Experience in Developing a Robot Control Software

Vahid Garousi
Software Quality Engineering Research Group (SoftQual)
Department of Electrical and Computer Engineering, University of Calgary
2500 University Drive NW, Calgary, AB Canada T2N 1N4, Canada
E-mail: vgarousi@ucalgary.ca

Abstract

This article presents an experience report in using the UML-driven development process to develop an object-oriented control software for a Sony AIBO robot. The entire project was a great learning experience for all the team members and we found the methodologies we used very effective in helping us design and develop a high-quality control system. We also feel that other software developers and practitioners, especially those developing robotic and embedded control software, would benefit from the experience and observations reported in this article.

Keywords: UML-driven development, State-driven behavior, Robot, Control software

1. Introduction

Intelligent robots have gained widespread use in various industries, e.g., aviation (those built by NASA), military, and the automobile manufacturing industry.

A robot has many components, mainly including hardware, software and mechanical parts. Developing software for robots is not a trivial task and people from industry and academia are constantly developing new ways to analyze, design, develop and test software systems for robots (for example, see the studies in (J.-A. Fernández-Madrigal, C. Galindoa, J. Gonzáleza, E. Cruz-Martína, and A. Cruz-Martína, 2008) (J. Kramer and M. Scheutz, 007).

RoboCup (www.robocup.org) is an international robotics competition founded in 1997 and is sponsored by many large corporations such as Cisco and 3M. The competition's aim is to develop autonomous soccer robots with the intention of promoting research and education in the fields of software engineering and artificial intelligence.

A fourth year design (capstone) project team at the University of Calgary consisting of four Software Engineering students and a supervisor (the author of this paper) has recently completed a RoboCup development project. We developed a control software (in C++) for the Sony AIBO robot, one of the popular robots used in the RoboCup competitions.

Our team used systematic software engineering techniques (such as UML-driven and state-driven development) to develop this object-oriented control software, and we found those techniques effective and very useful. The entire project was a great learning experience for all the team members and we believe the final product was a 'success story'. We feel that other software developers and practitioners, especially those developing robotic control software, would benefit from our experience as reported in this article. Our project has a website (V. Garousi, 2010) which contains the entire source code of the control software and also multimedia from our robots in action.

A brief background on the RoboCup competition is provided in the next section. We then report the techniques we have used in our robotic control software development project and our experience in using them:

- Development based on an existing robotic programming framework (Section 3)
- UML-driven analysis and design (Section 4)
- State-driven behavior development (Section 5)
- C++ implementation (Section 6)

Related works are discussed in Section 7. Conclusions and lessons learned are finally discussed in Section 8.

2. ROBOCUP

Sounding ambitious, the official goal of the RoboCup project is: "*By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, complying with the official rule of the FIFA, against the winner of the most recent World Cup.*" (H. Kitano and M. Asada, 2000).

One of the four RoboCup competitions is the RoboCup Soccer. One of the five leagues under the RoboCup Soccer is the Standard Platform League, formerly called Four-Legged League. From 2000-2007, the standard platform league used the Sony AIBO robots in the games. From 2007, the league's rules have changed to use the humanoid robots called NAO.

Since there is a larger body of knowledge and resource on AIBO robots compared to NAO, our team has decided to start the project from the AIBO and then to move to the humanoid NAO robots in near future. The Sony AIBO has been used as an inexpensive platform for robotic and artificial intelligence research by many groups worldwide, because it integrates a complete computer, vision system, and wireless Wi-Fi networking in a package less expensive than other conventional research robots.

Two snapshots from our AIBO in action are shown in Figure 1, in which the robot is running the soccer-playing control software developed by our team. More pictures and videos from our robot in action can be found online at (V. Garousi, 2010).

3. Development Based on an Existing Robotic Programming Framework

Tekkotsu (D. S. Touretzky and E. J. Tira-Thompson, 2005) (the name means "framework", literally "iron bones" in Japanese) is a popular open-source application development framework for robots. It was first developed to only support the Sony AIBO robot dog, but its latest version (4.0.1) supports a variety of other robotic platforms.

Tekkotsu has been created and is maintained at Carnegie Mellon University. It provides a layer of object-oriented abstraction above the original Sony OPEN-R robotic software development kit (SDK) that originally came with AIBO. Tekkotsu offers a variety of features, including an event routing architecture, a hierarchical state machine formalism for constructing behaviors, and an extensive collection of wireless remote monitoring and tele-operation tools. The latest version of Tekkotsu (version 4.0.1) is a large code-base consisting of 81,109 LOC C++ code.

Tekkotsu is by far the most popular development framework for AIBO robots (D. S. Touretzky and E. J. Tira-Thompson, 2005) and many other RoboCup teams worldwide have used the Tekkotsu framework. We thus decided to use it to develop customized soccer-playing control behavior for the AIBO in this project.

Tekkotsu is essentially a large library of C++ classes and several support tools. To give an insight on the feature-set provided by Tekkotsu, the class hierarchies under two of its main classes (ControlBase and EventListener) are shown in Figure 2 which have been adapted from Tekkotsu's online documentation (Tekkotsu Team, 2010).

There are numerous possibilities to develop customized control algorithms for the AIBO by extending subclasses from any of the classes shown under ControlBase. For example, we have extended the WalkEdit sub-class of ControlBase to develop specific walking (running) patterns for playing soccer.

The class BehaviorBase which itself is a sub-class of EventListener is the basis from which all ABIO behaviors should inherit. For complex behaviors, it is recommended to break aspects of the behaviors into independent "states", and use a state-machine formalism to control them (we will have an example on this later on). Overall, the Tekkotsu has an organized object-oriented class hierarchy which simplifies the task of developing specific control and behavior algorithms.

4. UML-Driven Analysis and Design

Since Tekkotsu has an object-oriented design, it was natural for us to follow a UML-driven development process for the AIBO's control software. To help us get started in the UML-driven system analysis and design, we used Tekkotsu's rich online documentation system (similar to the popular JavaDocs framework) (Tekkotsu Team, 2010). Furthermore, UML-based and state-driven design is a common practice for development of control and real-time software (e.g.,(H. Gomaa, 2000)(J. M. Bass, A. R. Brown, M. S. Hajji, D. G. Marriott, P. R. Croll, and P. J. Fleming, 1994)). There are even many commercial specific-purpose tools in this area, e.g., Matlab Simulink, IBM Rational Rose Real-Time. In the presence of an object-oriented development framework, it is nearly impossible to find any other effective approach than UML-based and state-driven behavior design for control software.

We developed component, class, state and sequence diagrams in the design phase. To provide insights into AIBO's control software architecture, a component diagram for the entire AIBO control system is shown in Figure 3. Tekkotsu interfaces with the AIBO through the Sony OPEN-R SDK. In Tekkotsu, all the events and information are received by the Controller component from the external AIBO devices (such as its wireless connection, camera and also buttons on it) and then passed to either of the components: Communication, Vision or *erouter* (an object of class type *EventRouter*).

Based on the system state that the controller is at, the event router object then passes the event to the right state object (of type StateNode). We have developed 20 soccer-specific states for this system, details of which are discussed in the following. Each of the states can be programmed to realize soccer-specific individual robot techniques (e.g., shooting) or team tactics (e.g., passing the ball to other robots). The control state objects create and issue suitable (soccer) motions for the present situation. The motion commands are sent to the right joints and actuators on the AIBO through the Tekkotsu's standard MotionManager component. More details about Tekkotsu's internal architecture can be found in its online resource(Tekkotsu Team, 2010).

Our next step was to analyze and design the robot's complex soccer-playing behavior. We followed Tekkotsu's recommendation to break the behavior down in into independent "states", and developed a state-machine formalism to control AIBO. The class diagram of the 20 states in our implementation is shown in Figure 4. Each of the 20 states is made a sub-class of Tekkotsu's StateNode class which is itself a sub-class of BehaviorBase.

As designed in Tekkotsu, a BehaviorBase object can receive events by adding itself to the list of listeners for the desired event stream. This list is maintained by the event router, erouter. Once the behavior is registered as a listener, its processEvent() method will be called when an event arrives.

The 20 states were designed and iteratively revised after extensive review of real soccer game techniques and tactics, and also by reviewing the official RoboCup rules(RoboCup Technical Committee, 2008). The big picture of the soccer-playing behavior is presented next by putting all these 20 states in a state-diagram-based behavior.

5. State-Driven Behavior

The behavior of most control systems is developed in a state-driven manner. The Tekkotsu framework as well as the official RoboCup rules (RoboCup Technical Committee, 2008) also strongly recommend state-driven behavior for AIBO robots. The UML state diagram of our control system is presented in Figure 5. The official RoboCup rules (RoboCup Technical Committee, 2008) require AIBOs' control software to have six minimum states: Initial, Ready, Set, KickOff (Playing), Penalized and Finished, which are shown in blue color in the state diagram. For brevity, the common postfix *StateNode* for all the states has not been shown in this diagram, e.g., *SearchForBallStateNode*.

As per the official RoboCup rules(RoboCup Technical Committee, 2008), when AIBO is in *InitialStateNode*, it should follow the Robocup rules: it should stand up, and turn on the light on front/back head button. AIBO will transition from this state to *ReadyStateNode* on back button press or the corresponding wireless message. In *ReadyStateNode*, AIBO looks for field markings and moves to a startup position. Once the KickOff (playing) event is received from the official referee system (through a wireless message), AIBO can start its customized game playing behavior. This is essentially where different RoboCup teams incorporate different creative game decision-making algorithms.

The 14 other states (shown in green color in Figure 5) were designed by our team to model soccer-playing control. A master state is *SearchForBall*, when AIBO pans and tilts its head in all directions looking for the ball. The state would transition to *TrackStateNode* once an orange area of considerable size and circle shape is detected.

While in *TrackStateNode*, AIBO tracks and approaches the ball. It also centers the ball within its feet. If the ball is moving towards the AIBO at a fast speed, it would exits to Sprawl in which it spreads its body out quickly to try to stop the forward movement of the ball. The moving of the ball towards the AIBO is detected by a vision controller module in which the following condition is checked: *currentOrangeColorArea-previousOrangeColorArea* > a given threshold (e.g., 30%). Note that, as per Tekkotsu's process architecture, this module is called 30 times a second.

In the tracking state, if the AIBO finds himself getting too close to another AIBO, the Reverse state becomes active and it backs away from the other AIBO to prevent getting a penalty. If there is a safe chance to roll the ball away from another AIBO (without leading to a penalty), the AIBO attempts to do so by activating the StealBall state (a suitable vision-based algorithm for this decision has been implemented).

In the trap state, AIBO reaches out and grabs the ball with his paws in front of him. Once the ball possession is done (by trapping it), it is time to search for the goal (net) by moving to the *SearchForNet* state. Once the opponent team's net is found, the AIBO positions itself behind the ball.

After proper positioning, the robot enters a decision making process in which it decides to either shoot the ball itself (note that three types of kicking), or pass the ball to a teammate AIBO if the other AIBO has a better position in terms of possibility to score (e.g., is closer to the net). If the net is straight in the front of the AIBO, chest kick is done. If there is an angle between net's position and AIBO's standing direction, left or right kick is done.

In any of the states, if the AIBO falls down (detected by reading the status of an onboard gyroscope), the Getup state becomes active and the control software sends the proper movement sequence to the actuators and joints to stand up the AIBO.

The control software also listens to the messages (events) received from the Wi-Fi wireless connection onboard of the AIBO and if a 'penalty' or (game) 'finished' message is sent by the official Robocup referee system (has a standard format), the AIBO will go to the corresponding state.

Once a behavior (state) is registered as an event listener in Tekkotsu, its processEvent() method will be called when an event arrives. Since the implementation details of each state are quite sophisticated, we first developed UML sequence diagrams for processEvent() method of each state in our UML-driven development process. For example, the sequence diagram for processEvent()for *SearchForBall* state is shown in Figure 6.

The processEvent()method is triggered by Tekkotsu's controller thread. An event object of type EventBase is passed in. The EventBase class forms the basis of communication between modules/behaviors in the Tekkotsu framework, and details about it can be found in the Tekkotsu documentation(Tekkotsu Team, 2010).

The event processing behavior of *SearchForBall* state involves the collaboration of the state object with three objects of class types: (1) MMAccessor, (2) EventRouter, and (3) VisionObjectEvent.

MMAccessor is the class in Tekkotsu which allows issuing motion and joint movement commands, e.g., stopping the AIBO, walking, kicking, etc. For example running the C++ statement MMAccessor<HeadPointerMC>(headpointer_id)->setWeight(0); will cause the AIBO's head to be reset by turning its angle to the initial up-right position. Class EventRouter handles distribution of events as well as management of state changes and timers. Class VisionObjectEvent extends EventBase to also include vision and visual information (e.g., colors).

The event processing behavior of *SearchForBall* starts with checking if the back button is pressed (which means hard freeze). This is actually done in the beginning of all states. Then, the global check for game finish event is conducted. Then, the event's properties are checked to see if it has been sent by another AIBO broadcasting that it has the ball and in that case, this particular AIBO will stop moving around and let the other one proceed.

The next condition is to check if the event is raised due to orange color been noticed in the latest image captured from the field. To get the ratio of the orange color area to the total image area (in pixels), a method called getArea() is called from the event object. If the area percentage is larger than a predefined threshold, the AIBO will consider that to be the actual ball and moves to the 'Track' state. In the calibrations we conducted, we found the threshold value of 7% a good choice for the above ratio. We observed cases that an orange-area percentage such as 3% could lead to false positives (detecting colors on other objects as orange which had resulted from too much light for example).

If the amount of the orange color is not considerable, the AIBO will keep looking around, which is systematically defined by looking left-wards, right-ward, up and down in sequence.

6. C++ Implementation

Since Tekkotsu is developed in C++, we developed the control software in C++ as well. Organized object-oriented structure of the Tekkotsu framework and its example tutorials made our development, maintenance and defect fixation tasks less challenging. A big proportion of our project was empirical calibration of different control settings (e.g., the orange color proportion as discussed above) which was a great experience.

As to the size of the code we developed, our team wrote 3,917 C++ LOC (all available as open-source online(V. Garousi, 2010)). Out of those, 2,454 LOC were specific for the state-driven behavior (implementing the 20 states). LOC breakdown for each of the 20 state classes is shown in Figure 7. Obviously, some of the states were more complex or had many steps to be done and thus had more LOC than others, e.g., the *ReadyState* had to initialize a lot of (control) parameters for the robot.

The remaining of our code was mostly for integration of (gluing) our customized behavior within the Tekkotsu framework, implementation of specific vision logic, such as detecting the orange-color ball, and blue-color goals (nets). As a measure of code complexity, the McCabe cyclomatic complexity (the cumulative number of control flows) of all the methods in the 20 state classes is 246.

As a source code example, partial listing of the method processEvent()for *SearchForBall* state is shown in Figure 8. The *cout* messages (seen in this code listing) are sent wirelessly to a remote computer as log data and were very helpful in debugging and tracing what the AIBO was actually doing.

7. Related Works

While a few existing works such as (G. Fortino, W. Russo, and E. Zimeo, 2004)(J. Murray, 2003) have proposed statecharts-based and UML development processes for mobile agents (including the AIBO), no existing work has reported the entire state-based development process for AIBO robots including the very-essential development aspects and also experiences on the topic. The lesson we learned in this context was that state-driven behavior development greatly simplified the control algorithm's development by helping us focus each behavior in its corresponding state.

This experience report also relates to the work of Kim et al. (M. Kim, S. Kim, S. Park, M.-T. Choi, M. Kim, and H. Goma, 2006) in which the authors applied the UML-based COMET design methodology (Concurrent Object Modeling and Architectural Design) to develop the control software of an intelligent service robot for the elderly, called T-Rot. Our works differs from that work in that we reported the implementation aspects of our system as well and that we focus on another type of robots (AIBO).

Our experience report complements the existing body of knowledge (e.g., (G. Fortino, W. Russo, and E. Zimeo, 2004)(J. Murray, 2003) (M. Kim, S. Kim, S. Park, M.-T. Choi, M. Kim, and H. Goma, 2006)) by adding the very-important implementation aspects and serves as another voice of evidence on the usefulness and effectives of UML-driven development processes for robotic applications. Together with the existing literature, these works start to build an empirical, gradually-generalizable experience base upon which developers of robotic systems can rely.

In terms of comparing the contributions and success of our approach to other competing designs and design methods, we could compare this development project to our past experience in developing control software using models other than state diagrams (e.g., our previous works in(V. Garousi, 2008)(C. Wiederseiner, S. A. Jolly, V. Garousi, and M. M. Eskandar, 2010). One clear advantage of state-driven behavior design in this project has been easier maintenance, better runtime performance, and also verification/validation of the system.

Furthermore, UML-based and state-driven design is a very common practice for development of control and real-time software and a large body of knowledge/experience (e.g.,(H. Gomaa, 2000)(J. M. Bass, A. R. Brown, M. S. Hajji, D. G. Marriott, P. R. Croll, and P. J. Fleming, 1994)) exists in this area. There are even many commercial specific-purpose tools in this area, e.g., Matlab Simulink, IBM Rational Rose Real-Time. Many commercial projects (e.g., (MathWorks Inc., 2010)(MathWorks Inc., 2010)) have used these practices for development of real control and distributed systems.

Hengst et al. report in (B. Hengst, D. Ibbotson, S. B. Pham, and C. Sammut, 2000) the software development details of the AIBO soccer robot developed at the University of New South Wales (Australia). That team did not use the Tekkotsu framework, but were instead interfacing with the Sony's OPEN-R SDK. The team does not seem to have used any formal design modeling language or approach in their development, nor have they provided all of their source code artifacts (except some brief example pieces). Thus, comparing our experience to theirs is not feasible.

Our open-source development artifacts (UML design models and source code) can benefits other practitioners/researchers working on AIBO applications. However, the artifacts are being offered 'as-is' and no warranty is offered.

8. Conclusions

We found the UML-driven development approach (especially the usage of state and sequence diagrams) to be very helpful in the project. It provided a higher-level of abstraction (compared to source code) and enabled early analysis, verification and also team communication on the control algorithms without getting into code-level implementation details.

As a future work, we plan to develop automated test suites (in unit, integration and system levels) for the AIBO control software.

Acknowledgements

This project was supported by the Discovery Grant no. 341511-07 from the Natural Sciences and Engineering Research Council of Canada (NSERC), IEEE Canadian Foundation and the Schulich Student Activities Fund. The system's source code was developed by 4th year design-project students Erik Clarke, Kevin Dorling, Matthew Sattlegger and Graham Wells.

References

B. Hengst, D. Ibbotson, S. B. Pham, and C. Sammut. (2000). The UNSW United 2000 Sony Legged Robot Software System, *Technical Report, School of Computer Science and Engineering, University of New South Wales,* 2000.

C. Wiederseiner, S. A. Jolly, V. Garousi, and M. M. Eskandar. (2010). An Open-Source Tool for Automated Generation of Black-box xUnit Test Code and its Industrial Evaluation, *Proceedings of the International Conference on Testing: Academic and Industrial Conference - Practice and Research Techniques*, pp. 118-128, 2010.

D. S. Touretzky and E. J. Tira-Thompson. (2005). Tekkotsu: A Framework for AIBO Cognitive Robotics, *Proc. of the American Conference on Artificial Intelligence*, 2005.

G. Fortino, W. Russo, and E. Zimeo. (2004). A statecharts-based software development process for mobile agents, *Information and Software Technology,* vol. 46, no. 13, pp. 907-921, 2004.

H. Gomaa. (2000). *Designing concurrent, distributed, and real-time applications with UML*: Addison-Wesley, 2000.

H. Kitano and M. Asada. (2000). The robocup humanoid challenge as the millennium challenge for advanced robotics, *Advanced Robotics,* vol. 13, no. 8, pp. 723-737, 2000.

J.-A. Fernández-Madrigal, C. Galindoa, J. Gonzáleza, E. Cruz-Martína, and A. Cruz-Martína. (2008). A software engineering approach for the development of heterogeneous robotic applications, *Robotics and Computer-Integrated Manufacturing,* vol. 24, no. 1, pp. 150-166, 2008.

J. Kramer and M. Scheutz. (2007). Development environments for autonomous mobile robots: A survey, *Autonomous Robots,* vol. 22, no. 2, pp. 101-132, 2007.

J. M. Bass, A. R. Brown, M. S. Hajji, D. G. Marriott, P. R. Croll, and P. J. Fleming. (1994). Automating the development of distributed control software, *IEEE Journal on Parallel & Distributed Technology: Systems & Applications,* vol. 2, no. 4, pp. 9-19, 1994.

J. Murray. (2003). Specifying Agent Behaviors with UML Statecharts and StatEdit, *Conference on Robot Soccer World Cup* pp. 145-156, 2003.

MathWorks Inc. (2010). ABB Accelerates Application Control Software Development for a Power Electronic Controller, [Online] available: http://www.mathworks.com/products/simulink/userstories.html, Last accessed: Nov. 2010.

MathWorks Inc. (2010). MathWorks Tools Help Toyota Design for the Future, [Online] available: http://www.mathworks.com/products/simulink/userstories.html, Last accessed: Nov. 2010.

M. Kim, S. Kim, S. Park, M.-T. Choi, M. Kim, and H. Goma. (2006). UML-based service robot software development: a case study, *International Conference on Software Engineering - Far east experience papers*, pp. 534-543, 2006.

RoboCup Technical Committee. (2008). RoboCup Four-Legged League Rule Book-2008 rules, [Online] available: http://www.tzi.de/spl/pub/Website/Downloads/AiboRules2008.pdf, 2008.

Tekkotsu Team. (2010). Tekkotsu Architectural Overview, [Online] available: http://www.tekkotsu.org/ArchitecturalOverview.html, Last accessed: May 2010.

Tekkotsu Team. (2010). Tekkotsu Reference Documentation, *http://www.tekkotsu.org/dox,* Last accessed: May 2010.

V. Garousi. (2008). Traffic-aware Stress Testing of Distributed Real-Time Systems Based on UML Models using Genetic Algorithms, *Elsevier Journal of Systems and Software (JSS), Special Issue on Model-based Software Testing,* vol. 81, no. 2, pp. 161-185, 2008.

V. Garousi. (2010). UofC Robocup Team Website, *http://www.softqual.ucalgary.ca/projects/robocup/,* Last accessed: May 2010.

AIBO is ready to shoot

Shooting the ball with chest
(the head is intentionally moved out of the way)

Figure 1. Two snapshots from our AIBO in action (full video is available online(V. Garousi, 2010))

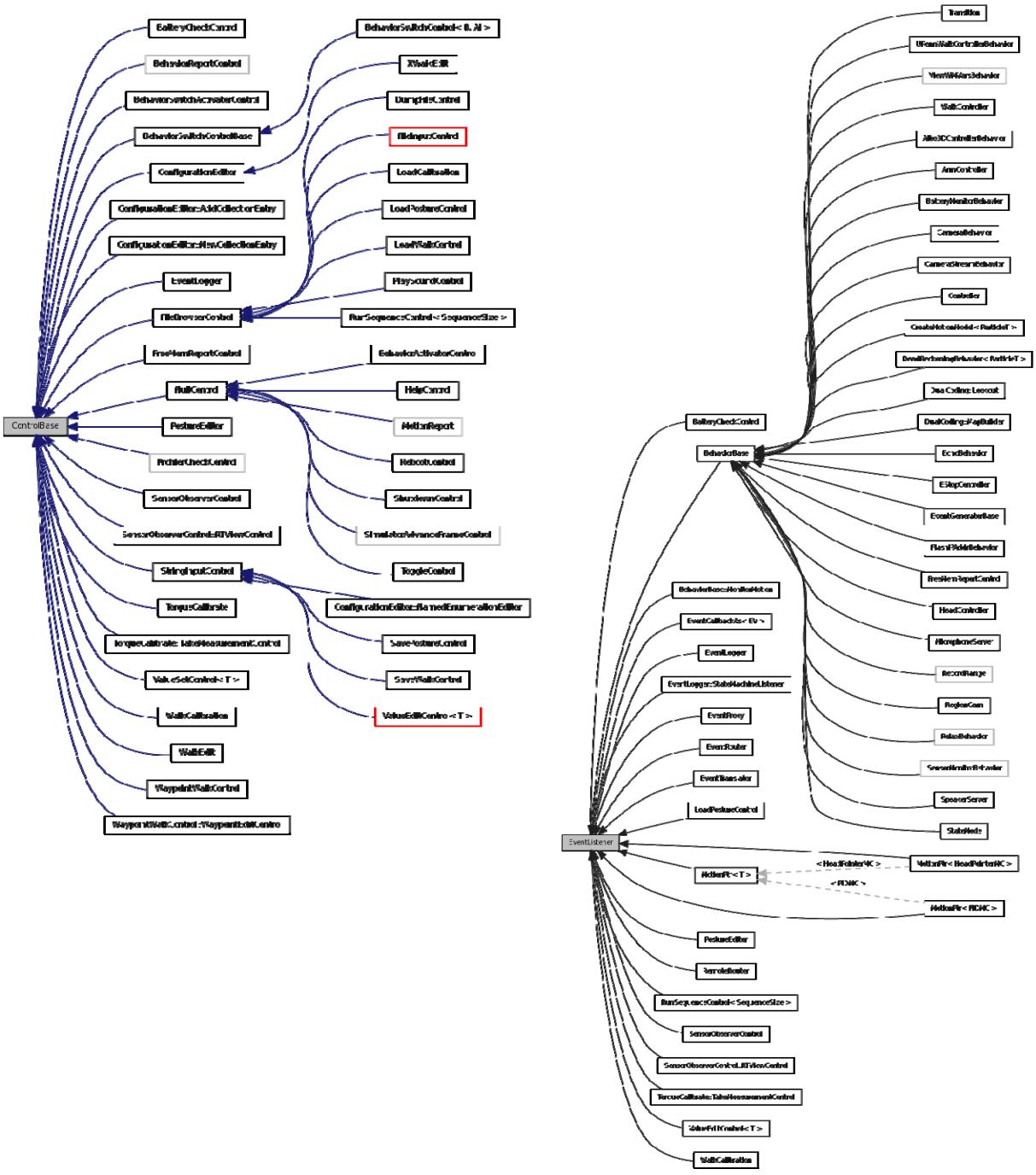

Figure 2. Class hierarchies under two main classes in Tekkotsu: ControlBase and EventListener (adapted from(Tekkotsu Team, 2010))

 ISSN 1913-8989 E-ISSN 1913-8997

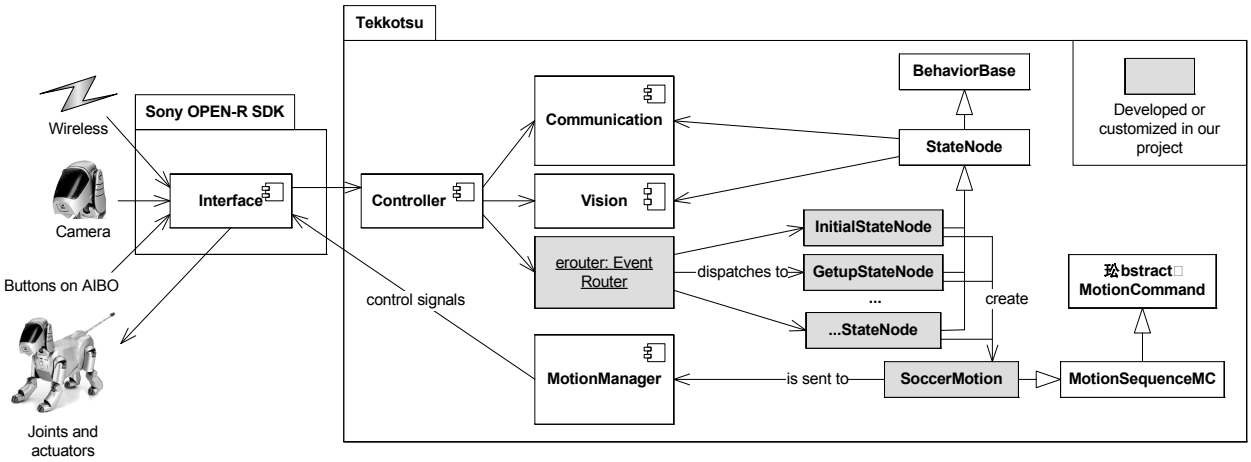

Figure 3. High-level component diagram for the entire AIBO control system

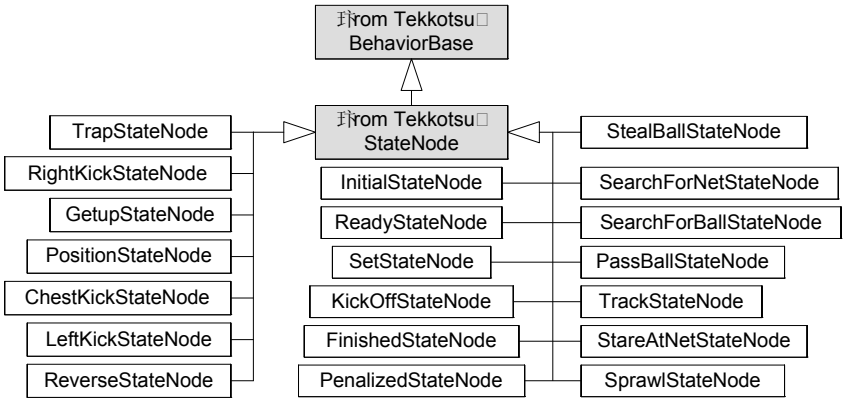

Figure 4. Control states defined as sub-classes of Tekkotsu's StateNode class

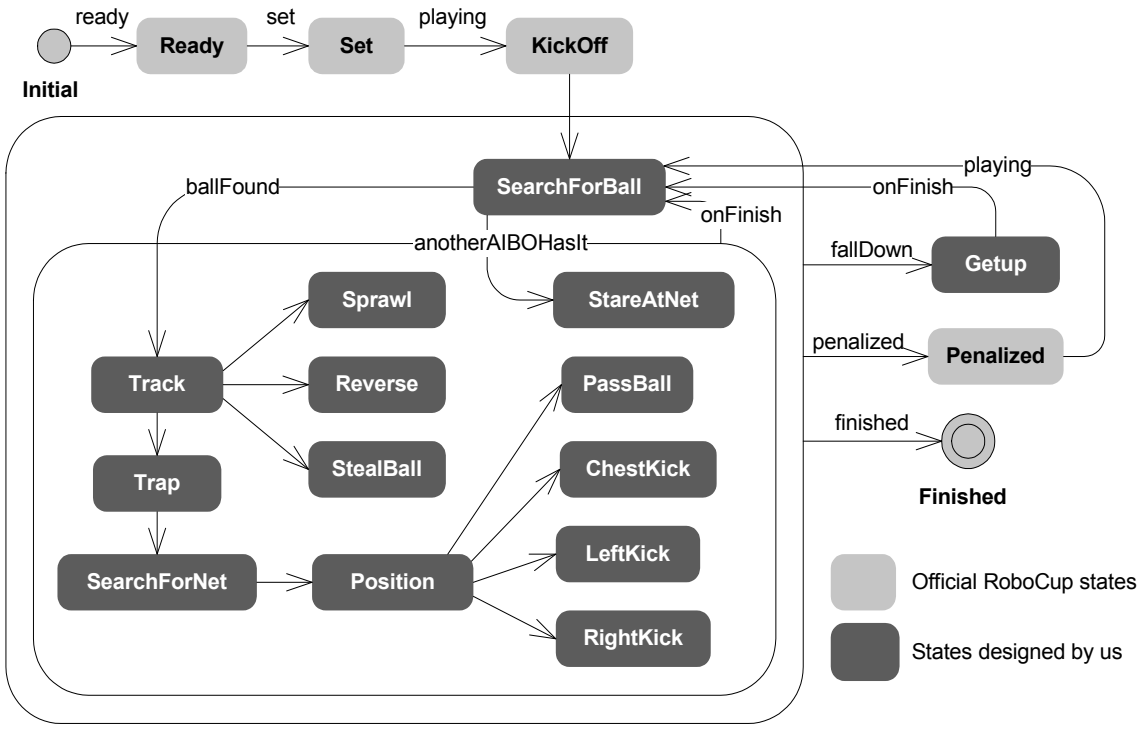

Figure 5. Soccer-playing state-diagram

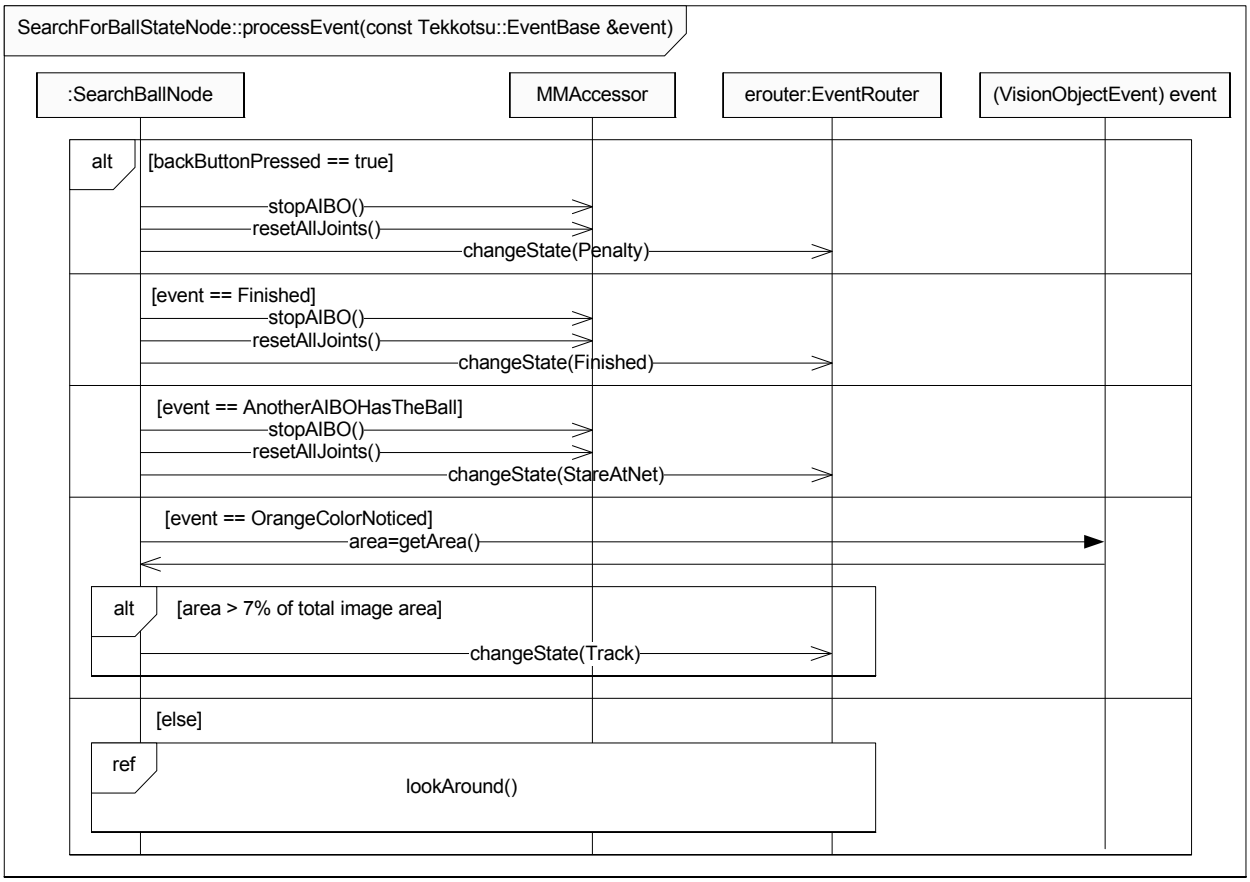

Figure 6. Sequence diagram for method processEvent()of *SearchForBall* state

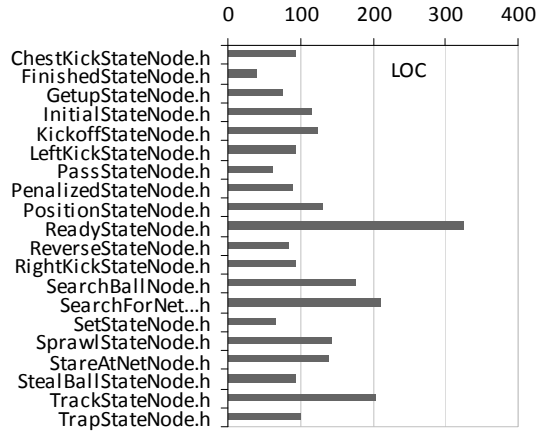

Figure 7. LOC measures of the 20 state-control classes

```
virtual void processEvent(const EventBase &event) {
    ...
    if( (event.getGeneratorID() == EventBase::finishedEGID &&
      event.getTypeID() == EventBase::statusETID) ) {
     cout << "Game Over." << endl;
     MMAccessor<WalkMC>(walker_id)->setTargetVelocity(0,0,0);
     MMAccessor<HeadPointerMC>(headpointer_id)->setWeight(0);
     erouter->postEvent(EventBase::soccerEGID, SoccerNS::GoFinished, EventBase::statusETID, 0);
     return;
    }
    if( (event.getGeneratorID() == EventBase::otherAIBOHasBallEGID &&
      event.getTypeID() == EventBase::statusETID) ) {
     cout << "Another AIBO has the ball. I will freeze (stare at the net)." << endl;
     MMAccessor<WalkMC>(walker_id)->setTargetVelocity(0,0,0);
     MMAccessor<HeadPointerMC>(headpointer_id)->setWeight(0);
     buttonPressed = false;
     searchControl=0;
     startTime=0;
     headTime=0;
     erouter->postEvent( EventBase::soccerEGID, SoccerNS::GoStare, EventBase::statusETID, 0);
     return;
    }
    bool seenBall = false;
    if(event.getTypeID()==EventBase::statusETID &&
        event.getSourceID() == SoccerNS::rc_orangeBallSID) {

     const VisionObjectEvent *ve = dynamic_cast<const VisionObjectEvent*>(&event);
     if(ve->getArea() > 0.07) // 0.07 is the empirically-calibrated value
     {
          cout << "AIBO sees the ball. Switching to track." << endl;
          searchControl = 0;
          seenBall = true;
          MAccessor<WalkMC>(walker_id)->setTargetVelocity(0,0,0);
          MMAccessor<HeadPointerMC>(headpointer_id)->setWeight(0);
          erouter->postEvent(EventBase::soccerEGID,SoccerNS::GoTrack,EventBase::statusETID, 0);
     }
     else
          cout << "Ball seen, but only has area of " << ve->getArea() << endl;
    }
    if (seenBall == false) // start searching in different directions, first lower left
    {
     if(get_time() - headTime > 1000)
     {
          switch(searchControl)
          {
          case 0:
               cout << "Looking Lower Left" << endl;
               searchControl++;
               MMAccessor<HeadPointerMC>(headpointer_id)->setWeight(1);
               MMAccessor<HeadPointerMC>(headpointer_id)->lookInDirection(1,1,-1);
               headTime = get_time();
               break;
                    ...
```

Figure 8. Example partial code listing