Integrating Abstract State Machines and Interpreted Systems for Situation Analysis Decision Support Design

Roozbeh Farahbod and Uwe Glässer
Software Technology Lab
Simon Fraser University
Burnaby, Canada
Email: {roozbehf, glaesser}@cs.sfu.ca

Abstract—A formal approach to the design of situation analysis and decision support systems is justified and unavoidable if one is interested in reproducibility/traceability of results, satisfaction of constraints, and a language to represent and reason about dynamic situations. In this paper, we propose the integration of two multiagent modeling paradigms, Abstract State Machines and Interpreted Systems, to develop a comprehensive framework for computational Situation Analysis (SA) as a basis for design and development of decision support systems. Due to the similarities of the underlying modeling concepts, a systematic integration of the two paradigms seems sensible, as each one has its particular focus and strength, complementing each other in several respects. Our approach builds on multiagent systems theories to formalize the distributed aspect, allows for reasoning about knowledge, uncertainty and belief change, and enables rapid prototyping of abstract executable decision support system models.

Keywords: Situation Analysis, Information Fusion, Decision Support Systems, Distributed Systems, Intelligent Systems, Resource Management, Abstract State Machines, Formal Specification, High Level Design

I. INTRODUCTION

In this paper, we propose the integrated use of two known mathematical modeling paradigms for analyzing and reasoning about behavioral aspects of multiagent systems as a basis for a comprehensive computational framework for high-level design and experimental validation of Situation Analysis (SA) decision support systems. Inspired by recent work at Defence Research and Development Canada at Valcartier\(^1\) [2], the work presented here originates from the observation that Abstract State Machines [3] and Interpreted Systems [4] provide very similar, although not identical, formal semantic frameworks for describing behavior of distributed systems in abstract operational, functional and declarative terms. Both approaches view distributed computations of concurrent and reactive systems as evolution of states, assuming multiple computational agents interacting with one another as well as with their operational environment representing the external world. The underlying computation model, in both cases, defines the behavior of a distributed system as the set of all admissible runs originating in a distinguished set of initial system states. Agents operate autonomously, performing their computation steps by evaluating conditions based on their partial view or local knowledge of a global state such that in any given run all the computation steps of every individual agent are linearly ordered.

Situation Awareness, a state in the mind of a human, is essential to conduct decision-making activities. It is about the perception of the elements in the environment, the comprehension of their meaning, and the projection of their status in the near future [5]. Agents develop an understanding of a situation based on a discrete perception and evaluation of events as they unfold over time and forecasting their anticipated evolution in the future. Situation Analysis (SA) is defined as a process, the examination of a situation, its elements, and their relations, to provide and maintain a product, i.e., a state of situation awareness for the decision maker and information fusion is a key enabler to achieve that state [6].

In [7], the rational for establishing a formal semantic foundation for the design of situation analysis and decision support systems is discussed in detail. In this light, a systematic approach combining Abstract State Machines and Interpreted Systems seems appealing, as each of the two semantic modeling frameworks has its particular focus and strength, complementing each other in several respects; at the same time, they both share common abstraction principles for describing distributed system behavior based on an abstract operational view of multiagent systems. Of specific interest here are the underlying notions of concurrency, reactivity and time, and mechanisms for reasoning about knowledge and uncertainty. Additionally, pragmatic considerations regarding practical needs for system design and development are relevant to support the systematic refinement of abstract specifications into executable models serving as a basis for rapid prototyping and experimental validation of decision support systems.

The paper structure is as follows. Section II outlines basic Abstract State Machine (ASM) concepts for modeling behavior of distributed systems and the CoreASM language for making ASM models executable on real machines, specifically for rapid prototyping purposes. Section III recalls relevant concepts of Interpreted Systems. Through an example, Section IV illustrates the similarities between the two approaches and shows how one can combine ASM and Interpreted Systems, while Section V addresses anticipated benefits of the proposed approach to SA. Section VI concludes the paper.

\(^1\)Using Interpreted Systems for Situation Analysis was first proposed in [1].
II. ABSTRACT STATE MACHINES

A central question in computing science is how to precisely define the notion of algorithm. Traditionally, Turing machines have been used in the study of the theory of computation as a formal model of algorithms [8]. For semantic purposes, however, this model is utterly inappropriate due to its fixed level of abstraction. The origin of the ASM modeling paradigm was the idea to devise a generalized machine model so that any algorithm, never mind how abstract, can be modeled at its natural level of abstraction, meaning that every computation step of the algorithm essentially translates into a corresponding step performed by the machine model. Theoretical foundations show that the notion of sequential algorithm and the one of parallel algorithm are captured in the aforementioned sense by the sequential ASM [9] and the parallel ASM [10] model respectively. For modeling distributed systems, the model of distributed ASM (DASM), a generalization of the two other models, provides an asynchronous computation model based on a multiagent paradigm. A formal proof that this model captures distributed algorithms is not known yet; still, there is considerable empirical evidence from extensive practical applications that this model faithfully reflects the common understanding of the notion of distributed algorithm.

Abstract State Machines [3] are known for their versatility in semantic modeling of algorithms, architectures, languages and protocols—extending to virtually all kinds of sequential, parallel and distributed systems. Widely recognized applications include semantic foundations of system design languages, like SDL [11], VHDL [12], SystemC [13]; programming languages, like JAVA [14], C# [15], Prolog [16]; distributed communication architectures [17]; Web service architectures [18] and mobile wireless networks [19]; among many others. Leaning towards practical applications of formal methods, ASM abstraction principles facilitate revealing the essential behavioral and architectural characteristics inevitably present in every conceptual system design—but often not in an explicit form—turning abstract requirements and design specifications into precise blueprints so as to analyze and reason about key system attributes prior to actually building the system.

The following outlines the basic concepts for modeling behavioral aspects of distributed systems in terms of abstract machine runs performed by a distributed ASM. We describe these concepts at an intuitive level of understanding using common notions and structures from computational logic and discrete mathematics. For further details, we refer to [3], [20].

A. Concurrency, reactivity and time

A distributed ASM, or DASM, defines the concurrent and reactive behavior of autonomously operating computational agents that cooperatively perform distributed computations. Intuitively, every computation step of the DASM involves one or more agents, each performing a single computation step according to their local view of a globally shared machine state. The underlying semantic model regulates interactions between agents so that potential conflicts are resolved according to the definition of partially ordered runs [17].

A DASM $M$ is defined over a given vocabulary $V$ by its program $P_M$ and a non-empty set $I_M$ of initial states. $V$ consists of some finite collection of symbols for denoting the mathematical objects and their relation in the formal representation of $M$, where we distinguish domain symbols, function symbols and predicate symbols. Symbols that have a fixed interpretation regardless of the state of $M$ are called static; those that may have different interpretations in different states of $M$ are called dynamic. A state $S$ of $M$ results from a valid interpretation of all the symbols in $V$ and constitutes a variant of a first-order structure, one in which all relations are formally represented as Boolean-valued functions.

Concurrent control threads in an execution of $P_M$ are modeled by a dynamic set AGENT of computational agents. This set may change dynamically over runs of $M$, as required to model a varying number of computational resources. Agents of $M$ normally interact with one another, and typically also with the operational environment of $M$, by reading and writing shared locations of a global machine state.

$P_M$ consists of a statically defined collection of agent programs $P_{M_1},...,P_{M_k}$, $k \geq 1$, each of which defines the behavior of a certain type of agent in terms of state transition rules. The canonical rule consists of a basic update instruction of the form

$$f(t_1,t_2,...,t_n):=t_0,$$

where $f$ is an n-ary dynamic function symbol and each $t_i$ $(0 \leq i \leq n)$ a term. Intuitively, one can perceive a dynamic function as a finite function table where each row associates a sequence of argument values with a function value. An update instruction specifies a pointwise function update: an operation that replaces a function value for specified arguments by a new value to be associated with the same arguments. In general, rules are inductively defined by a number of well defined rule constructors, allowing the composition of complex rules for describing sophisticated behavioral patterns.

A computation of $M$, starting with a given initial state $S_0$ from $I_M$, results in a finite or infinite sequence of consecutive state transitions of the form

$$S_0 \xrightarrow{\Delta S_0} S_1 \xrightarrow{\Delta S_1} S_2 \xrightarrow{\Delta S_2} \cdots,$$

such that $S_{i+1}$ is obtained from $S_i$, for $i \geq 0$, by firing $\Delta S_i$ on $S_i$, where $\Delta S_i$ denotes a finite set of updates computed by evaluating $P_M$ over $S_i$. Firing an update set means that all the updates in the set are fired simultaneously in one atomic step. The result of firing an update set is defined if and only if the set does not contain any conflicting updates (attempting to assign different values to the same location).

2Information on ASM applications and foundations is provided by the ASM Research Center at www.asmcenter.org.
A partially ordered run defines a class of admissible runs of $M$ rather than a particular run. In general, it may require more than one (even infinitely many) partially ordered runs to capture all admissible runs of $M$. From the coherence condition it follows that all linearizations of the same finite initial segment of a run of $M$ have the same final state.\(^5\)

Example. Door and window manager: Assume two propositional variables, $\text{door}$ and $\text{window}$, where $\text{door} = \text{true}$ means that ‘the door is open’ and $\text{window} = \text{true}$ means that ‘the window is open’. There are two distinct agents: a door-manager $d$ and a window-manager $w$.

\(^5\)Intuitively, a finite initial segment of a partially ordered run $\rho$ is a finite subset of $\Lambda$ corresponding to a (finite) prefix of $\rho$.

---

### CoreASM modeling suite

CoreASM \(^6\) is a lean, extensible and executable specification language together with a supporting tool environment for high-level design, experimental validation and formal verification (where appropriate) of ASM models. CoreASM has been designed with a well-defined plug-in architecture to offer the system architect the freedom and flexibility required to fully explore the problem space without restriction.

The CoreASM language focuses on the early phases of the software design process, emphasizing freedom of experimentation and the evolutionary nature of design being a creative activity. It encourages rapid prototyping of abstract machine models for testing and design space exploration facilitating agile software development \[^24\]. By minimizing the need for encoding in mapping the problem space to a formal model, it allows writing highly abstract and concise specifications, starting with mathematically-oriented, abstract and untyped models, gradually refining them down to more concrete versions with a degree of detail and precision as needed.

The CoreASM environment consists of a platform-independent engine for executing CoreASM specifications and a GUI for interactive visualization and control of simulation runs. The engine comes with a sophisticated and well defined interface thereby enabling future development and integration of complementary tools, e.g., for symbolic model checking and automated test generation. The design of CoreASM is novel and the underlying principles are unprecedented among the existing executable ASM languages \[^23\].

---

### III. Interpreted Systems

Fagin et al. introduced the notion of interpreted systems \[^4\] as a formal semantic framework for reasoning about knowledge and uncertainty in multiagent systems. We briefly recall here the basic notions and definitions.

---

\(^6\)CoreASM is an Open Source project and is readily available at www.coreasm.org
A. Runs and systems

According to [4], a multiagent system can be conceptually divided into two components: the agents \( A = \{1, \ldots, n\} \) and the environment \( e \), which can be viewed as a special agent. The global state of the system with \( n \) agents is defined to be an \((n+1)\)-tuple \((s_e, s_1, \ldots, s_n)\), where \( s_e \) is the state of the environment and \( s_i \) is the local state of agent \( i \). The set of all global states of the system is defined as \( G : L_e \times L_1 \times \ldots \times L_n \), where \( L_e \) is the set of possible states for the environment and \( L_i \) is the set of all possible local states of agent \( i \). To model the changes of the system’s global state in time, the notion of run is introduced as a function from time to global states \( G \), with the assumption that time ranges over natural numbers. A system can have many possible runs. The initial global state of a system with a possible run \( r \) is \( r(0) \). A pair \((r, m)\) consisting of a run \( r \) and a time \( m \) is referred to as a point in run \( r \).

A system \( R \) over \( G \) is defined as a set of runs over \( G \).

B. Actions, protocols, and programs

A round takes place between two points in a run, and a round \( m \) in run \( r \) is defined to take place between points \( r(m-1) \) and \( r(m) \). Agents and the environment change the global state by performing actions in rounds. Let \( ACT_i \) be the set of actions that can be performed by agent \( i \), and let \( ACT_e \) be the set of actions that can be performed by the environment. A joint action is a tuple \((a_e, a_1, \ldots, a_n)\) of actions performed by the environment and the set of agents, where \( a_e \in ACT_e \) and \( a_i \in ACT_i \) for \( i \) in \( 1 \ldots n \).

Joint actions cause the system to change its global state, and the change is modeled by a global state transformer function \( T : G \rightarrow G \) that is associated to each joint action \((a_e, a_1, \ldots, a_n)\). A transition function \( \tau \) is a mapping that associates a global transformer with each joint action. It is required that \( \tau(a_e, a_1, \ldots, a_n)(s_e, s_1, \ldots, s_n) \) be defined for each joint action \((a_e, a_1, \ldots, a_n)\) and each global state \((s_e, s_1, \ldots, s_n)\).

Agents perform actions according to some protocol, which is a rule for selecting actions. A protocol \( P_i \) for the agent \( i \) is formally defined as \( P_i : L_i \rightarrow \mathcal{P}(ACT_i) \setminus \{\emptyset\} \). A protocol \( P_i \) is deterministic if \( \forall s_i \in L_i \ | P_i(s_i)\rangle = 1 \). In a similar fashion, a protocol \( P_e \) for the environment is defined as a function from \( L_e \) to nonempty subsets of \( ACT_e \). A joint protocol \( P \) is a tuple \((P_1, \ldots, P_n)\) consisting of all the protocols \( P_i \), for each of the agents \( i = 1, \ldots, n \). Note that the environment’s protocol \( P_e \) is not included in the joint protocol. The protocol of the environment is usually supposed to be given and \( P \) and \( P_e \) can be viewed as the strategies of opposing players.

A context \( \gamma \) is defined as a tuple \((P_e, G_0, \tau, \Psi)\), where \( P_e \) is a protocol for the environment, \( G_0 \) is a nonempty subset of \( G \) describing the initial system state, \( \tau \) is a transition function and \( \Psi \) is an admissibility condition on runs specifying which runs are “acceptable”. Formally, \( \Psi \) is a set of runs; \( r \in \Psi \) if \( r \) satisfies the condition \( \Psi \). In practice, \( \Psi \) can be used to shrink down the system or to model fairness conditions. The combination of a context \( \gamma \) and a joint protocol \( P \) for the agents uniquely determines a set of runs.

Protocols are typically described by means of programs written in some programming language. A standard program for agent \( i \) is a statement of the form

\[
\text{case of } \begin{cases} 
\text{if } t_1 \text{ do } a_1 \\
\text{if } t_2 \text{ do } a_2 \\
\ldots 
\end{cases}
\]

where the \( t_j \)'s are standard tests for agent \( i \) and the \( a_j \)'s are actions of agent \( i \) (i.e., \( a_j \in ACT_i \)).

C. Interpreted systems

To incorporate knowledge into the framework, the notion of interpreted systems is introduced. Let \( \Phi \) be a set of primitive propositions over \( G \), describing basic facts about the system. An interpreted system \( I \) is a pair \((R, \pi)\), where \( R \) is a system over a set \( G \) of global states, and \( \pi \) is a set of interpretations of \( \Phi \) over \( G \). Thus, for every \( p \in \Phi \) and \( s \in G \), we have \( \pi(s)(p) \in \{true, false\} \). Notice that \( \Phi \) and \( \pi \) constitute additional structure on top of \( R \).

Let \( \mathcal{L}(\Phi) \) be the language of \( \Phi \), i.e., the set of well-formed formulas built using the primitive propositions and the two operators \( \neg \) and \( \land \). The satisfaction of formulas in \( \mathcal{L}(\Phi) \) at any point in the system is given by:

\[
(\mathcal{I}, r, m) \models p \iff \pi((r(m))(p)) = \text{true} \\
(\mathcal{I}, r, m) \models \phi \land \psi \iff (\mathcal{I}, r, m) \models \phi \land (\mathcal{I}, r, m) \models \psi \\
(\mathcal{I}, r, m) \models \neg \phi \iff (\mathcal{I}, r, m) \not\models \phi
\]

Two points \((r, m)\) and \((r', m')\) are said to be indistinguishable to agent \( i \), written as \((r, m) \sim_i (r', m')\), if agent \( i \) has the same local state in both points. The operator \( K_i, i \in A \) is defined to model the knowledge of agent \( i \) at any given point in a system. The semantics of \( K_i \) is defined as follows:

\[
(\mathcal{I}, r, m) \models K_i \phi \iff \forall (r', m') \ (r, m) \sim_i (r', m') \Rightarrow (\mathcal{I}, r, m) \models \phi
\]

which reads “at point \((r, m)\) in the interpreted system \( \mathcal{I} \), the agent \( i \) knows that \( \phi \) if and only if \( \phi \) holds in all the points that agent \( i \) cannot distinguish from \((r, m)\)”.

Finally, temporal operators \( \square \) (“always”), \( \Diamond \) (“eventually”), \( O \) (“next time”), and \( U \) (“until”) are defined to reason about temporal evolution of the system.

IV. DASM AND INTERPRETED SYSTEMS

It is interesting to observe the similarities between Abstract State Machines and Interpreted Systems, especially in capturing the notions of agents, concurrency, runs, update actions, and programs. In this section, we illustrate these similarities using a simple surveillance scenario originally presented in [2]. We will show how one can benefit from using ASM to model a multiagent system and still be able to apply the Interpreted Systems approach of [2] for situation analysis.

1Here we basically follow the introduction presented in [2].
A. The abstract model

The scenario involves two agents, agent\(_1\) and agent\(_2\), in a 2D environment (see Figure 1). Both agents are able to move in 8 possible directions and they can both sense other agent’s range \(\rho\) and bearing \(\theta\) with some error. The local state of each agent is composed of successive observations about the other agent’s position. The state of the environment contains the accurate target positions. Using the ASM notation, we have:

\[
\text{universe } \text{AGENT} = \{\text{agent}_1, \text{agent}_2, \text{env}\}
\]

\[
\text{universe } \text{MOVEMENT} = \{\text{N, NW, W, WS, S, SE, E, EN}\}
\]

// real position of agents, as part of the environment
\(\rho, \theta : \text{AGENT} \rightarrow \mathbb{R}\)

// observed range and bearing of agent 1 (made by agent 2)
\(\hat{\rho}_1, \hat{\theta}_1 : \rightarrow \mathbb{R}\)

\(\rho_{ih}, \theta_{ih} : \rightarrow \text{SEQUENCE}(\mathbb{R})\)

// observed range and bearing of agent 2 (made by agent 1)
\(\hat{\rho}_2, \hat{\theta}_2 : \rightarrow \mathbb{R}\)

\(\rho_{2h}, \theta_{2h} : \rightarrow \text{SEQUENCE}(\mathbb{R})\)

// error values set by the environment
\(\text{error}_\rho : \text{AGENT} \rightarrow \mathbb{R}\)

\(\text{error}_\theta : \text{AGENT} \rightarrow \mathbb{R}\)

where \(\rho(a)\) and \(\theta(a)\) are the real range and bearing of agent \(a\), \(\hat{\rho}_i\) and \(\hat{\theta}_i\) are the observed range and bearing\(^9\) and \(\rho_{ih}\) and \(\theta_{ih}\) are the history of observations of range and bearing of agent \(i\) by the other agent, and \(\text{error}_\rho(a)\) and \(\text{error}_\theta(a)\) are the observation errors of agent \(a\).

To make the model more general and scalable, real ranges and bearings are defined as functions over agents. The action sets of the two agents, ACT\(_1\) and ACT\(_2\) are equivalent to the Movement universe defined above.

According to the scenario, agent\(_1\) is stationary. Its purpose is to observe the position of agent\(_2\) and to send a message to a designated agent (outside of the system) if agent\(_2\) enters an area of interest (AOI). On the other hand, agent\(_2\) is actively moving toward agent\(_1\) until it finds out that it is “too close” to agent\(_1\), in which case it makes a U-turn.\(^10\) So, we define the programs of agent\(_1\) and agent\(_2\) as follows:

\[
\text{Agent1Program} \equiv \text{RecordObservation(agent}_2)\]

\[
\text{Agent2Program} \equiv \text{RecordObservation(agent}_1)\]

At this level, we abstract away from certain details. For example, since the scenario does not elaborate on how and to whom agent\(_1\) sends its messages, we leave the SendMessage rule abstract. Also, since the area of interest and the exact measures for “closeness” of agent\(_1\) and agent\(_2\) are not defined, the functions isInAOI and tooClose are left abstract as well. The rule RecordObservation\((a)\) is a placeholder for the actual task of maintaining the observation history.

An interesting observation is that the local state of agent\(_2\), as described by the scenario, seems to be missing an important piece: the movement direction of agent\(_2\), toward or away from agent\(_1\). At some point in time, agent\(_2\) observes that it is too close to agent\(_1\) and so makes a U-turn. The direction of the agent\(_2\) is a dynamic attribute of the agent and needs to be captured in the local state of the agent. Here, we model the direction of agent\(_2\) by a function \(\text{dir}\), which is initially set to \(\text{toward}\) and will be changed to \(\text{away}\) the first time agent\(_2\) observes that it is too close to agent\(_1\).\(^11\)

In this scenario, the environment models the uncertainty of error values. The action of the environment is to set the error values for observed ranges and bearings. Hence, the action set of the environment ACT\(_e\) is a set of tuples of the form \((\epsilon^i_{\rho}, \epsilon^i_{\theta}, \epsilon^i_{\rho}, \epsilon^i_{\theta})\) in which \(\epsilon^i_{\rho} \in E_{\rho}^i\) and \(\epsilon^i_{\theta} \in E_{\theta}^i\) are range and bearing observation errors for agent \(i\) and \(E_{\rho}^i\) and \(E_{\theta}^i\) are the corresponding error ranges. In our ASM model, the environment program models this behavior by non-deterministically choosing values from \(E_{\rho}^i\) and \(E_{\theta}^i\) and

\(^8\)Courtesy of A.-L. Jousselme and P. Maupin

\(^9\)A better approach would be to define \(\hat{\rho}\) and \(\hat{\theta}\) as functions over agents, but to match the original syntax of the scenario, we define them as individual indexed functions.

\(^10\)The description of the scenario is vague on whether agent\(_2\) will ever try to move back toward agent\(_1\) or not. We assume that it will not try to come close again.

\(^11\)One can also argue that the knowledge about the close encounter of the agents is implicitly encoded in the observation history of agent\(_2\) and the direction can be dynamically calculated based on the observation history.
updating $error_p$ and $error_0$ for agents $agent_1$ and $agent_2$.

EnvironmentProgram $\equiv$

\[
\text{forall } a \in \{agent_1, agent_2\} \text{ do}
\]
\[
\text{choose } e_p, e_0 \in E_p(a), e_0 \in E_0(a) \text{ do}
\]
\[
error_p(a) := e_p, \quad error_0(a) := e_0
\]

At this point, we have an abstract operational model of the scenario in form of a multiagent ASM.

B. Situation Awareness

Following the approach of [2], let $\Phi$ be the basic set of our propositions, $\phi_p \in \Phi$ be "the range of agent 2 crosses AOI", and $\phi_0 \in \Phi$ be "the bearing of agent 2 crosses AOI". The formula $\phi_{\text{aoi}} = \phi_p \land \phi_0$ will then be "agent 2 is in AOI". In the program of $agent_1$, the function $is\text{NAOI}(agent_2)$ represents the awareness of $agent_1$ about $\phi_{\text{aoi}}$, and it can be evaluated using the range and bearing of $agent_2$ as observed by $agent_1$:

\[
is\text{NAOI}(agent_2) = \hat{\rho}_2 \in \text{AOI}_p \land \hat{\theta}_2 \in \text{AOI}_0
\] (1)

It is important to note that since there is a possible error in the observation of $agent_1$, $is\text{NAOI}(agent_2)$ being true does not necessarily mean that $\phi_{\text{aoi}}$ holds as well; i.e., $agent_2$ could actually be outside of the area of interest. The value of $is\text{NAOI}(agent_2)$ simply reflects the state of awareness of $agent_1$ about the position of $agent_2$, which may differ from the reality. This is captured in (1) by using the observed range and bearing, $\hat{\rho}$ and $\hat{\theta}$, rather than the real values $\rho$ and $\theta$.

The same holds for the awareness of $agent_2$ about its distance from $agent_1$. Let $\phi_c \in \Phi$ be "the range of agent 1 is too close". The function $\text{tooClose}(agent_1)$ represents the state of awareness of $agent_2$ about the truth of $\phi_c$ and can be derived from the observed range of $agent_2$:

\[
\text{tooClose}(agent_1) = \hat{\rho}_1 < \text{threshold}_p
\] (2)

where $\text{threshold}_p$ is the minimum distance that $agent_2$ is willing to have with $agent_1$. Again, $\text{tooClose}(agent_1)$ being true does not necessarily mean that $\phi_c$ holds.

We can further extend the model and introduce non-trivial computable formulas such as $\phi_m = "agent 2 is coming toward agent 1"$. In general, derived functions can be defined to model the awareness of agents about the truth values of such formulas. For example, the following function can represent the awareness of $agent_1$ about $\phi_m$:

\[
\text{app\text{roaching}}_2 = \hat{\rho}_2h(\text{last}) < \hat{\rho}_2h(\text{last} - 1)
\] (3)

C. Situation Analysis

Once we have a proper model of the scenario, various types of queries can be used for situation analysis. Using the Interpreted Systems framework, Jousselme and Maupin suggest the following three types of queries for situation analysis [2]:

1) Queries about truth, such as "Does $\phi_{\text{aoi}}$ hold in a given state $s$?";

2) Queries about knowledge, such as "Does $agent_2$ know that $\phi_c$ holds in a given state $s$?"; and

3) Queries about time, such as "Does $\phi_{\text{aoi}}$ eventually hold in a run $r$ of the system?"

By combining Interpreted Systems and multiagent ASMs, these queries can not only be analyzed using the proposed methods in [2], but they can also be examined either by explicitly encoding the queries as derived functions (such as $is\text{NAOI}$ and $\text{tooClose}$) and running the executable model, or by applying the available model checking techniques for ASM [25], [26]. Thus, the approach of integrating ASM with Interpreted Systems is consistent with the proposed approach of [2]. Note that the idea of using model checking for situation analysis has first been proposed in [2].

D. Executable model

In order to put the model into practice and to realize what is practically feasible, one needs to experiment with the model; that is, the model has to be machine executable.

In this section, we produce an executable model of the scenario through refinement of the abstract rules and functions of the model presented in Section IV-A. We then use the CoreASM execution engine to run the model. Such a refinement, with the goal of producing an executable model, is interesting in two aspects: a) it helps finding ambiguities, missing pieces and loose-ends of the model and forces the system analyst/modeler to think clearly about the main concepts and their definitions, and b) it supports experimental validation through execution (simulation).

1) Real positions and rules of movement: The refinement of the routines $\text{MoveToward}$ and $\text{MoveAway}$ requires a proper encoding of the positions of the agents. Although it is not precisely stated in the scenario, the real positions of the agents appear to be relative to the positions of their observers. Regardless of the encoding one chooses, the observed positions and the rules of movement both depend on the real position values and must be defined consistently. In our refined model, we keep the actual position of every agent in an $(x, y)$ coordinate which makes it easier to define the movement routines. Real relative bearing and range values are simply calculated based on the actual positions of agents.

2) Observed values: An important piece that was left abstract in our model, and was also missing in the original scenario, is the definition of the observed range and bearing functions $\hat{\rho}_i$ and $\hat{\theta}_i$. These functions play an important role in the model and their formal definition is important for proper situation analysis. While the original model does not state how the observed values are produced, it is clear that the observed position values of agent $i$ are functions of both the real position of agent $i$ and the corresponding observation error. Hence, the following equations are reasonable candidates:

\[
\hat{\rho}_i = \rho(\text{agent}_i) + e_p^i, \quad \hat{\theta}_i = \theta(\text{agent}_i) + e_\theta^i
\]

So, the observation functions can be defined as derived functions over the actual positions of the agents. Another approach,
though less intuitive, would be to have the environment explicitly update the observed values in every step of the system.

3) **Recording observation**: Since we define the observation values as derived functions over the real positions of the agents, to record the observation history we simply add the current values of $\hat{\rho}_i$ and $\hat{\theta}_i$ to the observation histories of each agent:

\[
\text{RecordObservation}(agent_i) \equiv \\
\text{add } \hat{\rho}_i \text{ to } \rho_{ih} \\
\text{add } \hat{\theta}_i \text{ to } \theta_{ih}
\]

At this point, we have a machine executable model of the scenario and we can use the CoreASM execution engine to run a simulation of the scenario. We start with an initial state in which $agent_1$ is located at $(0, 0)$ and $agent_2$ is located at $(15, 10)$. To monitor the position of $agent_2$, we extend the program of the environment to print the current positions of the agents in every step. This is a sample output of the simulation:

agent1:(0, 0) - agent2:(15, 10)
agent1:(0, 0) - agent2:(14, 9)
agent1:(0, 0) - agent2:(13, 8)
agent1:(0, 0) - agent2:(12, 7)
agent1:(0, 0) - agent2:(11, 6)
agent1:(0, 0) - agent2:(10, 5)
Abstract Call: SendMessage(Agent 2 is in AOI.)
agent1:(0, 0) - agent2:(11, 5)
agent1:(0, 0) - agent2:(10, 6)
Abstract Call: SendMessage(Agent 2 is in AOI.)
agent1:(0, 0) - agent2:(9, 7)
Abstract Call: SendMessage(Agent 2 is in AOI.)
agent1:(0, 0) - agent2:(8, 8)
agent1:(0, 0) - agent2:(7, 9)
agent1:(0, 0) - agent2:(6, 10)
agent1:(0, 0) - agent2:(5, 11)
agent1:(0, 0) - agent2:(4, 12)
agent1:(0, 0) - agent2:(4, 13)
agent1:(0, 0) - agent2:(4, 14)

The simulation starts with $agent_2$ moving toward $agent_1$. When $agent_1$ observes that $agent_2$ is in the area of interest, it sends a message using the abstract routine SendMessage (which is intentionally left abstract). After some time, $agent_2$ realizes that it is too close to $agent_1$ and makes a U-turn, moving away from $agent_1$.

V. **ANTICIPATED BENEFITS OF OUR APPROACH TO SA**

The Interpreted Systems approach provides a formal framework for reasoning about knowledge and uncertainty and for dealing with belief change concepts in a distributed systems context. Hence, the underlying view of system behavior is geared towards theoretical aspects of system modeling, building on abstract mathematical concepts that appear to be further away from practical system design and development. Concepts like the global state transformer function $T : \mathcal{G} \rightarrow \mathcal{G}$ (explicitly addressing the entire state set), the abstract definition of protocol, and a loose notion of concurrency, while being adequate for theoretical considerations, leave a gap for practical applications. In light of practical considerations, design and development of novel decision support systems that are based on a formal situation analysis model, like the interpreted algorithmic belief change system proposed in [1], [2], [7], in fact, means considerable challenges regarding the software systems that implement multiagent behavior on real machines. Controlled experiments to systematically explore the practicability of SA concepts in meaningful application scenarios call for rapid prototyping of abstract system models. Given that realistic scenarios typically mean complex systems, established system design concepts are required to manage complexity through systematic approaches to modularization, refinement, validation and verification of high-level models, making software systems design predictable and reliable.

Abstract State Machines are known for their practical side of formal semantic modeling pertinent to high-level specification and design. This formalism has been used extensively over many years to model complex distributed systems, resulting in solid methodological foundations. Thus, ASM refinement and modularization techniques [27] are well established and have been used successfully in various real-life (industrial) applications. Systematic concepts and methods for bridging the gap between formal and empirical approaches, such as the ASM ground model principles and best practices [28], have proven useful for managing complexity in practical design contexts. ASM models of multiagent distributed systems are executable in principle and, when combined with CoreASM tool support, enable experimental validation of highly abstract requirements and design specifications through both symbolic execution (model checking) and simulation and testing, and facilitate interoperability with other tools and analytical techniques in an open tool environment (see www.coreasm.org). The ASM modeling approach has also been used as a basis for building knowledge based systems [29]; however, ASMs do not provide specific support for reasoning about knowledge and uncertainty, the way Interpreted Systems do.

In combination, Abstract State Machines and Interpreted Systems can provide a comprehensive semantic framework in which knowledge, information and uncertainty can be represented, combined, managed, reduced, increased and interpreted. This framework builds on multiagent systems theories to formalize the distributed aspect and allows for reasoning about knowledge, uncertainty and belief change; at the same time, it also enables rapid prototyping of abstract executable decision support system models for performing experiments to validate and reason about system designs prior to actually building the systems. Given the similarities of the underlying modeling concepts, which are based on common foundations of computational logic and discrete mathematics to formalize multiagent behavior in terms of runs of state transition systems, a systematic (coherent and consistent) integration of the two modeling paradigms appears a realistic goal. Thus, one can expect that the proposed framework, the formal modeling language and the computational tools that come with it will open up new avenues to computer based SA in a way that is not only theoretically sound but also allows to explore the practicalities of building novel decision support...
systems. Our future work will focus on a comprehensive case study that is supposed to demonstrate the feasibility and practical relevance of the approach in a realistic SA setting.

VI. FINAL REMARKS

Arguably, a formal computational approach to the design of situation analysis and decision support systems is justified and unavoidable if one is interested in reproducibility/traceability of results (e.g., explanations), satisfaction of constraints (e.g., how much time and memory are needed), and most of all a language to represent and reason about dynamic situations [7]. Taking the best from two mathematical approaches to multiagent system modeling seems a sensible and promising approach. More research and development needs to be done and much more complex examples need to be investigated to fully understand how a combined approach works in practice.

REFERENCES


