Model-Driven Behavior Specification for Robotic Teams

Alexandros Paraschos*
IAS Lab
TU-Darmstadt
Darmstadt, 64287, Germany
paraschos@ias.tu-darmstadt.de

Nikolaos I. Spanoudakis
Department of Sciences
Technical University of Crete
Chania, 73100, Greece
nikos@science.tuc.gr

Michail G. Lagoudakis
Department of ECE
Technical University of Crete
Chania, 73100, Greece
lagoudakis@ece.tuc.gr

ABSTRACT

Modern model-driven engineering and Agent-Oriented Software Engineering (AOSE) methods are rarely utilized in developing robotic software. In this paper, we show how a Model-Driven AOSE methodology can be used for specifying the behavior of multi-robot teams. Specifically, the Agent Systems Engineering Methodology (ASEME) was used for developing the software that realizes the behavior of a physical robot team competing in the Standard Platform League of the RoboCup competition (the robot soccer world cup). The team consists of four humanoid robots, which play soccer autonomously in real time utilizing the on-board sensing, processing, and actuating capabilities, while communicating and coordinating with each other in order to achieve their common goal of winning the game. Our work focuses on the challenges of coordinating the base functionalities (object recognition, localization, motion skills) within each robot (intra-agent control) and coordinating the activities of the robots towards a desired team behavior (inter-agent control). We discuss the difficulties we faced and present the solutions we gave to a number of practical issues, which, in our view, are inherent in applying any AOSE methodology to robotics. We demonstrate the added value of using an AOSE methodology in the development of robotic systems, as ASEME allowed for a platform-independent team behavior specification, automated a large part of the code generation process, and reduced the total development time.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies*; I.2.9 [**Artificial Intelligence**]: Robotics—*Commercial robots and applications*

General Terms

Design

Keywords

Agent-Oriented Software Engineering, Robotic Software Development, Intra-Agent Control, Model-Driven Engineering

*Work performed while at the Technical University of Crete.

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

1. INTRODUCTION

The Model-Driven Engineering (MDE) paradigm has gained popularity among software developers and a number of methodologies, models, and tools have been developed to facilitate task decomposition, enable software reusability, minimize coding mistakes, and allow for inexpensive software maintenance. Even though some of this technology has been extended to cover the needs of agent-oriented software development, it is rarely exploited to address the needs of robotic software. Indeed, the real-time constraints and the concurrency of device operation in robotics typically impose a low-level of coding, whereas the potential of programming high-level behaviors that exploit the lower-level functionalities at a meta-level of coding remains largely unexplored.

Agile processes address several modern software development needs, such as the need for coping with continuously changing requirements, the need for continuous evaluation, and the need for less bureaucracy related to the extensive production of models that few people (only the developers) can read [8]. Thus, agile processes are very useful for projects, such as the development of a RoboCup team, whereby an autonomous robot team competes against another and the behavior of the robots may need to change between games to cope with skilled opponents.

Given that robots capture the inherent properties of agents (autonomy, social ability, reactivity, proactiveness), in this paper, we show how an Agent-Oriented Software Engineering (AOSE) methodology can be used for specifying the behavior of multi-robot teams encompassing the model-driven and agile characteristics described above. More specifically, we focus on the Agent Systems Engineering Methodology (ASEME) [21] and the domain of robotic soccer. According to Schlegel et al. [17], software engineering for robotics is different in some important aspects from software engineering for other, even related, areas, such as distributed and real-time systems. Thus, in our work we had to adapt the ASEME process to accommodate the needs of robotic software development. In this context, we defined a generic transformation tool (IAC2Monas) for instantiating the statechart models of ASEME (the platform-independent models [11]) to our Monas robot software architecture [14] for integration with implemented functionalities and execution on our robots; this coupling provided automatic code generation and execution on a generic multi-threaded statechart engine along with the Monas software modules. As a result, instead of specifying complex team behavior using hundreds of lines of conventional code, the developer can now accomplish this task using an intuitive graphical representation

with advanced control modes. Our work demonstrates the added value of using the ASEME methodology in robotics, as it allowed for a platform-independent team behavior specification, automated a large part of the code generation process, eliminated common coding mistakes, and reduced the total development time.

2. BACKGROUND AND MOTIVATION

It is common practice for roboticists to specify a robot's behavior using conventional procedural code, whereby a complex arrangement of conditional statements determines what the robot is supposed to do in each condition. A higher-level practice is to specify a robot's behavior using the formalism of Finite State Automata (FSA) whose graphical representation with nodes (states) and edges (transitions) offers a more intuitive way of synthesizing the desired behavior. However, as robots become more complex and employ the computing power of modern processors, their programming also becomes more demanding, requiring concurrent and threaded code to support efficient implementations of advanced operations, such as machine learning and signal processing algorithms. Thus, there is a clear need for modern software engineering methods in developing robotic software.

Statecharts [6], a formal model familiar to software developers, have been widely used for specifying agent plans, even in the RoboCup domain [12, 13]. Murray [12], in particular, proposes the use of extended statecharts (with synch states for synchronizing the actions of different agents) for defining the behavior of RoboCup simulation players. This work is also supported by an editing tool (StatEdit). Both proposals [12, 13] support semi-automatic code generation for Robolog, a robot programming language based on Prolog. These approaches have been used only in RoboCup simulation leagues and it is not clear how they could be adapted for use on real robots. In that case, base functionalities, such as perception and locomotion, which are provided freely in the simulation leagues, will have to be inserted into the statechart. It is not straightforward how this can be done, when a procedural programming language, such as Python or C++, is used for implementing these functionalities.

Recent developments in Multi-Agent Systems (MAS) have demonstrated that high-level approaches, such as the Extensible Agent Behavior Specification Language (Xabsl) [15] and Petri Net Plans (PNPs) [25], can be utilized for the behavioral modeling of robots. Despite their different formal models, hierarchical FSAs for Xabsl and Petri Nets for PNPs, both approaches offer hierarchical decomposition of complex behaviors, concurrent action support within their formalism, and multi-robot coordination. Although, PNPs have a more compact representation than FSAs, they still require more semantics than statecharts. Integration with state-of-the-art frameworks (B-Human [16] for Xabsl and OpenRDK [1] for PNPs) provides a threaded, low-latency environment for efficient runtime execution. Analysis and validation of the designed models can be done using standard tools, due to the use of formal and widely-used representations. Both approaches have been employed successfully in the RoboCup competition. However, both of them model only behavioral, but not functional, aspects of the system. Moreover, inter-agent coordination protocols cannot be integrated directly into their formal models.

In their work, De Loach et al. [2] apply the Multi-agent Systems Engineering (MaSE) methodology for designing teams of cooperating robots. They use a top-down approach, starting from system goals and gradually refining them to simpler goals. They use sequence diagrams for designing interaction protocols and independent instances of finite state machines (called concurrent task diagrams) for designing the behavior of each identified agent role. However, their approach is quite limiting, as it allows only for bilateral conversations, thus favoring centralized coordination schemes. Moreover, the lack of hierarchical structure in the agent plans leads to flat, large, and complex representations.

Other authors, such as Gascuena and Fernandez-Caballero [5], used the Prometheus methodology [23] to specify the behavior of a robot. Prometheus provides specific diagrams for depicting the agent roles, their resources, and exchanged messages with other roles. It uses AUML Agent Interaction Protocol (AIP) diagrams (extended UML[1] sequence diagrams) for specifying agent interactions. However, the seven different types of diagrams they propose are always constructed manually anew. The support for implementation, testing, and debugging of Prometheus models is limited and available only for the JACK agent platform. Finally, they do not address the multi-tasking issue on a single robot, but rather identify each component/task as a distinct agent. Nevertheless, in practice a lot of information coming from sensors and other modules accomplishing specific tasks need to be processed concurrently and the timing between these tasks is critical. Finally, AUML agent coordination protocols are not integrated seamlessly in agent plans.

A method coming from the MDE community is presented by Schlegel et al. [17]. The authors argue for switching the traditional code-driven robotic software development to a model-driven one. They define strict interfaces for wrapping existing components and then utilize a statechart-based approach for specifying the behavior of robots. Then, they use MDE techniques for transforming the platform-independent model they define to executable code. Their approach is a significant step towards model-based software engineering for robots, lacking mostly in the multi-agent aspect, as there is no catering for agent interaction protocols definition.

The Agent Systems Engineering Methodology (ASEME) [21] fills this particular gap. ASEME supports a modular agent design approach and introduces the concepts of intra- and inter- agent control. The former defines the agent's behavior by coordinating the different modules that implement its own capabilities, while the latter defines the protocols that govern the coordination of the society of the agents. ASEME applies an MDE approach to multi-agent systems development, so that the models of a previous development phase can be transformed to models of the next phase. The transition from one phase to another is assisted by automatic model transformation leading from requirements to computer programs. The ASEME platform-independent model, which is the output of the design phase, is a statechart that can be instantiated in a number of platforms using existing Computer-Aided System Engineering (CASE) tools.

ASEME specifies three levels of abstraction for each phase of the software development process. The first is the *societal level*, in which the whole multi-agent system functionality is modeled. Then, the *agent level* zooms on each member of the society, i.e. the individual agent. Finally, the details that compose each of the agent's parts are defined in the

[1]The Unified Modeling Language (UML) is a standardized object-oriented modeling language: www.uml.org

Development Phase	Levels of Abstraction		
	Society Level	Agent Level	Capability Level
Requirements Analysis *AMOLA Models*	Actors `System Actors Goals (SAG): Actors`	Goals `SAG: Goals`	Requirements `SAG: Requirements per goal`
Analysis *AMOLA Models*	Roles and Protocols `System Use Cases (SUC), Agent Interaction Protocols (AIP)`	Capabilities `SUC, System Roles Model (SRM)`	Functionalities `SRM: Activities and Functionalities`
Design *AMOLA Models*	Society Control `Inter-Agent Control (EAC)`	Agent Control `Intra-Agent Control (IAC)`	Components
Implementation	Platform management code	Agent code	Capabilities code
Verification	Protocols testing	Agent testing	Component testing
Optimization	Number of instantiated agents	Agent resources	Code optimization

Figure 1: ASEME phases and AMOLA products.

capability level. The concept of *capability* is defined as the ability of an agent to achieve specific tasks that require the use of one or more *functionalities*. The latter refers to the technical solution(s) to a given class of tasks. Moreover, capabilities are decomposed to simple *activities*, each of which corresponds to exactly one functionality. Thus, an activity corresponds to the instantiation of a specific technique for dealing with a particular task (a unique characteristic compared to the other statechart-based approaches). ASEME is mainly concerned with the first two abstraction levels, assuming that development in the capability level can be achieved using classical (or even technology-specific) software engineering techniques.

In Figure 1, the ASEME phases, the different levels of abstraction, and the models related to each one of them are presented. ASEME uses the models of the Agent Modeling Language (AMOLA) [19]. The AMOLA metamodels have been formally defined using the Eclipse Modeling Framework of the Eclipse Modeling Project[2]. Eclipse technology has been employed for developing model transformations and graphical editing tools for both models and processes[3].

3. ROBOCUP, NAO, AND SPL

In its short history, the RoboCup competition [10] (robot soccer world cup) has grown to a well-established annual event bringing together the best robotics researchers internationally. To succeed in playing soccer autonomously, the core problems of artificial intelligence and robotics (perception, cognition, action, coordination) must be addressed simultaneously under real-time constraints. The proposed solutions are tested through soccer games in various leagues. A key aspect of most RoboCup leagues is the multi-agent environment. The robots in each team cannot simply act as individuals; they must focus on teamwork in order to cope effectively with an unknown opponent team and such teamwork requires coordination.

The Standard Platform League (SPL)[4] is among the most popular leagues, featuring four humanoid Aldebaran Nao robot players in each team. The Nao is a 58cm, 4.3Kg humanoid robot developed by Aldebaran Robotics in Paris, France. It is equipped with an x86 AMD Geode processor at 500 MHz, 256 MB SDRAM, 2 GB flash disk, two color cameras, two ultrasound sensors, an inertial unit (2 gyroscopes and 3 accelerometers), an array of force sensitive resistors on each foot, encoders on all servos, and a total of 21 degrees of freedom (4 in each arm, 5 in each leg, 2 in the head, and 1 in the pelvis). SPL games take place in a $4m \times 6m$ field marked with white lines on a green carpet with two colored (skyblue and yellow) goals. Each game consists of two 10-minutes halves and teams switch sides at halftime. There are several rules enforced by human referees during the game.

In SPL, all teams use the same robotic hardware and differ only in terms of their software. Therefore, research efforts focus on developing more efficient algorithms and techniques for visual perception, active localization, omni-directional motion, skill learning, individual robot behavior specification, and team coordination strategies. This paper focuses on the last two challenges.

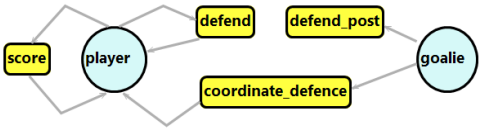

Figure 2: The System Actors Goals (SAG) model.

4. SOFTWARE ENGINEERING PROCESS

In this section, we describe the proposed model-based agent-oriented software engineering process in a step-by-step manner following the principles of AMOLA and using the RoboCup domain as our case problem.

4.1 Requirements Analysis Phase

In the requirements analysis phase, AMOLA defines the *System Actors and Goals* (SAG) model, containing the main actors in the system and their goals. For the Robocup domain, the actors are the players and the goalie of the team (see Figure 2). The player aims to score and defend, both goals depending also on the other players. The goalie aims to defend its post (individual goal), but also to coordinate the defense (depending on the players).

4.2 Analysis Phase

In the analysis phase, AMOLA proposes the *System Use Cases* (SUC) model, where the different activities that realize the agent capabilities are defined in a top-down decomposition process, the *Agent Interaction Protocol* (AIP) model, which specifies the coordination between agents, and, finally, the *System Roles Model* (SRM), through which the previously-defined activities are integrated to define the dynamic behavior of the roles of the agents. Initially, the SAG model from the previous phase is transformed to the SUC model (see Figure 3). The SAG goals are transformed to SUC use cases and the SAG actors to SUC roles. The modeler optionally adds roles and ≪includes≫ use cases. The goals transformed to use cases form the roles' capabilities.

In Figure 3, the *score* capability has been decomposed to simpler use cases using the ≪includes≫ relation. Thus, for scoring, the player can kick the ball towards the goal (*kick ball* use case) or participate in the coordinated *attack* use

[2]The Eclipse Modeling Project provides a unified set of modeling frameworks, tooling, and standards implementations: www.eclipse.org/modeling
[3]The AMOLA metamodels and ASEME transformation tools are freely available from: www.amcl.tuc.gr/aseme
[4]SPL Web Site: www.tzi.de/spl

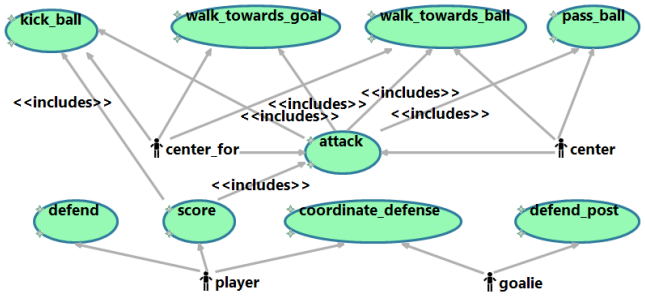

Figure 3: The System Use Cases (SUC) model.

Table 1: The AIP model for the *attack* protocol.

Participants	*center*	*center_for*
Engagement Rules	no robot has control of the ball ***and*** *center* is the robot closest to the ball ***and*** *center_for* is the robot farthest from the ball	
Outcomes	the *center_for* shoots to goal ***or*** an opponent takes control of the ball ***or*** the ball goes out of bounds	
Process	WalkTowardsBall. [passBall]	WalkTowardsGoal. [WalkTowardsBall. [kickBall]]

case. Note that the *attack* use case has also been associated with two new roles, the *center* and *center for*, which are connected to new use cases, decomposing further the *attack* use case. Thus, the *center* walks towards the ball and passes it to the *center for*, while the *center for* walks towards the opponent's goal post to receive the pass, then walks towards the ball and kicks it. The SUC model does not specify the order in which use cases are employed by the roles. The AIP and SRM models fill exactly this gap; the former specifies how to coordinate the cooperative roles' activities and the latter how to coordinate the individual role's activities.

Thus, as the human soccer team coach sketches the players' movements for a coordinated team action in real soccer, the robotic team coach uses the AIP model to sketch the robotic team's coordinated activities. The AIP model lists the participants along with the preconditions and postconditions in free text format. The process of each participant, however, is described formally using liveness formulas. Liveness formulas connect activities using the Gaia operators [24]. Briefly, $A.B$ means that activity B is executed after activity A, A^{ω} means that A is executed continuously (it restarts as soon as it finishes), $A|B$ means that either A or B is executed, $A||B$ means that A and B are executed in parallel, and $[A]$ means that A is optional.

The *attack* protocol with two participant roles (i.e. *center* and *center for*) is presented in Table 1. The rule for engaging in these roles is depicted in the second row, followed by the expected outcomes in the third row, and the process for each role defined using liveness formulas in the fourth row. In particular, the player closer to the ball becomes the *center* and the other players become *center for*. The *center* is expected to approach the ball and pass it to a *center for* who, in turn, is expected to be near the opponent's goal post to receive the pass and shoot to score. The protocol may terminate early, if an opponent takes control of the ball or the ball goes out of bounds.

The System Roles Model (SRM) defines each concrete role (corresponding to a SAG actor) by specifying the protocols in which the role participates and liveness formulas defining its dynamic behavior including the relevant process parts of the AIP model. Figure 4 shows the SRM for the player role. Note that this role can participate in the *attack* protocol either as a *center* or as a *center for*. While in the AIP model process part the activities are abstractly defined, in the SRM liveness formula all activities are connected to specific functionalities of the robot. The identified functionalities in our case are the following:

- *Sensors*, for collecting and filtering all data from the robot sensors (accelerometers, buttons, bumpers, etc.),
- *RobotController*, for listening to external information about the game state coming from the game controller,
- *LedHandler*, for managing the operation of the colored LEDs of the robot (eyes, ears, buttons),
- *MotionController*, for scheduling and executing motion commands (walk, kick, stand-up, special actions, etc.),
- *Vision*, for detecting the ball and the goals in the camera image and estimating their distance and bearing,
- *Localization*, for estimating the position and orientation of the robot and the ball in the field,
- *ObstacleAvoidance*, for planning obstacle-free paths in a local polar map using ultrasonic range measurements,
- *HeadHandler*, for managing the movements of the robot head and, thus, the camera (scanning, tracking, etc.).

The self-explained activities named *Stand, CalibrateCamera, CheckForBallObservation, ScanForBall, TrackBall, WalkTowardsBall, KickBall, PassBall, WalkTowardsOpponentGoal* are provided either directly by the above functionalities or by combining information coming from some of them (for example, *Vision* with *ObstacleAvoidance* and *MotionController* to realize *WalkTowardsBall*). Similarly to the work of Schlegel et al. [17], all functionalities are wrapped with standard interfaces. In our Monas architecture they are defined through XML configuration files.

4.3 Design Phase

In the design phase, AMOLA defines the *intEr-Agent Control* (EAC) model and the *Intra-Agent Control* (IAC) model, which are based on the formalism of *statecharts* and define both the functional and behavioral aspects [6] of the multi-agent system. The ASEME SRM2IAC tool is used to transform the process formulas of an AIP model protocol to an EAC model and the liveness formulas of an SRM role to an IAC model. The EAC and IAC models are statecharts, where the developer can insert events, conditions, and actions in the transition expressions, thus controlling each role's process either for satisfying the needs of a protocol (in the EAC model) or for coordinating the agent's capabilities (in the IAC model). There are six types of *states* in a statechart [6]:

- *start*, showing where execution starts
- *end*, showing where execution stops
- *or*, having sub-states (of any kind) related by "exclusive-or", i.e. only one is executed at any given time

Role: player
Protocols: attack: center, attack: center_for
Liveness:
player = Sensors$^{\omega}$ || RobotController$^{\omega}$ || LedHandler$^{\omega}$ || MotionController$^{\omega}$ || (initialize . activate)
initialize = Stand . CalibrateCamera
activate = Vision$^{\omega}$ || Localization$^{\omega}$ || ObstacleAvoidance$^{\omega}$ || HeadHandler$^{\omega}$ || decision$^{\omega}$
decision = CheckForBallObservation . (ScanForBall | action)
action = TrackBall || (WalkTowardsBall | KickBall | center | center_for)
center = WalkTowardsBall . [PassBall]
center_for = WalkTowardsOpponentGoal . [WalkTowardsBall . [KickBall]]

Figure 4: The SRM model for the *player* (participating as *center* or *center_for* in the *attack* protocol).

- *and*, having *or*-states as sub-states related by "and", i.e. all of them are executed concurrently
- *basic*, having no sub-states, representing an activity
- *condition*, offering only conditional transitions (also known as *OR-connector* or *conditional transition*)

The state at the highest level (the one with no parent state) is called the *root*. Each *transition* from one state (source) to another (target) is labeled by an *expression*, whose general syntax obeys the pattern `e[c]/a`, where e is the event that triggers the transition; c is a condition that must be satisfied for the transition to be taken, when event e occurs; and a is an action that takes place, when the transition is taken. All elements of the transition expression are optional. The *scope* of a transition is the lowest level *or*-state, which is a common ancestor of both the source and target states. When a transition occurs all states in its scope are exited and the target states are entered.

Having defined the statechart, as it is used in AMOLA [20], it is now possible to proceed to the definition of the inter-agent control (EAC) model. The EAC is a statechart that contains an initial (*start*) state, an *and*-state named after the protocol, and a final (*end*) state. The *and*-state contains as many *or*-states as the protocol roles, named after these roles. One transition connects the *start*-state to the *and*-state and another transition the *and*-state to the *end*-state. Transitions can be triggered by a timeout event or by the completion of the executed state activity. Thus, for the *attack* protocol, since the two participating roles operate simultaneously in parallel, the SRM2IAC tool transforms the following formula along with the process part of the AIP model protocol into a statechart:

```
action = center || center_for
```

The result of the automatic model transformation is depicted graphically in the form of an ordered rooted tree (Figure 5) that defines the statechart. In the fragment of the statechart shown in Figure 5, the reader can see the *center* role. The nodes of the tree (rounded rectangles) define the states (gray lines point to parent nodes in the tree structure). Each node includes the state name and the state type. Labeling the nodes properly helps the modeler identify the position of a node within the tree. For example, the $A.1$ label means that the node labeled with A is the parent of the node labeled with $A.1$. Nodes without a name coming from the formula, e.g. *start* nodes, are named after their label. Each *or*-state includes a *start*-state and, usually an *end*-state (except in the cases where a state loops infinitely to its self, thus no end state is needed) to note where execution starts and where it stops. The modeler can now define transition expressions for all the transitions (depicted with red/dark lines) using the grammar defined in Figure 6 in EBNF format [9]. In Figure 5, the modeler has just defined a condition for the transition having as source the *condition*-state at the bottom of the figure and target the *basic* state to its right named *passBall*. It checks if the ball is within an angle of 10 degrees from the current orientation of the robot's torso and within a distance of 6 cm from the center of its feet, i.e. the robot can kick the ball to pass it.

The intra-agent control (IAC) model is also initiated by the SRM2IAC tool for each role. Again, the modeler must define the transition expressions and the variables contained in these expressions. By convention, the user should not define new transitions, although the statechart formalism allows for transitions between any two states, because the resulting IAC model will no longer represent the process defined by the liveness formulas. Moreover, the modeler must ensure that the branches of the statechart that come from EAC models are transferred unchanged in the statechart (their transition expressions must not change). Only in this way are the protocols guaranteed to be executed as planned. The RoboCup *player*'s statechart (IAC model) is shown in Figure 7, with the zoom window focusing on the *center* part of the *attack* protocol. Notice that the example condition introduced earlier appears unchanged at the correct place.

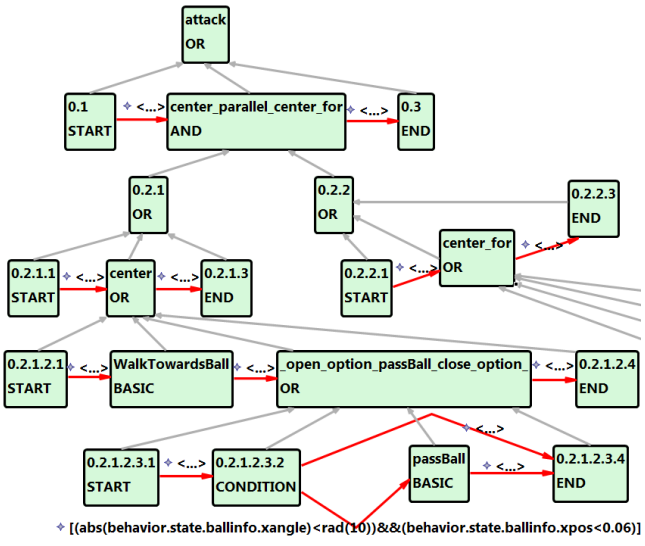

Figure 5: The EAC model for the *attack* protocol.

transitionExpression = [event] ['[' condition ']'] ['/' actions]
event = string
condition = expr | expr (compOp | logicOp) condition
| '(' condition ')' | notOp condition
actions = action | action ';' actions
action = expr | variable '=' expr | '`read_messages`'
| '`write_messages`' '.' topic '.' commType '.' msgType
expr = varVal | function '(' args ')'
function = string
args = varVal | varVal ',' args
varVal = variable | value
value = constant | stringLiteral
compOp = '<' | '<=' | '>' | '>=' | '==' | '!='
logicOp = '&&' | '||'
notOp = '!'
variable = host '.' topic '.' commType '.' msgType '.' member
| topic '.' commType '.' msgType '.' member
commType = '`signal`' | '`state`' | '`data`'
host = string
topic = string
msgType = string
member = string
stringLiteral = '"' string '"'
string = letter (letter | digit)*
letter = 'A'..'Z' | 'a'..'z' | '_'
digit = '0'..'9'
constant = ['+' | '-'] digit digit* ['.' digit digit*]

Figure 6: The transition expression grammar.

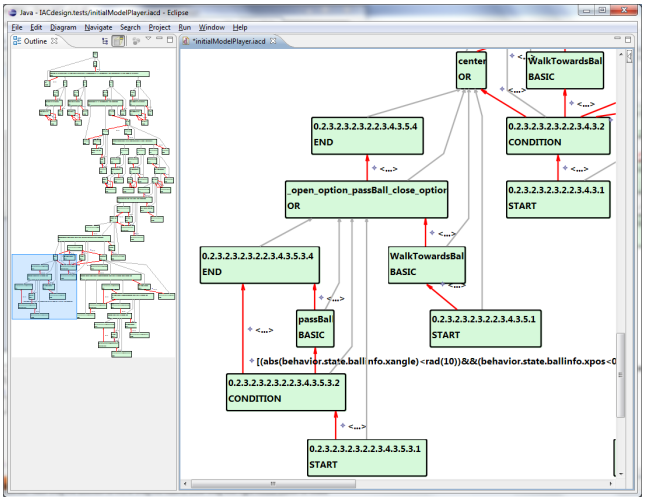

Figure 7: The IAC model for the RoboCup *player*.

4.4 Implementation Phase

To facilitate the code generation process, we built the IAC2Monas transformation tool [14], which translates the IAC model automatically to C++ source code adhering to the Monas architecture. IAC2Monas is a model-to-text (M2T) transformation tool. Thus, the platform-independent model (IAC) is transformed to a platform-specific model (code), which is subsequently cross-compiled to produce the executable for the robot. In order to build this tool we used the Xpand language offered by the Eclipse Modeling Project following the practice proposed by ASEME for the IAC2JADE transformation [21]. Xpand is used to define the templates for the required C++ classes, which are instantiated using information from the IAC metamodel elements, and is integrated with Xtend to handle the instantiation of complex expressions. An extract from the automatically generated statechart definition file (by the IAC2Monas tool) is presented in Figure 8. The reader can get an idea of how *or*-states, *basic* states, and *condition* states are defined in C++. The generation of a node (state) requires a name and the parent node as arguments, with the exception of the `Statechart` class, which appears only at the top of the hierarchy.

```
#include "AttackerPlan.h"

namespace { StatechartRegistrar<AttackerPlan>::
    Type temp("AttackerPlan"); }

AttackerPlan::AttackerPlan(Communicator* com) {

  _stc = new Statechart ( "Player", com );
  Statechart* Node0 = _stc;
  _states.push_back( Node0 );

  OrState* Node0212 = new OrState
    ( "RobotController_forever_", Node021 );
  _states.push_back( Node0212 );

  IActivity* ActivI02122 = ActivityFactory::
    Instance()->CreateObject( "RobotController" );
  _activities.push_back( ActivI02122 );
  BasicState* Node02122 = new BasicState
    ("RobotController",Node0212,ActivI02122);
  _states.push_back( Node02122 );

  ICondition* CondI02TO03 = new TrCond02TO03;
  _conditions.push_back( CondI02TO03 );

  _transitions.push_back( new TransitionSegment
    <State,State>(Node02, Node03, CondI02TO03) );
```

Figure 8: An extract of the auto-generated code.

To support this phase, we had to develop two important software libraries: (a) the communication framework both for inter-agent (i.e. between different agents) and intra-agent (i.e. between activities on a single agent) communication and (b) the statechart engine for executing statecharts.

Our communication framework [22] is based on the publish/subscribe messaging pattern [4] and supports multiple ways of communication, including point-to-point and multicast connections. The information that needs to be communicated between nodes (agents or activities) is formed as messages, tagged with appropriate topics, and relayed through a message queue for delivery. We used Google Protocol Buffers[5] to facilitate the serialization of data and the structural definition of the messages. Additionally, the blackboard paradigm [7] is utilized to provide efficient access to shared information stored locally at each node and is extended to support history queries and a mechanism that controls the information updates.

Our statechart engine [14] was built on top of existing open-source projects. Its main distinguishing characteristic

[5]Protocol Buffers are Google's language- and platform-independent, extensible mechanism for serializing structured data. http://code.google.com/apis/protocolbuffers

from other frameworks, e.g. UML and Boost[6], is the multi-threaded statechart execution that provides the required concurrency and meets the real-time requirements of the activities on each robot.

These libraries are linked to the automatically generated code at compilation time. A blackboard is instantiated (a) for each agent and (b) for each *or*-state that is a substate of an *and*-state. This way, the shared information is distributed, not only among network nodes (agents), but also among the concurrently executed parts of each agent. The latter allows for a significant increase in run-time performance, as it eliminates starvation and producer/consumer problems. The presented models do not include explicit communication activities, because coordination occurs through the `read_messages` and `write_messages` actions. These actions allow the use of shared information from the blackboards as variables in the transition expressions (see the grammar rules for *action* and *variable* in Figure 6).

5. EMPIRICAL EVALUATION

To empirically evaluate our approach, we compared it to our previous practice, i.e. using Aldebaran's middleware for Nao (NaoQi) which provides, besides the API, a platform for modular software development and a thread-safe mechanism for communication. As a proof of concept, a student familiar with both development methods was asked to develop the same behavior for the RoboCup team. The empirical results in terms of some development performance metrics are shown in Table 2. *Run-Time Performance* refers to the system load average over 5 minutes of execution. Values greater than one indicate CPU overload. The NaoQi-based agent had inferior performance, due to system overload caused by the vast amount of exchanged information between the modules. Writing C++ code had its impact on *Total Development Time*, which was considerably higher in NaoQi, whereas the statechart graphical editing tool allowed for quicker development. The ASEME-based agent consisted of more *Lines of Source Code*, however the vast majority of them were *Auto-Generated*. The NaoQi-based agent required the maintenance of several *State Variables* to indicate the current state of the agent, whereas in the ASEME-based agent it was represented explicitly in the statechart. The advantages of the ASEME approach were also reflected on the *Debugging Phase*, where NaoQi exhibited increased *Debugging Time* with a larger *Number of Bugs*.

Our experience from our RoboCup team[7] indicates that the model-based ASEME methodology with the automated transformation tools (SRM2IAC, IAC2Monas) is advantageous over our previous practice. New students familiarize themselves with robot team behavior specification in significantly less time. New ideas on team behavior can be quickly prototyped and existing behaviors can be easily explained. ASEME proves itself in behavior update or modification, which rarely involves the introduction of new functionality and typically amounts to changes in the agent's process and team protocols. This feature turned out to be extremely useful in the RoboCup 2011 competition, where we were able to modify our team behavior even at half-times or during timeouts. ASEME was one of this year's innovations that contributed to a significantly better team performance in the SPL games of RoboCup 2011 compared to RoboCup 2010 and led to winning the second place in the SPL Open Challenge Competition. The reader may watch our robot players in action at: www.kouretes.gr/aamas2012.mp4.

Table 2: Comparison of development methods.

Metric / Concept	NaoQi	ASEME
Run-Time Performance (load)	1.3	0.8
Development Phase		
Total Development Time	8 hours	5 hours
Lines of Source Code	490	826
Auto-Generated Lines	N/A	805
Number of State Variables	18	N/A
Debugging Phase		
Total Debugging Time	12 hours	5 hours
Number of Bugs	16	4

6. DISCUSSION

We faced several challenges in applying the ASEME AOSE methodology to multi-robot behavior specification, which, in our view, are inherent in this process and every AOSE practitioner will face in a similar endeavor.

Firstly, most AOSE methodologies take for granted that the agents communicate and coordinate through message passing. This does not always hold in robotic applications, where coordination can be based on diverse communication means, such as inter-agent messages, blackboards, or even sensory information. Even though ASEME defines interaction protocols based on the activities of the participants, the original statechart transition expression language [18] assumed that a FIPA[8]-like communication language would be used for message exchange. This is not true for most multi-robot applications, where the real-time constraints forbid the use of Java, on which the most successful agent platforms and those that comply to FIPA are based. The use of the publish/subscribe communication framework and the blackboard paradigm for local storage, forced us to modify the transition expression language.

Additionally, the transformation of the platform-independent model, typically the output of the design phase, to the platform-specific model is not straightforward. The computational limitations of robotic platforms make existing model-to-text transformations of the AOSE methodologies obsolete and new transformations need to be defined. Towards this end, it is very important that the AOSE methodology delivers a platform-independent model with a clear and compact meta-model. Statecharts offer more compact semantics than PNPs [25], behavior trees [3], and hierarchical FSAs [15], and additionally capture both the functional and behavioral aspects of the system.

Finally, behavior specification is not a trivial task. The development of the simple player, which served as our running example, led to a statechart with 99 states in a hierarchy with a depth of 17 (Figure 7). This shows the added value of starting with the early ASEME models and particularly using the automatic transformation of liveness formulas to a statechart, as opposed to starting the design directly with a statechart CASE tool, such as StatEdit [12], or using a flat statechart model, such as the plan diagrams of MaSE [2].

[6]Boost is a set of free peer-reviewed portable C++ libraries: www.boost.org

[7]TUC RoboCup team Kouretes: www.kouretes.gr

[8]Foundation for Intelligent Physical Agents www.fipa.org

7. CONCLUSION

In this paper we showed how the ASEME model-driven AOSE methodology can be extended for multi-robot behavior specification. The modeler is assisted by the existing graphical and model transformation tools of ASEME and by the IAC2Monas transformation tool that allows the automated code generation for the defined behavior coupled with a generic multi-threaded statechart engine and a blackboard publish/subscribe messaging system.

We discussed the challenges we faced, to share our experience with any AOSE practitioner aiming to move to robotic agents' development. The solutions proposed in our work can serve as a first guide on how to go about addressing such challenges.

Our future work lies in enhancing the code generation tool with tight checking functionalities for minimizing user errors, e.g. for semantic validation of the transition expressions. Moreover, we plan to work on making the graphical editing tools more efficient and more flexible in visualizing and manipulating statecharts. The collection of our tools for ASEME-based robot software development, including the entire code of our RoboCup team, has been released to the community through www.kouretes.gr/aseme.

8. ACKNOWLEDGMENTS

The authors would like to thank Mr Antonis Argyriou, Mrs Aggeliki Topalidou-Kyniazopoulou, Mrs Shabana Shaikh, and all members of the Kouretes team for their valuable assistance. Also, Chipita S.A.–Molto for sponsoring our team.

9. REFERENCES

[1] D. Calisi, A. Censi, L. Iocchi, and D. Nardi. OpenRDK: A modular framework for robotic software development. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1872–1877, September 2008.

[2] S. DeLoach, E. T. Matson, and Y. Li. Applying agent oriented software engineering to cooperative robotics. In *Proceedings of the 15th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 391–396. AAAI Press, May 2002.

[3] R. Dromey. From requirements to design: Formalizing the key steps. In *Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM)*, pages 2–11, September 2003.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[5] J. M. Gascuena and A. Fernandez-Caballero. Agent-oriented modeling and development of a person-following mobile robot. *Expert Systems with Applications*, 38(4):4280–4290, 2011.

[6] D. Harel and A. Naamad. The Statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5:293–333, 1996.

[7] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[8] J. Highsmith and M. Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[9] ISO/IEC. Extended Backus-Naur form (EBNF). 14977, 1996.

[10] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. Robocup: A challenge problem for AI. *AI Magazine*, 18(1):73–85, 1997.

[11] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[12] J. Murray. Specifying agent behaviors with UML statecharts and StatEdit. In *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] O. Obst. Specifying rational agents with statecharts and utility functions. In *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Computer Science*. Springer, 2002.

[14] A. Paraschos. Monas: A flexible software architecture for robotic agents. Diploma thesis, Technical University of Crete, Greece, 2010.

[15] M. Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Technische Universität Darmstadt, Germany, 2009.

[16] T. Röfer et al. B-Human team report and code release, 2009. Only available online: www.b-human.de.

[17] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, pages 1–8, June 2009.

[18] N. Spanoudakis. *The Agent Systems Engineering Methodology (ASEME)*. PhD thesis, Paris Descartes University, France, 2009.

[19] N. I. Spanoudakis and P. Moraitis. The agent modeling language (AMOLA). In *Proceedings of the 13th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, volume 5253 of *Lecture Notes in Computer Science*, pages 32–44. Springer, September 2008.

[20] N. I. Spanoudakis and P. Moraitis. Gaia agents implementation through models transformation. In *Proceedings of the 12th International Conference on Principles of Practice in Multi-Agent Systems (PRIMA)*, volume 5925 of *Lecture Notes in Computer Science*, pages 127–142. Springer, December 2009.

[21] N. I. Spanoudakis and P. Moraitis. Using ASEME methodology for model-driven agent systems development. In *Agent-Oriented Software Engineering XI, Revised Selected Papers of the 11th International Workshop AOSE 2010*, volume 6788 of *Lecture Notes in Computer Science*, pages 106–127. Springer, 2011.

[22] E. Vazaios. Narukom: A distributed, cross-platform, transparent communication framework for robotic teams. Diploma thesis, Technical University of Crete, Greece, 2010.

[23] M. Winikoff and L. Padgham. *Developing Intelligent Agent Systems: A Practical Guide*. Halsted Press, 2004.

[24] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[25] V. Ziparo, L. Iocchi, P. Lima, D. Nardi, and P. Palamara. Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 23:344–383, 2011.