification in Figure 2a indicates that whenever the rover places its spectrometer arm at a certain rock, it believes it's correctly positioned to examine that rock.

The specification in Figure 2b ensures that after the rover intends to transmit its remaining spectrometer data back to Earth, eventually it'll no longer contain data entries for which it doesn't have an associated belief saying that it has already downlinked that particular piece of information. This ensures, for example, that the rover's batteries don't run out before it finishes transmitting all gathered data.

Alleviating the state-space explosion problem

Clearly, using model-checking programs rather than design models poses significant challenges. Because actual programs are typically more elaborate than designs, the state-space explosion problem is exacerbated. So, the importance of state-space reduction techniques, particularly *abstraction* techniques (see the "Abstraction and Slicing" sidebar), is even greater.

One such abstraction technique is *property-based slicing*. This technique is similar to the program slicing that software engineering traditionally uses,[6] except that the slicing criterion is the property we later want to model check. As part of CASP, we've devised a property-based slicing algorithm for AgentSpeak that lets us remove agent plans that aren't relevant for model checking a certain property: we automatically generate the relevant slice before we translate the system.[7] The system's model generated in this way can then have a significantly smaller state space.

Slicing alleviates the state explosion in two ways. First, it removes plans that can't affect the truth (or falsity) of the formula in the slicing criterion. This is similar to the motivation for removing clauses in traditional logic programs: it reduces the length of computations of individual intentions.

For example, suppose an agent's original plan library doesn't include plans that let the rover react to possible alternative targets or to sundown. Then the agent wouldn't have more than a single intention at a time. Still, consider that the property to be checked is the same as in Figure 2a (that is, take that formula as the slicing criterion). Because the plans that make the agent transmit its data back to the ground team can only become intended after some point in the execution where place_spectrometer(R) has already happened, there's no need to consider that part of the intention's execution. It won't affect the property under consideration (in other words, that level of detail of the intention execution is irrelevant for the given property). The code slice that our slicing algorithm generates for the property in Figure 2a doesn't include those plans.

The second way slicing alleviates state explosion is by removing all plans used to handle particular external events. At any point during the computation associated with one intention, reachable states exist in which other intentions (other foci of attention) are created to handle events that belief revision might have generated. Slicing out such plans eliminates all such branches of the computation tree. An alternative reduction would be to avoid the environment generating such events in the first place (considering that they won't affect the property being verified anyway).

The specification in Figure 2b exemplifies this second type of state-space reduction. If that specification is used as the slicing criterion, we can safely remove the plan for reacting to possible ordinary targets (there's a particular one for reacting to rocks with green patches).

Although in this second example the slicing appears to be minor (just one plan isn't included in the slice), a considerable reduction of the state space can ensue, depending also on how dynamic the environment is. If the rover detects (and approaches) many possible targets while transmitting data back to Earth, this could generate many different system states in which the rover's attention is divided between two tasks (it must deal with two foci of attention simultaneously).

For a variation of the Mars rover example and the specifications shown in Figure 2, we obtained average improvements of 26 percent in terms of the time and memory required to model check the generated slices rather than the original pro-

$$\Box((\text{Does } amr \text{ place\_spectrometer(R)}) \rightarrow (\text{Bel } amr \text{ correctly\_positioned(R)}))$$

(a)

$$\Box((\text{Int } amr \text{ transmit\_remaining\_data(Day)}) \rightarrow \Diamond \neg((\text{Bel } amr \text{ data(spect,Rock,Day,\_)}) \wedge \neg(\text{Bel } amr \text{ downlink(ground,spect,Rock,Day)})))$$

(b)

Figure 2. Examples of properties using our specification language for the autonomous Mars rover scenario: (a) ensuring that the rover believes it's correctly positioned to examine a rock before doing so; (b) ensuring that the rover eventually transmits gathered data back to Earth whenever it intends to do so. (We use *amr* [autonomous Mars rover] to denote the agent.)

Abstraction and Slicing

Model-checking performance is sensitive to a system's state space—large systems require more time and memory to analyze. Furthermore, with concurrent systems (for example, agent execution environments), the state space grows exponentially with the number of concurrent components; this is the *state-space explosion problem*.

For model checking to scale to industrial-size examples, we must alleviate the state-space explosion problem. Consequently, researchers have proposed numerous state-space reduction techniques for use during model checking—abstraction being the most popular.

Abstraction can come in two forms: underapproximations and overapproximations of the original system. The former refers to analyzing a system that doesn't contain all the behaviors of the original. In this context, any erroneous behavior detected while model checking the abstract system is an error in the real system as well, but not finding any errors doesn't guarantee that no errors exist in the real system. Overapproximations, where the abstract system contains more behaviors than the original, preserves correctness, but an error found in the abstract system might not be present in the original.

Slicing is an abstraction technique used to reduce the portion of the program (system) that needs to be considered (called the slice) to satisfy a specific criterion (for example, the portion of the program that affects a variable's value at a specific program point).[1] A common use of slicing is to reduce the part of the program to consider during debugging.

Slicing can also reduce the program being considered before carrying out model checking. For model checking, the slicing criterion is to consider the program points that can affect the truth-value of propositions referred to in the temporal property being checked. This is called *property-based slicing* and creates a precise form of underapproximation of the original program. That is, with respect to the property being checked, all the behaviors of the original program, and only those, are present in the sliced (abstracted) program.

Reference

1. F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages*, vol. 3, no. 3, 1995, pp. 121–189.

gram. Interestingly, the slicing algorithms available with the target model checkers we've used can't detect the opportunities for slicing determined by the semantics of the BDI modalities. This shows the importance of having a specific slicing tool that works directly on AgentSpeak code. In addition, the slicing technique can have various other uses (for example, understanding AgentSpeak code in given plan libraries).

Toward dependable agent-based systems

Multiagent systems represent an approach to implementing distributed systems that departs radically from traditional approaches. In general, a much higher level of abstraction is used for such systems' design and implementation. The metaphors used in designing multiagent systems do more than just facilitate our conception of and interaction with the system.

Consider, for example, one of the most common sources of bugs in distributed systems: deadlocks or data races when accessing shared variables. Because of the frequency with which we find such errors in distributed systems, researchers have put considerable effort into practical model checkers to find this type of bug (examples of this appear elsewhere[2]). The multiagent-systems approach intrinsically avoids this kind of problem.

Typically, autonomous agents in a multiagent system must share and compete for resources. The usual metaphor is that certain agents own resources (of whatever type) and allocate them on the basis of some high-level mechanism—for example, one inspired by economic markets or negotiation techniques. Whatever approach we take for agent coordination, an agent's mental state is, by definition, private. Other agents have no access to an agent's mental state, except through high-level communication (based on the speech act theory). Certainly the protocols used—for example, in resource allocation mechanisms—must be verified because they too could suffer from common faults (such as starvation). Arguably, any such protocols that agents use can be independently checked for properties that guarantee that no such faults will occur. Ideally, we can expect to find, in the near future, libraries of such high-level agent coordination mechanisms already verified. Multiagent-systems practitioners could then simply use them off the shelf, knowing their properties and the ones agents must comply with to ensure certain properties of the overall system.

This points to a sophisticated use of compositional reasoning in verifying multiagent systems. Also, internal to each agent, some form of compositional reasoning is likely to be appropriate for verifying particularly complex agents. Recall the internal-action construct in AgentSpeak that we described earlier. We can give the code that implements such actions in any programming language and can use it to integrate legacy code. These algorithms can be independently verified, using traditional techniques for model-checking programs, to ensure they produce the expected results. When model checking the agents, we're interested only in the properties of their high-level reasoning, which takes for granted the details of individual actions.

So, automatic verification in multiagent systems can work on several levels of abstraction: the level of social interactions and social organizations; the individual level where the agent's reasoning is targeted; and also an even lower level concerning specific algorithms or legacy code (in the autonomous Mars rover scenario, this would correspond, for example, to the image-processing software used to check for a safe path to a certain rock). Thus, it seems that the various levels of abstraction in multiagent systems, which are metaphors for the various levels in human societies, might also help in verifying large systems. In addition, they could provide system designers and programmers with more natural ways of coping with the increasing complexity of their tasks.

It's still far from clear whether we'll be able to satisfactorily address issues related to the openness of multiagent systems (when varying numbers of heterogeneous agents may interact at a given time) and to evolving agents (that is, coping with emergent phenomena).

The Authors

Rafael H. Bordini is a lecturer in computer science at the University of Durham (but was a research fellow at the University of Liverpool when doing the research reported here). He's also associated with the program for post-graduate studies in computing at the Federal University of Rio Grande do Sul, Brazil. His research interests include agent-oriented programming languages, verification of multiagent systems by model checking, and applications of multiagent systems to social simulation. He received his PhD in computer science from the University of London (University College London). Contact him at the Dept. of Computer Science, Univ. of Durham, Durham DH1 3LE, UK; r.bordini@durham.ac.uk.

Michael Fisher is a professor of computer science in the University of Liverpool's Department of Computer Science. He also heads the Logic and Computation research group and is director of the Liverpool Verification Laboratory. His research interests include using logic in computer science and AI, particularly temporal reasoning, theorem proving, programming languages, and agent-based systems. Contact him at the Dept. of Computer Science, Univ. of Liverpool, Liverpool L69 3BX, UK; m.fisher@csc.liv.ac.uk.

Willem Visser works at the Research Institute for Advanced Computer Science and conducts his research within the Automated Software Engineering group at NASA Ames. His research focuses on doing model checking for actual programs and, more specifically, on Java PathFinder, a model checker for Java. He's also interested in testing. He received his PhD in computer science from the University of Manchester. Contact him at RIACS/NASA Ames Research Center, Moffett Field, CA 94035; wvisser@email.arc.nasa.gov.

Michael Wooldridge is a professor of computer science in the Computer Science Department at the University of Liverpool. He currently heads the department and is also a member of the Agent Applications, Research, and Technology research group, which carries out both pure and applied research in the area of autonomous agents and multiagent systems. His main interests have been in the use of formal methods for specifying and reasoning about multiagent systems. Other interests include agent-oriented software engineering and negotiation. Contact him at the Dept. of Computer Science, Univ. of Liverpool, Liverpool L69 3BX, UK; m.j.wooldridge@csc.liv.ac.uk.

However, although this area is still at an early stage, we think that combining deductive[8] and algorithmic verification techniques (such as those presented here) will have an important role in the verification of such types of multiagent systems in the future. ■

Acknowledgments

An EC Marie Curie Fellowship under contract number HPMF-CT-2001-00065 supported research at the University of Liverpool.

References

1. M. Wooldridge, *Reasoning about Rational Agents*, MIT Press, 2000.
2. W. Visser et al., "Model Checking Programs," *Proc. 15th Int'l Conf. Automated Software Eng.* (ASE 2000), IEEE CS Press, 2000, pp. 3–12.
3. A.S. Rao, "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language," *Proc. 7th Workshop Modeling Autonomous Agents in a Multiagent World* (MAAMAW 96), LNAI 1038, Springer-Verlag, 1996, pp. 42–55.
4. R.H. Bordini et al., "Model Checking AgentSpeak," *Proc. 2nd Int'l Joint Conf. Autonomous Agents and Multiagent Systems* (AAMAS 2003), ACM Press, 2003, pp. 409–416.
5. R. Washington et al., "Autonomous Rovers for Mars Exploration," *Proc. Aerospace Conf.*, IEEE Press, 1999, pp. 237–251.
6. F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages*, vol. 3, no. 3, 1995, pp. 121–189.
7. R.H. Bordini et al., "State-Space Reduction Techniques in Agent Verification," *Proc. 3rd Int'l Joint Conf. Autonomous Agents and Multiagent Systems* (AAMAS 2004), ACM Press, 2004, pp. 896–903.
8. M. Fisher, "Temporal Development Methods for Agent-Based Systems," to be published in *J. Autonomous Agents and Multiagent Systems*, 2004.