

Technical University of Crete
School of Electronic and Computer Engineering

Extending Kouretes Statechart Editor
for Executing Statechart-Based Robotic Behavior Models



Georgios L. Papadimitriou

Thesis Committee

Associate Professor Michail G. Lagoudakis

Assistant Professor Vasileios Samoladas

Dr. Nikolaos Spanoudakis

Chania, July 2014

Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

**Αναβάθμιση του Kouretes Statechart Editor
για Εκτέλεση Μοντέλων Ρομποτικής Συμπεριφοράς
Βασισμένων σε Διαγράμματα Καταστάσεων**



Γεώργιος Α. Παπαδημητρίου

Εξεταστική Επιτροπή
Αναπληρωτής Καθηγητής Μιχαήλ Γ. Λαγουδάκης
Επίκουρος Καθηγητής Βασίλειος Σαμολαδάς
Δρ. Νικόλαος Σπανουδάκης

Χανιά, Ιούλιος 2014

Acknowledgements

First of all, I would like to thank my family for their love and support during my long years of studying towards achieving my engineering Diploma. I would also like to give extra thanks to my brother Panos who's been a great asset to me and constantly helps me in every stage of my life, since I can remember myself. Next, I would like to thank my grandparents for their great support, and my aunt Catherine Geronikolou for helping me in any possible way during my thesis implementation. Moreover, I would like to thank my thesis supervisor, Dr. Michail G. Lagoudakis, for trusting me in investigating this diploma thesis topic, giving me the opportunity to work in my favorite scientific field. Another person I would like to thank, is my co-supervisor, Dr. Nikolaos Spanoudakis, for his exceptional help and willingness to aid me throughout the completion of my thesis. I would like to give exceptional thanks to my father, whose advices and self-sacrifices during the years made me realize in an early stage of my life, that you should be patient and work very hard in order to fulfill your ambitions.

Finally, I would like to thank some of the exceptional friends I met at TUC, especially Labros Papageorgiou (*Mr nice guy*), Lefteris Nikolakakis, Peristeropoulos Athanasios (*Mr roboto*), Thimios Floros, Kleomenis Papadopoulos, Aristomenis Efraimidis and Nektarios Mitakidis (*the evil sorcerer of TUC*).

To my father

Abstract

The development of high-level behavior for autonomous robots is a time-consuming task even for experts. The Kouretes Statechart Editor (KSE) is a Computer-Aided Software Engineering (CASE) tool, which allows to easily specify a desired robot behavior as a statechart model utilizing a variety of base robot functionalities (vision, localization, locomotion, motion skills, communication) developed within the Monas robotic software architecture framework. This thesis presents an extension to KSE, which allows to define generic agent behaviors using automatic framework-independent code generation, as long as the underlying software framework is written in the C++ programming language. This way a user can program behaviors for physical robots or software agents that can be executed on any platform using any C++ software framework. This thesis demonstrates the transparent use of the extended KSE in the SimSpark 3D soccer simulation, the Wumpus world, and the Starcraft Broodwar strategy game.

Περίληψη

Η ανάπτυξη υψηλού επιπέδου συμπεριφορών για αυτόνομα ρομπότ είναι μια αρκετά χρονοβόρα διαδικασία ακόμη και για τους ειδικούς αυτού του τομέα. Το Kouretes Statechart Editor (KSE) είναι ένα Computer-Aided Software Engineering (CASE) εργαλείο, το οποίο επιτρέπει τον εύκολο σχεδιασμό μίας επιθυμητής ρομποτικής συμπεριφοράς, η οποία βασίζεται σε διαγράμματα καταστάσεων, αξιοποιώντας μια ποικιλία βασικών ρομποτικών λειτουργιών (όραση, εντοπισμός, μετακίνηση, κινητικές δεξιότητες, επικοινωνία) που έχουν αναπτυχθεί στα πλαίσια της αρχιτεκτονικής ρομποτικού λογισμικού Monas. Στην παρούσα διπλωματική εργασία παρουσιάζουμε μία επέκταση του εργαλείου KSE, μέσω της οποίας μας επιτρέπεται να ορίσουμε συμπεριφορές αυτόνομων πρακτόρων χρησιμοποιώντας μια γεννήτρια που παράγει πηγαίο κώδικα, ο οποίος δεν έχει κάποια ιδιαίτερη εξάρτηση από το περιβάλλον στο οποίο πρόκειται να εκτελεστεί, αλλά προϋποθέτει μόνο ότι η ευρύτερη αρχιτεκτονική λογισμικού στην οποία θα ενσωματωθεί βασίζεται στην γλώσσα προγραμματισμού C++. Με αυτό τον τρόπο ο χρήστης μπορεί να προγραμματίσει συμπεριφορές για πραγματικά ρομπότ ή πράκτορες λογισμικού που μπορούν να εκτελεστούν σε οποιαδήποτε πλατφόρμα χρησιμοποιεί C++. Στην εργασία αυτή επιδεικνύουμε την διάφανη λειτουργία του αναβαθμισμένου εργαλείου KSE δημιουργώντας συμπεριφορές αυτόνομων πρακτόρων για το περιβάλλον προσομοίωσης ρομποτικού ποδοσφαίρου SimSpark 3D, για τον δημοφιλή κόσμο του Wumpus και για το παιχνίδι στρατηγικής Starcraft Broodwar.

Table of Contents

1 Introduction.....	10
1.1 Motivation.....	11
1.2 Contribution.....	12
1.3 Thesis Outline.....	12
2 Background.....	14
2.1 Model-Driven Engineering and the Eclipse Modeling Project.....	15
2.2 The Xpand Language.....	16
2.3 IAC2Monas.....	18
2.4 ASEME Methodology.....	19
2.5 Kouretes Statechart Editor (KSE).....	23
2.6 Statecharts.....	25
2.7 Blackboard architecture.....	26
2.7.1 The Blackboard Metaphor.....	27
2.7.2 The Blackboard Model of Problem Solving.....	32
2.8 The C++ Programming Language.....	35
3 Problem Statement.....	36
3.1 Autonomous Agent and Multi-Agent systems.....	37
3.2 Developing Autonomous Agent Behaviors.....	38
3.3 Team Kouretes.....	39
3.4 Robotics Simulators.....	40
3.5 Automatic Programming.....	41
3.6 Related Work.....	42
4 Our Approach.....	45
4.1 The KSE Generic C++ Generator.....	46
4.2 Generic Blackboard.....	48
4.3 Transition Expression and C++ Generator.....	49
4.4 Transition Expression Example.....	51
4.5 KSE Integration.....	52
4.6 Implementation.....	54

4.6.1	Creating a Generic Blackboard Interface.....	54
4.6.2	Why Use the Blackboard Problem-Solving Approach?.....	56
4.6.3	Developing our Generator using Xpand and Workflows.....	57
4.7	Editing the Statechart Engine.....	60
4.8	Summary.....	60
5	Results.....	60
5.1	Creating a Behavior for the SimSpark 3D Robotic Soccer Simulator.....	62
5.1.1	SimpSpark 3D Robotic Soccer Simulator.....	62
5.1.2	Simulation using SimSpark: The Soccer Server and the Monitor.....	62
5.1.3	Robot Model used by the Soccer Server.....	65
5.1.4	Communication between Agents and Soccer Server.....	67
5.1.5	Synchronization between the Server and the Agents.....	67
5.1.6	Simple Soccer Agent: C++ Framework for the SimSpark Simulator.....	68
5.1.7	Developing the Agent's Behavior using the Extended KSE.....	69
5.2	Cooperative Multi-Agent Behavior.....	75
5.3	Creating a Behavior for the Wumpus World Simulator.....	77
5.3.1	Platform Overview.....	77
5.3.2	Simulation using the Wumpus World Simulator.....	78
5.3.3	Wumpus Agent: C++ Framework for the Wumpus World Simulator.....	79
5.3.4	Developing the Agent's Behavior using the Extended KSE.....	80
5.4	Creating a Behavior for the Starcraft Brood War Strategy Game.....	84
5.5	Summary.....	86
6	Conclusion.....	87
6.1	Discussion.....	88
6.2	Future work.....	90
6.3	Lessons Learned.....	90
	References.....	91

List of figures

Fig. 2.1 oAW example workflow.....	17
Fig. 2.2 Workflow example.....	18
Table 2.1 Operators for Liveness Formula.....	21
Table 2.2 The liveness formula grammar in EBNF format.....	22
Table 2.3 Gaia operators transformation templates.....	24
Fig. 2.3 The blackboard architecture scheme.....	32
Fig 2.4 Basic components of the Blackboard Model.....	33
Fig. 3.1 The statechart model should be the same for real world and simulator.....	40
Fig. 3.2 The Yakinidu environment.....	45
Fig. 4.1 Our GGenerator as a new module to the old KSE.....	47
Fig 4.2 Use of our generic Blackboard interface as a middleware.....	49
Fig. 4.3 Transition expression grammar in EBNF format.....	50
Fig. 4.4 The scheme of our software architecture for different frameworks.....	51
Fig. 4.5 Code for the event part of the transition expression.....	52
Fig. 4.6 Statechart model for the surveillance.....	52
Fig. 4.7 Code for the action part of the transition expression.....	52
Fig 4.8 Code for the condition part of transition expression.....	53
Fig. 4.9 Initialize KSE with a new code generator.....	53
Fig 4.10 Selecting the GGenerator.....	54
Fig. 4.11 The BlackBoard interface class prior to it's configuration.....	55
Fig 4.12 The BlackBoard interface class after the configuration and the code generation.....	56
Fig. 4.13 BlackBoard interface Xpand template (pages 1-3).....	58
Fig. 4.14 BlackBoard interface Xpand template (page 4).....	58
Fig. 4.15 Condition Xpand template.....	59
Fig. 4.16 The workflow file for our GGenerator.....	59
Fig. 5.1 The dimensions of the field and the object markers on the field as perceived by an agent. .	63
Fig. 5.2 Soccer Monitor connected to a soccer simulation with 6 vs 6 robots (source: Wiki).....	64
Fig. 5.3 List of Soccer Monitor commands.....	65
Fig. 5.4 The real Nao robot (source:Wiki).....	66
Fig. 5.5 The virtual Nao in the simulation environment (source:Wiki).....	66

Fig. 5.6 Synchronization between Soccer Server and the agent (source: Wiki).....	68
Fig. 5.7 Fragment from Cognition.h.....	69
Fig. 5.8 Fragment from Cognition.cpp.....	69
Fig. 5.9 properties files for SimpleSoccerAgent framework.....	70
Fig. 5.10 Logical Agent Statechart model with all the transition expressions.....	73
Fig. 5.11. we add the agent on the environment.....	74
Fig. 5.12 The agent searches for the ball.....	74
Fig. 5.13 The agent goes towards the ball.....	74
Fig. 5.14 The agent takes possession of the ball.....	74
Fig. 5.15 Generated blackboard fragment (a).....	74
Fig. 5.16 Generated blackboard fragment (b).....	74
Fig. 5.17 Part of the generated code for our behavior.....	75
Fig. 5.18 Generated code for a transition expression.....	75
Fig. 5.19 Changes to Cognition's call function.....	75
Fig. 5.20 Changes to Cognition's init function.....	75
Fig. 5.21 Player_1 liveness formula.....	76
Fig. 5.22 Player_2 liveness formula.....	76
Fig. 5.23 Player_1 Statechart.....	76
Fig. 5.24 Player_2 Statechart.....	76
Fig. 5.25 Cooperative Behavior screen-shot 1.....	77
Fig. 5.26 Cooperative Behavior screen-shot 2.....	77
Fig. 5.27 Start team script.....	77
Fig. 5.28 Kill team script.....	77
Fig. 5.29 4x4 Wumpus World simulation.....	80
Fig. 5.30 7x7 Wumpus World Simulation.....	80
Fig. 5.31 Sample Agent Process method.....	80
Fig. 5.32 The properties files for the Wumpus World Framework.....	81
Fig. 5.33 Wumpus Statechart model with all the transition expressions.....	83
Fig. 5.34 Transition expression generated code.....	83
Fig. 5.35 Part of the generated blackboard.....	83
Fig. 5.36 Generated source code of our behavior.....	84
Fig. 5.37 Register our agent in Wumpsim class.....	84

Fig. 5.38 Two successive screen-shots from our behavior in the wumpus world simulator.....	84
Fig. 5.39 Properties files for StarCraft.....	85
Fig. 5.40 Liveness formula for StarCraft agent.....	85
Fig. 5.41 Statechart model for the StarCraft game.....	85
Fig. 5.42 Two successive screen-shots from our starcraft behavior.....	86
Fig. 6.1 Major differences between the three platforms we use.....	89

Chapter 1

Introduction

1.1 Motivation

Nowadays artificial intelligence systems are being applied to a wide variety of domains, like the Internet industry, the video-game industry, health-care industry and many more. These artificial intelligence systems are utilized to work autonomously towards achieving a user-defined goal, depending on the domain they are being applied to. Moreover, we divide these systems into single-agent systems and multi-agent systems (collection of single agents); as agents, we refer to the modules of the system that act autonomously. In either case, these systems can be described abstractly by the behavior they demonstrate when being applied to a specific environment. Thus, we can see that is of most importance to be able to define or edit the behavior of such a system at an abstraction layer that separates us from the hardware level of the system.

Developing an agent behavior for a specific environment from scratch can be a tedious task, even for experts. This led the programming community to develop various Computer Aided Software Engineering (CASE) tools, which simplify this kind of tasks. The majority of these tools are developed in order to work with a specific underlying hardware architecture. One such tool is the Kouretes Statechart Editor (KSE) CASE tool. KSE provides the user with an interface for defining a statechart-based agent behavior for the Monas framework. The Monas framework is a software architecture developed by our Robotic Soccer team Kouretes primarily for the Aldebaran Nao humanoid robots. KSE provides an easy and user-friendly graphical interface for all the phases of the development of an artificial intelligence agent behavior, according to the ASEME methodology, a model-driven software engineering methodology from the Agent-Oriented Software Engineering (AOSE) domain. Unfortunately, when we create a behavior using KSE we are bound to use the Monas framework and the real robot in order to test it. We wanted to be able to use the benefits provided by KSE in order to develop agent behaviors for other frameworks besides Monas. The need for a tool like this arises, because in many cases we want to be capable of testing

our developed team behaviors in a simulated environment (*e.g.* the [SimSpark 3D Simulator](#)), but not on the real Nao robots. However, the SimSpark platform is not compatible with the Monas robotic software architecture, which we use for deploying behaviors on the Nao robots. This is an important extension, because when we work on a simulated environment, we are free to work without the real robots and we can avoid many of the shortcomings imposed to us by hardware issues.

1.2 Contribution

Our approach aims towards expanding the KSE tool for generic agent behavior development. We do this by creating a new source code generator, the Generic C++ Generator, that can be configured during runtime in order to generate a behavior for any user-defined framework, as long as it supports C++. This gives us the opportunity of testing our statechart-based behaviors on a simulator rather than testing them on the real Nao robots.

We test our solution in three diverse environments: the SimSpark 3D Robotic Soccer Simulator, the Wumpus World Simulator, and the StarCraft Brood War strategy game. The first environment simulates a 3D world, where every agent in it is represented by a simulated Nao robot, the second is a 2D world, where an agent tries to solve the famous Wumpus maze, and the third is a famous strategy game developed by Blizzard. We develop and generate agent behaviors for these three environments, following the exact same procedure and, thus, we demonstrate the transparent use of the extended KSE in various C++-based domains.

1.3 Thesis Outline

As mentioned above, the main contribution of this thesis is the expansion of KSE with a new C++ generic source code generator (GGenerator) in order to provide a tool for developing statechart-based agent behaviors, according to the ASEME methodology, that can be applied to any user-defined framework, as long as this framework is implemented in the C++ programming language.

In Chapter 2, we give a brief report about the tools and the methods we use throughout the development of our GGenerator. At first, we present the Eclipse Modeling framework, along with the concept of model-driven engineering. Then, we review the various software tools (like IAC-Monas and KSE) that make use of the ASEME methodology and we discuss the advantages ASEME provides to model-driven engineering. At last, we introduce the concepts of statecharts and

blackboard software architecture.

In Chapter 3, we state our problem. We start by giving brief information about artificial intelligence software systems. More specifically, we talk about the artificial intelligence software agents that are a vital part of these systems and we introduce the concept of defining a behavior for such an agent. We, then, introduce various aspects of automatic programming, like generative programming and, finally, we present some of the related work that has been done for the aforementioned problem.

In Chapter 4, we start by giving a brief explanation for the need of a tool like Ggenerator and we also present our approach towards the solution of the problem stated in Chapter 3. Then, we explain in an abstract way our tool and the way we embedded it as a new module to the KSE framework. Afterward, we explain in a detailed way the steps we took in order to implement our new source-code generator. We discuss thoroughly the technical aspects of our implementation and the ideas and methodologies used. Next, we introduce the idea of a generic blackboard interface, which is a vital part of our software architecture. The idea of using a generic blackboard interface is explained thoroughly in this chapter, since it is the glue that holds our system together. Then, we present technical aspects of our new source code generator, like the new transition expressions grammar. We demonstrate the use of our new grammar for a sample test (*surveillance camera system*). We also present the idea of properties files and their use as configuration files for our blackboard interface. Our blackboard interface can target any C++ framework as long as it is configured with the appropriate properties files.

In Chapter 5, we present the results taken when using the new GGenerator along with the KSE tool for creating behaviors for three different and very diverse frameworks; SimSpark 3D Robotic Soccer Simulator, Wumpus World Simulator, and StarCraft Brood War. At first, we demonstrate a standard “*three-step procedure*” for developing agent behaviors for specific environments with the use of tools that the expanded KSE provides. In the first case, we created a behavior for an agent that can interact with SimSpark, in the second case, we created a behavior that can interact with the Wumpus simulator, and, in the third case, we developed a behavior that can interact with the StarCraft environment. Moreover, we demonstrate a cooperative scenario for the SimSpark 3D simulator. Before we close chapter 5, we give the results of our three simulations and we underline the fact that the same procedure was used in order to develop the behaviors for the three different frameworks. Part of the code that our GGenerator produces is presented for all of the aforementioned environments. This code includes the generated blackboard interface along with the

code generated for a transition expression and for the agent's behavior.

Finally in Chapter 6, we can find the conclusion of this thesis along with our ideas and plans for future work. We pinpoint the major differences between the three platforms we used in our examples and we discuss the contribution of this thesis and the way we want to use it in the future.

Chapter 2

Background

In this chapter we are going to discuss concepts that already exist and are used by this thesis. It acts both as a reference, as well as an introduction to the reader to familiarize with the various fields that this thesis combines. At first we present the tools we used in our implementation; then, we continue by describing the main data modules (*e.g statecharts*) and methodology (*ASEME, Blackboard architecture*) that we used towards building our software application.

2.1 Model-Driven Engineering and the Eclipse Modeling Project

Over the past five decades, software researchers and developers have been creating abstractions that help them program in terms of their design intent rather than the underlying computing environment – for example, CPU, memory, and network devices – and shield them from the complexities of these environments. Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively. **Model-driven engineering (MDE)** is a software development methodology which focuses on creating and exploiting domain models (that is, abstract re-presentations of the knowledge and activities that govern a particular application domain), rather than on the computing (i.e. algorithmic) concepts.

The MDE approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain). A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive

communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

The **Eclipse Modeling Project (EMP)** is a top-level project at Eclipse (www.eclipse.org/modeling, Budinsky F. 2003). In contrast, the core of the project, *EMF*, has been in existence for as long as the Eclipse platform itself. Today the modeling project is largely a collection of projects related to modeling technologies. This collection was formed to coordinate and focus on model-driven software development capabilities within Eclipse. The introduction of the Amalgamation project ushered in the beginnings of a *Domain Specific Language (DSL) focused* development environment, although it has a long way to go before mainstream developers can use it.

The Modeling project is organized logically into projects that provide the following capabilities: abstract syntax development, concrete syntax development, model-to-model transformation, and model-to-text transformation. A single project, the Model Development Tools (MDT) project, is dedicated to the support of industry-standard models. Another project within the Modeling project focuses on research in generative modeling technologies.

Specifically *EMP* consists of *EMF* (*Eclipse Modeling Framework*), *QVT* (*Query: Validation: Transaction*), *M2M* (*Model-to-Model transformation*), *M2T* (*Model- to-Text transformation*), *TMF* (*Textual Modeling Framework*) and *GMF* (*Graphical Modeling Framework*). *EMF* allows the developer to define a DSL language in an abstract syntax. *EMF* has as an output a model that describes a new language. *QVT* provides query, validation and transaction features for the *EMF* models. *M2M* provides Operational Mapping Language that allows model-to-model transformation for *EMF* models. *M2T* allows model-to-text by using *JET* (*Java Emitter Template*) or *Xpand* as a template engine. *TMF* is still under development and does not offer a lot of capabilities, but its purpose is to provide a textual editors with syntax highlighting, code completion and build for *EMF* models. In the other hand, *GMF* provides graphical editors for *EMF* models.

2.2 The Xpand Language

Xpand was developed from scratch as a part of the openArchitectureWare platform (oAW). While there are already many template languages available, the authors of the framework have realized that an effective template development is only possible with an easy-to-learn, domain-

specific language (DSL) for code generation, as well as good tool support. The Xpand language itself has a small but sufficient vocabulary. Beside its own capabilities, it can access functions implemented in the Xtend programming language, which is another domain-specific language that is contained in the oAW framework (Klatt B. 2007).

The openArchitectureWare system is based on a workflow engine which executes different processing steps, like model instantiation, validation, model2model and model2text transformations. Figure 2.1 shows an example workflow. The components can be configured arbitrarily so that it is possible to parse multiple models and combine them into an internal abstract syntax graph. Also, multiple transformers for model2model transformations with a validation step after each transformation could be configured, as well as multiple generators for different target artifacts.

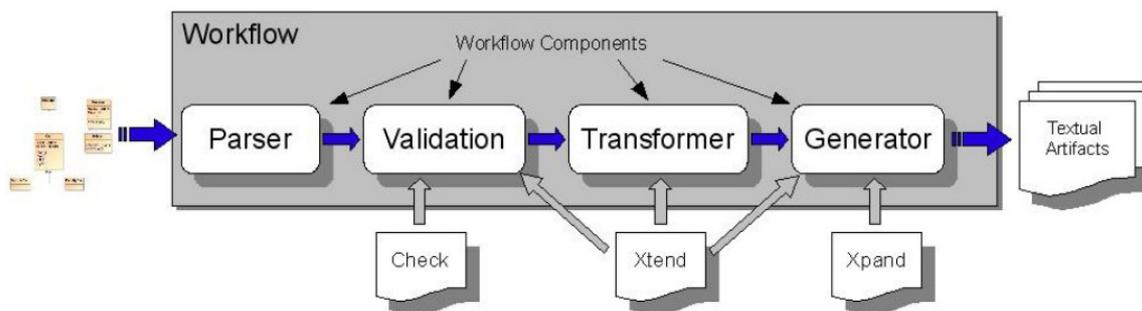


Fig. 2.1 oAW example workflow

In this workflow the code generation (or model2text transformation) step will be linked to the workflow definition.

The workflow is defined in an XML descriptor file. A generator can be configured as shown in figure 2.2. First of all, the generator class is defined in the component tag. Inside this tag the required meta models are referenced so that they can be accessed by the generator. Then, the main template that will be processed is referenced. The last more complex outlet tag sets the target directory for the generated artifacts and it includes the JavaBeautifier to produce better formatted Java code (<http://wiki.eclipse.org/Xpand>).

Example

...

```
<component id="generator"  
class="org.openarchitectureware.xpand2.Generator2">  
<metaModel idRef="emf"/>  
<metaModel idRef="uml2"/>  
<metaModel idRef="profile"/>  
<expand value="Template::define FOR mySlot"/>  
<outlet path="main/src-gen">  
<postprocessor  
class="org.openarchitectureware.xpand2  
.output.JavaBeautifier"/>  
</outlet>  
</component>
```

...

Fig. 2.2 Workflow example

Xpand itself is independent from the type of the source model. Different source models are handled by a parser linked to the openArchitectureWare workflow. Parsers can be written for any kind of source model but openArchitectureWare provides out-of-the-box parsers for EMF, Eclipse-UML2, different UML-Tools (MagicDraw, Poseidon, Enterprise-Architect, Rose, XDE) and textual models using the Xtext framework as well as for XML and Visio.

The advantages of Xpand are the fact that it is source model independent, its vocabulary is limited allowing for a quick learning curve while the integration with Xtend allows for handling complex requirements. Then, EMP allows for defining workflows that can help a modeler to achieve multiple parsings of the model with different goals.

2.3 IAC2Monas

IAC2Monas (Paraschos A. 2010) is a code generator, which extracts a statechart model in C++ language compatible with Monas architecture from a IAC model. IAC2Monas was developed by Alexandros Paraschos for Kouretes. IAC2Monas is developed in Xpand and java language and uses these java packages:

- org.eclipse.emf.mwe.utils.Reader
- org.eclipse.xpand2
- java.util.HashSet

- java.util.List
- java.util.Set
- java.util.StringTokenizer

2.4 ASEME Methodology

The *Agent Systems Engineering METHodology (ASEME)*, Spanoudakis N. 2009) is an Agent Oriented Software Engineering (AOSE) methodology for developing multi-agent systems. It uses the *Agent MOdeling LAnguage (AMOLA)*, which provides the syntax and semantics for creating models of multi-agent systems covering the analysis and design phases of a software development process. It 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 his capabilities, while the latter defines the protocols that govern the coordination of the society of the agents.

ASEME applies a model driven engineering approach to multi-agent systems development, so that the models of a previous development phase can be transformed to models of the next phase. Thus, different models are created for each development phase and the transition from one phase to another is assisted by automatic model transformation, including model to model (*M2M*), text to model (*T2M*), and model to text (*M2T*) transformations leading from requirements to computer programs.

The *ASEME Platform Independent Model (PIM)*, 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. The *Agent Modeling Language (AMOLA)*, Spanoudakis N. et.al. 2008) describes both an agent and a multi-agent system. The concept of functionality is defined to represent the thinking, thought and senses characteristics of an agent. Then, the concept of capability is defined as the ability to achieve specific goals (e.g. the goal to decide in which restaurant to have a diner this evening) that requires the use of one or more functionalities. Therefore, the agent is an entity with certain capabilities, including inter and intra-agent communication. Each of the capabilities requires certain functionalities and can be defined separately from the other capabilities.

The capabilities are the modules that are integrated using the intra-agent control concept to define an agent. Each agent is considered a part of a community of agents, i.e. a multi-agent system.

Thus, the multi-agent system's modules are the agents and they are integrated into it using the inter-agent control concept. The intra-agent control concept allows the assembly of an agent by coordinating a set of modules, which are themselves implementations of capabilities that are based on functionalities. Here, the concepts of capability and functionality are distinct and complementary.

The agent developer can use the same modules, but different assembling strategies, proposing a different ordering of the modules execution producing in that way different profiles of an agent. This approach provides an agent with a decision making capability that is based on an argumentation based decision making functionality. Another implementation of the same capability could be based on a different functionality, e.g. multi-criteria decision making based functionality.

Then, in order to represent system designs, *AMOLA* is based on statecharts, a well-known and general language and does not make any assumptions on the ontology, communication model, reasoning process or the mental attitudes (e.g. belief-desire-intentions) of the agents giving this freedom to the designer. The *AMOLA* models are related to the requirements analysis, analysis and design phases of the software development process. *AMOLA* aims to model the agent community by defining the protocols that govern agent interactions and each part of the community, the agent, focusing in defining the agent capabilities and the functionalities for achieving them. The details that instantiate the agent's functionalities are beyond the scope of *AMOLA* that has the assumption that they can be achieved using classical software engineering techniques.

In the analysis phase *ASEME* defines agent roles and protocols that govern their interaction. To model the behavior of the roles at this phase uses liveness formulas. The liveness formula is a process model that describes the dynamic behavior of the role by itself, i.e. in the systems role model (*SRM*) or the role's behavior inside a protocol, i.e. in the agent interaction protocol model (*AIP*). It connects all the role's activities using the *Gaia* operators (Table 2.1). The liveness formula defines the dynamic aspect of the role, that is which activities execute sequentially, which concurrently and which are repeating.

Operator	Interpretation
$x . y$	x followed by y
$x y$	x or y occurs
x^*	x occurs 0 or more times
x^+	x occurs 1 or more times

$x\sim$	x occurs infinitely
$[x]$	x is optional
$x \parallel y$	x and y interleaved

Table 2.1. Operators for Liveness Formula

An *AIP* defines one or more participating agent roles, the rules for engaging (why would the roles participate in this protocol), the outcomes that they should expect in successful completion and the process that they would follow in the form of a liveness formula.

The *System Roles Model (SRM)* is mainly inspired by the *Gaia* roles model. A role model is defined for each agent role. The role model contains the following elements: a) the interaction protocols that this agent will be able to participate in, b) the liveness model that describes the role's behavior. The liveness model has a formula at the first line (root formula) where activities or capabilities can be added. A capability must be decomposed to activities in a following formula. The liveness formula grammar has not been defined formally in the literature, thus it is defined here using the *Extended Backus-Naur Form (EBNF)*, which is a metasyntax or metametamodel notation used to express context-free grammars. It is a formal way to describe computer programming languages and other formal languages. It is an extension of the basic *Backus-Naur Form (BNF)* metasyntax notation. *EBNF* was originally developed by Niklaus Wirth (1996). The *EBNF* syntax for the liveness formula (Table 2.2), using the *BNF* style followed by Russell and Norvig, i.e. terminal symbols are written in bold.

After completing the functionality Table the engineer can pass to the design phase in which *EAC* and *IAC* models are created. The Inter-Agent Control (*EAC*) is defined as a statechart. It should be initialized by transforming the agent interaction protocols of the analysis phase to statecharts. Harel and Kugler (2004) present the statechart language adequately, but not formally. David's *UML* semantics for statecharts has been used as basis for the definition of the *AMOLA* statecharts as it is the first intended for object-oriented language implementation. These models not only formally describe the elements of the statechart, they also focus on the execution semantics. It is assumed that, as long as the language of statecharts is not altered, a statechart can be executed with any semantics available depending on the available CASE tool. The formal model that is adopted here-in is a subset of the ones presented in the literature as there are several features of the statecharts not used herein, such as the history states (which are also defined differently in these

works).

Before formally defining the statechart for the EAC model, the elements that compose the transition expressions are examined. Then, the transition expressions are defined in EBNF. Transitions are usually triggered by events. Such events can be:

- a sent or received (or perceived, in general) inter-agent message
- a change in one of the executing state's variables (also referred to as an intra-agent message)
- a time-out
- the ending of the executing state activity

liveness	→formula
formula	→leftHandSide " = " expression
leftHandSide	→string
expression	→term → parallelExpression → orExpression → sequentialExpression
parallelExpression	→term " " term " " ... " " term
orExpression	→term " " term " " ... " " term
sequentialExpression	→term "." term "." ... "." term
term	→basicTerm "(" expression ")" → " [" expression "]" → term " ★ " → term + → term " ~ " → " " term " ~ " number
basicTerm	→string
number	→digit digit number
digit	→"1" "2" "3" ...
string	→letter letter string
letter	→"a" "b" "c" ...

Table 2.2. The liveness formula grammar in EBNF format

The latter case is also true for a transition with no expression. Note that each state automatically starts its activity on entrance. A message event is expressed by $P(x,y,c)$ where P is the performative, x is the sender role, y the receiver role and c the message body. The items that the designer can use for defining the state transition expressions are the message performatives, the

ontology used for defining the messages content and the timers. An agent can define timers as normal variables initializing them to a value representing the number of milliseconds until they timeout (at which time their value is equal to zero). The transition expressions can use the time-out unary predicate, which is evaluated to true if the timer value is equal to zero, and false otherwise. Timers are initialized in the action part of a transition expression, while the time-out predicate can be used in both the event and condition parts of the transition expression depending on the needs of the designer.

Besides inter-agent messages and timers there is another kind of events, the intra-agent messages. The change of a value of a variable can have consequences in the execution of a protocol. The variables taking part in a transition expression imply the fact that they are defined in the closest common ancestor OR state of the source and target states of the transition or higher in the statechart nodes hierarchy. The intention regarding the performative definition is not to enumerate all possible performatives, the modeler can define such as he sees fit.

In the agent level, the Intra-Agent Control (*IAC*) is defined using statecharts in the same way with the Inter-Agent Control model (*EAC*). The difference is that the top level state (root) corresponds to the modeled agent (which is named after the agent type). One *IAC* is defined for each agent type.

ASEME provides a tool for transforming liveness formulas to statecharts. The Liveness2Statechart transformation is achieved by using the “Gaia operators transformation templates” (shown in Table 2, Spanoudakis N. et. al. 2009) for transforming the process part of the agent interaction protocol model to a statechart. Thus, the statechart models (*EAC* and/or *IAC*) of the design phase are auto-generated and the developer just needs to add the transition expressions.

2.5 Kouretes Statechart Editor (KSE)

The Kouretes Statechart Editor (KSE, Topalidou-Kyniazopoulou 2012) is a Computer-Aided Software Engineering (CASE) tool that was developed by Angelica Topalidou Kyniazopoulou for *Kouretes*. It 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).

KSE adopts the *Agent Systems Engineering Methodology (ASEME)* model-driven approach. Thus, it guides the developer through a series of design steps within a graphical environment that

leads to automatic source code generation . *KSE* was used for developing the behavior of the Nao humanoid robots of our team *Kouretes* competing in the *Standard Platform League* of the *RoboCup* competition.

table 2.3 *Gaia* operators transformation templates

Operator	Template	Operator	Template
$x \mid y$		$x \cdot y$	
x^*		$[x]$	
x^ω		x^+	
$x \parallel y$		$ x^\omega ^n$	

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. 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 (Harel D. Kugler H. 1998) 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.

2.6 Statecharts

Finite state machines (*FSM*) are computational models that consist of a set of states, an initial state, an input alphabet and a transition function that maps every legal state combination to an other legal state combination, given an input symbol. Hence, FSMs, “specifies the sequence of states an object goes through during its lifetime in responses to events, together with its responses to those events”. *FSMs* achieve better results from textural representations when describing reactive rather than transactional systems.

Statecharts are state diagrams, very useful for behavioral modeling. They differ from other forms of state diagrams, such as the classical finite state machines and its derivatives, because they address two major problems that mainly affect the number of nodes and transitions: hierarchy and orthogonality. Additionally, statecharts incorporate a powerful visual representation which improves the readability and understanding by the reader. Statecharts do not have a single formalism. Historically, the first one is Classical Harel’s statecharts, while the other two were developed almost concurrently —borrowing elements from each other —are the object-oriented version of Harel’s statechart (implemented in Rhapsody tool, *Harel D. Kugler H. 1998*) and the UML State Machine Diagrams. In this thesis, the formalism that is followed is a modified version of Rhapsody statecharts (each difference is stated explicitly).

There are three types of states in a statechart, OR-states, AND- states, and basic states. OR-states have sub-states that are related to each other by “exclusive-or ”, and AND-states have orthogonal components that are related by “and ”(they are executed in parallel). Basic states are those at the bottom of the state hierarchy, that is those that have no sub-states. The state at the highest level (the one with no parent state) is called the root. The active states at a specific time, consist the active configuration of the statechart.

The execution flow is decided from the transitions between the states. Each transition from one state (source) to another (target) can be labeled by an expression, whose general syntax is $e[c]/a$, where e is the event that triggers the transition; c is a condition that must be true in order for the transition to be taken when e occurs; and a is an action that takes place when the transition is taken. All elements of the transition expression are optional. A transition with an empty transition expression, all three parts missing, is called a null transition. Moreover, there are compound

transitions (CT). These transitions are sequences of transition segments, connected by special states (defined as connectors) between a source and a target state, or from another point of view, transitions that can have more than one source or target states. There are two kinds of CTs: AND-connectors and OR-connectors. AND connectors are of two types, joint transitions (more than one sources) and fork transitions (more than one targets). The most commonly used OR-connector is the conditional transition. The scope of a transition is the lowest level OR-state that 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.

Additionally, two more categories of states exist to help the realization of specific behaviours on statecharts: pseudo-states and transition connectors. In the former category we can locate START and END states, which represent the initial transition and a sink (a state with no outgoing transitions). In the latter category we can find out states that are used on compound transitions such as the junction, condition, fork and join connectors.

As being defined for FSMs, statecharts are changing configurations given an event. Then, none, one or more transitions (or compound transitions) are activated and change the active configuration of the statechart, leaving it in a legal —statecharts can never “stop” their execution in the middle of a transition segment, a pseudo-state, a connector or by activating a composite state and not its substate —and stable (no more null-transitions can be executed) configuration.

Problems arise when more than one transitions can be executed at a specific execution step, but each one leads to a different active configuration. The point is crucial as if the two or more transitions are in different scopes, the one with the lower scope has priority, but if the transitions are in the same scope, then we arbitrarily select one (the selection depends on the implementation).

Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart. If one of the statecharts becomes non-active (e.g. when the activity it controls is stopped) the other charts continue to be active and that statechart enters an idle state until it is restarted.

2.7 Blackboard architecture

Since in our thesis we build a blackboard software architecture (*Hayes-Roth B. 1985*), we feel obligated to present a brief report about blackboard systems and their mechanics.

A blackboard system is an artificial intelligence application based on the blackboard

architectural model, where a common knowledge base, the "blackboard", is iteratively updated by a diverse group of specialist knowledge sources, starting with a problem specification and ending with a solution. Each knowledge source updates the blackboard with a partial solution when its internal constraints match the blackboard state. In this way, the specialists work together to solve the problem. The blackboard model was originally designed as a way to handle complex, ill-defined problems, where the solution is the sum of its parts.

Blackboard systems are not new technology. The first blackboard system, the Hearsay-II speech understanding system, was developed nearly twenty years ago. While the basic features of Hearsay-II remain in today's blackboard systems, numerous advances and enhancements have been made as a result of experience gained in using blackboard systems in widely varying application areas.

Unlike most AI problem-solving techniques that implement formal models, the blackboard approach was designed as a means for dealing with ill-defined, complex applications. Unconstrained by formal requirements, researchers and developers have had considerable flexibility in inventing and applying advanced techniques to blackboard architectures. However, the lack of formal specifications has also contributed to confusion about blackboard systems and their proper place in the AI problem-solving toolkit.

2.7.1 The Blackboard Metaphor

Blackboard-based problem solving is often presented using the following metaphor:

Imagine a group of human specialists seated next to a large blackboard. The specialists are working cooperatively to solve a problem, using the blackboard as the workplace for developing the solution. Problem solving begins when the problem and initial data are written onto the blackboard. The specialists watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When a specialist finds sufficient information to make a contribution, she records the contribution on the blackboard, hopefully enabling other specialists to apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

This simple metaphor captures a number of the important characteristics of blackboard systems,

each of which is described separately below.

- ***Independence of expertise (I think, therefore I am.)***

The human specialists in the metaphor were not trained to work solely with that specific group of specialists. Our metaphorical specialists learned their expertise in vastly different situations. Some specialists have years of work experience, others recently received academic degrees, and still others are outside consultants brought in specifically for this particular problem. Each specialist is a self-contained expert on some aspects of the problem and can contribute to the solution independently of the particular mix of other specialists in the room.

Blackboard systems also have this functional modularization of expertise. Each knowledge module (called a Knowledge Source, or simply a KS) is a specialist at solving certain aspects of the overall problem. No KS requires other KSs in making its contribution. Once it finds the information it needs on the blackboard, it can proceed without any assistance from other KSs. Furthermore, without changing any other KSs, additional KSs can be added to the blackboard system, poorer performing KSs can be enhanced, and inappropriate KSs can be removed. KSs perform relatively large computations, reflecting the processing required to implement their specialty.

Rule-based systems are also modular, but at the level of individual rules. Unlike the large-grained scope of KSs, the small size of each rule prevents full independence. A pair of rules that implement iteration by using a counter value and a termination rule is an example of two rules that cannot be designed independently or removed individually without affecting the performance of the other rule.

- ***Diversity in problem-solving techniques (I don't think like you do.)***

There are vast differences in how human experts think about and solve problems. Yet, these differences do not prevent our metaphorical group of specialists from solving the problem.

In blackboard systems, the internal representation and inferencing machinery used by each KS is similarly hidden from direct view. The blackboard approach views each KS as a black box in which the internal workings are invisible from the outside. It does not matter if one KS is a forward-chaining rule-based system, another uses a neural network approach, another uses a linear-programming algorithm, and still another is a procedural simulation program. Each of these diverse approaches can make its contributions within the blackboard framework.

- ***Flexible representation of blackboard information (If you can draw it, I can use it.)***

Our metaphorical human specialists could use any intelligible doodles when adding their contributions to the blackboard. They might use formulas, diagrams, sentences, checklists, and numerous circles and arrows.

Representational flexibility is similarly important in blackboard systems. The blackboard model does not place any prior restrictions on what information can be placed on the blackboard. One blackboard application might use assertional blackboard data and require that consistency be maintained. Another application might allow incompatible alternatives to be maintained on the blackboard, with each alternative available for opportunistic¹ exploration of the solution.

- ***Common interaction language (What'd you say?)***

While flexible representation of blackboard information is important, there must also be a common understanding of the representation of the information placed on the blackboard in order for the specialists to interact. The formulas, diagrams, sentences, and checklists must be understood by all specialists who need to access the information. If our metaphorical specialists consisted of specialists of differing nationalities, the use of different languages on the blackboard would hamper or even prohibit sufficient interaction to solve the problem.

Similarly, KSs in blackboard systems must be able to correctly interpret the information recorded on the blackboard by other KSs. Private jargon shared by only a few KSs limits the flexibility of applying other KSs on that information. In practice, there is a trade off between the representational expressiveness of a specialized representation shared by only a few KSs and a fully general representation understood by all KSs. Finding the proper balance is an important aspect of blackboard-application engineering.

- ***Positioning metrics (You could look it up.)***

If the problem being solved by our human specialists is complex and the number of their contributions made on the blackboard begins to grow, quickly locating pertinent information becomes a problem. A specialist should not have to scan the entire blackboard to see if a particular item has been placed on the blackboard by another specialist.

One solution is to subdivide the blackboard into regions, each corresponding to a particular kind of information. This approach is commonly used in blackboard systems, where different levels, planes, or multiple blackboards are used to group related objects.

Similarly, ordering metrics can be used within each region, to sort information numerically,

alphabetically, or by relevance. Advanced blackboard-system frameworks provide sophisticated multidimensional metrics for efficiently locating blackboard objects of interest.

Efficient retrieval is needed to support the use of the blackboard as a group memory for contributions generated by earlier KS executions. An important characteristic of the blackboard approach is the ability to integrate contributions for which relationships would be difficult to specify by the KS writer in advance.

For example, a KS working on one aspect of the problem may put a contribution on the blackboard that does not initially seem relevant or immediately interesting to any other KS. Only until much later, when substantial work on other aspects of the problem has been performed, is there enough context to realize the value of the early contribution. By retaining these contributions on the blackboard, the system can save the results of these early problem-solving efforts, avoiding recomputing them later (when their importance is understood). Additionally, the system can notice when promising contributions placed on the blackboard remain unused by other KSs and possibly choose to focus problem-solving activity on understanding why they did not fit with other contributions.

Locating previously generated contributions of interest is dependent upon the context of other information being used by a KS. This makes a simple pattern-matching specification of the specific contributions difficult and computationally inefficient. Many contributions placed on the blackboard may never prove useful, and maintaining the state of numerous, partially completed patterns is expensive. Therefore, an important characteristic of blackboard systems is enabling an executing KS to quickly and efficiently inspect the blackboard to see if relevant information is present.

The developers of the original Hearsay-II system recognized that rule-like condition specifications of KS interest would be ineffective. Instead, they opted for a combination of simple triggering-condition specifications to be followed by a more detailed procedural examination of the blackboard before activating the KS for execution.

- ***Event-based activation (Is anybody there?)***

In the metaphor, specialists do not interact directly. Each specialist watches the blackboard, looking for an opportunity to contribute to the solution. Such opportunities arise when an event occurs (a change is made to the blackboard) that enables the specialist to act. Blackboard events include the addition of some new information to the blackboard, a change in some existing

information, or the removal of existing information. Some specialists may also respond to external events, such as receiving a telephone call, noticing it is lunch time, and so on.

KSs in blackboard systems are similarly triggered in response to blackboard and external events. Rather than having each KS scan the blackboard (as in the metaphor), each KS informs the blackboard system about the kind of events in which it is interested. The blackboard system records this information and directly considers the KS for activation whenever that kind of event occurs.

- ***Need for control (It's my turn.)***

What if most of the human specialists respond to an event and all rush to the blackboard simultaneously? Some means of ordering their contributions is needed. (A single piece of chalk is a simple control strategy, but one that favors the swiftest rather than the most appropriate specialist.)

A manager, separate from the individual specialists, can be used to restore civility at the blackboard. The manager's job is to consider each specialist's request to approach the blackboard in terms of what the specialist can contribute and the effect that the contribution might have on the developing solution. The manager attempts to keep problem solving on track, to insure that all crucial aspects of the problem are receiving attention, and to balance the stated importance of different specialist's contributions.

Blackboard systems have a similar approach to controlling KSs. A control component that is separate from the individual KSs is responsible for managing the course of problem solving. The control component can be viewed as a specialist in directing problem solving, by considering the overall benefit of the contributions that would be made by triggered KSs. When the currently executing KS activation completes, the control component selects the most appropriate pending KS activation for execution.

Importantly, the control component must be able to make its selection among pending KS executions without possessing the expertise of the individual KSs. Without such a separation, the modularity and independence of KSs would be lost. Therefore, the control component must be able to ask for estimates from triggered KSs in making its control decisions.

When a KS is triggered, the KS uses its expertise to evaluate the quality and importance of its contribution. Each triggered KS informs the control component of the quality and costs associated with its contribution, without actually performing the work to compute the contribution. Instead, each KS generates estimates of the computations that would be generated by using fast, low-cost, approximations developed by the KS writer. These estimates are of the form, "If I am

executed, I'll generate contributions of this type, with these qualities, while expending these resources." The control component uses these estimates to decide how to proceed.

- **Incremental solution generation (Step by step, inch by inch. . .)**

In the metaphor, the solution is generated incrementally as each specialist adds contributions to the blackboard. No single specialist can solve the problem. Instead, specialists refine and extended one another's contributions, building the solution incrementally.

Blackboard systems also operate incrementally. KSs contribute to the solution as appropriate, sometimes refining, sometimes contradicting, and sometimes initiating a new line of reasoning. Blackboard systems are particularly effective when there are many steps toward the solution and many potential paths involving those steps. By opportunistically exploring the paths that are most effective in solving the particular problem, a blackboard system can significantly outperform a problem solver that uses a predetermined approach to generating a solution. Now that we've considered the metaphor in detail, let's restate the blackboard model of problem solving.

2.7.2 The Blackboard Model of Problem Solving

A blackboard system architecture is presented in figure 2.2, and in figure 2.3 we can see that a system based in this architecture scheme consists of three main components. Below we give a thorough explanation for each one of these components.

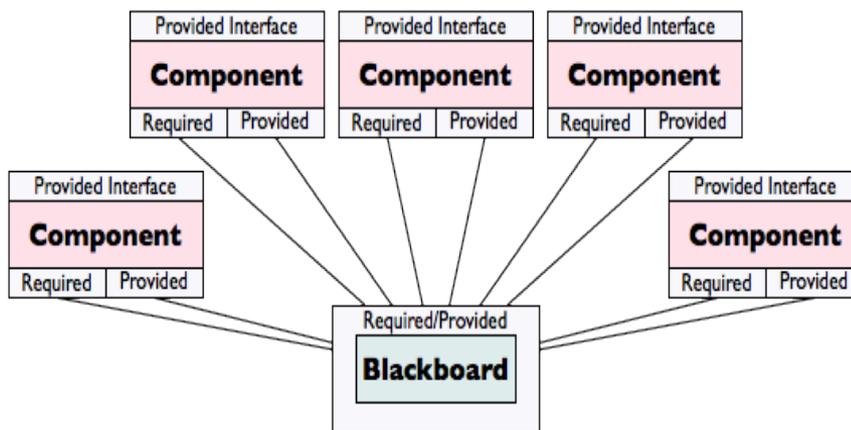


Fig. 2.3 The blackboard architecture scheme

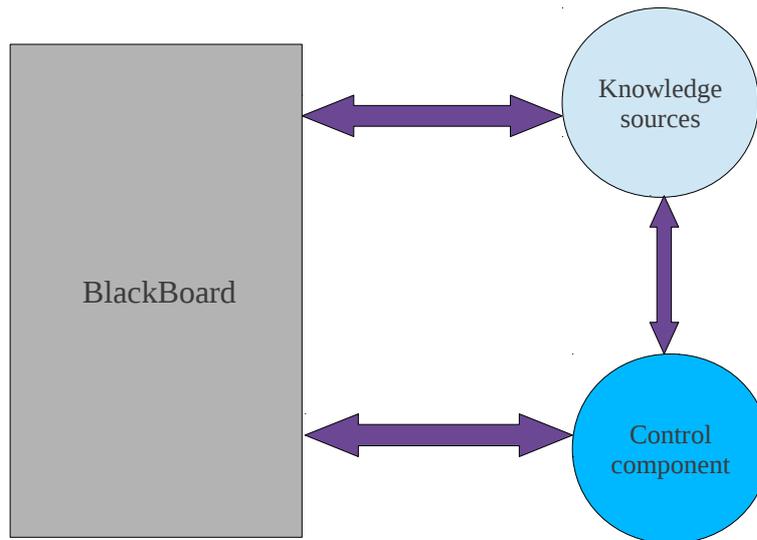


Fig. 2.4 Basic components of the Blackboard Model

- Knowledge sources (KSs) are independent modules that contain the knowledge needed to solve the problem. KSs can be widely diverse in representation and inference techniques.
- The blackboard is a global database containing input data, partial solutions, and other data that are in various problem-solving states.
- A control component makes runtime decisions about the course of problem solving and the expenditure of problem-solving resources. The control component is separate from the individual KSs. In some blackboard systems, the control component itself is implemented using a blackboard approach (involving control KSs and blackboard areas devoted to control).

Knowledge Sources

Each KS is separate and independent of all other KSs. A KS needs no knowledge of the expertise, or even the existence, of the others; however, it must be able to understand the state of the problem-solving process and the representation of relevant information on the blackboard.

Each KS knows the conditions under which it can contribute to the solution and, at appropriate times, attempts to contribute information toward solving the problem. This knowledge that each KS has about when to contribute to the problem-solving process is known as a triggering condition.

KSs are much larger grained than the individual rules used by expert systems. While expert systems work by firing a rule in response to stimuli, a blackboard system works by firing an entire knowledge module, or KS, such as an expert system; a neural net or fuzzy logic routine; or a procedure.

Unlike our metaphor, KSs are not the active agents in a blackboard system. Instead, KS activations (sometimes called KS instances) are the active entities competing for execution resources. A KS activation is the combination of the KS knowledge and a specific triggering context. The distinction between KSs and KS activations is important in applications where numerous events trigger the same KS. In such cases, control decisions involve choosing among particular applications of the same KS knowledge (focusing on the appropriate data context), rather than among different KSs (focusing on the appropriate knowledge to apply). KSs are static repositories of knowledge, KS activations are the active processes.

The blackboard

The blackboard is a global structure that is available to all KSs and serves as:

- a community memory of raw input data; partial solutions, alternatives, and final solutions; and control information
- a communication medium and buffer
- a KS trigger mechanism.

Blackboard applications tend to have elaborate blackboard structures, with multiple levels of analysis or abstraction.

Occasionally, a system containing subsystems that communicate using a global database is incorrectly presented as a blackboard system. (A set of FORTRAN routines using COMMON is an extreme example of this view). True blackboard systems involve closely interacting KSs and a separate control mechanism.

Control component

An explicit control mechanism directs the problem-solving process by allowing KSs to respond

opportunistically to changes on the blackboard database. On the basis of the state of the blackboard and the set of triggered KSs, the control mechanism chooses a course of action.

A blackboard system uses an incremental reasoning style: the solution to the problem is built one step at a time. At each step, the system can:

- execute any triggered KS
- choose a different focus of attention, on the basis of the state of the solution.

Under a typical control approach, the currently executing KS activation generates events as it makes contributions to the blackboard. These events are maintained (and possibly ranked) until the executing KS activation is completed. At that point, the control components use the events to trigger and activate KSs. The KS activations are ranked, and the most appropriate KS activation is selected for execution. This cycle continues until the problem is solved.

Blackboard systems support a variety of control mechanisms and algorithms, so a choice of opportunistic control techniques is available to the application developer.

2.8 The C++ Programming Language

C++ (pronounced cee plus plus) is a general purpose programming language (Stroustrup B. 2000). It has imperative, object-oriented and generic programming features, while also providing the facilities for low level memory manipulation. It is designed with a bias for systems programming (e.g. embedded systems, operating system kernels), with performance, efficiency and flexibility of use as its design requirements. C++ has also been found useful in many other contexts, including desktop applications, servers (e.g. e-commerce, web search, SQL), performance critical applications (e.g. telephone switches, space probes) and entertainment software, such as video-games. It is a compiled language, with implementations of it available on many platforms.

C++ is standardised by the International Organization for Standardization (ISO), which the latest (and current) having being ratified and published by ISO in September 2011. The C++ programming language was initially standardised in 1998. The current standard (C++ 11) supersedes these, with new features and an enlarged standard library. Before standardization, C++ was developed by Bjarne Stroustrup at Bell Labs, starting in 1979, who wanted an efficient flexible language (like C) that also provided high level features for program-organization. Many other

programming languages have been influenced by C++, including C#, Java, and newer versions of C (after 1998).

Chapter 3

Problem Statement

3.1 Autonomous Agent and Multi-Agent systems

An autonomous agent is an intelligent agent operating on an owner's behalf but without any interference of that ownership entity. An intelligent agent, according to a multiply cited statement (*source: wikipedia*) is described as follows:

"Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires."

Such an agent is a system situated in, and part of, a technical or natural environment, which senses any or some status of that environment, and acts on it, over time, in pursuit of its own agenda. Such agenda evolves from drives (or programmed goals). The agent acts to change part of the environment or of its status and influences what it sensed.

Non-biological examples include intelligent agents, autonomous robots, and various software agents, including artificial life agents, and many computer viruses. Biological examples are not yet defined.

A **multi-agent system** is a computerized system composed of multiple interacting intelligent agents within an environment. Multi-agent systems can be used to solve problems that are difficult or impossible for an individual agent to solve. Intelligence may include some methodical, functional, procedural or algorithmic search, find and processing approach.

Although there is considerable overlap, a multi-agent system is not always the same as an agent-based model (ABM). The goal of an ABM is to search for explanatory insight into the

collective behavior of agents (which don't necessarily need to be "intelligent") obeying simple rules, typically in natural systems, rather than in solving specific practical or engineering problems. The terminology of ABM tends to be used more often in the sciences, and Multi Agent Systems in engineering and technology. Topics where multi-agent systems research may deliver an appropriate approach include on-line trading, disaster response, and modeling social structures.

The agents in a multi-agent system have several important characteristics (*source: wikipedia*):

- **Autonomy:** the agents are at least partially independent, self-aware, autonomous
- **Local views:** no agent has a full global view of the system, or the system is too complex for an agent to make practical use of such knowledge
- **Decentralization:** there is no designated controlling agent (or the system is effectively reduced to a monolithic system)

Multi-agent systems, which have also been referred to as "self-organized systems", tend to find the best solution for their problems "without intervention". There is high similarity here to physical phenomena, such as energy minimizing, where physical objects tend to reach the lowest energy possible within the physically constrained world. For example: many of the cars entering a metropolis in the morning will be available for leaving that same metropolis in the evening.

The main feature which is achieved when developing multi-agent systems, if they work, is flexibility, since a multi-agent system can be added to, modified and reconstructed, without the need for detailed rewriting of the application. These systems also tend to be rapidly self-recovering and failure proof, usually due to the heavy redundancy of components and the self managed features, referred to above.

3.2 Developing Autonomous Agent Behaviors

The task of developing an autonomous agent behavior is a tedious one when you try to build it from scratch. One has to deal with many agent-modules that share a lot of diversities when building a software architecture for an autonomous agent. The problem for the user interested only in creating the behavior of the agent can, and should be, handled in a different and abstract way than the handling of the rest of the agent's basic modules, like the agent's hardware configuration (if any exists), or the agent's skills, communication etc.

Thus we can see that in order to focus our effort towards the development of the behavior, we must work on an abstraction layer that is independent of the underlying software and hardware architecture. The KSE tool, provides that layer to the user and propose a way of creating agent behaviors in standardized way exploiting the benefits that ASEME methodology has to offer. The problem with the aforementioned framework is that it is platform dependent and can work only when targeting a specific environment. KSE can generate behaviors for the Monas framework only.

Our Robocup Team Kouretes use statecharts to define the behavior of the Nao robots they use in the competitions they participate. The Kouretes Statechart Editor (KSE) is a tool, developed by our team, that allows the user to design quickly a new robot-behavior or change easily an existing robot-behavior. Unfortunately, when using KSE we cannot test a statechart-based behavior prior to it's use on a real robot, and this takes a significant amount of time and patience in most cases.

The motivation behind the work presented in this thesis is the need for using a simulator when modeling a robotic team behavior. Regularly testing new features on the real robots has several shortcomings, i.e. the robots need maintenance after some hours, a number of people are needed in order to set up an experiment with the robotic team in the lab, and the experiments themselves tend to take quite long to setup and demonstrate. Thus, we decided that we needed to use a simulator for modeling and testing team behavior and when the simulation proved successful, then move to field tests with the real robots. The SimSpark simulation environment, which is also used for the RoboCup 3D soccer simulation league, was the ideal candidate for our goals. However, the SimSpark platform was not compatible with our Monas robotic software architecture, which we use for deploying behaviors on the Nao robots.

3.3 Team Kouretes

Team Kouretes is the RoboCup team of the Technical University of Crete and the only RoboCup SPL team founded in Greece. The team was founded in 2006 and participates in the main RoboCup competition ever since in various leagues (Four-Legged, Standard Platform, MSRS, Webots), as well as in various local RoboCup events (German Open, Mediterranean Open, Iran Open, RC4EW, RomeCup) and RoboCup exhibitions (Athens Digital Week, Micropolis, Schoolfest). Distinctions of the team include: 2nd place in MSRS at RoboCup 2007; 3rd place in SPL-Nao, 1st place in SPL-MSRS, among the top 8 teams in SPL-Webots at RoboCup 2008; 1st place in RomeCup 2009; 6th place in SPL-Webots at RoboCup 2009; 2nd place in SPL at RC4EW

2010; and 2nd place in SPL Open Challenge Competition at RoboCup 2011 (joint team Noxious-Kouretes).

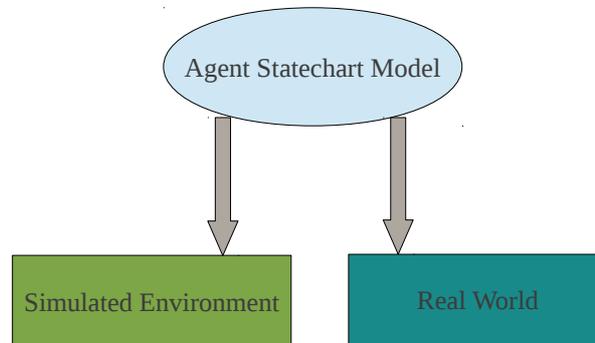


Fig. 3.1 The statechart model should be the same for real world and simulator

In the RoboCup 2012 competition, the team succeeded to proceed to the second round-robin round and rank among the top-16 SPL teams in the world. Recently, the team participated in AutCup 2012, in RoboCup Iran Open 2013, and in the RoboCup 2013 competition in Eindhoven. The members of the team are senior undergraduate and postgraduate ECE students of the Technical University of Crete working on their diploma thesis on RoboCup-related topics. RoboCup offers a great opportunity for research in artificial intelligence (*Kitano H. et.al. 1997*)

Kouretes started developing their own robotic software framework in 2008 and the code is constantly growing and gets maintained ever since. The team's available code repository includes a custom software architecture, a custom communication framework, a graphical application for behavior specification, and modules for object recognition, state estimation, obstacle avoidance, behavior execution, and team coordination.

3.4 Robotics Simulators

A robotics simulator is used to create embedded applications for a robot without depending physically on the actual machine, thus saving cost and time. In some cases, these applications can be transferred on the real robot (or rebuilt) without modifications. The term robotics simulator can refer to several different robotics simulation applications. For example, in mobile robotics applications, behavior-based robotics simulators allow users to create simple worlds of rigid objects

and light sources and to program robots to interact with these worlds. Behavior-based simulation allows for actions that are more biological in nature when compared to simulators that are more binary, or computational. In addition, behavior-based simulators may "learn" from mistakes and are capable of demonstrating the anthropomorphic quality of tenacity. In our case we want to be able to test our Kouretes robotic soccer team behavior in a simulated environment.

One of the most popular applications for robotics simulators is for 3D modeling and rendering of a robot and its environment. This type of robotics software has a simulator that is a virtual robot, which is capable of emulating the motion of an actual robot in a real work envelope. Some robotics simulators, such as Robologix and SimSpark 3D Robotic Simulator use physics engine for more realistic motion generation of the robot. The use of a robotics simulator for development of a robotics control program is highly recommended regardless of whether an actual robot is available or not. The simulator allows for robotics programs to be conveniently written and debugged off-line with the final version of the program tested on an actual robot. Of course, this primarily holds for industrial robotic applications only, since the success of off-line programming depends on how similar the real environment of the robot is to the simulated environment. Sensor-based robot actions are much more difficult to simulate and/or to program off-line, since the robot motion depends on the instantaneous sensor readings in the real world.

3.5 Automatic Programming

In computer science, the term **automatic programming** identifies a type of computer programming in which some mechanism generates a computer program to allow human programmers to write the code at a higher abstraction level.

There has been little agreement on the precise definition of automatic programming, mostly because its meaning has changed over time. David Parnas, tracing the history of "automatic programming" in published research (*Parnas D. 2001*), noted that in the 1940s it described automation of the manual process of punching paper tape. Later it referred to translation of high-level programming languages like Fortran and ALGOL. In fact, one of the earliest programs identifiable as a compiler was called Autocode. Parnas concluded that "automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer."

Generative programming (*source wikipedia*) is a style of computer programming that uses

automated source creation through generic frames, classes, prototypes, templates, aspects, and code generators to improve programmer productivity. It is often related to code-reuse topics such as component or model based software engineering and product family engineering.

Source code generation (*source wikipedia*) is the act of generating source code based on an ontological model such as a template and is accomplished with a programming tool such as a template processor or an integrated development environment (IDE). These tools allow the generation of source code through any of various means. A macro processor, such as the C preprocessor, which replaces patterns in source code according to relatively simple rules, is a simple form of source code generator.

Considering the above, we can see that in order to be able to develop agent behaviors for a variety of applications using an abstract methodology like ASEME, and test them in a variety of platforms and environments like real robots or a robotics simulators; we must have a tool that generates code for an agent behavior, based on a standard methodology, and also hasn't got any dependencies with the underlying framework it targets. Thus we need a source code generator for agent behaviors that can be configured during runtime in order to target any user specified framework.

3.6 Related Work

Here we give examples of already implemented software tools that try to solve problems closely related to ours. As we will see there are many CASE tools that provide the user with the capability of generating source code for an agent behavior. Here we will give a brief presentation for XABSL and Yakindu.

XABSL

The *Extensible Agent Behavior Specification Language* (XABSL, <http://www.xabsl.de/>) is a very simple language to describe behaviors for autonomous agents based on hierarchical finite state machines. XABSL was developed to design the behavior of soccer robots. Behaviors specified in XABSL proved to be very successful during Robocup since 2004. The German Team has won the competitions in the Standard platform league (using Sony Aibo robots) in 2004, 2005, and 2008. The Darmstadt Dribblers have won the Humanoid Kid Size competition in 2009 and 2010. However, the usage of the language is not restricted to robotic soccer. XABSL is a good choice to

describe behaviors for all kinds of autonomous robots or virtual agents like characters in computer games.

In order to start using XABSL tools for developing your agent behavior you only need three things:

- A text editor of your choice
- The XABSL-Compiler (ruby-based)
- The XabslEngine (C++ or Java library)

The behavior is described by a set of *xabsl* files. These have to be compiled to an intermediate code using the XABSL-Compiler. At the start-up of the agent this intermediate code is read by the XabslEngine which executes the behaviors during run-time. In XABSL complex behaviors are described as hierarchical finite state machines.

To use XABSL, you have to know four concepts: Agents, options, states and decision trees. In XASBL, an *agent* consists of a number of behavior modules called *options*. The options are ordered in a rooted directed acyclic graph, the *option graph*. Moreover options are behavior modules which make up the hierarchical decomposition of the complex agent behavior. Lower hierarchy levels consist of primitives behaviors which are composed into more complex behavior options. Each option is a finite state machine. The states of an option define the actions that are active. The actions of a state can reference other options, thus allowing the decomposition of a task into primitive options. Decision trees are responsible for the definition of the state transitions. Decision trees can reference *input symbols* in order to access input data such as the agent's world state or sensory data.

Yakindu

Yakindu Statechart Tools (SCT, <http://statecharts.org/>) is an open source tool for the specification and development of reactive, event-driven systems with the help of state machines. It consists of an easy-to-use tool for graphical editing and provides validation, simulation and code generators for different target platforms. The users come from both the industrial and academic sectors.

The first version of Yakindu Statechart Tools was released in 2008 as part of the research project MDA for Embedded systems. In this research project, model-based development processes for the development of embedded systems based on the Eclipse project were developed. Since mid-

2010 the Yakindu-Team has been working on Version 2.0. The first official version was released together with Eclipse version Juno.

Yakindu (*Figure 3.2*) is a free toolkit for the model driven development of embedded systems. Through the systematic use of models, it aims at an integrated development process as well as an increase in quality and maintainability. The Yakindu toolkit supports the development of both reactive, event-driven and data flow-oriented systems with the help of statecharts and block diagrams. The continuous support begins with graphical modeling tools, includes integrated validation and simulation, that allows for the early assessment of the models and offers efficient code-generators for the generation of source code for a target platform. Technologically, it is based on Eclipse-platform and integrates itself seamlessly into Eclipse-based workbenches and extends this in the direction of model-driven development. The main features of Yakindu Statechart tools are:

- smart combination of textual and graphical modeling
- syntactic and semantic validation of the state machines
- executable models via the simulation engine
- code generators for Java C and C++

The Yakindu toolkit allows to design embedded systems by using both statecharts and block-diagrams . The Yakindu Statechart Tools (SCT) allow graphical modeling based on Harel statecharts. They support all essential concepts like extended state variables, hierarchical states, orthogonal states (also known as And-States or parallel regions) or History-States. This corresponds to the concepts that are used in modelling languages such as UML. The convenient model-editor integrates features such as model validation and simulation as well as the generation of source code. Below we see a screen-shot from the environment of Yakindu.

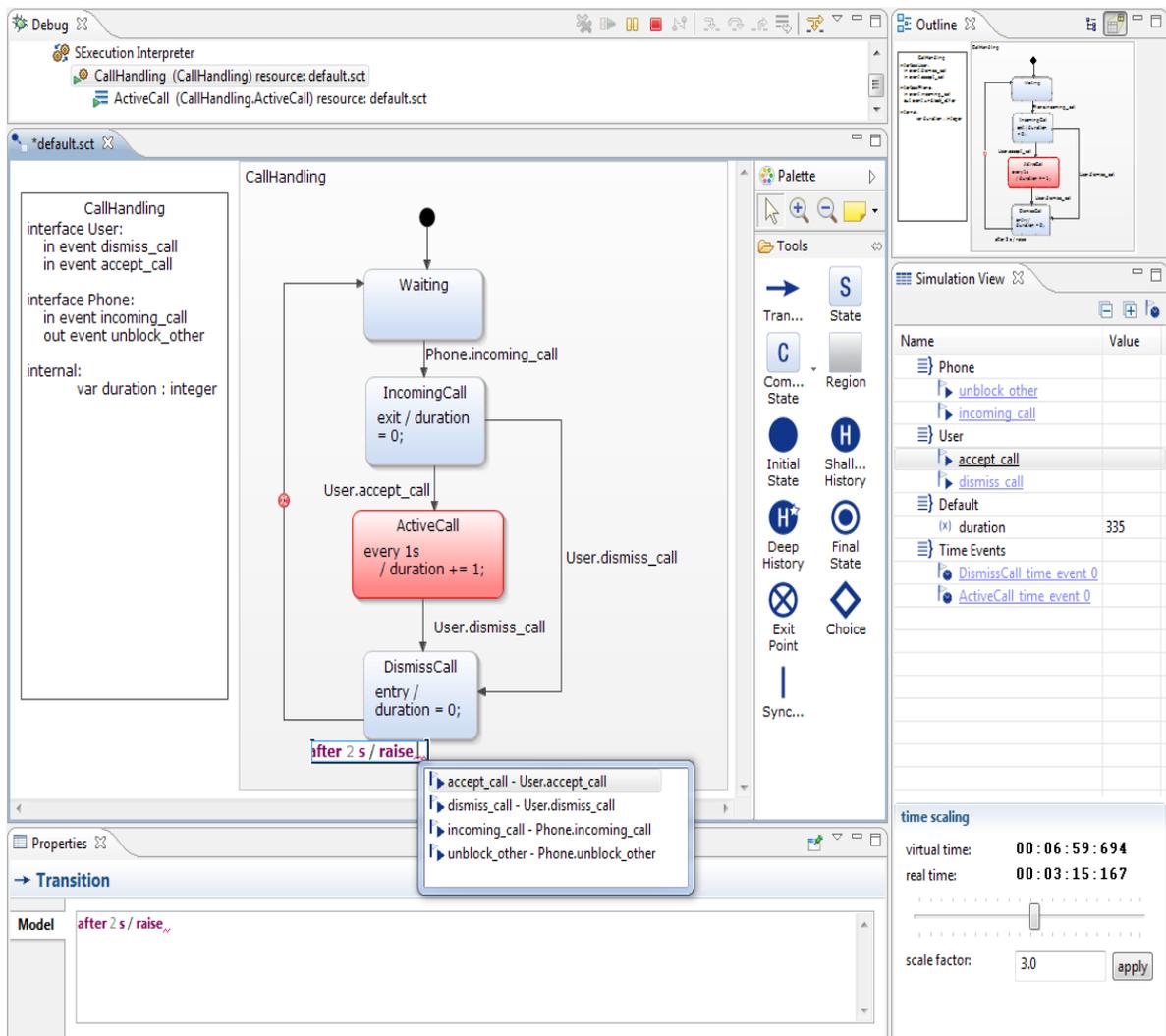


Fig. 3.2 The Yakindu environment

Chapter 4

Our Approach

As we mentioned in chapter 3, our Robocup Team Kouretes use KSE and statecharts to define the behavior of the Nao robots they use in the competitions they participate (*Topalidou-Kyniazopoulou et. al. 2013*). We also mentioned that the KSE tool, developed by our team, is used to design quickly a new robot-behavior or change easily an existing robot-behavior. Unfortunately, we are bound to use the real Nao robots to test the statechart-based behaviors that we produce when using KSE. This takes a significant amount of time and patience in most cases.

To avoid these problems we want to be able to test our statechart-based behaviors in a simulated environment prior to their use in a real robot. Various advantages arise when using a simulator for testing an agent behavior; the number of experiments we can try in a simulator can be enormous in contrast to the real world and this is crucial for machine learning algorithms. Also, we avoid any hardware issues that may occur when using a real robot. This means that we want be able to generate code for the simulation environment and for the real world based on the same statechart model.

Thus, we decided to develop a platform-independent code generation component for the KSE tool. To this end, we added a number of platform-specific parameterization features at the code generation tool. Adopting this parametric approach and provided the correct parameters of the underlying platform, we can now use the KSE tool for deploying platform-independent agents on any platform by exporting the generated code directly in the C++ programming language using the specification provided by the parameters.

4.1 The KSE Generic C++ Generator

The Kouretes Statechart Editor (KSE) is a tool which enables the developer to easily specify a desired agent behavior as a statechart model utilizing a variety of base robot functionalities. 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. This tool is build for developing behaviors for the Monas robot C++ architecture.

The KSE Generic C++ Generator is a tool that extends KSE in order to give the developer the ability to create behaviors for agents, independent of their architecture and their environment. By this we mean that the KSE GGenerator enables the user to create agent behaviors for different platforms and different environments, with the only prerequisite being C++. Also, the old KSE provides us with the functionality of automatic transformation of liveness formulas to statechart model (Text-to-Model (or T2M), transformation). As long as we are done with the T2M ransformation we proceed to the next phase of development. We edit the model produced and we generate code for it (Model-to-Text (or M2T), transformation). In this phase we cannot use the M2T transformation provided by KSE, since it's implemented to generate code only for the Monas architecture.

We meet the above demands by extending the statechart engine used on KSE with the addition of a generic blackboard interface, a new transition expression grammar for the engine and the creation of a new source code generator for our tool (a new Model-to-Text transformation). The blackboard interface is generated every time along with our statechart model source code by the KSE GGenerator during the M2T transformation and can target any C++ platform according to it's initialization. This interface along with our generator are integrated on KSE. By using the KSE GGenerator the user is able to create agent behaviors for any C++ framework. In figure 4.1 we show an abstract scheme of our GGenerator as a new module to the old KSE tool.

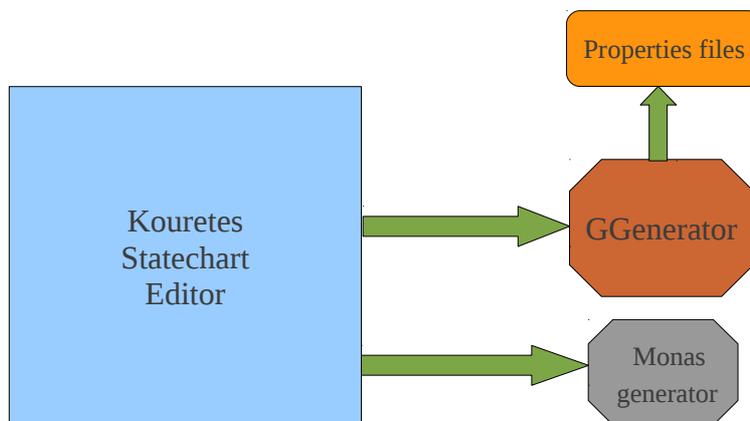


Fig. 4.1 Our GGenerator as a new module to the old KSE

4.2 Generic Blackboard

We add a generic blackboard interface to the statechart engine used on KSE, to be able to *bind* our statechart model with any user-specified framework. In order to achieve this the blackboard interface is created during the source code generation process and targets a specific environment according to its initialization and the parameters given. This way the user is not concerned about model-framework compatibility issues because the created blackboard interface acts as a middleware between the statechart model and the targeted environment and handles their *communication*.

The initialization of the blackboard interface depends on the *properties files*. The *properties files* are defined by the user during the early stages of the behavior development process. These are the *include_classes.txt* file and the *instances.txt* file. Both files are in text format and they are used by the KSE GGenerator tool. These files are responsible for generating a blackboard interface that targets a specific framework. In these files we add functions, variables and header files that are needed for the *communication* between our statechart model and the targeted environment. In the first one we add the header files we want to include in our interface, and in the second one we add the functions and the variable instances we need.

The creation of the properties files requires some knowledge about the framework we target. Before our blackboard interface is generated we have to point out the framework's modules that provides us with the information needed for the right communication between our statechart model and the environment. In most cases the information needed for creating an agent behavior, is the updated state of the targeted environment, or an update on the agent's sensor values, although we can use any kind of information the framework provides. Once we are aware of the framework's modules that provide the information we need we can easily create the properties files based on these modules. In chapter 5 we give three examples of how to create the properties files for three different platforms and we show the blackboard interfaces they generate for each one of them along with the properties files.

In our approach we also give the ability to the user to register variables in the blackboard interface by using the editing tool provided by KSE. This is extremely important because most of the times when we develop an agent behavior, we need many variables that a framework may not provide. These variables can be of any type and can be used in transition expressions along with the variables defined in the *properties files*. This combination helps us to make our system generic and also to avoid the tedious work of dealing with code every time we need to add a simple variable to

our statechart model. Also, this helps us for the inter-agent communication when we have more than one agents.

Finally when we work on a specific platform we need to register our generic blackboard interface along with our statechart model on it. This has to be done in the last stage of our behavior development after we have finished with the *properties files* creation, and the code generation. In this last stage we instantiate our model on the platform and we are ready to test our behavior. In figure 4.2 we can see an abstract scheme that shows the use of blackboard interface as a middleware between our statechart engine and the targeted framework, and in figure 4.3 we can see the software architecture of our system.

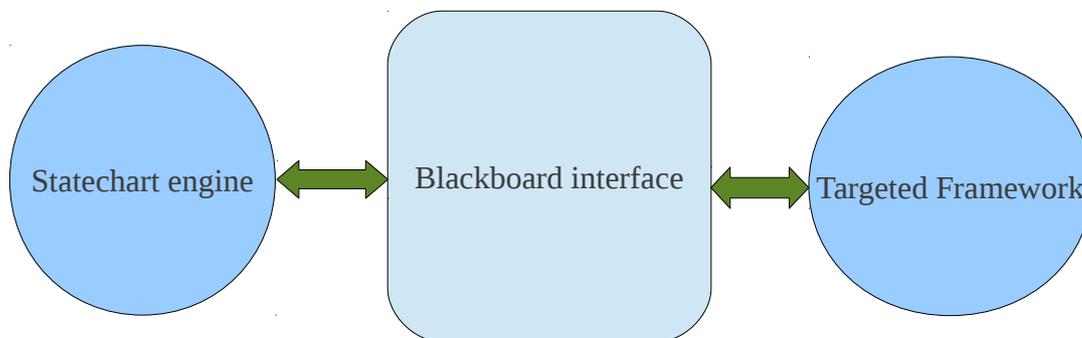


Fig 4.2 Use of our generic Blackboard interface as a middleware

4.3 Transition Expression and C++ Generator

Since we changed the statechart engine used on KSE by adding a generic blackboard interface, we also need a different C++ generator from the one the old KSE tool used. Also the new statechart engine supports a different syntax for the transition expressions. Thus we create a new C++ generator for our extended statechart engine using the Eclipse Modeling Framework along with Xpand. We also create a new grammar for the transition expressions used in the statechart. The grammar is shown in EBNF format in Fig. 4.3.

Every statechart model we create to describe an agent behavior, contains transition expressions that are responsible for the right execution of the statechart. These expressions are inserted manually by the user during behavior specification using the GGenerator tool and following the syntax shown in figure 4.3, for controlling the statechart execution. In a transition expression the user should be able to, optionally, define *events* and *conditions*, as well as multiple

actions if desired. Moreover in the transition expressions, the user should be able to use framework specific variables or user defined variables as *events conditions* and *actions*, for controlling the statechart execution. Either way these variables should be registered in the blackboard's interface in order for the transition expressions to work properly. In the first case we have the *properties files* discussed above to do the work for us. In the second case we use KSE to add any user defined variables we need for the expressions. It is very important to notice that we provide the user with the ability to create any kind of variables he finds necessary for the transition expressions during the graphical editing of the statechart model.

As we can see the transition expressions are a vital part of our statechart engine. This means that it should be relatively easy for a user to edit them, even without having a good knowledge of the underlying framework he works with. To meet these demands, we created a user-friendly grammar for our transition expressions, which is independent of the underlying platform, and uses common syntactic rules for expressions. In the examples that we present in the next section we will see that, although we create behaviors for two different environments, we use the same syntax for the transition expressions for all the statechart models that we create.

```

transition expression = [event] ["[" condition "]" ] [/actions]
event = string
condition = condExpr | condition operator condExpr
condExpr = variable operator variable
variable = string | varString
varString = (letter+) string | (letter+) string "." varString
actions = TimeoutAction | Action
TimeoutAction = TimeoutAction "." letter+ "." digit_list
Action = variable actionOp variable | Action ";" Action
string = letter_or_digit | letter_or_digit string
letter_or_digit = letter | digit
letter = "a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | ...
digit_list = digit | digit digit_list
operator = ">" | "<" | ">=" | "<=" | "==" | "&&" | "&" | "||" | "|" | "!="
actionOp = "="
digit = "0" | "1" | "2" | "3" | ...

```

Fig. 4.3 Transition expression grammar in EBNF format

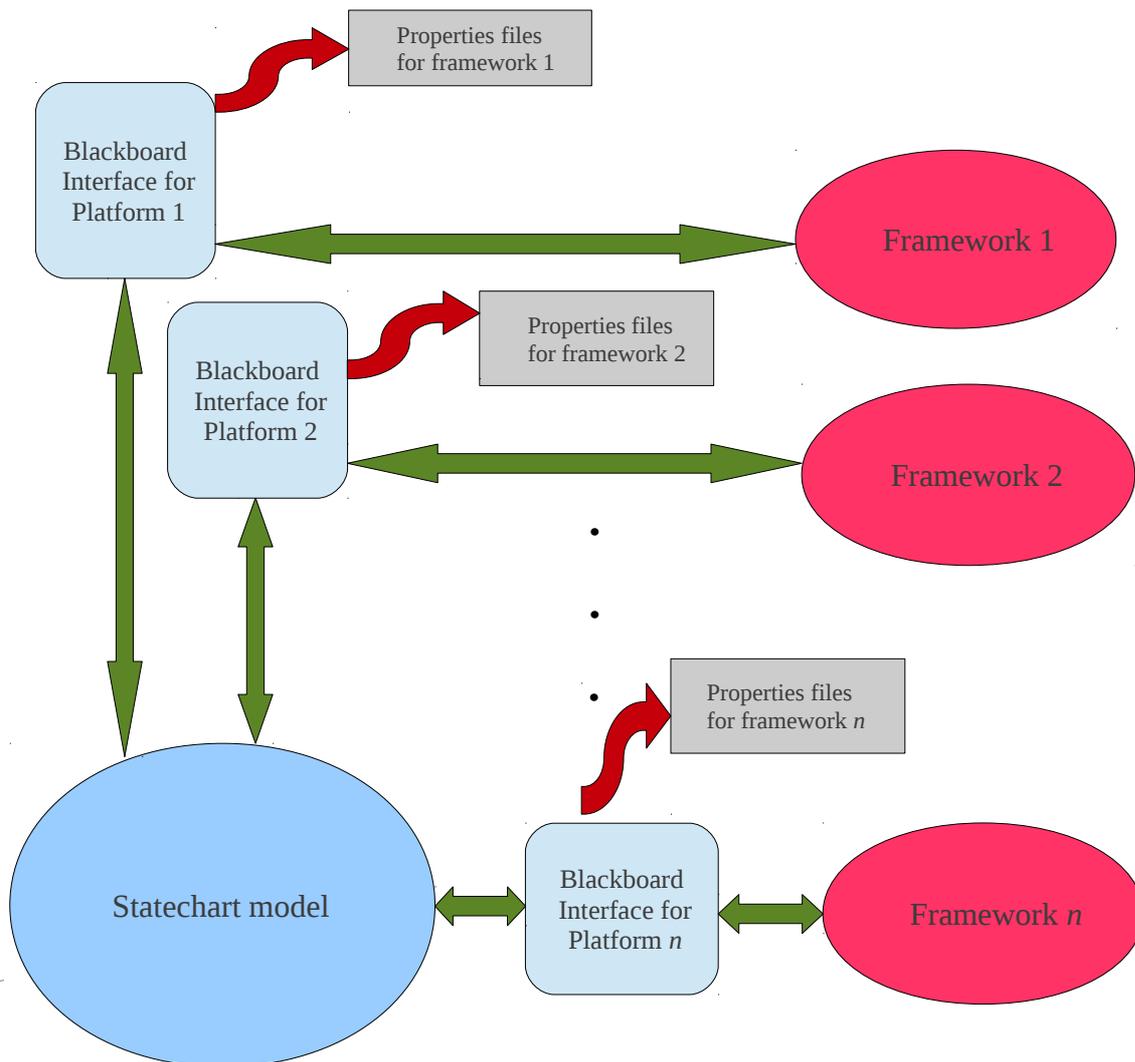


Fig. 4.4 The scheme of our software architecture for different frameworks

4.4 Transition Expression Example

In order to present the files and the code that is produced for a full transition expression according to our grammar rules we use a simple example. Below in figure 4.6 we show a statechart model which describes the behavior for a surveillance camera. The behavior we describe is simple: *Based on the transition expression shown in the statechart, the camera will choose when to scan the area. [event1=Night, x=systemOn, y=lightsOn, action1=triggerAlarm]*

Transition expression =event1[(x==true)&&(y==false)]/action1=false

```
#include "../IEvent.h"
class TrEvent_0_2_2TOScan : public
statechart_engine::IEvent {
public:
TrEvent_0_2_2TOScan(string str):IEvent(str) {;}
};
```

Fig. 4.5 Code for the event part of the transition expression

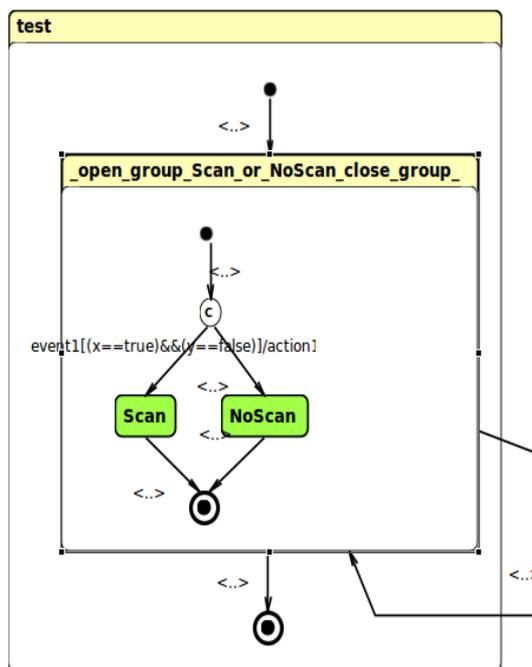


Fig. 4.6 Statechart model for the surveillance camera example

```
#include "../IAction.h"
class TrAction_test0_2_2 : public statechart_engine::
IAction {
public:
void UserInit () {}
int Execute()
{
this->_blk->action1=false;
return 0;
}
};
```

Fig. 4.7 Code for the action part of the transition expression

4.5 KSE Integration

As already mentioned above the KSE tool was used for creating behaviors for the Monas robot architecture. The tool is able to generate code for Monas based on a statechart model defined

by the user. The definition of the statechart model is based on liveness formulas. The liveness formulas are submitted manually by the user to KSE and they describe the agent behavior we want to create.

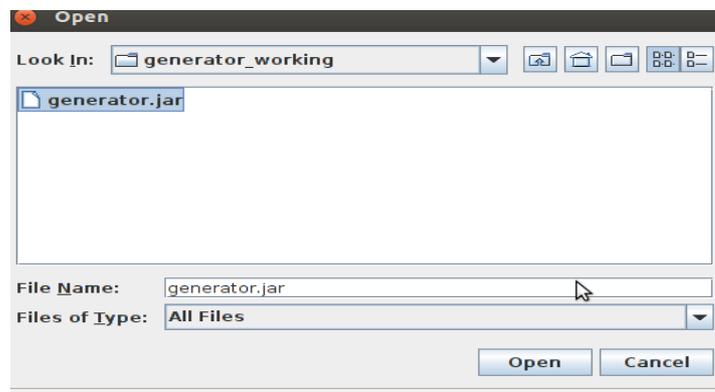
```
#include "../ICondition.h"
#include "../BlackBoard.h"

class TrCond_test0_2_20_2_3 : public statechart_engine::ICondition
{
public:
void UserInit () { }
bool Eval() {
if((this->_blk->x==true)&&(this->_blk->y==false))
{ return true; }
else{ return false; }}
};
```

Fig. 4.8 Code for the condition part of transition expression

In our extended KSE Generic C++ Generator we keep using the same process for creating an agent behavior, but we have made serious changes to the C++ generator of the tool, in order to be able to generate code for frameworks beside Monas. Since we wanted to use the same graphical interface provided by the old KSE for creating our agent behaviors we integrated our work on it. In figure 4.1 of section 4.2.1, we present an abstract model that shows how we registered our GGenerator as a new module to the old KSE. In this section we present the details of this procedure. We use an option provided by the old KSE tool in order to guide the KSE to use another generator for the creation of the statechart's source code. After this operation the new KSE GGenerator is ready to work as described above, and the source code that generates can work directly on a user-specified framework. In chapter 5 (Results), we give two examples of agent behavior creation using the KSE Generic C++ Generator. For creating a new generator for KSE we used Eclipse Modeling Framework along with Java and Xpand.



Fig. 4.9 Initialize KSE with a new code generator**Fig. 4.10** Selecting the GGenerator

4.6 Implementation

Until now we have shown in an abstract form our software architecture and the way we use it. In this section we present technical aspects of our software development. First we show the creation of our generic blackboard interface, next we present some of the templates we created in order to generate code using Xpand, and at last we show the created workflow that invokes our generator using Eclipse Modeling Tools.

4.6.1 Creating a Generic Blackboard Interface

In order to be able to *connect* our statechart engine with a user defined framework we decided to create a blackboard interface. Since our interface must not have any dependencies with underlying framework, we created a C++ class called *BlackBoard* to serve that cause. Prior to the user's configuration (properties files), the *BlackBoard* class holds information that is needed only in the statechart's mechanics(*e.g the timeouts mechanism, events mechanism*). So our effort was towards giving the ability to our tool to register to the *BlackBoard* class any kind of information needed during the code generation phase. We should point out that the user is obligated to update this information using the modules provided by the underlying framework for this cause. In figure 4.11 we show the *BlackBoard* interface class prior to the user's configuration and afterwards, for the simple surveillance camera example presented above, while in figure 4.12 we show the *BlackBoard* interface class after the code generation.

```

#include <iostream>
#include <string>
#include <list>
#include "MessageHub.h"
#include "TimeStamp.h"
using namespace std;
using namespace statechart_engine;

class BlackBoard
{
private:
    string name;
    string event; //current event
public:

    list<Time_stamp> *listTimeStamps;
    BlackBoard(){}
    BlackBoard(const string& str){name=str;}
    virtual ~BlackBoard(){}

    /** gives access to the instance of the blackboard */
    static BlackBoard& getInstance()
    {
        /** We have ONLY one instance of the blackboard */
        static BlackBoard instance;
        return instance;
    }

    bool checkEvent(string str)
    {
        if(event==str)&& str!="GameOver"
            {
                return true;
            }else
                return false;
    }
}

```

```

//sets blackboard name
void setBlackBoardName(const string& str)
{
    name=str;
}

//prints blackboard name
void printBlackBoardName()
{
    cout<<"BlackBoard name is "<<name<<endl;
}

//clearing triggered timeouts
void clearPassedTimeouts()
{
    list<Time_stamp>::iterator list_counter;
    for(list_counter=listTimeStamps->begin();list_counter!
    =listTimeStamps->end();++list_counter)
    {
        //if timeout is true => timeout is already
        triggered and has to be removed
        if(list_counter->getTrigger()==true)
        {
            listTimeStamps-
            >erase(list_counter);
        }
    }
} //closing function

#endif /* BlackBoard_H */

```

1 2

Fig. 4.11 The BlackBoard interface class prior to it's configuration

As we can see the skeleton code of the *BlackBoard interface* is remains intact during the code generation, and the only change that happens to the *BlackBoard* class is the registration of some new framework related variables needed for our behavior in order to work on the targeted environment. The new variable that comes from the properties files is an object called *worldInfo* which is of type *CameraInfo*. The *CameraInfo* is a class that belongs to the camera's environment and is included in the interface according to the properties files configuration. The boolean variables *x* and *y*, were registered during the editing of the statechart using the KSE editing tool. We demonstrate more thoroughly the use of our tool and the creation of properties files for a specific environment in chapter 5.

```

#ifndef BlackBoard_H
#define BlackBoard_H
#include "CameraInfo.h" //added using properties
files
#include <iostream>
#include <string>
#include <list>
#include "MessageHub.h"
#include "TimeStamp.h"

using namespace std;
using namespace statechart_engine;

class BlackBoard
{
private:
    string name;
    string event; //current event
public:

    list<Time_stamp> *listTimeStamps;
    bool x; //added using properties files
    bool y; //added using properties files
    CameraInfo worldInfo; //added using KSE editor

    BlackBoard(){}
    BlackBoard(const string& str){name=str;}
    virtual ~BlackBoard(){}

    /** gives access to the instance of the blackboard */
    static BlackBoard& getInstance()
    {
        /** the ONLY instance of the blackboard */
        static BlackBoard instance;
        return instance;
    }

    void setPlayMode(string str)
    {
        playMode=str;
    }

    string getPlayMode()
    {
        return playMode;
    }
}
1

bool checkEvent(string str)
{
    if(event==str)//&& str!="GameOver"
    {
        return true;
    }else
    {
        return false;
    }

    //sets blackboard name
    void setBlackBoardName(const string& str)
    {
        name=str;
    }

    //prints blackboard name
    void printBlackBoardName()
    {
        cout<<"BlackBoard name is "<<name<<endl;
    }

    //clearing triggered timeouts
    void clearPassedTimeouts()
    {
        list<Time_stamp>::iterator list_counter;
        for(list_counter=listTimeStamps->begin();list_counter!
        =listTimeStamps->end();++list_counter)
        {
            //if timeout is true => timeout is already
            triggered and has to be removed
            if(list_counter->getTrigger()==true)
            {
                listTimeStamps-
                >erase(list_counter);
            }
        }
    }

    //closing function

    //for use in spl only
    void attachTo(MessageHub* hub)
    {
        //implement for spl
    }
};

#endif /* BlackBoard_H */
2

```

Fig. 4.12 The BlackBoard interface class after the configuration and the code generation

4.6.2 Why Use the Blackboard Problem-Solving Approach?

The blackboard model offers a powerful problem-solving architecture that is suitable in the following situations.

- Many diverse, specialized knowledge representations are needed. KSs can be developed in the most appropriate representation for the data they are to handle. For example, one KS might be most naturally written as a rule-based system while another might be written as a

neural-net or fuzzy-logic routine.

- An integration framework is needed that allows for heterogeneous problem-solving representations and expertise. For example, a blackboard is an excellent framework for combining several separately established diagnostic systems.
- The development of an application involves numerous developers. The modularity and independence provided by large-grained KSs in blackboard systems allows each KS to be developed and tested separately. The software-engineering benefits of this approach apply during design, implementation, testing, and maintenance of the application.
- Uncertain knowledge or limited data inhibits absolute determination of a solution. The incremental approach of the blackboard system will still allow progress to be made.
- Multilevel reasoning or flexible, dynamic control of problem-solving activities is required in an application.

The blackboard approach has been applied in numerous areas, including the following (*source: Wikipedia, Hayes-Roth, B. 1985*):

- | | |
|-------------------------------|---------------------------|
| • knowledge-based simulation | • symbolic learning |
| • sensory interpretation | • planning and scheduling |
| • knowledge-based instruction | • data fusion |
| • design and layout | • computer vision |
| • command and control | • case-based reasoning |
| • process control | |

In each of these applications, the scope of the problem to be solved was the prime factor in selecting a blackboard approach. That is, deciding whether to use a blackboard approach should be based on the problem-solving requirements discussed above, rather than the specific application area.

4.6.3 Developing our Generator using Xpand and Workflows

In this section we present some of the Xpand templates we created during the development of our GGenerator. The templates we show are the condition template along with our *BlackBoard interface* template. Finally we present the workflow we created in order to invoke our GGenerator

and make a stand alone application.

<pre> «FILE "BlackBoard.h"-> #ifndef BlackBoard_H #define BlackBoard_H «ReadHeaderProperties()» #include <iostream> #include <string> #include <list> #include "MessageHub.h" #include "TimeStamp.h" using namespace std; using namespace statechart_engine; class BlackBoard { private: string name; string playMode; //gameMode public: list<Time_stamp> *listTimeStamps; «ReadClassProperties()» </pre> <p style="text-align: center;">1</p>	<pre> «IF !variables.isEmpty -> «DeclareVariables(variables)» «ENDIF-> BlackBoard(){ BlackBoard(const string& str) {name=str;} virtual ~BlackBoard(){ /** gives access to the instance of the blackboard */ static BlackBoard& getInstance() { /** the ONLY instance of the blackboard */ static BlackBoard instance; return instance; } </pre> <p style="text-align: center;">2</p>	<pre> void setPlayMode(string str) { playMode=str; } string getPlayMode() { return playMode; } bool checkEvent(string str) { if(playMode! ="GameOver")//==str//&& str! ="GameOver" { return true; }else return false; } </pre> <p style="text-align: center;">3</p>
--	--	--

Fig. 4.13 BlackBoard interface Xpand template (pages 1-3)

```

//sets blackboard name
void setBlackBoardName(const string& str)
{ name=str; }
//prints blackboard name
void printBlackBoardName()
{ cout<<"BlackBoard name is "<<name<<endl; }
//clearing triggered timeouts
void clearPassedTimeouts()
{
    list<Time_stamp>::iterator list_counter;
    for(list_counter=listTimeStamps->begin();list_counter!=listTimeStamps->end();++list_counter)
    {
        //if timeout is true => timeout is already triggered and has to be removed
        if(list_counter->getTrigger()==true)
        { listTimeStamps->erase(list_counter); }
    }
}
//closing function
//for use in spl only
void attachTo(MessageHub* hub)
{ //implement for spl }
};
#endif /* BlackBoard_H */
«ENDFILE»

```

4

Fig. 4.14 BlackBoard interface Xpand template (page 4)

```

«IMPORT IAC»
«EXTENSION JavaHelpers::NodeHelper»
«EXTENSION JavaHelpers::TransExpr»
«DEFINE Condition(String modelName) FOR IAC::Transition»
    «IF HasCondition(TE)-»
        «FILE TransitionName(modelname+source.label+target.label)+".h" transitions_outlet»
#include "../ICondition.h"
#include "../BlackBoard.h"
// «name»
class «ConditionName(modelname+source.label+target.label)» : public statechart_engine::ICondition {
public:    void UserInit () { }
        bool Eval() {
            /* «getConditionOfExpression(TE)» */
«getNaothConditionExpression(TE)»          }
};
        «ENDFILE»
        ICondition* «ConditionNameInst(modelname+source.label+target.label)» = new
«ConditionName(modelname+source.label+target.label)»;
        _conditions.push_back( «ConditionNameInst(modelname+source.label+target.label)» );
«ENDIF-»
«ENDDEFINE»

```

Fig. 4.15 Condition Xpand template

```

<?xml version="1.0"?>
<workflow>
    <property name="model" value="src/Models/StateChartExample_new.iac" />
    <property name="src-gen" value="src-gen" />
    <property name="statechart" value="src-gen" />
    <property name="activities" value="src-gen/activities" />
    <property name="transitions" value="src-gen/transitions" />
    <!-- set up EMF for standalone execution -->
    <!--<bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
        <platformUri value=".." />
    </bean-->
    <!--RegisterEcoreFile value="platform:/resource/IAC_EMF/metamodel/IAC.ecore"/-->
    <!-- load model and store it in slot 'model' -->
    <component class="org.eclipse.emf.mwe.utils.Reader">
        <uri value="{model}" />
        <modelSlot value="model" />
        <firstElementOnly value="false" />
    </component>
    <component class="org.eclipse.xpand2.Generator">
        <metaModel id="mm" class="org.eclipse.xtend.typesystem.emf.EmfMetaModel" >
            <metaModelPackage value="IAC.IACPackage"/>
        </metaModel>
        <expand value="mainTemplate::model FOR model"/>
        <outlet path= "{statechart}" append="true" />
        <outlet path= "{activities}" name="activities_outlet" />
        <outlet path= "{transitions}" name="transitions_outlet" append="true"/>
        <beautifier class="org.eclipse.xpand2.output.JavaBeautifier"/>
    </component> </workflow>

```

Fig. 4.16 The workflow file for our GGenerator

4.7 Editing the Statechart Engine

We should point out that we made some enhancements in the statechart engine that the old KSE was using. Firstly, as we mentioned before, we added a blackboard interface to the statechart engine and we created the syntax used for the transition expressions. Secondly, we changed the code for the *events*.

In the Monas implementation the statechart checks for valid transitions starting from the state that is active in the lowest level (activity) when the statechart configuration is about to change. Moreover, it did not explicitly support events. This way, events were checked as conditions in the transition expressions. Our work added the support for events, and the statechart checks for valid transitions with respect to events starting from the highest level states when the statechart configuration is about to change.

4.8 Summary

In this chapter we tried to explain our idea of using a generic blackboard interface in order to achieve a connection between a statechart-based autonomous agent behavior and a user defined framework. To do so, we expanded our statechart engine with a new class called BlackBoard and we gave the user the ability to utilize this class according to the environment he targets. We should also mention that we do not only achieve communication (between our behavior and the targeted framework) via the blackboard interface, but synchronization also. As long as the user utilizes the class properly using the properties files, he can then generate and edit behaviors for the platform he works with.

Closing, we present all sorts of technical aspects concerning our implementation by showing parts of the code that our GGenerator generates, the Xpand templates that are used for this cause the workflow that invokes our new generator, and the changes we made in the statechart engine. We also show the connection of our GGenerator with the old KSE and how this extension solves our problem.

Chapter 5

Results

In this chapter we are going to test our approach using three different platforms to work with as an attempt to provide objective results. Since our main subject is to show that our KSE C++ GGenerator is platform independent we use three diverse frameworks for our examples. The only thing that the three platforms have in common is that they are both implemented in C++.

In the first example we work with SimSpark 3D Robotic Soccer Simulator , and we develop an agent behavior for it. In the second example we work with Wumpus World Simulator, and we create a behavior for it. Moreover we create a team of agents for the SimSpark in order to demonstrate a cooperative scenario between the agents. At last, we develop a behavior for a very famous video game; the Starcraft Brood War strategy game. In all these cases we use the KSE C++ GGenerator that exploits the benefits of ASEME to develop our behaviors. The procedure we follow for generating our behaviors is identical for all the platforms although they simulate completely different environments. In the first one we have a 3D world with noisy sensor values, in the second one we have a two-dimensional environment with no noise added in the perception of our agent, and in the third we have a 3D dynamic environment with multiple agents and no noise. Below we give a brief explanation of the three platforms we use and their operation. Firstly, we present the SimSpark 3D Robotic Soccer Simulator, secondly we present the Wumpus World Simulator, and finally the StarCraft game platform. In a next section we discuss more thoroughly about the major differences among these platforms.

As we discussed in *section 4* our purpose is to work on the high level of agent programming. By this we mean that we want to focus on developing the “brain” of the agent without having to deal with low-level programming like the agent's basic actions or agent-simulator communication issues (e.g agent-SimSpark communication). The two frameworks we present below are utilized in order to meet these demands. They both provide a number of agent basic activities for the simulator they target, and they both handle the communication between the agent and the simulator, giving us

the freedom to implement only the behavior of the agent.

Finally we demonstrate the development process of creating agent behaviors for these specific environments using the KSE C++ Generic Generator, and at last we present results from the use of our created behavior for the two simulation worlds.

5.1 Creating a Behavior for the SimSpark 3D Robotic Soccer Simulator

5.1.1 SimSpark 3D Robotic Soccer Simulator

Platform overview

The overall system consists of the Soccer Server and the agents, i.e. the player programs. The Soccer Server simulates the physical world: The playground, the ball and the bodies of the players according to the laws of physics. As parts of the body, the sensors and effectors of the players are simulated by the Soccer Server as well.

An agent is the “brain” of a player. It is an autonomous program to control the simulated body. The implementation of agents is explained separately. The interaction between the Soccer Server and an agent is performed by messages which contain the sensations and action commands, respectively. The system works cyclically with basic cycles of 20 msec:

- 1. The server sends individual server messages with sensations to the agents.*
- 2. The agents can decide for new actions depending on their beliefs about the situation.*
- 3. The agents can send their agent messages to the server for desired actions.*
- 4. The server collects the agents messages and calculates the resulting new situation (poses of the players, ball movement etc.) according to the laws of physics and the rules of the game.*

5.1.2 Simulation using SimSpark: The Soccer Server and the Monitor Soccer Server

The Soccer Server simulates the physical world for simulated soccer. It is based on SimSpark, a generic physical multi agent simulator system for agents in three-dimensional environments (<http://simspark.sourceforge.net/wiki/>). It uses the *Open Dynamics Engine* (ODE) for

detecting collisions and for simulating rigid body dynamics. *ODE* allows accurate simulation of the physical properties of objects such as velocity, inertia and friction.

Besides the physical simulation, the simulator maintains the states of a soccer match according to the decisions of an automated referee. The referee decides about the game states according to the soccer rules of the *RoboCup* competitions. The server informs the agents about game states and prevents players from forbidden locations, e.g. crossing the halfway line before kick-off.

Parameters of the simulator can be changed by the various *configuration* files(*rb-files*) provided by *Simspark*. The field coordinates have their center in the middle of the playground, the x-axis points to the opponent goal. The dimensions of the soccer field are $x = 18\text{m}$ by $y = 12\text{m}$. The center spot has a radius of 1.5 meters. Each goal is $y = 2.1\text{m}$ by $x = 0.6\text{m}$ with a height of $z = 0.8\text{m}$. The penalty area to each goal is $y = 3.9\text{m}$ by $x = 1.8\text{m}$. The soccer field is surrounded by a border of 10 meters in each direction. Space outside this border area is not reachable by an agent. The soccer ball has a radius of 0.042 meter and a mass of 26 grams. For an up to date list of all values please refer to (*./rcsserver3d/naosoccersim.rb*).

At each corner of the soccer field, and at the goal posts, a distinctive flag is placed. The positions of these flags are fixed and known to each agent. Agents perceive the relative position of a subset of these flags and are therefore able to localize themselves on the soccer field. Agents distinguish flags through their identifier as shown in *Fig 5.1*. While the markers for the flags are placed on ground level ($z = 0.0\text{m}$), the goalpost markers are placed on the top of each goalpost at a height of $z = 0.8\text{m}$.

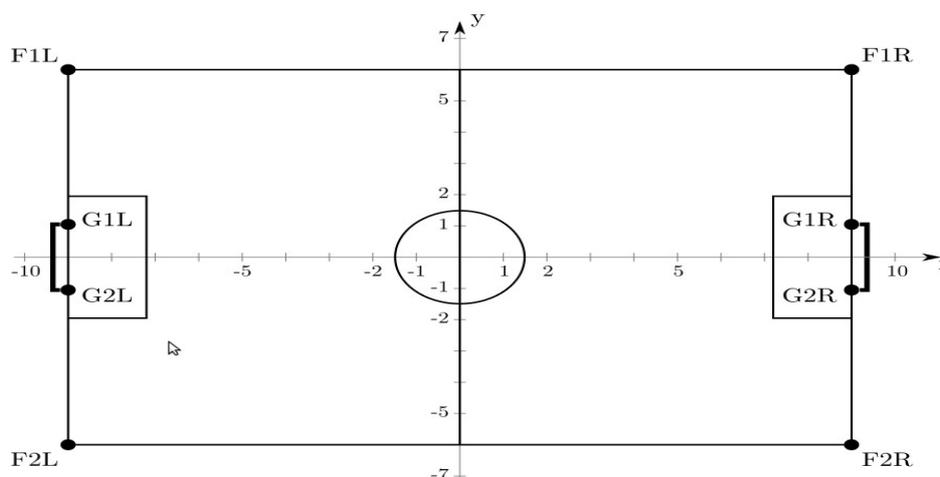


Fig. 5.1 The dimensions of the field and the object markers on the field as perceived by an agent

Simspark Monitor

Simspark provides also a visualization: The *Soccer Monitor* visualizes the ongoing match on the playground. It serves as a user interface and allows for interventions by a human referee, especially for game start and interrupts (e.g. in case of game stuck).

SimSpark provides an internal monitor and an external one. The first one is part of the *SimSpark* server. It can be enabled by editing the *configuration* files of the server. The second one is called *rcssmonitor3d* and it either connects to a running *SimSpark* instance or replays a simulation run from a log file.



Fig. 5.2 Soccer Monitor connected to a soccer simulation with 6 vs 6 robots (source: Wiki)

The *SimSpark* monitor renders the current simulation it and displays the running time, the play mode, and the goal scores, respectively. The monitor serves also as a user interface and accepts commands by key or mouse. These commands either control the movement of the monitor camera or send instructions back to the server as controls of the human referee. Below in *Fig. 3* we give a table of the commands that the monitor accepts from the user.

Key	Function
q	quit monitor
left mouse	mouse look
pageup, keypad plus, right mouse	move camera up
pagedown, keypad minus	move camera down
a, left arrow	move camera left
d, right arrow	move camera right
w, up arrow	move camera forward
s, down arrow	move camera backward
1	camera to left goal
2	camera to left corner
3	camera to middle left
4	camera to middle right
5	camera to middle
6	camera to right corner
7	camera to right goal
l	free kick left
r	free kick right
k	kick off
b	drop ball
n	cycle selected agent
e	clear selection
lctrl+s	enter numeric agent selection mode (via l or r followed by a digit)
x	kill selected agent
m	move selected agent FreeKickDist meters back
p	pause the playback of a log file
f	move one step forward in the log file while paused
b	move one step backwards in the log file while paused
l	toggle forward/backward playback of log file

Fig. 5.3 List of Soccer Monitor commands

5.1.3 Robot Model used by the Soccer Server

The simulated robot is based on the real robot *Nao* from the French company Aldebaran (cf. <http://www.aldebaran-robotics.com>). This robot is used in many scientific projects all over the world, it is also used in the Standard Platform League of *RoboCup* (cf. <http://www.tzi.de/spl>). Its height is about 57cm and its weight is around 4.5kg. Details of the physical properties are presented on the *Wiki*. We should point out that the simulator supports other robot models too.

Actually, there are some differences between the real and the simulated robot. The simulated robot has 22 degrees of freedom, while the real one has only 21 (because the HipYawPitch joints are controlled by only one motor). The motors of the simulated robot can be controlled only by setting an angular speed, while the motors of the real robot are controlled via torque and stiffness. Not all sensors of the real robot are available by the simulated one. Moreover, instead of the raw sensory data, the simulation provides preprocessed data in some cases (e.g. for the vision data).

Simulated robot effectors:

Below we present several effectors of the simulated robot.

- Each joint can be controlled separately by related hinge joint commands. The figure “Joints

of the Nao model” shows all joints of the simulated robot with their names and their identifiers.

- The say effector allows to communicate textual “voice” messages to other agents via the Soccer Server. Note that other communication between agents (e.g. via sockets) are not permitted by the rules.
- Further effectors are dedicated to initialization (see *Wiki for a detailed information*).

Simulated robot perceptors

The robot is equipped with several perceptors. *SimSpark* uses the notion “*perceptor*” because some sensation messages contain preprocessed data (“*percepts*”) instead of raw sensor data. The simulated robot has the following perceptors:

- Joint perceptors report the current angle of each joint.
- Gyroscope and accelerometer keep track of radial and axial movements of the upper torso in the three dimensional space.
- Force resistance perceptors in each foot indicate the actual pressure on it.
- The visual perceptor presents objects from preprocessed images of the camera at the head. The view range is 120 degrees horizontally, and 120 degrees vertically.
- The hear perceptor presents say messages from other players in textual format.
- The game state perceptor informs about the actual play time and play mode.

For more details about the formats and contents of the messages please refer to (<http://sims-park.sourceforge.net/wiki/>).



Fig. 5.4 The real Nao robot (source:Wiki)



Fig. 5.5 The virtual Nao in the simulation environment (source:Wiki)

5.1.4 Communication between Agents and Soccer Server

The communication between the Soccer Server and the agents is realized by message exchange using TCP (details are described in the *Wiki*). After starting, an agent must connect to the Soccer Server. Then it has to send the initialization messages (see below).

For data transfer, the messages between the server and the agents are packed as byte streams. The messages use S-expressions (“symbolic expressions”) as their basic data structure. S-expressions are either strings, or lists of simpler S-expressions. They can be easily parsed.

The agent interacts with the Soccer Server like a central control program communicates with sensors and effectors of a real robot. Actually, some sensations are not presented as raw data but in an already preprocessed form as “percepts”, and *SimSpark* uses the term “*perceptor*” instead of “*sensor*”. All sensations of a single cycle are sent together as a server message which has to be parsed for access to the information of sensors. Similarly, the effector commands of a single cycle should be packed by the user-defined agent, to an agent message.

For more information about the messages transmitted during the agent-Soccer Server communication, and about their format please refer to (<http://sims-park.sourceforge.net/wiki/>) or the *SimSpark* user manual.

5.1.5 Synchronization between the Server and the Agents

As already presented in the overview, the basic cycle has a length of 20 msec. At each cycle, the server calculates the actual situation depending on the previous situation and the commands received from the agents. The calculation regards the physical laws and the implemented rules of soccer play. Furthermore, the server calculates the individual sensor information for each agent according to the new situation including the pose of the agent. This information is sent to the agents by a server message at every cycle, but not all perceptors are available at each cycle: Most importantly, the vision information comes only at each third cycle.

At each cycle, an agent can process the server message and decide for the next actions. He can send related effector messages to the server at each cycle. In sync mode, the server waits for the agent messages of all agents until it starts to calculate the new situation. This results in a deadlock, if one agent does not send its messages. The sync mode can be switched on (off) by setting the flag `agentSyncMode` to true (false) in the *configuration* file *spark.rb*. In Real Time mode (if sync mode is switched off), the server will not regard the agent messages which do not come in time. If an agent message comes in a later cycle, it will be processed in that cycle. Vice versa, the server will

send a server message at every cycle, which remains in the message stream until it is read by the agent. If an agent misses to read a message in time, then there can be several server messages in the stream, and the synchronization can be lost.

The exchange of messages is interleaved as depicted in Fig. 5 “Synchronization between Soccer Server and agent”. This corresponds to the time needed to process information in reality. Hence, an action command sent by the agent at cycle t will be processed by the server at cycle $t+1$ and the result can be observed by the agent not before cycle $t+2$. This must be regarded for controlling, i.e. the control needs an appropriate forethought.

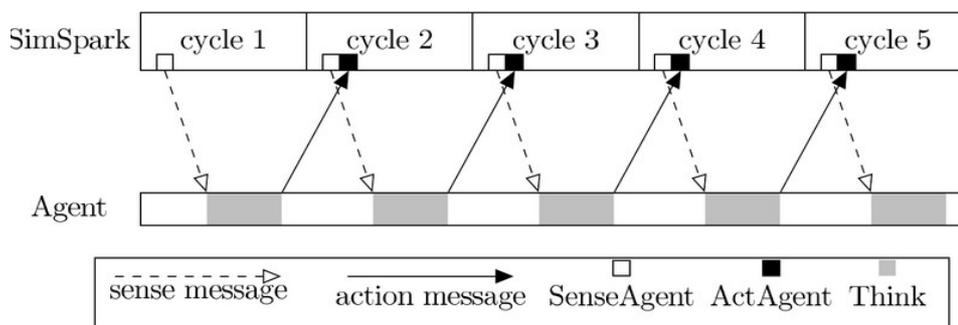


Fig. 5.6 Synchronization between Soccer Server and the agent (source: Wiki)

5.1.6 Simple Soccer Agent: C++ Framework for the SimSpark Simulator

Simple Soccer Agent (Mellman, Krause T. 2012) is a C++ software architecture which is able to run on the SimSpark simulator. The underlying framework we chose is based on the software release of the RoboCup team Berlin United - Nao Team Humboldt; it is written in C++ and provides basic activities, such as Walk, Turn Left, Scan For Ball or Stand Up. More importantly it provides an easy interface for the user who want to focus only in developing the behavior of the agent. When using Simple Soccer Agent, the user is not concerned about implementing basic activities for his agent nor implementing the communication and synchronization with the Soccer Server (Mellman H., Krause T 2010).

The interface provided in order to change the *sample-agent's* behavior (e.g. for better coordination) is the *Cognition* class interface (Fig. 5.7 and Fig. 5.8). As its name suggests, the framework uses this class to provide an easy access to the user to all the robot's updated sensor

values, and all the information about the state of the simulated world. Based on these information the robot must execute its next action. A user can change the behavior of the robot by changing the files *Cognition.cpp* and *Cognition.h* accordingly.

In order to register a new component to use for cognition, we use the interface provided by *Cognition's* initialization function (Fig. 5.7). If we want to edit the behavior of the agent we edit the *Cognition's* call function (Fig. 5.8). (For more information please refer to the SimpleSoccerAgent documentation <http://www.naoteamhumboldt.de/en/projects/simple-soccer-agent/>).

```
void init(naoth::PlatformDataInterface& platformInterface) {
platformInterface.registerCognitionInput(theSensorJointData);
platformInterface.registerCognitionInput(theInertialSensorData);
platformInterface.registerCognitionInput(theFSRData);
platformInterface.registerCognitionInput(theAccelerometerData);
platformInterface.registerCognitionInput(theGyrometerData);
platformInterface.registerCognitionInput(theVirtualVision);
platformInterface.registerCognitionInput(theImage);
platformInterface.registerCognitionInput(theFrameInfo);
platformInterface.registerCognitionInput(theSimSparkGameInfo);
std::cout << "Cognition register end" << std::endl; }//end init
```

Fig. 5.7 Fragment from *Cognition.h*

```
void Cognition::call() {
// perceive the world information
perception();
// make a decision what to do next
decide();
} //end call
```

Fig. 5.8 Fragment from *Cognition.cpp*

5.1.7 Developing the Agent's Behavior using the Extended KSE

In this example, we use GGenerator to create a behavior for an agent that plays soccer. Our simple behavior can be described as follows: *The agent searches for the ball in the soccer field and, if the ball is found, the robot walks towards the ball and takes it in possession. If the game is over, the robot pauses.* The first step is to create the properties files; these files are shown below. In fact, we just need to include the appropriate header files to gain access to those framework-provided variables that allow the user (and the statechart!) to gain information about the game and the perception of the ball, but also request the execution of a motion.

Our purpose was to work only in the high level of agent programming (thus we mean the “brain” of the agent) and not to have to deal with low level programming such as agent actions (Walk, Kick etc), or agent-simulator technical aspects (e.g communication with the simulator). The framework we choose meet these demands by providing basic activities like *Walk, Turn Left, Scan*

For *Ball* or *Stand Up* for our agent, and also handles the communication with the simulator.

We divide the process of creating a behavior for a specific framework in three stages. In the first stage we create the properties files that will configure our blackboard interface in order to connect our statechart model with the specified platform. In the second one we use the KSE GGenerator to create our behavior and generate code for it and in the last one we instantiate our generated behavior within the targeted framework.

Stage 1: Generic Blackboard configuration

When we are about to create a blackboard interface for a specific framework we focus on two things. First we have to be able to connect our interface with the framework and second we need to instantiate our statechart model as a new module in the framework. The second part takes place after the code generation and we don't have to worry about it yet. For the first part we need to know some basic things about the framework we target, like how to obtain the information we need for our statechart model. In this case the information we need includes the updated sensor readings for the robot and the state of the simulated world (*time elapsed, team name, player number etc.*). We also need to know how to use the provided robot-actions(*Walk, Scan etc.*).

Once we know the modules of the framework that provides us with that kind of information, we create the blackboard's *properties files*. The *properties files* consist of the files *include_classes.txt* and *instances.txt*. The first file includes any header files necessary for the update of the robot's sensor readings. The second file, instantiates the required modules to achieve the right communication between the blackboard and the targeted platform. The properties files that were used in this example are shown below and they are responsible for generating a blackboard interface that *connects* our statechart model with the Simple Soccer Agent framework (we discuss further about the properties files in a previous section). Now the blackboard interface that will be created during the code generation will act like a middleware between the simulator and our statechart model.

<i>include_classes.txt</i>	<i>include_instances.txt</i>
<i>BallPercept.h</i>	<i>BallPercept theBallPercept;</i>
<i>MotionRequest.h</i>	<i>MotionRequest theMotionRequest;</i>
<i>SimsparkGameInfo.h</i>	<i>SimsparkGameInfo gameState;</i>

Fig. 5.9 *properties files for SimpleSoccerAgent framework*

Stage 2: Using the KSE Generic C++ Generator

The next step is to use the KSE graphical interface to create our behavior. We start by providing the liveness formulas that describes our agent behavior abstractly:

$$\begin{aligned} \text{LogicalAgent} &= \text{Init} . (\text{Play} \mid \text{NoPlay})^+ \\ \text{Play} &= [\text{StandUp}] . (\text{PlayBall} \mid \text{ScanBall}) \\ \text{PlayBall} &= \text{Turn} \mid \text{Walk} \end{aligned}$$

The first formula indicates that our behavior (LogicalAgent) will execute Init (for initialization of the player) and then will choose one or more times between Play or NoPlay exclusively (depending on the current game state). The second formula suggests that our behavior may execute StandUp (if needed) and then will choose between PlayBall or ScanBall exclusively (depending on whether the ball is visible or not). Finally, the third formula indicates that our agent will choose between Turn or Walk exclusively (depending on where the ball is seen).

As soon as we provide the KSE GGenerator with the liveness formulas, the initial statechart model is generated and the user has to associate it to a source code repository that provides the code for the basic agent activities (Walk, Scan, etc). If the framework does not provide some activity, our tool generates the corresponding skeleton code and the user is asked to provide the corresponding C++ code using the built-in editor in KSE. Note that the abstract behavior specification of the liveness formula specifies what activities are included in the desired behavior, but gives no information on when execution switches from one activity to another. It is the execution of transition expressions that make this switching between activities possible.

After the statechart is created, we use the statechart editor tool of KSE to add the necessary variables we are about to use in the transition expressions. In our case these are: myInertialSens, ballFound, and horizontalAngle. The first variable is of type class and holds information about the posture of the robot. The second one is of type bool and is true if the ball is seen by the robot and false otherwise. The last one is of type double and holds information about the horizontal angle between the robot and the ball. These are user-defined variables, which are updated from the framework- provided variables inside the statechart, and are used in our transition expressions for controlling the statechart execution. They are added in the blackboard interface using the editing tool provided by the KSE. The next step is to edit the transition expressions, according to the grammar, using the KSE editor. The final statechart model with all the transition expressions is shown in *Fig 5.10*. The last step of our procedure is to use GGenerator to produce the C++ code for this statechart model and transfer the generated files to the source folder of the framework, so that it

is compiled and used in the target framework.

Stage 3: Instantiate our generated behavior in the framework

To execute the generated agent behavior we have to instantiate it within the target framework. This is a framework-specific step and depends on the requirements set by the framework itself. In the present example, it amounts to simply adding a couple of C++ lines (see *Fig. 5.19 and Fig. 5.20*) in the configuration files of the framework to register the generated behavior as a new module within the framework. This is the only handwritten C++ code during our behavior generation procedure and is done only once, since any updates to the statechart simply modify the behavior, but remain registered in the framework. In *Fig 5.11-5.14* we show four screenshots from the execution of the LogicalAgent behavior. Initially, we register the agent in the environment; as soon as the agent interacts with the environment, he starts to search for the ball; when he finds the ball he goes towards it, and takes it in possession.

By following the procedure described above we created a simple behavior for the Simpark Robotic Soccer simulator in three simple steps. We should point out that only in stage three of the procedure we wrote C++ a small amount of code by hand. We should also notice that if the framework doesn't provides us with the basic agent actions, our tool generates the skeleton code for these actions used in the behavior although we have to code them by hand afterwards. This happens when a repository with agent's basic activities is not available. At last in *Fig. 5.15-5.20* we show parts of the code that is produced during the generation process. As we can see in *Fig 5.16* the generated blackboard includes the files *BallPercept.h* and *MotionRequest.h* as long as their instances as expected to do according to the properties files we used. In *Fig 5.17* we see a small part of the generated code for our behavior.

Finally we see code fragments produced for a transition expression in *Fig 5.18*, and in *Fig 5.19 and 5.20* we see in bold the changes we made in the framework in order to instantiate our behavior in it.

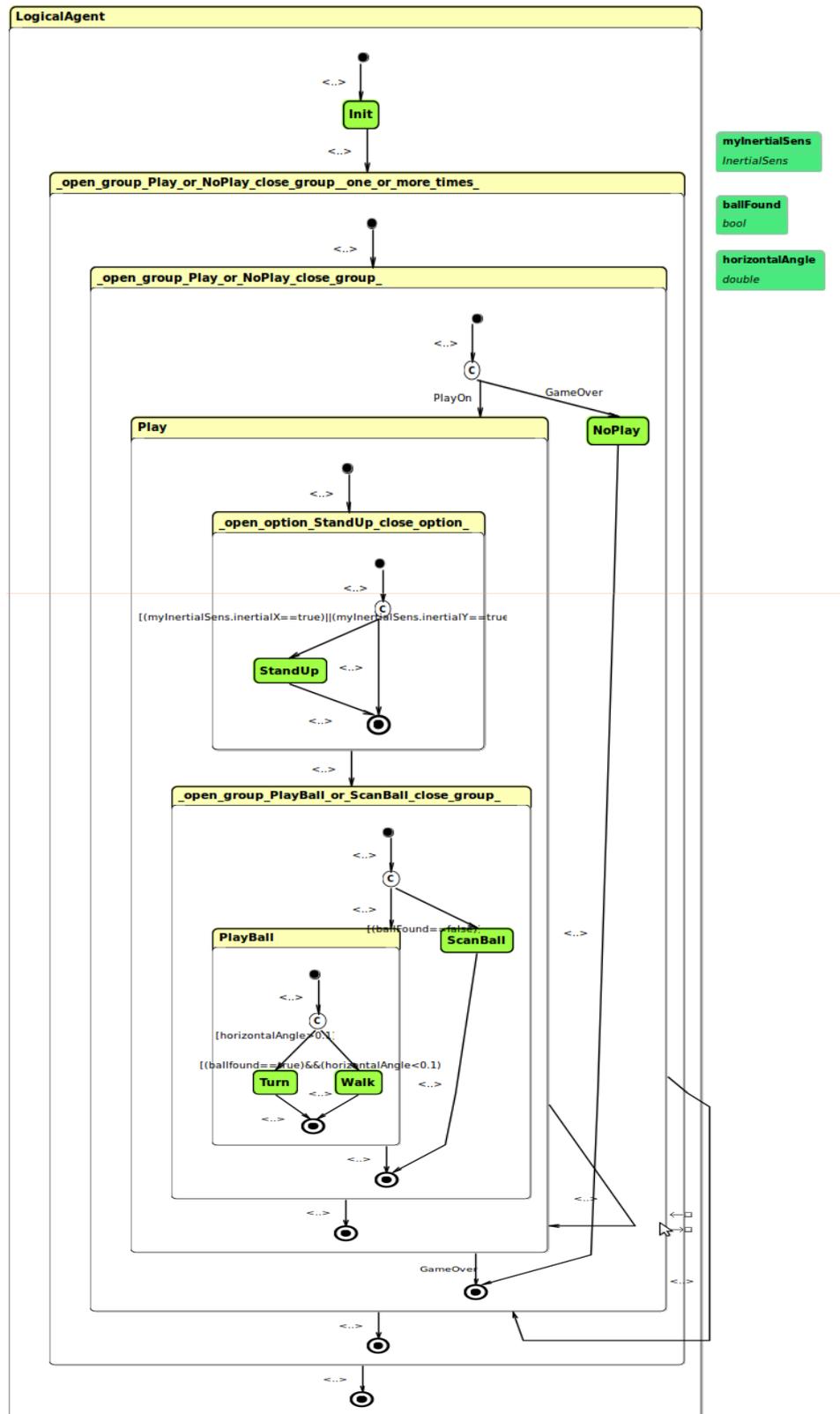


Fig. 5.10 Logical Agent Statechart model with all the transition expressions

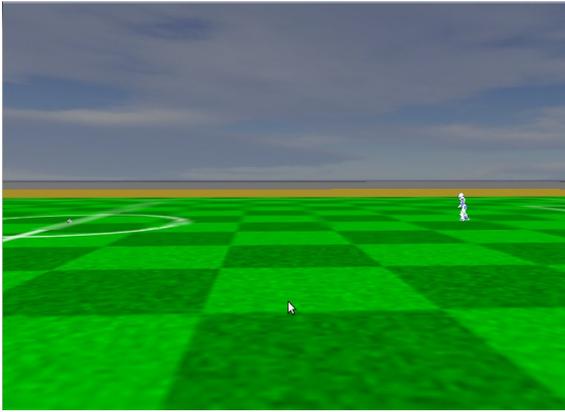


Fig. 5.11. we add the agent on the environment

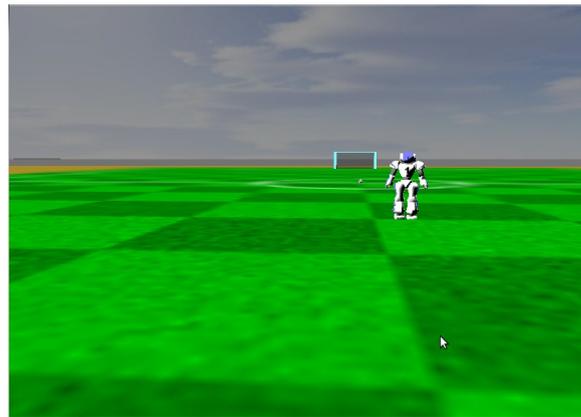


Fig. 5.12 The agent searches for the ball



Fig. 5.13 The agent goes towards the ball



Fig. 5.14 The agent takes possession of the ball

```
#include "BallPercept.h"
#include "MotionRequest.h"
#include <iostream>
#include <string>
#include <list>
#include "TimeStamp.h"
using namespace std;
using namespace statechart_engine;
```

Fig. 5.15 Generated blackboard fragment (a)

```
public:
  BallPercept theBallPercept; /*ballPercept*/
  MotionRequest theMotionRequest; /*request
for motion*/
  list<Time_stamp> *listTimeStamps;
  BlackBoard({})
  BlackBoard(const string& str){name=str;}
  virtual ~BlackBoard({})
```

Fig. 5.16 Generated blackboard fragment (b)

```

class LogicalAgent {
public: LogicalAgent (MessageHub* com) { _statechart
= new Statechart ( "Node_LogicalAgent", com );
Statechart* Node_0 = _statechart;
_states.push_back( Node_0 );
StartState* Node_0_1 = new StartState ( "Node_0_1",
Node_0 ); //Name:0.1
...

```

Fig. 5.17 Part of the generated code for our behavior

```

void Cognition::call() {
perception();
//starting the statechart
if(mr_logic->getStatechartIsRunning()==false) {
mr_logic->Start(); //statechart runs in a new
//thread
}else if(BlackBoard::getInstance().getPlayMode()==
"GameOver") {
//stops the thread that runs the statechart
mr_logic->Stop();
} //closing else

```

Fig. 5.19 Changes to Cognition's call function

```

class
TrCond_LogicalAgent0_3_2_3_3_3_20_3_2_3_3_3_4 :
public statechart_engine::ICondition {
public:
void UserInit () { }
bool Eval() {
/* [(ballfound==true)&&(horizontalAngle<0.1)] */
if((this->_blk->ballFound==true)&&(this->_blk-
>horizontalAngle<0.1)){ return true; }else{ return
false; } } };

```

Fig. 5.18 Generated code for a transition expression

```

void init(naoth::PlatformDataInterface& platformInterface)
{ platformInterface.registerCognitionInput(theSensorJointData);
platformInterface.registerCognitionInput(theInertialSensorData);
platformInterface.registerCognitionInput(theFSRData);
platformInterface.registerCognitionInput(theAccelerometerData);
platformInterface.registerCognitionInput(theGyrometerData);
platformInterface.registerCognitionInput(theVirtualVision);
platformInterface.registerCognitionInput(theImage);
platformInterface.registerCognitionInput(theFrameInfo);
platformInterface.registerCognitionInput(theSimSparkGameInfo
);
std::cout << "Cognition register end" << std::endl;
//register our agent in the framework
com=new MessageHub();
mr_logic=new LogicalAgent(com); //end init

```

Fig. 5.20 Changes to Cognition's init function

5.2 Cooperative Multi-Agent Behavior

In this section we follow the procedure described above in order to create two different behaviors for agents that are part of the same robotic team. The team consists of the agent that is described in the previous section and the two newly created agents; `player_1` and `player_2`. Thus we can demonstrate the concept of cooperation between the agents. We have already shown the statechart of chief-player (Logical Agent) in the previous section. Now we show below the two different liveness formulas and the corresponding statecharts that each one describes a different agent behavior (`player_1` and `player_2` accordingly) .

For running and stopping our agents simultaneously we created two simple scripts that use simple linux commands. The start.sh script starts the agents and connects them to the SimSpark server. The kill.sh stop the agents. Below we present screen-shots from the cooperative scenario and we also show the scripts mentioned above.



Fig. 5.25 Cooperative Behavior screen-shot 1



Fig. 5.26 Cooperative Behavior screen-shot 2

```
#!/bin/bash
# Start script for 3D Simulation
team="KOURETES"
AGENT_BINARY=naoth-simspark
BINARY_DIR="."
NUM_PLAYERS=3 #can change according to our will
for ((i=1;i<=NUM_PLAYERS;i++)); do
  echo "Running agent No. $i"
  "$BINARY_DIR/$AGENT_BINARY" &> /dev/null &
  sleep 2
done
```

Fig. 5.27 Start team script

```
#!/bin/bash
# kill script for stoping the agents
# Kill agents
AGENT="naoth-simspark"
killall $AGENT
sleep 1
```

Fig. 5.28 Kill team script

5.3 Creating a Behavior for the Wumpus World Simulator

5.3.1 Platform Overview

The Wumpus Simulator is a simple C++ framework for simulating the Wumpus World

described in Russell and Norvig's "Artificial Intelligence: A Modern Approach". In this world the agent's goal is to climb down in a cave find the gold hidden in it, and return back alive.

Unfortunately for the user there are many traps in this cave along with a monster (*Wumpus*) that's waiting to eat him. The user is armed with only one arrow. Thus he has to depend on his "brain" in order to achieve his goal and come out of the cave alive. The simulator provides us with the updated state of the world each time before we decide our next move. The state of the world is defined by a set of boolean variables. These are *Stench*, *Breeze*, *Glitter Bump* and *Scream*. The available actions for an agent in this environment are predefined (*Forward*, *Turn Left*, *Turn Right*, *Shoot*, *Grab*, *Climb*)

As we can see is a much more different environment than the SimSpark simulator. The Wumpus Simulator is written in C++ by Larry Holder (holder@wsu.edu).

5.3.2 Simulation using the Wumpus World Simulator

The simulator works by generating a new world and a new agent. Before each try on this world, the agent's `Initialize()` method is called, which you can use to perform any pre-game preparation. Then, the game starts. The agent provided by the user must call the *Process* method in order for the agent to return an action, which is performed in the simulator. Prior to *Process* call the agent must have the updated state of the world given by *Percept* class. This continues until the game is over (agent dies or leaves cave) or the maximum number of moves (1000) is exceeded. When the game is over, the agent's *GameOver* method is called.

After the game is over, the agent is deleted. So, you may want to store some information in the agent's destructor method to be reloaded during the agent's constructor method when reborn for a next trial. If additional trials have been requested, then a new *Wumpus* world is generated, and the process continues as described above. Scoring information is shown at the end of each try.

Simulator Options

The wumpus simulator takes a few options, as described below.

-size <N> lets you to set the size of the world to NxN (N>1). Default is 4.

-trials <N> runs the simulator for N trials, where each trial generates a new wumpus world. Default

is 1.

-tries <N> runs each trial of the simulator N times, giving your agent multiple tries at the same world. Default is 1.

-seed <N> lets you set the random number seed to some positive integer so that the simulator generates the same sequence of worlds each run. By default, the random number seed is set semi-randomly based on the clock.

-world <file> lets you play a specific world as specified in the given file. The **-size** option are ignored, and each try and trial uses the same world. The format of the world file is as follows (all lowercase, must appear in this order):

size N

wumpus N N

gold N N

pit N N

pit N N

...

where N is a positive integer. Some error checking is performed. A sample world file is provided in `testworld.txt` as part of the simulator.

Running a Wumpus World simulation

To try out the simulator, install the code on a UNIX system (or a system that has the 'make' program installed and a C++ compiler). Type 'make' to build the 'wumpsim' executable. Then, type './wumpsim'. You should see a randomly-generated 4x4 world, information about the game state, and a prompt for the next action. When the game is over, scoring information is provided as an output to the screen. In *Fig. 5.29* we show a screen-shot for a randomly generated 4x4 world, while in *Fig. 5.30* we see a 7x7 world.

5.3.3 Wumpus Agent: C++ Framework for the Wumpus World Simulator

Wumpus Agent provides a C++ *sample-agent* that can interact with the Wumpus World Simulator. More specific the framework provides basic agent activities needed for the Wumpus environment such as *Forward*, *Turn Left*, *Turn Right*, *Shoot*, *Grab*, *Climb*, and also provides an easy interface for editing the behavior of the *sample-agent* according to user needs.

In order to create a behavior for the Wumpus World Simulator we use the interface provided by the *Percept class*, and the interface provided by the *Agent class* of the *Wumpus Agent* framework. The *Percept class* provides access to the information about the simulated world's *state*. This information is updated by the simulator before the agent executes his next move. Based on the information that is stored in *Percept class* the agent must choose his next action. This class is used by the *Agent class* interface. The user only has to edit the interface's *Process* method (Fig. 5.31) in order to create a new behavior.

```

george@ubuntu: ~/Desktop/Thesis_3_14/wumpus-2.5-original
george@ubuntu:~/Desktop/Thesis_3_14/wumpus-2.5-original$ ./wumpsim
Welcome to the Wumpus World Simulator v2.5. Happy hunting!

Trial 1, Try 1 begin

World size = 4x4
+---+---+---+
|   | P |   |
+---+---+---+
|   | G |   |
+---+---+---+
|   |   | P |
+---+---+---+
| A>| W | P |
+---+---+---+
Current percept = [Stench=0,Breeze=0,Glitter=0,Bump=0,Scream=0]
Agent has gold = 0, agent has arrow = 1
Current score = 0

```

Fig. 5.29 4x4 Wumpus World simulation

```

george@ubuntu: ~/Desktop/Thesis_3_14/wumpus-2.5-original
Trial 1, Try 1 begin

World size = 7x7
+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
| W | G |   |   | P |   |   |
+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
| A>|   |   |   |   |   |   |
+---+---+---+---+---+---+
Current percept = [Stench=0,Breeze=0,Glitter=0,Bump=0,Scream=0]
Agent has gold = 0, agent has arrow = 1
Current score = 0

```

Fig. 5.30 7x7 Wumpus World Simulation

```

Action Agent::Process (Percept& percept) {
char c;
Action action;
bool validAction = false;
while (! validAction) {
validAction = true; cout << "Action? "; cin >> c;
if (c == 'f') { action = FORWARD; } else if (c == 'l') { action = TURNLEFT; }
else if (c == 'r') { action = TURNRIGHT; } else if (c == 'g') { action = GRAB; }
else if (c == 's') { action = SHOOT; } else if (c == 'c') { action = CLIMB; } else { cout << "Huh?" <<
endl;
validAction = false; } } return action; }

```

Fig. 5.31 Sample Agent Process method

5.3.4 Developing the Agent's Behavior using the Extended KSE

In this example, we create a behavior for the Wumpus world: *the agent tries to find the gold in the maze and escape alive*. Our goal is not to solve the maze, but to simply demonstrate how easy it is to create a behavior for this environment using GGenerator. As we discussed in the previous section in order to develop a behavior for a specific framework we have to follow three simple steps. In the first step we *connect* the framework with our *statechart engine* via the generic blackboard interface (*properties files*). In the second one we use the KSE GGenerator to create our behavior and generate code for it, and in the last one we register our generated behavior to the platform.

Stage 1: Generic Blackboard configuration

To connect our statechart to the Wumpus World platform we create the properties files first. Below in figure 25 we show the properties files used in this example (We discuss further about the properties files and the specific framework interface handling in a previous section).

include_classes.txt	include_instances.txt
WumpusWorld.h Percept.h Action.h	WumpusWorld ww;

Fig. 5.32 The properties files for the Wumpus World Framework

Stage 2: Using the KSE Generic C++ Generator

In this step we use the KSE GGenerator interface to create our behavior for the Wumpus World. We do this the same way we did our first example. First we provide the KSE tool with the liveness formulas that describe abstractly the behavior of the agent we want to create.

```
wumpie=init.(Play|NoPlay)+
      Play= percept.Move
Move=forward | turnLeft | turnRight | grab | shoot | climb
```

The first formula indicates that our behavior (Wumpie) will execute Init and then will choose one or more times between Play or NoPlay exclusively. The second formula suggests that our behavior will

execute Percept and then Move. Finally, the third formula indicates that our agent will choose one of the moves Forward, TurnLeft, TurnRight, Grab, Shoot, Climb (depending on the information gathered so far). From these liveness formulas, the initial statechart model is generated and the user has to associate it to a source code repository that provides the code for basic agent activities (Forward, Percept, Init, etc.).

In the next step, we use the KSE tool to add the necessary variables we are about to use in the transition expressions of the statechart. In this case these are: *glitter*, *stench*, *breeze*, *turnedRight*, *turnedLeft*, *scream*, *gold*, *arrow*, *posX*, *posY*. These are user-defined variables that are registered in the blackboard during the code generation. Once we add the transition expressions according to the grammar rules, we are ready to generate code for our behavior. In figure 26 we show the statechart created for our behavior with all the transition expressions.

Stage 3: Instantiate our generated behavior in the framework

The registration of our created behavior as a new module to the Wumpus World framework is done as in the previous example in two steps. In the first step we add our generic statechart engine source files to the source folder of the framework. In the second step we use the interface provided by the Wumpsim class that is part of the framework in order to instantiate our generated statechart model in it. This class provides us with the updated state of the agent's environment. In this class we instantiate our behavior.

The code added to Wumpsim class to instantiate our model it's the only piece of code we wrote by hand during the agent behavior development. In figures 5.34-5.36 we show fragments from the source code generated for a transition expression, our model and the blackboard for this example.

As we can see the blackboard interface has been generated according to the *properties files* that we provided to the tool and the addition of variables we made using the KSE GGenerator editing tool. We should also notice that the transition expression and the agent behavior code looks exactly the same with the code generated in the previous example. Thus we show that our tool, and the code it produces for our statechart model does not have any framework dependencies. At last, in figure 5.38 we show two screen-shots from the execution of our behavior within the Wumpus environment.

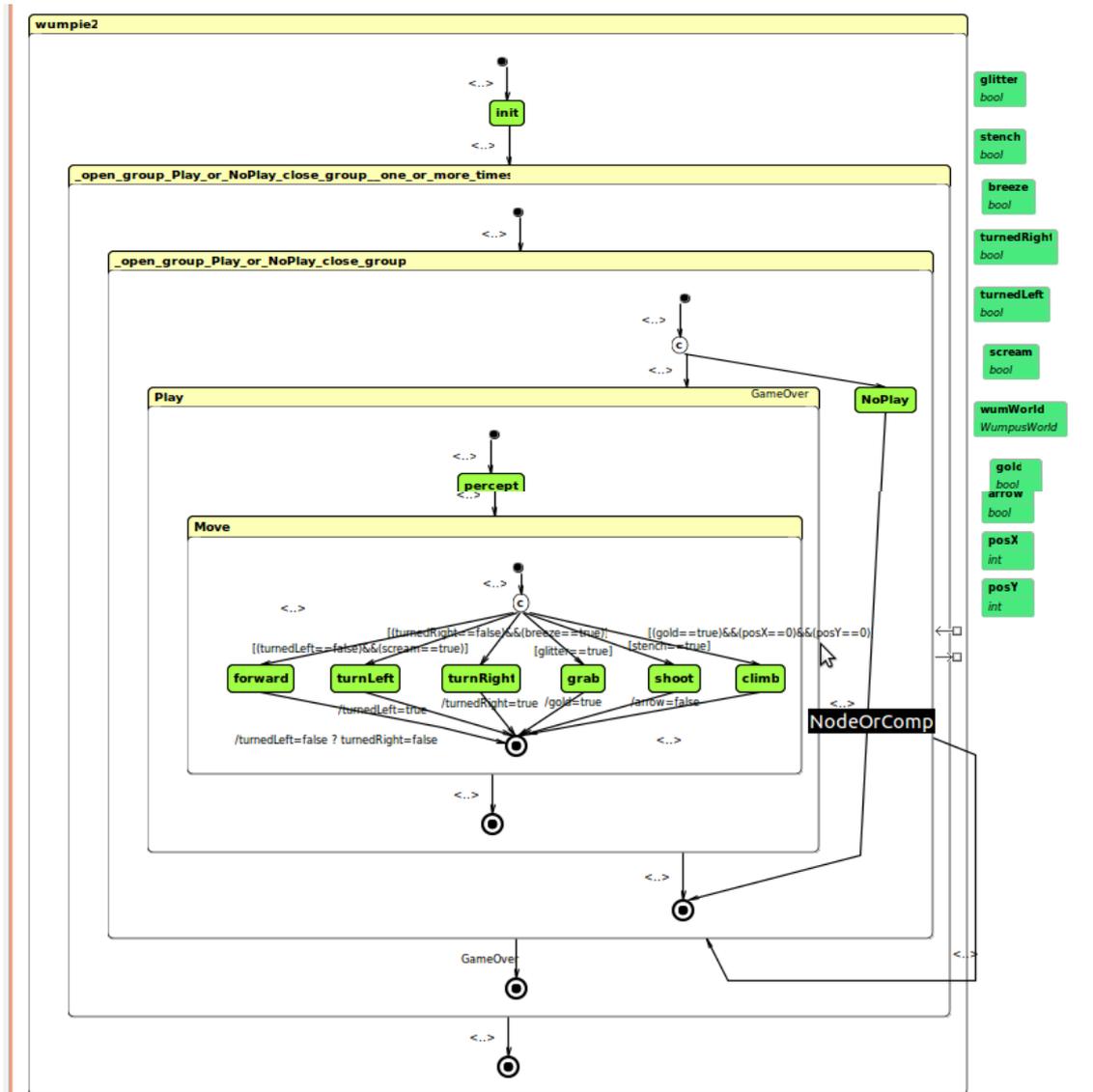


Fig. 5.33 Wumpus Statechart model with all the transition expressions

```
// 0.3.2.3.3.2TOturnLeft
class TrCond_wumpie0_3_2_3_3_20_3_2_3_3_4 : public
statechart_engine::ICondition {
public:
void UserInit () { }
bool Eval() {
/* [(breeze==true)&&(hasTurned==false)] */
if((this->_blk->breeze==true)&&(this->_blk->hasTurned==false)){
return true; }else{ return false; } } };
```

Fig. 5.34. Transition expression generated code

```
bool hasTurnedRight;
bool getVar_hasTurnedRight(){ return
hasTurnedRight; }
void setVar_hasTurnedRight(bool v)
{hasTurnedRight=v; }
void addWumpusWorld(WumpusWorld *ww)
{ wumWorld=ww; }
```

Fig. 5.35 Part of the generated blackboard

```

class wumpie {
public:
wumpie (MessageHub* com) {
_statechart = new Statechart ( "Node_wumpie", com );
Statechart* Node_0 = _statechart;
_states.push_back( Node_0 );
StartState* Node_0_1 = new StartState ( "Node_0_1",Node_0 );
...

```

Fig. 5.36 Generated source code of our behavior

```

wumpie* wumpAgent;
BlackBoard::
getInstance().addWumpusWorld(wumpusWorld);
wumpAgent->Start();

```

Fig. 5.37 Register our agent in Wumpsim class

```

Statechart started...
World size = 4x4
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   | P |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| A> | P | G | W | P |
+---+---+---+---+
Current percept = [Stench=0,Breeze=1,Glitter=0,Bump=0,Scream=0]
Agent has gold = 0, agent has arrow = 1
Current score = 0
turnRight executed.

```

```

World size = 4x4
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   | P |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| A> | P | G | W | P |
| Av |   |   |   |   |
+---+---+---+---+
Current percept = [Stench=0,Breeze=1,Glitter=0,Bump=0,Scream=0]
Agent has gold = 0, agent has arrow = 1
Current score = -1
forward executed.
World size = 4x4

```

Fig. 5.38 Two successive screen-shots from our behavior in the wumpus world simulator

5.4 Creating a Behavior for the Starcraft Brood War Strategy Game

StarCraft: Brood War is the expansion pack for the award winning military science fiction, real-time strategy video game StarCraft. Released in 1998 for Windows and Mac OS, it was co-developed by Saffire and Blizzard Entertainment. The expansion pack introduced new campaigns, map tilesets, music, extra units for each race, and upgrade advancements. The campaigns continue the story from where the original StarCraft ended. Brood War was critically well-received, with reviewers praising it for being developed with the care of a full game rather than as an uninspired extra. As of 31 May 2007, StarCraft and Brood War have sold almost ten million copies combined. The game is especially popular and professional players and teams participate in matches, earn

sponsorships, and compete in televised matches. Moreover the game is a test-bench for artificial intelligence applications. A StarCraft Brood War competition is held annually where artificial intelligence agents compete with each other and they also compete against humans.

Following the three-stages process described in previous sections we manage to create and instantiate an agent behavior for the StarCraft Brood War game. Our behavior is simple; if we find idle workers we order them to go to work. Thus, the player can focus on mobilizing the troops and building rather than dealing with idle workers. Below we show the properties files, the liveness formula and the statechart model that we use in this example. We won't explain technical details since the procedure for the creation of the StarCraft behavior is identical with the one described in the sections 5.2 and 5.3

<i>include_classes.txt</i>	<i>include_instances.txt</i>
BWAPI.h	

Fig. 5.39 Properties files for StarCraft

starcraft_bot = (Play | NoPlay)+

Play = doStuff | idle

Fig. 5.40 Liveness formula for StarCraft agent

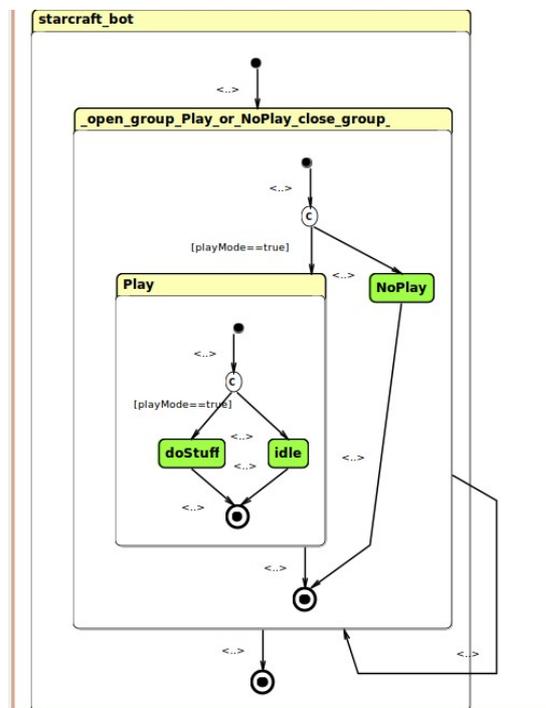


Fig. 5.41 Statechart model for the StarCraft game



Fig. 5.42 Two successive screen-shots from our starcraft behavior

We should mention that SimSpark 3D and Wumpus World are open source platforms. This, of course doesn't apply for StarCraft. StarCraft is a commercial game and in order to be able to interact with it we use the BWAPI interface. The BWAPI interface and the sample agents it provides use Visual Studio in order to compile and work. Although, since the BWAPI interface is written in C++, we have no problem in using it. As we have already clarify before, our tool can target any platform as long as this platforms supports C++.

5.5 Summary

As we can see in the examples given above, the procedure we followed in order to create our behaviors for each one of the three aforementioned frameworks (*SimSpark*, *Wumpus* and *StarCraft Brood War*) is identical. Although these three frameworks share a bunch of differences we deal with them the same way. To do so, we assume that the user knows basic things about the *mechanics* of the framework he targets. Our aim is to provide an ideal tool to the user who want to create a statechart-based behavior for the platform he works with, without having to deal with the writing of the source code for it. We also give the user the ability to modify at will, an already developed behavior via a user-friendly graphical interface.

In this chapter we try to give an idea of a *standard* way to deal with any kind of framework when using the KSE GGenerator as long as this framework is implemented in C++. If we can

standardize the procedure needed for this kind of job, we can save a significant amount of time during the developing of the agent's behavior and exploit the benefits provided by our tool.

Chapter 6

Conclusion

6.1 Discussion

In this thesis, we presented an extension to our CASE tool KSE that allows to specify statechart-based agent behaviors independently of the underlying software framework, relying on the C++ programming language as the common code base. We demonstrated the use of our tool in three completely different environments: SimSpark 3D Robotic Simulator, Wumpus World Simulator, and StarCraft Brood War strategy game. Moreover, we showed that the process we followed to generate sourcecode for the three different statechart models follows the same pattern and does not have any framework dependencies. Additionally, we presented a cooperative scenario for the SimSpark simulator.

It is easy to see that the three environments presented in Chapter 5 (*SimSpark*, *Wumpus World*, and *StarCraft Brood War*) are completely different to each other. Since we are aiming at showing that our KSE C++ Generic Generator is platform-independent and not aware of the environment that is going to generate code for, we pinpoint some of the differences of the three simulation environments we use for our examples.

SimSpark 3D Robotic Soccer Simulator simulates a 3D world, provides a physics engine for this world, and represents a dynamic environment. Moreover, the agents that participate in this simulator have a physical representation and are modeled as robots that can *sense* the environment, and take a decision according to their *senses*. In order to do so, an agent in this simulator must be able to *communicate* with the server that provides the simulation environment and the sensor values for the robot. Additionally, all the information the robot gets from the framework (*sensor values*) is “noisy” and the agent has to deal with concepts of uncertainty. As we can see, an agent has to take a lot of things into consideration in order to make even simple moves in this “noisy” environment. An action in this environment can have different results than expected, since the simulator adds noise to the perception of the agent. At last, the simulator provides a monitor for depicting the simulation.

In the Wumpus World, we have a two-dimensional static environment and the agent does not

have a physical representation in it. The agent doesn't have to connect to a server and we don't need to connect a monitor for watching the simulation. All magic happens in our terminal via text mode graphics. In this world, our agent is defined by his position on the world grid. The agent receives his *sensor* values with no noise added on them by the main program. Thus, once the agent makes a move in this world this move will be executed with the expected behavior contrary to the previous simulation.

The StarCraft environment is widely-known (since it's a famous game) and represents a strategy game, where agents need to plan ahead in order to achieve their goal. In Fig. 6.1, we show the major differences among the three simulation environments.

<i>SimSpark 3D Robotic Soccer Simulator environment</i>	<i>Wumpus World Simulator environment</i>	<i>StarCraft Brood War environment</i>
Stochastic	Deterministic	Deterministic
Dynamic	Static	Dynamic
Adds noise	No noise added	No noise added
Physical representation	No physical representation	No physical representation
Uncertainty	No uncertainty	Uncertainty
Sequential	Episodic	Sequential
Continuous	Discrete	Continuous
Multi-agent	Single-agent	Multi-Agent
Competitive, Cooperative	non-Competitive, non-Cooperative	Competitive, Cooperative

Fig. 6.1 Major differences between the three platforms we use

The aim of our work is to provide the user with an easy way to abstractly define statechart-based agent behaviors. Our contribution relies on the fact that the GGenerator does not generate code for a specific platform, like the old KSE did for Monas architecture; but, according to its initialization, can target any environment that uses C++. We achieve this by creating a blackboard interface that acts as a middle-ware between our statechart model and the targeted environment for every different platform. We should point out that in order for a user to use GGenerator for a specific framework, the user should know basic things about the framework. Assuming this is a fact, we presented a standard way of dealing with diverse frameworks. In Chapter 5, we demonstrate the transparent use of our GGenerator for three different platforms.

At last, we have shown how we exploited the KSE ability of working with arbitrary

generators, in order to embed our new software architecture in it and expand it in order to use the C++ GGenerator.

6.2 Future work

Our future plans, which served as motivation for this work, is the use of our statechart tools in order to automatically and massively generate team behaviors for RoboCup. Then, we plan to test these generated behaviors in the SimSpark 3D Robotic Simulator within an evolutionary framework for discovering suitable soccer team behaviors. As soon as this task is completed, we plan to test these behaviors on the real Nao Robots that are used by our team Kouretes. We are also looking forward in expanding our code generator in order to support other programming languages, like Java, and thus expand the range of frameworks that can be used in conjunction with KSE, such as the Jade framework (Java-based) .

Another domain which is of interest, is to use our statechart tools for creating behaviors for video-game agents (*like quake, unreal, racing simulation etc*). Video game environments offer a great opportunity for the user who wants to focus in developing autonomous agent behaviors. They can be multi-agent (*quake*) or single agent (*tetris*) environments. They provide a variety of skills for the game-agents (depending on the game), a simulated virtual world, and they offer an easy way to interact with this world and get results (*start game, add agent, save game, etc.*). This way, the user is not concerned with game graphics issues or with uncertainty during an agent *action* (*walk, shoot, etc.*) and is free to focus only in developing the behavior of the agent or the team of the agents that participate in the game. Given this, we can see that these environments can serve as a test bench for a large portion of artificial intelligence applications.

6.3 Lessons Learned

While working towards the completion of this thesis, the first thing that I learned is that when you have to work with advanced software architectures, like robot architectures, organization during studying is priceless. I also learned that, when you have to use someone else's code, the first thing to do is to read the manual (*if any exists*) and the second and more important is to contact the author (*Thank you Manolis and Vaggelis*).

Next, I learned that patience is a great asset to anyone during compilation time and that you can have an enormous number of versions of the software application you develop, until you reach your final version (and this helps a lot in the organization part). Finally, and most important,

working for this thesis made me understand that when you love your work, *all work and no play does not make Jack a dull boy.*

References

Benjamin Klatt. Xpand: A Closer Look at the model2text Transformation Language. University of Karlsruhe, Germany (2007)

Bjarne Stroustrup. The C++ Programming Language. Addison- Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

Budinsky Frank, Brodsky Stephen A., and Merks Ed. Eclipse Modeling Framework. Pearson Education (2003).

Harel, D., Kugler, H.: The RHAPSODY Semantics of Statecharts (Or on the Executable Core of the UML). In: Integration of Software Specification Techniques for Application in Engineering (1998)

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

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. Robocup: A challenge problem for AI. *AI Magazine*, 18:73–85 (1997)

Mellmann Heinrich, Yuan Xu, Thomas Krause, and Florian Holzhauser. NaoTH Software Architecture for an Autonomous Agent. Institut für Informatik, Humboldt-Universität zu Berlin (2010)

Mellmann Heinrich, Yuan Xu, and Thomas Krause Common Platform Architecture A Simple and Clean Architecture for Participation in SPL and Simulation 3D. Institut für Informatik, Humboldt-Universität zu Berlin (2010)

Paraschos, A.: A flexible software architecture for robotic agents. Diploma thesis, Technical University of Crete, Greece (2010)

Parnas D. *Software Fundamentals: Collected Papers by David L. Parnas* (2001)

Russell Stuart and Norvig Peter . *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition (2003)

Spanoudakis, N., Moraitis, P.: 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*. Springer (2008)

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

Topalidou-Kyniazopoulou, A., Spanoudakis, N.I., Lagoudakis, M.G.: A CASE Tool for Robot Behavior Development. In: Chen, X. et al. (eds.) *RoboCup 2012: Robot Soccer World Cup XVI*. pp. 225–236 Springer, Heidelberg (2013)

Topalidou-Kyniazopoulou, A.: *A CASE (computer-aided software engineering) tool for robot-team behavior-control development*. Diploma thesis, Technical University of Crete, Greece (2012)