Simple Concurrency for Robotics with the Roboscoop Framework

Andrey Rusakov Jiwon Shin Bertrand Meyer
Chair of Software Engineering
Department of Computer Science
ETH Zürich, Switzerland
{andrey.rusakov, jiwon.shin, bertrand.meyer}@inf.ethz.ch

Abstract— **Concurrency is inherent to robots, and using concurrency in robotics can greatly enhance performance of the robotics applications. So far, however, the use of concurrency in robotics has been limited and cumbersome. This paper presents *Roboscoop*, a new robotics framework based on Simple Concurrent Object Oriented Programming (SCOOP). SCOOP excludes data races by construction, thereby eliminating a major class of concurrent programming errors. Roboscoop utilizes SCOOP's concurrency and synchronization mechanisms for coordination in robotics applications. We demonstrate Roboscoop's simplicity by comparing Roboscoop to existing middlewares and evaluate Roboscoop's usability by employing it in education.**

I. INTRODUCTION

Advanced robotic systems are composed of many components that can operate concurrently. Running these components concurrently would enable robots to meet their full potential. Researchers have noticed the value of concurrency in robotics three decades ago. Kanayama [1] proposed a simple message passing mechanism to synchronize a robot's multiprocessing units, and Ingemar and Gehani [2] showed how Concurrent C, a general purposed language, can be applied in robotics. Despite the early effort, little progress has been made in using concurrency in robotics.

Introducing concurrency to robotics poses a great challenge. Enabling concurrency in a robotic system requires its software to support concurrent execution. Traditional concurrent programming techniques such as standard "threading libraries" do not, however, offer any safety guarantees. Hence, the programmer must write concurrent programs carefully to avoid common pitfalls of concurrency such as data races and deadlocks. Consequently, most robotic software make only elementary use of concurrency or avoid it all together.

This paper introduces *Roboscoop*, a new robotics framework based on *Simple Concurrent Object Oriented Programming* (SCOOP). Unlike standard approaches that are complex and error-prone and require additional qualifications, SCOOP makes concurrency simple and safe. SCOOP is free of data races by construction. Built on top of SCOOP, Roboscoop enables programmers to express robot's behaviors in a natural way without worrying about the program's safety. In addition to SCOOP's concurrency and synchronization mechanisms, Roboscoop provides a library support for robotics, *Behavior-Signaler-Controller* design for coordination of tasks, and infrastructure for integration with external frameworks. We demonstrate Roboscoop's simplicity by comparing it to existing middlewares and present Roboscoop in education that enables even novice programmers to program concurrent robotic software. Current Roboscoop contains behaviors and tools necessary for the operation of differential drive robots and has been tested on two different robots.

This paper is organized as follows: After presenting related work in Section II, the paper presents the core of Roboscoop framework in Section III. Section IV presents an example of using Roboscoop for the task of exploring an unknown area. Section V compares Roboscoop against other middlewares and also presents an evaluation of Roboscoop in education. The paper concludes with final remarks in Section VI.

II. RELATED WORK

In the last decade, many middlewares have been proposed to ease the development of robotic coordination and control. Some of the more popular middlewares include Urbi [3], MOOS [4], Microsoft Robotics Developer Studio [5], ROS [6], and LCM [7]. Two recent surveys [8], [9] provide a comprehensive overview and comparison of middlewares. In addition, the related work section of MIRA [10] provides a compact survey of middlewares with the largest impact. Here, we highlight how they compare to Roboscoop. Section V-A presents a more-detailed comparison.

Mission Orientated Operating Suite (MOOS) uses store/fetch mechanism for message-passing. Robot Operating System (ROS) uses a combination of publish/subscribe and service-based message-passing models. The Lightweight Communications and Marshalling (LCM) utilizes only a publish/subscribe message-passing model and additional marshalling tools to support low-latency applications. MIRA, a recently introduced robotics middleware with a focus on performance, also uses publish/subscribe message-passing model and supports multi-threaded accesses through slots. While all these middlewares concentrate on the inter-process communication, Roboscoop does coordination and concurrency on the level of a single application. MOOS, ROS, and LCM applications can be written in general-purpose languages such as C++ and Java, and their application support for concurrency is limited to the standard "threading library" approach. On the contrary, Roboscoop provides the language support for concurrency.

ROS SMACH [11] is a ROS-independent Python library for building hierarchical state machines for specifying models of complex robotic behavior. SMACH supports parallel execution of state behaviors, but its applicability is limited to scenarios where all possible states and state transitions can be described explicitly. Concurrency in SMACH is bound to the language support, more precisely, to Python threads. Roboscoop's concurrency is based on SCOOP that offers simpler and safer concurrency than Python threads.

Microsoft Robotics Developer Studio (MRDS) is a .NET-based robotics programming environment that uses the Concurrency and Coordination Runtime library for parallelism. Based on .NET, the library manages asynchronous parallel tasks through message-passing. Unlike MRDS that introduces *work items*, Roboscoop does not have to introduce any additional intermediate abstractions for concurrency. In Roboscoop, concurrency can be reached on the object level.

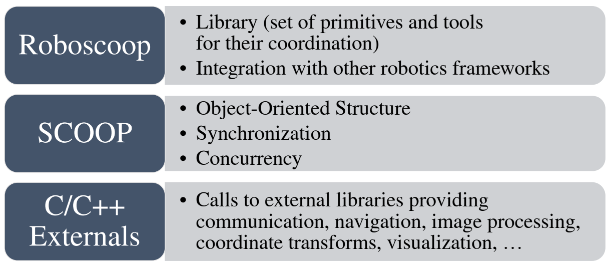
Urbi is a software platform for robotics, which along with the C++ support brings urbiscript [12] - a parallel event-based object-oriented script language. Urbiscript and SCOOP go in the same direction by providing tools for coordination and concurrency on the language level. Their difference lies in their complexity. For concurrency, urbiscript introduces a set of unmatched additional mechanisms and syntactic extensions to the language while SCOOP introduces only one additional keyword.

In addition to middlewares, some languages have also been proposed for robotics. Concurrent C [2] provides concurrency on top of a general-purpose language and has been proposed for usage in robotics. SCOOP's advantage over non-object-oriented concurrency languages such as Concurrent C is its ability of reasoning about different robot components as programming objects. This results in more modular design and reusable applications. As a general-purpose language, SCOOP provides more flexibility than domain-specific languages. For instance, Task Description Language [13], based on the task tree data structure, has nodes that can contain only commands, goals, monitors, or exceptions. SCOOP used in Roboscoop retains its full flexibility of a general-purpose language.

The Roboscoop framework builds on top of the work by Ramanathan et al. [14]. The authors presented a controller for hexapod using SCOOP. Roboscoop is an extension of SCOOP's concurrent coordination into a full-fledged robotics framework. In addition to SCOOP, it provides a library support for building up robotic behaviors, *Behavior-Signaler-Controller* design for task coordination, and communication infrastructure for integration with the external frameworks.

III. ROBOSCOOP FRAMEWORK

The Roboscoop framework is a set of classes and programming tools that aims to ease the development of robotics applications. The framework is composed of three parts – Roboscoop, SCOOP, and C/C++ externals – as shown in Figure 1. Roboscoop contains a library that provides a set of primitives and tools for coordination of different robotic behaviors. SCOOP is the base language of Roboscoop and gives Roboscoop the power of concurrency and synchronization. C/C++ externals enables Roboscoop users to integrate existing libraries in robotics. This section describes the three components of Roboscoop in detail.



Roboscoop
• Library (set of primitives and tools for their coordination)
• Integration with other robotics frameworks
SCOOP
• Object-Oriented Structure
• Synchronization
• Concurrency
C/C++ Externals
• Calls to external libraries providing communication, navigation, image processing, coordinate transforms, visualization, …

Fig. 1: The Roboscoop framework structure

A. *SCOOP – Simple Concurrent Object Oriented Programming*

SCOOP model [15], [16] is an extension of a standard sequential object-oriented model, *Eiffel*, and the Design by Contract paradigm [17] to support concurrency. The model provides a higher level of abstraction through object-orientation and minimizes common concurrency-specific mistakes. Programmers using SCOOP can represent, for instance, various parts of a robot as objects and manipulate them concurrently without the burden of threads. SCOOP eliminates data races by construction.

Every object in SCOOP is associated with a certain *processor* called its *handler*, for the object's entire lifetime. Contrary to CPU, SCOOP *processor* is just an abstract independent unit of control and is typically implemented as a thread. Only the object's *handler* is allowed to execute features, i.e, a set of possible actions, on the object. Each object has exactly one *handler*, but a single *processor* can handle multiple objects. When two objects are handled by different *processors*, they are called **separate** to each other. **separate** objects can operate concurrently.

Because features of a **separate** object can only be executed by its own handler, it may seem that the *handler* is not allowed to execute features on other **separate** objects. But if in the scope of the current object (*client*), there is a need to call some feature on a **separate** object (*supplier*), the *client* can ask the *supplier* to execute it on its behalf. Such a call is named a *separate call* and it can be executed asynchronously.

An object can be *separate* or *non-separate*. To distinguish between the two semantics in SCOOP, the **separate** keyword is used in object's type declaration. If the declared object has a **separate** type, it indicates that it may reside on a different *processor* relative to the current object. **separate** type is also used for arguments in a feature's signature to denote the intention of a *separate call*.

The following example illustrates how SCOOP can coordinate robotic eyes of a humanoid to look at a moving ball. We assume that the robot is equipped with a camera that provides the ball's location. In the example, *left_eye* , *right_eye* , *ball* and the current object are **separate** objects and hence have different *handlers*:

```
look ( left_eye , right_eye : separate EYE; ball : separate BALL)
  require
    ball . is_visible
    left_eye . is_ready and right_eye . is_ready
```

```
local
   position : POINT_3D
do
   position := ball . position
   left_eye .move (position )
   right_eye .move (position )
end
```

The above code contains several Eiffel keywords – **require**, **local**, **do**, and **end** – and a SCOOP keyword, **separate**. The **require** clause contains the *precondition*, conditions that must be satisfied before the body of *look* (in **do** clause) can be executed. In the above example, the call will not be executed until *ball* is visible and both *left_eye* and *right_eye* are ready to be actuated. To ensure this, SCOOP runtime waits until this condition is **True**; this synchronization mechanism is known as *wait conditions*.

Another synchronization mechanism, named *wait by necessity*, helps the synchronization of execution inside the body of a routine. When the current object requires the result of a *separate call* for further computation, its *handler* automatically waits at the point where the *separate call* returns the result; need for the results necessitates the waiting. A similar idea was also recently introduced in other modern programming languages including C++ and Java with the notion of *futures*. Unlike C++ and Java, SCOOP does not require programmers to introduce any additional constructions or auxiliary objects to support this type of synchronization. In the example, *wait by necessity* forces the execution to wait for the result of *ball . position* to be stored in the local variable *position* before proceeding. Once the value is stored in *position*, calls of the *move* feature can be executed asynchronously on both *left_eye* and *right_eye*, causing the eyes move simultaneously.

SCOOP's concurrency mechanism, *separate calls*, and synchronization mechanisms, *wait conditions* and *wait by necessity*, allows programmers use concurrency to express complex robotic behaviors in a natural way, without the pain.

B. Roboscoop

The core of Roboscoop is the Roboscoop library. Built on top of SCOOP, the library uses SCOOP synchronization mechanisms to enable concurrency and coordination in robotic applications. Using the Roboscoop library, programmers can create various robot behaviors easily and coordinate these behaviors in a natural way. The Robscoop library achieves concurrency and coordination as follows:

Concurrency

Concurrency in Roboscoop is simple and fine-grained. Writing concurrent programs in Roboscoop does not require programmers to create or manage threads and synchronization primitives such as mutexes or semaphores. Declaring **separate** types and applying aforesaid synchronization mechanisms are sufficient to make code concurrent, enabling programmers to concentrate on the robotics issues instead. In addition, SCOOP model enables Roboscoop to achieve concurrency on the object level. In turn, programmers can create fine-grained concurrent applications. The model also shields programmers from data races providing an exclusive access to the shared resources.

Coordination

Roboscoop coordinates robotic behaviors through the interaction among *behaviors*, *signalers*, and *controllers*. Behavior defines tasks a robot performs. Signaler contains information about the state of particular subsystems, e.g., robots, sensors, actuators, behaviors, etc. Controller controls actuators based on the behavior and the state of sensors. The *Behavior-Signaler-Controller* interaction is crucial to the coordination of robotic behaviors in Roboscoop.

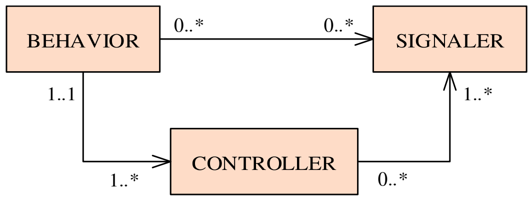


Fig. 2: The relationship among behavior, signaler, and controller. A behavior is associated with one or more controllers and zero or more signalers. A controller, in turn, is associated with one or more signalers.

Figure 2 shows the interaction among the three components graphically. *Behavior* handles high-level sophisticated robotic tasks. As a single programming unit, behavior coordinates multiple simultaneously-operating controllers to perform a particular task. *Signaler* contains information about the current state of other components such as sensor data or behavior's state. When a signaler is used in the precondition of a feature, it enables the feature to synchronize the execution when the desired conditions are satisfied. *Controller* implements low-level coordination tasks and is in charge of controlling actuators based on sensor data and state of the algorithm. Coordination in Roboscoop works by behavior objects creating, launching, and orchestrating their controller objects when the signalers are in desired states. Section IV presents a concrete example that illustrates the *Behavior-Signaler-Controller* interaction.

Structure

The Roboscoop library is organized into *subclusters* – sets of classes. Each subcluster is responsible for one type of functionality. The most important subsclusters are as follows:

- SIGNALER: Signaler subcluster contains commonly used Signaler classes. Exemplary classes in the subcluster include *STOP_SIGNALER* that indicates whether or not some particular task was requested for a stop, and *ODOMETRY_SIGNALER* that contains odometry-related information, i.e., position and orientation in space and current speed and speed-related features.
- BEHAVIOR: Behavior subcluster contains several simple Behaviors for differential drive robots. Exemplary classes in the subcluster include *WANDER_BEHAVIOR* and *TANGENT_BUG_BEHAVIOR*, an implementation of the tangent bug algorithm [18].

(a) SmartWalker – the robot for ambient assisted living (AAL) (b) Educational robot Thymio-II with PrimeSense RGBD sensor

Fig. 3: Roboscoop demonstrators

- CONTROLLER: Controller subcluster contains controllers for the behavior subcluster. Basic controllers such as *PID_CONTROLLER* are collected here.
- COMMUNICATION: Communication subcluster provides resources for communication with robots. Classes in this subcluster send commands to robots and receive necessary data back. This subcluster also contains functionality for Roboscoop to integrate with external frameworks.
- UTILS: Utils subcluster contains useful tools for creating non-standard behaviors such as non-linear behavior or event-based behavior. For instance, the timing tools in the subcluster enable delayed and repeated invocation.

C. *Interoperability*

The underlying Eiffel's support for C/C++ *external calls* eases the integration of external libraries and other frameworks into Roboscoop. Programmers using Roboscoop can use functionality of existing external C/C++ solutions such as image processing libraries without losing the benefits of Roboscoop's easy concurrency and coordination; applying *external calls* on a separate object can easily make it parallel.

Current version of Roboscoop is integrated with ROS and can communicate both ways (publishing and subscribing) with ROS-based applications. Any robot with ROS interface can therefore be programmed using Roboscoop.

The Roboscoop framework has already been used in a real-life setup for two different robots (see Figure 3): in ambient assisted living (AAL) and in education.

IV. EXAMPLE

This section demonstrates how to develop robotic applications with Roboscoop. We assume that our robot is a differential drive robot equipped with forward and ground sensors to detect obstacles in the front and distance to the ground, respectively. The task is to explore an unknown area. To accomplish this task, our robot must go straight when there is no obstacle in front, turn when there is an obstacle, and stop when there is a hole in the ground. We implement the task of exploring an unknown area in class *EXPLORATION_BEHAVIOR*.

The *EXPLORATION_BEHAVIOR* class contains three separate *SIMPLE_CONTROLLER* objects, each object responsible for one type of control – going straight, turning to avoid obstacles, or stopping. It launches the controllers asynchronously to achieve the desired behavior.

```
class EXPLORATION_BEHAVIOR feature
   ctrl_a , ctrl_b , ctrl_c : separate SIMPLE_CONTROLLER

   start
  do
     launch ( ctrl_a , ctrl_b , ctrl_c )
  end

  launch (a, b, c: separate SIMPLE_CONTROLLER)
  do
     a. go_straight
     b. turn_to_avoid_obstacles
     c.stop_on_emergency
  end
```

The *SIMPLE_CONTROLLER* class has references to the objects of the **separate** types *ODOMETRY_SIGNALER* , *RANGE_SIGNALER* and *ROBOT* and use them as arguments for the separate calls. *SIMPLE_CONTROLLER* also implements *go_straight* , *turn_to_avoid_obstacles* , and *stop_on_emergency* features. These features continuously call *go*, *turn* and *stop* with aforementioned separate arguments.

Figure 4 shows the object diagram of our example. All objects are **separate** and thus can execute calls asynchronously. All three *SIMPLE_CONTROLLER* objects can access the objects of *ODOMETRY_SIGNALER*, *RANGE_SIGNALER*, and *ROBOT* through *separate calls*.

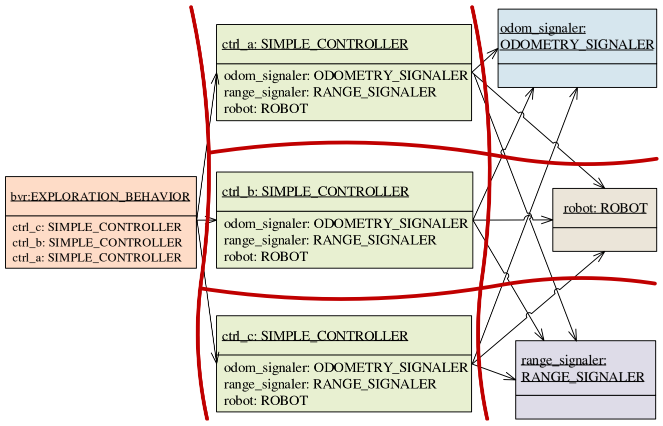

ctrl_a: SIMPLE_CONTROLLER
odom_signaler: ODOMETRY_SIGNALER
range_signaler: RANGE_SIGNALER
robot: ROBOT
ctrl_b: SIMPLE_CONTROLLER
odom_signaler: ODOMETRY_SIGNALER
range_signaler: RANGE_SIGNALER
robot: ROBOT
ctrl_c: SIMPLE_CONTROLLER
odom_signaler: ODOMETRY_SIGNALER
range_signaler: RANGE_SIGNALER
robot: ROBOT
bvr:EXPLORATION_BEHAVIOR
ctrl_c: SIMPLE_CONTROLLER
ctrl_b: SIMPLE_CONTROLLER
ctrl_a: SIMPLE_CONTROLLER
odom_signaler: ODOMETRY_SIGNALER
robot: ROBOT
range_signaler: RANGE_SIGNALER

Fig. 4: Each object in the diagram is separate. Thick curvy lines show the borders between SCOOP processors. The behavior object (on the left) coordinates three controller objects (in the middle). Each of the controllers uses information from the signalers for actuating the robot (on the right).

The *go_straight* feature makes the robot move forward at a constant speed if 1) the robot is not moving, 2) the ground has no hole, and 3) there is no obstacle in front of the robot. The three requirements are easily translated into precondition in the feature *go* as shown below:

```
go (odom_sig: separate ODOMETRY_SIGNALER;
      range_sig : separate RANGE_SIGNALER; robot: separate ROBOT)
  require
     not odom_sig.is_moving            −− 1)
     not range_sig .has_ground_hole    −− 2)
     not range_sig . has_obstacle      −− 3)
  do
     robot . apply_default_speed
  end
```

The *turn_to_avoid_obstacles* feature makes the robot turn if 1) the robot is moving, 2) the ground has no hole, and 3) the robot detects an obstacle in front. As before, the requirements are translated into preconditions naturally:

```
turn  (odom_sig: separate ODOMETRY_SIGNALER;
       range_sig: separate RANGE_SIGNALER; robot: separate ROBOT)
  require
    odom_sig.is_moving                  -- 1)
    not range_sig.has_ground_hole       -- 2)
    range_sig.has_obstacle              -- 3)
  do
    if range_sig.is_obstacle_left then
      robot.turn_right
    elseif range_sig.is_obstacle_right then
      robot.turn_left
    end
  end
```

The implementation of the *stop* feature is also simple and natural. The robot must stop if 1) it is moving, and 2) the ground has a hole.

```
stop  (odom_sig: separate ODOMETRY_SIGNALER;
       range_sig: separate RANGE_SIGNALER; robot: separate ROBOT)
  require
    odom_sig.is_moving               -- 1)
    range_sig.has_ground_hole        -- 2)
  do
    robot.stop
  end
end
```

The code snippets demonstrate how roboscoop ensures a close correspondence between the behavioral requirements and the SCOOP preconditions. It eases the translation of desired behavior into code and simplifies the coordination.

Canceling approach

The *EXPLORATION_BEHAVIOR* can coordinate different possible control algorithms for autonomy, but it is sometimes necessary to interact with the robot directly. One such a situation is interruption/cancellation of an ongoing process. The *STOP_SIGNALER* class fulfills just the role. Programmers can easily apply the cooperative cancellation mechanism using a **separate** object of type *STOP_SIGNALER*.

To be interruptible, the controller must be aware of the state of *stop_signaler*. This can be achieved by using the precondition as shown below:

```
stop_signaler: separate STOP_SIGNALER

start_cancellable_control
  do
    from until stop_requested (stop_signaler) loop
      control (example_signaler, stop_signaler)
    end
  end

control  (b: separate SOME_SIGNALER; s: separate STOP_SIGNALER)
  require
    b.is_ready or s.is_stop_requested
  do
    if (not s.is_stop_requested) then
      robot.execute_reactive_control
    else
      robot.stop_smoothly
    end
  end

stop  (s: separate STOP_SIGNALER)
  do
    s.set_stop_requested (True)
  end
```

The *control* feature waits until the robot is ready to be controlled or stop is requested. If no stop is requested and the robot is ready, it performs its usual control. Otherwise, if stop is requested, it stops the robot smoothly. After starting the cancellable control, all we need to do to cancel the execution is to call *stop* feature with *stop_signaler* as an argument. This approach of stopping an execution can also be used to handle other types of interruption such as user-interaction or a timer.

V. EVALUATION

The main features of Roboscoop are its simplicity and usability. We demonstrate the framework's simplicity and usability by comparing it with other approaches and reporting students' experience with Roboscoop in educational context.

A. Comparison to other approaches

We compare SCOOP, Roboscoop's base language, with C++ with *boost* library [19] and urbiscript [12]. Boost is a popular C++ library with concurrency support and is extensively used in many robotics middlewares including ROS. urbiscript is the base language for Urbi, a robotics framework with concurrency support. The comparison of these languages is on mechanisms or syntactic support for parallelization and coordination using events.

First comparison is on parallelization. Let us consider a task of commanding a robot to lift its arm and bending its leg simultaneously. A program that accomplish such a task can be written in C++ with boost as follows:

```
boost::thread a_thread, l_thread;
a_thread = boost::thread(&Arm::lift, &arm);
l_thread = boost::thread(&Leg::bend, &leg);
```

In this example, *lift* method of the class *Arm* and *bend* method of the class *Leg* will be executed simultaneously, each by a separate thread. The problem is that the thread objects have to be managed manually. To facilitate this issue, recently *async* syntax was announced in C++11. *async* constructions automatically create and spawn threads allowing tasks be executed asynchronously:

```
auto a = std::async(launch::async, &Arm::lift, &arm);
auto b = std::async(launch::async, &Leg::bend, &leg);
```

Unfortunately, neither C++ cases guarantee that the execution will not be interrupted by the other threads accessing the same *arm* or *leg* objects. The only way to prevent the data race is by using, for example, mutexes.

urbiscript uses special connectors, "," and "&", for concurrency in addition to the standard sequential ";" connector. The "," connector launches the first statement in background, and immediately proceeds to executing the next statements. The "&" connector is similar to ",", but it "waits" for all the statements to be known to start their concurrent execution. The same program of lifting a robotic arm and bending a robotic leg can be written in urbiscript as follows:

```
{
  {arm. lift },
  {leg .bend},
};
```

urbiscript's code is much simpler than C++ one, but the above syntax is only valid for behaviors composed only of special connectors. For instance, dynamically-created jobs cannot be composed only of special connectors, and for such behaviors, urbiscript requires the *detach* function, which serves a similar role to C++ threads. The *detach* function takes a block of code that will be run in its own thread of execution and returns a handle to the job. In addition, just as in C++, the above code gives no guarantee against data races. To provide mutual exclusion in urbiscript, Mutex objects must be used. Mutex in urbiscript is a particular use case of the Tags mechanism, where statements and code blocks can be tagged and then manipulated by the corresponding Tag objects. Tags feature urbiscript with ability to interrupt execution of the tagged code safely. Tags also introduce additional useful events which arise every time when entering and leaving tagged sections. Mutex objects exploit these events to turn tagged code blocks into the critical sections. The following urbiscript snippet ensures mutual exclusion for the shared resources by tagging particular pieces of code to Mutex objects.

```
var control_arm = Mutex.new;
var control_leg = Mutex.new;
{
  control_arm: {arm. lift },
  control_leg : {leg .bend},
};
```

In SCOOP, **separate** objects and *separate calls* enable concurrent execution. In addition, SCOOP is free of data races by construction. As a result, SCOOP's code for the same task of lifting a robotic arm and bending a robotic leg is simple as urbiscript's but with a guarantee of no data race:

```
move_limbs (arm: separate ARM; leg: separate LEG)
  do
    arm. lift
    leg .bend
  end
```

The "wrapper" routine, *move_limbs*, reflects a programmer's intention to run *arm* and *leg* concurrently. SCOOP guarantees the "wrapper" an exclusive control over its **separate** arguments. This guarantee prevents the execution from data races, which means there is no need to use mutexes in SCOOP unlike in C++ or Urbi. If *separate calls* require synchronization, *wait by necessity* mechanism described in Section III-A can be used.

More complex and interactive robotic behaviors require the robot to react to various events. In this second comparison, we discuss how the three languages support coordination using events as an example. In event-driven programming style or paradigm, external actions, i.e., events, determine the execution flow, where an event can be a sensor value or a message from other threads among others. When an event occurs, associated callback functions or event handler functions are triggered. This process repeats for every event.

In C++, a handler can be subscribed for a particular *signal* (event) using the function *connect*. After the signal is emitted, the handler is automatically called. For example, a program that commands a robot to go to the charging station when its battery is low can be written in C++ as follows:

```
int main () {
 signal<void ()> lowChargeSignal;
 lowChargeSignal.connect(bind(&Robot::goToChargingStation, &robot));
 while (chargeLevel >= 0.2) { sleep(1); }
 lowChargeSignal();
}
```

As the example code illustrates, the C++'s interface for handling events is cumbersome. Moreover, it requires additional constructions if events are represented as conditions.

urbiscript supports events in two ways: event objects and "conditional" events. urbiscript's mechanism for creating an event object for explicit event-handling is similar to the C++. The process requires three steps: defining an event, specifying its handlers, and emitting the event. In urbiscript, *Event* class serves this role; in SCOOP, *EVENT* class does the same.

To handle "conditional" events, urbiscript introduces **at** construct. SCOOP, on the other hand, utilizes its built-in structure. SCOOP's *wait condition* expresses the necessity for the feature application to wait to execute until the precondition clause (conditional expression) holds. The task of a robot going to its charging station when its battery is low can be written in urbiscript as follows:

```
at (chargingLevel < 0.2) {
    robot .goToChargingStation;
  }
```

In SCOOP, the same task can be implemented elegantly inside the **require** clause. As a SCOOP's synchronization mechanism, *wait condition* ensures that the robot goes to the charging station only when its battery is low:

```
go_charging (b: separate BATTERY_SIGNALER)
  require
    b. charging_level < 0.2
  do
    robot. go_to_charging_station
  end
```

As we have illustrated, C++ mechanisms for concurrency and coordination leave a lot of low-level flexibility for programmers, therefore require to do many things by hands. As the complexity of robotic systems grows, following every detail of threads' interaction will become insurmountable. urbiscript and SCOOP overcome this challenge with their built-in support for concurrency.

The key difference between urbiscript and SCOOP is their complexity. To introduce concurrency, urbiscript requires a set of unmatched additional mechanisms and syntactic extensions to the language. SCOOP handles concurrency with only one additional keyword: **separate**. It retains natural modes of reasoning about programs where correctness conditions for sequential applications become wait conditions in case of concurrency. SCOOP provides a more solid and coherent solution because the programming model answers both concurrency and coordination issues. This, in turn, makes

Roboscoop a better environment for design, development, and maintenance of robotic applications.

B. Roboscoop in education

Roboscoop's simplicity and ease-of-use make it a great framework to teach software engineering and robotics. To evaluate its potential in education, we used the framework in our multidisciplinary, master's-level robotics programming course. Our students used the Roboscoop framework to coordinate different behaviors of Thymio II [20], an education robot with a differential drive (shown in Figure 3b). The course had 11 students – six ME students, four CS students, and one EE student. Mechanical engineering students had limited programming experience, knowing only one language (C, Java or Matlab) and having programmed no more than class assignments. Out of 11, only three students had prior exposure to Eiffel, the base language of SCOOP. Despite the limited experience of the students, most students managed to learn and program in Roboscoop after a single assignment, equivalent to roughly 40 hours of work. With the standard "thread libraries", only the most experienced students of computer science could achieve similar level of concurrent programming in the same time frame.

At the end of the semester, we asked the students to comment on their experience with Roboscoop. A majority of the students reported the framework is intuitive to understand. Only four students found the framework unintuitive, but three of the four stated that the difficulty stemmed from their inexperience with object-oriented programming. Once they understood the concepts of object-oriented programming, they did not face much difficulty understanding Roboscoop. The fourth student reported that he found Roboscoop difficult to use but did not elaborate further. While the evaluation is based on a small group of motivated students, we believe that it is indicative of the Roboscoop framework's simplicity and ease-of-use. To enhance our understanding of the framework's strengths and weaknesses, we plan on conducting a broader study with more users.

VI. CONCLUSION

This paper presented the *Roboscoop* framework and demonstrated how its underlying SCOOP programming model allows for easy and natural use of concurrency in robotics. The paper proved that Roboscoop is a suitable solution for solving coordination issues in robotics and that SCOOP, as a concurrency model, can scale up to the size of a framework. A direct comparison to other middleware shows that developing concurrent applications for robotics with Roboscoop is easier and more natural. This claim is supported by our evaluation of the framework in education, where even novice programmers could use concurrency without much effort.

Current Roboscoop supports differential drive robots and contains behaviors and tools necessary for the operation of these robots. Although performance, an important goal of concurrency models, was not the main focus of this work, Roboscoop showed itself suitable for two different differential-drive robots, an education robot and a robotic rollator. Future development of Roboscoop includes support for more types of robots and the library extension to support more commonly-used behaviors and tools. We plan to continue evaluating future versions of Roboscoop both in education and in research.

Acknowledgments

We thank Sebastian Nanz, Benjamin Morandi and Alexey Kolesnichenko for discussions on the Roboscoop architecture. This work was partially supported by the Hasler Foundation through the Roboscoop project and by the European Research Council under the European Union's Seventh Framework Programme (ERC Grant agreement no. 291389).

REFERENCES

[1] Y. Kanayama, "Concurrent programming of intelligent robots," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, vol. 2, 1983, pp. 835–838.

[2] I. J. Cox and N. Gehani, "Concurrent C and robotics," in *1987 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, Mar 1987, pp. 1463–1468.

[3] J.-C. Baillie, "URBI: towards a universal robotic low-level programming language," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005, pp. 820–825.

[4] P. Newman, "Moos - mission orientated operating suite," Department of Ocean Engineering, MIT, Tech. Rep., 2006.

[5] J. Jackson, "Microsoft robotics studio: A technical introduction," *IEEE Robotics Automation Magazine*, vol. 14, no. 4, pp. 82–87, Dec 2007.

[6] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *IROS*, 2009.

[7] A. Huang, E. Olson, and D. Moore, "LCM: Lightweight communications and marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2010, pp. 4057–4062.

[8] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, Feb 2007.

[9] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, Jan 2012.

[10] E. Einhorn, T. Langner, R. Stricker, C. Martin, and H.-M. Gross, "Mira - middleware for robotic applications," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[11] J. Bohren and S. Cousins, "The SMACH high-level executive [ros news]," *Robotics Automation Magazine, IEEE*, vol. 17, no. 4, pp. 18–20, Dec 2010.

[12] J.-C. Baillie, A. Demaille, M. Nottale, and Q. Hocquet, "Tag: Job control in urbiscript," in *5th National Conference on Control Architecture of Robots*, 2010.

[13] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *in Proceedings of the Conference on Intelligent Robots and Systems (IROS*, 1998.

[14] G. Ramanathan, B. Morandi, S. West, S. Nanz, and B. Meyer, "Deriving concurrent control software from behavioral specifications," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[15] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[16] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.

[17] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[18] I. Kamon, E. Rimon, and E. Rivlin, "Tangentbug: A range-sensor-based navigation algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, 1998.

[19] [Online]. Available: http://www.boost.org/

[20] [Online]. Available: http://www.thymio.org/