

Case Study: A Software Concept for Autonomous Robot

Dr. M. Popovic
University of Novi Sad
Trg D. Obradovica 6
21000 N. Sad, Yugoslavia
+381-21-58-165

Dr. V. Kovacevic
University of Novi Sad
Trg D. Obradovica 6
21000 N. Sad, Yugoslavia
+381-21-58-165

I. Velikic
MicronasNIT
Fruskogorska 11a
21000 N. Sad, Yugoslavia
+381-21-350-701

Z. Jurca
University of Novi Sad
Trg D. Obradovica 6
21000 N. Sad, Yugoslavia
+381-21-58-165

rt_pop@krt.neobee.net rt_kovac@krt.neobee.net

velikic@micronasnit.com zjurca@krt.neobee.net

www.micronasnit.com

ABSTRACT

Thanks to recent advances in technology, autonomous robots are becoming today commercially available products in area of consumer electronics, and there are a lot of ongoing research and development activities, in this respect, in both industry and academia. Control units of these autonomous robots are based on DSPs and microcontrollers, such as MAS and PUC families manufactured by Micronas GmbH, Germany. Software for such robots must cover multimedia, communications and control functions. This paper focuses on the problem how to partition these functions and how to organize the corresponding software components. Traditionally, multimedia software performs digital signal processing on bit-streams, communications protocols are implemented as finite state machines that react to events (communications messages and timer triggers), while control software reads sensors, processes these inputs in order to control the system, and writes calculated values to actuators. Time requirements for such systems include both soft and hard real-time limits. The question of programming language and OS is also discussed. The paper proposes a software concept for such a system. The concept is than illustrated on an example of autonomous robot Caddy, which carries golf bag, and follows its master.

Categories and Subject Descriptors

D.4.7 [Operating Systems]: Organization and Design – *Real-time systems and embedded systems.*

General Terms

General terms are: Design, Experimentation, Standardization, and Verification.

Keywords

Keywords are: autonomous robots, software, multimedia, communications, control, OS, soft real-time and hard real-time.

1. INTRODUCTION

Today autonomous robots are attracting more and more attention worldwide in both academia and industry. Contests of football robotic teams [7], and similar, are held annually and autonomous robots, as pets and home devices, such as dogs and vacuum cleaners, are already available commercially. Control units of these autonomous robots are typically based on digital signal processors (DSPs) and microcontrollers. In our current work we use MAS DSPs and PUC microcontroller families, manufactured by Micronas GmbH, Germany [8]. One of the most challenging issues, in this area, is how to develop, test, and verify software for such applications. A number of articles that address these issues have been published recently, but there is a need for additional research efforts in this area.

Software for autonomous robots, in area of consumer electronics, must cover multimedia, communications and control functions. This paper focuses on the problem how to partition these functions and how to organize the corresponding software components. Traditionally, multimedia software performs digital signal processing on bit-streams, communications protocols are implemented as finite state machines that react to events (communications messages and timer triggers), while control software reads sensors, processes these inputs in order to control the system, and writes calculated values to actuators. Time requirements for such systems include both soft (for most of the multimedia and communications software) and hard real-time limits (for some of the control behaviors), and it is a question how to satisfy them.

The question of programming language and OS to be used for such applications is, at least in our opinion, also open. Some of the authors are proposing, and using, their own proprietary operating systems and/or programming languages for their specific applications [4, 9-10, 13-14]. Our position is that in the area of consumer electronics there is a need for standardization through acceptance of de facto standards, such as C and Java.

The paper proposes a software concept for this type of system, as an answer to above open issues. The concept is strongly based on leJOS, an open source Java platform originally intended for LEGO Mindstorms [1], which we have successfully ported on PUC 303x microcontroller. For the sake of clarity, the concept is illustrated on an example of autonomous robot Caddy, which carries golf bag, and follows its master.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISPC '03, March 31 – April 3, 2003, Dallas, Texas.

Copyright 2000 ACM 1-58113-000-0/00/0000...\$5.00.

The text of the paper is organized as follows. In section 2 of the paper we describe three kinds of domain-specific components needed in modern autonomous robots for consumer electronics. These are robot control, communications protocols, and digital signal processing components. The purpose of this section is to highlight different paradigms used for the development of those domain-specific components. In section 3 we present our software concept that we use for the autonomous robots in area of consumer electronics. The concept is illustrated in section 4 on an example of autonomous robot Caddy prototype. Sections 5 and 6 contain closing conclusions and list of references cited in the text of the paper, respectively.

2. DOMAIN-SPECIFIC COMPONENTS

2.1 Control Software

The most widely used paradigm for autonomous robot control today is *Behavior Control*, which has been founded by Rodney Brooks [2-5] at the MIT Artificial Intelligent Laboratory. It originates from the study of insect world, and it was originally invented under the name of *Subsumption Architecture*. In the theory of behavior control the complete robot behavior may be specified by a set of individual behaviors of different priority. An individual behavior is a triple (*Condition, Action, Priority*). *Condition* is a condition that determines when to apply a given individual robot behavior (for example a light is too bright). *Action* specifies the action to be taken by the robot when the specified condition exists (e.g. turn away the camera from the light source to imitate the human reaction). *Priority* is a priority of an individual behavior. In subsumption architecture more individual behaviors may be triggered simultaneously (if the specified conditions are fulfilled), and the individual behavior with the highest priority will be executed first. While executing it will “inhibit” lower level behaviors. After the highest priority behavior has been conducted, the system proceeds with the next priority behavior, until no more active behaviors are pending. The conditions are typically detected through external physical sensors, but they may be detected also by internal virtual sensors, such as timers, which are kind of “time sensors”. The corresponding actions are normally conducted by the usage of actuators, such as motors etc.

In 1991, Brooks has invented a special programming language for behavior programming [4]. The *Behavior Language* is a rule-based real-time parallel robot programming language that compiles into a modified and extended version of the subsumption architecture. Last year, Brian Bagnall proposed a new platform for behavior programming based on leJOS [1]. leJOS is similar to Java SDK made by Sun Microsystems. The main difference between them is that leJOS has been made to run on a computer (LEGO Mindstorms RCX brick) with only 32 kB of RAM, so leJOS can be thought of as Java micro-micro edition. leJOS supports most of the Java features, including fixed priority preemptive scheduling of program threads including time-slicing (with a slice of 128 RCX brick instructions), Java event model (timer, button, and sensor listeners are supported), Java exception handling (to partially enforce programming by contract), and recursion (only 10 levels deep). leJOS currently does not support garbage collection (although the work on its implementation is in progress). leJOS Java API does not include standard Java API packages not relevant for robot programming (e.g. programming graphical components, creating applets, JavaBeans etc.), but it

provides some RCX platform extensions instead. The platform extensions include 3 output ports for motor control, 3 input ports to read sensors (specified by *type* and *mode* settings), RCX buttons, system time (in milliseconds), battery power indication, multiple programs, LCD, Speaker, IR communication, and timers support. Additionally, for the robotics programming it includes behavior control and some navigation classes.

Threads are very useful for robotics programming because each thread can be used to control a separate behavior. The Java runtime supports deterministic scheduling algorithm known as fixed priority scheduling. Each Java thread is given a priority between `MIN_PRIORITY` and `MAX_PRIORITY`. CPU scheduling is fully preemptive (the higher priority process is scheduled immediately). Thread can yield the CPU only to other threads of the same priority. When all of the runnable threads in the system have the same priority, round-robin scheduling is used. Additionally, leJOS supports time-slicing, which is optional in standard Java [16].

If a robot has to repeat an action periodically, *Timer* class and *TimerListener* interface may be used. The action to be executed periodically is placed in a body of the *timedOut()* function of an object that implements *TimerListener* interface. The period of an action and the timer listener object reference are the arguments of the *Timer* class constructor. The *timedOut()* function is called every time the timer fires.

Another important interface is a *SensorListener* interface. An object implementing this interface may be notified when a sensor’s input changes, if it registers to a sensor by calling the sensor’s *addSensorListener* function. A Sensor object notifies a sensor listener object by calling its *stateChanged* function.

The Behavior API is defined by *Behavior* interface and *Arbitrator* class. An object that implements the Behavior interface defines one individual robot behavior. The Behavior interface has the following three functions:

- *boolean takeControl()* – returns a Boolean value to indicate whether the behavior should become active.
- *void action()* – initiates an action when the behavior becomes active.
- *void suppress()* – it should immediately terminate the action initiated by the *action()* function.

The complete robot behavior is defined with an array of individual behaviors, see Figure 1. The priority of an individual behavior is determined by its array index. The higher index array number corresponds to the higher priority level.

Arbitrator object (an instance of *Arbitrator* class) regulates when each of the behaviors will become active. The Arbitrator class constructor sets a reference to an array of behavior objects, sets current behavior to `NONE`, creates, and requests the current behavior action thread (instance of *BehaviorAction* class). So the Arbitrator is executed in a main Java thread, while the current behavior *action()* function is executed in a separate thread. Those two threads have the same (primordial) priority, and since the leJOS supports time-slicing, they are concurrent.

In its *start()* function, which is typically called from the *main()* function, the arbitrator enters the infinite loop, where it calls

takeControl() functions of individual behaviors, one by one, starting from the end of an array of behavior objects (highest priority), and ending at the beginning of the array (lowest priority). If the *takeControl()* function of some individual behavior returns true, and if it is not NONE or current behavior, arbitrator waits for the current behavior's *action()* function to be finished (if it is not already finished), calls current behavior's *suppress()* function, updates current behavior variable, and calls behavior action thread's *execute(int index)* function to initiate new behavior (*index* is new behavior's index).

Those two threads are synchronized by the Boolean variable (critical section indicator) *done*. This variable is false while action thread executes *action()* function, and otherwise it is true. This means that *action()* is an atomic operation, i.e. it can not be interrupted by the *suppress()* operation.

Behavior objects use *Sensor* and *Motor* objects, and *Navigator* class, to control the robot. The *Sensor* object provides a means for reading data from input ports. leJOS normally supports three input ports (S1, S2, and S3), but more can be added if needed. There are two settings that must be specified for each sensor port: *type* and *mode*. The standard types of sensors are: light, touch, rotation, temperature, and raw (raw data between 0 and 1023). The standard sensor modes are: Angle, Boolean, Degrees Celsius, Degrees Fahrenheit, Edge counter, Percent, Pulse counter, and raw.

The *Motor* class is used to control a DC current to the output ports, which are connected to the motor drivers. Most of its functions are tailored toward controlling motors, but they may be used to control almost any actuator.

Navigator class provides the leJOS Navigator API, a convenient set of functions to control a robot. There are functions for moving robot to any location. A Navigator object automatically keeps track of the current angle and robot's (x, y) coordinates, and it can be used for any robot with differential steering. The *Navigator* class function members are: *forward*, *backward*, *travel*, *stop*, *rotate*, *gotoAngle*, *gotoPoint*, *getX*, *getY*, and *getAngle*.

It is important to emphasize that the leJOS behavior control API is a modified version of the model proposed by Rodney Brooks. His model prevents higher level classes from being used in individual behaviors. For e.g. the *Navigator* class access the motors directly, but this is forbidden by the original behavior control model. In the original model the motors are controlled exclusively by the lowest level behavior.

An example of a typical array of behavior objects and its relation to Sensor, TimeNavigator (an instance of *TimingNavigator* class, a successor of *Navigator* class) and Motor objects is shown in Figure 2. In this example there are four behavior objects: Move, LeftHit, RightHit, and ReturnHome. The behavior objects are listed in accordance to increasing order of priority, i.e. Move is the lowest priority and ReturnHome is the highest priority behavior. LeftHit and RightHit actually have the same importance, but since they must be members of the same array, they have different priorities. Move tries to direct robot to a given position, LeftHit and RightHit react to obstacles by backing off and changing robots angle, and ReturnHome is a timer listener, which brings robot back to starting position after a given timer expires. LeftHit and RightHit are sensor listeners, listening to

touch sensors connected to ports S1 and S3, respectively. TimeNavigator controls two motors (connected to ports A and C), and services navigation requests issued by behaviors.

2.2 Communications Protocols

Each communications software component requires a set of common routines, which are needed for the communication with other entities. These routines include buffer (*get*, *ret*), message (*send*, *receive*), and acyclic timer (*start*, *restart*, *stop*) management routines (available through the communications kernel API).

A communication protocol is a set of rules guiding the communication among related entities, and it may be modeled as a finite state machine (FSM). Typically, a FSM is implemented as a function containing two-level nested *switch-case* constructs (first switch-case selects the processing for the current state, while the nested one selects the processing of the received message) [15]. The FSM implementation function calls buffer, message, and timer service routines provided by the communications kernel API.

The layered communication architecture is modeled as group (family) of FSMs. In our practice [15] we are implementing a group of FSMs as a single monitor task, which executes an infinite loop with a series of function calls to functions implementing individual FSMs – we use so called nested monitors, or cooperative multitasking system. The implementation is justified with the nature of the protocol: it is a process with stable states, so it doesn't monopolize the processor. Of course, the processor must be selected according to specified communication system dimensions and desired QoS. Since the correctness of the implementation is an important issue, we have developed a methodology for the formal verification of the implementation of the group of FSMs [11]. We used our Bluetooth protocols stack implementation for the case study presented in this paper.

2.3 Digital Signal Processing Software

Typical digital processing application, such as MP3 player, is partitioned between a microcontroller and a DSP coprocessor. A microcontroller provides control functions (user interface: buttons and LCD, and MMC handling) and optionally less demanding DSP blocks, while DSP coprocessor performs heavy-duty processing of MAC-based DSP blocks.

Common DSP application includes A/D conversion, an algorithm, composed of a set of DSP blocks, to be executed for each signal sample, and D/A conversion. The main characteristic of the digital signal processing is that it is a data flow processing – each DSP block is defined as some transformation of data present at its input to resulting data present at its output. The whole processing must be finished before next signal sample is available, i.e. the real-time period is defined by the signal sampling period.

The resources (RAM and CPU clock) are determined based on the worst case analysis. The main problem is the tradeoff between the accuracy (quality) and the resources needed for the application (which determine price and power consumption).

Software is organized as a single monitor task (infinite loop with sequential series of DSP steps) that rests upon I/O interface's

handling software (interrupt service routines, DMA handling, and buffering – typically double buffering).

3. A SOFTWARE CONCEPT

In previous section of the paper we described three different paradigms used for the development of three different kinds of domain-specific components (robot control, communications, and DSP). In this section of the paper we present a software platform and the method of assembling of those domain-specific components into integrated software for the autonomous robots in area of consumer electronics. The assembling method respects all of the constraints imposed by the domain-specific components, and corresponding paradigms. To summarize, constraints enforced by the domain-specific components are as follows:

- The robot control component requires a software platform for robotics applications (e.g. leJOS). Most of its tasks are hard real-time tasks, with typical frequencies of 10 to 25 Hz (i.e. period of 40-100 ms) [5].
- The communications component requires a software platform that supports buffer, timer, and message management (a set of common communications routines – communications kernel). Typical implementation language is C. The group of communications protocols may be implemented as a single soft real-time task, which must perform required QoS.
- The DSP component typically requires an embedded hardware-software platform that is completely dedicated to digital signal processing. The most frequently used programming languages are assembler and C. It is implemented as a single infinite-loop monitor task, where the algorithm complexity determines the footprint of the processor.

In our approach we see the leJOS as software platform that satisfies all of the above constraints, primarily through its support for the native (assembler and C) functions, and fixed priority preemptive scheduling with time-slicing. Java support for native functions provides a means for seamless integration of domain-specific components, because there is no need to change software platform or programming language. This is very important because all of the communications and DSP components may be integrated with only minor changes of their source codes. On the other hand, Java thread and event models enabled us to formulate the method of assembling domain-specific components into the target software.

The assembling method is illustrated in Figure 3. The domain-specific components execute as Java threads. The highest priority threads are behavior Arbitrator and Action threads (if there is a need, more threads could be added at this priority in certain applications). The original Bagnall's behavior Arbitrator can not be used in our approach. Two modifications to the original leJOS behavior model are needed, as follows:

- (1) The Arbitrator's *start()* method must be implemented as a run-through code rather than falling into infinite while-true loop. This can be achieved simply by

deleting “while (true) {” and corresponding “}” lines of original code.

- (2) The Arbitrator's *start()* method is called from newly added timer listener object's *timedOut()* method rather than from Java application's *main()* function. The corresponding timer should fire with application-specific period (for example 40 ms). This means that the behavior control is time-driven software. (Alternatively, it may be implemented as event-driven software if it is called by sensor listener(s) or as time-and-event-driven software if it is called by sensor(s) and timer listeners).

The communications component runs as a middle priority thread see Figure 3. Similarly to Arbitrator's *start()* method, communications monitor task code must be implemented as a run-through code, rather than infinite loop with series of function calls to functions implementing individual FSMs. More precisely, once requested, communications thread will execute its *run()* method (a series of calls to FSM functions) and terminate. This is necessary in order to provide certain time budget for the lowest priority thread (executing DSP code).

The communications component may be driven by the timer and timer listener pair of objects (time-driven solution), or by sensor and sensor listener pair of objects (event-driven solution) as shown in Figure 3. In the first case, a timer object periodically calls *timedOut()* method of timer listener object, which in turn calls *start()* method of the communications thread object. The timer period depends on the desired communications speed.

The second solution requires modification of the original leJOS *Sensor* class. New type of sensor, a “communications controller”, must be introduced. Data present at the input port that is connected to a communications sensor is equal to the contents of the communications controller (e.g. Bluetooth Baseband Controller) status register. The status register typically consists of individual status bits, such as “Transmit FIFO Empty”, “Receive FIFO Full”, etc. If the status of the communications controller changes, Sensor object calls *stateChanged ()* method of the corresponding sensor listener object, which in turn calls communications thread's *start()* method. Once started the communications thread executes its *run()* method, by calling native run-through C code, and terminates. It should be noted that in the second solution we still need a timer object that activates C native code to maintain communications timers.

The lowest priority thread executes infinite loop of native C code, which performs digital signal processing related tasks. It is always active and almost always running, with short interruptions by the higher priority threads. The interruptions are always finite and limited to relatively small time intervals, because both behavior control and communication protocols are processes with stable states. The longest interruption time interval equals to sum of the longest behavior *action()* and communications FSM state transition. Since DSP thread typically consumes most of the CPU time, its complexity determines the CPU footprint.

4. CASE STUDY: ROBOT CADDY

The software concept proposed in this paper is validated on an example of autonomous robot Caddy prototype, which has two independent motors, digital compass sensor (currently not used) and a simple camera (commercially available on a PC market). In

autonomous mode of operation Caddy follows its owner by “watching” the owner through the camera. The robot control unit is built around Micronas PUC 303x microcontroller. The Caddy’s owner can control the Caddy manually, using pushbuttons on a remote controller, or through voice commands recognized by the remote controller. The Caddy and its remote controller are connected through the Bluetooth wireless technology.

The prototype of the robot Caddy is shown in Figure 4.

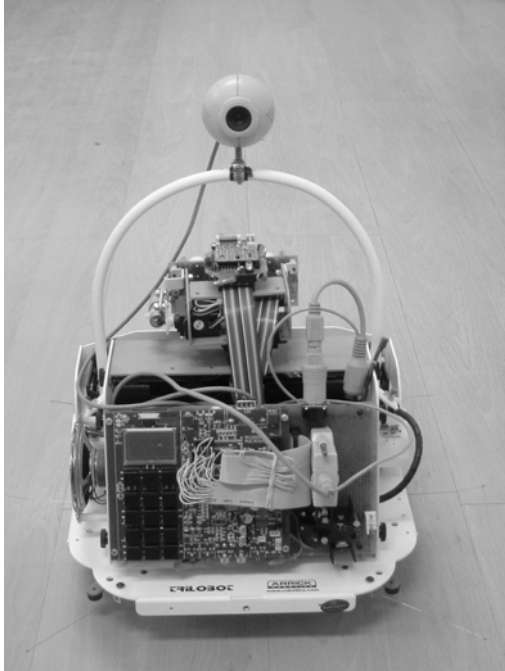


Figure 4: The robot Caddy prototype.

5. CONCLUSIONS

In this paper we have described three different paradigms used for the development of three different kinds of domain-specific components of autonomous robots in area of consumer electronics, namely: robot control, communications, and DSP software components. An open issue addressed by this paper is how we integrate those different types of components, which impose different constraints on platform, language, and time. As a solution we proposed a software organization concept that covers a software platform and the method of assembling of those domain-specific components into the target software. The assembling method respects all of the constraints imposed by the domain-specific components, and corresponding paradigms. The concept has been successfully validated on the robot Caddy prototype.

6. REFERENCES

- [1] Bagnal, B. Core Lego Mindstorms Programming, Prentice Hall PTR, Upper Saddle River, NJ 07458, 2002.
- [2] Brooks, R.A. A Robust Layered Control System for a Mobile Robot, A.I. Memo 864, MIT Press, 1985.
- [3] Brooks, R.A. A Robot that Walks; Emergent Behaviors from Carefully Evolved Network, A.I. Memo 1091, MIT Press, 1989.
- [4] Brooks, R.A. The Behavior Language; User’s Guide, A.I. Memo 1227, MIT Press, 1990.
- [5] Brooks, R.A., and Stein, L.A. Building Brains for Bodies. Autonomous Robots, 1 (1994), 7-25, Kluwer Academic Publishers, 1994.
- [6] Bryan, C.F. Design and Implementation of a Supervisory Software for Intelligent Robot. Proceedings of ’99 ACM Southeast Regional Conference, 1999.
- [7] Lenser, S., Bruce, J., and Veloso, M. CMPack: A Complete Software System for Autonomous Legged Soccer Robots. Proceedings of AGENTS ’01 (Montreal, Quebec, Canada, May 2001), ACM Press, 204-211.
- [8] Micronas company presentation is available at: www.micronas.com
- [9] Payton, D.W., and Bihari, T.E. Intelligent Real-Time Control of Robotic Vehicles, Communications of the ACM, Vol. 34, No. 8, Aug. 1991, 48-63.
- [10] Pembeci, I., Nilsson, H., and Hager, G. System Presentation – Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. Proceedings of PPDP ’02 (Pittsburg, Pennsylvania, October 2002), ACM Press, 168-179.
- [11] Popovic, M., Kovacevic, V., and Velikic, I. A Formal Software Verification Concept Based on Automated Theorem Proving and Reverse Engineering. Proceedings of ECBS ’02 (Lund, Sweden, April 2002), IEEE Press, 59-66.
- [12] Rylatt, R.M., Czarnecki, C.A., and Routen, T.W. Learning Behaviors in a Modular Neural Net Architecture for a Mobile Autonomous Agent. Proceedings of EUROBOT ’96 (Kaiserslautern, Germany, October 1996), IEEE Press, 84-88.
- [13] Schwan, K., Bihari, T., Weide, B.W., and Taulbee, G. High-Performance Operating System Primitives for Robotics and Real-Time Control Systems, ACM Transactions on Computer Systems, Vol. 5, No. 3, Aug. 1987, 189-231.
- [14] Stewart, D.B., and Khosla, P.K. Rapid Development of Robotic Applications using Component-Based Real-Time Software. Proceedings of IEEE International Conference on Intelligent Robots and Systems (Pittsburg, Pennsylvania, August 1995), IEEE Press, 465-470.
- [15] Velikic, I., Basicovic, I., Popovic, M., and Kovacevic, V. Comparative Analysis of Different Finite State Machine Design and Implementations for Communication Protocols. Proceedings of the 14th International Conference on Systems Science (Wroclaw, Poland, Sept. 2001), Wroclaw Tech. University Press, Vol III, 62-69.
- [16] Sun Microsystems, Inc. The Java Tutorial. <http://java.sun.com/docs/books/tutorial/>

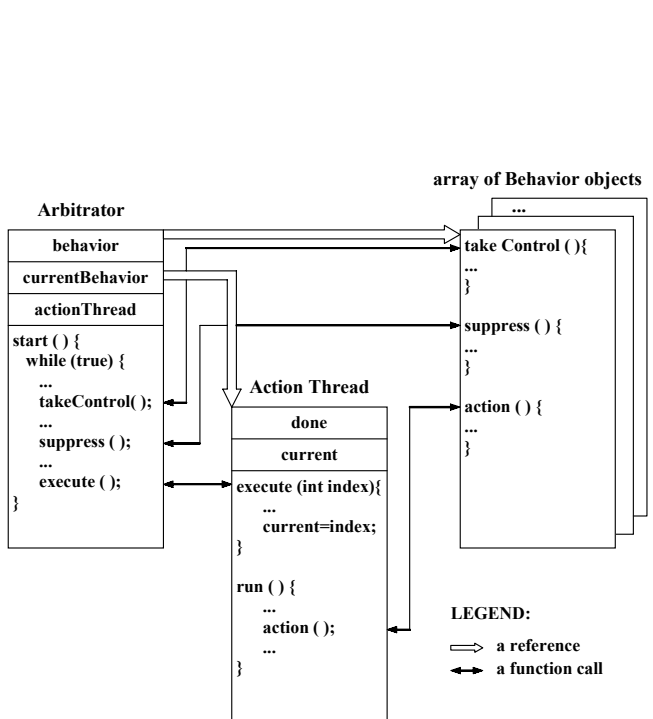


Figure 1: The structure of the behavior control implementation.

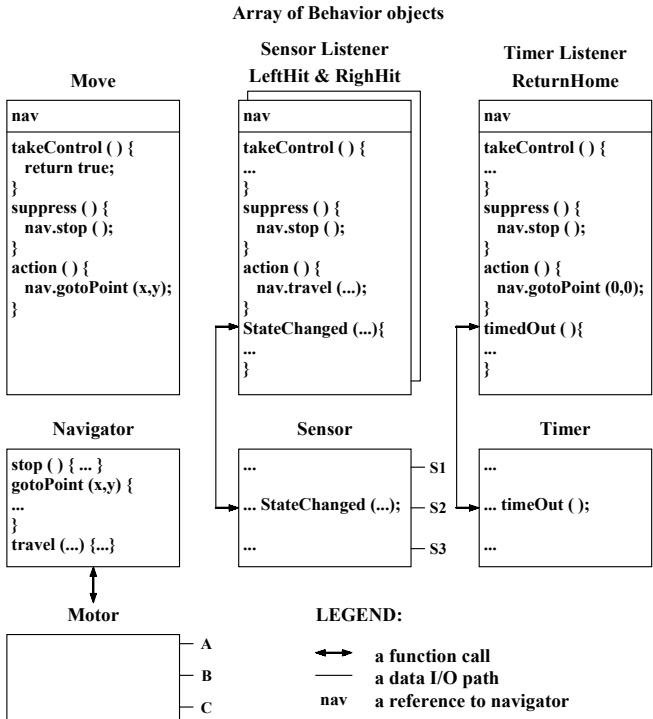


Figure 2: An example of an array of behavior objects.

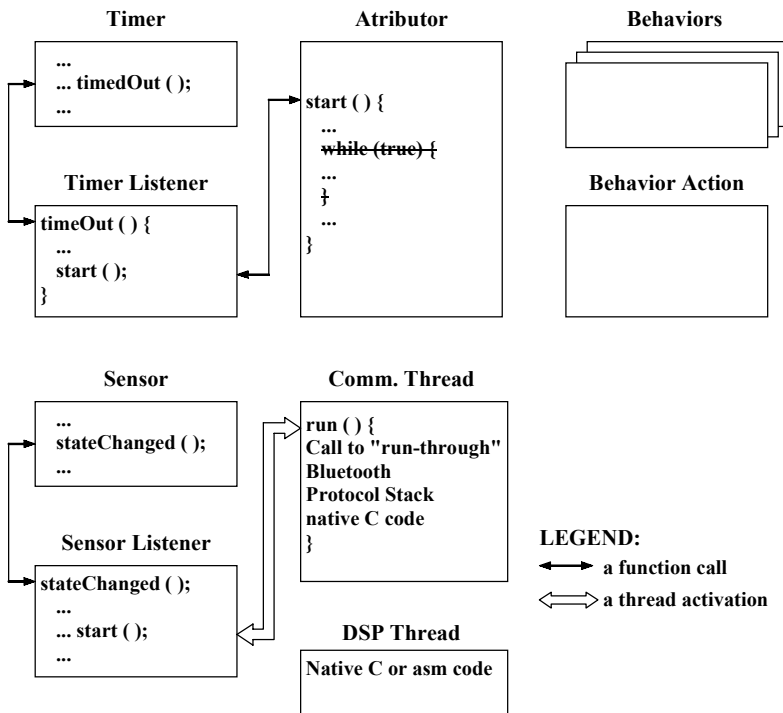


Figure 3: The proposed software concept illustration.