Concurrency Patterns for Easier Robotic Coordination

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

***Abstract*— Software design patterns are reusable solutions to commonly occurring problems in software development. Growing complexity of robotics software increases the importance of applying proper software engineering principles and methods such as design patterns to robotics. Concurrency design patterns are particularly interesting to robotics because robots often have many components that can operate in parallel. However, there has not yet been any established set of reusable concurrency design patterns for robotics. For this purpose, we choose six known concurrency patterns – *Future*, *Periodic timer*, *Invoke later*, *Active object*, *Cooperative cancellation*, and *Guarded suspension*. We demonstrate how these patterns could be used for solving common robotic coordination tasks. We also discuss advantages and disadvantages of the patterns and how existing robotics programming frameworks can support them.**

I. INTRODUCTION

In the early days of computer science, people focused on developing algorithms. As software became more complex, it became necessary to develop systematic methods for handling software, giving birth to software engineering. Software side of robotics is walking the same path. In the early days of robotics, software aspect focused mainly on algorithmic development; however, as robots and their software become more complex, applying good software engineering techniques to robotics software gains more importance. In particular, the lessons, such as design patterns, that people learned in software engineering can also benefit robotics.

Software design patterns are reusable solutions to commonly occurring problems in software development. Concurrency patterns, which deal with the multi-threaded programming paradigm, are particularly interesting to robotics because robots often have many components that can operate concurrently. Running these components concurrently would enable robots to meet their full potential, but concurrent programming requires additional knowledge from programmers and thus restricts the usage of concurrency in robotics for non-specialists. Applying concurrency patterns would ease the development of concurrent robotics software as they provide templates for solving various problems using concurrency even for novice programmers.

This paper provides an overview of six existing concurrency patterns that are useful solutions for common robotic coordination tasks. The patterns – *Future*, *Periodic timer*, *Invoke later*, *Active object*, *Cooperative cancellation*, and *Guarded suspension* – are chosen for their concurrent nature, applicability to robotic coordination, and evidence of use in existing robotics frameworks. The paper aims to help programmers who are not yet experts in concurrency to identify and then to apply one of the common concurrency-based solutions for their applications on robotic coordination.

The rest of the paper is organized as follows: After presenting related work in Section II, it proceeds with a general description of concurrency approach for robotics in Section III. Then the paper presents and describes in detail a number of concurrency patterns related to robotic coordination in Section IV. Section V discusses advantages and disadvantages of using those patterns for robotic tasks. The paper concludes with final remarks in Section VI.

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 no. 291389).

II. RELATED WORK

Proper application of software engineering principles and methods would help robotics programmers bridge the gap between the growing complexity of robotics software and existing robotics approaches. Developing useful robotics libraries [1], [2] and frameworks [3], [4] is one way of sharing and accumulating robotics knowledge. The idea of reusing existing software parts has evolved into *Component-Based Software Engineering* (CBSE), where composing systems from off-the-shelf and custom-built components is preferred over traditional programming [5]. However, CBSE requires standardization of components' interfaces – a difficult goal to achieve for the vast amount of existing solutions.

Another way to tackle robotics software complexity could be using design patterns as in general computer science. "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma *et al.* [6] is a great digest of object-oriented solutions that many programmers have reused. Later work by Fowler [7] provides more design patterns for applications in enterprise domain. The patterns from these books are, however, developed for general purpose programming. Therefore, although many of those object-oriented patterns can be applied to robotics, they do not reflect any structural robotics specifics such as, for example, Sense-Plan-Act control loop, or a behavior-based architecture, or in general, having a hardware part that interacts with the real world.

On the robotics side, there has been little work on introducing design patterns to robotics. The only notable work is by Graves *et al.* [8] on design patterns for behavior-based robotics. Its applicability is, however, limited to the class of man-machine interaction. It is important to note that

behavior-based robotic architectures, such as a subsumption architecture by Brooks [9], are essentially concurrent, and thus could benefit from concurrency patterns.

Concurrency patterns have been developed for general programming, and implementations exist in different object-oriented programming languages [10], [11], [12], [13]. Leijen *et al.* [14] shows results of developing a task parallel library using concurrency patterns. Russel *et al.* [15] identifies a collection of control-flow patterns, many of which have concurrent nature and could be helpful in robotic coordination. The idea of coupling parallel tasks with robotics field has also been proposed [16]. In the paper, the authors share their experience of teaching three concurrency patterns *Pipeline*, *Delta*, and *Black Hole* and applying the knowledge on the small robotics setup. However, the patterns considered in the paper are quite simple and more applicable to data processing than to robotic coordination. Secondly, the paper emphasizes the educational aspect, where robotics is one of the encouraging factors to learn concurrency, not vice versa. Therefore, it is insufficient to serve as a reference.

Mentioned attempts of combining software design patterns with the field of robotics demonstrate a need for establishing a set of pattern solutions in robotics programming. To our knowledge, there is no widely accepted reference for design patterns in robotics, and certainly not for concurrency patterns in robotic coordination. This work demonstrates how concurrency patterns could benefit robotics programming, especially for tasks of robotic coordination. We also speculate about how modern programming robotics frameworks can support proposed solutions.

III. CONCURRENCY IN ROBOTICS

Many robots are inherently concurrent; they have many sensors and actuators functioning in parallel, often executing several tasks simultaneously. In such situations, robotics applications could benefit from the use of concurrency. Unfortunately, concurrent programming requires additional expertise and is usually more difficult than sequential programming due to non-determinism and specific pitfalls such as data races or deadlocks. This does not mean concurrency should be ignored by roboticists, especially by non-experts in programming. Concurrency brings easier implementation of simultaneous execution of behaviors and more natural reasoning of the system, thus we can empower roboticists if modern approaches, tools and frameworks together support concurrency and we utilize accumulated knowledge such as *design patterns*.

Most robotics applications are complex systems where parts can operate concurrently on many different levels. Here we distinguish between *data-centric* applications and *task-centric* applications of concurrency. Examples of *data-centric* applications include pipeline and filtering, where data and its processing play the main role. In contrast, *task-centric* applications are built with an idea of multiple cross-dependent tasks taking place at the same time. A representative example of a *task-centric* application in robotics is the use of behavior-based architectures such as *subsumption architecture* by Brooks [9].

Another widely used approach to robotic coordination is the *event-driven* programming [17] where external actions (events) determine the execution flow. Event can represent a sensor value or a message from the current or other threads. Every time an event occurs, it triggers associated callback functions or event handlers. Concurrency and event-driven approaches are not mutually exclusive – they can complement each other, such as event loop can be implemented with multiple threads to handle callbacks concurrently.

In general, concurrency could benefit many components of robotic applications. In this paper, we focus on *task-centric* concurrency and a selection of concurrency patterns as described in Section IV.

Frameworks

Many robotics programming frameworks were proposed to simplify creation of robotics applications. We chose ROS [4], Urbi [18] and Roboscoop [19] to discover their potential for implementing considered patterns. All three frameworks have concurrency support in their underlying languages – *C++* in ROS, *urbiscript* in Urbi and *SCOOP* in Roboscoop.

ROS is a popular robotics middleware, which has implementations in several programming languages including *C++*. ROS represents robotics system as a set of executable *nodes* and uses publish/subscribe and service-based message-passing model to communicate between them. ROS is essentially event-driven; it uses its messages as events to execute callback functions. Although ROS is not positioning itself as a concurrency framework, each ROS *node* can use *C++* features for concurrency. ROS also provides *MultiThreadedSpinner* and *AsyncSpinner* classes to handle callbacks concurrently. In addition to communication facilities, ROS provides a wide variety of robotics libraries or *stacks*.

Urbi is a software platform for robotics, which in addition to the *C++* support, introduces *urbiscript* – a parallel event-based object-oriented script language. *urbiscript* is used as an orchestration language – it provides tools and syntactic extensions for coordination and concurrency support. Among features of *urbiscript* are a support for events including conditional events and a tagging mechanism that allows to tag sections of code and manipulate them with corresponding *Tag* objects. Urbi is interoperable with ROS.

Roboscoop is a concurrency robotics framework built on *SCOOP* – Simple Concurrent Object-Oriented Programming[20]. *SCOOP* model provides simple and safe concurrency and eliminates data races by construction. Along with concurrency, *SCOOP* supports event-driven approach. Roboscoop is a robotics library written in the Eiffel programming language. Through *external calls*, Roboscoop facilitates reuse of existing *C++* code such as robotics libraries. In addition, Roboscoop provides interface with ROS, thus any ROS applications can also be integrated into Roboscoop.

IV. PATTERNS

A design pattern is a description of a known design problem brought together with good practices for resolving

it. Software design patterns, however, do not provide any universal reusable components; they only explain a structure and the way the solution should be implemented. This differs design patterns from the CBSE in addressing an integration problem. An integration problem comes from complexity of using multiple different libraries and solutions in a single robotics system. Unlike components, patterns do not provide ready-to-use building blocks – the solution often needs to be programmed for each individual case over again. Despite this, design patterns help to accumulate common knowledge for the field and patterns' names become a terminology for programmers. Patterns ease identification of common design problems and the application of a solution to each problem. Also, they do not have to be standardized and thus are more flexible than components, since design patterns are not bound to a particular interface, framework or a programming language.

This section presents several concurrency design patterns, namely *Future*, *Periodic timer*, *Invoke later*, *Active object*, *Cooperative cancellation*, *Guarded suspension*. The criteria for selection included 1) concurrent nature of patterns; 2) applicability to robotic coordination; 3) evidence of use in existing robotics frameworks. We do not claim that the provided list of patterns is exhaustive with respect to these criteria.

We demonstrate each of the patterns according to the following structure:

1) intent;
2) applicability to robotics coordination;
3) possible applications in robotics;
4) use in robotics frameworks;
5) related patterns;
6) references to implementation.

The structure above is a subset of the template for describing patterns provided by Gamma et al. [6]. The additional value of the above list is that it takes into account robotics aspects including examples, possible applications and applicability to commonly used robotics architectures. We provide sample code for each pattern and references to implementation. In addition, to emphasize practical usefulness, we demonstrate how they are exploited in existing robotics frameworks, namely, ROS, Urbi and Roboscoop.

A. Future

1) Intent: Start a task asynchronously in the background and provide a "hook" for retrieving the result later.

2) Applicability: Use the *Future* pattern when a time consuming computation may run in parallel, while a robot still needs to execute lower-level control.

3) Possible applications: Object recognition tasks can run in the background of an exploration process during the search and rescue scenario. In general, the pattern can be used in tiered architectures for running deliberation tasks.

4) In frameworks: Roboscoop uses *futures* implicitly through the *wait by necessity* synchronization mechanism of *SCOOP*. *C++11* standard has a language support for futures [21, p. 1191] with the *std::future<T>* templated class. However, we could not find any use of it in current ROS implementation nor in Urbi. A simplified *C++* implementation of the aforementioned object recognition example would look as follows:

```
int recognizedObjectId() { ...
} // Object recognition.

std::future<int> f = std::async(recognizedObjectId);
// Continue with other computations and retrieve the result later:
int id = f.get();
```

5) Related patterns: Future can be used in the implementation of *Active object*.

6) References: [11, p. 332] [14].

B. Periodic timer

1) Intent: Repeat a task or an iteration of a task regularly in a predefined interval of time.

2) Applicability: Use the *Periodic timer* pattern when a task needs to be applied repeatedly in parallel with the currently running task.

3) Possible applications: Periodic timer can be used for programming a robot which checks the battery level every minute, or for another robot that uses two sensors with sufficiently different update rates. In general, several independent control loops with different predefined frequencies can be implemented with this pattern.

4) In frameworks: In ROS, timer can be implemented either with the *ros::Timer* class, or with the *ros::Duration* class by manually calling a *sleep()* function after each iteration. However, for concurrency support, programmers need to use the multi-threaded version of ROS spinner, as it is shown in the following code:

```
int getBatteryLevel(){ ...
} // Request to the hardware.

void checkBatteryLevel(const ros::TimerEvent& e) {
  if (getBatteryLevel() < 10) playAlarm();
}
ros::NodeHandle n;
ros::Timer t; // Create a timer and assign a callback function.
t = n.createTimer(ros::Duration(60), checkBatteryLevel, false);
ros::AsyncSpinner spinner(4); // Create a spinner using 4 threads.
spinner.start(); // Start listening for callbacks.
```

Roboscoop provides the *Timer* class to run separated tasks repeatedly. Urbi facilitates it with the *every* syntactic construct, which can be assigned to a repeatable task. *every* receives a duration between task executions as a parameter.

5) Related patterns: Invoke later pattern can be implemented through the current pattern.

6) References: [11, p. 298].

C. Invoke later

1) Intent: Run a task after a certain delay.

2) Applicability: Use *Invoke later* when execution of an additional task taking place in parallel needs to be delayed.

3) Possible applications: Invoke later may be useful in a scenario when a robot returns to the base if nothing new was explored in last 5 minutes. The pattern can be used for simulating initialization time in robotic systems with no feedback. Another example is switching to a stand-by mode or calling emergency when the robot got stuck for some time.

4) In frameworks: In ROS, the *ros::Timer* class can be used with an additional parameter for stopping the timer after the first execution. In Urbi, the only way we found is to start a task in a parallel section of code with a *sleep()* function right before it. Roboscoop provides the *Invoker* class.

5) Related patterns: Invoke later can be implemented via a single-iteration *Periodic timer*.

6) References: [11, p. 296].

D. Active object

1) Intent: Associate an object with its own thread and decouple method invocation from method execution in order to simplify access to the object from other threads. Provide the client interface that only forwards calls to the active object, which resides on a separate thread and is responsible for scheduling and running the execution.

2) Applicability: Use the *Active object* pattern when robot's parts such as actuators and sensors are represented as active subsystems.

3) Possible applications: A robotic leg that is accessible during the execution of a movement control, to provide its coordinates can be implemented as an *Active object*. In general, the pattern can be used for representing actuators, sensors and subsystems of the robot as *Active objects*.

4) In frameworks: Roboscoop uses one of the main concurrency mechanisms of *SCOOP* called *separate call*, which corresponds to the *Active object* pattern. In *SCOOP*, for its lifetime, each object is associated with its handler - an abstract processor usually implemented as a thread. ROS and Urbi do not provide off-the-shelf support for *Active object*.

The following code demonstrates on the robotic leg example how the pattern can be implemented in *C++*. The *LegProxy* class is the only interface available to the client. It contains the methods available on the robotic leg but does not implement them. Instead, the proxy class creates method requests and enqueues them using *Scheduler*. To the client, proxy provides the *future* objects for retrieving results later.

```
class LegProxy{
private:
  Scheduler scheduler; // Object that manages the methods queue.
  Servant servant;     // Object where methods are implemented.

public:
  // getX() method can be implemented similarly.
  shared_ptr<Future<bool>> move(int x, int y){
    shared_ptr<Future<bool>> f(new Future<bool>());
    shared_ptr<MethodRequest> mr(new MoveRequest(x, y, f, servant));
    scheduler.enqueue(mr);
    return f;
  }
};
```

Two auxiliary classes *Mutex* and *Future* are used to support the pattern's mechanisms. *MethodRequest* and its descendant classes *MoveRequest* and *XRequest* are used to encapsulate the calls to *Servant*, who is the real executor of the *LegProxy*'s methods. The *guard()* method in these classes is used for enforcing the synchronization policies, which can be different for each method.

```
class MethodRequest { // Base class for all method requests.
public:
  virtual bool guard() = 0;
  virtual void execute() = 0;
};

// XRequest class can be implemented similarly.
class MoveRequest : public MethodRequest {
private:
  shared_ptr<Future<bool>> f_ptr;
  Servant &servant;
  static Mutex mtx;
  int xVal; int yVal;
public:
  MoveRequest(int x, int y, shared_ptr<Future<bool>> res,
      Servant& sr) : servant(sr), f_ptr(res) { xVal = x; yVal = y;}
  bool guard() { return !mtx.isLocked();} // One request at the time.
  void execute() {
    mtx.lock();
    f_ptr->set(servant.move(xVal, yVal));
    mtx.unlock();
  }};
```

Servant is the class where all the methods are implemented.

```
class Servant {
public:
  int getX() { ...
  } // Request to the hardware.
  bool move(int x, int y) { ...
  } // Move the leg. Return true if success.
};
```

ActivationQueue is a container for method requests. *ActivationQueue* is managed by *Scheduler*. The *Scheduler* runs its own thread. It keeps track of the available method requests in the *ActivationQueue* and executes applicable requests asynchronously in separate threads. The results of execution can be accessed through the *future* objects in the *LegProxy* class. For the current example of a robotic leg, the *Active object* pattern can be used as follows:

```
LegProxy leg;
shared_ptr<Future<bool>> fmove = leg.move(100, 200);
shared_ptr<Future<int>> fx = leg.getX();

while (!(fx->isAvalaible())){}
cout << "x = " << fx->get() << endl;
```

5) Related patterns: Inside the implementation of *Active object*, the *Future* pattern is often used to return the result of the method, if any. *Periodic timer* can be implemented as an *Active object* which repeatedly invokes the same method.

6) References: [10, p. 369], [22].

E. Cooperative cancellation

1) Intent: Ensure that the task remains in a valid state after cancellation. Send the cancellation request via a shared cancellation token object.

2) Applicability: Use the *Cooperative cancellation* pattern when there is a need to safely cancel possibly concurrently running tasks and leave the system in a correct state.

3) Possible applications: Cooperative cancellation can be used for emergency stop for a robot. Another possible application is switching between types of low-level control depending on higher level plan. In general, in tiered architectures, the pattern can be useful for stopping lower level tasks. Implementing cancellable control loops can also be done with the help of the *Cooperative cancellation* pattern.

4) In frameworks: ROS contains the *actionlib* stack [23], which implements standardized interface for preemptable tasks. Also, for canceling and shutting down the whole ROS application, two methods can be used: *ros::isShuttingDown()* and *ros::ok()*. Roboscoop offers the *StopSignaler* class to be used as a cancellation token for *Cooperative cancellation*. Urbi handles the cancellation issue differently – it uses a special *leave* event that occurs each time a task represented by a section of code is left. *leave* is also fired when execution of a task is canceled, so programmers can use it to trigger the clean-up. In general, a cancelable control loop implementation looks in *C++* as follows:

```
// Need to specify next two functions for each particular task.
void iterationOfControl () { ... }
void stopSafely () { ... }

void startControl (CancellationToken token) { // Control loop.
  while (!token.isStopRequested)
    iterationOfControl ();
  stopSafely ();
}
CancellationToken cancel = new CancellationToken();
startControl (cancel); // Starting the control loop.
cancel.requestStop (); // Triggering stop later by another thread.
```

5) Related patterns: None.

6) References: [24].

F. Guarded suspension

Robotic tasks that jointly perform desired robotic behaviors can often be expressed as multiple interleaving actions applied "at the right time", where "right time" means certain conditions to be met before an action starts, i.e., the action's *precondition*. In sequential case, actions are executed step by step, and the *preconditions* correspond to having all previous steps done. When actions can be executed concurrently, it is harder to follow the execution flow; however, using *preconditions* in a concurrent context, we can define for each action an important synchronization point and therefore, ease the coordination of tasks while still receiving the benefits of parallel execution.

Concurrency, while beneficial, is not without a caveat. When actions are applied concurrently to robotic parts such as sensors or actuators, they may overwrite the effect of each other, resulting in a deadlock. To avoid data races and to prevent this type of deadlocks, exclusive access to shared resources must be guaranteed. The *Guarded suspension* pattern is used for this purpose.

1) Intent: Execute an action on a shared resource only when the lock is acquired on the resource and the precondition for the action is satisfied.

2) Applicability: Use the *Guarded suspension* pattern when a desired robotic behavior can be expressed with a set of actions and their preconditions.

3) Possible applications: An application of a humanoid robot that follows with its eyes a ball object only when the ball is visible can be implemented with the *Guarded suspension* pattern. In general, the pattern can be used for robotic coordination that involves multiple sensors and actuators repeatedly changing their states, such as generating gaits or movement patterns.

4) In frameworks: Roboscoop uses the underlying *SCOOP* model's synchronization mechanism called *wait conditions*. This mechanism guarantees exclusive access to the arguments of the method call, and the execution of the method's body starts only when the corresponding *precondition* is satisfied. Urbi allows conditional events with the *at* syntactic construct, but for acquiring a lock on the object, one should manually use *Mutex* objects. ROS has neither automatic precondition checks nor automatic locks on shared resources, which means the whole pattern must be implemented manually. The following *C++* code implements the pattern for the humanoid robot example above. We assume a moving ball that can provide (x,y) coordinates, which is visible when its coordinates are within the range of -100 and 100. We assume functions *followBall* and *updateBall* are executed by different threads.

```
class Eye {
  private:
    bool isBallVisible ;
    std::mutex m; std:: condition_variable cv;
  public:
    void followBall (Ball ball) {
      std:: unique_lock<std::mutex> lk(m, std:: try_to_lock );
      std:: lock(lk); // Automatically released when out of scope.
      cv.wait(lk, []{return isBallVisible ;});
      // Code for moving the eye here.
    }
    void updateBall (Ball ball) {
      std:: unique_lock<std::mutex> lk(m, std:: try_to_lock );
      std:: lock(lk); // Automatically released when out of scope.
      isBallVisible = abs(ball.x) < 100 && abs(ball.y) < 100;
      cv. notify_all ();
    }};
```

5) Related patterns: None.

6) References: [11, p. 183], [12, p. 445], [20, p. 68].

V. DISCUSSION

The previous section provided a number of concurrency patterns that can solve commonly occurring robotics problems. In this section, we summarize the value of those patterns for robotic coordination and discuss the aspects to which robotics programmers should pay attention when implementing these concurrency patterns.

Future, *Periodic timer*, and *Invoke later* are useful patterns for coordination of time-related tasks where proactive, periodic, or delayed execution takes place. The fact that all three frameworks – ROS, Urbi, and Roboscoop – support the functionality attests to its importance in robotics. On the negative side, these patterns could make the execution flow obscure and require "concurrent" thinking from programmers.

The main purpose of using the *Active object* pattern is to increase understandability of the robotic system. Treating robot's parts as active objects and putting the burden of synchronization under the hood of the pattern provides a clearer client interface for each object. This simplifies concurrent thinking of robot's parts, bringing coordination closer to the sequential case. On the other side, using a separated thread for each object may result in squandering computational resources when many objects are involved. In general, the pattern is tedious to implement from scratch,

but currently, only Roboscoop provides native support for the mechanism. The pattern would be more widely used when other frameworks also provide built-in support.

Cancellation of concurrently running robotic behaviors is another important and popular coordination task. It is no surprise that all three frameworks support it in one form or another. Applying *Cooperative cancellation* provides a safe solution that can be applied at different levels, from a primitive control to a deliberation level of applications. However, this approach assumes designing each task with the cancellation scenario in mind, which may cause difficulty.

Coordination using *Guarded suspension* enables robotics programmers to express desired behaviors in terms of *preconditions*. Of the three frameworks, only Roboscoop supports this mechanism natively, thanks to its base language *SCOOP*. An advantage of *Guarded suspension* is a natural translation of robotic behaviors into source code [25]. However, naive implementation of this pattern may result in wasting computational resources since corresponding threads are not doing any active work.

There are several benefits to using concurrency patterns for robotic coordination. Reasonably-chosen and properly-implemented design patterns ensure flexibility and extendability of the system, thus improving its software quality. Concurrency patterns can also help programmers avoid common concurrency pitfalls. In some cases, concurrency eases implementation of tasks that are otherwise hard to implement, e.g., simultaneous execution of behaviors. This is particularly useful for robotics because it is often easier to reason about a robotic system as a set of concurrently running parts and their tasks, especially when it comes to applications on coordination.

Despite the benefits, patterns should be used with care. Although patterns can improve software design, they do not necessarily result in better design. Improvements strongly depend on suitability of the patterns for each particular application and context. Applying patterns usually increases complexity of the application because patterns often introduce additional components to the system, which can result in over-engineering and additional debugging costs, since patterns may significantly change the execution flow. Also, using concurrency brings non-determinism and additional concurrency-related issues, such as data races or deadlocks. Consequently, concurrent applications are usually harder to write, test, and debug than sequential applications. In general, because of additional design components, patterns often introduce an overhead to application performance. In case of concurrency patterns for *task-centric* applications, performance is sacrificed for the sake of easier coordination. This means that the gain in performance depends on the synchronization costs, and for complex coordination tasks, the gain can be sufficiently lower than in *data-centric* applications.

VI. CONCLUSION

This paper presented six different concurrency patterns and explained how the patterns can be used in robotic coordination. Concurrency can greatly simplify robotic coordination, and concurrency patterns can ease the implementation significantly, especially when supported by programming frameworks. However, there has not yet been any established set of reusable concurrency design patterns for robotics. We propose this paper as a starting point for bringing design patterns to robotics and thus minimizing the gap between robotics and software engineering.

REFERENCES

[1] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *IEEE Int. Conf. on Robotics and Automation*. IEEE, 2011, pp. 1–4.

[2] M. Rickert, "Efficient motion planning for intuitive task execution in modular manipulation systems," Dissertation, Technische Universität München, Munich, Germany, 2011.

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

[4] 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 *2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2009.

[5] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Towards component-based robotics," in *2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2005, pp. 163–168.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[8] A. R. Graves and C. Czarnecki, "Design patterns for behavior-based robotics," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Trans. on*, vol. 30, no. 1, pp. 36–41, 2000.

[9] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE J. of*, vol. 2, no. 1, pp. 14–23, 1986.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[11] D. Lea, *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[12] P. Kuchana, *Software architecture design patterns in Java*. CRC Press, 2004.

[13] R. Schmocker and A. Kolesnichenko, *Concurrency Patterns in SCOOP*. ETH-Zürich, 2014.

[14] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Acm Sigplan Notices*, vol. 44, no. 10. ACM, 2009, pp. 227–242.

[15] N. Russell, A. H. Ter Hofstede, and N. Mulyar, "Workflow controlflow patterns: A revised view," 2006.

[16] M. C. Jadud, J. Simpson, and C. L. Jacobsen, "Patterns for programming in parallel, pedagogically," in *ACM SIGCSE Bulletin*, vol. 40, no. 1. ACM, 2008, pp. 231–235.

[17] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 186–189.

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

[19] A. Rusakov, J. Shin, and B. Meyer, "Simple concurrency for robotics with the Roboscoop framework," in *2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2014.

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

[21] "Programming languages – C++." [Online]. Available: https://isocpp.org/files/papers/N3690.pdf

[22] R. G. Lavender and D. C. Schmidt, "Active object–an object behavioral pattern for concurrent programming," 1995.

[23] [Online]. Available: http://wiki.ros.org/actionlib/

[24] A. Kolesnichenko, S. Nanz, and B. Meyer, "How to cancel a task," in *Proceedings of the 2013 Int. Conf. on Multicore Software Engineering, Performance, and Tools (MUSEPAT)*, 2013.

[25] G. Ramanathan, B. Morandi, S. West, S. Nanz, and B. Meyer, "Deriving concurrent control software from behavioral specifications," in *2010 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2010.