Deriving Concurrent Control Software from Behavioral Specifications

Ganesh Ramanathan
Siemens AG, Switzerland
ganesh.ramanathan@siemens.com

Benjamin Morandi, Scott West, Sebastian Nanz, Bertrand Meyer
Chair of Software Engineering, ETH Zurich
firstname.lastname@inf.ethz.ch

***Abstract*— Concurrency is an integral part of many robotics applications, due to the need for handling inherently parallel tasks such as motion control and sensor monitoring. Writing programs for this complex domain can be hard, in particular because of the difficulties of retaining a robust modular design. We propose to use SCOOP, an object-oriented programming model for concurrency which by construction is free of data races, therefore excluding a major class of concurrent programming errors. Synchronization requirements are expressed by waiting on routine preconditions, which turns out to provide a natural framework for implementing coordination requirements in robotics applications. As demonstration application, we describe a control program for hexapod locomotion, whose implementation closely follows the corresponding behavioral specification given by the biological model. We compare the architecture with solutions expressed in more traditional approaches to robotic control applications.**

I. Introduction

Complex robotic systems such as autonomous mobile robots are typically composed of many concurrently operating components such as sensors and actuators. The corresponding software must, as a consequence, schedule and coordinate many parallel activities such as sensing, moving, and planning. The development of such applications using conventional concurrent programming approaches is challenging, partly because of the common pitfalls of concurrent programming, such as data races and deadlocks, but also because it is often difficult to keep concurrent code maintainable, reusable, and to show that it adheres to specification.

We present an application of an object-oriented programming model for concurrency to the development of robotic control programs, alleviating these problems. The approach, termed *Simple Concurrent Object-Oriented Programming (SCOOP)*, excludes data races by construction and as a result helps the programmer avoid typical concurrent programming mistakes. It also retains for concurrent programs the robust module architecture made possible by object-oriented design. While SCOOP was not designed for robotic programming – but instead with the goal of simplifying the development of mainstream concurrent applications – we found it to be useful for expressing coordination tasks in the robotics domain while ensuring a close correspondence between specification and code.

The main case study of the paper is the design of a control program for a walking hexapod robot; as a well-understood problem with many implemented solutions, it serves to clearly illustrate our approach. To provide an intuition for the central mechanisms of SCOOP early, we describe part of this problem and our solution now.

In order to preserve stability while walking, a hexapod has to ensure that a group of legs is only allowed to protract (lift the legs off the ground and move them to the front) if (1) they are retracted (moved to the back) and (2) the partner group is firmly planted on the ground. A rule such as this can be seen as part of the specification of a hexapod; any correct implementation will have to abide by it. In a traditional implementation using thread libraries (Java threads, POSIX threads, etc.) these elements of the specification will be expressed indirectly through condition variables, guaranteeing that the first leg group will not protract until signaled by the partner group. Two problems arise: since the synchronization requirements are not directly expressed in the programming language, this obscures the intention of the programmer and the original specification; programming errors are easily introduced as the programmer has to insert the signaling instructions by hand.

With SCOOP, the requirements can be explicitly stated as *wait conditions* of a routine using the **require** keyword, meaning that the routine execution is automatically delayed until the conditions are true. Assume that the following routine (method) *begin_protraction* is part of a class that defines state and behavior of a group of legs, and implements the first phase of leg protraction (lifting the legs off the ground).

```
begin_protraction (partner_signaler, my_signaler: separate SIGNALER)
    require
        my_signaler.legs_retracted
        partner_signaler.legs_down
    do
        legs.lift
        ...
    end
```

When *begin_protraction* (*partner_signaler*, *my_signaler*) is called, the runtime system attempts to lock objects *partner_signaler* and *my_signaler*; if this succeeds, no other thread can change their state, thus avoiding data races. In a next step the wait conditions get evaluated. For example, requirement (2) above is checked by evaluating the condition *partner_signaler.legs_down*. If all wait conditions evaluate to true, then the body of the routine can be executed safely, as it now fulfills the synchronization conditions. If one of the conditions evaluates to false or the runtime system cannot lock the objects because another thread has locked them, the routine will not proceed; the runtime system will continue to try locking the objects and evaluating the wait conditions

until it can proceed.

The remainder of the paper expands on the hexapod case study. We first discuss related work in Section II. We provide the specification of a walking hexapod by presenting a biological model in Section III. The basics of the SCOOP concurrency model are introduced in Section IV. Section V discusses the SCOOP implementation of the hexapod control program. We compare our approach with traditional sequential, multi-threaded, and event-based programming in Section VI. Section VII discusses how our approach scales up to larger applications, and we conclude in Section VIII.

II. Related Work

An early attempt to provide a concurrent programming language for robotics was made with Concurrent C [1], which uses rendezvous-style communication to provide synchronization on top of a general-purpose language. URBI [2] is an object-oriented script language that is coupled with useful primitives for the parallelization of tasks and for flexible handling of events. It enables the expression of complex synchronization constraints using conditions. Other languages for robotic control have a more domain-specific character. Frob [3] is based on a functional language core and provides a variety of useful abstractions for programming robotic applications in a declarative way. TDL [4] is an extension of C++ that provides explicit syntax for task-level control synchronization. Similar to URBI, SCOOP takes a middle ground: it provides a higher level of abstraction than Concurrent C through object-orientation, thus allowing for a more modular design; on the other hand it provides fewer restrictions on programmers to express their intentions than a domain-specific language.

Ada [5] has been used to implement a control program for a walking hexapod in an object-oriented manner, but mainly to provide a case study for Ada programming rather than to suggest Ada as a language for robotic control. WalkNet [6] is a framework for behavior-based modeling of walking systems. It uses a set of finite state machines that are arranged in a network in such a way that they can communicate with each other. Each finite state machine selects an action based on the state of the network and then autonomously executes the action. SCOOP is not an equivalent of WalkNet, but its mechanism could be used for the implementation of this framework.

Several middlewares build on top of existing programming languages to ease the development of concurrent control software for robots. A survey can be found in [7]. The related work section of a middleware called LCM [8] by Moore et al. presents another short survey that includes MOOS [9], CARMEN [10], and ROS [11]. These middlewares provide standards, principles, applications, and libraries to support tasks such as base and sensor control, obstacle avoidance, localization, path planning, mapping, and simulation. SCOOP makes a contribution as a programming language and as such does not directly provide support for common robotics tasks. However, we discuss in Section VII the design of a robotics framework based on SCOOP.

III. Hexapod: biological specification

Hexapods in nature have six symmetrically placed legs and are capable of walking with different patterns of leg movement (gaits). Several studies have been conducted on the mechanism of individual leg movement and inter-leg coordination. These studies have produced well-defined models of neural objects and rules of coordination between them. For the purpose of creating control software for hexapod locomotion, it is natural to base a specification on such models.

A. Hexapod leg

Each leg of a hexapod is capable of executing a set of actions to propel the body. Legs have three degrees of freedom provided by three angular joints connecting the coxa, femur and tibia segments, as depicted in Figure 1.

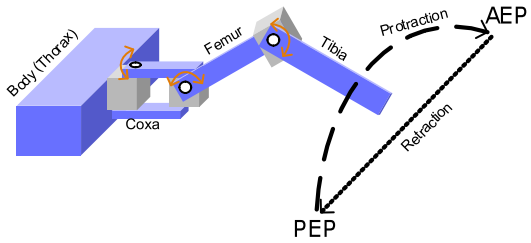

Fig. 1. Hexapod leg

The leg is raised from the *Posterior Extreme Position* (PEP) and swings itself to the *Anterior Extreme Position* (AEP) where it is placed on the ground. This is commonly termed as *protraction* of the leg. On reaching the AEP, the leg is kept on the ground and swivels the body forward, using the foot as a pivot. This is commonly termed as *retraction* of the leg.

B. Hexapod gaits

There are three kinds of leg-movement patterns in a hexapod: the *ripple*, *wave* and *tripod* gaits. The ripple gait is the slowest and allows very precise movement (only one leg is raised at a given time), while the tripod gait is the fastest (three legs raised simultaneously).

We have chosen to implement the tripod gait as it is the most precarious and hence correct coordination is critical. Figure 2 illustrates the tripod gait.

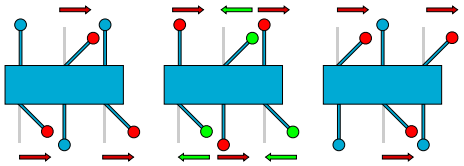

Fig. 2. Tripod gait

Two sets of tripod legs execute alternating sequences of protraction and retraction. The first group lifts the legs off the ground and swings to the AEP. The second group then lifts and swings to the AEP, concurrently with the first set which now retracts to the PEP. This cycle is repeated continuously.

We show the requirements for the tripod gait, limiting ourselves to walking on even, flat surfaces [12]:

(R1) A leg group may lift only if the partner group is planted on ground.
(R2) Protraction (retraction) starts on completion of retraction (protraction).
(R3) 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.
(R4) Protraction should end (legs depressed) only when the partner group's retraction completes. This is to avoid dragging of legs of the own group.

C. Hexapod coordination for locomotion

Extensive biological studies have been conducted to deduce the biological model of locomotion in hexapods [12], [13]. Porcino [13] describes the coordination in the form of pattern generators and inhibitory links.

A coupled set of neurons, the *Central Pattern Generator* (CPG), provides periodic electrical pulses which are 90° out of phase with respect to each other. These pulses trigger the motor mechanism to perform the protraction-retraction cycle. The pulses are generated continuously; the motor mechanism requires certain conditions to be satisfied before the action is executed. These conditions are enforced by "inhibitions", or gates, that block signal transmission. For example, a protraction invoking pulse will be blocked if the sensory neuron recognizes a high load, which occurs when a neighboring leg is raised.

Such inhibitions can be directly and mechanically expressed by SCOOP's wait conditions in our control program for a hexapod robot, as shown in Section V-B.

IV. The SCOOP concurrency model

In this section we turn to a more detailed description of the SCOOP concurrency model. Object-oriented programming models are well-suited to capture in software the components of a robot: classes define the types of components, and each component is an instance of such a class; abstraction and refinement can be included with inheritance relationships between the classes. There have been many proposals to extend the sequential object-oriented model to support concurrency; in practice, however, most applications simply extend an object-oriented language with a thread library. In thread-based models, concurrency is the result of multiple threads where each of them executes a sequence of instructions on a shared set of objects, in parallel to the other threads.

In SCOOP [14], [15], *processors* take over the role of threads. A processor is an abstract concept that must not be confused with a CPU. Just like a thread, a processor is an independent unit of control that executes instructions on a number of objects. The main difference between a thread and a processor lies in the relationship to objects. In SCOOP, every object can only be accessed by one processor: its *handler*. Each object is handled by exactly one processor; but one processor can handle multiple objects. This principle prevents data races with respect to single objects.

To illustrate the model, we take a system with one controller, one sensor, one actuator, and a mechanism that is regulated by the actuator. Figure 3 shows the objects and the associations to processors; every object that is located in a region labeled by one of the processors *p*, *q*, and *r* is handled by the respective processor.

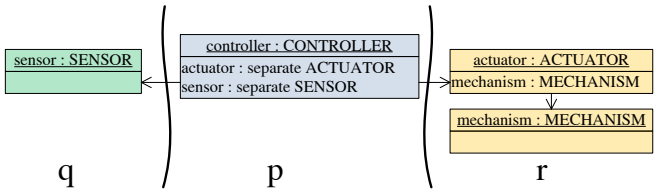

Fig. 3. SCOOP example layout

While processor *p* can access the controller, SCOOP ensures that *p* can neither directly access the sensor, the actuator, nor the mechanism. If *p* wants to access the sensor then it must ask processor *q* to do the job on its behalf. To guarantee atomicity, *p* must acquire a lock on *q* before sending a job request.

The SCOOP approach expresses such locking requirements through the formal argument list of the corresponding routine. Consider the following code that shows a feature of the controller, handled by *p*, that starts the actuator with the value of the sensor.

```
activate (sensor: separate SENSOR; actuator: separate ACTUATOR)
  require
    actuator.is_ready
  local
    value: INTEGER
  do
    value := sensor.value
    actuator.start (value)
  end
```

Before executing *activate*, *p* must acquire the locks on the handlers of the formal arguments, i.e. processors *q* and *r* which are handling the sensor and the actuator, respectively. Thanks to the locks, data races cannot occur on the set of objects handled by *q* and *r*.

Note the **separate** keyword in the signature. A type marked with this keyword denotes an objects that may be handled by a different processor than *p*. This extension of the type system enables programmers to distinguish routines with asynchronous evaluation semantics from ordinary synchronously evaluated routines.

In our system, the actuator can only be started if it is ready. In SCOOP, such a synchronization requirement is expressed as a wait condition, written after the keyword **require**. Processor *p* cannot execute the activation feature until the wait condition is satisfied, i.e. the query *actuator.is_ready* is true.

Once the wait condition is satisfied, processor *p* can start with the execution of the routine body. Figure 4 illustrates the interaction between the processors. In the first step, *p* calls *sensor.value* to get the value of the sensor handled by *q*. Processor *p* waits automatically until *q* returns the result,

as subsequent instructions in the activation feature depend on the result. This synchronization scheme is called *wait by necessity*.

In the second step, *p* makes a call to start the actuator, hence sending a request to *r*. However, *p* does not wait until *r* executed the request because there is no result – the feature is evaluated asynchronously, possibly involving a call to the mechanism. Lastly, *p* asks all locked handlers, here, *q* and *r*, to unlock as soon as they are done with the execution of the requests *sensor.value* and *actuator.start* (*value*) that have been added during the execution of the feature *activate*.

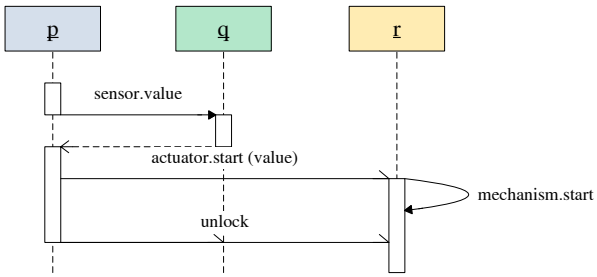

Fig. 4. SCOOP example interaction

V. Hexapod: Implementation

The SCOOP mechanisms serve as the basis for the implementation of the control program for a walking hexapod robot. SCOOP makes it possible to write code that closely reflects the hexapod specification presented in Section III. We review the hardware setup first.

A. Robot hardware

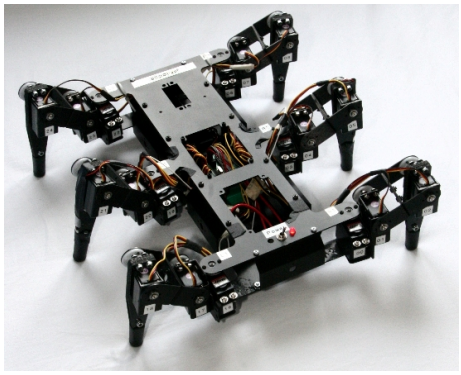

Fig. 5. The hexapod robot

Figure 5 shows the complete robot. Each leg of the robot consists of three servo motors and represents the three key segments: coxa, femur and tibia. Servo motors have a self-contained control loop mechanism ensuring that it achieves and maintains the desired position. Hence, the control program only needs to provide the desired position to the servo motor.

To determine whether the leg is planted on the ground, the feet of the two middle legs are equipped with a force sensor. Angle sensors provide confirmation that a leg has reached the AEP / PEP. For this, the sensors are installed next to the coxa servo motors of the middle legs.

The control program runs on a PC host, transmits commands, and receives sensor values over a wireless link from the hexapod.

B. Hexapod control in SCOOP

The biological hexapod described in Section III-C contains a pattern generator for each leg. The pattern generator produces a periodic signal for the protraction-retraction cycle. The sensory ganglions inhibit or excite actions by taking inputs from the leg's sensory system and also taking inputs from state of neighboring legs. Given the coordination rules, we can now develop an object model that is in almost direct correspondence with the biological equivalent.

1) Concurrent object model: Figure 6 shows typical objects (class instances) in the design of the hexapod.

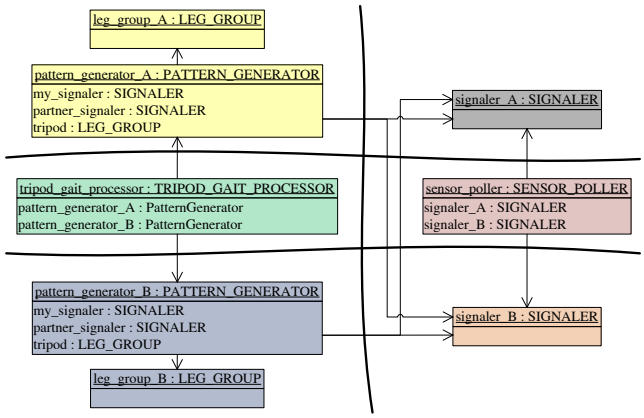

Fig. 6. Design

The *tripod gait processor* has two *pattern generators* for each of the two leg groups and triggers them to walk or stop. Upon receiving the signal, each pattern generator begin its sequence of protraction-retraction cycles. The pattern generator uses the actuation services of the *leg group* to cause leg movements; sensory data provided by the *signaler* tells the pattern generator about the state of the leg. The *sensor poller* is a common service which polls the sensors on the hardware and updates the signalers.

2) Execution: This section shows how the locomotion and the coordination of the two leg groups is implemented, based on the rules described in Section III-B. We highlight the natural translation of these rules to wait conditions.

The pattern generator, like its biological counterpart, triggers protraction and retraction cycles continuously.

```
walk
    do
        checklegs (my_signaler)
        from until stop_requested (my_signaler)
        loop
            begin_protraction (partner_signaler, my_signaler)
            ensure_protraction (my_signaler)
            complete_protraction (partner_signaler)
            execute_retraction (partner_signaler, my_signaler)
```

```
    end
end
```

Protraction is divided into three phases: raising the legs, swinging to the AEP and then finally depressing to the ground. We now examine each of the routines called in the loop above.

```
begin_protraction (partner_signaler, my_signaler: separate SIGNALER)
    require
        my_signaler.legs_retracted
        partner_signaler.legs_down
        not partner_signaler.protraction_pending
    do
        tripod.lift
        my_signaler.set_protraction_pending (true)
    end
```

The first two wait conditions implement the rules (R1) and (R2). The third wait condition is implicitly necessary for the case where both leg groups are retracted and ready to protract. The call to *tripod.lift* is asynchronous and there can be a delay between the call and the moment the legs are physically raised. Without the third wait condition, it would be possible that one leg group executes *begin_protraction* just after the other group finished *begin_protraction*, because the sensors could indicate that both leg groups are on the ground. We create virtual sensor information to indicate when a protraction is in progress and use it to prevent the above situation. Routine *ensure_protraction* is the continuation of the protraction.

```
ensure_protraction (my_signaler: separate SIGNALER)
    require
        my_signaler.legs_up
    do
        my_signaler.set_protraction_pending (false)
        tripod.swing (my_signaler.stride_length, my_signaler.stride_ratio,
                my_signaler.retraction_time)
    end
```

The wait condition ensures that the leg group is raised before we proceed with the swing. This wait condition is not an explicit part of the rules (R1) - (R4), as it is a constraint of the protraction itself. Note that it is not necessity to wait in a busy way until the leg group is lifted. The leg group lifts asynchronously and the pattern generator proceeds automatically as soon as the signaler is locked and the leg group is lifted. After the swing has been executed, the leg group is ready to be dropped, subject to wait conditions expressing rule (R4). This is shown in feature *complete_protraction*.

```
complete_protraction (partner_signaler: separate SIGNALER)
    require
        partner_signaler.legs_retracted
    do
        tripod.drop
    end
```

With this set of operations the leg group has been raised from the PEP and planted on the AEP. Now it is time to retract the leg group with a call to *execute_retraction*.

```
execute_retraction (partner_signaler, my_signaler: separate SIGNALER)
    require
        my_signaler.legs_down
        partner_signaler.legs_up
    do
        tripod.retract (my_signaler.retraction_time)
    end
```

The retraction is guarded by a wait condition that covers rules (R2) and (R3). Now the loop jumps back to a call to the feature *begin_protraction*, whose wait condition *my_signaler .legs_retracted* will trigger waiting until the retraction completed.

VI. Comparison of programming approaches

This section contrasts the SCOOP approach with more traditional approaches: sequential, event-driven, and multi-threaded programming.

A. Sequential programming

It is quite easy to structure the hexapod's tripod gait as a sequence of operations.

```
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true) {
    lead.Raise();
    lag.Retract();
    lead.Swing();
    lead.Drop();

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

The solution defines a lead and a lag group. The lead leg group performs the protraction and the lag leg group performs the retraction. After each iteration the leg groups change their roles. Each of the calls, *Raise*, *Retract*, *Swing* and *Drop*, is synchronous; It returns only when the completion is confirmed by the sensors.

The sequential program is by construction incapable of performing concurrent actions and this results in less efficient walking. Also, when trying to scale the problem to a more robust hexapod with autonomous legs, the code may become more complicated.

B. Event-driven model

The publisher-subscriber model, in which clients register themselves with a provider to receive updates, supports a decoupled system. The subscription request and update callbacks occur over simple interfaces. Some languages even provide a built-in mechanism to support this methodology. For the hexapod leg coordination, each leg subscribes to event information from its neighbors. In the callback handlers, the control is established; upon receiving a neighbor's retraction start event, a leg can start protracting.

Such models, heavily used in middlewares such as MOOS [9] and ROS [11], result in complex control flow due to the interactions of event handlers. Maintaining clear control flow is trivial in SCOOP as sequential reasoning can be applied, up to wait-conditions and wait-by-necessity. Still, these two mechanisms support direct reasoning better than an event-driven approach, where the events are separated from control-flow structures.

C. Multi-threaded programming

The following C# code is an example of a multi-threaded hexapod control program:

```
private object m_protractionLock = new object();

private void ThreadProcWalk(object obj) {
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.
          AbortRequested) {
        lock (m_protractionLock) {
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        leg.Partner.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
```

Here the two threads execute a cycle of protraction and retraction and are synchronized with each other using semaphores. Threads use a shared lock to compete for the protraction privilege. A sensory-polling thread periodically queries the sensors and signals the semaphores.

The multi-threaded approach suffers from an extra level of indirection for conditional synchronization. Conditions are represented by variables that are used to define signals between threads. These condition variables must be manually identified and managed (setting, unsetting, locking, and unlocking). In SCOOP, expressions represent these conditions and waiting emerges directly from predicates on the state.

However, the conveniences of SCOOP are not free. They require a runtime system which can do implicit locking and unlocking, notify objects of changes, and other operations. This overhead means that, in general, the SCOOP model will not generate implementations which are as efficient as their multi-threaded counterparts, making these more suitable in situations where speed requirements are stringent. Also, learning a new language such as SCOOP requires some effort, but can be quickly amortized by the benefits of a structured approach. Within a robotics framework, as outlined in Section VII, several languages may serve different purposes, providing an opportunity for combining approaches.

VII. Towards a framework

We believe that our approach can scale to larger robotics applications, and plan to develop a framework for robotic control on the basis of SCOOP within a three-layer architecture [16]. In this architecture, a control layer implements one or more stateless feedback control loops that couple sensors to actuators. The result is a primitive behavior of the robot. The hexapod control software developed in this paper is entirely located in the control layer. Next, a sequencer layer interacts with the control layer to fulfill a task. The sequencer layer is stateful and selects the primitive behaviors the control layer should use at a given time. A deliberator layer performs time-consuming deliberative computations. It can either produce plans for the sequencer layer or it can respond to specific queries from the sequencer layer.

In a future framework, SCOOP could be used on all layers. Each of the layers will be located on a different set of processors and the framework will ensure inter-layer access. For interoperability reasons, the framework will have interfaces to existing middlewares that can be used on all layers. The framework will be the basis for the development of complex robotics applications using SCOOP. The abstractions of the framework and the modularity of SCOOP programs will be helpful for implementing large robotics applications.

VIII. Conclusion

The SCOOP model of concurrency is a novel approach to the problem of coordinated robotic control. The features of atomicity and wait-conditions allow a style of programming that permits close association of a behavioral specification with the implementation. This is an important result as one of the most difficult challenges in creating software is correctly transitioning from requirements to implementation. In general, lowering the effort required to perform this transition means less time spent on development and also more correct software.

Acknowledgments. This work is part of the SCOOP project at ETH Zurich, which has benefitted from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award) and ETH (ETHIIRA).

References

[1] I. J. Cox and N. H. Gehani, "Concurrent programming and robotics," *Int. J. of Robotics Research*, vol. 8, no. 2, pp. 3–16, 1989.
[2] J.-C. Baillie, "URBI: Towards a universal robotic low-level programming language," in *IROS'05*, 2005.
[3] J. Peterson, G. D. Hager, and P. Hudak, "A language for declarative robotic programming," in *ICRA'99*, 1999, pp. 1144–1151.
[4] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *IROS'98*, vol. 3, Oct 1998, pp. 1931–1937.
[5] B. Thirion and L. Thiry, "Concurrent programming for the control of hexapod walking," *Ada Letters*, vol. XXII, no. 1, pp. 17–28, 2002.
[6] H. Cruse, T. Kindermann, M. Schumm, J. Dean, and J. Schmitz, "Walknet–a biologically inspired network to control six-legged walking," *Neural Networks*, vol. 11, no. 7-8, pp. 1435–1447, 1998.
[7] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *RAM'08*, 2008.
[8] D. Moore, E. Olson, and A. Huang, "Lightweight communications and marshalling for low-latency interprocess communication," Computer Science and Artificial Intelligence Laboratory, MIT, Tech. Rep., 2003.
[9] P. M. Newman, "MOOS: Mission orientated operating suite," Department of Ocean Engineering, MIT, Tech. Rep., 2008.
[10] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit," in *IROS'03*, 2003, pp. 2436–2441.
[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.
[12] V. Dürr, J. Schmitz, and H. Cruse, "Behaviour-based modelling of hexapod locomotion: linking biology and technical application," *Arthropod Structure & Development*, vol. 33, no. 3, pp. 237–250, 2004.
[13] N. Porcino, "Hexapod gait control by a neural network," in *IJCNN'90*, 1990, pp. 189–194.
[14] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.
[15] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.
[16] E. Gat, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*, 1997, pp. 195–210.