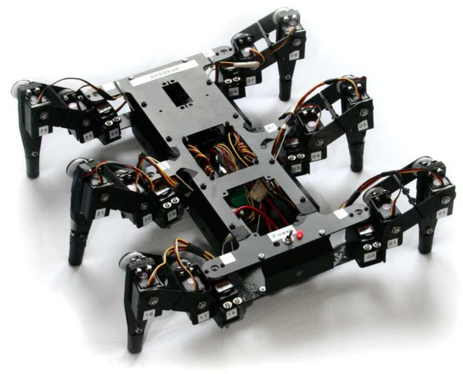
Chair of
Software Engineering



SCOOP for Robotics

Implementing bio-inspired hexapod locomotion

Research Project

By:	Ganesh Ramanathan
Supervised by:	Dr. Sebastian Nanz
	Benjamin Morandi
	Scott West
	Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

Robotics engineering is a unique inter-disciplinary field where specializations like Mechanical, Electronics and Software Engineering merge coherently to provide the most tangible result – physical movements and actions with a purpose. For software engineers, this provides an opportunity to see their program in action, literally. Hitherto use of conventional programming paradigms like sequential code, Object oriented model and multi-threading has managed to provide the required control program implementation. But as computing hardware used on robots continue to increase in sophistication, engineers now have the opportunity to implement more complex behaviors in the robots. Complex behaviors, invariably, contain intricate concurrency and coordination scenarios. The oft used multi-threaded programming is focused on concurrent execution of routines with the threads coordinating amongst themselves using primitives like semaphores etc.

Behavioral models usually follow natural language thinking and tend to be easily represented in terms of inter-linked and coordinating objects. Here we see that the concurrency aspect of the robotic behavior can no longer be described (or formulated) using just the object model – we need an additional description (and thought process) to implement concurrency and coordination. Such a separation of thought process is not just unsettling for the robotics engineer, but is also lamentable. While multi-core and many-core processors are becoming the order of the day, the programming model remains parochial in comparison. Object-oriented concurrency, which has been for long a “subconscious longing” of programmers, is fast emerging as a reality. In this research we assert that Object Oriented Concurrency, as implemented by SCOOP (Systematic Concurrent Object Oriented Programming) is a paradigm that provides opportunity for “natural and unified thinking” in robotics programming. SCOOP in addition to being a framework for object-oriented concurrency can also be viewed as contract based concurrency. This is a powerful combination for programming in general and robotics programmer in specific (as the concern for correctness and safety is perhaps higher in their realm).

In this research, we have taken the example of implementing a control program for walking of a Hexapod robot. The problem is solved using different variants (programming methodology and specific technology) including SCOOP. One of the implicit objectives of this research is that it kindles further interest amongst researchers and student to explore SCOOP.

Acknowledgement

The Chair for Software Engineering at the ETH Zürich has been very kind in giving me the opportunity to work on a cutting edge concept like SCOOP. Moreover, to support a project involving inter-disciplinary aspects (as in robotics) requires tremendous commitment and confidence towards the fundamental basis of SCOOP. I am impressed that the group behind SCOOP has consistently shown their belief in the concept – which infuses faith in application developers like me. At the end of this project I realize that SCOOP is being nurtured by very clever people who not only know the problems of concurrency but are also attempting to solve it in an elegant manner. I thank my supervisors Dr. Sebastian Nanz, Benjamin Morandi and Scott West who have patiently guided me through difficult stages of this development and Dr. Bertrand Meyer for this wonderful experience and learning.

Doing a research project in addition to having a family and a regular job is an intricate jugglery of time and energy. This usually resulted in my taking time away from family, something which I feel bad about. I lovingly thank my wife who not only supported me patiently while I worked on this project but also encouraged me to go further. And all this while she single-handedly managed our impatient 3-year old son and plus a new born baby. Now, that's a concurrency challenge I hold in awe. I should also thank my son who questioned as to why the robot still moved its legs when picked up from the ground. But then he is in the phase of life where every sentence starts with a why (we lose this habit as we get educated!). As an afterthought I have added this scenario while trying to explain the advantage of SCOOP.

CONTENTS

Figures

1 Introduction

1.1 What this project is about...

Simple Concurrent Object Oriented Programming (SCOOP) is one of the seminal works in object-oriented concurrency. Apart from making concurrency integral part of the object model, it also couples Design by Contract philosophy (Nienaltowski & Meyer, 2007). The concept of object-oriented concurrency is indeed attractive to many programmers, especially those who need to keep the object model as the only view of their design. Robotics programmers deal with tangible entities like sensors, actuators and aggregates of such in their software. These entities are, of course, concurrently operating and coordinating with each other. Given such a domain, object-oriented concurrency is a perfect paradigm to base the solution model. For readers who are not familiar with SCOOP, we recommend reading first the section “SCOOP for Mortals” to get a fast-track introduction.

The aim of this project is to use the SCOOP methodology of concurrent programming in a robotics control example. Based on a prior experience of using SCOOP for robotics (Nienaltowski P. , 2007), we wish to formally study its attractiveness in handling concurrency and coordination. Hexapod is an interesting example of robotic walking machine. We could have as well chosen any other robotic device – arm robot, quadruped, wheeled robot etc., but walking robots hold a certain fascination and does have cunning coordination requirement. In addition to using SCOOP to program the hexapod and comparing it against some standard programming paradigms, this example will serve as a teaching aid to encourage students of concurrent programming to think in a different dimension (when using SCOOP). Such examples help in breaking away from unjustly writing SCOOP programs that looks like another flavor of multi-threaded program. For those who have spent a long time solving problems like the producer-consumer, Santa – Elves – Reindeer, Dining Savages etc. using Multi-threading concepts, it is often difficult to adapt to the object-oriented way of concurrency. The hexapod example looks into the biological model of walking and attempts to provide a solution that not only looks and feels object-oriented but also feels “natural”. That in essence is what this project is about.

1.2 ...and what it is not about

Hexapods, walking robots, insects, and optimal algorithms for walking are very specialized topics on which there are numerous researches being done and papers being published. This project is not attempting to propose a better algorithm or biological model. It tries to draw inspiration and ideas from what has been done but is not in that league. For example, hexapod walking already has a solution model called WalkNet (Durr, Schmitz, & Cruse, 2004). WalkNet models the behavior of the legs and their coordination as instances of finite state machines which coordinate with each other. Its proposed mechanism is a high-level description of hexapod walking and does not describe the actual concurrency mechanism. It should be constantly kept in perspective that SCOOP is an extension to provide object-oriented concurrency in the Eiffel language. But, this work does bravely put forward that solution like WalkNet or other robotic programming models can take advantage of the SCOOP way of thinking in its implementation – which is to remain object-centric from the perspective of concurrency.

Also, this research is not about undermining all other programming models in favor of SCOOP. I have made sincere efforts while working with different variants to achieve the best possible solution (almost with a mindset that it is the only programming paradigm available to me). SCOOP is about providing a better way of writing concurrent programs – so it is no surprise that it comes up with some winning arguments.

2 Hexapod Locomotion

The similarity of the walking systems in the cat and the cockroach suggests that the number of ways of optimally constructing a walking system is quite limited. - Keir Pearson (1976)

2.1 Introduction

Study of locomotion in living beings is an amazingly exciting subject. Researchers have been studying subjects ranging from the microscopic Amoeba, insects, horses to humans (on legs, skates, pogo-stick inclusive). Meanwhile, people at the robotics community have been following these researches and re-using it to develop practical “walking-machines”. Like everything to do with Artificial Intelligence, there are also attempts to solve the walking problem disregarding the biological aspect. There are differences in approach and philosophy, but fortunately there has been no recorded case of a violent clash.

First, a short explanation on why particularly Hexapod (six legs) instead of biped, quadruped, centipede or millipede (or even wheeled motion which is not found in nature). Hexapod locomotion offers a combination of simplistic sensory mechanism coupled with well documented gait mechanism (i.e. coordination of legs). More than six legs, like in centipede or millipede, simplifies the problem to a level where we will not strain the intelligence of concurrency and coordination. Less than six legs, like in quadruped or biped requires balance sensation coupled with “real-time” like processing – which is an overdraw for SCOOP (not the concept itself, but the infrastructure it uses for implementation). Hence, the choice of hexapod fits this experiment optimally. In case for a future research the choice of Quadruped seems better the Hexapod hardware can be re-used (by having two middle legs fold in!).

2.2 Mechanics of walking

Locomotion of any entity involves transporting the body while negotiating the terrain. Nature has constructed all biological entity with legs to enable locomotion. As the number of legs in a creature increase, its capability to negotiate difficult terrains also increases while the agility decreases (imagine a human rock climber who has eight legs instead of two). Hence depending upon the environment in which the creature thrives and its motive in life, nature has endowed it with appropriate number of legs. The hexapod, needless to say, is a creature consisting of six legs. Insects like the Cockroach and the Stick Insect are classical examples.

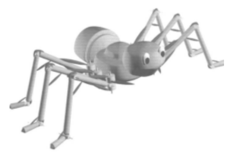


Figure 1: Hexapod insect (fictitious)

Each leg of the hexapod is constructed of three essential joints which are linked by three segments. The segments are powered by muscle pair and they have some form of sensory mechanism to provide positional feedback. The leg itself also has a sensor to determine the load borne by it.

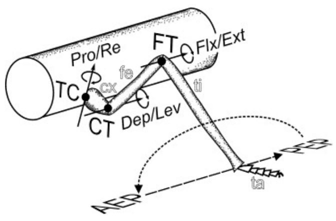


Figure 2: Hexapod leg (cx=Coxa, fe=Femur, ti=Tibia and ta=Talon)

The fundamental motions of the leg are protraction and retraction. Protraction involves lifting the leg from the Posterior Extreme Position (PEP) and swinging it to Anterior Extreme Position (AEP). It is executed by the Levator muscle raising the Femur (fe) segment (about the CT joint) followed by the Protractor swinging the Coxa (cx) forward (about the TC joint). The Depressor pushes the leg back on ground on reaching the AEP, after which retraction is triggered. Retraction causes just the Coxa to rotate (while keeping the feet on ground) resulting in the body being propelled forward. Retraction continues till the PEP is reached after which the leg is ready to protract again. The distance between AEP and PEP is commonly known as the Stride.

2.2.1 Gaits – The Tripod gait

Literatures (such as (Porcino, 1990)) on hexapod locomotion describe three main gaits (or patterns of leg movement) – the Ripple, Wave and Tripod. The Tripod gait is employed for high speed locomotion (like when either being chased by a predator or when chasing a prey). In tripod gait locomotion, three legs remain on ground while other three are up in air swinging. Since stability requirement dictates that the Center of Gravity (COG) should be within the bounds of the legs on ground, the tripod sets are formed as shown below.

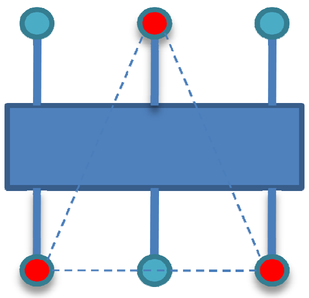


Figure 3: Tripod legs

Tripod gait is the most precarious gait for the hexapod as it cannot execute gaits in which just two legs are on ground (though it is possible geometrically if the insect's mass is evenly distributed across the vertical axis). It is easy to deduce the requirements (constraints) for Tripod gait (we limit ourselves to walking on even and flat surfaces):

1. A Tripod group may lift only if the partner group is planted on the ground.
2. Protraction starts only on completion of retraction and vice versa (this rule is true for all gaits)
3. Retraction of a leg group may only start when the partner group is raised (this is to avoid dragging of legs of the partner group).
4. Protraction should end (i.e. the legs should be depressed) only when the partner groups retraction ends (again, to avoid dragging).

The above four rules are sufficient to cause a hexapod to execute optimal Tripod walking on a flat surface. Change in direction of motion is effected by unequal retraction on the two sides. A simplified turning can take place by not moving the vertex leg of the tripod and only executing protraction-retraction on the base legs.

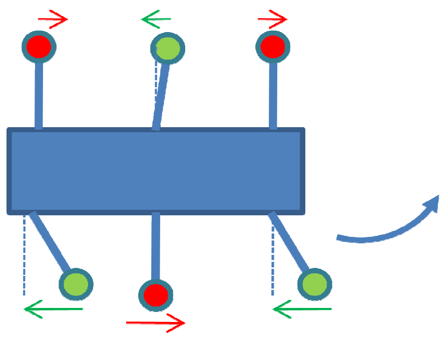


Figure 4: Turning of a hexapod

The speed of retraction determines the speed of locomotion (and hence the protraction has to occur at speed greater than or equal to retraction). Practically, one (and only) tripod-set of legs is retracting at any given time causing the hexapod to move "smoothly" – which is not true in exact sense there is finitely small delay in retraction to start because it has to "wait" for the partner leg to lift (in its protraction).

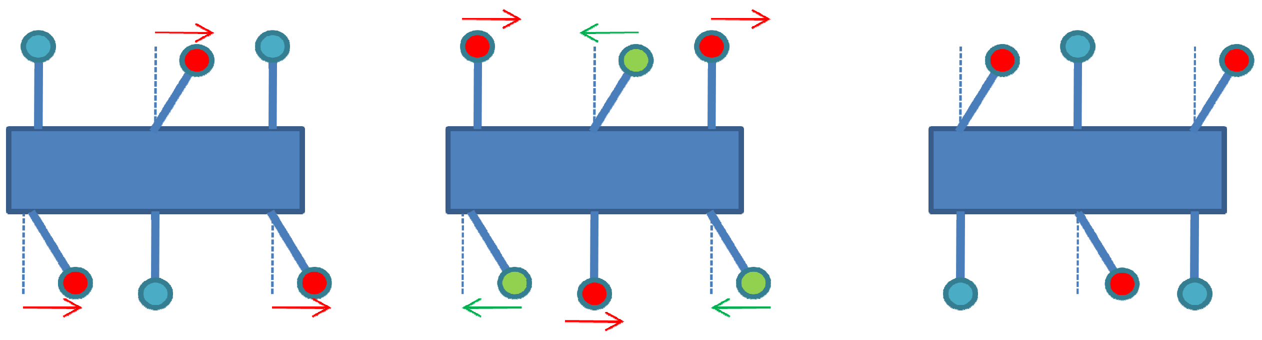


Figure 5: Tripod gait walking

2.2.2 Other Gaits – Ripple and Wave

A concise description of these two gaits are reproduced hereby from a web resource (Oricom)

*In the **Wave Gait**, all legs on one side are moved forward in succession, starting with the rear-most leg. This is then repeated on the other side. Since only 1 leg is ever lifted at a time, with the other 5 being down, the animal is always in a highly-stable posture.*

One conjectures that the Wave cannot be speeded up very much. If you try to shorten the suspension phases, the steps will get shorter, and/or the legs will have to pump faster. If you try to overlap the suspension phases, this would entail lifting adjacent legs simultaneously, and lead to partial collapse of that part of the body.

*The final stride is the **Ripple Gait**. At first glance the timing of this gait looks somewhat complicated, however, the key to understanding is to recognize that, on each side a local wave comprising non-overlapping lift phases is being performed, and that the 2 opposite side waves are exactly 180 degrees out of phase with one another. For instance, if L3 and R3 are considered to represent the start of each local wave, then notice that R3 starts to move exactly in the middle of the L3-L2-L1 side wave.*

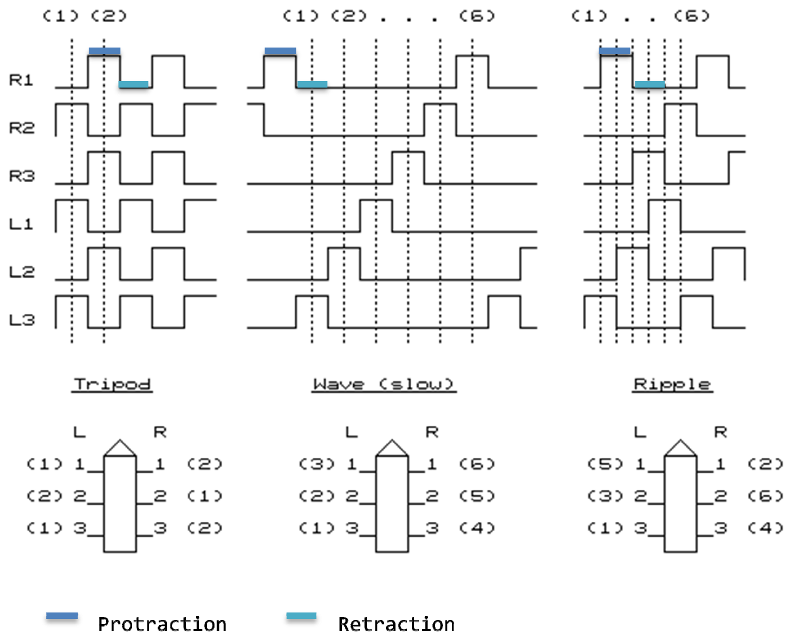


Figure 6: Gait sequences

2.3 Sensory and Motor components

Locomotion is achieved by neural circuits that provoke muscular actuation after considering sensory inputs. The six hexapod legs are similar in its construction although the geometry of the legs may vary – for example in some insects the hind legs are longer. Figure 7 shows the essential components of the leg:

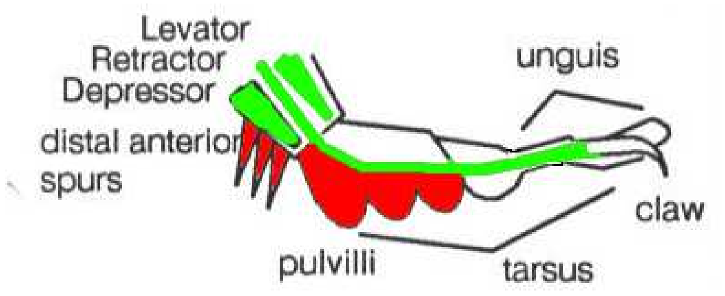


Figure 7: Sensory-Motor mechanism in a hexapod leg

The Levator and Depressor cause the leg to be lifted from the ground or pushed down to the ground. The Retractor causes the leg to be pulled back to the PEP (and also causes the claw to retract). The Distal Anterior Spurs sense the angle or leg with respect to the body while the Pulvilli act as load sensors for the leg.

2.4 Biological Model of Leg coordination

One of the most descriptive works published on the subject of biological models for hexapod walking (Porcino, 1990) describes the coordination in form of Pattern Generators and Inhibitory Links. Essentially, the Central Pattern Generator (CPG), which is a coupled set of neurons, provides periodic electrical pulses which are 90 degree out of phase with respect to each other.

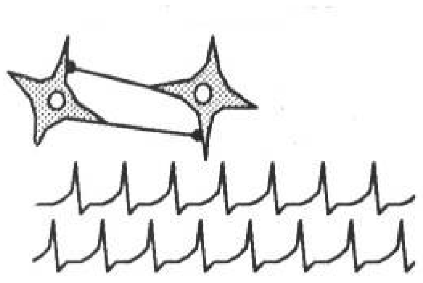


Figure 8: Pattern generator

These pulses serve to trigger the Motor mechanism to perform protraction-retraction cycle. The pulses are generated continuously; however the Motor mechanism requires certain conditions to be satisfied before an action is executed. These conditions are enforced as “inhibitions” or gate that block signal transmission. For example, a protraction invoking pulse will be blocked by the sensory neuron that originates from the load sensor if it recognizes a high load (i.e. when one of the neighboring legs is raised).

There are several hypotheses about the actual implementation of neural logic in biological entities. It is useful to study the simplest one as it serves to understand the concept of provocation and inhibition links (which is fundamentally used in various combinations in other hypotheses).

Figure 9 below summarizes the excitation and inhibition behaviors – the arrows indicate provoking signals while the circles indicate the inhibitions. The central pattern generator provokes the swing (protraction) to occur, provided that it itself is provoked by the backward angle sensor and not inhibited by the forward angle sensor. This occurs when the leg is in the PEP.

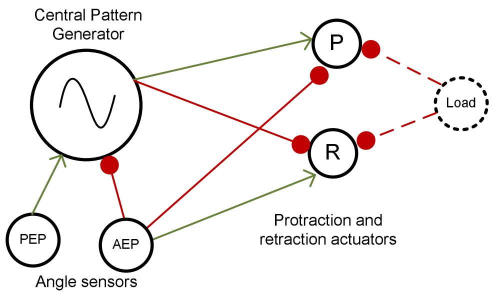

Central Pattern Generator
P
Load
R
PEP
AEP
Protraction and retraction actuators
Angle sensors

Figure 9: Simplified model of coordination

The flexor motor neuron responsible for the swing action is inhibited by the forward angle sensor – note that this in addition to the inhibition action of the forward angle sensor on the central pattern generator. Hence after the flexor has been provoked by the CPG, it continues to swing the leg until the forward angle sensor begins inhibition (i.e. AEP is reached). The retraction is triggered by the forward angle sensor only (i.e. when the leg has reached the AEP, it is designed to begin retraction). Once the leg has reached the PEP, the backward angle sensor provokes the CPG again, and the cycle repeats itself.

A more elaborate interaction between the sensory and motor mechanism is illustrated in figure below:

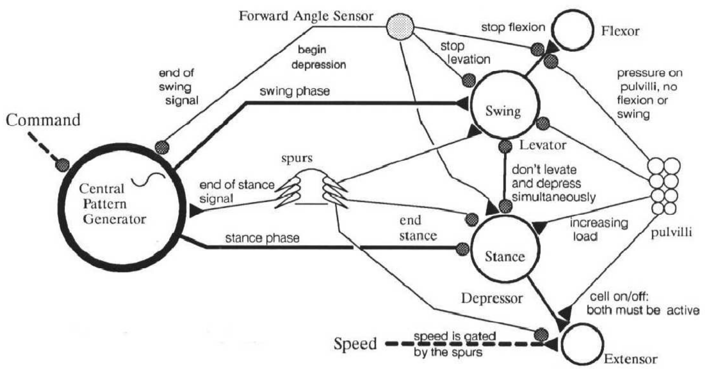


Figure 10: Extended model of coordination

On paying closer attention, we can question the existence of the Central Pattern Generator on each leg. The reflex actions caused by the coupling of the sensory and motor neuron are sufficient to produce walking of the leg. But the walking of the whole hexapod (i.e. all six legs together) results in a clumsily orchestrated locomotion. There is frequent stumbling and recovering (caused due to “race condition” like situation when two legs think they are ready to protract). For this purpose, each leg has a CPG which functions like a pacemaker. CPGs of all the legs are interlinked in a ladder like grid (see figure below).

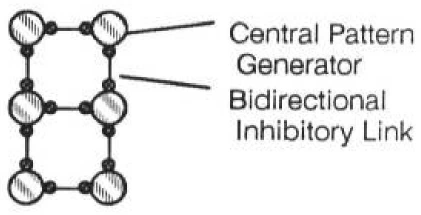


Figure 11: Bi-directional link between CPGs

A complete representation of the model is shown in the figure below:

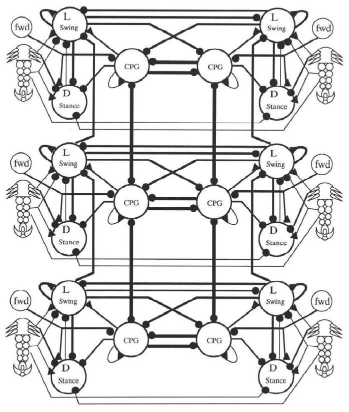


Figure 12: Complete representation of network

Between the networked CPGs there are several rules in action. The fundamental target of these rules is to preserve the stability of the body – for example, it prevents adjacent legs from lifting simultaneously. Several other rules are added to make the walking smooth and efficient. A detailed description of these rules can be found in (Durr, Schmitz, & Cruse, 2004).

2.4.1 Biological model to Object model

Object-oriented thinkers can immediately identify various entities in the hexapod that are appropriate to be modeled as object types and work out there inter-relationships (Body *containing* Legs, each Leg in turn *contains* Sensors and Actuators, and so on…). We are more or less adept in this translation. But when it comes to explaining concurrency and coordination in our OO model, we need another sheet of paper. Which is not so in the biological model (Figure 10, for example, seems sufficient). It doesn't take long to realize that this is because we do not have (yet) anything well known as "Object-Oriented Concurrency".

For the moment, we will close this section by hinting that there is a sort of contractual interaction between the components of the biological model – inhibitions are states where the contract is not satisfied and hence further execution of the action has to held up until the contract is satisfied. Such "contract based coordination" is happening between autonomous and concurrent entities. We shall reflect upon this in later sections.

3 Hexapod Robot – Hardware

No brain in nature comes without a body and hence no artificial intelligence shall be left without one - Josef Schmitz

This section describes the robot hardware in an abstraction that is required to understand the implementation of the control program. The exact details of the construction and maintenance are segregated out to Appendix A of this document.

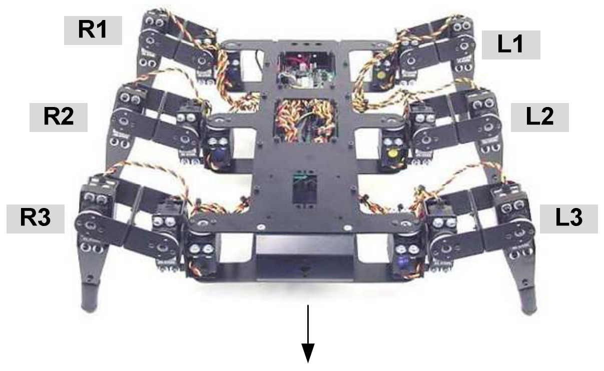


Figure 13: Hexapod robot

3.1 Sensory and Motor devices

Each leg of the robot consists of three servo motors and represents the muscular actuator for the three key segments (coxa, femur and tibia) found in a living hexapod. Servo Motors have a self-contained control loop mechanism that ensures that the commanded position is achieved and maintained. This implies that the control program needs to provide the position requirement to the Servo Motor and can be assured that the stand-alone loop executes the command.

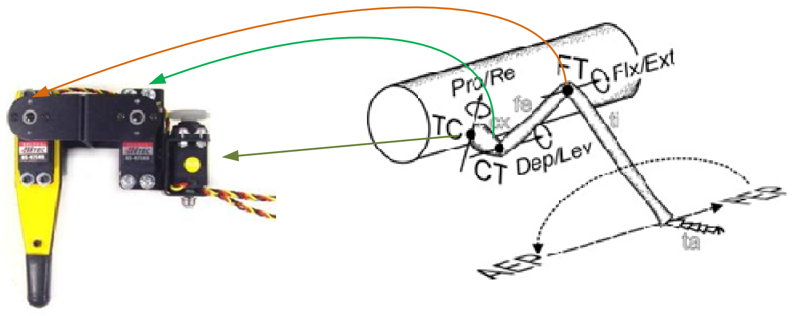


Figure 14: The robot leg vis-à-vis a real hexapod leg

In order to determine if the leg is planted on the ground or not, a force sensor is placed on the foot of the two middle legs (R2, L2). When pressure is applied on the force sensor, the analog value read changes from 255 (no force) to 0 (full force). For the hexapod, when the load is shared by all legs, the force sensed by the sensor is around 180. These values may vary with usage and hence the control program should provide for a way to calibrate the value. An easy method is to hold the robot in air during initialization phase and ask the program to note down the zero- load value of sensor.

Finally, the confirmation of the leg reaching the AEP / PEP is provided by a position feedback from the servo (Angle Sensor). This signal feedback is taken "Coxa" servo motors of R2 and L2.

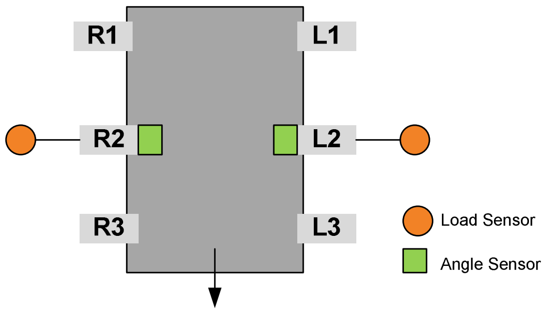


Figure 15: Placement of sensors on hexapod robot

Why just limited sensors?

Our objective is to implement the Tripod gait. In order to save the cost and complexity of the hardware, we try to limit to the bare minimum needs – just good enough to test the concept. In Tripod gait, we have two groups of tripod-legs. The load sensor and angle sensor give us representative information about the group's state:

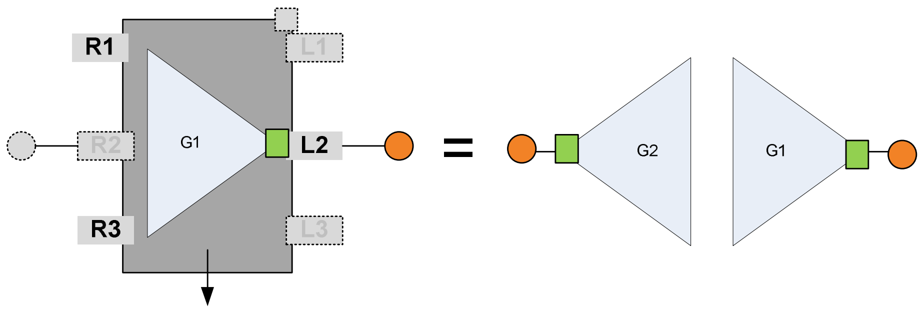


Figure 16: Tripod leg groups

From the diagram above we see that using the angle sensor and load sensor on L2, we can determine if the legs or group G1 are retracted/protracted and if they are up or down – of course, it is truly only leg L2's position.

3.2 Communication

The control program will run on a PC host and transmit commands and query sensor values from the hexapod. We are saving on complexity and cost of on-board processing hardware (which would have been something like a "small board computer"). To make the robot autonomous (read un-tethered), the commands and query happen over a wireless serial communication. The hexapod contains a controller board which receives commands over the serial port and commands the motors connected to it. This board also takes in four analog signals as input. The servo motors and sensors are connected to this board. The wireless communication is established by ZigBee technology.

3.3 What are we missing…?

Unlike a real hexapod, we do not have sensors on each leg – we assume the sensory inputs coming from the two representative legs (one for each tripod group) as sufficient for the sake of simplicity. Having sensory inputs on all legs will result in a “correct” hexapod which is capable of not just Tripod walking on flat surface, but more expressive like being able to negotiate obstacles, uneven terrains etc. As stated at the beginning of this document, it is not the intention to create a perfect robot, but a “just enough” configuration that will help prove SCOOP’s advantage in robotics control programming. Having all sensors adds not only to the cost but also to maintenance requirement. The control program will (or should) conceptually remain similar when we hypothetically consider a full-fledge robot.

4 The Problem of Control

Take all the answers you understand, and just let it go.. Mysteries of the Nile

4.1 Introduction

The goal is to implement the Tripod gait locomotion in our hexapod robot and for this purpose we have already built up the hardware and required software infrastructure. Now, we need to create a control program that will use the software framework to drive the robot legs into Tripod gait. Before we define the problem in detail, we need to look into the definition of a full-fledged hexapod in order to see how the reduced form still reflects conceptually the problem of implementing any kind of gait.

4.2 Walking of a Hexapod

A hexapod found in nature is certainly not limited to walking on level ground without any obstacles. If nature did produce “level-surface-no-obstacle” species, there is no doubt that it also promptly made it extinct! The true hexapod has sensors on all feet and is hence capable of practically walking on different terrains.

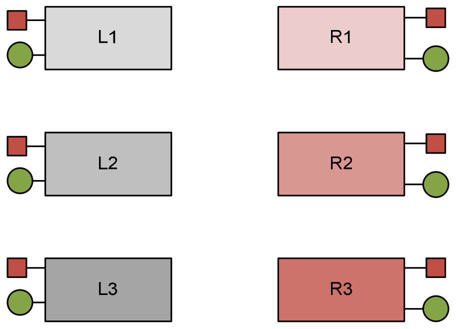
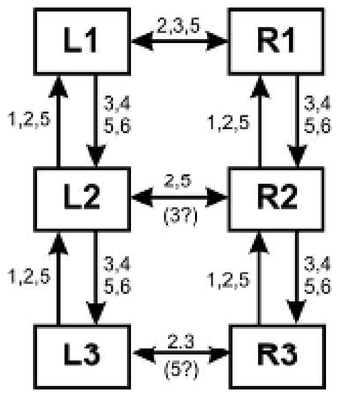
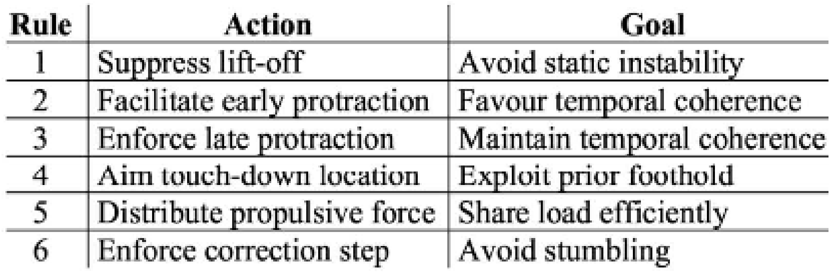


Figure 17: Schematic of hexapod legs with sensors

Each leg is capable of operating autonomously and makes the key decisions about its action after considering the state of its neighbors. For example, each leg must on its own decide how far to raise, swing and drop itself. If the leg’s targeted AEP happens to be over a ground level that is higher than the PEP, then it should drop until its load sensor indicates that the total load is being shared. Such control loop is local to the leg “node”. But, it must consult with neighboring legs to decide when to raise, drop or retract. Again as an example, the leg should not lift if one of the neighbor’s feet is not yet on ground. In this model of behavior each leg consists of a decentralized control unit which coordinates with other legs to achieve locomotion. The work of Durr (Durr, Schmitz, & Cruse, 2004) is perhaps the most authoritative and descriptive in this regard. In summary, a fully capable hexapod should implement a program that follows this model or something similar. In essence, such models specify leg objects executing protraction-retraction cycles subject to certain coordination contracts with other legs. The following diagram illustrates these coordination rules:





Rule	Action	Goal
1	Suppress lift-off	Avoid static instability
2	Facilitate early protraction	Favour temporal coherence
3	Enforce late protraction	Maintain temporal coherence
4	Aim touch-down location	Exploit prior foothold
5	Distribute propulsive force	Share load efficiently
6	Enforce correction step	Avoid stumbling

Figure 18: Biological model of leg coordination (from Dürr)

From the above coordination scheme, Rules 1, 2 and 3 are usually considered for implementation in robots. Rule 4 is enforced by static geometric consideration (i.e. setting the legs AEP close to the preceding legs PEP). Rule 5 is interesting, but does not contribute to locomotion. Rule 6 requires more elaborate sensors arrays to be placed on a leg (so as to detect if a neighboring leg treads on).

4.2.1 Requirement for Sensor – Actuator closed loop controls

If a servo motor has inbuilt closed loop control, why do we need sensors to confirm its action? I.e. if for example the coxa servo is commanded to reach AEP, and it will do this standalone, why do we need the position sensor? The existence of the position sensor is to provide a temporal sensing of *when* the action is completed. Similarly, the load sensors confirm that the leg has indeed been planted on or raised from the ground.

Hence, a leg's raise, swing, drop and retraction operations are closed loop control – for example, in case of drop operation the servo (Coxa) will be actuated until the force sensor reading confirms the action (i.e. leg has been raised or planted firmly on ground).

```
public override void Drop()
{
    //Get the current angle of Coxa
    ushort temp = m_Leg_aep_raised[1];

    //Until leg is planted, move Coxa down incrementally.
    while (IsRaised)
    {
        temp += 10;
        Driver.Instance.MoveJoint((ushort)(m_id + 1), temp, 100);
    }
}
```

4.2.2 Summary

The legs are autonomous units which have to execute the basic protraction-retraction cycles while taking note of its neighbor's state. The actions of protraction and retraction are themselves closed loop controls which need to constantly monitor the sensors and correct the actuators.

The problem can also be abstractly summarized as – **"How do we achieve decentralized and autonomous operations which are contractually subject to be cognizant of states of other such autonomous objects?"**

4.3 Problem Statement – Tripod Walking of the Hexapod Robot

In our hexapod robot, sensory information is limited to one pair of lateral legs (L2, R2) –i.e. in a Tripod leg group only one leg provides the representative sensory information. The information available on such a leg are: 1) Is Leg Retracted / Protracted and 2) Load on the leg. Since our robot is pre-destined to be operated on flat level surface, we can safely assume that the load measured on one leg will be more or less same as on the other two. The angle sensor on just one representative leg in a group of three is also a fair compromise since our hexapod robot supports "group commands " – i.e. we ask a set of leg servos to move together (this ensures that the position stated by the representative leg is also valid for other two legs in the group).

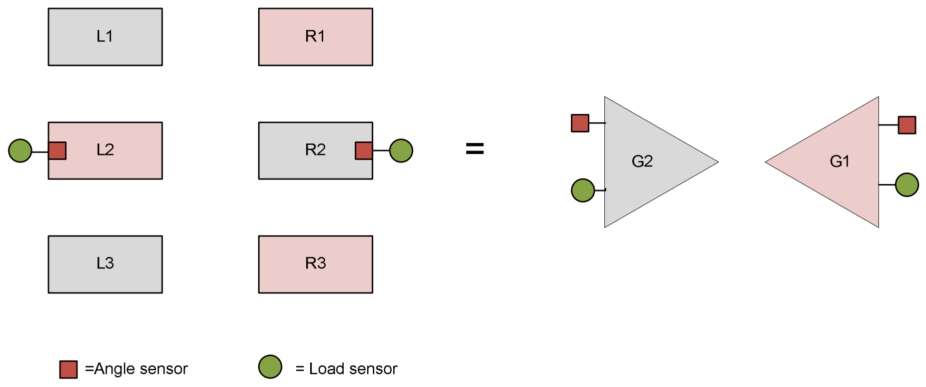


Figure 19: Equivalent representation of tripod pair

Fundamentally, in this problem we have to make the two Tripod Leg groups execute the protraction-retraction cycle while obeying the following rules:

1. Protraction of a leg group may begin only when the partner leg group is on ground. This is essential to maintain stability.
2. Protraction should not end (i.e. the leg should not drop on the ground) unless the partner group is finished retraction. This is to avoid dragging caused when the partner leg is still retracting.
3. Retraction should start only when the partner legs are raised (in the process of protraction). This is to avoid dragging.
4. Retraction should begin only at the AEP.
5. Protraction should only begin at PEP.

In order to simplify the coordination steps, the entire protraction can be broken down to three steps – 1. **lift** the leg, 2. **swing** to AEP but don't drop to ground and finally 3. **drop** to ground. Breaking down the walking behavior of each leg gives us the opportunity to implement inter-leg coordination rules (listed above).

Reflecting back upon the ideal hexapod robot which has sensory information on all legs (and hence is capable of all gaits) we see that the reduced form in tripod walking still needs to conceptually implement the autonomous operation of the leg groups while establishing coordination, albeit in a reduced quantity.

5 Treading the known path

Eleven persons take eleven paths – an Indian proverb.

5.1 Simple Sequential Program

Years of experience (and conditioning) with sequential programming has given us the knack to simplify even the most “Embarassingly Parallel” problem to a set of sequential instructions. Such conversion of problems which are naturally parallel to sequences are done by often compromising on program efficiency. For example, it is quite easy, and even convenient, to structure the hexapod’s Tripod gait as a sequence. The following program is a result of such thinking:

```
public void SequentialMoveSync()
{
    TripodLeg lead = m_legGroupA;
    TripodLeg lag = m_legGroupB;

    while (true)
    {
        lead.Raise();

        lag.Retract();

        lead.Swing();

        lead.Drop();

        TripodLeg temp = lead;
        lead = lag;
        lag = temp;
    }
}
```

The code above provides a nearly respectful solution. Since the motion of the two groups of legs is in a way alternating, the sequence starts off by assuming a “Lead” and “Lag” group. The code above is self-explanatory (strangely though, we expect procedural code to be Spaghetti – the panacea being Object Orientation). Each of the action calls (raise, retract, swing and drop) is synchronous – i.e. they return only when the action completion has been confirmed by the sensor. Once the legs are raised (in preparation for swing), the partner is triggered to retract following which the leg itself proceeds with the swing. Now, the method call to retract will not return until the closed loop control has ended with the angle sensor being signaled. Hence, swing of the lead leg has to wait (unnecessarily).

5.1.1 Impression

The sequential program looks neat and compact and is to a large extent is intuitive. Functionally, it is incapable of delivering the concurrent action and results in inefficient walking. Also, when try to scale up the problem to a full blown hexapod with autonomous legs, the code will get complex. This limitation will come up when trying to solve any kind of non-trivial robotics application that needs concurrent processing.

5.2 Multi-threaded Variant

Since each of the action (lift, swing, drop and retract) is synchronous in nature, the actions of a leg's swing during the partner's retract cannot be executed concurrently in a single-threaded program. We can rectify this situation easily using two separate threads of execution for each of the two leg groups. The following pieces of code snippets show the gist of the approach:

```
public void Walk()
{
    m_threadA = new Thread(new ParameterizedThreadStart(ThreadProcWalk));
    m_threadA.Name = "GroupA Thread";
    m_threadB = new Thread(new ParameterizedThreadStart(ThreadProcWalk));
    m_threadB.Name = "GroupB Thread";
    m_threadA.Start(m_legGroupA);
    m_threadB.Start(m_legGroupB);
}

private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState !=ThreadState.AbortRequested)
    {
        Console.WriteLine(leg + " Protraction - waiting for protraction lock");
        lock (m_protractionLock)
        {
            Console.WriteLine(leg + " Protraction - wait for partner drop");
            leg.Partner.DroppedEvent.WaitOne();

            Console.WriteLine(leg + " Protraction - calling Raise");
            leg.Raise();
        }

        Console.WriteLine(leg + " Protraction - calling Swing");
        leg.Swing();

        Console.WriteLine(leg + " Protraction - waitng for partner retraction");
        leg.Partner.RetractedEvent.WaitOne();

        Console.WriteLine(leg + " Protraction - calling Drop");
        leg.Drop();

        Console.WriteLine(leg + " Protraction - waitng for partner raise");
        leg.Partner.RaisedEvent.WaitOne();

        Console.WriteLine(leg + " Protraction - calling Retract");
        leg.Retract();
    }
}
```

Here the two threads execute a cycle of protraction and retraction and are synchronized with each other using Semaphores (in the code example, we use the AutoResetEvent – a Microsoft .Net specific synchronization primitive (Microsoft) with additional feature of automatically releasing the semaphore when the first thread goes through.). Additionally, a shared "lock object" is used by the threads to contest for the protraction privilege.

Apart from above two thread instances driving the leg motions, there is a "sensory polling" thread which periodically queries the sensor states and signals the semaphores. This thread raises events and the handler of the event signals the semaphores:

```
protected virtual void sensoryNode_Sensation(object sender, SensoryEventArgs e)
{
```

```
            Load = e.Values[(1 - (int)m_group) * 2];
            IsRaised = Load > m_legNoLoad;
            IsRetracted = e.Values[(1 - (int)m_group) * 2 + 1] == 0x30;

            if (IsRaised)
                RaisedEvent.Set();
            else
                DroppedEvent.Set();

            if (IsRetracted)
                RetractedEvent.Set();
        }
```

5.2.1 Impression

Multi-threaded programs are a time-tested way to handle concurrent and coordinated execution of programs and it proves its mettle here by solving the problem. Having said that, the reader should take a step back and ruminate over the larger problem – which is to model a robotic control behavior while keeping the object oriented model in focus. It is widely accepted that multi-threaded execution of routines is in a way orthogonal to the object-way of thinking. That is, we now "thread" through routines instead of objects and set up signaling between threads instead of objects. There are already works such as (Meyer, 1993) and (Papathomas, 1995) which elegantly argue for Object Oriented Concurrency and hence we skip the repetition here. The problem is highlighted in this simple hexapod control problem. For robotics control involving non-trivial concurrency and coordination, if we wish to evolve solutions based on object models (or "behavioral models" as outlined by Durr et al (Durr, Schmitz, & Cruse, 2004)), then we need to seek an answer using some form of Object Oriented Concurrency.

5.2.1.1 Handling a change in model

At this juncture it is useful to highlight an example to reinforce the above argument. We consider the scenario where the rule of coordination has to be modified.

It was discovered late during the implementation that one of the biological model proposes that protraction of a leg is triggered not by the dropping of partner leg, but by sensing that the leg's load is reduced (because the load is now shared by the partner legs. i.e. due to the effect of partner leg's placement on the ground).

To modify the multi-threaded program so that the alternative biological behavior is reflected, we need to setup new semaphore signaling:

```
Console.WriteLine(leg + " Protraction - waiting for protraction lock");
lock (m_protractionLock)
{
    Console.WriteLine(leg + " Protraction - wait for load reduction");
         leg.ReducedLoadEvent.WaitOne();

    Console.WriteLine(leg + " Protraction - calling Raise");
    leg.Raise();
}
```

We shall later reflect back on this example when exploring Object Oriented Concurrency to see how its proposed methodology can handle such changes.

5.3 Event based programming

Events are special mechanism provided by the programming framework wherein a supplier of state-full information can register one or more callback delegates which will be invoked when certain states are attained. The callback delegates are reference to routines within the context of the client.

```
m_legGroupA.Raised += new EventHandler<EventArgs>(m_legGroupA_Raised);
m_legGroupA.Retracted += new EventHandler<EventArgs>(m_legGroupA_Retracted);
m_legGroupA.Dropped += new EventHandler<EventArgs>(m_legGroupA_Dropped);


m_legGroupB.Raised += new EventHandler<EventArgs>(m_legGroupB_Raised);
m_legGroupB.Retracted += new EventHandler<EventArgs>(m_legGroupB_Retracted);
m_legGroupB.Dropped += new EventHandler<EventArgs>(m_legGroupB_Dropped);
```

The orchestration code is written in the event handlers:

```
private void m_legGroupA_Raised(object sender, EventArgs e)
{
    //Console.WriteLine("LegGroup A Raised");
    m_legGroupA.Swing();
    m_legGroupB.Retract();
}
```

The events are “wired-up” so that callback handlers invoke further action which would result in other events being raised, and so on until the cycle is completed. The following diagram illustrates a part of the hexapod’s event chain (the whole diagram and explanation have been avoided for the sake of brevity).

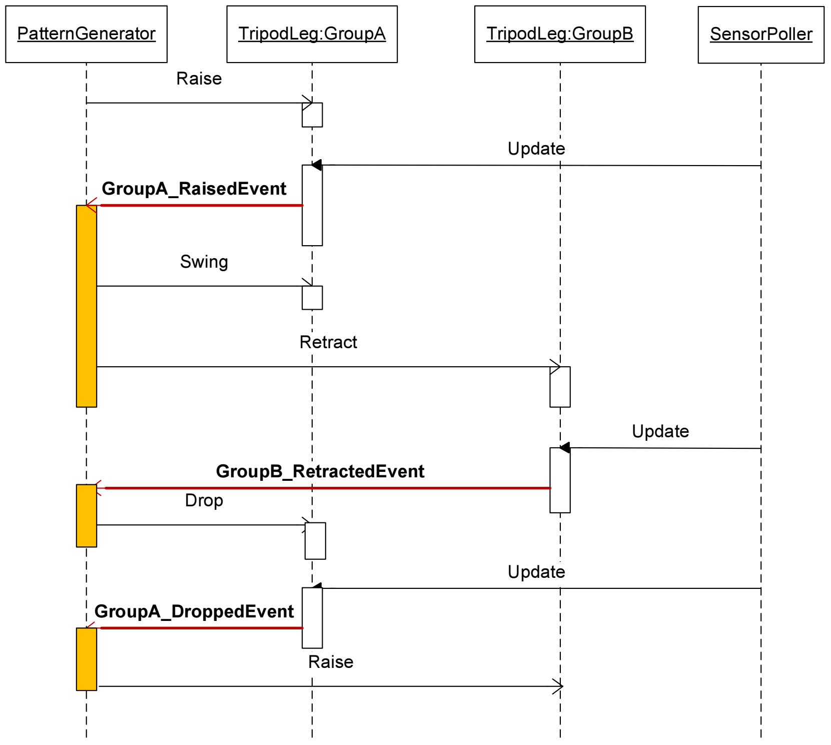


Figure 20:Sequence diagram illustrating events

5.3.1 Impression

Event based programming is very helpful as long as we don't need to establish complex orchestration – upon which the program (the "wiring-up") gets complicated and difficult to follow. The example illustrated here is already quite simple (we are dealing with two tripod groups, instead of six independent legs), and yet we see the program developing in a convoluted fashion. Depending upon the programming language, we have to be also careful about the thread context in which the events are raised – in .NET for example, the event delegate is invoked on the callers thread, which means the callback routine cannot be blocking or even slow in execution. Such considerations result in building up of asynchronous methods (which utilizes some form of background thread anyway). Note that in the above example, methods like drop and retract are asynchronous.

Hence, it is doubtful that event based programming can provide a satisfactory answer to the concurrency and coordination problems in robotics. The fact that it does provide answers for truly event like scenarios is not being doubted (for example, the SensorPoller raising Update events on new sensor states is elegant indeed), but using event wiring-up to establish orchestration can lead to highly un-maintainable code.

5.4 Subsumption Architecture

In one of the seminal works on robotic control programming, Brooks (Brooks, A robust layered control system for mobile robots, 1985) proposes an architecture for behavior based control at the lowest level of abstraction. The proposed model, called the Subsumption Architecture stems heavily from studies in Artificial Intelligence (Brooks, Intelligence without reason). Implementing a true Subsumption model in robotics control is not a trivial task and requires careful planning. Some flavors of Subsumption architecture have evolved that simplifies the behavior modeling to a higher abstraction. The Java™ library for Subsumption provided in the open source platform of LeJos is one such example (Lejos). For the sake of simplicity, we take this reduced version of the architecture to test its usefulness.

5.4.1 Prototype Implementation

The core of Subsumption model is to define behaviors and their respective sensory stimulus. Each such behavior will execute as long as its sensory condition allows it to. Hence there is no central controller or orchestrator. The behavior is triggered or inhibited by what happens in its environment. Hence the behavior, programmatically described, would be:

```
public interface IBehaviour
{
    bool MayExecute { get; }
    void Execute();
    void Suppress();
}
```

The original idea behind Subsumption model is that the behaviors are loosely coupled and depending upon the stimulus from the environment, they execute "themselves" (i.e. they are autonomous). On a computational device like the PC behaviors would need the processor to execute their action. Thus we need an arbitrator which facilitates this process of providing a behavior the processor space when it is needed. Note that the arbitrator is not an orchestrator or coordinator! It simply finds out if a behavior needs to execute itself, and if so allocates it the processor.

```
public class Arbitrator
{
    private List<IBehaviour> m_behaviours;
    private BackgroundWorker m_backgroundThread = new BackgroundWorker();

    public Arbitrator(List<IBehaviour> behaviours)
    {
        m_behaviours = behaviours;
    }

    public void Start()
    {
        m_backgroundThread.WorkerSupportsCancellation = true;
        m_backgroundThread.DoWork += new DoWorkEventHandler(m_backgroundThread_DoWork);
        m_backgroundThread.RunWorkerAsync(m_behaviours);
    }

    private void m_backgroundThread_DoWork(object sender, DoWorkEventArgs e)
    {
        List<IBehaviour> behaviours = e.Argument as List<IBehaviour>;
        while (!m_backgroundThread.CancellationPending)
        {
            foreach (IBehaviour behaviour in behaviours)
            {
                if (behaviour.MayExecute)
```

```
                    behaviour.Execute();
                else
                    behaviour.Suppress();
            }
        }
    }
}
```

With this infrastructure in place we are now ready to create behaviors and let them loose. For example, we create a behavior for "raising a leg" which based on few stimuli:

```
public class Raise : IBehaviour
{
    private TripodLeg m_leg;

    public Raise(TripodLeg leg)
    { m_leg = leg; }

    #region IBehaviour Members

    public bool MayExecute
    {
        get {return m_leg.IsRetracted && !m_leg.Partner.IsRaised; }
    }

    public void Execute()
    {
        m_leg.Raise();
    }

    public void Suppress()
    {
        m_leg.Retract();
    }

    #endregion
}
```

Once we have defined such behavior, they can be handed over to the arbitrator. Note that the sequence in which the behaviors are stacked into the arbitrator is not relevant.

```
    public void SubsumptionWalk()
    {
        IBehaviour a_raise = new Raise(m_legGroupA);
        IBehaviour a_swing = new Swing(m_legGroupA);
        IBehaviour a_drop = new Drop(m_legGroupA);
        IBehaviour a_retract = new Retract(m_legGroupA);

        IBehaviour b_raise = new Raise(m_legGroupB);
        IBehaviour b_swing = new Swing(m_legGroupB);
        IBehaviour b_drop = new Drop(m_legGroupB);
        IBehaviour b_retract = new Retract(m_legGroupB);

        List<IBehaviour> walking = new List<IBehaviour>();
        walking.Add(a_raise); walking.Add(b_retract); walking.Add(a_swing); walking.Add(a_drop);
        walking.Add(b_raise); walking.Add(a_retract); walking.Add(b_swing); walking.Add(b_drop);

        Arbitrator arbitrator = new Arbitrator(walking);

        arbitrator.Start();
    }
```

5.4.2 Impression

Subsumption architecture is a very clever idea to model behaviors in artificial intelligence. It results in extremely loosely-coupled autonomous systems. In true sense, the aspect of coordination is reduced down to the pre-condition for behavior to manifest. The concurrency aspect is not answered by this model (or, it is not even the target to try and provide an alternative paradigm for concurrency here). It is difficult to realize an important point here, but will become clearer after we have looked into SCOOP – architectural patterns like the Subsumption can benefit greatly from object oriented concurrency.

5.5 Microsoft Robotics Framework

The Microsoft Robotics Framework harnesses two concepts – Concurrency and Coordination Runtime (CCR) and Decentralized Software Services (DSS) to provide a flexible and powerful robotics programming platform (Microsoft). It further augments these infrastructures with an easy to use Visual Programming Language and a Simulation environment.

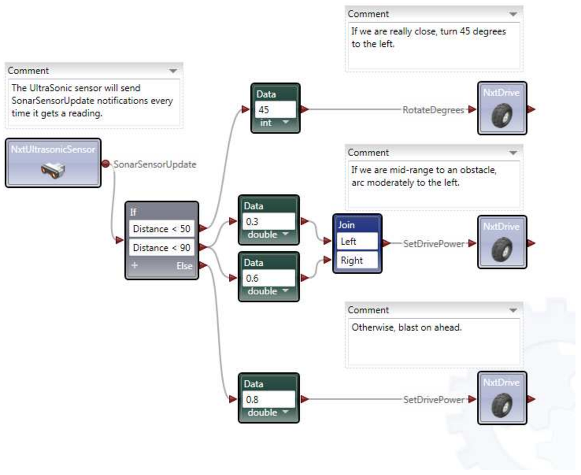

Comment
The UltraSonic sensor will send SonarSensorUpdate notifications every time it gets a reading.
NxtUltrasonicSensor
SonarSensorUpdate
If
Distance < 50
Distance < 90
Else
Data
45
int
Data
0.3
double
Data
0.6
double
Data
0.8
double
Join
Left
Right
RotateDegrees
SetDrivePower
SetDrivePower
NxtDrive
NxtDrive
NxtDrive
Comment
If we are really close, turn 45 degrees to the left.
Comment
If we are mid-range to an obstacle, arc moderately to the left.
Comment
Otherwise, blast on ahead.

Figure 21: Example of networked services

In the above diagram (as viewed from the Visual Programming Language), we see services (represented as blocks) are linked to each other via messages and events. Each of service operates concurrently with others and is hosted on pre-determined processors or host machines.

Since the core of the platform is the DSS infrastructure, we take a short look at its mechanism:

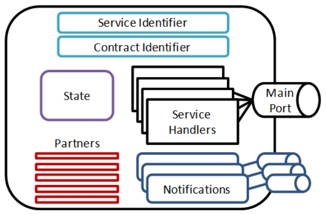


Figure 22: The model of a DSS node (from Microsoft)

Each "DSS Node" (i.e. an instance of service) has port into which messages can be sent. The messages are queued up and sent to Service Handlers according to their attributes. The Service Handler looks up to see if it needs partner services (in the same node or another) to be running and if not the partner service is triggered to start. Each node can publish notifications that can be subscribed from other services.

5.5.1 Prototype Implementation

The solution is modeled using Sensor, Actuator (leg servo) and Action condition nodes. The Sensor node polls the load and position sensor and raises notifications on conditions like "leg raised", "leg dropped", "leg protracted" and "leg retracted". The Sensor node can also be queried for current value or can be subscribed for by the client to receive values based on limits (example "on_load_change"). The Action nodes evaluate the sensor information and decide on actuation.

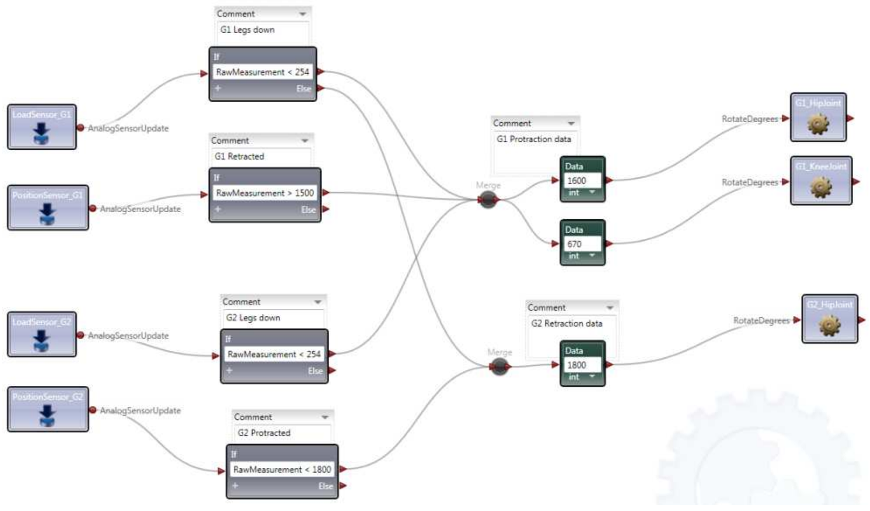


Figure 23: Part of the hexapod walking implemented as services

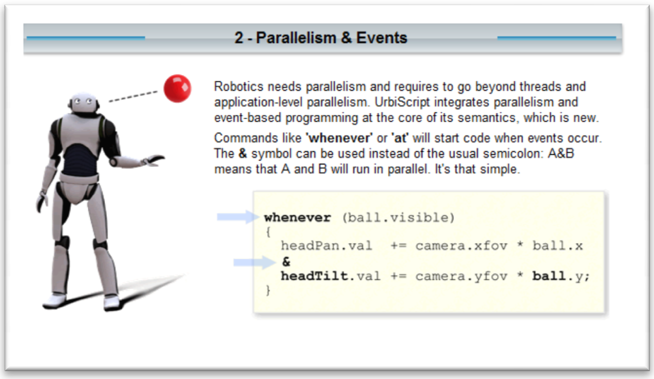
The above figure shows the implementation of protraction behavior of one leg group (G1) in the Visual Programming Language©. Also, to illustrate the coordination, the retraction behavior of corresponding leg group is shown. The three pre-conditions for protraction (legs down, in PEP and partner legs down) are expressed in the "If" evaluators. The Merge nodes here act as simple Boolean AND condition. In addition to the protraction behavior of a leg group, the opposite leg group's retraction is also included in the diagram. Note that since all the nodes process information concurrently, the condition evaluators for protraction and retraction happen in parallel.

5.5.2 Impression

Microsoft's Robotics Framework is based on two conjoining concepts - concurrency and coordination together with distributed services. This merger is not necessarily robotic programming specific but serves many other applications too. Implementation-wise, the framework requires detailed knowledge of service creation and hosting. Services, don't necessarily map to domain objects found in natural language description. For example, a hexapod leg is not a meaningful service whereas sensor, actuator and coordinator are. We can conclude that this framework is certainly not object-oriented but service-oriented. How much this helps in development of robotic control program is doubtful.

5.6 URBI – Universal Robotics platform

URBI from Gostai (Gostai) is a result of research on robotics programming conducted across more than 20 universities and seeks to address the fundamental issues (also proposes radically new paradigms). We are interested in the concurrency aspect, and the following example highlighted in the URBI website interests us immensely:



2 - Parallelism & Events

Robotics needs parallelism and requires to go beyond threads and application-level parallelism. UrbiScript integrates parallelism and event-based programming at the core of its semantics, which is new.

Commands like '**whenever**' or '**at**' will start code when events occur. The **&** symbol can be used instead of the usual semicolon: A&B means that A and B will run in parallel. It's that simple.

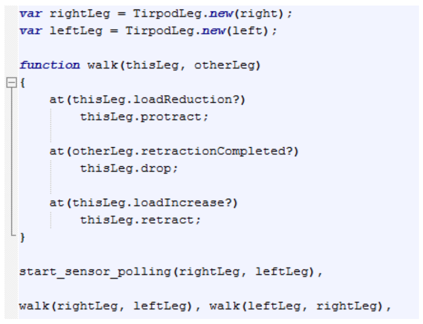
```
whenever (ball.visible)
{
  headPan.val  += camera.xfov * ball.x
  &
  headTilt.val += camera.yfov * ball.y;
}
```

Figure 24:Description of URBI (from Gostai)

Readers interested in application of concurrency are strongly encouraged to read the SDK document of URBI (Gostai, 2009). There are several ingenious constructs for parallelism and event based programming that may be of academic interest too.

5.6.1 Prototype Implementation

Though the program in URBI for the hexapod was not completed to full maturity, we have managed to construct the core business logic of walking and test its working:



```
var rightLeg = TirpodLeg.new(right);
var leftLeg = TirpodLeg.new(left);

function walk(thisLeg, otherLeg)
{
    at(thisLeg.loadReduction?)
        thisLeg.protract;

    at(otherLeg.retractionCompleted?)
        thisLeg.drop;

    at(thisLeg.loadIncrease?)
        thisLeg.retract;
}

start_sensor_polling(rightLeg, leftLeg),

walk(rightLeg, leftLeg), walk(leftLeg, rightLeg),
```

Surprisingly, the implementation is simple, intuitive and very extensible. The most powerful constructs used here are “at” which spins off a concurrent event monitoring. The parallel routines are started by simply specifying them one after the other delimited by comma. In the above snippet, the routine start_sensor_polling runs in parallel with two instances of walk. Each instance of walk spins off three concurrent event monitoring.

5.6.2 Impression

URBI is a serious technology for robotics programming and has very powerful features. But as a language, it is placed in another league (un-typed, prototype based...). If we discount the war of languages, URBI has done well to address the issues faced by robotics programmers. Taking a lateral view of URBI, we also realize that many of its features are a result of failure in “marriage” of object-oriented programming with Concurrency (see (Meyer, 1993)).

6 SCOOP for Mortals

Judging by the looks of the two parties, the marriage between concurrent computation and object-oriented programming - a union much desired by practitioners in such fields as telecommunications, high performance computing, banking and operating systems appears easy enough to arrange. This appearance, however, is deceptive: the problem is a hard one. – Bertrand Meyer

6.1 Introduction

Most programmers have a good knowledge of concurrent programming using multiple threads and use synchronization primitives like mutexes and semaphores. In a way it has become natural to think of parallel tasks as executed by threads and coordination achieved by monitors, semaphores and mutexes. Having lived in this mindset for long, it is sometimes tough to think of problems in concurrency in a term other than multi-threading. Object Oriented Concurrency requires the thought process to be re-aligned back to Object Orientation. SCOOP, an Object Oriented Concurrency mechanism, presents an opportunity to explore this thought process. Currently available reading materials on SCOOP are elaborate to achieve the academic correctness quality, which could be deterring to the “eager-beaver” code enthusiast. We now present a “quick start” guide to SCOOP (Simple Concurrent Object Oriented Programming) which will enable you to realize projects after going through a short learning curve.

For the sake of illustrative explanation, let us the take the hypothetical example of a control system in an automobile. The central processor [C] coordinates the operation of various autonomous elements like the engine [E], the fuel system [F] and the dashboard display [D]. We will refer to these entities as [C], [E], [F] and [D] to avoid tedious repetition of names, and also implicitly encourage generalization.

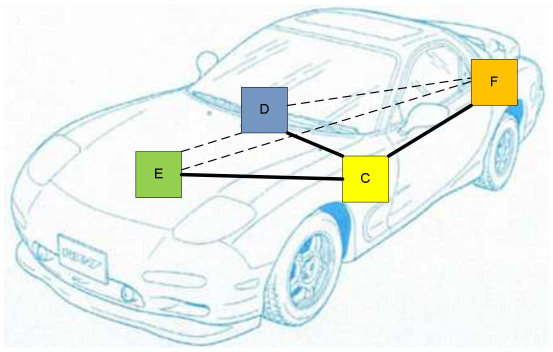


Figure 25: Example of separate entities in a car control system

The processor of the [D] can query the status of [E], [C] and [F] and report it on its screen. Also each of the components can report their status to display [D] (like [E] reporting “engine oil level low”). The controllers themselves have specific tasks to perform – the engine controller [E] for example has to measure sensor values and control the engine’s operation. Here we encounter a scenario of several autonomous entities (having state and behavior) that execute actions concurrently and need to coordinate with each other. The advantage of Object Oriented Concurrency is that the above object diagram is also sufficient to explain concurrency and coordination (instead of resorting to additional “threading scheme” diagram).

6.2 Objects are hosted by Processors

In multi-threading paradigm, routines that are required to run concurrently are executed by different threads. Hence, when the engine’s main control loop requires to be run concurrently with, say, the fuel system’s control routine, we would launch two threads to take care of the routines. The threads are agnostic about the engine or fuel system objects and its semantics. That is, the thread's main purpose in life is to execute the routine and not care about the object's state. For example, if it is forbidden to change the engine's ignition parameters while its main control is being executed, we need to build in additional infrastructure in our multi-threaded program to satisfy this domain rule.

With SCOOP, neither routines nor threads are our focus. In the SCOOP way of thinking every object is hosted by a "Processor" which is responsible for executing all its behavior (routines). The term Processor is a bit abstract and should not be necessarily correlated with physical processors (or processor core). It could practically turn out to be a physical processor (or core) or it could be a thread responsible for execution of a particular objects routines.

The object's host processor is determined during its declaration. A simple keyword "**separate"** designates the object instance to be hosted on a Processor that is separate from the current object's host.

Let's look at this example:

```
class
        CAR_CONTROLLER
creation
        make

feature {NONE}
        E : separate ENGINE_CONTROLLER

feature {ACCESS}
        make
        do
                create E.make
        end
```

Of course, the `E` need not be a class member, but could be a routine's local member that lasts only for the scope of the routine.

In the code example above the CAR_CONTROLLER creates an instance of the ENGINE_CONTROLER, but by additionally giving the type info as **separate**. This guarantees that the [E]'s processor is not the same as [C]. Similarly, [C] can instantiate separate instances of [E] and [F] which are hosted on separate processors. The processors of [E] and [F] are not only separate with respect to [C], but also with respect to each other. For the sake of example, let us assume that [C] instantiates [D] without the keyword separate (and hence [D] and [C] are served by the same processor).

```
class
        CAR_CONTROLLER
creation
        make

feature {NONE}
        E : separate ENGINE_CONTROLLER
        F : separate FUEL_CONTROLLER
        D : DISPLAY_CONTROLLER
```

This results in a state where interaction between object instances will have to cross a "processor boundary" depending upon their affinity. In the following figure, except for interaction between [C] and [D], all others have to cross this imaginary boundary (we will informally term this as *separate* calls – indicated by dotted line in the figure)

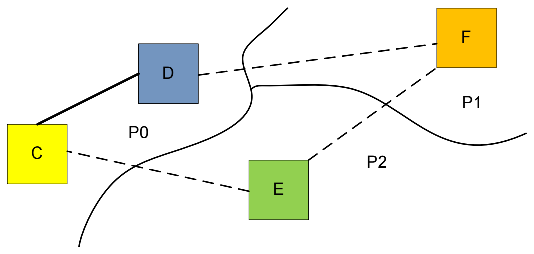


Figure 26: Separateness of objects

Use of **separate** keyword causes the runtime to allocate an object instance to a distinct underlying thread. However, there are no explicit guarantees or knowledge of this in compile time (i.e. we do not know the underlying thread behind the object). In order to enforce static knowledge of hosting, we can use Processor Tags.

```
px, py: PROCESSOR
E : separate<px> ENGINE_CONTROLLER

F : separate<px> FUEL_CONTROLLER

C : separate<py> CAR_CONTROLLER
```

From the above code snippet, one can conclude that the [E] and [F] are hosted by the same processor while [C] is in handled by another processor. Usefully, the object's handler can also be referred by `.handler` accessor and hence can be used as processor tag :

```
D : separate<C.handler> DISPLAY_CONTROLLER
```

The above statement makes sure that [D] is on the same processor as [C].

Q. If we don't explicitly specify the processor with the **separate<p>** *instead of just* **separate** *do object implicitly get handled by different processors?* A. Yes, the objects are logically on different processors.
Q. Can we pass around the reference obtained from .handler as regular object? Yes you can.
Q. If Processor is eventually (always) a thread, why not simply call it a thread? The processor's execution is implemented by a thread, but the processor itself is an abstract entity which denotes the object's host. It takes care of executing routines in the object's context, ensuring its protected access, scheduling etc. Hence a thread and the processors are not synonymous.
Q. Is there a way to find out to which processor an object belongs or what objects are handled by a processor? Yes, using the .handler accessor.
Q. On a multi-core machine can I specify the physical processor? Not in the present implementation, but this is one of the foreseen requirements.

6.3 Separate Objects are exclusive resources

Before we can "do" anything on a separate object, we need to obtain exclusive access to it. That is, at any given time a separate object can be operated on (queried or commanded) only exclusively by another object. A client who wishes to access a separate object must first obtain a "lock" on its processor. Take note that the lock is obtained on the processor and not on the target object itself. Which means that if [F] has obtained a lock on processor p0 (refer to figure above), then [E] has to wait if it wants to obtain access to [D]. Syntactically, to obtain the lock on processor of a separate object we need to pass the instance of the object as an argument of a routine call. The following code snipped clarifies this:

```
class CAR_CONTROLLER
…
        E: separate<px> ENGINE_SYSTEM
        F: separate<px> FUEL_SYSTEM
        D: DISPLAY

initialize is
        do
                E.initialize -- This is NOT ok! E is not an argument of the enclosing routine (initialize)
                initialize_parts(E, F, D)
                D.show(“Ready”) –- This is ok. D is not separate.
        end

initialize_parts(e:separate ENGINE_SYSTEM; f:separate FUEL_SYSTEM; d:DISPLAY) is
        do
                d.show(“Wait. Initializing..”)
                e.initialize –- Now its fine as e is an argument.
                f.initialize
        end
```

In the above snippet, E and `F` are separate objects (vis-à-vis [C]). In order to call a function (or query), we need to first obtain “locks” or access privilege on these object’s processor. We do this by passing them as actual arguments to the call `initialize_parts` which takes two separate arguments and one non-separate argument. The routine call to `initialize_parts` will have to first obtain locks on all separate arguments before it can proceed. It does not bother about obtaining lock on the non-separate argument (the `d`). The routine will wait indefinitely until the locks are obtained. An object’s lock can be obtained if it is not already owned by someone else and if its processor is not busy. We come to more on this later in the next section.

This raises some questions… let’s try to clear up some air around the strange looking concepts and constructs. We will now try to answer some questions.

Q. First of all why always lock when you might intend to just make some queries without any side-effects?

A. The rule that all separate entities should be locked in the context of a routine call seems a bit harsh when just need to execute queries on the separate object. SCOOP provides a side-door escape for separate entities which we need to query but not involve locking – this is done by declaring the separate variable additionally as detached type:

```
        initialize_parts(e:separate ENGINE_SYSTEM; f:?separate FUEL_SYSTEM; disp:DISPLAY) is
                do
                        disp.show(“Wait..”)
                        if f.is_ok then
                                e.initialize
                        end
                end
```

Within the body of `initialize_parts` we cannot execute any function calls on the entity `f` though queries are allowed. The above rules also apply to separate entities that are local to a routine.

Q. Why use routine calls to implicitly cause lock procurement? (Why not have syntax supports like `lock(x)`, `x.foo, release(x)`..?)

A. Looking at it the other way round, every time we lock something, we want to execute a set of commands or queries on it exclusively. Capsulating this part (where we want do something exclusively) in a routine prompts you to extract out your exclusive access behavior to a distinct portion in the code.

Q. What happens when some of the separate objects in the routine argument cannot be locked while some others *can be locked?*

A. If locks cannot be obtained on all the arguments, we let go all what we have and try again (until all locks are available). If we have routine call like `r(a1, a2, a3:separate A)` where `a2` is busy but not `a2` and `a3`, then the routine first tries (and succeeds) to get lock on `a1`. Then, when it fails to get lock on `a2`, releases the lock on `a1`.

Q. Is it not tiring to write routines which involve calls on just one separate object?

A. Yes it is. That is why the SCOOP library supports a wrapper method *asynch* for invoking commands.

Q. What is "lock passing"? Can the lock obtained during the course of routine call be passed to another separate object (i.e. another processor)?

Lock passing occurs (or needs to occur) when in the scope of a routine where locks on separate objects have been obtained, we make further call on one of the separate object and in that call we pass some of the "locked" objects as arguments. The following example illustrates this situation:

```
class CAR_CONTROLLER
…
change_engine_mode(e:separate ENGINE_CONTROLLER; f:separate FUEL_CONTROLLER)
do
   e.set_mode(MODE_SPORT, f)
   pump_state := f.pumpstate
   D.showtext("Engine mode is:" + e.get_mode)
end
-----------------------------------------------------------------------------
class ENGINE_CONTROLLER
…
set_mode(mode:INTEGER; f: separate FUEL_CONTROLLER)
do
  If mode = MODE_SPORT then
     f.set_pumpspeed(HIGH)
  end
end
```

Going by our explanation on separate calls, `e.set_mode` should be asynchronous (as it is invoked on the separate object `e`). Let us suppose that `e.set_mode` is invoked asynchronously but `set_mode` waits to acquire lock on f. In this case we would end up in a deadlock because down the line in `change_engine_mode` we make a query `e.get_mode` which will wait for e to be free (but then e is waiting in the routine `set_mode` for `f` to be free, which will not occur until `change_engine_mode` completes). To solve this situation, we pass all locks obtained in routine `change_engine_mode`, including the implicit lock on `current` to `e` when we invoke any method on `e` that involves an attached separate argument. Additionally, such a method invoke is synchronous – i.e routine `change_engine_mode` halts until `e.set_mode` completes. This in short is lock passing.

6.4 Separate Calls

Once we have the separate object within the grasp of the enclosing routine, we are ready to work with it with guaranteed exclusivity. All function calls on separate entities are asynchronous by nature – i.e. they will return immediately while the routine called is guaranteed to be executed at some point in time. Coming back to the previous example:

```
        initialize_parts(e:separate ENGINE_SYSTEM; f:separate FUEL_SYSTEM; disp:DISPLAY) is
                do
                        disp.show("Wait..")
                        e.initialize
                        f.initialize
                end
```

The call disp.show will, as usual, be a synchronous call as it is executed on a non-separate object. The calls e.initialize and f.initialize will return immediately. The actual execution of initialize within instance of engine and fuel will commence at some point in time, which may be even after the initialize_parts is finished. What is certainly guaranteed is that the lock on engine and fuel will not be given to anyone else until the completion of the pending asynchronous calls. This is kind of obvious since the invocation of the asynchronous calls sets the corresponding processors as busy.

Locks can be passed temporarily to routine calls occurring within a routine which has managed to obtain the locks:

```
 initialize_parts(e:separate ENGINE_SYSTEM; f:separate FUEL_SYSTEM; disp:DISPLAY) is
        do
                disp.show(“Wait..”)
                if check_engine(e, disp) then
                        e.initialize
                        f.initialize
                end
        end


check_engine(e:separate ENGINE_SYSTEM; disp:DISPLAY) : BOOLEAN is
        do
                e.start_measurements
                disp.show(“Measuring engine parameters..”)
                result := e.measurements_ok
        end
```

The call check_engine passes the lock on engine that was acquired during initialize_parts and is returned back once check_engine completes.

Yet another aspect that is interesting is that in check_engine the call to e.start_measurements will be executed asynchronously and the routine will proceed until the query e.measurements_ok. The query will not go through until all previous asynchronous calls on engine have been completed. This can often be used as kind of join semantic.

6.5 Pre-conditions

Pre-condition and post-condition of a routine together with the class invariant provides the contract for execution (helping in ensuring correctness). In a separate object routine call scenario, the pre-condition plays a special role when the variable involved in the pre-condition evaluation is another separate object – instead of raising an exception on violation, the pre-condition statement waits until the query returns true. Until then, all locks acquired by the routine are released.

```
class ENGINE_SYSTEM
…
        fuel: separate<px> FUEL_SYSTEM

        initialize is
                do
                        initialize_internal(fuel)
                end

        initialize_internal(f:separate FUEL_SYSTEM) is
                require
                        fuel_sys_ok : f.is_ok
                        engine_not_disabled : not Current.is_disabled
```

```
do
        …
end
```

6.6 Type System

Since the separate keyword adds to the type information, we need an extended type system that provides rules for assignment and argument passing. A strong type system ensures that the compiler (and runtime) is aware of the rules while dealing with assignment and argument passing involving separate types. Why do we need this? Lets take a simple example:

```
class CAR_LIGHTING_CONTROLLER
        c : CAR_CONTROLLER

        test_headlights(central_controller : separate CAR_CONTROLLER)
        local
                v : INTEGER
        do
                ….
                v := central_controller.battery_voltage
                c := central_controller
        end
```

Whats wrong here? Someone calls `test_headlights` and passes as argument an instance of separate [C] (imagining that it will be required). Now in the method `test_headlights`, `central_controller` is being assigned to a local member of type `CAR_CONTROLLER`, but non-separate. This is obviously wrong! (If we allow this, then the processor boundaries all get mangled up!). So, if a type system that can dictate that non-separate variant of a type is actually specialization of its separate variant, then the check for type hierarchy during assignment will avoid the error (i.e. the compiler will disallow and instance of base type to be assigned to a variable of derived type).

This means in addition to the regular type info we need to now additionally qualify it with its separateness attribute or more precisely its processor. We have also seen that separate instances can further be distinguished if it nullable or not (attached or detached type). Detached separate instances allow only queries to be executed and hence need no locking. This now adds to the type checking requirement – if we received a detached separate instance in a routine call, we cannot assign it to a local member of type attached separate. Modfiying the previous example:

```
class CAR_LIGHTING_CONTROLLER
        c : separate CAR_CONTROLLER

        test_headlights(central_controller : ?separate CAR_CONTROLLER)
        local
                v : INTEGER
        do
                ….
                v := central_controller.battery_voltage
                c := central_controller
        end
```

In the above code snippet, the `test_headlights` received a detached instance of `car_controller` in the hope that it would be used just for querying `battery_voltage`. But then the assignment to `c` attempts to change the rule, which should be pointed out by the type checker.

From the above examples, we can conclude that we need a type system that evaluates three factors together – the processor, the attached/detached qualifier and the abstract type itself. This requirement on the type system is expressed as (α, β, Y) where α represents the processor, β the "attachment" and Y is the abstract type itself.

For practical purposes, we can begin with simple rules:

1. Non-separate type is a subtype of separate type.
2. Detached types (nullable) are subtypes of attached types.
3. As well known, in inheritance hierarchy derived types are subtype of the parent.

With these simple set of rules, we can already enforce the basic rule – an instance of base type cannot be assigned to a variable of subtype.

The type system is explained elegantly in the lectures notes of the Concurrent Programming -2 (ETHZ, 2008) and the reader is encouraged to study them. We reproduce here few slides to complete this discussion.

The three parts of the type system are summarized as below:

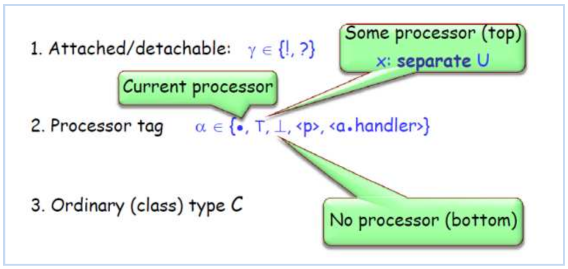

1. Attached/detachable: $\gamma \in \{!, ?\}$
Some processor (top)
x: separate U
Current processor
2. Processor tag $\alpha \in \{\bullet, \top, \bot, \langle p \rangle, \langle a.handler \rangle\}$
3. Ordinary (class) type C
No processor (bottom)

Figure 27: Components of the SCOOP type system

Some examples to illustrate the above mentioned notational system:

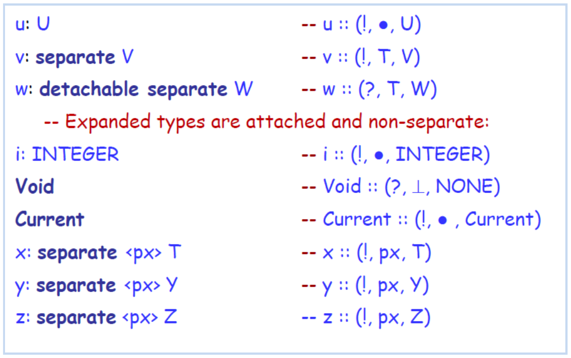
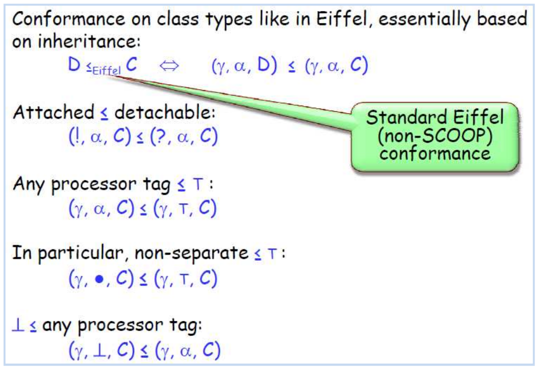

```
u: U                            -- u :: (!, ●, U)
v: separate V                   -- v :: (!, T, V)
w: detachable separate W        -- w :: (?, T, W)
   -- Expanded types are attached and non-separate:
i: INTEGER                      -- i :: (!, ●, INTEGER)
Void                            -- Void :: (?, ⊥, NONE)
Current                         -- Current :: (!, ● , Current)
x: separate <px> T              -- x :: (!, px, T)
y: separate <px> Y              -- y :: (!, px, Y)
z: separate <px> Z              -- z :: (!, px, Z)
```

Figure 28: Examples of the type system in use

And finally, we apply the basic conformance rule:



Conformance on class types like in Eiffel, essentially based on inheritance:

$D \leq_{Eiffel} C \Leftrightarrow (\gamma, \alpha, D) \leq (\gamma, \alpha, C)$

Standard Eiffel (non-SCOOP) conformance

Attached $\leq$ detachable:

$(!, \alpha, C) \leq (?, \alpha, C)$

Any processor tag $\leq \top$:

$(\gamma, \alpha, C) \leq (\gamma, \top, C)$

In particular, non-separate $\leq \top$:

$(\gamma, \bullet, C) \leq (\gamma, \top, C)$

$\perp \leq$ any processor tag:

$(\gamma, \perp, C) \leq (\gamma, \alpha, C)$

Figure 29: Simplified type rules

With this, we complete our discussion of the type system and the short introduction to SCOOP

7 Finding answers in SCOOP

Get your facts first, and then you can distort them as much as you please. – Mark Twain

7.1 Introduction

SCOOP facilitates creation of an object model that is naturally interwoven with concurrency and coordination mechanism. In order to reap benefit of this, we will begin by defining the static object structure and then try to see if our requirement for concurrent control can be described within it (and not in addition to it). We will use results of various researches in biological modeling of hexapod walking in designing the program. Such an attempt will verify the possibility of "natural language design" when using SCOOP. Finally we reflect upon the solution and judge the merits.

7.2 Design

Recalling from chapter 2, the biological hexapod entity contains for each leg a central pattern generator (CPG) that produces cyclic signal for protraction and retraction. The sensory ganglions inhibit or excite actions generated by the CPG after considering the inputs from the leg's sensors. The CPG of each leg, which is a part of network of other leg CPGs, is additionally triggered or inhibited by the neighboring leg. Armed with the analysis of the problem (chapter 4), we can develop an object model that is almost parallel to the biological equivalent.

The object model of the hexapod (Tripod walker in specific) is presented here (showing object instances):

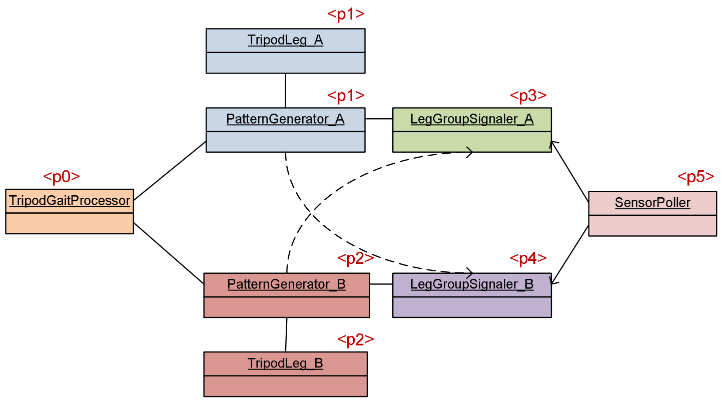


Figure 30: Object model of the hexapod

The **TripodGaitProcessor** has two **PatternGenerator**s for each of the two leg groups and triggers them to walk or stop. The **PatternGenerator** uses the actuation services of the **TripodLeg** (i.e. to cause leg movements) and sensory cognition provided by the **LegGroupSignaler** (which tells the pattern generator about the physical state of the leg). The **SensorPoller** is common services which polls the sensors on the hardware and updates the **LegGroupSignaler** with the current state.

Now, let us consider how concurrent operation of the leg groups and their inter-coordination can be realized. The central gait processor (here the **TripodGaitProcessor**) will trigger both pattern generators to begin walking. On receipt of this signal, the pattern generators leap into producing the protraction – retraction cycles. But before the action is executed the "contractual" restraints are checked and whenever they are satisfied, the action is processed further. It is important to note the term "whenever" in the previous statement – it means that the action that is withheld due to an unsatisfied contract will wait until the condition is fulfilled.

Since in SCOOP we have an easy mechanism to establish parallel executors (i.e. two separate objects on different processors), we can easily parallelize the objects shown in Figure 30. In the figure we have indicated the separateness by fill color (and the stereotype <*processor*>). Ok, that seemed easy enough to parallelize the hexapod's decentralized components. What about the coordination? Referring back to the problem statement for hexapod robot to recall the rules:

1. Protraction of a leg group may begin only when the partner leg group is on ground. This is essential to maintain stability.
2. Protraction should not end (i.e. the leg should not drop on the ground) unless the partner group is finished retraction. This is to avoid dragging caused when the partner leg is still retracting.
3. Retraction should start only when the partner legs are raised (in the process of protraction). This is to avoid dragging.
4. Retraction should begin only at the AEP.
5. Protraction should only begin at PEP.

It is not difficult to interpret the rules as pre-conditions for the action to take place. Traditionally, semantics of pre-condition of a routine dictate that when the pre-condition is not satisfied, the routine's execution will not occur and the caller is returned a contract exception. In case of a separate routine call, instead or returning an exception the routine is held in abeyance until the pre-condition is satisfied. We can use this wait semantics to the fullest advantage in concurrency situation. The two pattern generators indirectly coordinate by querying each other's signaler.

The easiest way to check the possibility of "SCOOPing" the above requirements is to jump right into code implementation.

```
walk
        -- The main cycle of locomotion
    do
        from
        until
            my_signaler.stop_requested
        loop
            begin_protraction (partner_signaler, my_signaler)
            ensure_protraction (my_signaler)
            complete_protraction (partner_signaler, my_signaler)
            execute_retraction (partner_signaler, my_signaler)
        end
    end
```

The above code snippet shows the core routine of the **PatternGenerator**, two parallel executions of which take place in the separate objects instances. In the main loop, first we wish to raise the legs in preparation for protraction. Before a leg group is raised, we need to satisfy the Rule 1 and 5 (*"Protraction of a leg group may begin only when the partner leg group is on ground..."* and *"Protraction should only begin at PEP"*) Let us try to model this as pre-condition statements for the `begin_protraction` routine:

```
begin_protraction (partner, me: separate LEG_GROUP_SIGNALER)
        --Raise legs after confirming that:
        -- 1. Legs are in PEP
        -- 2. Partner groups legs are on ground
        -- 3. Partner group's lift has not be initiated
    require
        my_legs_retracted: me.legs_retracted
        partner_down: partner.legs_down
        partner_not_protracting: not partner.protraction_pending
    do
        tripod.lift
        me.set_protraction_pending (True)
    end
```

The `begin_protraction` needs to access the two leg group signalers to find out about the current status – the locks on these two signalers are obtained by virtue of them being controlled arguments. The first two pre-conditions are reflective of the rules of coordination. The precondition `me.legs_retracted` queries the separate signaler if the leg group is already in the PEP, while the second precondition `partner.legs_down` queries the other group's signaler if the legs are placed on the ground.

The third precondition not `partner.protraction_pending` is interesting – we do not find this in the coordination rules, but we will require this to ensure both leg groups don't begin protraction simultaneously. Consider the situation where in the body of `begin_protraction`, the call `tripod.lift` is executed. Now we would ideally expect the legs to start rising so that when the body of `begin_protraction` ends and the opposite group if it should be in the moment of also calling `begin_protraction`, will not progress on precondition `partner.legs_down`. But practically, it might happen that the `tripod.lift` does not cause the legs to rise, in which case the opposite group is also capable of entering into the routine and calling `tripod.lift`. This would result in a disastrous consequence where both groups end up lifting their legs!

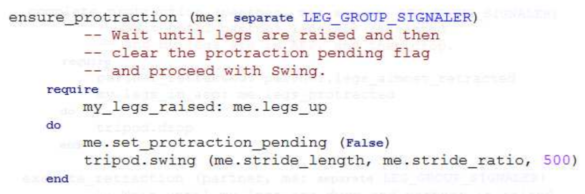
We can solve the above problem in an alternative manner:

```
…
tripod.lift
from until me.legs_raised
do
        scoop_sleep(100)
end
…
```

This will ensure that the routine `begin_protraction` does not exit until it has made sure that the legs have indeed risen. Explicit spin-wait statements, though pragmatic, are indicative of something inappropriate. Further investigation into neural networks involved in locomotion lands us into a concept called Proprioception which is the sensory knowledge of what the limb is doing at that moment. We can use the sensory node of the leg group's signaler to store this stateful information and add a simple Boolean flag call `is_protraction_pending` which signifies that a leg group is in the attempt of lifting.

We find that the mechanism of waiting pre-condition supports us in achieving the coordinating the concurrent activities.

Once the leg group's lift action has been invoked, the loop proceeds next to ensure the lift and progress with swing:



```
ensure_protraction (me: separate LEG_GROUP_SIGNALER)
        -- Wait until legs are raised and then
        -- clear the protraction pending flag
        -- and proceed with Swing.
    require
        my_legs_raised: me.legs_up
    do
        me.set_protraction_pending (False)
        tripod.swing (me.stride_length, me.stride_ratio, 500)
    end
```

Note that the proprioceptive state of protraction pending has been set to false here. The swing is set to occur at speed same or higher as the retraction speed. The next stage is to wait for the opposite group to finish retraction and then drop the leg to ground (and begin retraction).

```
complete_protraction (partner, me: separate LEG_GROUP_SIGNALER)
        -- Wait until partner legs are retracted
        -- and my legs are in AEP, and then drop.
    require
        partner_retracted: partner.legs_almost_retracted
        my_legs_in_aep: me.legs_protracted
    do
        tripod.drop
    end

execute_retraction (partner, me: separate LEG_GROUP_SIGNALER)
        -- Wait until my legs are dwon and partner is raised,
        -- and then start retraction.
    require
        my_legs_down: me.legs_down
        --my_legs_strained: me.legs_fully_loaded
        partner_up: partner.legs_up
    do
        tripod.retract (me.retraction_time)
    end
```

In the above code statements, the pre-condition `partner.legs_retracted` for routine `complete_protraction` serves to implement Rule 2. Similarly, the pre-conditions `me.legs_down` and `partner.legs_up` implement Rules 3 and 4.

The SensorPoller obtains lock on the signalers and updates them with the latest sensor values. Notice that none of the routines involved in protraction and retraction phases hold lock on the signalers until the physical action is complete – i.e. for example when leg retraction is executed, the call `tripod.retract` returns immediately after triggering the servo to move and does not wait for the leg to reach PEP. This waiting occurs in the pre-condition of the next method in the loop (`begin_protraction`). Since the time for which any client holds the signaler object is minimal, the sensor poller gets ample opportunity to update the signalers.

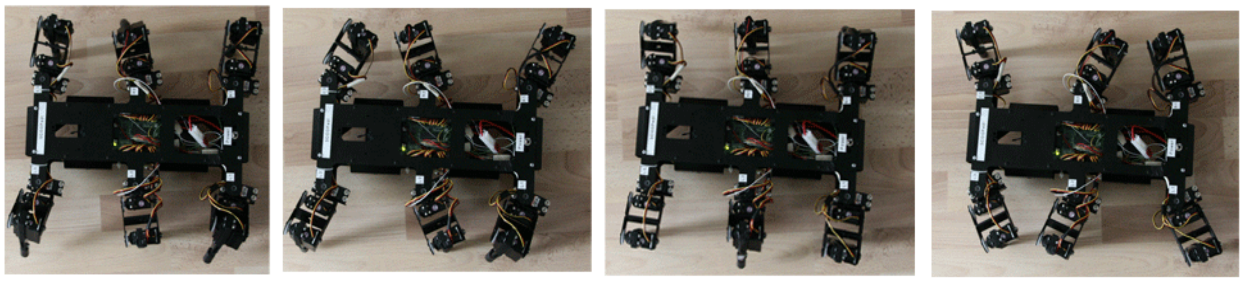
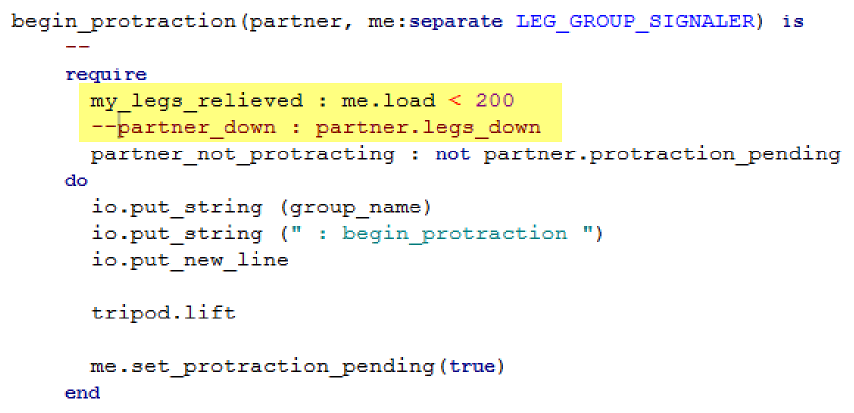


Figure 31: Hexapod robot walking

7.2.1.1 Handling changes to behavior rules

We now come back to an issue highlighted in the section on multi-threading (Handling a change in model). The issue was a change in the condition that causes a leg to rise – instead of using the signal that confirms grounding of the opposite leg group, we need to base it on the signal that the leg group's load is reduced (which is consequent to grounding of the opposing leg group).

This is achieved with a change in pre-condition:



```
begin_protraction(partner, me:separate LEG_GROUP_SIGNALER) is
        --
        require
          my_legs_relieved : me.load < 200
          --partner_down : partner.legs_down
          partner_not_protracting : not partner.protraction_pending
        do
          io.put_string (group_name)
          io.put_string (" : begin_protraction ")
          io.put_new_line

          tripod.lift

          me.set_protraction_pending(true)
        end
```

It is not surprising that such changes are easy to achieve – orchestration rules can mostly be re-modeled as pre-conditions (in fact orchestration rules are pre-conditions, but unless we have some sort of formal proof for this, it cannot be stated as a rule).

Let us consider a further example of extending the behavior. Supposing, it is wished that the robot should “freeze” its walking state when picked up from the ground (during its walking) and then when placed back it should continue from the point where it stopped. To simplify this requirement, let us state that the phases of swing and retraction should be “conscious” of the robot’s situation on the ground.

We can translate this to a routine that will kick when it detects the anomaly during retraction:

```
retraction_follower(me: separate LEG_GROUP_SIGNALER)
        require
                me.legs_up or me.retracted
        do
                if not me.retracted then
                        legs.stop
                        continue(me)
                end
        end

continue(me: separate LEG_GROUP_SIGNALER)
        require
                me.legs_down
        do
                legs.retract
        end
```

It is indeed very interesting to see how behavior can be modeled using the contractual pre-condition that causes waiting until the condition is satisfied. Note that this could be solved even more elegantly by using a separate agent call to stop the legs whenever a retracting leg senses no load. Or, perhaps even more elegantly by specifying it as an invariant and then handling the asynchronous exception. But both these concepts are still in planning or implementation stage in the SCOOP project.

7.3 Impression

There is no doubt about the elegance in the object-oriented code with embedded concurrency – it not only enables implementing programs based on natural models (or natural language thinking), but also preserves object orientation in such control programs. The implicit mechanism that provides for safe access of separate objects does away with conscious coding to lock and protect resources. It is noteworthy that writing a robotic control program like the hexapod in a language that supports design-by-contract already moves it closer where its correctness can be stated (correctness also being fundamental requirement to robotic control program, perhaps more critical considering that the error would have a physical manifestation).

Use of a pre-condition on separate calls which function as wait-until condition simplifies the implementation of coordination. This fact is well illustrated by the code implementation demonstrated above.

The facility of extending the semantics of once routine with separate objects proves invaluable for easy implementation of so called "thread safe singletons":

```
io_port:separate SERIAL_COMM is
        -- The separate singleton instance of SERIAL_COMM.
        once
            create Result.make(0)
        end
```

One aspect that could have proved useful in the implementation of Hexpod's infrastructure is support for separate agents. This would be an ideal solution to the SensorPoller for knowing its recipient. Separate agents however do not work due to a bug in the SCOOP implementation.

7.3.1.1 Are these observations generic?

It could be easy to get carried away because of SCOOPs facilitation of easy hexapod implementation. We will briefly state some other automation and robotics control example to see if the strengths that we have identified above still hold.

7.3.1.1.1 Arm Robot

The implementation of arm robot (described in (Nienaltowski P. , 2007)) uses separate calls and condition synchronization to achieve a seemingly simple robotic action. The arm robot is equipped with a light sensor on the claw and the arm is positioned over the ground and starts scanning to detect colored objects. On finding an object the turntable is stopped and the arm is lowered over the object. The claw, which was also opened during the drop operation, is now closed to grab the object, after which the arm is raised while the turntable simultaneously moves to a bin position (where the object is dropped).

In this example we see slight concurrency and coordination requirements (i.e. not as elaborate as the hexapod, but then not all robotic applications are necessarily exotic!). Use of SCOOP as the programming tool implements the control program for the arm robot in an elegantly object oriented way.

7.3.1.1.2 Double shaft Elevator control

A practical implementation of double shaft elevator (Nienaltowski P. , 2007) involves concurrent control of two motors while taking inputs from floor buttons. The solution model uses asynchronous messaging (i.e. separate calls) to achieve coordination. This is in contrast to the hexapod's solution which uses pre-condition to achieve coordination. The implementation of elevator program serves to reinforce the assertion that SCOOP's object-centricity provides opportunity for the application developer to translate the solution from natural thinking (pressing of a button to achieve some action can easily be thought of as messaging than imagining a process waiting for the button to be pressed).

8 Conclusion

One needs to be slow to form convictions, but once formed they must be defended against the heaviest odds – Mahatma Gandhi

We will now conclude upon the results of this project, but without repeating the debate between object-oriented concurrency versus traditional paradigms. There is no doubt that robotic control benefits immensely from object-oriented concurrency methodology – we now need to pragmatically view the application of SCOOP to robotic control (in comparison to traditional methods).

8.1 Strengths

SCOOP provided the following benefits in programming the hexapod. Some benefits are also jotted down by drawing on experience with other examples.

- Augmenting the type with **separate** keyword greatly simplifies the definition of concurrent and autonomous entities in the object model.

- A well defined strong type system seamlessly handles separateness – it elegantly provides compile time check for assignment and argument passing conformance when dealing with separate objects.

- Rule to access a separate object after implicitly locking its processor enforces programmatic discipline in handling autonomous entities. Without this, it can get easy to violate what should have been a protected access to a resource (for example imagine a scenario of one client calling `motor.get_position` before making a decision and meanwhile another client calls `motor.move)`

- Separate calls (not involving lock passing) are executed asynchronously while queries are synchronous – this is not only useful but also intuitive. This feature is very interesting for process control scenarios where we wish to operate on an aggregate of separate entities and finally wait for their completion (for example a routine consisting of asynchronous calls like `turntable.move_to_end`, `arm.drop`, `claw.open` followed by `pos := turntable.position` which waits until all executions on `turntable` are completed.)

- Separate calls on object that result in routine with loop-like action causes the processor to be set as busy and hence prevent access from clients. (For example a motor controller that has routine like move which has a `while position_reached motor.rotate` like loop provides a perfectly protected closed loop controllers).
- Precondition evaluation on separate entities with wait like semantics, as mentioned several times in this document, is a priceless tool for establishing coordination and synchronization.
- Once routine returning separate objects functions as classic "thread-safe singleton" in a simple manner. Such routines function to serve out unique resources like serial port, network port access etc.

8.1.1 Influence of SCOOP on robotics programming

We have already seen that robotics programming has to deal with entities like sensors and actuators which further aggregate to functional parts like arm, leg, conveyors etc. They also have to function in coordination with non-physical entities like databases and control logic. It is not difficult to see the benefit of applying object-oriented concurrency here. We now discuss the specific features and mechanisms of SCOOP that can be applied to robotics programming environment.

8.1.1.1 Creating and accessing separate objects

Concurrent behavior in robotics is well known. Behaviors that can be exhibited by an object are defined in its routines but the concurrency and coordination is left open – which is solved in SCOOP by simply declaring the object instance to be separate. After which point the separate object instance can be treated as one would imagine interacting with the physical entity itself. The following examples emphasize this aspect:

```
arm : separate ARM
turntable: separate TURNTABLE
…
arm.raise
turntable.move_to_end
```

With the above code construct we aim to create two separate entities, the arm and the turntable and wish to treat them as concurrently working components. To accomplish this, all that is required is to qualify the instance of arm with extra information about its separateness. Once this is done, the routine call (`raise`) is by nature asynchronous and so the turntable's `move_to_end` will be executed concurrently. What about executing more than one behavior in parallel in an object (say `arm.raise` and simultaneously `arm.openClaw`)? This is not possible in the SCOOP model as the call to `arm.raise` sets its processor busy and will wait until that is completed. If indeed arm's raise and claw open action can happen concurrently, then the arm needs contain a "claw" object to which it can delegate the openClaw action. On careful thought, what seems to be a draconian rule based on SCOOP's view of concurrency (i.e. only one routine execution in a object's context) is actually conformant to behavioral modeling – which is to have an entity perform a (single) behavior in response to stimulus and return to idle state (refer to concept of subsumption). An object performing multiple actions concurrently is a vestigial product of multi-threaded approach. It is easy to see that with the approach adopted by SCOOP it is possible to reason about the correctness – the arm's raise action is guaranteed to run under conditions that are not being modified by some other client.

8.1.1.2 Race condition free concurrency

Race conditions occurs where an object's ownership is shared among multiple clients and hence its state is liable to be changed in response to routine calls from the clients. In SCOOP, The ownership of a separate object is held by only one client under clearly defined rules. The rules are also flexible enough to allow the programmer to pass references of the separate object without endangering safety. Consider the following example:

```
class SERVO

get_position:INTEGER
set_poistion(val:INTEGER)
```

Obviously in a scenario where the an instance of SERVO can be accessed concurrently, there is no guarantee about making a set_position call based on the value obtained from get_position (in between the get_position and set_position, another client instance could have called set_position and hence altered the state). The only way to solve this in multi-threaded program is to have a shared lock object. In SCOOP, this problem is solved by the client obtaining lock on the processor of the SERVO instance – which guarantees that only one client has access to the services of the SERVO until it relinquishes the lock on the processor.

```
Class TURNTABLE

go_to_nearest_end(servo:separate SERVO)
do
        if servo.get_postion > 50 then
                servo.set_position(100)
        else
                servo.set_position(0)
        end
end
```

In the code example above, it is guaranteed that in the scope of the routine go_to_nearest_end, no other client can call set_position on the servo instance.

8.1.1.3 Pre-conditions on separate calls

Perhaps the most powerful feature SCOOP offers to handle coordination in concurrent situations is the wait semantics on pre-conditions of separate calls. Essentially it allows one to concentrate on the sequence of behaviors and delegate waiting in the behavior's contractual pre-condition. Let us consider a situation where an orchestrator wishes to invoke sequence of actions on components while taking care of coordination. Say in an arm robot we wish the arm equipped with a gripping claw to lower itself until the claws are over an object and then grip and pick up the object. This happens concurrently with the turntable moving to the target position.

```
class CONTROLLER
…
pick_object(arm:separate ARM; claw:separate CLAW; table:TURNTABLE)
do
        table.move(50)
        claw.open
        arm.lower
        claw.close
        arm.raise
end
--------------------------------------------------
class CLAW
…
close
requires
        object_in_claw : claw_obj_sensor = true
do
        gripper.close
end
--------------------------------------------------
class ARM
…
lower
requires
        claw_open : claw.gripper.is_open
do
        servo.move(100)
end


raise
requires
        claw_closed : claw.gripper.is_closed
do
        servo.move(100)
end
```

In the above code snippets, we see that the controller calls for actions on the turntable, arm and claw. The routines are executed asynchronously and the pick_object terminates. The claw's close action will proceed as soon as the object is within the claw, whereas the arm's raise action will wait until the gripper is closed. In such scenarios we see that the request for a behavior and the actual execution subject to pre-conditions are neatly decoupled. I.e. we now rely on contracts to implement behavior – the claw's close behavior, for example, contractually requires an object to be within it. In the case of separate calls we know that this execution will wait until the contract is satisfied.

From the above illustration, we see that the orchestrator can fire off requests on separate objects and the coordination is achieved by the pre-conditions. Of course, the coordination has been stowed away into the pre-conditions, which means that routines now need to reflect behavior. For example the arm's `lower` routine is specifically meant for lowering to grab an object. If the arm can also lower without any purpose then it needs to be modeled as a separate routine (say, we have two routines `lower_to_pick` and another `lower`). Alternatively, the orchestrator can directly use pre-condition clauses on separate objects to achieve the waiting. Let us modify the example cited above to prove the point:

```
class CONTROLLER
…
pick_object(arm:separate ARM; claw:separate CLAW; table:TURNTABLE)
do
        table.move(50)
        claw.open
        lower_arm_to_pick(arm, claw, table)
        claw.close
        arm.raise
end

lower_arm_to_pick(arm:separate ARM; claw:separate CLAW; table:TURNTABLE)
requires
        claw.gripper.is_open
        table.position = 50
do
        arm.lower
end
```

We see that the contractual condition helps us now reason about the correctness of the program - we can ascertain that the arm will not be lowered to pick up the object unless the claw has opened and the table is in the right position.

8.1.1.4 Separate agents

A classical use of agents is in implementing publisher-subscriber scenario. In robotics, we often find this being used in propagation of sensory information. Usually an entity exists which represents the sensor and one or many clients of this sensor will be interested in receiving updates whenever the value changes. Let us assume that the sensor has to poll a hardware subsystem underneath to retrieve the current value after which it will send out updates based on the change criteria (say, the difference between the current and last known value). This means that the sensor is logically a separate object and hence has to send out its update to clients which are separate. Provision of separate agents caters exactly for this scenario. The following example demonstrates this feature:

```
Class SENSOR
LIST[separate agents] subscribers

Subscribe(separate agent)
Do
        Subscribers.add(s)
End

Notify
Do
        Foreach(subscriber in subscribers)
                Subscriber.update(value)
```

```
end
```

8.1.1.5 *Once routines*

In hardware related programming, we often encounter resources that have a unique instance in the system and is concurrently accessible. Typically these are modeled as thread-safe singletons (Gamma et al). While such singletons guarantee existence of a unique instance, the actual resource instance needs to be protected against concurrent access. For example, a serial port is not only unique but also used in a mutually exclusive manner. In SCOOP, the extension of **once** routines provides for such infrastructure:

```
Port:separate SERIAL_PORT
once
        result := serialport.make
end
```

8.1.2 SCOOP'S place in robotic systems architecture

Most robotic architectures are layered constructions although there are exceptions to this rule like in the subsumption model. A comprehensive survey of such architectures can be found in (Simmons & Kortenkamp). Layered architectures focus on abstracting functions from the lowest sensory-actuation to high-level planning and scheduling. We take a concrete example to explain the intent of such architectures – consider the function of an arm robot that must pick up packages from a conveyor and stack it in bins. The highest layers concerns itself in planning which packages should be given priority and how to handle conflicts etc. It then puts together tasks, which are usually hierarchal, and passes it on to the executive layer which will then schedule the tasks like picking packages with certain barcode and moving it to specific bins. The executive layer carries out such tasks using the services provided by the behavioral layer. The lowest layer (behavioral) is capable of causing the physical actions on the robot and gathering feedback regarding the environment. Such feedbacks, like how many bins are still empty, are passed back to the planning layer for its cognition.

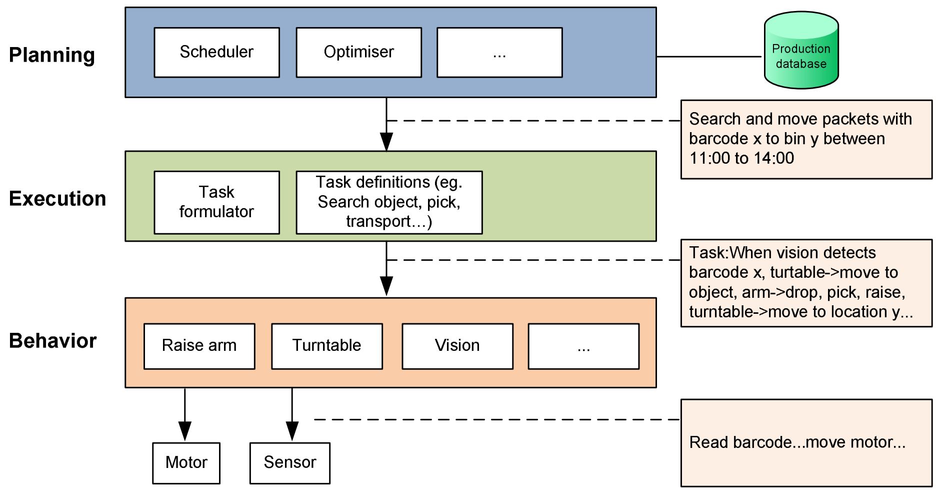

Planning
Scheduler
Optimiser
...
Production database
Search and move packets with barcode x to bin y between 11:00 to 14:00
Execution
Task formulator
Task definitions (eg. Search object, pick, transport...)
Task:When vision detects barcode x, turtable->move to object, arm->drop, pick, raise, turntable->move to location y...
Behavior
Raise arm
Turntable
Vision
...
Motor
Sensor
Read barcode...move motor...

Figure 32: Layered architecture for robotics control

SCOOP's methodology is not proposed as a framework for construction of any of the layers described above. Operations like task planning, plan correction, scheduling, execution control, behavior definition are not addressed by SCOOP. What is indeed elegantly possible is the implementation of any of such framework layers using SCOOP. In each of the aspect in such frameworks, there is need for handling concurrency and coordination. Even in the highest layer planning may have to be conducted concurrently and the progress of each plan construction may depend on the intermediate result of others (or from environmental feedbacks).

In this research we have clearly demonstrated the advantage of using object-oriented concurrency in the executive and behavioral layers (namely gait processor and leg coordination). Investigating systematic usage of SCOOP in a sample robotics framework would be a very interesting and fruitful effort.

8.2 Weakness

As with any practical programming paradigm and environment certain shortcomings in implementation of SCOOP hinders efficient development of robotics control programs. A summary of such problems are listed below:

- Difficult, if not impossible, to debug a SCOOP program.
- Tool to view states of separate entities, which should be logical must, is missing.
- Race conditions are eliminated, but a compile time indication of possible deadlock situations would establish SCOOP as a “safe concurrency” platform. This is missing.
- Poor performance of pre-condition evaluation on separate entities. It takes significant time to progress in the routine after a held-up precondition is satisfied.
- Missing implementation of separate agents causes compromised design implementation.
- Bugs in implementation make the platform not yet mature for serious projects.

8.3 Retrospection and closing thoughts

We have summarized earlier the attempts of using sequential, multi-threaded, event based, URBI, Microsoft Robotics Framework and Subsumption architecture as programming models. Taking a step back and comparing with what has SCOOP has to offer, we realize that the conceptual gain obtained from Object Orient Concurrency is immense. Frameworks like the Microsoft Robotics and URBI are moving in the right direction of providing a model for concurrency that does not involve explicit threading and synchronization. But take note that these are dedicated products for robotics programming – the fact that SCOOP proves to be a strong alternative speaks for its generic nature in solving concurrency and coordination problem.

In addition to various concept that attempt to provide better concurrency and coordination mechanism (for robotics and otherwise), there is also the movement to structure behavior modeling (arising out of bio-inspiration and artificial intelligence quest). We briefly looked into the Subsumption architecture and noted its strength in simplifying autonomous systems to decentralized nodes of behaviors. After having looked at SCOOP’s mechanism, we realize that implementation of subsumption like patterns are simplified by object separateness and contract oriented concurrency. In subtle terms, behavior finds and equivalence in routine coupled with its pre-conditions. Thanks to object-oriented concurrency, we don’t have to essentially think of Routines = Behavior, but Object = Stateful Behavior.

With further experiments, projects and research interest in SCOOP we can not only prove its mettle, but also strengthen it with feedbacks and new ideas. We hope that this research has served to prompt such exploration and questioning (and answering!).

Appendix – A: Hardware Construction

a. The Hardware

Legs

The Hexapod hardware is a kit manufactured by LynxMotion (www.lynxmotion.com). The exact model description being Hexapod BH-3 (www.lynxmotion.com/Category.aspx?CategoryID=33).

The Hexapod contains six legs, set of three placed symmetrically across the central axis. Each leg has three servo motors (Hitec HS-485HB) which has robust carbonite gears (datasheet : http://www.hitecrcd.com/servos/show?name=HS-485HB)

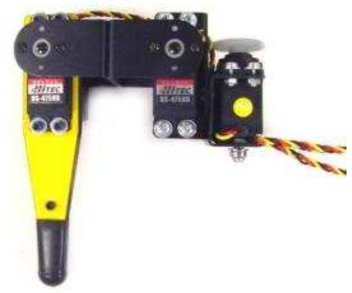


Figure 33: Servo motor and its placement on the leg

Note: The hex screw-heads on the hardware are to SAE standard – use hex key size 3/32.

Servo Controller

The servos are connected to a device called SSC-32, which is a servo controller with serial interface. Servo Controllers accept commands on the serial port and position the servo using pulse width modulation. Each of the servos is connected to the board on pins that are addressed. On the hexapod, the right legs (3 x 3 servos) are connected to addresses 0 through 8, while the left servos are connected to addresses 16 through 24. The communication protocol of the SSC-32 is described in the document Tool\SSC32_Reference.pdf.

Figure 34: Servo controller board

The connections on the SSC-32 are shown on the schematic below:

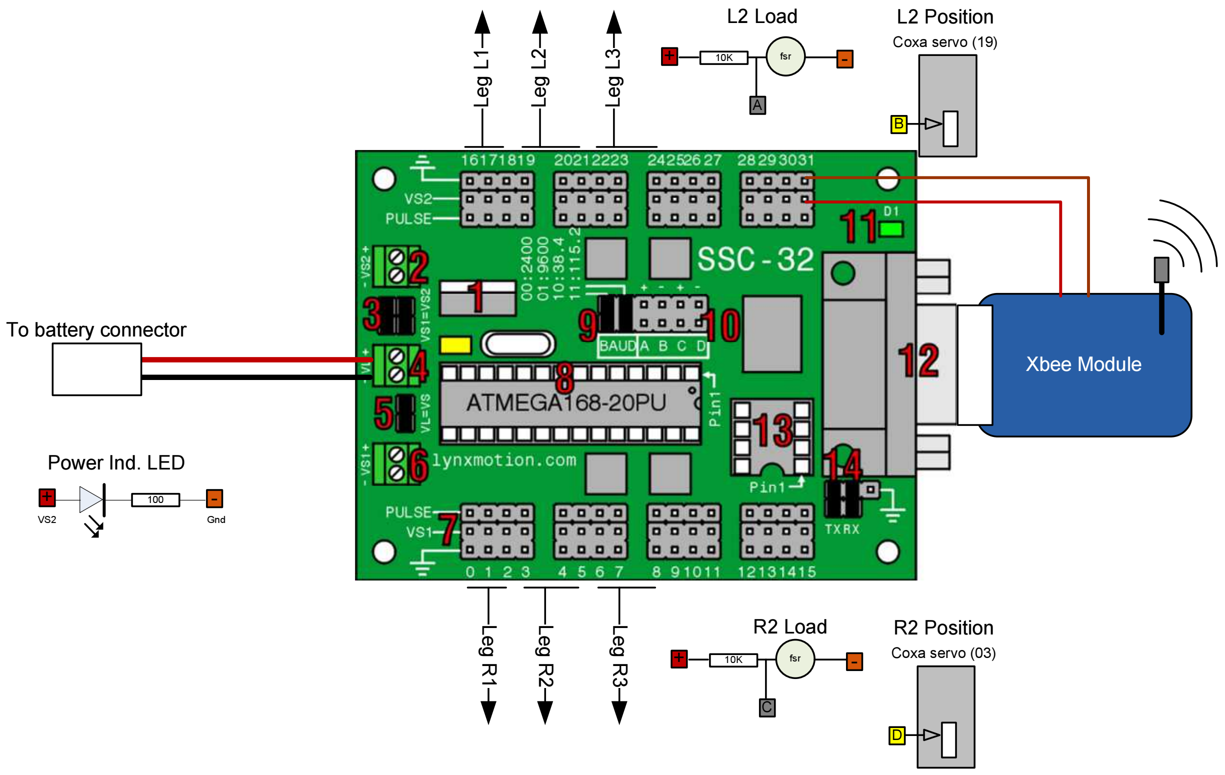


Figure 35: Wiring schematic

Refer to the SSC-32 manual for detailed description of the board.

Sensors

The SSC-32 has four input pins labeled A, B, C and D. We have connected the load and angle sensors of leg R2 to pins A and B respectively and similarly the load and angle sensors of leg L2 are connected to pins C and D.

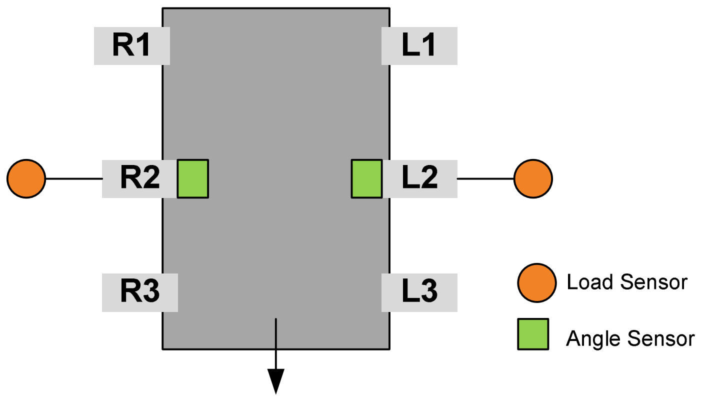


Figure 36: Location of sensors

Load Sensor

The load sensor is a force sensitive resistor (FSR) which is a thin film like element placed on the heel of the leg.

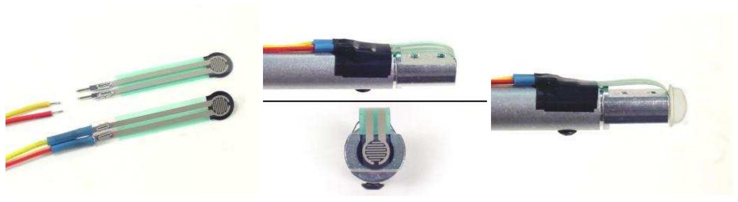


Figure 37: Load sensor construction

As the load on the sensor increases, its resistance drops. This can be converted to voltage by using a simple resistance ladder:

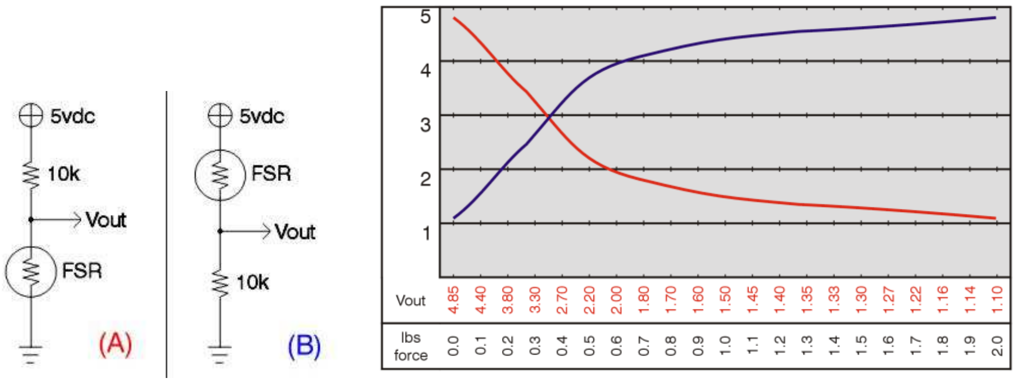


Figure 38: Load-Output characteristics of the load sensor

We have adopted configuration (A) – this means that as the load is increased, the analog value read changes from approximately 255 (no load) to about 160 (full load).

Since the no load value can drift away from 255 (for example due to tightening of the black cap), the control program should provide for a method to calibrate the no load condition.

Angle Sensor

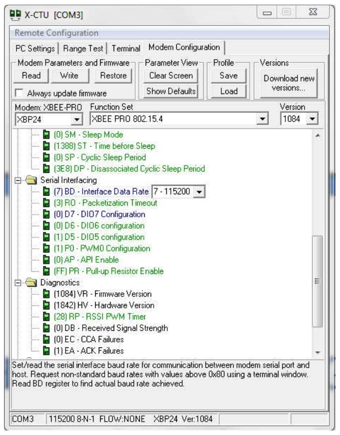
The actual position feedback of the Servo motors (i.e. the “Coxa” motors of R2 and L2) are obtained by tapping into the position sensing potentiometers within the servos. This involves a small modification of the Servo motor wherein the central tap of the potentiometer is wired out of the casing. The servo is commanded using the pulse width duration (500 – 2500 us = 0 – 180 degrees, linear). On the analog input pin (which measures 0..254), we get a change of 20 for change in pulse width of 500.

Communication to PC

The communication between the SSC-32 and the PC host program takes place via a wireless (ZigBee) serial link. The SSC-32’s serial port (DB9) is plugged directly into the ZigBee module. On the PC side, the ZigBee module is connected via an USB cable. The PC side module is set to function as a virtual serial port UART. The driver of this virtual serial port is usually built into the OS and should install itself. If required, this driver can also be downloaded from the FDTI website (http://www.ftdichip.com/FTDrivers.htm). Following which the Device Manager recognizes the module as a normal serial port (COMx). A tool can be used to configure the modules – this tool (X-CTU) is archived in the project SVN folder \Tools.

Figure 39: ZigBee module

The modules are configured to support serial communication at 115200 bps, 8 bits, no parity.



```
Modem: XBEE-PRO   Function Set            Version
XBP24             XBEE PRO 802.15.4       1084
    (0) SM - Sleep Mode
    (1388) ST - Time before Sleep
    (0) SP - Cyclic Sleep Period
    (3E8) DP - Disassociated Cyclic Sleep Period
  Serial Interfacing
    (7) BD - Interface Data Rate  7 - 115200
    (3) RO - Packetization Timeout
    (0) D7 - DIO7 Configuration
    (0) D6 - DIO6 configuration
    (1) D5 - DIO5 configuration
    (1) P0 - PWM0 Configuration
    (0) AP - API Enable
    (FF) PR - Pull-up Resistor Enable
  Diagnostics
    (1084) VR - Firmware Version
    (1842) HV - Hardware Version
    (28) RP - RSSI PWM Timer
    (0) DB - Received Signal Strength
    (0) EC - CCA Failures
    (1) EA - ACK Failures
Set/read the serial interface baud rate for communication between modem serial port and
host. Request non-standard baud rates with values above 0x80 using a terminal window.
Read BD register to find actual baud rate achieved.
COM3   115200 8-N-1  FLOW:NONE   XBP24  Ver:1084
```

Figure 40: Configuration of the ZigBee module

On the tool, for both module set the Interface Data Rate = 115200 and DI07 Configuration = Disable. This is the only setting that is needed on the default. This profile is also stored in the \Tools folder as .pro file.

Battery

The battery is located under the SSC-32 board (taped to the bottom plate). It is a Ni-Mh 7.5V 2800mAh bank with Tamiya connectors.

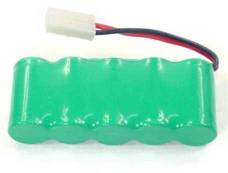


Figure 41: Battery

The battery is connected to the SSC-32 board (which also jumpers the supply to servos) via a single pole toggle switch.

b. Extending the robot

One definitive area where the robot might be required to be extended for future projects is addition of sensors. The existing SSC-32 board is capable of accepting three analog inputs. Each analog input can be expanded to multiple digital inputs using a ladder circuitry (theoretically, each analog input can be expanded to eight digital inputs, but perhaps up to three is more practical). In case more analog inputs are desired, it is recommended to add a separate input circuitry and couple it to the serial communication on the existing board.

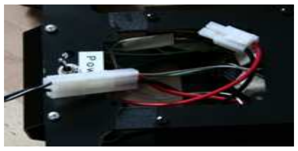
Appendix – B: Hexapod User's Guide

This part of the document guides you step-by-step how to use the Hexapod robot with SCOOP program for demonstration. Please make sure that you read and understand this guide before using the robot.

a. Charging the Battery

The battery pack built into the robot should be fully charged before a demonstration. Weak batteries will cause the servo control to fail.

1. Disconnect the Tamiya connectors on the robot and connect the end marked B to the charger.



2. The charger should be set to 5-Cells, 1000mA setting.

3. Connect the charger to mains supply. The LED will light up in red color to indicating charging process.
4. Once the batteries are fully charged, the LED will turn green. At this point, the battery can be disconnected from the charger and connected to the robot supply input. It is not possible to interchange the polarity during plugging.

b. Software Installation (Estimated time 1 hour)

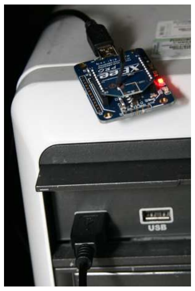
Refer to the document readme.txt in the project store \Distributable\ValidVersion for the latest installation instructions.

1. Copy the entire folder hexapod_v2 into your local disk location
2. Copy the hexapod.dll from the hexapod_dll folder to a local disk location (say c:\hexapod\dll\)
3. In the System Enviroment variables, add the folder location of the dll (above) to the Path.
4. Start Eiffel studio and open the hexapod_v2 project file and compile.

Alternatively, you can also copy the executable from the \Distributable \ValidVersion\Executable folder instead of using the source code. Steps 2 and 3 have to be carried out in any case.

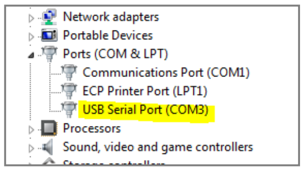
c. Setting up serial communication hardware

1. Connect the ZigBee PC-side module using the USB Connector.



2. If it is the first time that you are connecting the ZigBee module, the Windows OS will take few minutes in installing the driver. . If required, this driver can also be downloaded from the FDTI website (http://www.ftdichip.com/FTDrivers.htm).
3. The red LED on the ZigBee module should start blinking as a confirmation.

Note: after successful installation of the driver, the plugged in module should show up as a COM port in the device manager:

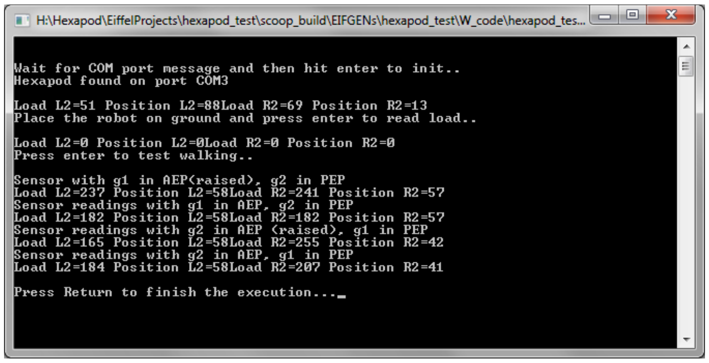


```
Network adapters
Portable Devices
Ports (COM & LPT)
    Communications Port (COM1)
    ECP Printer Port (LPT1)
    USB Serial Port (COM3)
Processors
Sound, video and game controllers
```

d. Testing the robot

The demo kit comes along with a test program called hexapod_test.exe which can be used to confirm that the hardware is in order.

1. Set up serial communication hardware (described above) if not already done.
2. Make sure that the Software Installation steps have been completed.
3. Power on the robot. The red LED near the power switch and the green LED on the controller board should light up.
4. Start the test program and follow the instructions on the console. The test program will cause the robot legs to be actuated and also the sensor states are confirmed. Should any of the steps fail, please refer to the section "Troubleshooting".

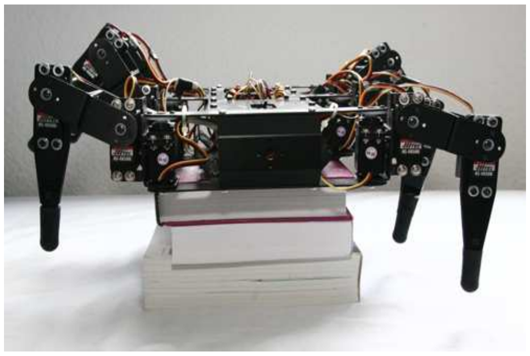


```
H:\Hexapod\EiffelProjects\hexapod_test\scoop_build\EIFGENs\hexapod_test\W_code\hexapod_tes...

Wait for COM port message and then hit enter to init..
Hexapod found on port COM3

Load L2=51 Position L2=88Load R2=69 Position R2=13
Place the robot on ground and press enter to read load..

Load L2=0 Position L2=0Load R2=0 Position R2=0
Press enter to test walking..

Sensor with g1 in AEP(raised), g2 in PEP
Load L2=237 Position L2=58Load R2=241 Position R2=57
Sensor readings with g1 in AEP, g2 in PEP
Load L2=182 Position L2=58Load R2=182 Position R2=57
Sensor readings with g2 in AEP (raised), g1 in PEP
Load L2=165 Position L2=58Load R2=255 Position R2=42
Sensor readings with g2 in AEP, g1 in PEP
Load L2=184 Position L2=58Load R2=207 Position R2=41

Press Return to finish the execution..._
```

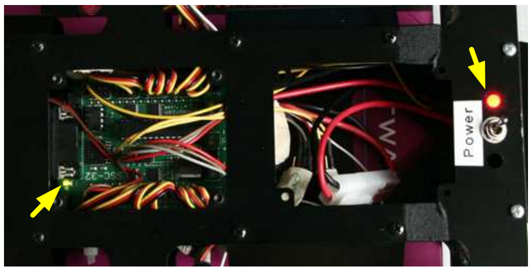
e. Running the Demo program

Executable program of the demonstration, along with the source code are provided in the distribution folder \Distributable\ValidVersion\Executable. You may either run the .exe or run the source code in Eiffel Studio 5.7.

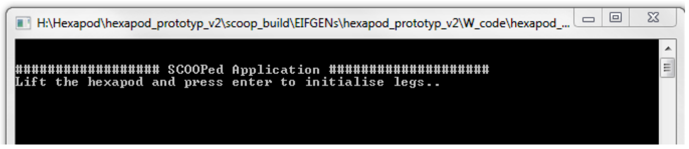
Before starting the demo program place the robot on a "perch" (a good set of fat books, or a sturdy carton would serve just right) so that the legs are dangling.



Power on the robot and ensure that the LEDs near the power switch and on the servo control board are lighted up.

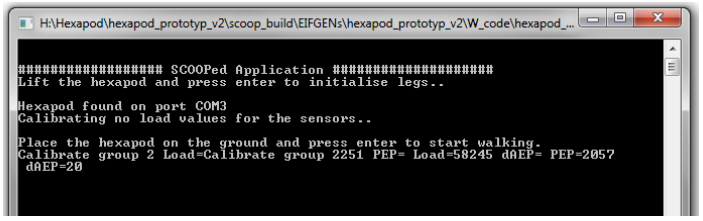


Start the demo executable, or run the source code from Eiffel studio. The console statement will confirm the initialization of the serial port and ask you to press enter to initialize the robot.



```
H:\Hexapod\hexapod_prototyp_v2\scoop_build\EIFGENs\hexapod_prototyp_v2\W_code\hexapod_...

################### SCOOPed Application ####################
Lift the hexapod and press enter to initialise legs..
```

On doing so, all legs will be pulled back to retracted position and also the no-load sensor reading will be taken for calibration.

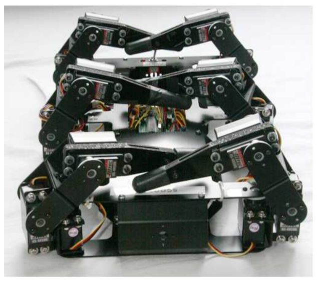
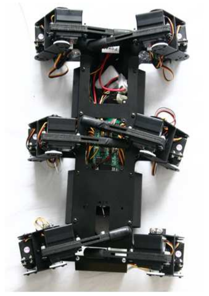


```
H:\Hexapod\hexapod_prototyp_v2\scoop_build\EIFGENs\hexapod_prototyp_v2\W_code\hexapod_...

################### SCOOPed Application ####################
Lift the hexapod and press enter to initialise legs..

Hexapod found on port COM3
Calibrating no load values for the sensors..

Place the hexapod on the ground and press enter to start walking.
Calibrate group 2 Load=Calibrate group 2251 PEP= Load=58245 dAEP= PEP=2057
 dAEP=20
```

Take the robot off the "perch" and set it on the ground. The console message prompts you to press enter to begin walking. On hitting enter key, the robot begins walking. If you pick up the robot during its walking, the protraction of the legs would stop automatically. When placed back on ground the walking will continue. To stop walking, simply close the program's console window or switch off the robot's power supply. Notice that the green LED on the servo controller board is flickering and also the ZigBee module near the PC has the LEDs for Tx-Rx flickering.

f. Storing the Robot

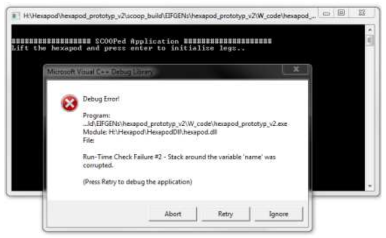
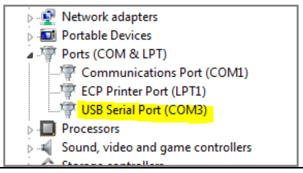
1. Switch off the power supply
2. Disconnect the battery connector.
3. If you intend to store the robot away for more than a month, then it is advisable to discharge the batteries. To do so, connect the batter to the charger and press the yellow button. The discharge process will start (LED will be yellow) and once finished the LED switches off.
4. Fold the legs upwards so that the feet lie on the top surface.

5. Put the robot in a clean box (or stiff bag) and place in a relatively dry place. For long term storage, put few de-hydrant crystal pouches in the box or bag.

g. Troubleshooting

Problem	**Solution**
Robot does not power up. The LED near the power switch and on the controller board does not light up.	Check if battery is charged (connect it to the charger and if after an hour it is still charging then most probably it was out of charge). Make sure that the connectors are snapped in fully. Check the connection terminal screws on the board.
The LEDs light up, but the robot does not respond.	Check if battery is charged. Check if the RX LED on the controller board flashes. If not refer to troubleshooting of serial communication.

Exception message from hexapod.dll on trying to run the robot:  H:\Hexapod\hexapod_prototyp_v2\scoop_build\EIFGENs\hexapod_prototyp_v2\W_code\hexapod_...
`################### SCOOPed Application ###################`
`Lift the hexapod and press enter to initialise legs..`
Microsoft Visual C++ Debug Library
Debug Error!
Program:
...ld\EIFGENs\hexapod_prototyp_v2\W_code\hexapod_prototyp_v2.exe
Module: H:\Hexapod\HexapodDll\hexapod.dll
File:
Run-Time Check Failure #2 - Stack around the variable 'name' was corrupted.
(Press Retry to debug the application)
Abort Retry Ignore
	Check if the ZigBee PC side module is connected and that the red LED is blinking.
	
	Check if the robot is powered on
	
	Check if the ZigBee module on the robot is securely plugged in.
	
	Check Windows Device manager to see if the ZigBee module is recognized as a COM port:  Network adapters
	Portable Devices
	Ports (COM & LPT)
	Communications Port (COM1)
	ECP Printer Port (LPT1)
	USB Serial Port (COM3)
	Processors
	Sound, video and game controllers

The load sensor reads value < 225 when the robot is off ground (i.e. no-load value).	Check if the rubber cap on the feet (R2 / L2) has slid up too tightly. If so, gradually pull it down till the sensor reading is just above 240. Re-secure with electrical isolation tape.
Sensor reads are erratic or fixed to 255	Check the connection (A B C D) on the SSC32 board
The robot freezes midway during walking	Check if the green LED on the SSC32 board is flashing. If so, check if load sensor value of the raised leg is > 240 (if not, diagnose sensor problem) If the green LED on the SSC32 board is not flashing then the SensorPoller task has hung-up. We know of a possible problem in the SCOOP's scheduler – the only solution is to re-start the program.

h. Tools

For troubleshooting or extension of the robot, the following tools will be required (depending upon the task):

Software

1. Lynxmotion SS32 control panel (SS32.exe)
2. ZigBee configuration tool X-CTU

Hardware

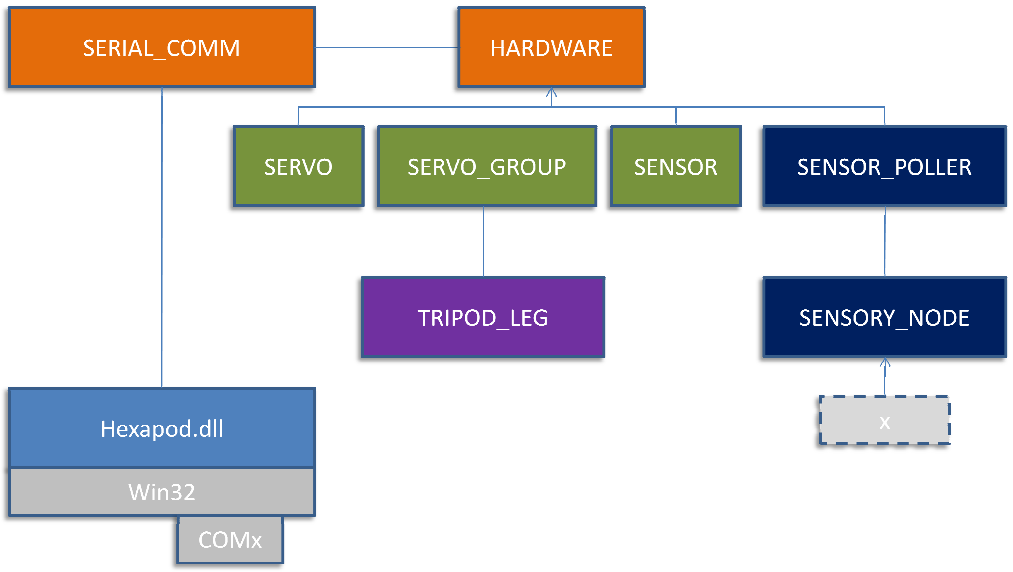
1. Philips head screw driver no. 2 and 3
2. Screw driver size 0
3. Hex-key, SAE 3/32
4. Pliers
5. Electrical Insulation tape (Textile)
6. Soldering station

i. Precautions

1. The servo motors are capable of high torque movements – do not place fingers in the way of the leg segments when the robot is powered on.
2. The battery holds considerable charge – do not short the terminals as it can cause melting of the connecting wires and consequent injuries.
3. Always use screw-drivers and hex-keys of correct size when servicing the robot. Note that the hex-head screws are of SAE size 3/32.
4. Never try to move the servo motor beyond the physical limits, or when it is powered on – it can permanently damage the gears.
5. Do not control the robot so as to make full speed impact of feet on hard ground. This can cause rapid wear out of the gears.
6. Never try to connect any kind of power supply other than 6VDC and capable of supplying at least 2000mA.
7. Do not use the wireless serial communication in environments sensitive to electro-magnetic emissions (refer to ZigBee standards for specifications).
8. Use the robot on surfaces only.

Appendix – C: Framework Software

One of the objectives of this project is to create a teaching aid to illustrate SCOOP's conceptual strength in adapting to different concurrency and coordination problem. It is also foreseen that in future researchers and students may be interested in implementing alternative designs for control. To avoid future developers reworking on integration of the hexapod hardware, a selected sets of EIFFEL classes and C DLL has been provided. This section describes the interface and usage of this infrastructure.



Hexapod.dll

The hexapod.dll is a C DLL that acts as a wrapper to establish communication between the Eiffel program and the hexapod hardware. It not only establishes serial communication, but also provides functions to command the servos and read sensor values. Hence, this component is closely tied to the hardware. In order to decouple the business classes of the Eiffel program from the implementation of the hexapod.dll, yet another wrapper class (SERIAL_COMM) exists in the framework.

Runtime model

The hexapod.dll is loaded into the process space of the Eiffel program runtime and is unloaded only on end of execution.

8.3.1 Interface functions

8.3.2 Open Communication Port

```
int Open(usigned short portNumber)
```

Opens a serial communication port of the specified number (eg. portNumber = 10 results in attempt to open "COM10"). Argument for port specification is numeric instead of the widely expected string – this is to avoid the complexity of handling string pointers on Eiffel program part.

8.3.2.1 Arguments
portNumber : Unsigned 16 bit number specifying the COM port to open

8.3.2.2 Preconditions
None

8.3.2.3 Postconditions
The port is opened and the handle is stored else -1 is returned.

8.3.2.4 Returns
Positive non-zero Integer representing the Handle to the opened port (if successful) else returns 0 if port could not be opened. Returns - *n* if a port with handle *n* is already open.

8.3.2.5 Side effects
Stores the opened port handle.

8.3.2.6 Exceptions
Passes any Win32 error as exception to the calling program.

8.3.3 Close Communication Port

```
void Close(void)
```

Closes an existing serial port if open.

8.3.3.1 Arguments
None

8.3.3.2 Preconditions
A valid port communication handle exists.

8.3.3.3 Postconditions
The existing serial port communication is closed.

8.3.3.4 Returns
None.

8.3.3.5 Side effects
Clears the stored port handle.

8.3.3.6 Exceptions
Passes any Win32 error as exception to the calling program.

8.3.4 Get Hardware Version Information

```
unsigned short GetFirmwareVersion()
```

Returns an unsigned 2-byte number indicating the Major version number. It is foreseen that this function is used as a "beacon" to confirm that the serial port is attached to a powered-up Hexapod hardware. The query for firmware version should be the very first command sent to the Hexapod device.

8.3.4.1 Arguments
None.

8.3.4.2 Preconditions
A valid port communication handle exists and the Hexapod hardware is connected one of the computer's serial port and switched on. This should be the very first command sent to the device.

8.3.4.3 Postconditions
Returns a 16 bit unsigned value.

8.3.4.4 Returns
Zero if query for version failed (i.e. the serial port has no live Hexapod device) or a non-zero positive value indicating the major version number.

8.3.4.5 Side effects
None.

8.3.4.6 Exceptions
Passes any Win32 error as exception to the calling program.

8.3.5 Getting Sensor Values

```
unsigned int GetSensors()
```

Returns a 4-byte unsigned integer where each byte represents the value of the sensor input (there are fours sensor inputs on the Hexapod device board).

8.3.5.1 Arguments
None.

8.3.5.2 Preconditions
A valid port communication handle exists and the Hexapod hardware is connected one of the computer's serial port and switched on.

8.3.5.3 Postconditions
Returns a 32 bit usigned value.

8.3.5.4 Returns
Returns a 4-byte value where the MSB = sensor value on Port A , followed by B, C and D and each value ranges from 0x00 to 0xFF.

8.3.5.5 Side effects
None.

8.3.5.6 Exceptions
Passes any Win32 error as exception to the calling program.

8.3.6 Commanding Servos

```
int SetServo(byte addr, unsigned short value, unsigned short time)
```

Commands a servo motor to move to target position in the time specified.

8.3.6.1 Arguments
addr: The address of the servo (port id. On the servo controller board). Refer Appendix A for details.

value: The desired servo position (valid range 500..2500, refer Appendix A for details)

time: The time in which the desired position should be reached. For the first SetServo command this argument is ignored and results in motion at highest speed. Setting time=0 also causes full speed movement.

8.3.6.2 Preconditions
A valid port communication handle exists and the Hexapod hardware is connected one of the computer's serial port and switched on. The value should be in the specified range.

8.3.6.3 Postconditions
Sends the command to the Hexapod device.

8.3.6.4 Returns
-1 to indicate failure, else returns 1.

8.3.6.5 Side effects
None.

8.3.6.6 Exceptions
Passes any Win32 error as exception to the calling program.

```
int SetServoGroup(byte group, unsigned int pos_1_0, unsigned int pos_1_1,
unsigned int pos_1_2, unsigned int pos_2_0, unsigned int pos_2_1, unsigned
int pos_2_2, unsigned int time)
```

This function, an optimiser for the Tripod gait, enabled synchronised movement of the tripod leg. The synchronised motion helps in avoiding the "unnatural" effect of sending nine SetServo commands (to three joints each of three legs) which causes an perciptable lag in motion.

8.3.6.7 Arguments
group: number 1 or 2 to indicate right-based or left-based group.

pos_<s>_<j>: The desired servo position (valid range 500..2500, refer Appendix A for details) for a joint <j> in leg(s) belonging tripod side <s> (1=base, 2=vertex)

time: The time in which the desired position should be reached. For the first SetServo command this argument is ignored and results in motion at highest speed. Setting time=0 also causes full speed movement.

8.3.6.8 Preconditions
A valid port communication handle exists and the Hexapod hardware is connected one of the computer's serial port and switched on. The value should be in the specified range.

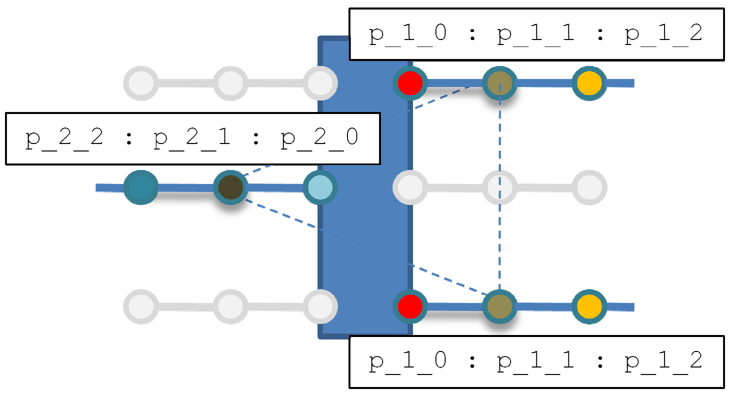
8.3.6.9 Postconditions
Sends the command to the Hexapod device.

8.3.6.10 Returns
-1 to indicate failure, else returns 1.

8.3.6.11 Side effects
None.

8.3.6.12 Exceptions
Passes any Win32 error as exception to the calling program.

8.3.6.13 Example
Function call `SetServoGroup(1, pos_1_0, pos_1_1, pos_1_2, pos_2_0, pos_2_1, pos_2_2, 2000)` will cause positions to be set on below illustrated motors and all movements will start and execute in 2 seconds.



Eiffel Classes

SERIAL_COMM

This class provides to the Input-Output functions of the Hexapod hardware – i.e. to read the sensor values and to command the servos. As mentioned previously, this class acts as a wrapper to the C-Dll functions provided by Hexapod.dll component. Note : Access to serial port functions via hexapod.dll should be controlled as an exclusive resource. Using SERIAL_COMM from multiple threads could cause overlapped transmission of serial packets and would result in erratic behaviour of the device.

Invariant

The port is opened and the handle is stored.

make (port_number: NATURAL_16)

Creates an instance of Serial Communication.

Arguments

port_number : The COM port number to open. Specify zero to cause Auto-discovery.

Side effects

Stores the opened port handle.

command_servo (addr, value, time: NATURAL_16)

Causes servo at specified address to move to value specified in given time. Refer to section Commanding Servos

command_servo_group(base, pos_1_0, pos_1_1, pos_1_2, pos_2_0, pos_2_1, pos_2_2, time : INTEGER)

Causes a group of servos on the Tripod group to move synchronously. Refer to section Commanding Servos read_sensors : NATURAL_32

Returns a 4-byte value containing the four sensor values. Refer to section Sensors.

HARDWARE

The thread-safe singleton provider of SERIAL_COMM instance.

```
	io_port:separate SERIAL_COMM is
			-- The separate singleton instance of SERIAL_COMM.
			Once
				create Result.make(0)
			end
```

The class HARDWARE should be inherited by any type that wishes to use SERIAL_COMM facility.

Wrapper Classes

The following wrapper classes provide access to hardware functions by utilizing the Hexapod.dll external calls.

SERVO: Represents a single servo motor (addr: 0..31) attached to the SSC-32 board.

SERVO_GROUP: Convenience class to utilize group commanding of servos. Refer 8.3.6.

SENSOR: Represents a sensor port on SSC-32 (A / B / C / D). Addressed as 0..3, the class has just one public query `value` which returns the value on the port as single byte.

Bibliography

Brooks, R. (1985). *A robust layered control system for mobile robots.* Boston: MIT.

Brooks, R. *Intelligence without reason.* MIT Artificial Intelligence Lab.

Durr, Schmitz, & Cruse. (2004). Behaviour based modeling of hexapod locomotion.

ETHZ. (2008). *Concurrent Programming-2*. Retrieved from http://se.ethz.ch/teaching/2009-S/0268/index.html

Gamma. *Design Patterns.*

Gostai. (n.d.). *Gostai - Robotics for everyone*. Retrieved from Gostai: http://www.gostai.com/

Gostai. (2009, November). *URBI SDK.* Retrieved from Gostai: http://www.gostai.com/downloads/urbi-sdk-2.0/doc/urbi-sdk.pdf

Lejos. (n.d.). *Behaviour based programming*. Retrieved from Lejos: http://lejos.sourceforge.net/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm

Meyer, B. (1993, September). Systematic Concurrent Object-Oriented Programming. *Communications of the ACM* , pp. 56-80.

Microsoft. (n.d.). *CCR and DSS*. Retrieved from http://www.microsoft.com/ccrdss/

Microsoft. (n.d.). *MSDN.* Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent.aspx

Nienaltowski, & Meyer. (2007). Contracts for Concurrency.

Nienaltowski, P. (2007). *Practical framework for contract-based concurrent object-oriented programming.* Zürich: ETH.

Oricom. (n.d.). *Multi-leg gait analysis*. Retrieved from Oricom: http://www14.addr.com/~oricom//projects/leg-time.htm

Papathomas, M. (1995). Concurrency in Object-Oriented Programming Languages. In Nierstrasz, & Tsichritzis, *Object-Oriented Software Composition.* Prentice Hall.

Porcino, N. (1990). Hexapod gait control by a neural network. *International Joint Conference on Neural Networks*, (pp. 189-194). San Diego.

Simmons, R., & Kortenkamp, D. Robotics Systems Architectures and Programming. In Reid.