A CASE Tool for Robot Behavior Development

Angeliki Topalidou-Kyniazopoulou[1], Nikolaos I. Spanoudakis[2], and Michail G. Lagoudakis[1]

[1] Department of ECE, Technical University of Crete, 73100, Chania, Greece
atop87@gmail.com, lagoudakis@intelligence.tuc.gr
[2] Department of Sciences, Technical University of Crete, 73100, Chania, Greece
nikos@science.tuc.gr

Abstract. The development of high-level behavior for autonomous robots is a time-consuming task even for experts. This paper presents a Computer-Aided Software Engineering (CASE) tool, named Kouretes Statechart Editor (KSE), which enables the developer to easily specify a desired robot behavior as a statechart model utilizing a variety of base robot functionalities (vision, localization, locomotion, motion skills, communication). A statechart is a compact platform-independent formal model used widely in software engineering for designing software systems. KSE adopts the Agent Systems Engineering Methodology (ASEME) model-driven approach. Thus, KSE guides the developer through a series of design steps within a graphical environment that leads to automatic source code generation. We use KSE for developing the behavior of the Nao humanoid robots of our team Kouretes competing in the Standard Platform League of the RoboCup competition.

1 Introduction

Computer-Aided Software Engineering (CASE) tools improve productivity and quality in software development [1]. However, they are not widely used for robot behavior development, even in domains, such as RoboCup, where robot behavior needs to be frequently modified. It is quite common for a RoboCup team to find itself in a place where the code realizing the behavior of its robots needs to be urgently modified, for example during half-time because of some unexpected opponent strategy. The time constraints and the programmers' stress in such situations consist a recipe for failure. CASE tools can be really helpful in this context, as they offer ways to make behavior development and modification quicker and less error-prone.

Recent advances in Agent Oriented Software Engineering (AOSE), Model-Driven Engineering (MDE), and Domain Specific Languages (DSLs) allowed us to define a novel model-driven process for developing collaborative robot behaviors [2]. This process, however, lacked the assistance of a CASE tool that would allow the graphical editing of the behavior models. The work presented in this paper aims to fill this gap by proposing the Kouretes Statechart Editor (KSE) CASE tool, which enables the developer to easily specify a desired robot behavior as a statechart model utilizing a variety of base robot functionalities (vision, localization, locomotion, motion skills, communication). A statechart [3] is a compact

© Springer-Verlag Berlin Heidelberg 2013

platform-independent formal model used widely in software engineering for designing software systems. KSE adopts the Agent Systems Engineering Methodology (ASEME) model-driven approach [4] and assists the developer from the analysis phase to the design and code generation phases. More specifically, KSE supports (a) the automatic generation of the initial abstract statechart model using compact liveness formulas, (b) the graphical editing of the statechart model and the addition of the required transition expressions, and (c) the automatic source code generation for compilation and execution on the robot. KSE has been developed using the Eclipse Modeling Project[1] technologies and has been integrated with our Monas software architecture [5] and our Narukom communication framework [6], which provide the base functionalities. KSE is used for developing the behavior of the Aldebaran Nao humanoid robots of our team Kouretes competing in the RoboCup Standard Platform League (SPL).

In the rest of the paper, after examining the background technologies in Section 2, we present our ASEME-based robot behavior development method in Section 3 and the main features of KSE, including design and implementation choices, in Section 4. Subsequently, we present the results of a first empirical evaluation in Section 5. Finally, we discuss our findings and related work in Section 6 before concluding with Section 7.

2 Background

ASEME [4] 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 a Model-Driven Engineering (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, and is referred to as the Intra-Agent Control (IAC) model. ASEME uses the models of the Agent Modeling Language (AMOLA) [7]. The AMOLA metamodels have been formally defined using the Eclipse Modeling Framework (EMF) of the Eclipse Modeling Project. Eclipse technology has been employed for developing model transformations and graphical editing tools for both models and processes.

Our communication framework, Narukom [6], is based on the publish/subscribe messaging pattern [8] 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. Three types of messages are supported: (i) *state*, which remain in the blackboard

[1] The Eclipse Modeling Project provides a unified set of modeling frameworks, tooling, and standards implementations: `www.eclipse.org/modeling`

until they are replaced by a newer message of the same type, (ii) *signal*, which are consumed at the first read, and (iii) *data*, which are time-stamped to indicate the precise time the values they carry were acquired. We used Google Protocol Buffers[2] to facilitate the serialization of data and the structural definition of the messages. Additionally, the blackboard paradigm [9] 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 software architecture, Monas [5], provides an abstraction layer from the Nao robot and allows the synthesis of complex robot software as XML-specified Monas modules and/or statechart modules. Monas modules focus on specific functionalities (vision, motion, etc.) and each one of them is executed independently at any desired frequency completing a series of activities at each execution. Statechart modules are executed using a generic multi-threaded statechart engine [5], which is built on top of existing open-source projects, provides the required concurrency, and meets the real-time requirements of the activities on each robot. The base functionalities utilized by a statechart can be implemented as Monas modules and include the following: *Sensors*, for collecting and filtering measurements from the robot sensors; *RobotController*, for handling external signals on the game state; *MotionController*, for managing and executing robot locomotion and special actions; *Vision*, for obtaining visual object observations; *Localization*, for estimating the position of the robot and the ball in the field; and *HeadHandler*, for managing the movements of the robot head (camera).

3 ASEME-based Behavior Development

ASEME suggests a strict hierarchical decomposition of the desired robot behavior into smaller activities until a level of provided base activities is met. Each design step is supported by a formal model. These models are automatically transformed when moving from one step of the design process to the next. Briefly, the process begins with the specification of a set of liveness formulas (analysis phase) which are converted to an initial statechart model; the statechart is subsequently enriched (design phase) and is converted to source code.

Liveness formulas describe and connect the activities included in the desired behavior in a formal way, similar to regular expressions. Each formula is a rule decomposing one activity (shown on the left side) into a number of interconnected smaller activities (shown on the right side). Activities on the right side of a formula are connected using the Gaia operators [10]. Specifically, `A.B` means that activity `B` is executed after activity `A`, `A*` means that `A` is executed zero or more times, `A+` means that `A` is executed one or more times, `A~` means that `A` is executed indefinitely (it resumes as soon as it finishes), `A|B` means that either `A` or `B` is executed exclusively, `A||B` means that `A` and `B` are executed concurrently, and `[A]` means that `A` is optional. Using liveness formulas, the developer can hierarchically decompose the desired behavior into specific activities until

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

the existing base activities are reached, however without specifying the precise conditions under which individual activities are chosen.

The statecharts [3] are formal models that describe complex processes and control structures using directed graphs with nodes (states) and edges (transitions). Six types of nodes or states are allowed: *start*–states, indicating the entry of execution in a complex state, *end*–states, indicating the exit of execution from a complex state, *or*–states, indicating complex states with mutually exclusive sub-states (only one sub-state is executed at each time), *and*–states, indicating complex states with sub-states of type *or* which are executed concurrently, *basic*–states, indicating the execution of base activities, and *condition*–states, providing the ability to make conditional transitions. The state at the highest level (the only one without a parent) is called the *root*. Each transition from one state (source) to another (target) is characterized by an expression whose syntax follows the pattern e[c]/a, where e is the event triggering the transition, c is the condition that needs to be satisfied for the transition to take place when e occurs, and a is an action executed when the transition is taken. All the elements of an expression are optional. If the expression lacks an event, the condition is evaluated when the source state finishes execution.

The liveness2IAC tool [11] is used to transform the liveness formulas to statecharts and the IAC2Monas tool [5] is used to transform the statechart automatically to C++ source code adhering to the Monas architecture. With the use of the latter, the platform-independent model (statechart) is transformed to a platform-specific model (source code), which is subsequently cross-compiled to produce the executable for the robot.

4 The Kouretes Statechart Editor CASE Tool

KSE is a CASE tool designed to support all steps of ASEME-based behavior development through an intuitive graphical interface. In particular, liveness formulas are given in plain text and are automatically converted to an initial statechart model, where the designer can graphically add the appropriate transition expressions. The syntax of transition expressions is formally specified by an EBNF grammar [12]. Each statechart can be associated with a source code repository containing the base activities; in our case, a repository of Monas activities. KSE also allows the creation of statecharts from scratch (without liveness formulas) and graphical editing and modification of any existing statechart. To ensure that the designer will not produce an invalid statechart with respect to Harel's statechart language [3] and the EBNF grammar, KSE offers a validation procedure which identifies mistakes in the statechart and warns the user. The final statechart is automatically converted to source code which is integrated with the associated source code repository and is cross-compiled for execution.

4.1 KSE Example

We provide a simple example to demonstrate KSE and the behavior development process. Consider a very simple behavior, whereby a robot listening to

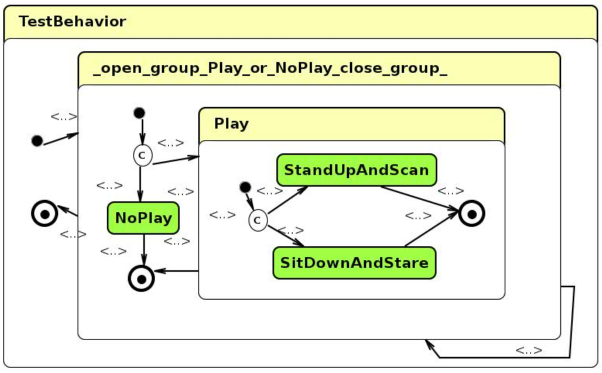


Fig. 1. KSE example: the generated statechart model from the liveness formula

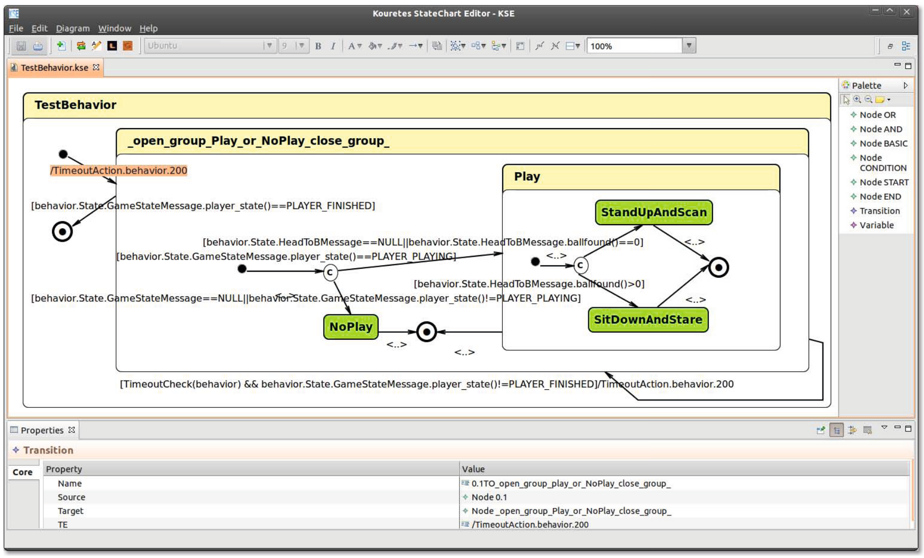
SPL's game controller executes the following actions, whenever it enters the `PLAYER_PLAYING` state: *sit down when you see the ball and track it, stand up and scan for the ball when you lose it.* In any other game state, it does nothing.

The first step in creating a behavior with KSE is to describe the behavior with liveness formulas. The two liveness formulas for this simple behavior are:

```
TestBehavior = (Play | NoPlay)+
Play = SitDownAndStare | StandUpAndScan
```

The first formula indicates that our behavior (`TestBehavior`) will choose one or more times between `Play` and `NoPlay` exclusively. `NoPlay` is a base activity, which handles game states different than `PLAYER_PLAYING`, and is not analyzed further. `Play` is refined in the second formula, which simply states that `Play` will have to choose one of the two base activities, `SitDownAndStare` or `StandUpAndScan`, exclusively. `SitDownAndStare` commands the robot to sit down and stare at the (visible) ball, whereas `StandUpAndScan` commands the robot to stand up and scan for a ball. Note that this decomposition specifies *what* activities are included in the desired behavior, but gives no information on *when* execution will switch from one activity to another.

As soon as the formula is provided to KSE, the initial statechart model is generated and the user has to associate it to a source code repository, which provides the base functionalities and in which the code of the new statechart is going to be integrated. At this point, the user can initialize the graphical representation (Figure 1) of the automatically created statechart model in order to edit the transition expressions or the activities of the *basic*-states. Note that each activity in the liveness formula has become a node/state in the statechart. The yellow-color-labeled rounded rectangles indicate *or*-states, the green-colored rounded rectangles *basic*-states, and the blue-color-labeled rounded rectangles *and*-states (none in this example). A node with a circled `c` is a *condition*-state, whereas solid black nodes correspond to *start*-states and circled black nodes to *end*-states. The model hierarchy is preserved in the graphical node enclosures.


Kouretes StateChart Editor - KSE
File Edit Diagram Window Help
Ubuntu
9
100%
TestBehavior.kse
TestBehavior
_open_group_Play_or_NoPlay_close_group_
/TimeoutAction.behavior.200
[behavior.State.GameStateMessage.player_state()==PLAYER_FINISHED]
Play
StandUpAndScan
[behavior.State.HeadToBMessage==NULL||behavior.State.HeadToBMessage.ballfound()==0]
[behavior.State.GameStateMessage.player_state()==PLAYER_PLAYING]
<..>
<..>
C
C
[behavior.State.HeadToBMessage.ballfound()>0]
[behavior.State.GameStateMessage==NULL||behavior.State.GameStateMessage.player_state()!=PLAYER_PLAYING]
<..>
<..>
NoPlay
SitDownAndStare
<..>
<..>
<..>
[TimeoutCheck(behavior) && behavior.State.GameStateMessage.player_state()!=PLAYER_FINISHED]/TimeoutAction.behavior.200
Palette
Node OR
Node AND
Node BASIC
Node CONDITION
Node START
Node END
Transition
Variable
Properties
Transition
Core

Property	Value
Name	0.1TO_open_group_play_or_NoPlay_close_group_
Source	Node 0.1
Target	Node _open_group_Play_or_NoPlay_close_group_
TE	/TimeoutAction.behavior.200

Fig. 2. KSE example: the complete statechart model with all the transition expressions

An empty transition expression (inidicated by `<..>`) implies that no event is required for triggering, the condition is evaluated to `true`, and no action is executed. Such a transition is used to indicate default execution paths, when all other non-empty transition conditions evaluate to `false`. To ensure proper execution of the statechart, the user must define the appropriate transition expressions. In our example, we have to provide six transition expressions: when to continue with `(Play | NoPlay)`, when to leave `(Play | NoPlay)`, when to choose `Play`, when to choose `NoPlay`, when to choose `SitDownAndStare`, and when to choose `StandUpAndScan`. These conditions take into account information delivered by incoming messages from existing Monas modules indicating the game state and whether the ball is visible or not. The complete statechart with all transition expressions is shown in Figure 2. It is worth noting the loop transition on `(Play | NoPlay)`, which executes a timeout action (200 msec) so that the transition to the target state can only take place at a certain frequency enforced by the `TimeoutCheck` function in the condition. The same action is taken when this state is entered the first time (transition from its *start*-state).

At this point, the user can validate the statechart model and generate the source code. Classes are generated for the model, for the activity of each *basic*-state, and for each transition. Additionally, if the activity of a *basic*-state is not already provided, a class template will be generated in which the user must define the corresponding functionality using conventional C++ code. The user can edit the source code of the activity corresponding to a *basic*-state directly within KSE. In this example, the user just needs to define the activities of the three *basic*-states: `NoPlay`, `SitDownAndStare`, and `StandUpAndScan`. A sample of the auto-generated code is shown in Figures 3 and 4. In our example, the size

```
#include "TestBehavior.h"
#include "transitionHeaders.h"
using namespace statechart_engine;
namespace {StatechartRegistrar<TestBehavior>::Type temp("TestBehavior");}
TestBehavior::TestBehavior(Narukom* com) {
  _statechart = new Statechart ( "Node_TestBehavior", com );
  Statechart* Node_0 = _statechart;
  _states.push_back( Node_0 );

  StartState* Node_0_1 = new StartState ( "Node_0_1", Node_0 ); //Name:0.1
  _states.push_back( Node_0_1 );
...
```

Fig. 3. KSE example: an extract of the generated code for the `TestBehavior` statechart

```
#include "architecture/statechartEngine/ICondition.h"
#include "messages/AllMessagesHeader.h"
#include "tools/BehaviorConst.h"
class TrCond_TestBehavior0_20_2:public statechart_engine::ICondition {
public:
  void UserInit(){_blk->updateSubscription("behavior",msgentry::SUBSCRIBE_ON_TOPIC);}
  bool Eval() {
    boost::shared_ptr<const GameStateMessage> var_621149599 = _blk->readState<
         GameStateMessage> ("behavior");
    boost::shared_ptr<const TimeoutMsg> msg= blk->readState<TimeoutMsg>("behavior");
    return ( (msg.get()!=0 && msg->wakeup()!="" && boost::posix_time::from_iso_string
         (msg->wakeup())<boost::posix_time::microsec_clock::local_time()) && (
         var_621149599.get()!=0 && var_621149599->player_state()!=PLAYER_FINISHED) );
  }
};
#include "architecture/statechartEngine/IAction.h"
#include "architecture/statechartEngine/TimoutAciton.h"
class TrAction_TestBehavior0_20_2:public statechart_engine::TimeoutAction {
  public:TrAction_TestBehavior0_20_2():statechart_engine::TimeoutAction("behavior"
       ,200){;}
};
```

Fig. 4. KSE example: the generated code for the loop transition on `(Play | NoPlay)`

of the total auto-generated code is 35.5 KB and with the user-defined activities for the three *basic*-states it increases to 50.9 KB. Therefore, about 70% of the code for this simple behavior has been automatically generated.

4.2 KSE Design and Implementation

To design and implement KSE, we chose the eclipse platform and the technologies offered by the Eclipse Modeling Project, in particular the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF), and the Xpand language. This choice was apparent, mainly because of the fact that the ASEME tools already used these technologies, but also because other modern CASE tools, such as Yakindu[3], are also based on them.

For creating the liveness formulas, the designer uses a simple text editor. The liveness2IAC transformation tool transforms the liveness formulas to a statechart instance based on the transformation templates for Gaia operators [11]. This text-to-model transformation uses the formal definition of liveness formulas and the `Statechart` metamodel defined in EMF ecore format, shown graphically in Figure 5. According to the `Statechart` metamodel, a *Model* consists of *Node*, *Transition*, and *Variable* instances. A node represents a state in the statechart and has a *name* (providing a description of the state), a *label* (indicating its unique position in the hierarchy), an *activity* (hosting a path to the source code implementing the functionality executed when the state is active), and a *type*

[3] A free toolkit for model-driven development of embedded systems: `www.yakindu.org`

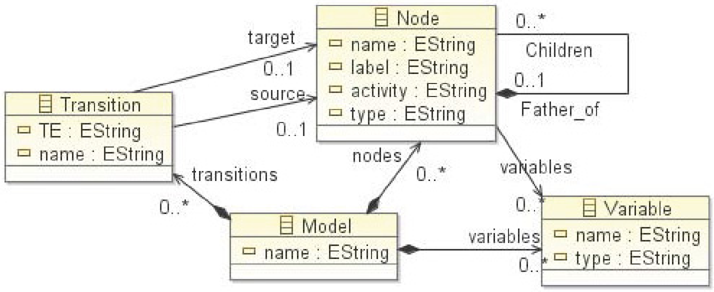


Fig. 5. The `Statechart` metamodel in EMF ecore format

(indicating the type of the state: *or*-state, *and*-state, etc.). Nodes aggregate their *Children* (sub-nodes) and reference their *Father* (parent state). Variables can be defined by the designer and have a *name* and any desired data *type*. Transitions have a *name*, one *source* node, one *target* node, and a transition expression *TE*. Using the EMF representation of the ecore metamodel as Java classes, we used the Java language to write a recursive algorithm that builds the correct statechart by parsing the input liveness formulas.

For editing the statechart we used the GMF technology that allows to associate the ecore metamodel elements with graphical components and dynamically create the graphical model. GMF provides tools for the programmer to define validation rules for the model, if any, and the relevant error or warning messages. Using the GMF API, we also implemented a copy-cut-paste functionality for graphical views, which is not automatically supported by GMF. Thus, the designer can use KSE to edit graphically any part of a statechart, even copying and pasting parts from statecharts of different models.

The IAC2Monas transformation tool has been built using the Xpand language, which is used to define the templates for the required C++ classes. These templates are instantiated using information from the `Statechart` metamodel elements, for example name, label, activity, type, and children of a node.

5 KSE Evaluation

To obtain an empirical evaluation of our CASE tool, 28 ECE undergraduate students taking the Autonomous Agents class at the Technical University of Crete were asked to use KSE and evaluate it in one of their laboratory sessions. The plan of this 2-hour lab session was to go through a short tutorial on using KSE, study a complete SPL Goalie behavior as an example (shown in Figure 6 without the transition expressions), and finally develop their own SPL Attacker behavior using KSE and the functionalities of the Goalie behavior. The predefined functionalities were contained in a Monas source code repository. The students worked in small teams of two or three people per team. None of them had any prior experience with CASE tools, KSE, Monas, SPL, or RoboCup in general. This lab session was run three times to accommodate all students in the four

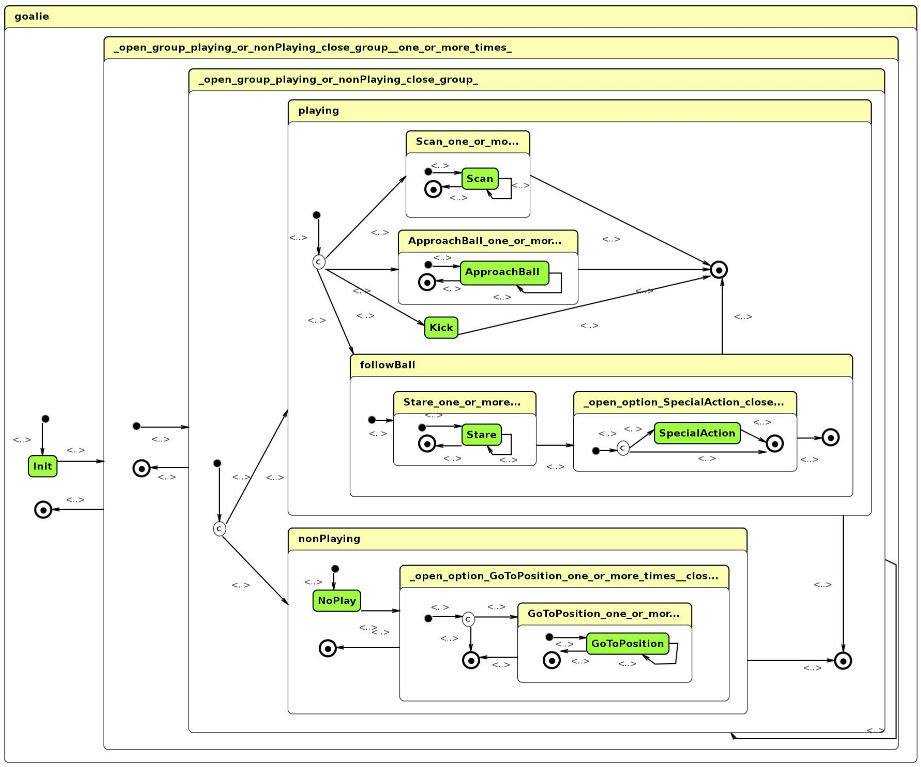

goalie
_open_group_playing_or_nonPlaying_close_group__one_or_more_times_
_open_group_playing_or_nonPlaying_close_group_
playing
Scan_one_or_mo...
Scan
ApproachBall_one_or_mor...
ApproachBall
Kick
followBall
Stare_one_or_more...
Stare
_open_option_SpecialAction_close...
SpecialAction
Init
nonPlaying
NoPlay
_open_option_GoToPosition_one_or_more_times__clos...
GoToPosition_one_or_mor...
GoToPosition

Fig. 6. The statechart of the provided SPL Goalie behavior (expressions not shown)

available work stations. At the end of each lab session, a quick SPL game took place with the four developed attackers split in two teams of two players each.

The results were in general positive for KSE as a CASE tool, but also for the concept of ASEME-based behavior development. Both seemed to be pretty understandable, even though most students were not familiar with Agent-Oriented Software Engineering. All student teams were able to go through the provided material and deliver the requested SPL Attacker behavior. The great bet, won by KSE in this evaluation, was that all student participants succeeded to create a simple SPL robot behavior and even enjoyed watching their players in a game without having to go through the typical lengthy training procedures required for student members of an SPL team.

The participating students were asked to fill in an anonymous user satisfaction questionnaire after the lab session. The total number of responders was 19. A small sample of the user responses is shown in Table 1. The overall assessment of KSE was positive. The main negative comment was that the long transition expressions on the model were cluttering the view of the statechart graph. Based on this comment, in the latest version of KSE the user can choose to hide part(s) of each expression to improve readability of the model.

Table 1. Summarized responses to the KSE user satisfaction questionnaire

Question	Very Easy	Easy	Normal	Difficult	Very Difficult
The liveness formulas was ... to edit.	21.05%	**63.16%**	15.79%	0.00%	0.00%
The statechart was ... to edit.	0.00%	31.58%	**57.89%**	10.53%	0.00%
The navigation to the KSE menu was ...	10.53%	42.11%	**47.37%**	0.00%	0.00%
The use of KSE was ... in general.	0.00%	**57.89%**	26.32%	15.79%	0.00%

6 Related Work and Discussion

Our research revealed two CASE tools that relate most to our work, namely XabslEditor and Yakindu. We briefly review these tools below.

The Extended Agent Behavior Specification Language (XABSL) [13,14] is a simple text-based language for describing behaviors of autonomous agents based on hierarchical finite state machines. XABSL was originally developed for soccer robots behavior specification, but can be used for all kinds of autonomous robots or virtual agents. XabslEditor[4] is a text editor for XABSL, having also the capability to represent graphically the hierarchical finite state machines that describe the agents' behavior. XabslEditor also provides a compiler for XABSL.

The Yakindu toolkit supports the development of both reactive, event-driven and data flow-oriented systems with the help of finite state machines, statecharts (according to Harel), and block diagrams. Yakindu provides graphical modeling tools with integrated validation and simulation, which allow for the early assessment of the models and offers efficient code generators for a target platform.

The main features of the KSE, XabslEditor, and Yakindu tools are summarized in Table 2. KSE compares favorably with the other tools in terms of supported features. A major advantage of KSE for the design process is the analysis tool, which enables the user to abstractly and compactly define the desired behavior using liveness formulas. A small set of liveness formulas can lead to a large statechart model, therefore the user can save a significant amount of time by seeding the design through the analysis tool. As an example, the statechart of the SPL Goalie behavior in Figure 6 was initiated by only four liveness formulas.

The user can configure KSE and choose his/her favorite text editor or programming environment for viewing and editing activities. The default configuration of KSE uses our IAC2Monas code generator for integrating statecharts into our Monas architecture, but KSE can be reconfigured to use any desired source code generator and any grammar for transition expressions in order to generate code for another platform. For example, we have configured KSE to generate code for the popular JADE agent platform using the IAC2JADE tool [15] and a grammar assuming FIPA-ACL[5] communication between the agents.

To facilitate rapid behavior updates, we have configured KSE to store all statecharts and models within the source code repository of our Monas architecture. Thus, all developed statecharts (behaviors) are available at any time and the developer can choose, on the fly, which statechart to execute on the robot.

[4] XabslEditor has been developed by Nao Team Humboldt and is freely available: `www.naoteamhumboldt.de/en/projects/xabsleditor`

[5] The Foundation for Intelligent Physical Agents - Agent Communication Language Specification: `www.fipa.org`

Table 2. Feature comparison of XabslEditor, Yakindu, and KSE

Feature	**XabslEditor**	**Yakindu**	**KSE**
Supported Platforms	java	eclipse helios	linux, windows
Open Source	√	free-ware	√
Model Validation	√	√	√
Analysis Tool			√
Model Simulation		√	
Multiple Editing Tabs	√	√	√
Symbol Auto-Completion	√		
Graphical Editing		√	√
Reusability of Graphical Components			√
Code Generation	√	√	√
Integrated Code Editing	√		√
Customization of Code Generator			√

In general, it is important that the user-defined activities are generic enough to be re-used in a variety of statecharts. This way, a developer can design and test new behaviors, which do not entail the definition of new functionality, just by editing statecharts and reusing existing functionalities.

Recently, we proposed the use of liveness formulas and statecharts also for specifying agent interaction protocols [2]. We showed how an attack protocol can be modeled and then inserted in the respective robot behavior. This feature is fully supported by KSE, as the designer can define the coordinated action and then copy and paste the relevant parts to the interacting robots' statecharts, thus ensuring the correct execution of the protocol. Consider, for example, the following simple attack protocol defining three roles, two instances of *center-for* and one of *center*: the *center* assumes control of the ball and then passes the ball to one of the *center-for*s. The liveness formula for this protocol begins with:

```
attack = center || center-for || center-for
```

Then, in a robot's IAC model the designer defines in a liveness formula that the player will assume either a *center* or a *center-for* role, when attacking:

```
attack = center | center-for
```

When the statechart is generated in the second case, the designer can simply copy the relevant states from the previously edited protocol definition statechart and paste them over the auto-generated *center* and *center-for basic*-states.

7 Conclusion

In this paper, we presented the Kouretes Statechart Editor (KSE), a graphical CASE tool for robot behavior development. KSE offers a number of features that make it suitable for domains, such as RoboCup, where behavior modifications are frequent and require quick and error-proof solutions. The KSE tool is freely available along with other ASEME-based tutorials and tools developed by the RoboCup SPL team Kouretes from `www.kouretes.gr/aseme`.

Acknowledgments. The authors would like to thank Alexandros Paraschos and all members of the Kouretes team for their valuable assistance and our sponsor Chipita S.A.–Molto.

References

1. Baik, J., Boehm, B.: Empirical analysis of CASE tool effects on software development effort. ACIS International Journal of Computer and Information Science 1, 1–10 (2000)
2. Paraschos, A., Spanoudakis, N.I., Lagoudakis, M.G.: Model-driven behavior specification for robotic teams. In: Proceedings of The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Valencia, Spain (June 2012)
3. Harel, D., Naamad, A.: The Statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5, 293–333 (1996)
4. Spanoudakis, N.I., Moraitis, P.: Using ASEME methodology for model-driven agent systems development. In: Weyns, D., Gleizes, M.-P. (eds.) AOSE 2010. LNCS, vol. 6788, pp. 106–127. Springer, Heidelberg (2011)
5. Paraschos, A.: Monas: A flexible software architecture for robotic agents. Diploma thesis, Technical University of Crete, Greece (2010)
6. Vazaios, E.: Narukom: A distributed, cross-platform, transparent communication framework for robotic teams. Diploma thesis, Technical University of Crete, Greece (2010)
7. Spanoudakis, N.I., Moraitis, P.: The agent modeling language (AMOLA). In: Dochev, D., Pistore, M., Traverso, P. (eds.) AIMSA 2008. LNCS (LNAI), vol. 5253, pp. 32–44. Springer, Heidelberg (2008)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35, 114–131 (2003)
9. Hayes-Roth, B.: A blackboard architecture for control. Artificial Intelligence 26(3), 251–321 (1985)
10. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)
11. Spanoudakis, N.I., Moraitis, P.: Gaia agents implementation through models transformation. In: Yang, J.-J., Yokoo, M., Ito, T., Jin, Z., Scerri, P. (eds.) PRIMA 2009. LNCS, vol. 5925, pp. 127–142. Springer, Heidelberg (2009)
12. ISO/IEC: Extended Backus-Naur form (EBNF). 14977 (1996)
13. Loetzsch, M., Risler, M., Jungel, M.: XABSL - a pragmatic approach to behavior engineering. In: 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5124–5129 (October 2006)
14. Risler, M.: Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines. PhD thesis, Technische Universität Darmstadt, Germany (2009)
15. Spanoudakis, N., Moraitis, P.: Modular JADE agents design and implementation using ASEME. In: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT), Toronto, Canada (2010)